



Collective Intelligence

Course Notes for IPM-22fmiCOLLIEG

Tamás Takács, Zoltán Barta
PhD students, Department of Artificial Intelligence
Faculty of Informatics

2025

Preface

These notes accompany the Master's course *Collective Intelligence* (IPM-22fmiCOLLIEG), offered by the Department of Artificial Intelligence, Faculty of Informatics, Eötvös Loránd University (ELTE). The material was prepared by Tamás Takács and Zoltán Barta, PhD students, under the supervision of Dr. László Gulyás. The course offers a comprehensive introduction to collective intelligence, focusing on agent-based modeling, game theory, and multi-agent reinforcement learning (MARL). The accompanying exercises were developed for interactive use in Google Colab.

Course Structure

The course is divided into three main thematic blocks:

- **Agent-Based Modeling:** Fundamentals and simulation of collective behavior in agent-based systems.
- **Game Theory:** Formal analysis and modeling of agent interactions as games.
- **Multi-Agent Reinforcement Learning:** Training and coordination of intelligent agents using RL principles.

Assignments

Two assignments are required for course completion:

- **Assignment 1:** Individual project in agent-based modeling.
- **Assignment 2:** Group project on a selected MARL topic. Example topics include: Cooperative Mingle, Formation, Mechanism Design, Particle Swarm Optimization, Pathfinding, Patrolling, Search and Rescue, Sorting, and TurtleBot3s.



Practice Notebooks

The primary course content is delivered through interactive Google Colab notebooks. The table below summarizes the practices and provides access links.

Practice	Title / Access Link
1	Introduction to Collective Intelligence
2	An Introduction to NetLogo
3	NetLogo Simulations
4	NetLogo Model Design
5	Games: Models of Multi-Agent Interaction
6	Solution Concepts for Games
7	Introduction to Single-Agent RL
8	Introduction to Multi-Agent RL
9	Introduction to TorchRL
10	Centralized Training with Decentralized Execution
11	Heterogeneous Teams in MARL (MADDPG)
12	Communication in MARL

Note: Practice 1 is available exclusively via the ELTE SharePoint system.

Included Materials

This document contains the preface, all practice materials (2–12), both assignments, and, as an appendix, the first course presentation in PDF format.

✓ 2. Practice - An Introduction to NetLogo

 Tamás Takács, PhD student, Department of Artificial Intelligence

 90 min read

 January 22, 2025

 Collective Intelligence



ELTE | IK
INFORMATIKAI KAR



DEPARTMENT
OF ARTIFICIAL
INTELLIGENCE



This practice introduces NetLogo and the basics of agent-based modeling. Students will get familiar with key concepts: what an agent is, the types of agents in NetLogo, and how simulations run. The aim is to show why NetLogo is a good tool for learning ABMs and help students identify the main components they will need. After this practice, students should be ready to start building more complex models.

✓ Table of Contents

- **2.1 Why NetLogo?**

- 2.1.1 Close Competitors
- 2.1.2 Code Comparison
- 2.1.3 New Competitors

- **2.2 Building Blocks**

- 2.2.1 The Environment
- 2.2.2 The Agents
- 2.2.3 The Observer
- 2.2.4 Starting Up NetLogo
- 2.2.5 Appetizer Commands
- 2.2.6 The Color Property
- 2.2.7 Shapes
- 2.2.8 The Breed Property
- 2.2.9 The Pen Mode Property
- 2.2.10 Moving the Agents
- 2.2.11 Patches

- **2.3 Programming in NetLogo**

- 2.3.1 Commands
- 2.3.2 Interface Elements
- 2.3.3 Reporters
- 2.3.4 Styling
- 2.3.5 Variables

- 2.3.6 Agentsets
- 2.3.7 Conditionals
- 2.3.7 Loops
- 2.3.8 Lists
- 2.3.9 Program Structure
- 2.3.10 Higher-Order Procedure
- 2.3.11 Breeds

NetLogo is a **programmable modeling environment** for simulating natural and **social phenomena**, based on **Logo** by Seymour Papert. It is designed to model **complex system** development over time.

1. **Complex System** a system made up of many interacting components or agents, where the interactions give rise to emergent behaviors that cannot be easily predicted from the behavior of individual components (congested road network).
2. **Social Phenomena**: behaviors, patterns, or events that arise within societies due to the interactions and relationships among individuals or groups (voting patterns, Romania 2024).
3. **Logo**: high-level, interpreted, dynamically-typed programming language designed for educational purposes. Comes with functional paradigms, symbolic processing and turtle graphics.
4. **Programmable Modeling Environment**: a software tool that allows users to create, simulate, and analyze computational models of complex systems. It provides a framework for defining agents, their behaviors, and interactions within an environment (NetLogo).

▼ **2.1 Why NetLogo?**

Are there any other programmable modeling environments out there? **Yes, there are.**

Many competing modeling environments use programming languages inspired by or similar to Logo. Most of these platforms integrate another high-level programming language, such as Java, to enhance the user experience. However, much of the Java code in these systems has become outdated.

2.1.1 Close Competitors:

Feature	GAMA	
License	Open-source (GPL v3.0)	Open-sou
Programming Language	GAML (Gama Modeling Language) for simulations; Java for extensions	NetLogo l
Operating Systems	Cross-platform: Windows, Linux, macOS	Cross-pla
Primary Domain	Spatially explicit agent-based simulations	Social an
User Support	Tutorials, manual, FAQ, forums, documentation, selected publications, examples	Documen
GIS Capabilities	Advanced GIS support, allowing integration and manipulation of spatial data	Basic GIS
3D Capabilities	Supports 3D simulations and visualizations	Basic 3D
Learning Curve	Moderate; requires understanding of GAML and modeling concepts	Beginner-
Performance	Suitable for large-scale simulations; performance depends on model complexity	Best suite
Debugging	Reports syntactic and semantic errors and gives semantic warnings that indicate “flaws in the logic of the model”	Limited a
Simulation Speed	Slowest (tested on Game of Life)	Fastest (t
Multi-Threading	Yes	No
Latest Version	2.11 (as of January 20, 2025)	6.4.0 (as

Reference: Raab, R., Lenger, K., Stickler, D., Granigg, W., & Lichtenegger, K. (2022). An Initial Comparison of Selected Agent-Based Simulation Tools in the Context of Industrial Health and Safety Management. Proceedings of the 2022 8th International Conference on Computer Technology Applications, 106-112. Presented at the Vienna, Austria. doi:10.1145/3543712.3543745

2.1.2 Code Comparison:

```
to go
  ask patches
  [ set live-neighbors count neighbors with [ living? = true ] ]
  ask patches
  [ if living? = true and live-neighbors < 2 [ cell-death ]
    if living? = true and live-neighbors > 3 [ cell-death ]
    if living? = false and live-neighbors = 3 [ cell-birth ] ]
  set living-percentage (count patches with [ living? = true ])
  / (count patches)
  tick
end
```

Figure 1: Main Procedure in NetLogo.

```
def go(){
  ask(patches()){
    living_neighbors = count(neighbors(1,1).with(living == true))
  }
  ask(patches()){
    if(living == true & living_neighbors < 2) {cell_death()}
    if(living == true & living_neighbors > 3) {cell_death()}
    if(living == false & living_neighbors == 3) {cell_birth()}
  }
  living_percentage = (count(patches().with
    { living == true })/count(patches()))
  tick()
}
```

Figure 2: Main Procedure in Repast Symphony ReLogo.

```
reflex go {
  ask life_cell {
    live_neighbors <- self neighbors_at 1 count each.living;
  }
  ask life_cell {
    if living = true and live_neighbors < 2 {do cell_death;}
    if living = true and live_neighbors > 3 {do cell_death;}
    if living = false and live_neighbors = 3 {do cell_birth;}
  }
  living_percentage <- life_cell count each.living / length(life_cell);
}
```

Figure 3: Main Procedure in GAMA.

2.1.3 New Competitors:

New competitor modeling environments aim to leverage multi-threading and more efficient, faster programming languages as their backbone to outperform traditional Java-based platforms. The market remains highly competitive, with ongoing efforts to develop the most comprehensive and versatile programmable modeling software.

Table 1. A comparison of four ABM frameworks covering objective categories focusing on ease of use, available functionality and performance. Colours represent implementation quality. Red: poor/none, Yellow: basic, Green: good, Blue: clear class leader. Further details corresponding to the superscript numbers are given in the main text.

	Agents.jl 4.2	Mesa 0.8	NetLogo 6.2	Mason 20.0
	Objective property comparisons.			
Core	Core design decisions and aspects that cannot be changed or implemented by users			
Continuous Space	Yes	Yes	Yes	Yes
Graph Space	Yes, and mutable	Only undirectional	Link Agents (not a Space)	Networks (not a Space)
Grid Space	Yes	Yes (+Hexagonal)	Yes	Yes (+Hexagonal, Triangular)
OpenStreetMap Space	Yes	No	No	No
Dimensionality	Any ¹	2D	2D & 3D (separate applications)	2D & 3D (complicated install for 3D)
License permissiveness	MIT	Apache v2.0	GPL v2	Academic Free License
Mixed-agent models	Yes	Yes	Yes	Yes
Simulation termination	After 'n' steps or user-provided boolean condition of model state	Explicitly written user loop	Manually by pressing a button on the interface, stop command in code	When Schedule is empty, or user provided custom finish function
Parameter types	Anything	Anything	Float64, Lists Hashtables and Assoc. Arrays in the Table extension	Anything
Modeling and Analysis in the same language	Yes, Julia v1.5+	Yes, Python v3+	No	Yes, Java but designed to work within the console or GUI of the applet
Maximum memory capacity	Hardware limits	Hardware limits	1 GB Manually expanded by increasing JVM heap	1 GB Manually expanded by increasing JVM heap
Distributed computing²	Yes	No. BatchRunnerMP is only multithreaded	No. BehaviorSpace is only multithreaded	Yes
Interop with external libraries	Yes, also couples to anything in Python / R / C / C++ seamlessly.	Yes, modular design.	Partial, via the Extensions API. JVM languages (Scala, Clojure)	Partial. Extensions in the 'contrib' directory. No simple user API
Language ecosystem integration	By Design. Examples: black box optimization, differential equations	Any of Python's analytical tools can be used	Complex. Must create plugins or use Control API	Warned against (e.g. Random), provides custom types in place of Java primitives
Browser-based online ABM execution	No	No	Yes (NetLogo Web)	No
Data collection	Any chosen parameter / property or function mapped over them. Aggregating and filtered aggregate functions	Any chosen parameter / property. Aggregating functions. No conditional options	boolean, number, string and lists of these types.	Inspectors track & chart any parameter / property. Entire model saved to disk via checkpointing, no custom export

Reference: Datseris, G., Vahdati, A. R., & DuBois, T. C. (2022). Agents.jl: a performant and feature-full agent-based modeling software of minimal code complexity. *SIMULATION*, 100(10), 1019-1031.

doi:10.1177/00375497211068820

Other notable advantages of NetLogo:

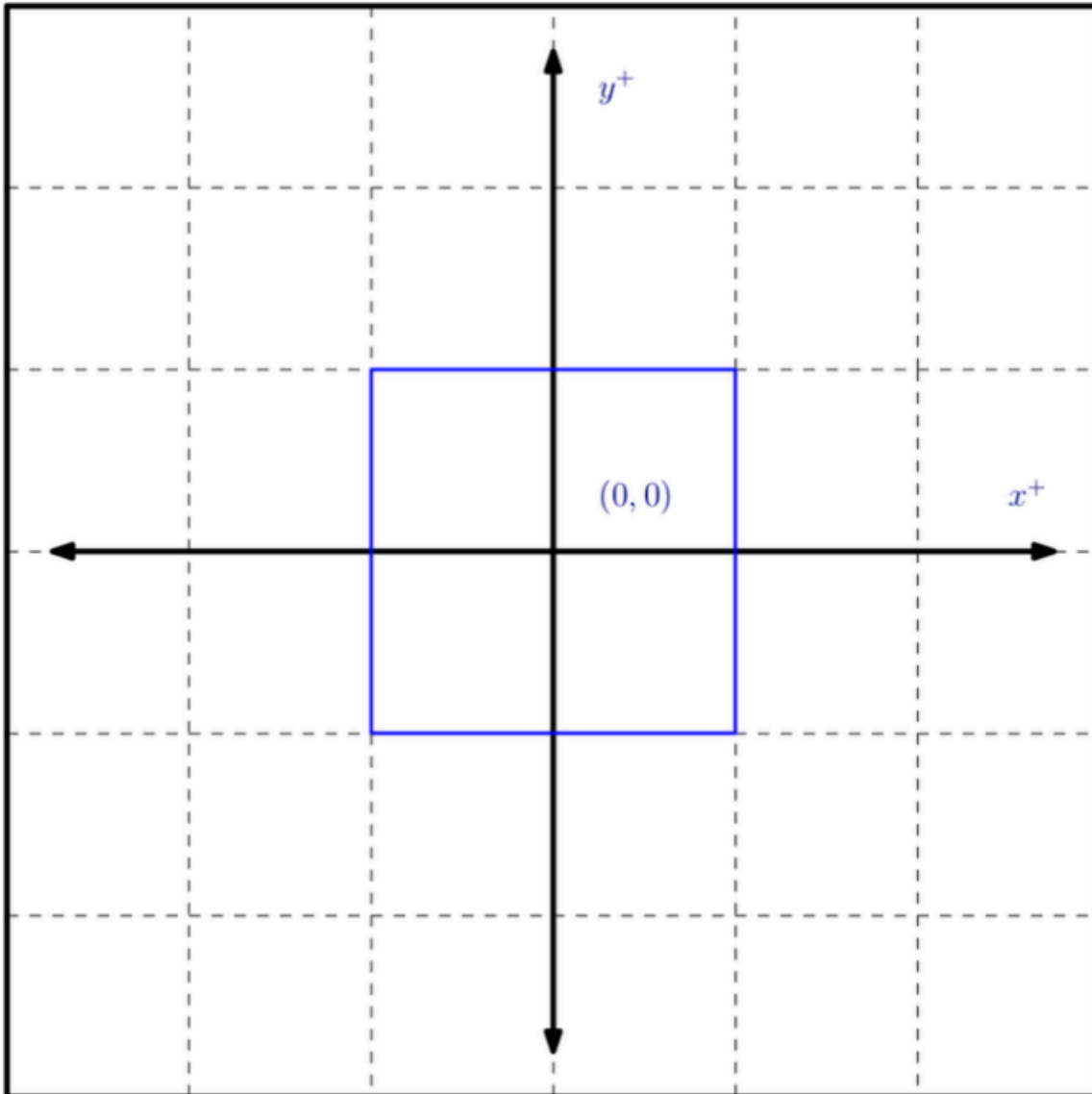
- Extensive documentation (literally contains everything a good documentation needs)
- Huge collections of pre-written simulations on Biology, Medicine, Physics, Chemistry and more

- Very easy to get into

✓ 2.2 Building Blocks

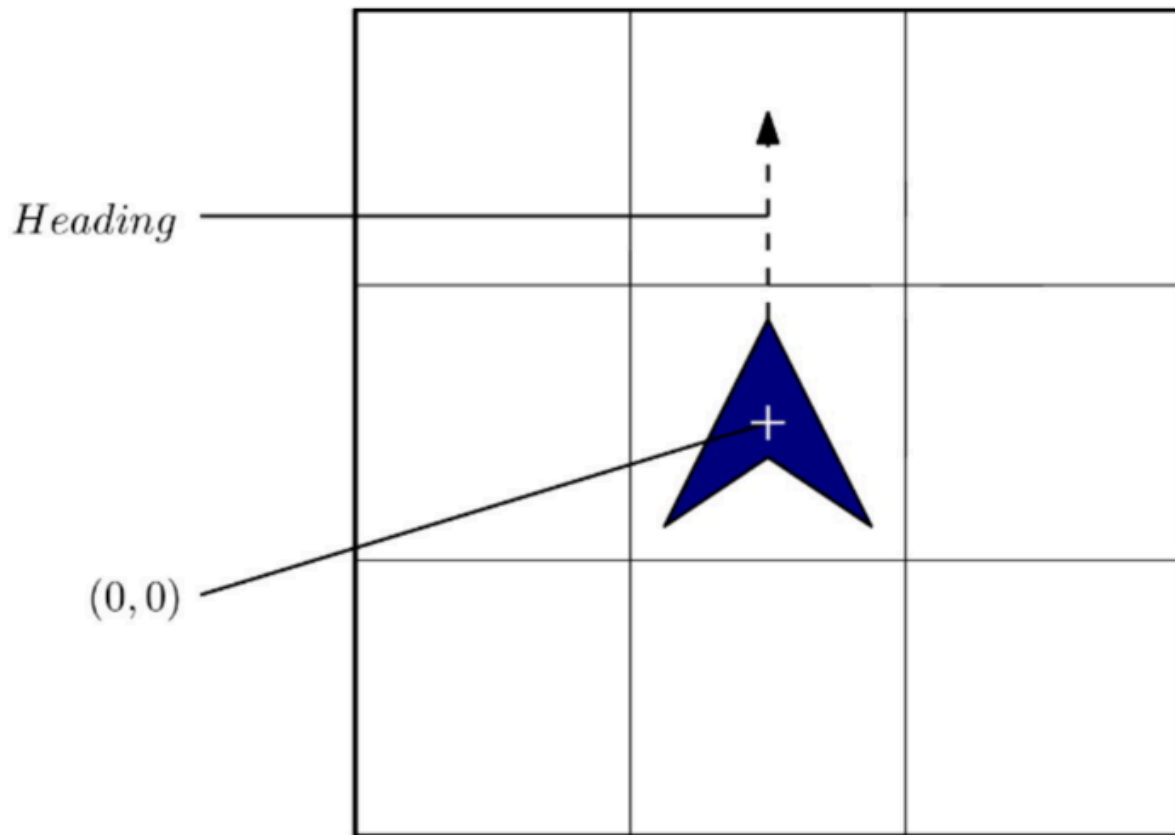
2.2.1 The Environment

The whole world is a **discrete grid**. Each basic region is called a **patch**.



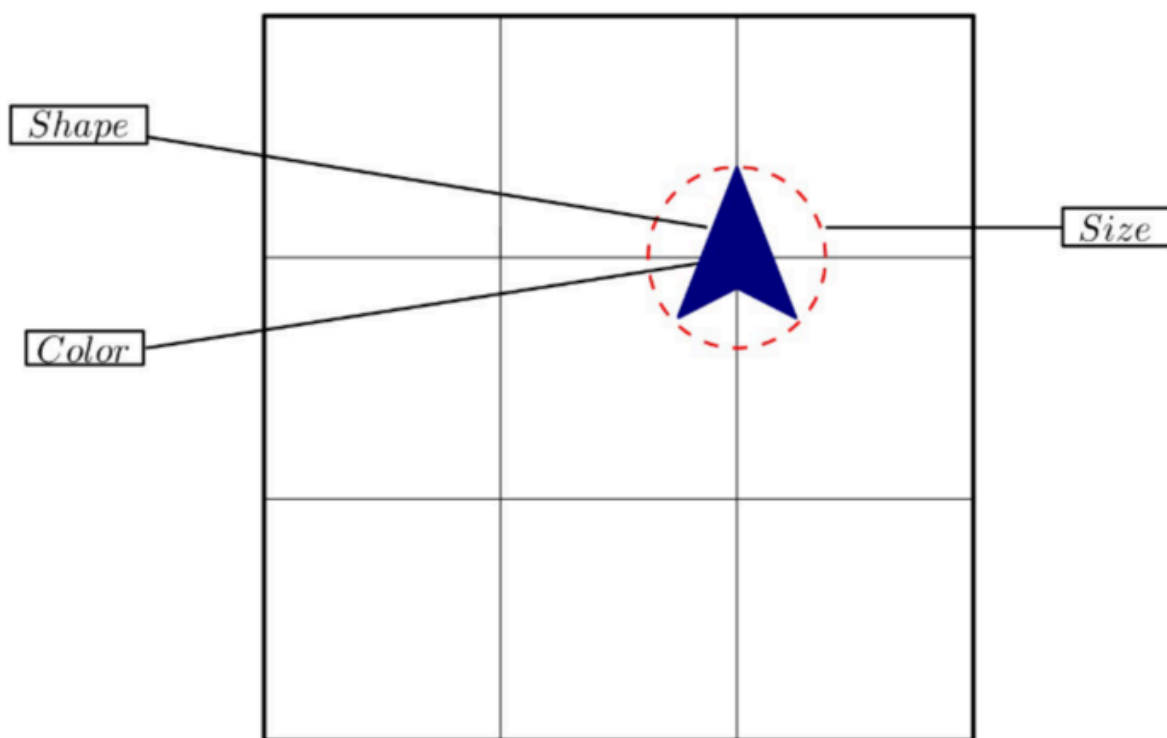
2.2.2 The Agents

The environment is composed of **agents called turtles** that can independently move. Each turtle has a **position, coordinates, and a heading**, expressed in degrees. 0° is north.



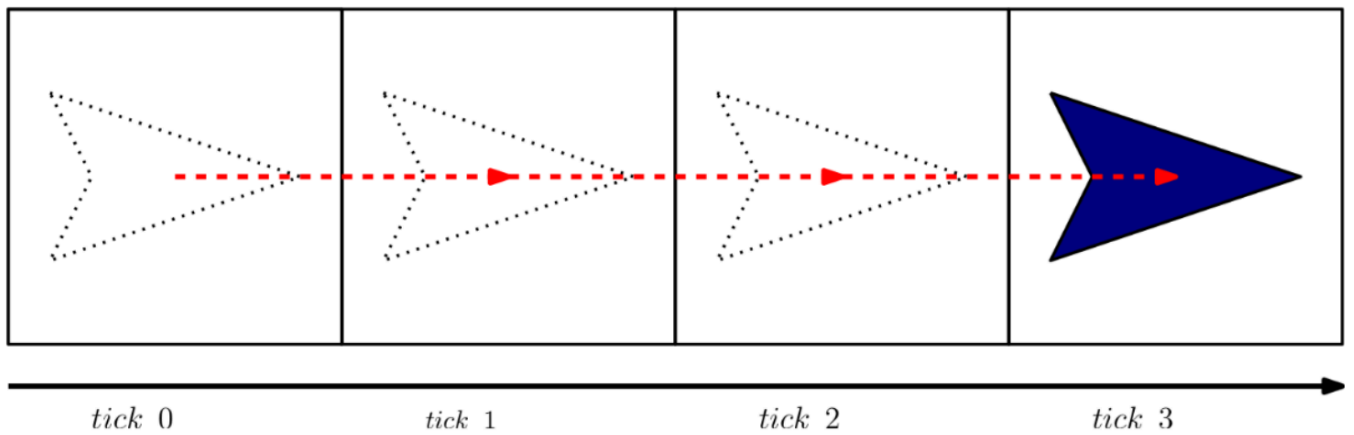
Reference: <https://ccl.northwestern.edu/netlogo/>

Agents possess descriptive features as well such as their **size**, **color** and **shape**. (Mostly used for visualization purposes)



Reference: <https://ccl.northwestern.edu/netlogo/>

Just like space, time in simulations is also discrete, progressing in units called **ticks**. A tick represents a moment in simulation time during which agents perform their actions. By default, a scheduler ensures agents act in a random order each tick, though this behavior can be customized as needed.



Reference: <https://ccl.northwestern.edu/netlogo/>

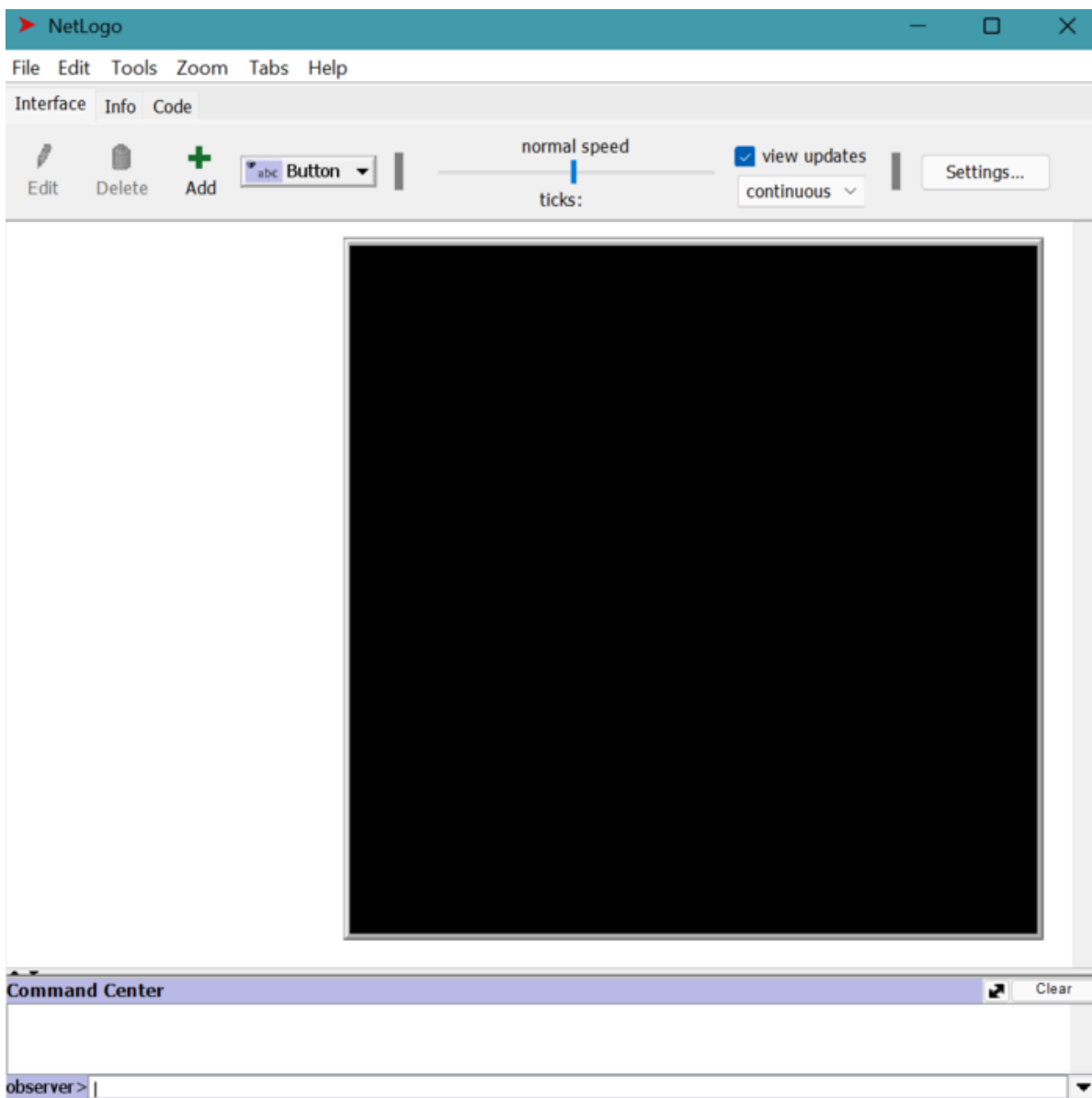
Each agent is equipped with a set of properties:

- **who**: A unique identifier assigned to each agent, distinguishing it from others in the simulation.
- **heading**: The direction the agent is facing, typically measured in degrees.
- **xcor and ycor**: The agent's coordinates on the grid, defining its position in the simulation environment.
- **shape, size, color**: Visual attributes of the agent, determining how it appears in the simulation.
- **hidden**: A boolean property indicating whether the agent is visible or hidden in the simulation.

2.2.3 The Observer

The **observer** in NetLogo acts as an overseer, responsible for managing and modifying the environment and agents without being an agent itself. It can execute commands to create, move, or modify turtles, patches, and links, as well as control the simulation by adjusting global settings, running procedures, and monitoring overall behavior.

2.2.4 Starting Up NetLogo (6.4.0)



Opening NetLogo presents a minimalist interface with a blank project, including an empty grid window by default. The easiest way to begin interacting with this environment is through the **Command Center**.

By default, you are acting as the **observer**, which grants full control over the entire environment with a global perspective. In observer mode, commands are executed at the global level and can directly manipulate the environment, agents, and simulation settings.

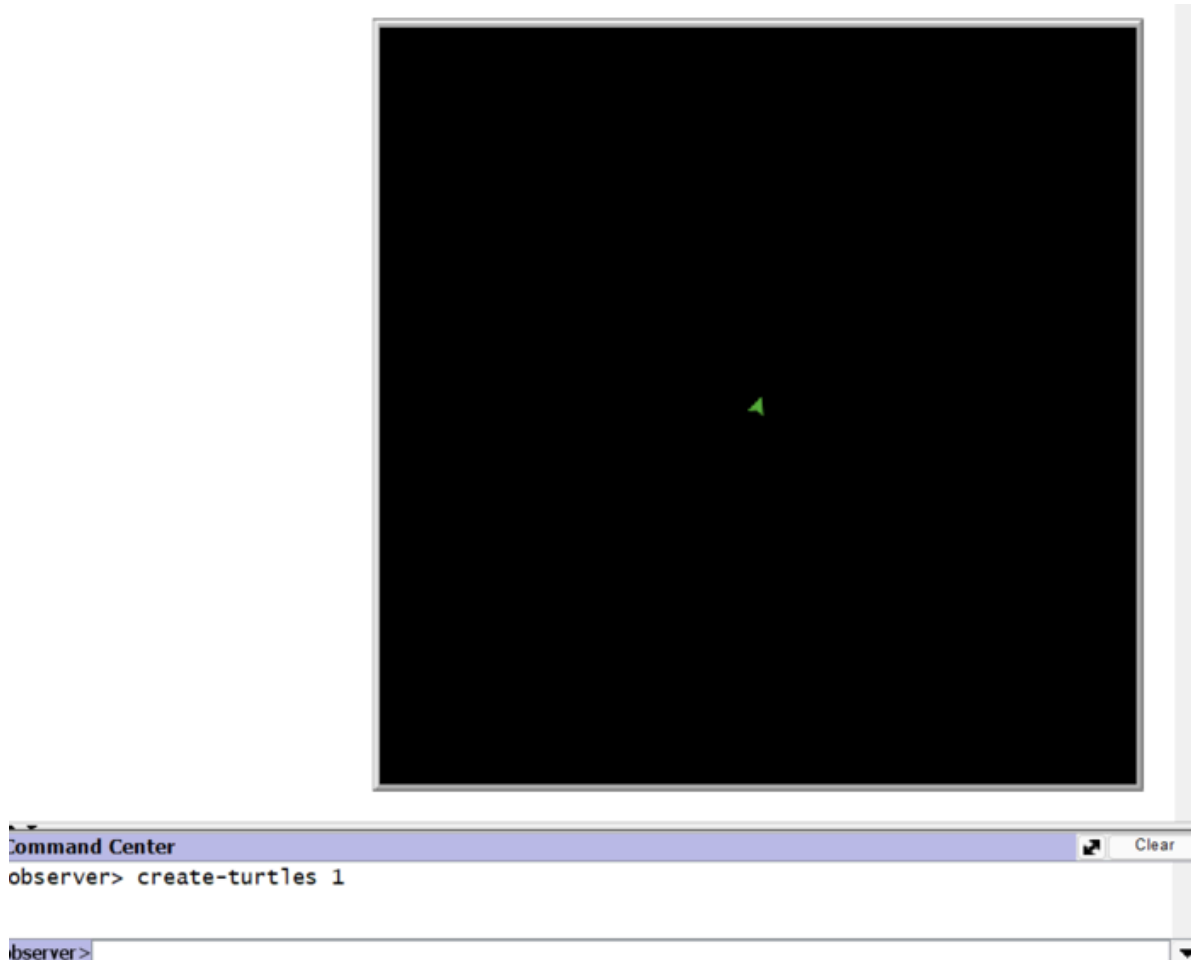
2.2.5 Appetizer Commands

```
create-turtles 1
```

This code may appear simple, but it performs multiple actions behind the scenes:

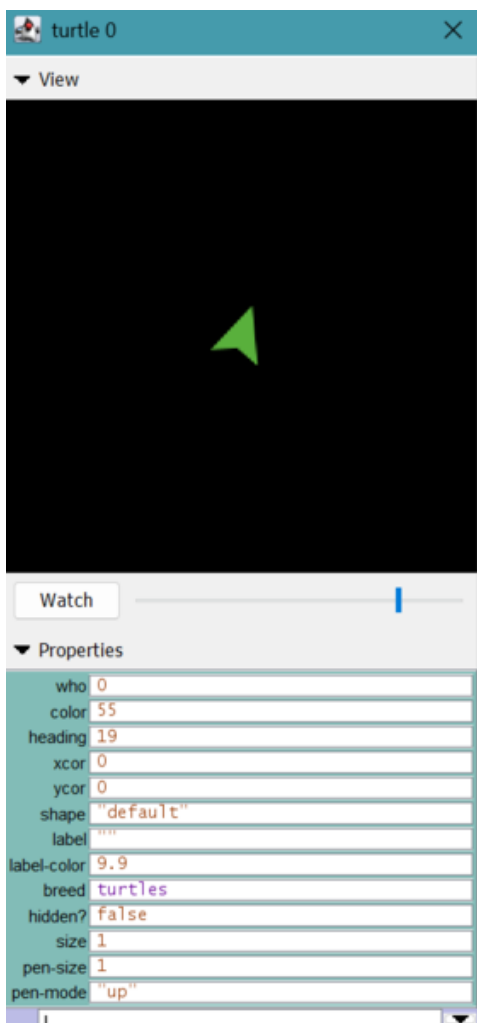
- **Action:** It creates **1 new turtle** in the simulation.
- **Default Properties:** The newly created turtle is assigned default values for its properties, such as:
 - A unique identifier (who number, in this case it will be 0).
 - Randomly chosen initial heading (direction).
 - A random xcor and ycor (position) within the world's boundaries.
 - Default visual properties like shape, color, size, and hidden status (not hidden by default).

- **Scope:** This command is executed from the **observer** context, meaning the observer initiates the creation of the turtle(s) in the environment.



`inspect turtle 0`

The command `inspect turtle` is used to inspect all properties of a turtle.



In this window, you can observe additional properties of the turtle that were not mentioned previously:

- **label:** A text string displayed next to the turtle. By default, it is empty, but it can be customized to display numbers, words, or other information.
- **label-color:** The color of the text in the turtle's label, displayed as a numerical color code.
- **breed:** The classification of the turtle, used to group turtles into subcategories for specific behaviors or roles. By default, turtles belong to the `turtles` breed.
- **pen-size:** The width of the pen used by the turtle when drawing on the grid, measured in pixels.
- **pen-mode:** Determines the drawing behavior of the turtle's pen. Possible values include `up` (not drawing), `down` (drawing as it moves), or `erase` (erasing lines as it moves).

There is also an **input field** at the bottom of the Inspect window, which allows you to directly modify properties or execute commands for the selected agent or object (such as a turtle, patch, or link) in the simulation.

For example, to change the label of a turtle, you can use the following command:

```
set label "Shrek"
```

The `set` command modifies the properties of the agent selected in the Inspect window. However, if you want to do this from the **observer level**, you need to use the `ask` command to specify which agent you are targeting:

```
ask turtle 0 [set label "Donkey"]
```

This observer-level command is slightly modified from the original, with the inclusion of the `ask` keyword, along with the `breed` (`turtle`) and the unique identifier (`who`) of the agent in question. The `ask` command in NetLogo allows the observer to direct specific agents (or groups of agents) to perform actions.

NOTE:

1. The `set` command works within the context of the selected agent and can modify its own properties directly.
2. Using the `ask` command, an agent can modify the properties of another agent. When an agent uses `ask`, it essentially "steps into" the context of the target agent(s).

Exercise:

1. Use the `create-turtles` command in the **Command Center** to add a new turtle.
2. Right-click on the newly created turtle (Turtle 1) and select "Inspect" to open its Inspect window.
3. In the input field of Turtle 1's Inspect window, try changing Turtle 0's label to "Shrek" again.

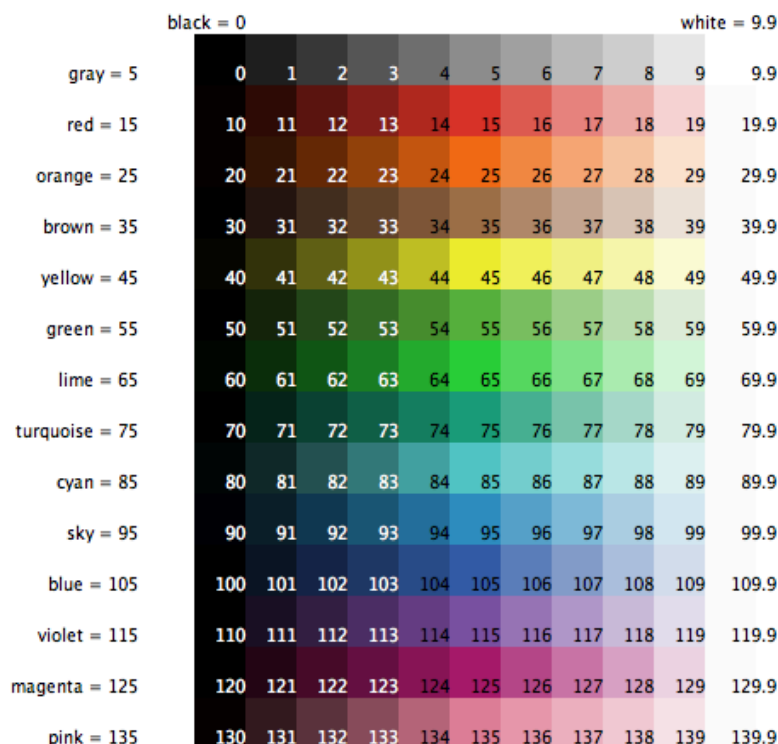
2.2.6 The Color Property

You might have noticed that the `color` of **Turtle 0** is set to 55, which might seem unusual. This is a greenish color in NetLogo's color scheme. Additionally, the `label-color` is set to 9.9, which corresponds to white.

Let's try changing the `label-color` of **Turtle 0** to a different value, such as 19.9:

```
set label-color 19.9
```

Surprisingly, nothing seems to happen. So, let's explore what's going on behind the scenes and understand how colors are coded in NetLogo.



Reference: <https://ccl.northwestern.edu/netlogo/bind/article/shapes-and-colors-in-netlogo.html>

NetLogo uses a **continuous color scale** based on numbers ranging from 0 to 140. These numbers represent specific colors on the NetLogo color wheel:

- **Whole Numbers:** Each whole number corresponds to a base color (e.g., 0 is black, 15 is red, 65 is green, etc.).

- **Decimal Points:** Decimal values (e.g., 19.9) create shades or variations of the base color. For example:
9.9: The lightest shade of a color, often close to white and 19.9: A lighter variation of red.

Let's try changing the `label-color` of **Turtle 0** to a value that is outside the bounds of NetLogo's color system:

```
set label-color 155
```

Interestingly, the `label-color` turns red. This happens because NetLogo handles out-of-bounds color values by applying the following formula: $\text{color} = \text{set_color} \% 140$ (graceful handling).

2.2.7 Shapes

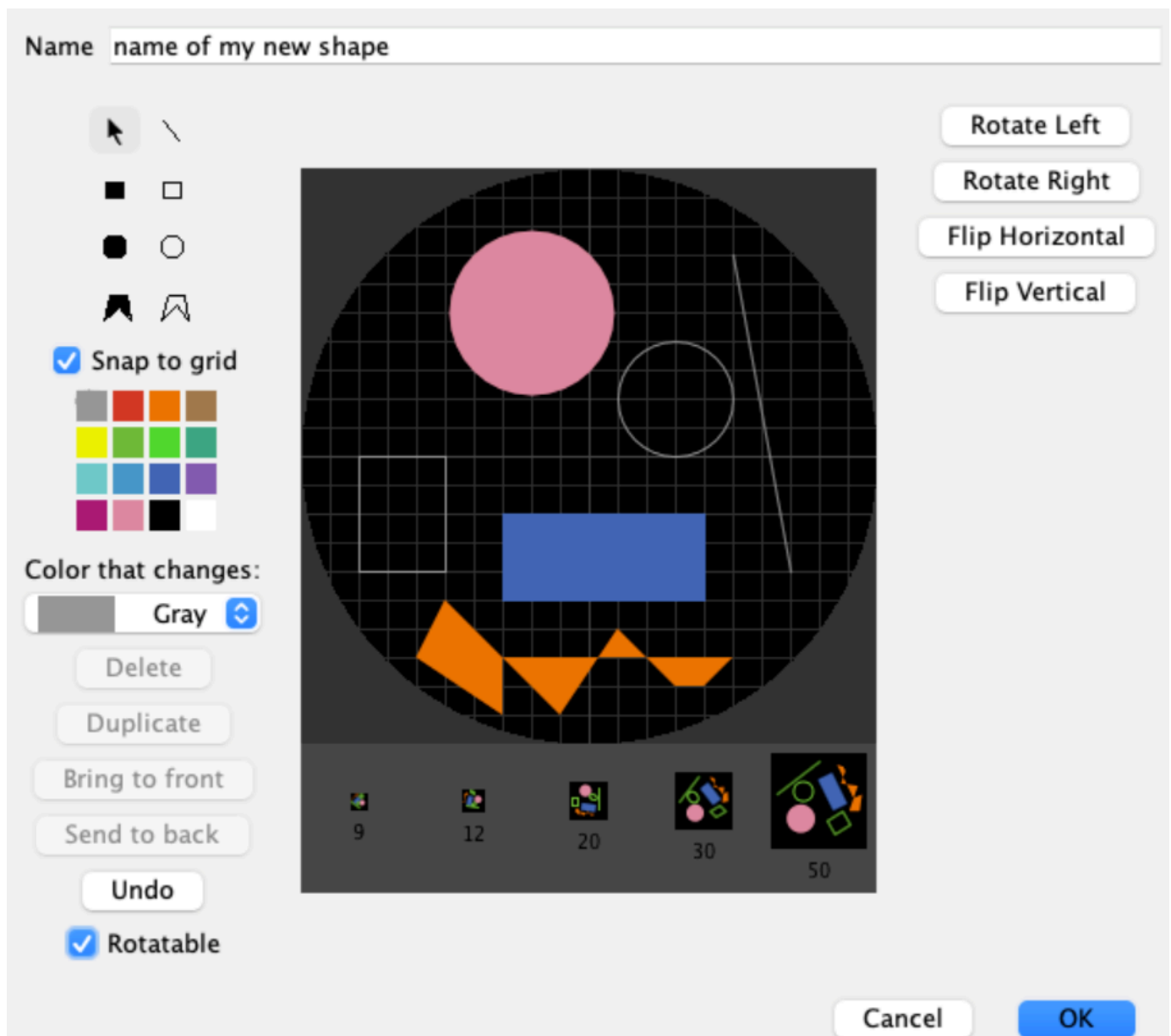
For some phenomena, modeling how agents **look** can be just as important as modeling their behavior. In other cases, creating visually appealing and creative visualizations can enhance our understanding and enjoyment of the modeling process.

NetLogo uses **vectorized shapes** for turtles, which are built from basic geometric components. By default, turtles use the `default` shape, but NetLogo also provides a library of pre-defined shapes that can be assigned to turtles to represent different roles or states visually. These shapes can be customized to suit the needs of your model.



Other notable shapes include the following: `airplane`, `bug`, `butterfly`, `person`, `house`, `car`.

NetLogo has way more turtle shapes than the default ones for us to choose from. All we need to do is to click the **Import From Library** button, which will bring up a long list of shapes to choose from.



Reference: <https://ccl.northwestern.edu/netlogo/bind/article/shapes-and-colors-in-netlogo.html>

2.2.8 The Breed Property

The **breed** property defines a classification of agents, specifying their roles within the system. NetLogo provides a fallback breed called `turtles`, which is the default class for all agents unless explicitly assigned to another breed. This ensures that agents always have a default classification, even if no additional breeds are defined.

You can define additional breeds to represent different roles or behaviors in the system. For example, in a simulation of hunters and prey, you could create two separate breeds:

- **Hunters:** Agents that have specific goals and actions, such as chasing prey.
- **Prey:** Agents with different behaviors, like avoiding hunters or foraging for resources.

2.2.9 The Pen Mode Property

The **pen-mode** property enables agents to leave a visual trail, following their trajectory as they move around the environment. The property can take the following values:

- **up:** The pen is lifted, and no trail is drawn as the agent moves.
- **down:** The pen is lowered, drawing a trail along the agent's path.

- **erase** : The pen erases any previously drawn trails as the agent moves.

This feature allows for the creation of intricate visual patterns, showing emergent behaviors in multi-agent systems through simple movement rules. Let's set the `pen-mode` of **Turtle 0** to `down`.

```
set pen-mode "down"
```

2.2.10 Moving the Agents

Basic movement in NetLogo involves the following commands:

- **forward** : Moves the agent in the direction specified by its current `heading` property.
- **right** and **left** : Adjust the `heading` value of the agent, changing its direction of movement.

The **left** command subtracts the specified angle from the current `heading`, while **right** adds the specified angle to it. It's important to note that in NetLogo, the `heading` value is measured in degrees, with **0 degrees** representing north. For example:

- If an agent's `heading` is **180** (facing south) and you execute `right 180`, the agent will turn to face north (back to a `heading` of 0).

Exercise: Use **Turtle 0** to draw a perfect **equilateral triangle** (a triangle with three equal sides and 60-degree angles) on the simulation grid.

You will use the **pen-mode** property and the basic movement commands (`forward` and `right`) to accomplish this.

2.2.11 Patches

In NetLogo, there are four types of agents: **turtles**, **patches**, **links**, and the **observer**. Commands can be directed to any of these agents, including patches.

Patches are arranged in a grid with each patch having specific coordinates. The patch at coordinates (0, 0) is called the **origin**, and the coordinates of other patches are determined by their horizontal and vertical distances from this origin.

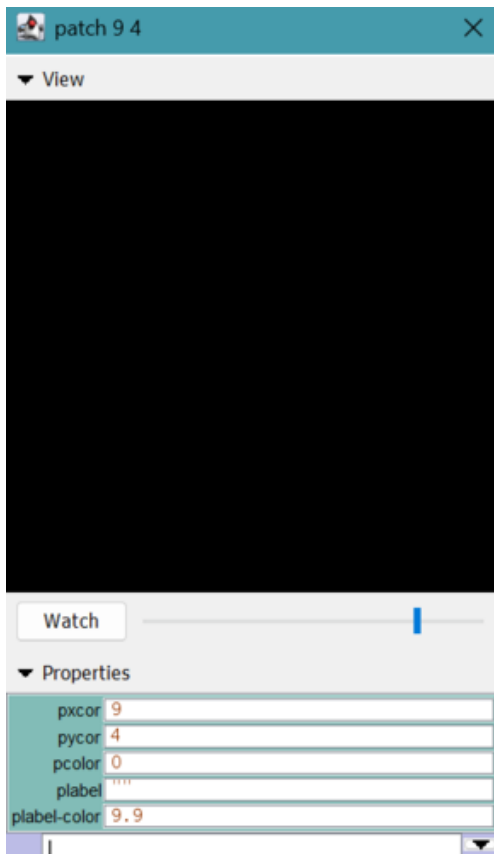
- **pxcor** : The horizontal coordinate (increases as you move to the right).
- **pycor** : The vertical coordinate (increases as you move upward).

These coordinates work similarly to the standard mathematical coordinate plane.

Commands in NetLogo can target a **specific turtle** or **specific patch** or the **entire set of turtles or patches**.

	Turtles	Patches
One	<code>ask turtle 0 [set color red]</code>	<code>ask patch 2 3 [set pcolor red]</code>
All	<code>ask turtles [set color red]</code>	<code>ask patches [set pcolor red]</code>

See <https://ccl.northwestern.edu/netlogo/docs/dictionary.html> for additional commands.



Patches also have a set of properties that can be manipulated, such as their **color** or **label**. For example, you can change the color of a patch to standard white by setting its color property to 9.9 . Here's how you can do it:

```
ask patch 9 4 [set pcolor 9.9]
```

So what would be needed to create, for example, a chessboard pattern on the grid? Is there anything beyond the Inspect Window and the Command Center to write more complex code, such as loops, creating breeds, handling complex data structures, and manipulating multiple elements at once? Of course, there is: **The Code Tab**.

✓ 2.3 Programming in NetLogo

Instructions to agents can be classified according to three criteria:

- whether they are built into NetLogo (**primitive**) or user implemented (**procedure**)
- whether the instruction produces an output (**report**) or not (**command**)
- whether an instruction takes inputs or not

NOTE:

- NetLogo is case insensitive, so case conventions are purely for reader convenience.

2.3.1 Commands

Commands are procedures that don't have any output, but only side effects on the environment.

```
to go
  clear-all
  create-turtles 10
```

```
ask turtles [ forward 1 ]
end
```

This code defines a procedure called `go`, which performs the following actions:

1. **clear-all**: Resets the environment by clearing all agents, patches, and any previously drawn elements on the grid.
2. **create-turtles 10**: Creates 10 new turtles, each with default properties like random positions and headings.
3. **ask turtles [forward 1]**: Asks all turtles to move forward by 1 step in the direction they are currently facing.

This command can then be called in the **Command Center** with the following line:

```
go
```

2.3.2 Interface Elements

Is there another way to interact with the Code Pane from the Interface Tab? **Yes**, through **Interface Elements**, which allow users to modify and interact with the simulation without directly changing the code. The most notable elements include:

- **Button**: Executes a specific procedure or command when clicked.
- **Slider**: Adjusts numeric values dynamically to control variables.
- **Switch**: Toggles between `true` and `false` for boolean variables.
- **Chooser**: Allows selection from a predefined list of options.
- **Input**: Accepts user-provided text or numeric input.
- **Monitor**: Displays the current value of a variable in real time.
- **Plot**: Visualizes data over time or for specific conditions.
- **Output**: Prints text or data to a log-like area in the interface.
- **Note**: Displays descriptive text or instructions for the user.

Exercise: Figure out which **Interface Element** would be appropriate to call this command once without interacting with the **Command Center**.

Can you also determine how to call this function continuously within a loop?

2.3.3 Reporters

Reporters are procedures that compute a value and report it.

```
to-report double [ num ]
  report 2 * num
end
```

The above code defines a reporter named `double` that performs the following actions:

1. **Input Parameter**: It takes a single input, `num`, which is the value to be processed.
2. **Computation**: It multiplies the input (`num`) by 2.
3. **Reporting the Result**: The `report` keyword is used to return the computed value (i.e., `2 * num`).

This reporter can be called in the **Command Center** or within other procedures to compute the double of a given number. For example:

show double 5

In the code above, `num` acts as an **input parameter** to the command.

Exercise: Figure out which **Interface Element** would be most appropriate to monitor the value of the `double` reporter.

2.3.4 Styling

There isn't an official NetLogo style guide. Nonetheless the official documentation is fairly consistent and follows some good habits:

- use camel case beginning with a lower-case letter for procedure (e.g. `myProcedure`, Java style)
- do not use underscores in names
- name command procedure with nouns and reporters with verbs

2.3.5 Variables

Variables in NetLogo can be divided into three main groups:

- **Local variables**, defined as part of a procedure: `let <name> <value>`
- **Agent variables**, defined as part of each agent: `<agent*>-own [<name(s)>]`
- **Global variables**, accessible by every agent and procedure: `globals [<name(s)>]`

Exercise: Create a global variable named `radius` and set its value to 5 in a command named `setup`. Subsequently, create a reporter called `calculate-area` that calculates the area of a circle. In this function, create a local variable named `area` that computes the area of the circle using the formula $\text{area} = \pi \times \text{radius}^2$, where `radius` is the global variable. The reporter should then return the computed area of the circle.

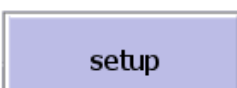
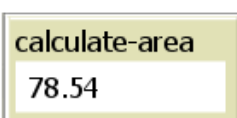
Use a **Button** interface element to run the `setup` command once, then utilize a **Monitor** interface element to show the result **with 3 decimal places**!

Hints:

- Use the `globals` keyword to define the global variable `radius`.
- Use the `set` command to assign the value 5 to `radius`.
- `pi` is already a predefined constant in NetLogo, so you don't need to define it manually.

Solution:

► Click to show/hide solution



NOTE:

- NetLogo variables are dynamically typed.

- Primitive types are **numbers, booleans, lists, strings**, along with the usual operations: `+`, `-`, `*`, `/`, `^`, `>`, `>=`, `=`, `!=`, `<`, `<=`, `and`, `or`, `not`, `xor`.
- **All numbers are floating points**; be aware of approximations.
- When performing arithmetic operations **be aware of spaces**: the lack of parentheses might bring ambiguity in parsing the operation and result in something different.

2.3.6 Agentsets

When asking to update an agent variables a **subset of all the agents**, called `agentset`, can be used. An `agentset` contains one or more agents, all of the same type, and it's always randomly ordered.

```
ask one-of turtles [ <command> ]
```

The `one-of` primitive in NetLogo randomly selects one agent from a given set of agents, such as turtles, patches, or links. For example, `one-of turtles` randomly selects one turtle from the current set of turtles. Additionally, you can create subsets of agents using conditions (e.g., `turtles with [color = red]`) and then instruct these specific subsets with targeted commands.

```
let some-patches patches with [ pxcor < 3 ]
ask some-patches [ set pcolor red ]
```

2.3.7 Conditionals

Conditionals in NetLogo allow agents to make decisions based on specific criteria using commands like `if`, `ifelse`, and `ifelse-value`. For example, `if pcolor = black [set pcolor white]` changes a patch's color to white only if its current color is black.

```
if (<condition>) [ <command(s)> ]
```

```
ifelse (<condition>)
  [ <command(s) if true]
  [ <command(s) if false]
```

```
ifelse-value (<condition>)
  [ <reporter(s) if true]
  [ <reporter(s) if false]
```

```
if (random-float 1 < 0.5)
  [ show "heads" ]
```

```
ifelse (random-float 1 < 0.5)
  [ show "heads" ]
  [ show "tails" ]
```

```
ask turtles [
  set color ifelse-value (energy < 0)
    [ red ]
    [ green ]
]
```

Conditions are logical expressions (=, <, >, and, or, etc.) that evaluate to `true` or `false`.

NOTE:

- When using `if` or `ifelse` in an `ask` block, the condition is evaluated for each agent individually.

```
ask turtles [ if xcor > 0 [ set color red ] ]
```

2.3.7 Loops

Loops in NetLogo allow repeated execution of commands, enabling dynamic and iterative behaviors. Common looping constructs include `repeat`, which runs a block of commands a fixed number of times, and `while`, which runs as long as a specified condition is true.

```
loop [ <command(s)> ]
```

```
repeat <num> [ <command(s)> ]
```

```
foreach <list> [ [<item>] -> <command(s)> ]
```

```
loop [ ifelse (counter > 100)
  [ stop ]
  [set counter counter + 1]
]
```

```
repeat 5 [
  ask one-of turtles [ set color red ]
]
```

```
foreach [1 2 3] [ [num] -> show num * 2 ]
```

NOTE:

- Loops inside an `ask` block are executed independently for each agent.

```
ask turtles [ repeat 5 [ forward 1 ] ]
```

2.3.8 Lists

Lists in NetLogo are ordered collections of items, which can include numbers, strings, agents, or other lists. They are data structures that support operations like adding, removing, or accessing elements.

```
( list <element(s)> )
```

```
[ element(s) ]
```

```
( list 1 "two" true)
```

```
[ 1 "two" true ]
```

NOTE:

- In NetLogo, lists are **immutable, ordered, and potentially heterogeneous**.

Some examples:

```
let colors ["red" "blue" "green"]
show item 1 colors
```

The output will be the first element or `item` in the list, which is `blue`.

```
let my-list [1 2 3]
set my-list replace-item 1 my-list 99
```

`my-list` will contain the values of `[1, 99, 3]` after using `replace-item`.

The `lput` primitive command adds an element to the end of a list, while `fput` adds an element to the beginning.

2.3.9 Program Structure

The flexibility of NetLogo and its agent-centered way of building models quickly escalates to complex models that are difficult to work with.

Try to keep your structure as close as possible to:

- **global variable** declaration;
- **agent variable** declaration;
- **setup procedure**, in which global variables are initialized, agents are created and the environment is initialized;
- **go procedure**, which implements one cycle of the simulation.

2.3.10 Higher-Order Procedure

Even though NetLogo is not a higher-order language we can simulate this behavior using **anonymous procedures/reporters**.

```
[ [ <var(s)> ] -> <body> ]
```

- `[]`: Encloses the entire anonymous procedure or reporter.
- `[<var(s)>]`: Specifies input variables (like function parameters) in a nested bracket. These variables can be used within the body.
- `->`: Indicates the start of the body of the procedure or reporter.
- `<body>`: The actual commands or expressions to execute. If it's a reporter, the result of this expression is returned.
- *Higher-order procedure:**

```
[ [ x y ] -> setxy y x ]
```

Anonymous procedures assigned to variables (tasks):

```
globals [ stack push ]

to setup
  set stack [] ; Initializes the stack as an empty list.
  set push [el -> set stack lput el stack] ; Defines the push task.
  run push 10
end
```

Higher-order reporter:

```
foreach [1 2 3] [ [x] -> show x * x ]
```

Unlike traditional procedures or reporters, anonymous ones are not stored in the **Code Tab** and cannot be reused unless redefined. While NetLogo doesn't directly support higher-order functions, anonymous procedures allow for similar behavior in many cases.

NOTE:

- **Map, filter, and reduce** are basic constructors that allow efficient and elegant operations on lists.
- `map` applies an anonymous reporter to every element in a list.
- `filter` applies a predicate (in the form of an anonymous reporter) to a list and returns only those items that satisfy the predicate.
- `reduce` applies an anonymous reporter from left to right, resulting in a single value.

```
map [ a -> a * a ] [ 1 2 3 ]
```

```
filter [ a -> a > 5 ] [ 1 9 2 ]
```

```
reduce [ [a b] -> a + b ] [ 1 9 2 ]
```

✓ 2.3.11 Breeds:

In NetLogo breeds are a way to "subclass" the turtle type.

```
breed [ <single name> <agentset name> ]
```

```
breed [ hunter hunters ]
breed [ prey preys]
```

After a breed has been created, the `ask` command can be used with the breed name (e.g., `ask hunters`) to execute actions for agents of that specific breed. All commands and properties applicable to turtles can also be used with the newly defined breed.

Exercise: Now that we've learned how to interact with the Code Pane, call functions from the Interface Tab, and use loops through interface elements, your task is to create a custom command called `setup-chessboard`. This command will clear the simulation environment and create a classic chessboard pattern on the grid using black and white patch colors.

```
breed [ hunter hunters ]  
breed [ prey preys]
```

After a breed has been created, the `ask` command can be used with the breed name (e.g., `ask hunters`) to execute actions for agents of that specific breed. All commands and properties applicable to turtles can also be used with the newly defined breed.

Exercise: Now that we've learned how to interact with the Code Pane, call functions from the Interface Tab, and use loops through interface elements, your task is to create a custom command called `setup-chessboard`. This command will clear the simulation environment and create a classic chessboard pattern on the grid using black and white patch colors.

Optional:

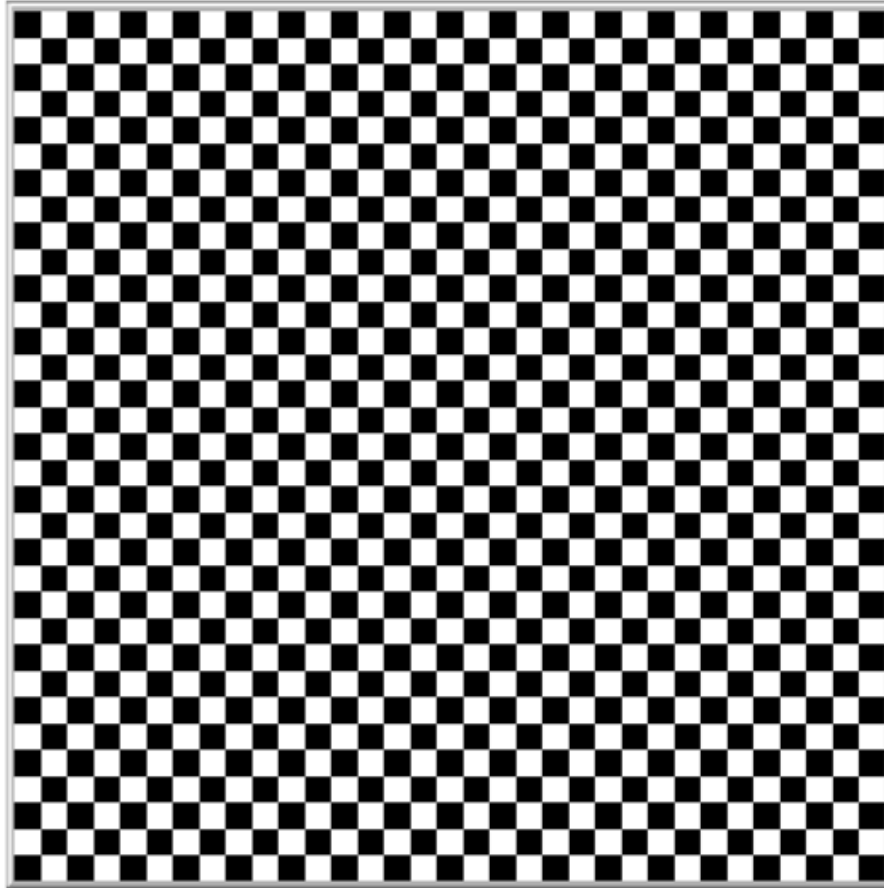
- Modify the grid size to fit different chessboard sizes (e.g., 8x8, 16x16).
- Use custom patch colors instead of black and white.
- Add another **Button** to clear the chessboard or overlay agents on specific patches.

Solution:

▼ Click to show/hide solution

```
to setup-chessboard  
  clear-all  
  ask patches [  
    if (pxcor + pycor) mod 2 = 0 [ set pcolor black ]  
    if (pxcor + pycor) mod 2 = 1 [ set pcolor white ]  
  ]  
end
```

setup-chessboard



Exercise: Design a **garden pattern** on the NetLogo grid using patches and turtles. The garden will consist of alternating flowerbeds (colored patches) and turtles (acting as flowers) placed in specific areas.

Tasks:

1. Write a procedure called `setup-garden` to:
 - Clear the environment.
 - Color the patches in a checkerboard pattern to represent flowerbeds.
 - Place turtles (flowers) only on the green patches.
2. Customize the turtles:
 - Set their **shape** to a flower (use `circle` if `flower` is unavailable in your version).
 - Randomize their **size** and **color** to make the garden more realistic.
3. Use **ticks** to control the simulation. On each tick, make the turtles **grow** slightly (increase their size).

Hints:

- Use the `ask patches` command to create the checkerboard pattern.
- Use the `ask turtles` command to set their properties dynamically.
- Utilize loops, conditions (`if` statements), and lists where needed.

Solution:

▼ Click to show/hide solution

```
to setup-garden
  clear-all
```

```

ask patches [
  if (pxcor + pycor) mod 2 = 0 [ set pcolor green ]
  if (pxcor + pycor) mod 2 = 1 [ set pcolor brown ]
]

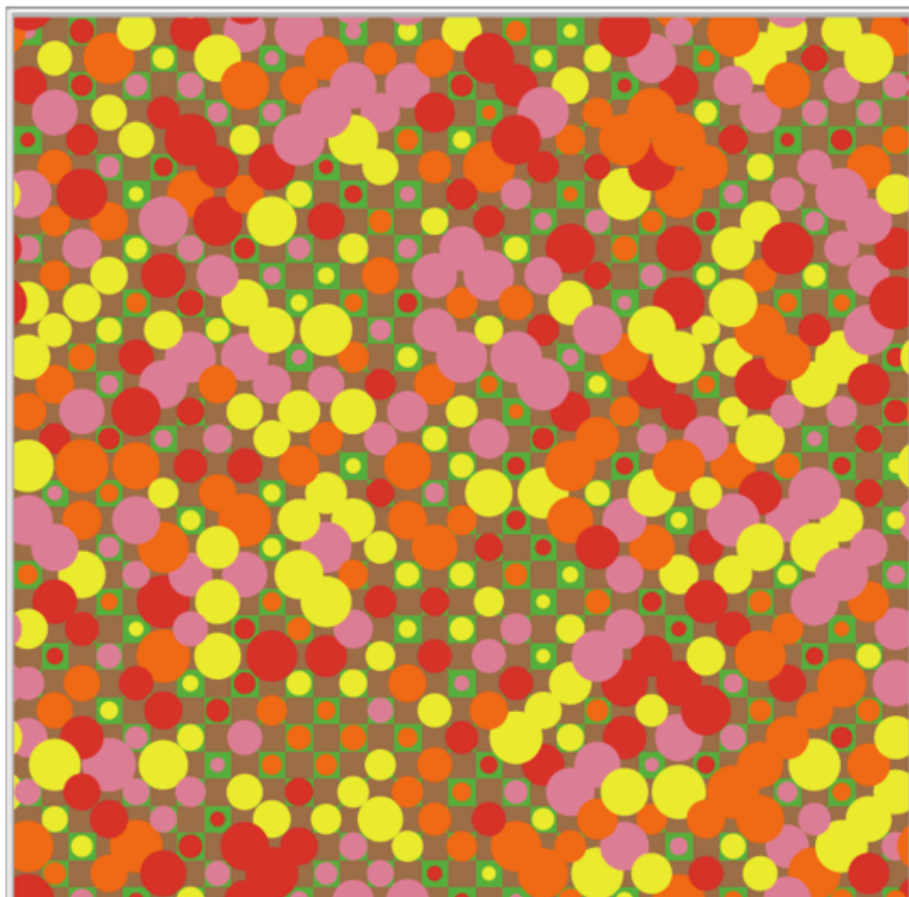
ask patches with [pcolor = green] [
  sprout 1 [
    set shape "circle"
    set size random-float 1.5 + 0.5
    set color one-of [red pink yellow orange]
  ]
]
reset-ticks
end

to grow
  ask turtles [
    set size size + 0.1
  ]
  tick
end

```

setup-garden

grow 



✓ 3. Practice - NetLogo Simulations

👤 Tamás Takács, PhD student, Department of Artificial Intelligence

🕒 90 min read

📅 January 22, 2025

📚 Collective Intelligence



ELTE | IK
INFORMATIKAI KAR



DEPARTMENT
OF ARTIFICIAL
INTELLIGENCE

NetLogo



This practice notebook introduces students to the fundamental structure of NetLogo's basic models and the resources available in the NetLogo Model Library. Through hands-on exercises, students will gain experience in loading and analyzing standard models, executing multiple simulations in parallel, and systematically documenting their NetLogo code. The activities also reinforce core skills developed in Practice 2, guiding students in combining essential NetLogo building blocks to design and implement elementary agent-based models.

✓ Table of Contents

- **3.1 Models Library**
 - 3.1.1 Benchmark Simulations
- **3.2 The Fire Model**
 - 3.2.1 Mechanics
 - 3.2.2 Running the Simulation
 - 3.2.3 The Code
 - 3.2.4 Preview Commands Editor
 - 3.2.5 BehaviorSpace
 - 3.2.6 The Info Tab
- **3.3 Schelling's Segregation Model**
 - 3.3.1 Plot Interface Element
 - 3.3.2 The Code
 - 3.3.3 Modifying the Code
- **3.4 Virus on a Network Model**
 - 3.4.1 Links
 - 3.4.2 Network Building
 - 3.4.3 The Main Loop

Last Practice

In our last practice, we covered:

- An introduction to NetLogo and its competitor programmable modeling environments
- The advantages of using NetLogo
- World and Agents
- Turtles and their properties
- The Observer
- Reporters and Commands
- Variables and their types
- Conditionals and loops
- Lists and higher-order functions
- Tasks, including `map`, `filter`, and `reduce`
- Working with Breeds

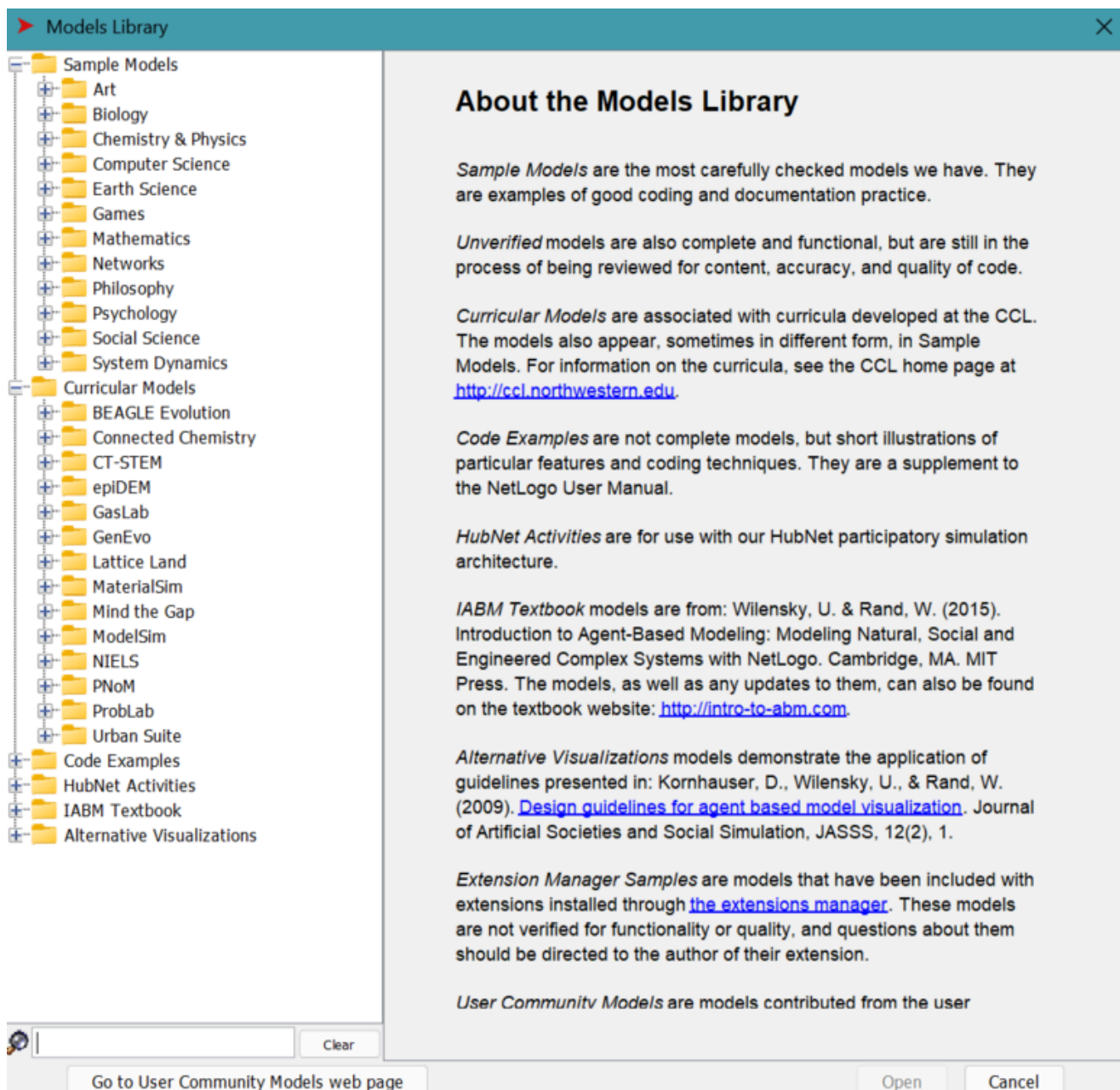
That's quite a lot! Now, let's dive into how these concepts are applied in simulations using the **Models Library**.

✓ **3.1 Models Library**

The **NetLogo Models Library** is a comprehensive resource included with NetLogo, containing pre-built simulation models spanning diverse disciplines such as biology, economics, physics, and social sciences.

- It includes over **200 models**, categorized for easy navigation (e.g., Biology, Social Science, and Computer Science).
- Models like *Wolf Sheep Predation* and *Segregation* have become benchmarks for studying agent-based systems.
- Models are **fully editable**, allowing users to modify parameters, add features, or adapt them for custom research needs.

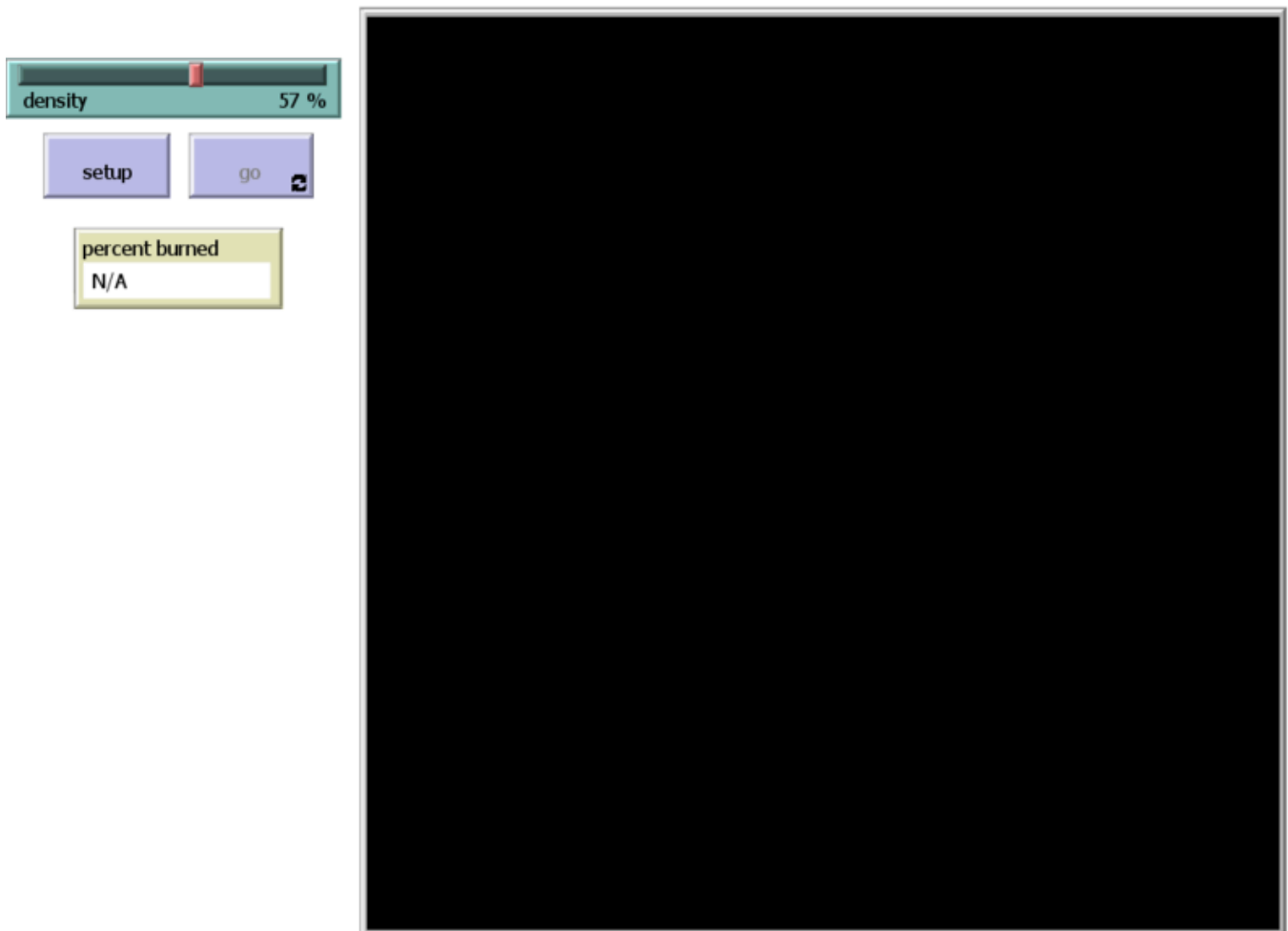
3.1.1 Benchmark Simulations



Your **Model Library** should look somehow like this under *File > Models Library*.

3.2 The Fire Model

File > Models Library > Sample Models > Earth Science > Fire



Interface Tab

The **Fire Model** is a simple yet illustrative agent-based simulation used to study the spread of fire through a forest, represented on a 2D grid. The environment consists of three primary elements:

1. **Trees:** Represented as green patches on the grid.
2. **Fires:** Represented by red turtles that simulate active fire.
3. **Embers:** Representing fading fire agents, marking burned areas.

3.2.1 Mechanics

The `setup` button initializes the forest grid based on a key variable called `density`. This global variable determines the proportion of the grid occupied by trees and is controlled by a slider in the interface. The `density` variable is only utilized during the setup phase and remains constant throughout the simulation, even if the slider is adjusted during runtime.

The `go` button initiates the core simulation loop, which governs the spread of fire:

- Fire agents ignite adjacent trees based on predefined rules.
- Burned trees transition into embers, and fire agents disappear once their task is complete.

The model includes a monitor labeled `percent burned`, which reports the percentage of the grid area affected by fire. This value is calculated by a reporter function that dynamically tracks the burned areas during the simulation.

3.2.2 Running the Simulation

Let's run the simulation with the initial `density` value of **57%**.

Exercise: Run the `setup` command in the Interface tab and press the `go` button to start the simulation.

Observe what happens to the forest:

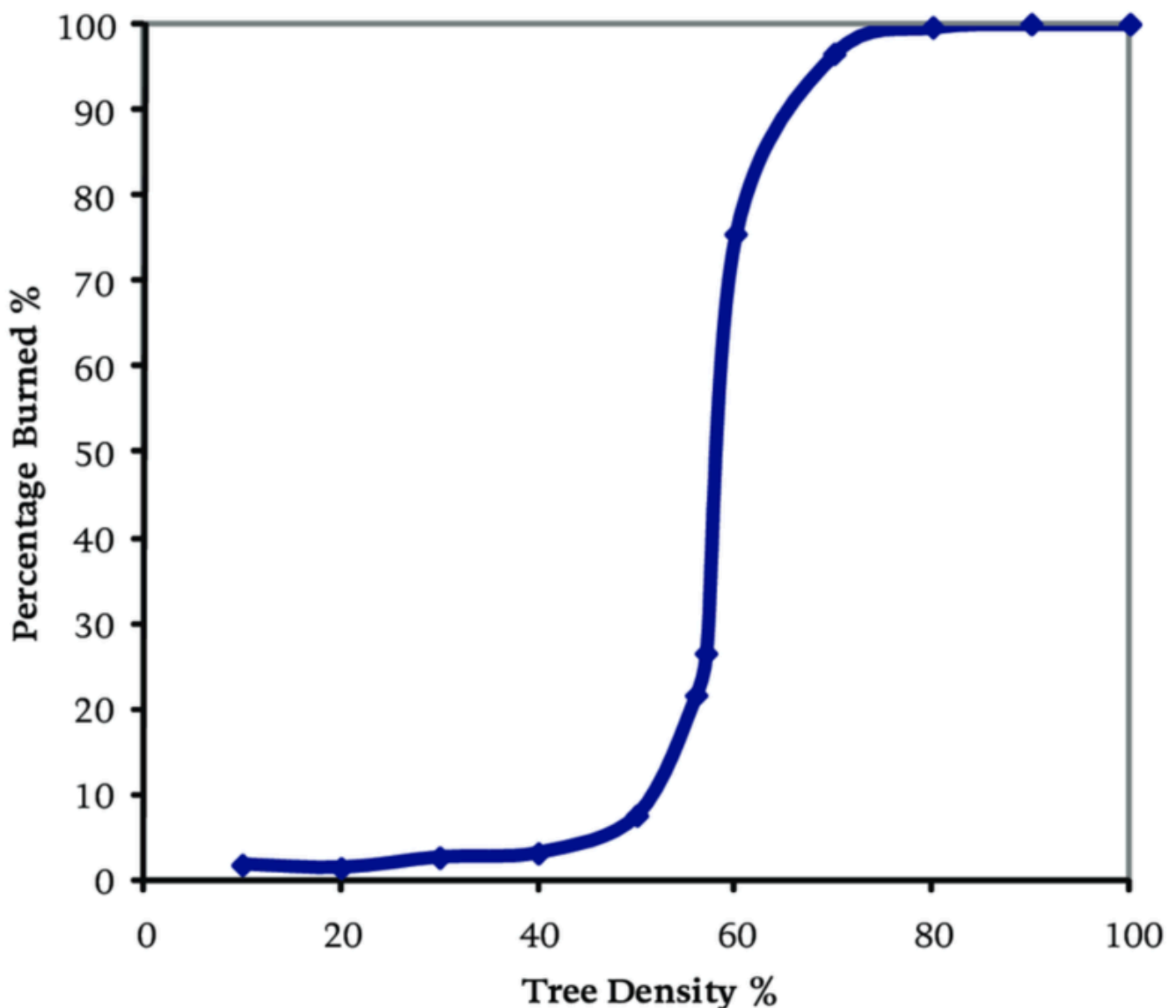
- Does the forest burn down completely? If yes, why? If no, why not?
- What is the **percent burned** value displayed in the monitor?
- Try to explain your observations based on the behavior of fire spread before looking into the code.

Exercise: Find the **epidemic threshold**—the critical tree density where the fire just barely spreads across the entire forest (agents reach the right side of the grid):

- Adjust tree density and observe when the fire reaches the right edge.
- Record the **percent burned** at densities above, below, and near the threshold.
- Explain why the fire behaves differently at these densities.

In modeling, this **epidemic threshold** is more accurately referred to as a **phase transition**—a point where a small change in an input parameter causes a dramatic shift in the system's behavior, leading to the collapse of a previously stable state.

For this simulation, there is a closed-form solution: with a tree density of 57%, there is nearly a 0% chance of the fire reaching the right side of the grid. However, as density increases to 62%, this probability jumps to nearly 100%.



3.2.3 The Code

Reference: NetLogo - 1997 Uri Wilensky.

Initial Setup

```
globals [
  initial-trees    ;; how many trees (green patches) we started with
  burned-trees     ;; how many have burned so far
]

breed [fires fire]    ;; bright red turtles -- the leading edge of the fire
breed [embers ember] ;; turtles gradually fading from red to near black

to setup
  clear-all
  set-default-shape turtles "square"    ;; turn turtles into square shapes
  ask patches with [(random-float 100) < density]    ;; make some green trees
  [ set pcolor green ]

  ask patches with [pxcor = min-pxcor]    ;; make a column of burning trees
  [ ignite ]

  set initial-trees count patches with [pcolor = green]    ;; set tree counts
  set burned-trees 0
  reset-ticks
end
```

Helper Commands

```
;; creates the fire turtles
to ignite ;; patch procedure
  sprout-fires 1
  [ set color red ]
  set pcolor black
  set burned-trees burned-trees + 1
end

;; achieve fading color effect for the fire as it burns
to fade-embers
  ask embers
  [ set color color - 0.3    ;; make red darker
    if color < red - 3.5    ;; are we almost at black?
    [ set pcolor color
      die ] ]    ;; Removes the turtle, only the black patch remains
end
```

Main Loop

```

to go
  if not any? turtles ;; either fires or embers
    [ stop ]
  ask fires
    [ ask neighbors4 with [pcolor = green]
      [ ignite ]
      set breed embers ]
  fade-embers
  tick
end

```

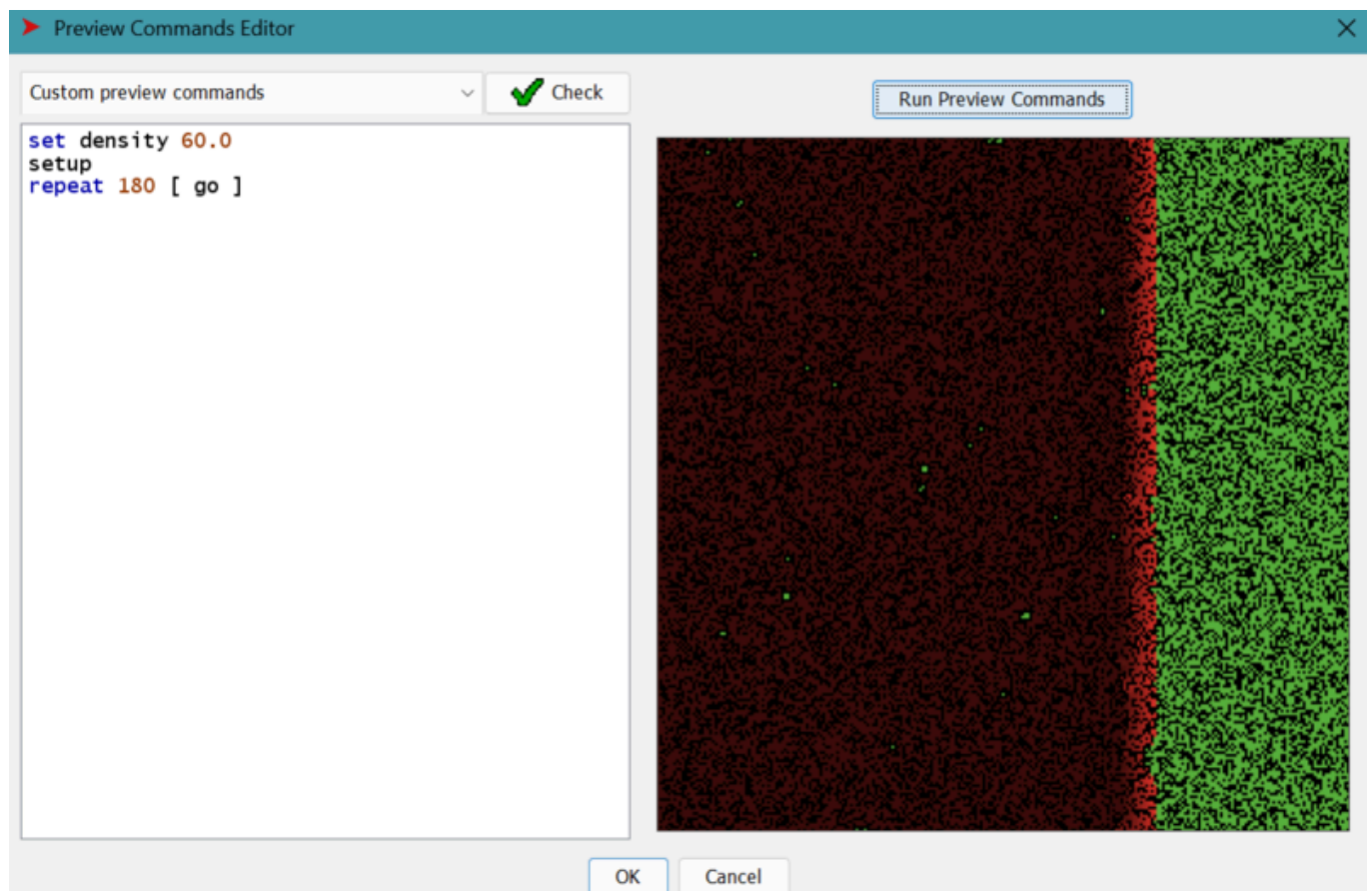
Exercise: Modify the code to allow the fire to spread to all 8 neighboring patches (not just the 4 directly adjacent ones).

Run the `setup` command in the Interface tab and press `go` to start the simulation.

- Observe how the fire spreads. Does it behave differently with the change?
- Does the forest burn down completely? If yes, why? If no, why not?
- Determine the new `density` for which a **phase transition** happens. How does it compare to the original density threshold?
- Do you think the critical density needs to be halved or adjusted differently? Explain why this might be the case.

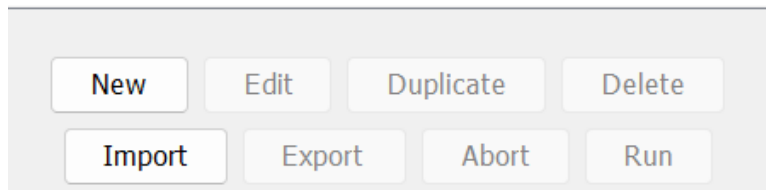
3.2.4 Preview Commands Editor

The **Preview Commands Editor** in NetLogo is a helpful tool that allows you to test your `setup` and `go` commands before running an experiment. It ensures that your commands are valid and that the model behaves as expected, helping to catch errors early and avoid wasting time on misconfigured experiments.



3.2.5 BehaviorSpace

BehaviorSpace in NetLogo is a tool that allows you to run and analyze parameterized experiments by automating the execution of your model with different variable combinations. The **Preview Commands Editor** is connected to **BehaviorSpace** because it allows you to validate and fine-tune the **setup** and **go** commands used in your experiment before running it.



Pressing the **New** button brings up the following window:

The image shows the "Experiment" editor window. It has a teal title bar with a red arrow icon and a close button. The main content area is light gray. At the top, there is a green message box that says "Welcome to the new BehaviorSpace experiment editor! We added some new features to this window. If you would like to learn more about them, you can hover over the labels or click the 'Help' button at the bottom of the window to read our updated documentation." Below this, there is a text field for "Experiment name" with the value "experiment". Underneath, there is a section titled "Vary variables as follows (note brackets and quotation marks):" with a text area containing ["density" 37]. Below that is a "Repetitions" field with the value "1". There is a checkbox labeled "Execute combinations in sequential order" which is checked. Below that is a section titled "Measure runs using these reporters as metrics:" with a text area containing "count turtles". There is another checkbox labeled "Run metrics every step" which is checked. Below that is a "Run metrics when" field. There is a section titled "Pre experiment commands:" with two sub-sections: "Setup commands:" containing "setup" and "Go commands:" containing "go". Below these are sections for "Stop condition:", "Post run commands:", and "Post experiment commands:". At the bottom, there is a "Time limit" field with the value "0". At the very bottom, there are three buttons: "OK", "Help", and "Cancel".

- **Experiment Name:** Assigns a name to the experiment for identification.
- **Vary Variables as Follows:** Specifies which variables will change during the experiment and their values.
`["density" 30 40 50 60 70]`
- **Repetitions:** Defines how many times the experiment will repeat for each parameter combination. Set it to 100 to average the results over 100 runs at each density value.
- **Execute Combinations in Sequential Order:** Ensures that variable combinations are tested in the order specified.
- **Measure Runs Using These Reporters as Metrics:** Specifies the values to record during the experiment.
`count fires`
- **Run Metrics Every Step:** Records the specified metrics at every step of the simulation.
- **Pre Experiment Commands:** Commands to initialize the simulation before each run.
- **Stop Condition:** Defines when the simulation should stop.
- **Time Limit:** Specifies a maximum runtime for the simulation in seconds (0 means no limit).

Each experiment in NetLogo can be exported as an **.xml file**, allowing it to be reused or shared later to ensure reproducibility.

NetLogo also efficiently bundles all components—**interface elements (in XML)**, the **Info tab**, and the **Code tab**—into a single **.nlogo file** for easy management and distribution.

✓ 3.2.6 The Info Tab

NetLogo includes a built-in **documentation tool** that helps users effectively document their models. This tool's structure can also serve as a skeleton for documenting other projects. It typically follows the structure below:

- **What is it?:** A brief overview of the model and its purpose.
- **How it works?:** Explains the underlying mechanics, rules, and logic driving the model's behavior.
- **How to use it?:** Instructions for running the model, including details on controls, sliders, buttons, and other interface elements.
- **Things to Notice:** Key behaviors or patterns to observe when running the model.
- **Things to Try:** Suggestions for experimenting with the model, such as changing parameters or testing specific scenarios.
- **Extending the Model:** Ideas for adding new features or expanding the model's functionality.
- **NetLogo Features:** Highlights specific NetLogo commands or tools used in the model.
- **Related Models:** Lists similar models in the NetLogo library or other related projects.
- **Credits and References:** Acknowledgments for contributors and references to materials that inspired or informed the model.
- **How to Cite:** Citation format for the model, useful for academic or research purposes.
- **Copyright and License:** Details on the model's licensing terms and copyright information.

Utilizes Markdown syntax, just like this document.

Exercise: Create a reporter called `percent-burned` that calculates and reports the percentage of burned trees in the simulation.

Modify the code to use `neighbors` instead of `neighbors4`, allowing the fire to spread to all 8 neighboring patches.

- Set up a parameterized experiment in BehaviorSpace with the following conditions:
 - `density` set to 37%.
 - 100 repetitions of the experiment.

- Metrics logged at every simulation step.
- Analyze the experiment results at step 25:
 - Calculate the mean and standard deviation of the percent-burned values across all runs.
 - Determine if any of the runs reached 90% burned trees.

Solution:

▼ Click to show/hide solution

```
to-report percent-burned
  report (burned-trees / initial-trees) * 100
end
```

Exercise: Update the model so that fires start from all edges of the grid lattice instead of just one side.

Run the simulation and observe how the fire spreads:

- Determine the density percentage at which the phase transition occurs when fire spreads from all directions.
- Compare the results to the original setup where fire started from one edge.
- Is the fire approximately four times as effective when starting from all edges? Explain your observations.

Solution:

▼ Click to show/hide solution

```
ask patches with [(pxcor = min-pxcor) or (pxcor = max-pxcor) or (pycor = min-pycor)
  [ ignite ]
```

Exercise: Extend the functionality of the model by incorporating the effect of wind on fire spread.

This means that fire can spread not only to immediate neighbors but also to the neighbors of those neighbors.

- Modify the fire-spreading logic to include wind as a factor influencing the spread of fire.
- Determine the **phase transition** density for fire spread under wind influence.

Solution:

▼ Click to show/hide solution

```
to go
  if not any? turtles ;; either fires or embers
    [ stop ]
  ask fires
  [ ;; Spread to all patches within a radius of 2
    ask patches in-radius 2 with [pcolor = green]
      [ ignite ]
    set breed embers
  ]
  fade-embers
```

```
    tick
  end
```

Exercise: Extend the functionality of the model by adding tree regrowth at each step.

Modify the code so that trees can regrow with a certain percentage per step, controlled by a **slider** (e.g., `regrowth-rate`).

- Add a slider to the interface to control the regrowth percentage.
- Update the `go` procedure to allow patches (previously burned or empty) to regrow into trees based on the slider value.
- Run your experiments again and observe how regrowth affects fire spread dynamics.
- Determine the new phase transition density with tree regrowth enabled.

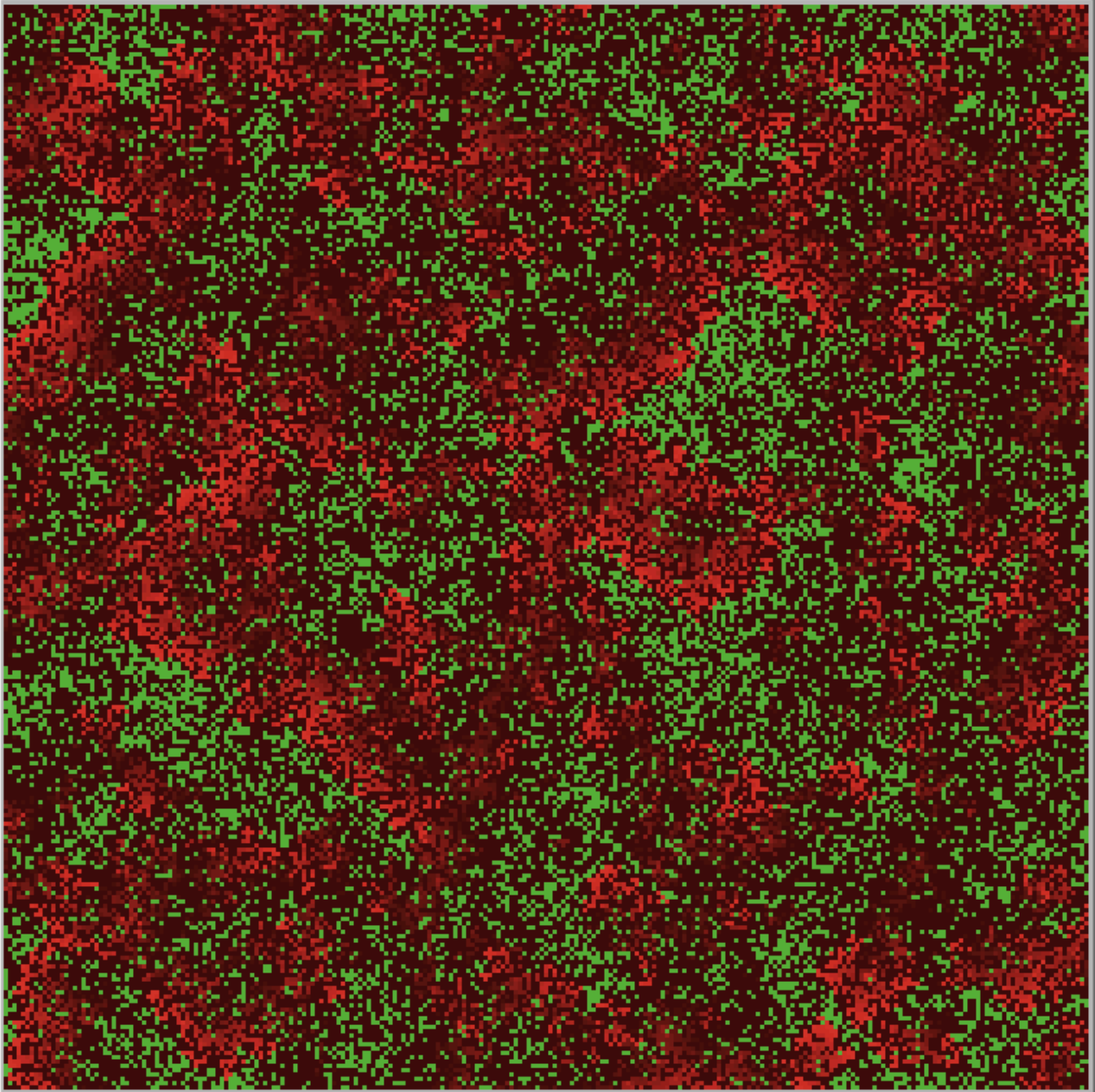
Solution:

▼ Click to show/hide solution

```
to go
  if not any? turtles ;; either fires or embers
    [ stop ]
  ask fires
  [ ;; Spread to all patches within a radius of 2
    ask patches in-radius 2 with [pcolor = green]
      [ ignite ]
    set breed embers
  ]

  regrow-trees
  fade-embers
  tick
end

to regrow-trees
  ask patches with [pcolor < red - 3.5]
  [
    if random-float 100 < regrowth
      [ set pcolor green ] ;; Regrow a tree
  ]
end
```



Exercise: Update the model to calculate the percentage burned based on the number of burned patches currently on the map, rather than using the initial number of trees (since trees can now regrow).

- Modify the `percent-burned` reporter to dynamically calculate the percentage of burned patches at any given moment.
- Ensure the calculation accounts only for patches with `pcolor = black`.

Solution:

▼ Click to show/hide solution

```
to-report percent-burned
  report ((count patches with [pcolor < red - 3.5] / (251 * 251)) * 100)
end
```

Extra Exercise: Create and analyze a parameterized simulation using the following steps:

1. Set up a simulation in BehaviorSpace with:

- density values ranging from 25% to 30% (in increments of 1%).
 - regrowth-rate values ranging from 2% to 5% (in increments of 1%).
 - 10 repetitions for each combination of parameters.
 - A time limit of 10 seconds per simulation.
2. Use the newly created percent-burned reporter to record the percentage of burned trees at the last step of each simulation.
 3. Export the simulation results to a CSV file.
 4. In Python:
 - Calculate the mean of the percent-burned values at the last step for each parameter combination.
 - Plot the results in 4 separate histograms, one for each regrowth-rate value (2%, 3%, 4%, and 5%).
 - Each histogram should display the percentage burned for density values from 25% to 30% (6 bars).

✓ 3.3 Schelling's Segregation Model

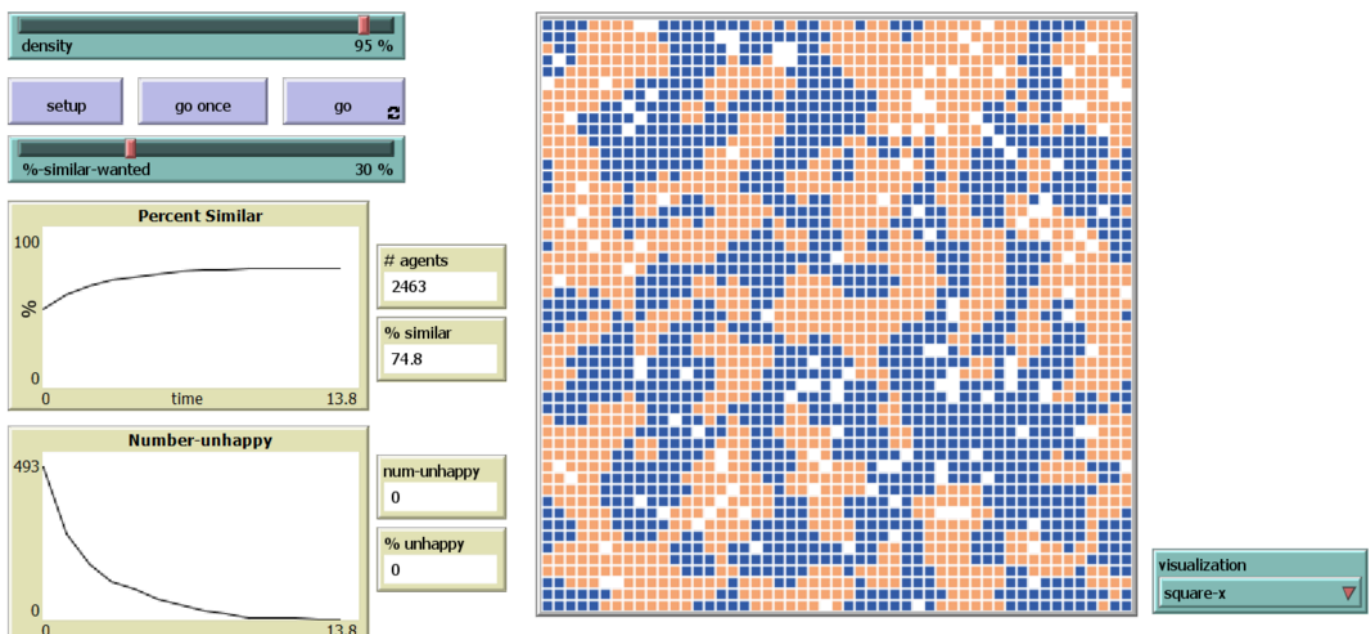
File > Models Library > Sample Models > Social Science > Segregation

Schelling's model of segregation has been regarded as one of the first agent-based models to address a significant social issue. It was created by Thomas Schelling, an economist who won the Nobel Prize for his contributions to economics during the Cold War. The segregation model naturally focuses on the social issue of segregation. He published it in 1972 and was originally called **Schelling's Tipping Model**.

In Schelling's original concept, the model represented a traditional **American urban landscape**, primarily inhabited by Black and White Americans. Racism and its resulting segregation were significant societal issues, and Schelling aimed to understand the mechanisms behind how racism influenced the urban landscape.

The basic idea was to create two types of agents, each with a preference for the composition of their neighboring agents expressed as a percentage. For example, a similar-wanted value of 30% means an agent will be considered happy if at least 30% of its neighbors are of the same type. This concept of happiness is crucial, as the goal of the model was to explore the tolerance level at which segregation does not occur while ensuring that all agents remain happy.

In this model, if an agent is unhappy, it attempts to move to a random empty location.



The density setup go and go once interface elements should be familiar by now. What they do here is exactly the same as what they did at the **Fire Model**.

There are four monitors with different reporters in the model:

- One reports the **total number of agents**, which remains constant and reflects the density value.
- The **percent-similar** monitor calculates the average percentage of an agent's neighbors that are the same color as the agent.
- The **num-unhappy** monitor reports the total number of unhappy agents.
- The **percent-unhappy** monitor reports the percentage of unhappy agents on the grid.



Fun Facts:

1. In 1972, Schelling did not have access to computers capable of performing complex calculations in hours. Instead, he used a checkerboard with pennies and dimes, manually moving them around to observe the effects of his model.
2. Schelling described segregation as a "macro behavior" resulting from "micro motives." He later wrote a book titled *Micromotives and Macrobehavior*.

3.3.1 Plot Interface Element

The plot interface element in NetLogo allows us to visualize how a global variable changes over time, providing insights into how the system responds to small changes in the environment.

The screenshot shows the 'Plot' window in NetLogo. The window has a title bar with a red arrow icon and a close button. Inside, the 'Name' field is set to 'Percent Similar'. The 'X axis label' is 'time', 'X min' is '0', and 'X max' is '5'. The 'Y axis label' is '%', 'Y min' is '0', and 'Y max' is '100'. There are checkboxes for 'Auto scale?' (checked) and 'Show legend?' (unchecked). Below these are expandable sections for 'Plot setup commands' and 'Plot update commands'. A table lists the pen settings:

Color	Pen name	Pen update commands	
Black	percent	plot percent-similar	 

At the bottom, there is an 'Add Pen' button and a row of four buttons: 'OK', 'Apply', 'Help', and 'Cancel'.

- **Plot Name:** Specifies the name of the plot, which will be displayed in the interface.
- **X-Axis Settings:** Allows you to label the x-axis and set its minimum and maximum values.
- **Y-Axis Settings:** Allows you to label the y-axis and set its minimum and maximum values.
- **Auto-Scale:** Automatically adjusts the minimum and maximum values for better scaling during the simulation.
- **Legend:** Adds a legend to identify what each pen represents.

- **Setup Commands:** You can add commands to run before plotting begins, such as initializing variables or preparing the environment.
- **Pen Options:**
 - **Multiple Pens:** Track multiple variables on the same plot using different pens with different colors.
 - **Pen Names:** You can assign a unique name to each pen for clarity.
 - **Plot Types:** Pens support bar plots, point plots, and line plots. In your example, a line plot is used.
 - **Plotting Intervals:** You can specify how frequently the pen updates, controlling the time intervals for plotting.
 - **Basic Pen Updates:** A typical pen update looks like the following: `plot <reporter>`

Exercise: Explore the tipping points in the model with a density of **95%**.

- Find the tolerance level where the number of unhappy agents reaches 0, and no segregation occurs.
- Identify the tolerance level where more than 90% of agents remain unhappy, and the simulation does not converge (does not end).

While studying population dynamics of two groups of equal size, Schelling found a threshold (B_{seg}) such that:

- ($B_a < B_{\text{seg}}$) leads to a random population configuration.
- ($B_a \geq B_{\text{seg}}$) leads to a segregated population.

The value of (B_{seg}) was approximately ($\frac{1}{3}$).

3.3.2 The Code

Reference: NetLogo - 1997 Uri Wilensky.

So what do the X marks on the grid represent when `setup` is run?

The X marks indicate agents who are currently unhappy with their living situation. These agents will move to a new location in the next simulation step. Agents that are not marked with X are satisfied and will remain in their current position.

Global Variable Definitions

```
globals [
  percent-similar ; on the average, what percent of a turtle's neighbors
                  ; are the same color as that turtle?
  percent-unhappy ; what percent of the turtles are unhappy?
]
```

Turtle Properties (each turtle will have these)

```
turtles-own [
  happy? ; indicates whether at least %-similar-wanted percent of
        ; that turtle's neighbors are the same color as the turtle
  similar-nearby ; how many neighboring patches have a turtle with my color?
  other-nearby ; how many have a turtle of another color?
  total-nearby ; sum of previous two variables
]
```

Setup

Breeds are **not used in this model** because agents are only created during the setup phase, their color (representing race) does not change, and their movement is not influenced by their color. This simplifies the implementation.

```
to setup
  clear-all
  ; create turtles on random patches.
  ask patches [
    set pcolor white
    if random 100 < density [ ; set the occupancy density
      sprout 1 [
        ; 105 is the color number for "blue"
        ; 27 is the color number for "orange"
        set color one-of [105 27]
        set size 1
      ]
    ]
  ]
  update-turtles
  update-globals
  reset-ticks
end
```

Helper Functions

The `[color]` of `myself` is used to explicitly reference the `color` of the turtle executing the command (the caller), distinguishing it from the `color` of the agents in the `agentset` (`turtles-on neighbors`).

For visualizations, there are two `if` statements controlling different representations. In the case of the square visualization, there is an `ifelse` within the `true` branch of the first `if`. If the agent is happy, it is displayed as a square; if not, it is displayed as an X shape.

```
to update-turtles
  ask turtles [
    ; in next two lines, we use "neighbors" to test the eight patches
    ; surrounding the current patch
    set similar-nearby count (turtles-on neighbors) with [ color = [ color ] of myself ]
    set other-nearby count (turtles-on neighbors) with [ color != [ color ] of myself ]
    set total-nearby similar-nearby + other-nearby
    set happy? similar-nearby >= (%similar-wanted * total-nearby / 100)
    ; add visualization here
    if visualization = "old" [ set shape "default" set size 1.3 ]
    if visualization = "square-x" [
      ifelse happy? [ set shape "square" ] [ set shape "X" ]
    ]
  ]
end
```

Here, two local variables are defined. These should not be confused with agent variables, as they are not updated here but are only used within the `sum` command. Their role is determined by the calling order in the `setup` command.

```

to update-globals
  let similar-neighbors sum [ similar-nearby ] of turtles
  let total-neighbors sum [ total-nearby ] of turtles
  set percent-similar (similar-neighbors / total-neighbors) * 100
  set percent-unhappy (count turtles with [ not happy? ]) / (count turtles) * 100
end

```

Main Loop

If all turtles are happy, stop the simulation; otherwise, move the unhappy turtles and update both agent and global variables.

```

to go
  if all? turtles [ happy? ] [ stop ]
  move-unhappy-turtles
  update-turtles
  update-globals
  tick
end

```

```

to move-unhappy-turtles
  ask turtles with [ not happy? ]
    [ find-new-spot ]
end

```

The `find-new-spot` command is applied to all unhappy turtles. It rotates the agent in a random direction (right turn) and moves it forward by a random distance between 0 and 10 patches. If the destination is already occupied by another agent, the turtle continues searching for a new spot on the grid.

This computation might seem inefficient. Let's create a new command that identifies empty patches on the grid and moves agents randomly to one of those patches.

```

to find-new-spot
  rt random-float 360
  fd random-float 10
  if any? other turtles-here [ find-new-spot ] ; keep going until we find an unoccupied patch
  move-to patch-here ; move to center of patch
end

```

Exercise: Create a new `find-new-spot` command to optimize agent movement.

- Instead of having the agent repetitively move to random spaces, first infer all empty patches on the grid where no turtles are present.
- Update the agent to move randomly to one of the inferred empty patches.
- How does this updated command perform compared to the original one?

Solution:

▼ Click to show/hide solution

```

to find-new-spot
  let empty-patches patches with [not any? turtles-here]

```

```

    if any? empty-patches [
      move-to one-of empty-patches
    ]
  end

```

3.3.3 Modifying the Code

Exercise: Add a **goodness** level to the agents in the simulation and modify their behavior based on this attribute.

1. For the `setup` stage, add two sliders:
 - `orange-good-percentage` : Controls the percentage of orange agents who are "good" (e.g., 80% means 80% of orange agents are good).
 - `blue-good-percentage` : Controls the percentage of blue agents who are "good".
2. Create a slider called `%-good-wanted` to control the percentage of good neighbors required for an agent to be happy.
3. Modify the model dynamics so that:
 - Agents prioritize having enough "good" neighbors based on `%-good-wanted`.
 - If the required number of good neighbors is not met, agents fall back to checking similarity in race (`%-similar-wanted`).
 - If neither condition is met, the agent becomes unhappy and moves.
4. Run the simulation with the following parameters:
 - Set `orange-good-percentage` and `blue-good-percentage` to 50%.
 - Test what happens when `%-good-wanted` is larger than `%-similar-wanted`.
5. Observe and analyze the results:
 - How does the newly added "good" attribute affect segregation?
 - How does it influence tipping dynamics when agents move?

Solution:

▼ Click to show/hide solution

Two new agent attributes are created: `good-neighbors` and `good?`

```

turtles-own [
  happy?           ; whether the turtle is happy
  similar-nearby   ; count of neighboring turtles of the same color
  other-nearby     ; count of neighboring turtles of different colors
  total-nearby     ; total number of neighbors
  good-neighbors   ; count of "good" neighbors
  good?           ; whether this turtle is "good"
]

```

The `setup` command is extended to handle the new dynamics introduced by `good?` attribute.

```

to setup
  clear-all
  ; create turtles on random patches
  ask patches [
    set pcolor white

```

```

if random 100 < density [   ; set the occupancy density
  sprout 1 [
    ; Assign color randomly
    set color one-of [105 27] ; 105 = blue, 27 = orange
    set size 1
    ; Assign good or bad status based on sliders
    ifelse color = 105 [   ; Blue agents
      set good? (random-float 100 < blue-good-percentage)
    ] [
      set good? (random-float 100 < orange-good-percentage)
    ]
  ]
]
update-turtles
update-globals
reset-ticks
end

```

The `update-turtles` command has been extended to prioritize good agents in the neighborhood. If the required number of good agents is not met, it falls back to the similarity-based threshold for happiness.

```

to update-turtles
  ask turtles [
    ;; Count good neighbors
    set good-neighbors count (turtles-on neighbors) with [good?]
    ;; Count neighbors by color
    set similar-nearby count (turtles-on neighbors) with [color = [color] of myself]
    set other-nearby count (turtles-on neighbors) with [color != [color] of myself]
    set total-nearby similar-nearby + other-nearby
    ;; Determine happiness
    ifelse good-neighbors >= (%-good-wanted * total-nearby / 100) [
      set happy? true ;
    ] [
      set happy? similar-nearby >= (%-similar-wanted * total-nearby / 100) ; Fall
    ]
    ;; Visualization
    if visualization = "old" [ set shape "default" set size 1.3 ]
    if visualization = "square-x" [
      ifelse happy? [ set shape "square" ] [ set shape "X" ]
    ]
  ]
end

```

Exercise: Can you think of any other visualization types that could be used here? Instead of showing the two types of agents, one could display one of their attributes to provide a different perspective.

Add at least two new visualizations!

Solution:

▼ Click to show/hide solution

Changing the appearance of the agents is not straightforward in this implementation. Extending the visualization to show agents in different colors based on their goodness level can completely disrupt how the environment dynamics work. This is because, in the `update-turtles` command, nearby similar and other agent counts are calculated based on the `color` attribute. If the `color` attribute is altered for visualization, it directly impacts the dynamics of the model.

To address this issue, one solution is to store the agent's original color in an `original-color` attribute during the `setup` phase. This ensures the model can always reference the original color for dynamics. Additionally, agents can have a separate `display-color` attribute that is used solely for visualization, allowing their appearance to change without affecting the underlying dynamics.

```
turtles-own [
  happy?          ; whether the turtle is happy
  similar-nearby  ; number of neighboring turtles of the same color
  other-nearby    ; number of neighboring turtles of different colors
  total-nearby    ; total number of neighbors
  good-neighbors  ; number of "good" neighbors
  good?          ; whether the turtle is "good"
  original-color  ; stores the turtle's initial color
  display-color   ; controls the visualization color
]

to setup
  clear-all
  ask patches [
    set pcolor white
    if random 100 < density [
      sprout 1 [
        set color one-of [105 27] ; 105 = blue, 27 = orange
        set original-color color ; Store the initial color
        set display-color color ; Initialize display-color to match the original
        set size 1
        ;; Assign good or bad status based on sliders
        ifelse color = 105 [ ; Blue agents
          set good? (random-float 100 < blue-good-percentage)
        ] [
          set good? (random-float 100 < orange-good-percentage)
        ]
      ]
    ]
  ]
  update-turtles
  update-globals
  reset-ticks
end
```

```
to update-turtles
  ask turtles [
    ;; Count good neighbors
    set good-neighbors count (turtles-on neighbors) with [good?]
  ]
end
```

```

;; Count neighbors by original color
set similar-nearby count (turtles-on neighbors) with [original-color = [original
set other-nearby count (turtles-on neighbors) with [original-color != [original
set total-nearby similar-nearby + other-nearby

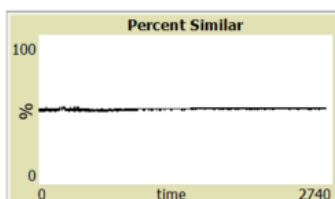
;; Determine happiness
ifelse good-neighbors >= (%good-wanted * total-nearby / 100) [
  set happy? true
] [
  set happy? similar-nearby >= (%similar-wanted * total-nearby / 100)
]

;; Visualization
if visualization = "old" [
  set shape "default"
  set display-color original-color
  set size 1.3
]
if visualization = "square-x" [
  ifelse happy? [ set shape "square" ] [ set shape "X" ]
  set display-color original-color
]
if visualization = "happy" [
  set shape "square"
  ifelse happy? [ set display-color green ] [ set display-color red ]
]

if visualization = "goodness" [
  set shape "square"
  ifelse good? [ set display-color pink ] [ set display-color black ]
]

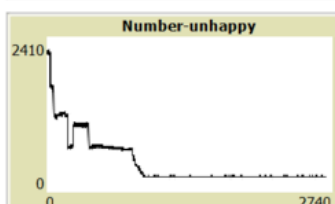
set color display-color
]
end

```



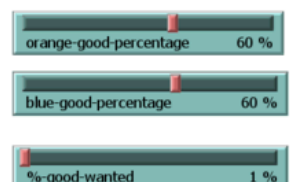
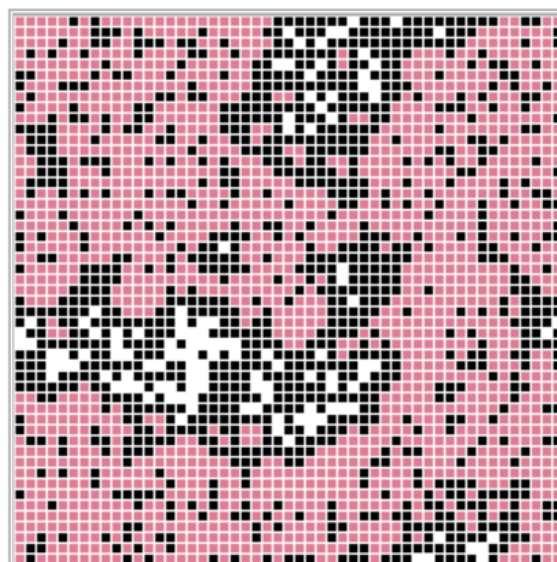
agents
2478

% similar
51.1



num-unhappy
234

% unhappy
9.4



Extra Exercise: Modify the code to include a controllable number of agent types in the environment, rather than just orange and blue.

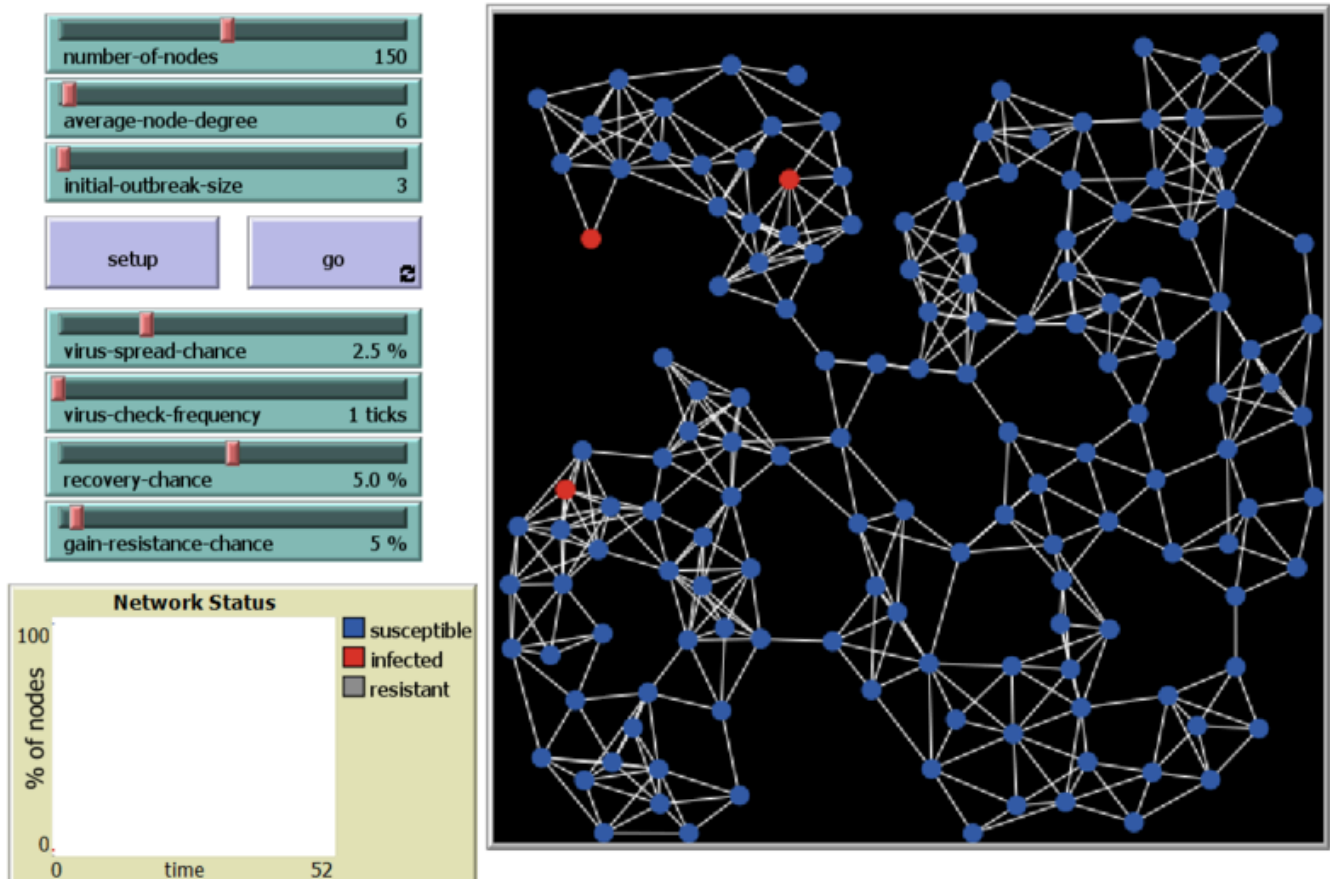
- Add a new slider to control the number of agent types.
- Ensure each agent type is assigned a unique color dynamically based on the number of types specified.
- Update the dynamics to account for similarity checks across all agent types.

✓ 3.4 Virus on a Network Model

File > Models Library > Sample Models > Networks > Virus on a Network

The **Virus on a Network** model was created by Uri Wilensky in 2008 to simulate the spread of a virus within a social network. The simulation begins with a network-building process, where parameters such as `number-of-nodes`, `average-node-degree`, and `initial-outbreak-size` define the structure of the network. This setup creates a network that closely resembles the structure of a social network. The model uses agents and **links** to represent nodes and their connections, respectively.

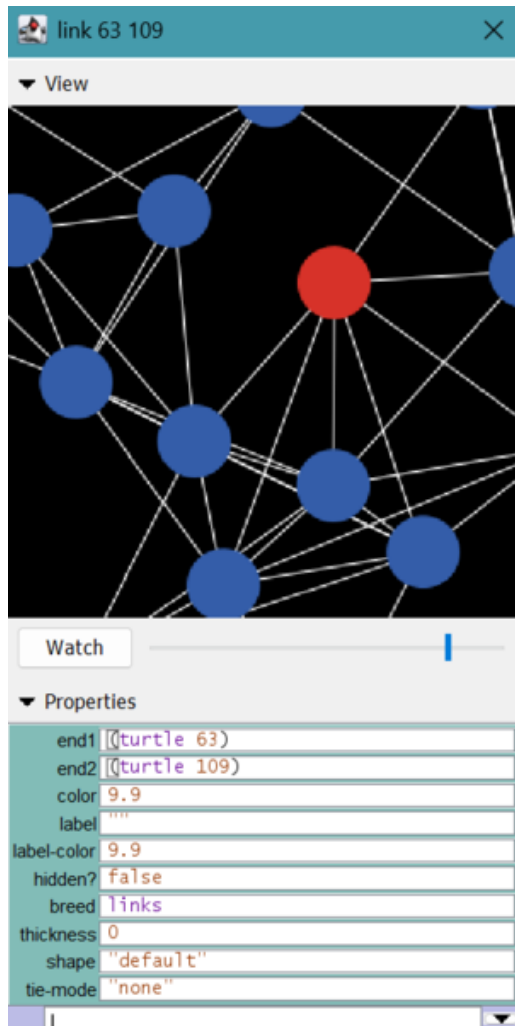
Remember: Links are also considered as **agents** in the NetLogo environment.



This graph does not follow a Rényi or Barabási graph but instead represents a **random geometric graph (RGG)**. In graph theory, a **random geometric graph** is the simplest type of spatial network. It is an undirected graph created by randomly placing N nodes in a metric space and connecting two nodes with a link if and only if their distance is within a specified range, such as smaller than a given neighborhood radius, r .

3.4.1 Links

Links are used to represent connections or relationships between agents. They are commonly used in models involving networks, such as social networks, transportation systems, or ecological relationships. Link also have properties and can be inspected the same way as **turtles** or **patches**.



Links have the following **unique** properties compared to turtles and patches:

- **end1** and **end2** : Represent the two agents connected by the link, specifying the source and target nodes.
- **hidden?** : Indicates whether the link is hidden from view, similar to how it is used in the Fireflies simulation.
- **thickness** : Specifies the visual thickness of the link on the grid.
- **shape** : Defines the shape of the link. The default is a straight line, but other options are available (e.g., arrows for directed links).
- **tie-mode** : Tie connects two turtles so that the movement of one turtles affects the location and heading of another. When a link's **tie-mode** is set to "fixed" or "free" **end1** and **end2** are tied together. If the link is directed **end1** is the "root agent" and **end2** is the "leaf agent". That is when **end1** moves (using **fd**, **jump**, **setxy**, etc.) **end2** also moves the same distance and direction.

When a link between two nodes is **undirected**, **end1** is always the older node, meaning the node with the lowest who ID. However, when the link is **directed**, **end1** represents the source node, and **end2** represents the target node.

Examples:

Create an undirected link between turtle 0 and turtle 1:

```
ask turtle 0 [ create-link-with turtle 1 ]
```

Create an directed link between turtle 0 and turtle 1:

```
ask turtle 0 [ create-link-to turtle 1 ]
```

Query all links connected to turtle 0:

```
ask turtle 0 [ show link-neighbors ]
```

Change link properties (the same as with turtles or patches):

```
ask links [ set color red set thickness 0.5 ]
```

Note: Once the first link has been created, directed or undirected, all unbreeded links must match (links also support breeds, much like turtles).

3.4.2 Network Building

Let's see how it is done in the code:

```
turtles-own
[
  infected?          ;; if true, the turtle is infectious
  resistant?        ;; if true, the turtle can't be infected
  virus-check-timer  ;; number of ticks since this turtle's last virus-check
]

to setup
  clear-all
  setup-nodes
  setup-spatially-clustered-network
  ask n-of initial-outbreak-size turtles
    [ become-infected ]
  ask links [ set color white ]
  reset-ticks
end
```

The `setup` command, when called, first creates all the nodes in the network based on the `number-of-nodes` parameter. It then creates links between these nodes according to the RGG process. After that, it randomly selects `initial-outbreak-size` turtles to become infected. Finally, it sets the color of all links to white.

Node Setup:

```
to setup-nodes
  set-default-shape turtles "circle"
  create-turtles number-of-nodes
  [
    ; for visual reasons, we don't put any nodes *too* close to the edges
    setxy (random-xcor * 0.95) (random-ycor * 0.95)
    become-susceptible
    set virus-check-timer random virus-check-frequency
  ]
end
```

At the start of the simulations are agents become susceptible (not infected nor resistant to the virus). This also sets their color blue. Then their virus check timer becomes a number between 0 (inclusive) and `virus-check-frequency - 1` (exclusive)

```
to become-susceptible ;; turtle procedure
  set infected? false
  set resistant? false
  set color blue
end
```

The `num-links` is the number of links needed to be created so that the network will have a proper `average-node-degree`. We know on average how many links will be connected to a node, the number of nodes, and also that the network is undirected, so the calculation is divided by 2.

While the global link count remains below the required threshold, an agent is selected from a filtered agentset. This agentset consists of all other turtles that are not part of the calling agent's link-neighborhood. From this set, the closest agent is chosen based on `distance myself`. If a valid choice is made (i.e., the selected agent is not empty), an undirected link is created between the two agents.

The second line is purely for visualization purposes, using spring and force-based physics to adjust the positions of the nodes.

```
to setup-spatially-clustered-network
  let num-links (average-node-degree * number-of-nodes) / 2
  while [count links < num-links ]
  [
    ask one-of turtles
    [
      let choice (min-one-of (other turtles with [not link-neighbor? myself])
        [distance myself])
      if choice != nobody [ create-link-with choice ]
    ]
  ]
  ; make the network look a little prettier
  repeat 10
  [
    layout-spring turtles links 0.3 (world-width / (sqrt number-of-nodes)) 1
  ]
end
```

✓ 3.4.3 The Main Loop

If all turtles are either resistant or susceptible, the simulation stops. Otherwise, each agent increments its `virus-check-timer` by 1. When the timer reaches its threshold, it resets. The main loop concludes with the virus spreading (handled in a separate command) and the agents performing a virus check.

```
to go
  if all? turtles [not infected?]
  [ stop ]
  ask turtles
  [
    set virus-check-timer virus-check-timer + 1
  ]
end
```

```

        if virus-check-timer >= virus-check-frequency
            [ set virus-check-timer 0 ]
        ]
    spread-virus
do-virus-checks
tick
end

```

This command asks the infected agentset to check all their non-resistant neighbors and attempt to infect them based on the `virus-spread-chance`. If successful, the neighboring agents become infected, their color changes to red, and their `infected?` variable is set to `true`.

```

to become-resistant ;; turtle procedure
    set infected? false
    set resistant? true
    set color gray
    ask my-links [ set color gray - 2 ]
end

```

```

to become-infected ;; turtle procedure
    set infected? true
    set resistant? false
    set color red
end

```

```

to spread-virus
    ask turtles with [infected?]
        [ ask link-neighbors with [not resistant?]
            [ if random-float 100 < virus-spread-chance
                [ become-infected ] ] ]
end

```

The virus check asks all turtles that are infected and due for a check to attempt recovery. If recovery is successful, the agent has a chance to gain resistance based on the `gain-resistance-chance`. If successful, the agent becomes `resistant`; otherwise, it becomes `susceptible`. If recovery is unsuccessful, the agent remains `infected`.

```

to do-virus-checks
    ask turtles with [infected? and virus-check-timer = 0]
        [
            if random 100 < recovery-chance
            [
                ifelse random 100 < gain-resistance-chance
                    [ become-resistant ]
                    [ become-susceptible ]
            ]
        ]
end

```

Exercise: Experiment with the variables that influence the dynamics of the virus:

- Adjust the following hyperparameters in both directions:

- - virus-spread-chance
 - virus-check-frequency
 - recovery-chance
 - gain-resistance-chance
- Observe and record the effects of these changes on the simulation dynamics.
- Analyze which hyperparameter the model is most sensitive to and explain why this might be the case.

Exercise: Increase the `number-of-nodes` and the `average-node-degree` hyperparameter.

- Observe how the changes affect the dynamics of the virus spread.
- Compare the susceptible/resistant ratio to the previous configuration.

Exercise: Set the `gain-resistance-chance` to 0%.

- Observe whether the model can recover from the virus without agents gaining resistance.
- With `virus-spread-chance` set at 2.5%, identify the tipping point where the virus can no longer spread effectively.

Rényi Solution:

▼ Click to show/hide solution

```
to setup-renyi-network
  let num-links (average-node-degree * number-of-nodes) / 2
  while [count links < num-links]
  [
    ask one-of turtles
    [
      let choice one-of other turtles with [not link-neighbor? myself]
      if choice != nobody [ create-link-with choice ]
    ]
  ]
  ; arrange nodes for better visualization
  layout-spring turtles links 0.3 (world-width / (sqrt number-of-nodes)) 1
end
```

Barabási Solution:

▼ Click to show/hide solution

```
to setup-barabasi-network
  clear-all
  set-default-shape turtles "circle"

  ;; Initialize with two connected nodes
  create-turtles 2 [
    setxy random-xcor random-ycor
    become-susceptible
    set virus-check-timer random virus-check-frequency
  ]
  ask turtle 0 [ create-link-with turtle 1 ]

  while [count turtles < number-of-nodes] [
```

```

create-turtles 1 [
  setxy random-xcor random-ycor
  become-susceptible
  set virus-check-timer random virus-check-frequency
]

;; new-turtle = turtle with no link
let new-turtle one-of turtles with [not any? my-links]
;; calculates total-degree of all nodes (at start its 2)
let total-degree sum [count my-links] of turtles
;; generate a random value for preferential attachment
let r random-float total-degree
let cumulative-sum 0
let selected-node nobody

ask turtles [
  set cumulative-sum cumulative-sum + count my-links
  if (cumulative-sum >= r and selected-node = nobody) [
    set selected-node self
  ]
]

if selected-node != nobody [
  ask new-turtle [
    create-link-with selected-node
  ]
]

;; Optional: improve visualization
repeat 10 [
  layout-spring turtles links 0.3 (world-width / (sqrt number-of-nodes)) 1
]
end


```

✓ 4. Practice - NetLogo Model Design

 Tamás Takács, PhD student, Department of Artificial Intelligence

 90 min read

 January 22, 2025

 Collective Intelligence



ELTE | IK
INFORMATIKAI KAR



DEPARTMENT
OF ARTIFICIAL
INTELLIGENCE

NetLogo



This practice notebook presents a range of methods and techniques for developing agent-based models from the ground up. As a foundational example, we will construct a basic variation of Axtell's Economy model, incrementally applying core modeling concepts. The notebook also features selected large-scale ABM projects from the broader modeling community to illustrate the structure and standards of professional NetLogo models. By the end of this practice, students will be equipped with the skills necessary to design and implement complex agent-based models independently and will be prepared to complete the first assignment.

✓ Table of Contents

- **4.1 Simple Economy Model**
 - 4.1.1 The Rules
 - 4.1.2 First Step - setup
 - 4.1.3 Second Step - go
 - 4.1.4 The `transact` Command
 - 4.1.5 Key Characteristics of the Emergent Distribution
 - 4.1.6 Extending the Model
- **4.2 Flattening The Curve**
 - 4.2.1 Modeling Commons
 - 4.2.2 The Code
 - 4.2.3 The `go` Loop
 - 4.2.4 Traveling
 - 4.2.5 Infecting
 - 4.2.6 Closing Borders
 - 4.2.7 Not Living
 - 4.2.8 Cure
- **4.3 Game Development?**
 - 4.3.1 The Play Loop
 - 4.3.2 Movement

Last Practice

In our last practice, we covered:

- The Models Library and Benchmark Simulations
- The Fire model
- Extensions of the Fire model
- Phase Transitions and Tipping Points
- BehaviorSpace experiments
- The Info Tab
- Schelling's Segregation model
- Extensions of Schelling's Segregation model
- Plotting
- The Virus on a Network model
- Links
- Extensions of the Virus on a Network model

That's quite a lot again! Now, let's dive into how you can design and create your own NetLogo model completely from scratch.

✓ 4.1 Simple Economy Model

In 1996, Josh Epstein and Rob Axtell published one of the first definitive books on agent-based modeling and social science, *Growing Artificial Societies*, which featured artificial economic agents.

We will create a simple model of economic agents, inspired in part by Epstein and Axtell's work and a paper by Dragulescu and Yakovenko (2000).

Dragulescu, A., & Yakovenko, V. M. (2000). Statistical mechanics of money. The European Physical Journal B, 17(4), 723–729. doi:10.1007/s100510070114

4.1.1 The Rules

- 500 people start off with \$100 each (starting with a uniform distribution).
- At every tick, each person gives \$1 to another person randomly.
- If you run out of money, you can't give any more money away until someone gives you money.

Simple, right? Let's try to implement it in NetLogo!

4.1.2 First Step - setup

Designing the `setup` procedure is usually the first step, followed by the `go` procedure. These two components are not always required, but they are a style heavily utilized in NetLogo. Let's start with an empty project:

File > New

First, let's set our grid parameters before adding anything to the model. These can be edited to your liking; however, in this example, we will use the following parameters:

- **Location of Origin:** Corner, Bottom Left
- **max-pxcor:** 500
- **min-pycor:** 80
- **Patch size:** 1

- **Font size:** 10
- **Frame rate:** 30

One of the first things to add is a `setup` button next to the 2D environment grid. After adding the `setup` button, go to the **Code Tab** and create a basic skeleton for the `setup` function. A `setup` function should always include the `clear-all` and `reset-ticks` commands.

```
to setup
  clear-all
  ;; setup code
  reset-ticks
end
```

Looking at the first rule of the exercise, it requires us to create 500 agents and assign them \$100 as a starting point. This can be implemented using the `create-turtles` command and defining agent attributes. Additionally, we can set them to a circular shape with green color and size 2 for convenience.

```
turtles-own [ wealth ]

create-turtles 500 [
  set wealth 100
  set shape "circle"
  set color green
  set size 2
]
```

Running the `setup` creates the turtles, but we cannot see them because all turtles are created at the `(0, 0)` patch by default. Let's modify the code so that the agents spawn in specific locations.

Note: NetLogo will create all turtles at the `(0, 0)` patch if no location is specified in the command.

```
to setup
  clear-all
  create-turtles 500 [
    set wealth 100
    set shape "circle"
    set color green
    set size 5
    setxy wealth random-ycor
  ]
  reset-ticks
end
```

The goal is to visualize the wealth distribution. Rich agents should move to the right side of the grid, while poorer agents move to the left. Initially, all agents start in a random row with their x-coordinate (horizontal alignment) set to match their wealth.

4.1.3 Second Step - `go`

In the `go` procedure, turtles must transact their wealth if they have any. This can be done by calling a `transact` command (defined later). All `go` commands should end with the `tick` command.

```
ask turtles with [ wealth > 0 ] [ transact ]
```

This ensures only agents with wealth can give money to others. However, our grid size only allows a total wealth of \$500 for a single agent. To avoid errors, we limit the simulation with the following condition:

```
ask turtles [ if wealth <= max-pxcor [ set xcor wealth ]]
```

This ensures that agents with wealth exceeding 500 do not move further right on the grid. The final `go` procedure looks like this:

```
to go
  ask turtles with [ wealth > 0 ] [ transact ]
  ask turtles [ if wealth <= max-pxcor [ set xcor wealth ] ]
  tick
end
```

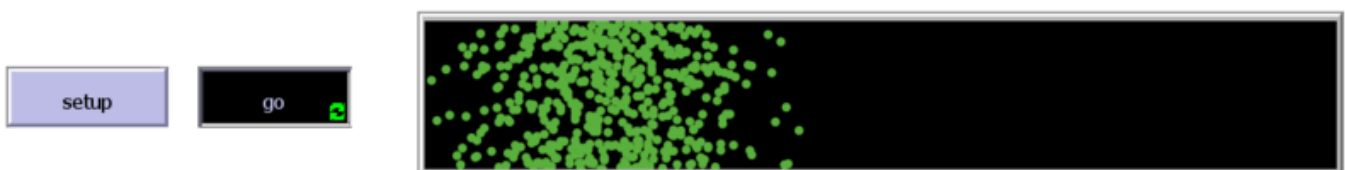
Don't forget to also add the `go` Button to the Interface Tab.

4.1.4 The `transact` Command

The `transact` command reduces one unit of wealth from the agent and gives it to a random agent. The `set` command is used to reduce wealth, while `ask one-of other turtles` is used to transfer wealth to a random recipient.

```
to transact
  set wealth wealth - 1
  ask one-of other turtles [ set wealth wealth + 1 ]
end
```

At this point, the simulation should work as intended. Agents will move horizontally across the grid, visualizing the wealth distribution as the simulation progresses.



Exercise: Run the `setup` command in the Interface tab and press the `go` button to start the simulation.

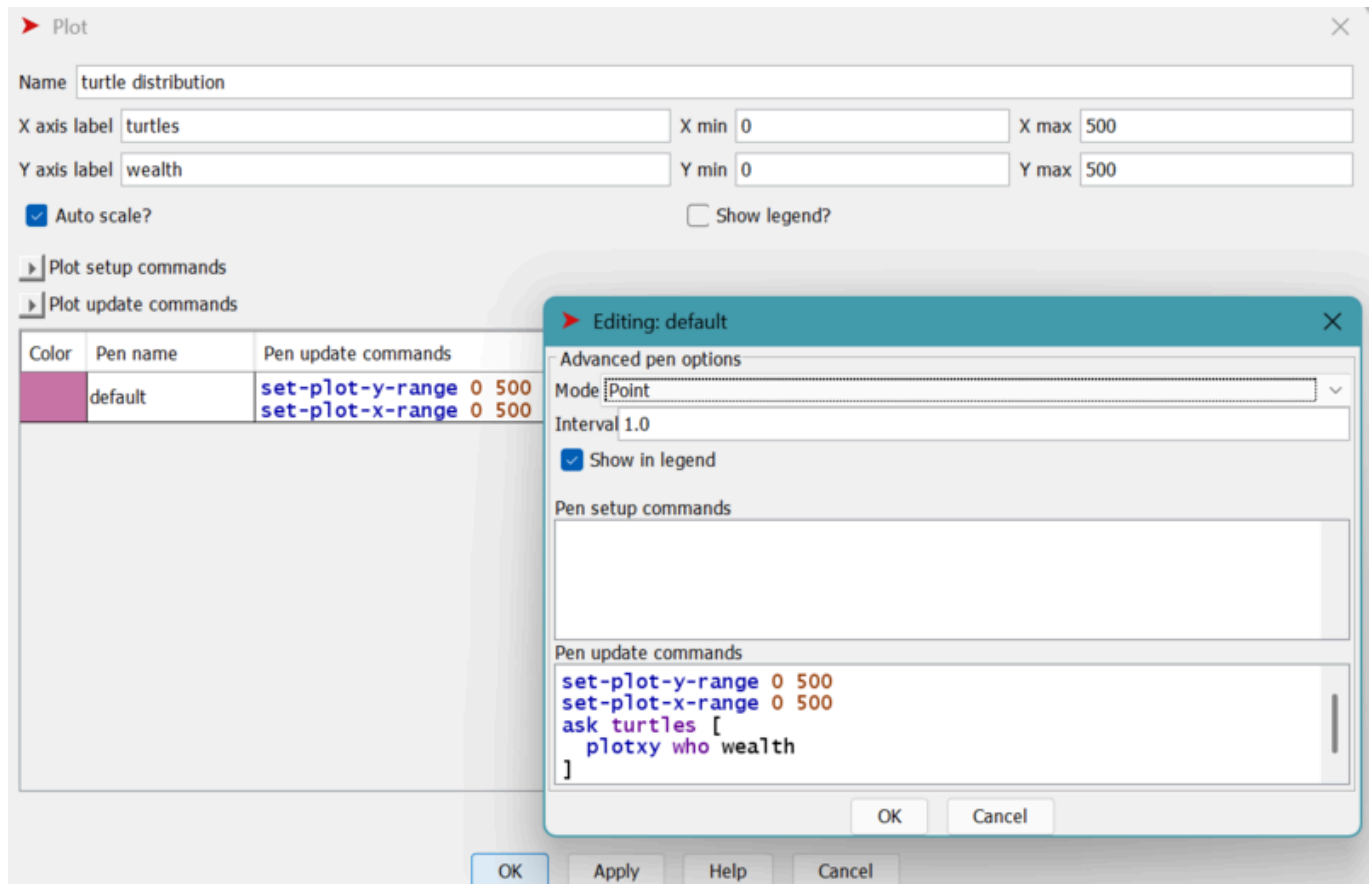
Analyze the wealth distribution in the system:

- Based on your observations, hypothesize the probability density function (PDF) that the wealth distribution might follow.
- What factors could contribute to the shape of the wealth distribution?
- Provide a formal justification for your guess before verifying with additional experiments.

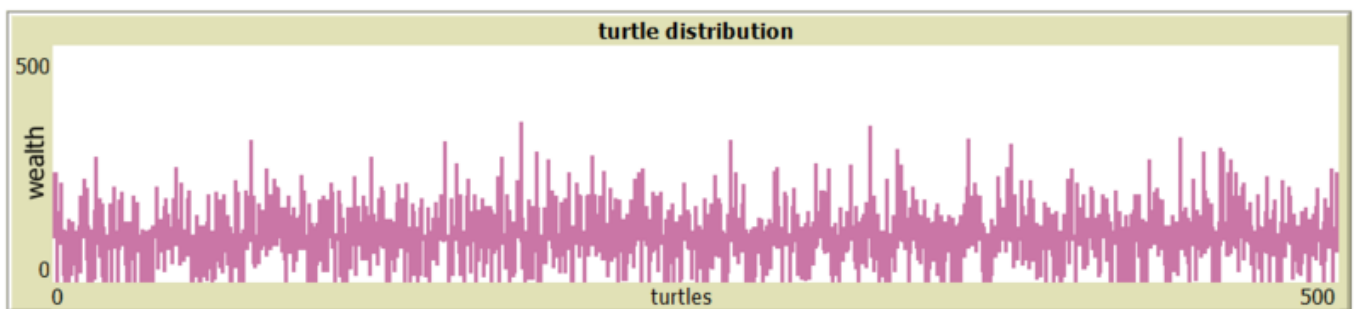
The simulation works, agents are moving around, and wealth is being redistributed. However, to better understand the inner dynamics of the model, we need additional insights through **plots** and **monitors** that provide valuable analytics.

Let's create a **point plot** that visualizes the wealth distribution. The **x-axis** will represent each agent's unique who ID, and the **y-axis** will represent their wealth. The plot will be named "**Turtle Distribution**", and we will also include a **legend** displaying the total wealth in the system, which remains constant at 50,000(100 * 500 agents).

To achieve this, we need to plot each agent's who ID against their respective wealth as points on the graph. This can be done easily using the following configuration:



We set the plot's maximum x and y ranges using the `set-plot-x-range` and `set-plot-y-range` commands. These define the bounds of the plot: the **x-axis** for the agents' IDs and the **y-axis** for their wealth. Then, we ask the turtles to plot their x and y coordinates, which correspond to their who ID and wealth, respectively. This ensures each agent's wealth is represented as a point on the plot.



Exercise: Observe the wealth distribution in the simulation. Over time, the dynamics of the simulation result in a noisy distribution of wealth among the agents. As the simulation progresses through many steps, the wealth starts to become increasingly unequal.

- What part of the simulation dynamics contributes to this phenomenon?
- Why does wealth inequality emerge despite the random nature of the transactions?

Exercise: Create another plot, but this time switch the roles of the x and y axes. In this plot, we aim to explore how different amounts of wealth are distributed among agents. Follow these steps:

- Use a **bar plot** with wealth as the x-axis and the number of turtles as the y-axis.

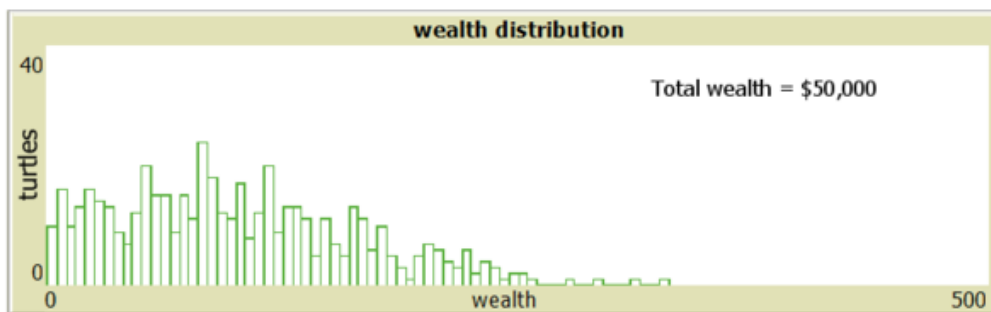
- Set the interval for the wealth bins to 5.
- Set the **maximum x value** to 500 (wealth) and the **maximum y value** to 40 turtles.
- Label the plot **wealth distribution** and add a legend indicating the total number of wealth (\$50,000).

Think about what this plot reveals. How does the wealth distribution change over time?

Solution:

▼ Click to show/hide solution

```
set-plot-y-range 0 40
set-plot-x-range 0 500
histogram [ wealth ] of turtles
```



For additional analytics, we want to calculate two key metrics:

1. **The wealth of the top 10% of agents** in the environment (to understand how much wealth is controlled by the richest agents).
2. **The total wealth of the bottom 50% of agents** (to assess the distribution of wealth among the poorer agents).

To achieve this, we can create two reporters in our NetLogo code: one to calculate the total wealth of the **top 10% of agents** and another to calculate the total wealth of the **bottom 50% of agents**.

```
to-report top-10-pct-wealth
  report sum [ wealth ] of max-n-of (count turtles * 0.10) turtles [ wealth ]
end
```

```
to-report bottom-50-pct-wealth
  report sum [ wealth ] of min-n-of (count turtles * 0.50) turtles [ wealth ]
end
```

The `max-n-of` reporter retrieves the **n turtles** with the **highest wealth values**, where **n** is 10% of the total number of turtles (in this case, `count turtles * 0.1`). Similarly, the `min-n-of` reporter retrieves the **n turtles** with the **lowest wealth values**, based on the `wealth` attribute.

To complete the task, add two respective monitor elements to the interface.

wealth of top 10%
11273

wealth of bottom 50%
11959

We can also plot these values on a **Plot Element** to observe how the wealth distribution changes over time. With the reporters for the **top 10% wealth** and **bottom 50% wealth** already created, these values can be dynamically added to a plot during the simulation.

Plot

Name: wealth by percent

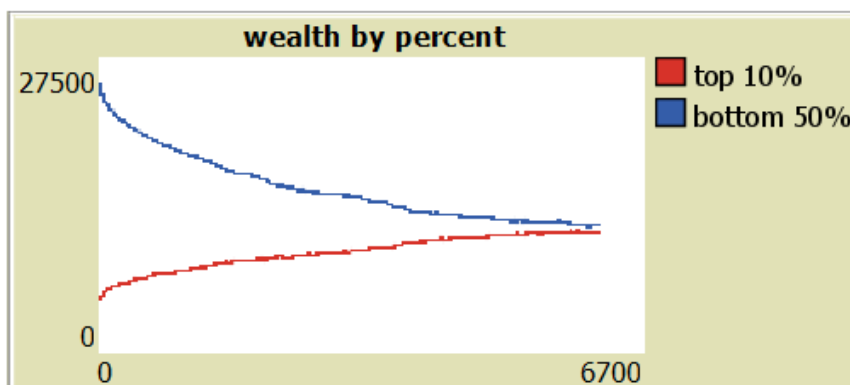
X axis label: X min: X max:

Y axis label: Y min: Y max:

☒ Auto scale? ☒ Show legend?

Color	Pen name	Pen update commands	
Red	top 10%	<code>plot top-10-pct-wealth</code>	<input type="button" value="Edit"/> <input type="button" value="Delete"/>
Blue	bottom 50%	<code>plot bottom-50-pct-wealth</code>	<input type="button" value="Edit"/> <input type="button" value="Delete"/>

Which results in something like this:



As the simulation progresses, the lines on the plot will likely cross each other. Over time, the **top 10% of agents** will control the majority of the wealth in the economy, while the **bottom 50% of agents** will control only around **20% of the total wealth**. This illustrates the emergence of wealth inequality in the system. Is this always the case?

Exercise: Using the BehaviorSpace experiment tool, create an experiment named `wealth-distribution` to measure the **top 10%** and **bottom 50%** wealth values in the system.

- Ensure the experiment collects these metrics only at the **end of the simulation**; intermediate steps are not needed.
- Set the experiment to run for a total of **10 repetitions** to account for variability.
- Limit the simulation to **10,000 ticks**.
- Analyze the final wealth distribution from the results.

Based on the dynamics of this wealth redistribution system, the wealth distribution tends to evolve into a **Boltzmann-Gibbs distribution**, which is also referred to as an **exponential distribution** in statistical mechanics.

4.1.5 Key Characteristics of the Emergent Distribution:

1. The probability of an agent having wealth (w) follows: $[P(w) = \frac{1}{\langle w \rangle} e^{-w/\langle w \rangle}]$ where $(\langle w \rangle)$ is the average wealth in the system.
2. The random exchange of wealth between agents leads to this exponential distribution, as it mirrors the energy exchange dynamics observed in gas molecules in statistical mechanics.

Exercise: Create a `Slider` element to control the number of agents in the simulation dynamically. Follow these steps:

- Add a global variable (e.g., `agent-count`) to store the number of agents.
- Configure the `Slider` in the Interface tab with appropriate minimum, maximum, and default values (e.g., min: 10, max: 500, default: 100).
- Ensure the `setup` procedure uses the value of `agent-count` to create the corresponding number of turtles.
- Update the plot configurations dynamically to reflect changes in the number of agents:
 - Set the x-axis range of the plot to `0` to `agent-count`.
 - Ensure the plots scale dynamically as the number of agents changes.

4.1.6 Extending the Model

Exercise: Modify the rules of the system to allow agents to go into debt. Specifically:

- Remove the restriction that prevents agents with `0` wealth from giving money.
- Allow agents to give money even if their wealth drops below `0`.

Observe the effect of this rule change on the wealth distribution over time:

- Does the system still follow an exponential distribution?
- What happens to the wealth dynamics as debt accumulates in the system?

Solution:

▼ Click to show/hide solution

```
to go
  ask turtles [ transact ]
  ask turtles [ if wealth <= max-pxcor [ set xcor wealth ] ]
  tick
end
```

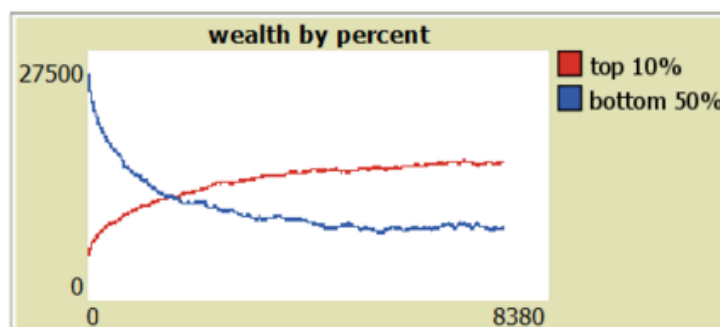
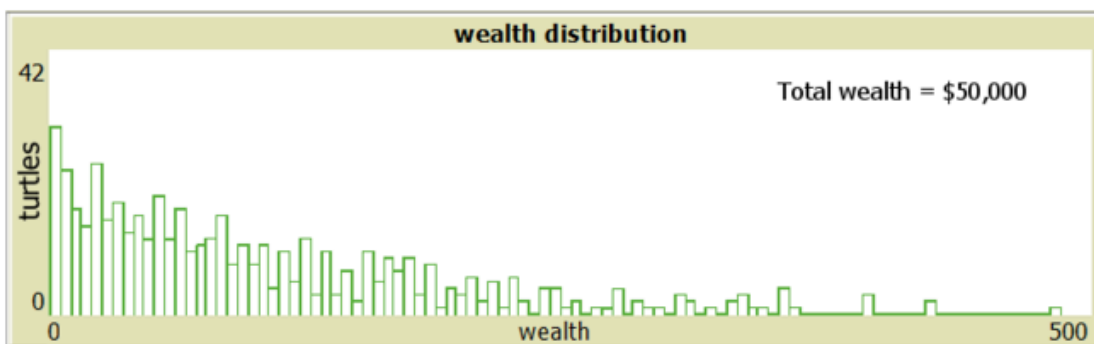
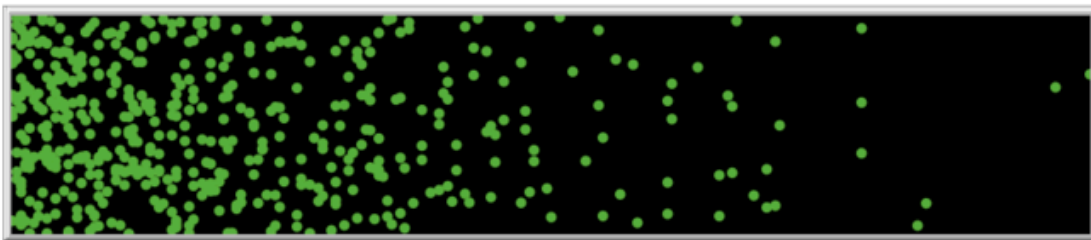
Exercise: Change the transaction rule so agents give out more money per transaction (e.g., from \$1 to \$5 or more).

- What happens to the wealth distribution?
- Does inequality increase or decrease?

Solution:

▼ Click to show/hide solution

```
to transact
;; give a dollar to another turtle
set wealth wealth - 2
ask one-of other turtles [ set wealth wealth + 2 ]
end
```



wealth of top 10%

15290

wealth of bottom 50%

8010

Exercise: Change the rules so that richer agents have a lower chance of receiving money, based on their wealth.

- Try different probability functions, such as $1 / \text{wealth}$ or $1 / \sqrt{\text{wealth}}$.
- Observe how this affects the wealth distribution and inequality.

Inverse Attained Wealth Probability:

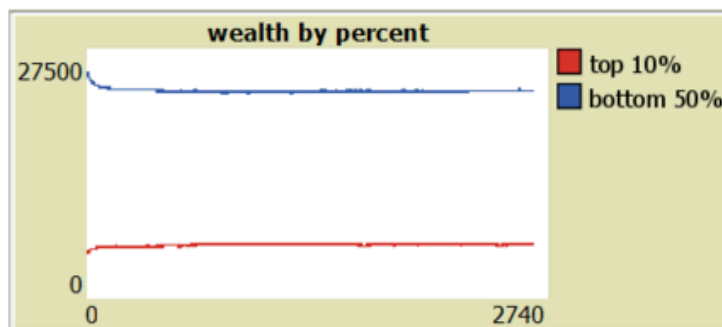
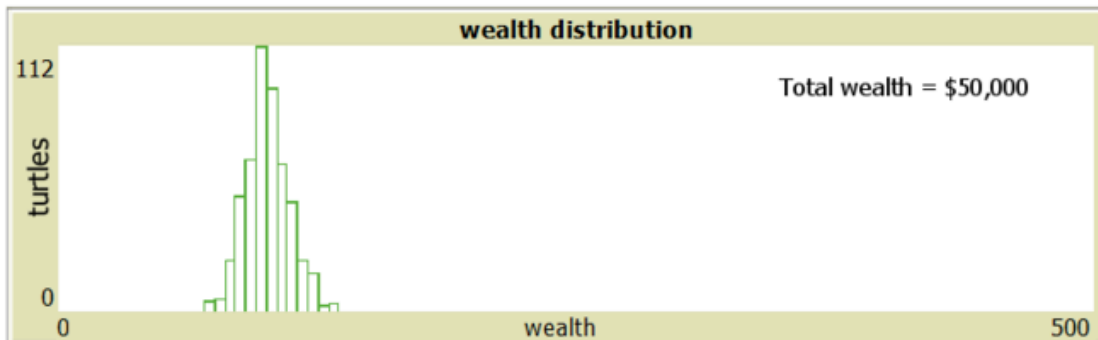
▼ Click to show/hide solution

```
to transact
set wealth wealth - 1
let recipient one-of other turtles with [
```

```

    random-float 1 < (1 / max (list sqrt wealth 1))
  ]
  if recipient != nobody [
    ask recipient [ set wealth wealth + 1 ]
  ]
end

```



wealth of top 10%
5963

wealth of bottom 50%
22998

Total Wealth Proportional Probability:

▼ Click to show/hide solution

```

to transact
  let total-wealth sum [ wealth ] of turtles
  let recipient one-of other turtles with [
    random-float 1 < (1 - (wealth / total-wealth))
  ]
  if recipient != nobody [
    set wealth wealth - 1
    ask recipient [ set wealth wealth + 1 ]
  ]
end

```

Exponential Decay Function:

▼ Click to show/hide solution

```

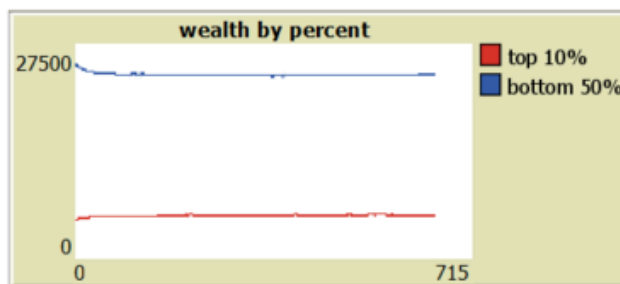
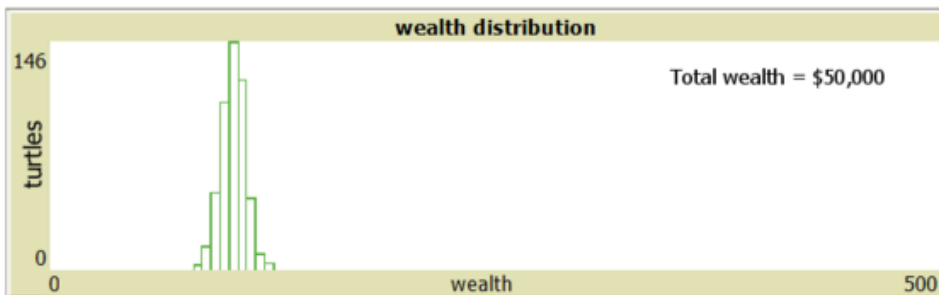
to transact
  let total-wealth sum [ wealth ] of turtles

```

```

let recipient one-of other turtles with [
  random-float 1 < exp(-0.1 * wealth) ;; Higher wealth = smaller probability
]
if recipient != nobody [
  set wealth wealth - 5
  ask recipient [ set wealth wealth + 5 ]
]
end

```



wealth of top 10%
5580

wealth of bottom 50%
23715

Exercise: Modify the giving rule so that wealthier agents give out more money based on the percentage of their wealth.

- Change the rule so that every agent gives out a fixed percentage (e.g., 5%) of their current wealth in each transaction.
- Experiment with different percentage values (e.g., 2%, 10%, or 20%).
- Observe how this rule affects the wealth distribution and inequality over time.

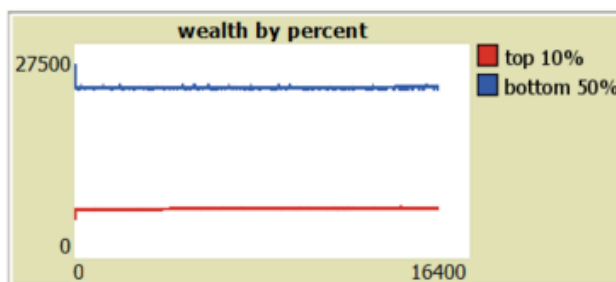
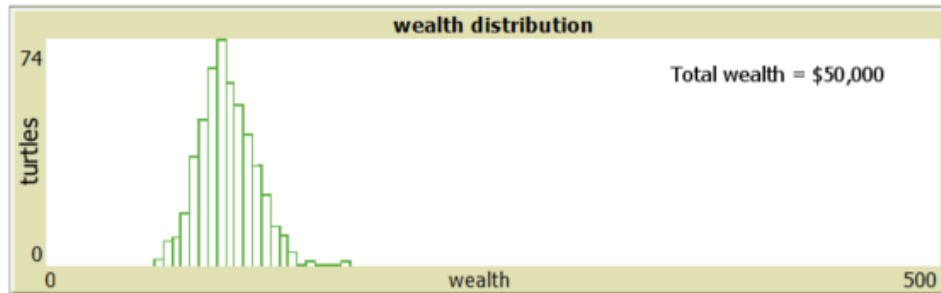
Solution:

▼ Click to show/hide solution

```

to transact
  let amount-to-give max (list floor (wealth * (transact-percentage / 100)) 1)
  set wealth wealth - amount-to-give
  ask one-of other turtles [
    set wealth wealth + amount-to-give
  ]
end

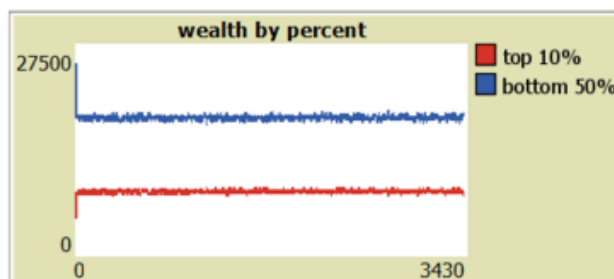
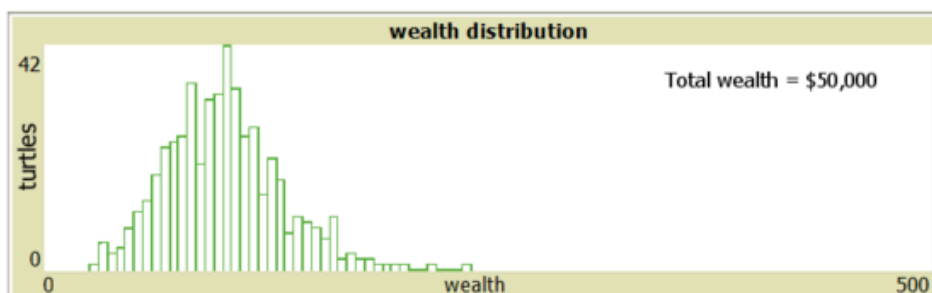
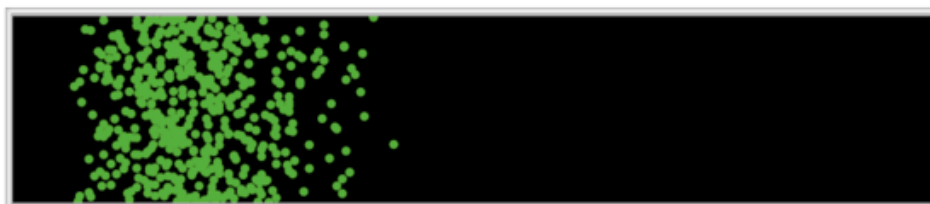
```



wealth of top 10%
6414

wealth of bottom 50%
22005

With 5%:



wealth of top 10%
8364

wealth of bottom 50%
18342

With 20%:

This percentage-based payment functions as a form of taxation, disproportionately impacting wealthier agents more than poorer ones. As the tax percentage decreases, the wealth distribution becomes more uniform, with agents converging toward similar living standards. Conversely, increasing the tax percentage flattens the wealth distribution, reducing inequality. However, if the tax rate becomes excessively high, the system reverts to an exponential **Boltzmann-Gibbs distribution**.

Fun Facts:

1. The original model of Epstein and Axtell was called the Sugarscape model, which can be found in the **Models Library**.
2. This redistribution phenomenon aligns with the broader concept in economics and sociology that, in **systems with random exchanges and without redistributive policies, wealth tends to become concentrated among a small fraction of the population**.
3. The simulation outcome mirrors the energy distribution among particles in an ideal gas, where energy is exchanged randomly during collisions.

Extra Exercise: Modify the model to introduce the following features:

- Add a new attribute called `transaction-cap` to turtles, representing the maximum amount an agent can give in a single transaction (based on their wealth).
- Introduce a reputation system: Each agent starts with a `reputation` value of 100. Agents with higher reputation are more likely to receive money.
- Update the `transact` procedure:
 - Agents give an amount based on their `transaction-cap` (e.g., up to 10% of their wealth).
 - The recipient is selected randomly but weighted by their `reputation` (higher reputation = higher chance).
 - Reputation increases for agents who receive money and decreases for agents who give money.
- Update the plots:
 - Create a plot to track the average reputation of agents over time.
 - Modify the wealth distribution plot to include both wealth and reputation effects.

Observe how introducing `transaction-cap` and reputation affects the wealth and reputation

✓ **4.2 Flattening The Curve** distribution over time. Try varying the initial values for `transaction-cap` and `reputation` to see their effects on inequality.

The **COVID-19 pandemic**, one of the most significant global health crises in recent history, exposed the challenges of managing an unprecedented epidemic that disrupted economies and healthcare systems worldwide. The complexity of disease transmission, coupled with the uncertainty surrounding symptoms, prevention strategies, and public health guidelines, made decision-making **incredibly difficult for policymakers**.

In such critical situations, simulation models, particularly agent-based models (ABMs), can play a huge role in exploring potential scenarios and evaluating intervention strategies. While it is impossible to fully replicate the intricacies of a global epidemic, abstracting key elements into a model allows for a deeper understanding of disease dynamics and the potential impact of policy decisions.

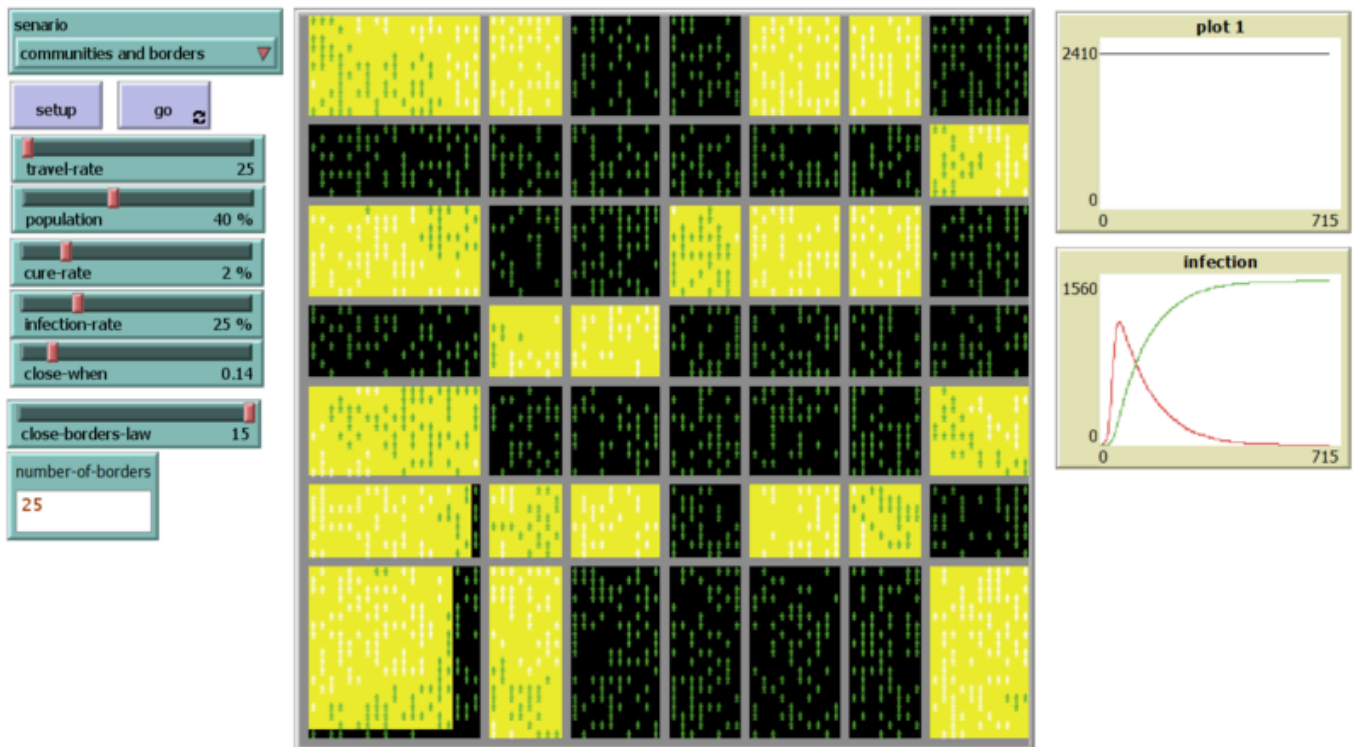
However, modeling public health crises comes with its own set of challenges. Simulations must account for limited data, uncertainty in hyperparameters, and the inherent variability of human behavior. Furthermore, developing and deploying these models requires rigorous validation and ethical considerations, as they inform decisions that directly impact millions of lives.

While not a substitute for real-world experimentation, agent-based models offer a valuable decision-support tool for analyzing scenarios, testing assumptions, and informing evidence-based policy-making during epidemics and other complex healthcare crises.

4.2.1 Modeling Commons

[Modeling Commons](#) is a website for sharing and discussing agent-based models written in NetLogo. There are at least 1,000 models contributed by modelers from around the world.

One notable model that gained attention during the COVID-19 pandemic was titled "[Covid-19: How quarantine can flatten the curve](#)". This model, while relatively simple in design, focused on exploring the effectiveness of quarantine measures in reducing the spread of infection and "flattening the curve" within a population.



4.2.2 The Code

Reference: Modelling Commons - Vsevolod Suschevskiy

The code adheres to the typical NetLogo structure, with global variables, breeds, and agent-specific attributes clearly defined. The `borders` global variable serves as a flag to indicate whether the borders are currently closed.

The primary agents in the model are the `actor` breed, which are responsible for carrying and spreading the infection. The `borders` breed, on the other hand, is used primarily for visualization purposes and to act as immovable barriers that restrict movement within the environment. Each `actor` is equipped with an additional attribute, `days`, which tracks the number of days an agent has been infected with the virus.

```
globals [  
  borders? ;; Flag to track if borders are closed  
]  
  
breed [actors actor] ;; Main agents representing the population  
breed [borders border] ;; Temporary breed for creating border visuals  
  
actors-own [  
  days ;; Tracks the number of days an actor has been infected  
]
```

The `setup` code primarily focuses on initializing the environment, using a somewhat convoluted approach to visualize the borders and set up the simulation. It involves generating the borders through algorithmic placement, spawning healthy agents within the grid, and designating one of the agents as the initial infected individual.

```

to setup
  clear-all
  reset-ticks
  clear-plot

  set borders? FALSE ;; Initialize borders as open

;; Scenario-based setup
(ifelse senario = "base" [
  ;; Base scenario: spawn a proportion of the population on black patches
  ask n-of (count patches with [pcolor = black] * population / 100) patches [
    sprout-actors 1 [
      set shape "person"
      set color white ;; Healthy actors start as white
    ]
  ]
  ask one-of actors [set color red] ;; Infect one random actor
]
;; Communities and borders scenario
senario = "communities" or senario = "communities and borders" [
  create-borders 10 [
    setxy 0 0
    set heading one-of [90 270] ;; Borders align horizontally or vertically
  ]

;; Draw initial border lines
ask borders [
  repeat 4 [
    repeat sqrt (count patches) [
      set pcolor grey ;; Create a horizontal line of borders
      fd 1
    ]
    rt 90 ;; Turn to create the next segment
    repeat sqrt (count patches) + sqrt (count patches) / 4 [
      set pcolor grey ;; Extend the border vertically
      fd 1
    ]
  ]
]

;; Add additional borders for larger grids
if sqrt (count patches) > 20 [
  ask borders [
    setxy sqrt(sqrt(count patches)) sqrt(sqrt(count patches)) ;; Place at center
    repeat 4 [
      repeat sqrt (count patches) [
        set pcolor grey
        fd 1
      ]
      rt 90
      repeat sqrt (count patches) + sqrt (count patches) / 4 [
        set pcolor grey
        fd 1
      ]
    ]
  ]
]

```

```

    ]
  ]
]

;; Create a central border
create-borders 1 [
  setxy sqrt(count patches) / 2 sqrt(count patches) / 2
  set heading 90
]

;; Extend central borders
ask border 10 [
  repeat 2 [
    repeat sqrt (count patches) [
      fd 1
      set pcolor grey
    ]
    rt 90
  ]
]

;; Clean up temporary borders
ask borders [die]

;; Spawn actors based on population percentage
ask n-of (count patches with [pcolor = black] * population / 100) patches with [pcolor = bl
  sprout-actors 1 [
    set shape "person"
    set color white
  ]
]
ask one-of actors [set color red] ;; Infect one random actor
]
[stop] ;; End setup if no valid scenario
)
end

```

4.2.3 The go Loop

The go loop is relatively straightforward but relies on several utility functions to handle various aspects of the simulation. Here's a breakdown of its functionality:

- The loop terminates if there are no infected agents (red-colored actors) remaining.
- Agents move around the map using the `travel` function.
- Infection spreads to neighboring agents through the `infect` function.
- If the proportion of infected agents surpasses the `close-when` threshold, borders are closed using the `close-borders` function.
- The `not-live` function handles deaths among infected agents.
- Recovery is managed via the `cure` function.
- Finally, the simulation advances to the next tick, and the loop continues.

```

to go
  ;; Stop if there are no infected (red) actors
  if not any? actors with [color = red] [stop]

  ;; Model dynamics
  travel ;; Move actors around
  infect ;; Spread infection
  if count actors with [color = red] / count actors > close-when [
    close-borders ;; Close borders when infection threshold is met
  ]
  not-live ;; Handle deaths from infection
  cure ;; Handle recovery

  tick ;; Advance simulation time
end

```

4.2.4 Traveling

The `travel` function is straightforward yet effectively simulates the concept of movement. It selects a proportion of agents standing on black patches (indicating areas not blocked by borders) based on the `travel-rate`. If the `travel-rate` is set to 100, all agents will travel; if it is set to 1, only 1% of the agents will travel. The exact number of agents selected to travel depends on the `density` global parameter.

All selected traveling agents are temporarily moved to patch `(0, 0)`. At this location, the patch agent (the patch itself) is asked to instruct all actors on it to move to a random patch that is both unoccupied and has a black color. This ensures that agents do not move back to their original location, facilitating valid movement across the grid.

```

to travel
  ;; Move a proportion of actors based on travel-rate
  ask n-of (count actors-on patches with [pcolor = black] * travel-rate / 100)
  actors-on patches with [pcolor = black] [
    setxy 0 0
  ]
  ;; Move actors to random unoccupied black patches
  ask patch 0 0 [
    ask actors-here [
      if not any? patches with [not any? turtles-here and pcolor = black] [stop]
      move-to one-of patches with [not any? turtles-here and pcolor = black]
    ]
  ]
end

```

4.2.5 Infecting

The `infect` function is straightforward, with no complex dynamics. It instructs all infected actors (red-colored agents) to check their eight neighboring patches. For each neighboring patch, if an actor is present and not already infected, there is a chance, determined by the `infection-rate`, that the actor becomes infected. If the neighboring actor is already infected, no action is taken.

```

to infect
;; Infected actors (red) spread infection to healthy neighbors
ask actors with [color = red] [
  ask neighbors [
    ask actors-here with [color = white] [
      if random 100 <= infection-rate [set color red]
    ]
  ]
]
end

```

4.2.6 Closing Borders

The `close-borders` procedure operates only in the `communities` and `borders` scenario and when the borders are still open. It selects a specified number (`number-of-border`) of patches to serve as initial border closures, changing their color to yellow. The procedure then iteratively propagates the closure by expanding the yellow area to neighboring black patches, repeating this process `close-borders-law` times. This ensures that travel, which can only occur on black patches, is restricted. Finally, the `borders?` flag is set to indicate that the borders are now closed.

```

to close-borders
  if not borders? [
    if senario = "communities and borders" [
      ;; Mark random patches as closed borders
      ask n-of number-of-borders patches with [pcolor = black] [
        set pcolor yellow
      ]
      ;; Expand borders by specified steps
      repeat close-borders-law [
        ask patches with [pcolor = yellow] [
          ask neighbors with [pcolor = black] [set pcolor yellow]
        ]
      ]
      set borders? TRUE ;; Borders are now closed
    ]
  ]
end

```

4.2.7 Not Living

The `not-live` function, which handles the process of dying, is also relatively simple in this simulation. Infected actors (red-colored agents) increment their `days` counter at every tick to track how long they have been infected. As the number of days increases, the probability of dying also increases. The chance of death reaches a maximum of 20%, simulating a higher mortality risk over time. This essentially means that each agent has the following probability of dying:

$$P(\text{not-live}_i) = \begin{cases} 0 & \text{if } \text{days_infected}_i \leq 30 \\ \left(\frac{\text{days_infected}_i - 30}{100} \right) \times \frac{1}{5} & \text{if } \text{days_infected}_i > 30 \end{cases}$$

```

to not-live
  ;; Infected actors (red) age and may die over time
  ask actors with [color = red] [
    set days days + 1
    if random 100 <= days - 30 [
      if random 100 < 20 [die] ;; Chance of death increases with time
    ]
  ]
end

```

4.2.8 Cure

The cure process can occur for actors that are infected. The probability of an infected actor i being cured can be represented by the following formula:

$$P(\text{cure}_i) = \left(\frac{\text{cure_rate}}{100} \right) \times \frac{1}{2}$$

Exercise: Experiment with the hyperparameters of the model.

- Adjust the values for each hyperparameter (e.g., population, infection-rate, travel-rate, cure-rate, and others).
- Log the results of each simulation run, including the shape of the infection curve and the total number of deaths.
- Analyze how each hyperparameter influences the dynamics of the simulation, particularly the infection curve and mortality rate.

Exercise: Adjust travel and border rules to minimize deaths.

- Fix the infection-related parameters (e.g., infection-rate, cure-rate, etc.) to their default values.
- Modify the travel and border-related rules (e.g., travel-rate, close-borders-law, and number-of-borders).
- Run the simulation and log the results for each configuration, focusing on the total death count.
- Can you find a set of hyperparameters that minimizes the death count?

Exercise: Modify the infection dynamics by allowing agents to infect others within a variable radius.

- Add a slider `infection-radius` in the Interface tab to allow the user to control the radius of infection.
- Modify the `infect` procedure so that it checks all agents within the specified radius, not just neighbors.
- Run simulations with different values of `infection-radius` and observe how the infection spreads over time.

Solution:

▼ Click to show/hide solution

```

to infect
  ask actors with [color = red] [
    ask turtles in-radius infection-radius [
      if [color] of self = white [ ;; Check if the neighbor is healthy

```

```

        if random 100 <= infection-rate [
            set color red ;; Infect the agent
        ]
    ]
]
end

```

Exercise: Add a vaccination strategy to the model.

- Create a slider `vaccination-rate` that determines the percentage of healthy actors vaccinated at the start.
- Vaccinated actors should be immune to infection and represented by a unique color (e.g., blue).
- Modify the `setup` procedure to randomly vaccinate a proportion of the population based on `vaccination-rate`.
- Analyze the effect of vaccination on the infection curve and total deaths.

Solution:

▼ Click to show/hide solution

```

actors-own [
    days ;; Tracks the number of days an actor has been infected
    vaccinated? ;; Tracks if an actor is vaccinated
]

set vaccinated? false

vaccinate
ask one-of actors with [not vaccinated?] [set color red]

to vaccinate
    ask n-of (vaccination-rate * count actors / 100) actors with [not vaccinated?] [
        set vaccinated? true
        set color blue ;; Vaccinated actors are immune and represented by blue
    ]
end

```

```

to infect
    ask actors with [color = red] [
        ask turtles in-radius infection-radius [
            if [color] of self = white and [not vaccinated?] of self [
                if random 100 <= infection-rate [
                    set color red ;; Infect the agent
                ]
            ]
        ]
    ]
end

```

Extra Exercise: Add super-spreader agents with higher infection rates.

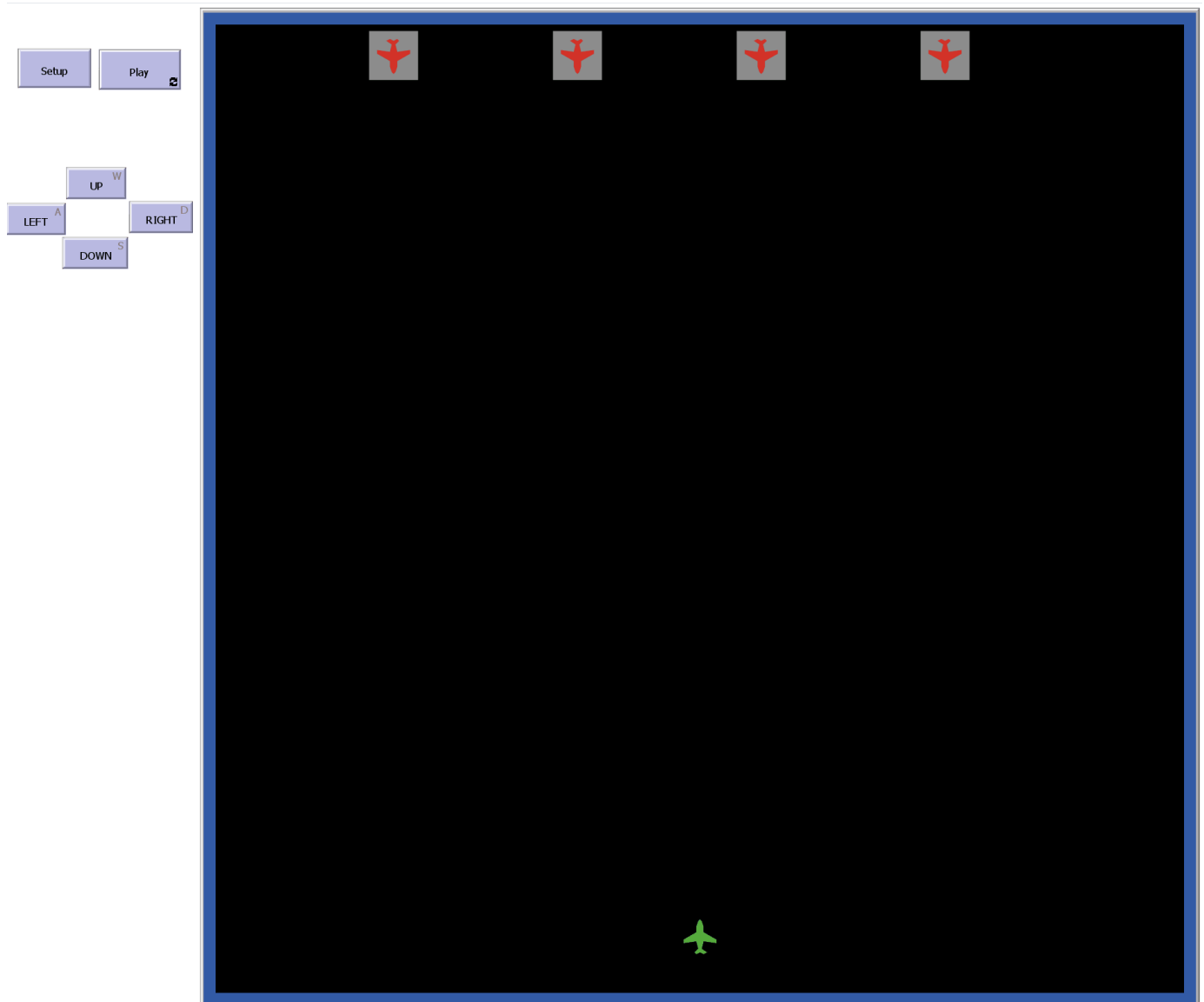
- Create a certain percentage of the population as super-spreaders at the start of the simulation.
- Super-spreaders should have a higher infection radius and infection rate compared to regular agents.
- Modify the setup and infect procedures to account for super-spreaders.

✓ 4.3 Game Development?

The implementation and development of this project were carried out by Ramesh Maddegoda in June 2021.

Can you create simple games in NetLogo? **Absolutely!** By leveraging the built-in tools and features, you can design simulations that use keyboard inputs such as W, A, S, and D for movement, as well as button clicks to trigger specific events within the environment. NetLogo also supports **other mouse and keyboard inputs**, allowing for a wide range of interactive possibilities.

Let's explore what creative games can be built in NetLogo!



Let's just quickly go through what makes this game being a thing possible. Let's look at some global variables, agent attributes and breeds.

```

breed [ players player ]
breed [ player-bullets player-bullet ]
breed [ enemy-bullets enemy-bullet ]
breed [ landing-zones landing-zone ]
breed [ enemies enemy ]
breed [ explosions explosion ]
breed [ final-statuses final-status ]

; Global variables
globals [
  mouse-was-down
  stop-game
]

; Private variable
players-own [
  health
]

enemies-own [
  health
]

```

There are several breeds utilized in this code, each serving a distinct and specific purpose. Let's break them down:

- **Player:** Represents the main agent controlled by you.
- **Player-bullets:** These are the bullet agents fired by the player.
- **Enemy-bullets:** Bullets fired by enemy agents targeting the player.
- **Landing-zones:** Designated areas where enemy agents spawn.
- **Enemy:** Represents the hostile agents that the player must defeat.
- **Explosion:** Simulates explosions using orange agents that scatter outward from the point of impact.
- **Final-statuses:** Used to display emojis or symbols on the screen at the end of the game by coloring specific patches.

The `setup` command is straightforward but crucial for initializing the game. It creates the player agent and positions it at the bottom of the grid. The player is assigned a plane-shaped appearance, customized color, heading, and a health attribute, which is shared by both `players` and `enemies` as agent-specific attributes. Additionally, it sets up the landing zones where enemy agents will spawn and creates the enemy agents with similar properties to the player. Finally, the `setup` command draws the blue borders around the grid to define the play area visually.

```

to setup

  clear-all
  reset-ticks

  set stop-game false

  setup-players
  setup-landing-zones
  setup-enemies

  set mouse-was-down false

```

```

    ask patches with [ count neighbors != 8 ]
    [ set pcolor blue ]
end

```

```

to setup-players
  create-players 1
  ask players [
    set shape "Airplane"
    set color green
    set size 3
    setxy 40 5
    set heading 0
    set health 100
  ]
end

```

```

to setup-enemies
  ask enemies [
    set shape "Airplane"
    set size 3
    set color red
    set heading 180
    set health 100
  ]
end

```

```

to setup-landing-zones
  let x 0
  create-landing-zones 4
  ask landing-zones [
    set shape "square"
    set size 5
    set color grey
    set x ( x + 15 )
    setxy x (max-pycor - 3)
    hatch-enemies 1 [
      create-link-from myself [
        set color black
      ]
    ]
  ]
]
end

```

The `stop-game` command is responsible for managing the game's state, determining when the game should end based on specific conditions. The `mouse-was-down` command tracks the status of mouse clicks, enabling interactions or triggering events within the simulation based on user input.

4.3.1 The Play Loop

The play loop is designed to run indefinitely but can be terminated when the `stop-game` global variable is set to `true`. This variable is triggered when either all enemy planes' health reaches 0 or the player's health is depleted.

```
to play
  tick
  if stop-game = true [
    stop
  ]

  player-rules
  player-bullet-rules
  enemy-bullet-rules
  enemy-rules
  explosion-rules
  check-mouse-button
end
```

The `player-rules` command is straightforward: it checks if the player's health has reached 0. If so, it triggers the `game-over` command, which creates the emoji effect and sets the `stop-game` variable to `true`, effectively ending the game.

```
to player-rules
  ask players [
    if health <= 0 [
      game-over
    ]
    set label round(health)
    facexy mouse-xcor mouse-ycor
  ]
end
```

```
to game-over
  hatch-final-statuses 1 [
    setxy 40 40
    set shape "face sad"
    set size 15
    set label ""
    set color yellow
  ]
  set stop-game true
end
```

The `enemy-rules` command follows a similar structure to `player-rules` but includes additional logic for enemy behavior. It checks whether an enemy's health has dropped to 0, and if so, triggers the `explode` command and removes the agent. If an enemy is far from the player (beyond a distance of 50), it patrols the environment in a predefined motion. However, if it comes within 50 units of the player, it begins to approach and fire bullets. The explosion effect for enemies is also handled within this command. Let's take a closer look at the `explode` command.

```
to explode
  hatch-explosions 25 [
```

```

    set shape "Default"
    set color orange
    set size 2
    set heading random 360
    set label ""
  ]
  sound:play-note "Gunshot" 0 64 2
end

```

The `explode` command simply hatches turtles of the `explosion` breed, creating the visual representation of the explosion. However, it does not handle their movement. The movement and behavior of these explosion agents are managed by the `explosion-rules` command, which is executed in the `play` loop.

```

to explosion-rules
  ask explosions [
    fd 0.01
    if [pcolor] of patch-here = blue [
      die
    ]
  ]
end

```

The `x-bullet-rules` command controls the bullet behavior in the model. It asks all bullet agents to move forward by one unit. The bullets are removed under specific conditions:

- if they reach the borders of the grid
- if they enter the landing zones
- or if they collide with enemy bullets within a radius of 3.

Additionally, if a bullet hits an enemy within a radius of 3, it reduces the enemy's health by 0.01. When bullets collide or hit their targets, they trigger the `explode` command, which hatches `explosion` agents to create a visual effect, similar to the explosions caused by enemy deaths.

```

to bullet-explode
  hatch-explosions 3 [
    set shape "Default"
    set color grey
    set size 1
    set heading random 360
    set label ""
  ]
  sound:play-note "Gunshot" 50 64 2
end

```

The `check-mouse-button` command is another key part of the model. It monitors whether the mouse button was pressed during the previous tick. If it detects that the mouse was clicked, it hatches a bullet from the player's position.

✓ 4.3.2 Movement

Lastly, the `go-up` command handles player movement, allowing the player to move upward when the `w` key is pressed.

```

to go-up
  ask players [
    set heading 0
    if ycor < max-pycor [
      set ycor (ycor + 1)
    ]
  ]
end

```

The player's vertical position (`ycor`) stays within the defined bounds, preventing the player from exceeding the grid's upper limit.

Extra Exercise: Add Power-ups

- Create a new `power-up` breed that spawns randomly on the grid.
- Make power-ups provide benefits to players, such as increased speed, additional health, or temporary immunity.
- Ensure power-ups disappear after a certain time or after being collected by the player.

Extra Exercise: Add Obstacles

- Create stationary or moving obstacles that block bullets and player movement.
- Ensure obstacles are randomly generated during the `setup` procedure of the game.

Extra Exercise: Multiplayer Support

- Add support for multiple players controlled by different keys (e.g., `WASD` for Player 1 and `Arrow Keys` for Player 2).
- Implement cooperative gameplay where players work together to defeat enemies.

Extra Exercise: Add difficulty levels:

- Implement a difficulty scaling system where the number and health of enemies increase as the player progresses.
- Optionally, increase the speed or frequency of enemy bullets over time.

✓ 5. Practice - Games: Models of Multi-Agent Interaction

👤 Tamás Takács, PhD student, Department of Artificial Intelligence

🕒 90 min read

📅 January 22, 2025

🏷️ Collective Intelligence



ELTE | IK
INFORMATIKAI KAR



DEPARTMENT
OF ARTIFICIAL
INTELLIGENCE



Game Theory

['gām 'thē-ə-rē]

A theoretical framework
for conceiving social
situations among
competing players.

Multi-agent systems are composed of autonomous agents whose interactions shape the behavior of complex environments. To rigorously analyze and design such systems, it is essential to employ formal frameworks that describe how agents make decisions, respond to one another, and distribute rewards or penalties. This practice notebook introduces students to the foundational concepts and models from game theory that underpin multi-agent interaction, including normal-form games, repeated games, stochastic games, and settings with partial observability and communication. Through a series of structured examples and simulations, students will learn to represent, analyze, and experiment with a variety of multi-agent environments. By the end of this module, students will be able to formally describe multi-agent scenarios, identify appropriate models for different classes of problems, and implement simulations that highlight the core principles of strategic interaction.

These formal models are grounded in **game theory**, and are known as **games**. Games serve as abstract representations of interaction, ranging from one-shot choices to ongoing processes in complex environments. Depending on what agents can observe and how the environment evolves, we can classify games into a hierarchy of increasing complexity.

Table of Contents

- **5.1 Hierarchy of Game Models**
 - 5.1.1 Necessary Imports
- **5.2 Normal-Form Games**
 - 5.2.1 Rock-Paper-Scissors (Zero-Sum Matrix Game)
 - 5.2.2 Coordination Game (Common-Reward)
 - 5.2.3 Prisoner's Dilemma (General-Sum)
 - 5.2.4 Types of Normal-Form Games
- **5.3 Repeated Normal-Form Games**
 - 5.3.1 Formal Structure
 - 5.3.2 Finite vs Infinite Repetition
 - 5.3.3 Example: Repeated Prisoner's Dilemma
- **5.4 Stochastic Games (Markov Games)**

- 5.4.1 Definition
- 5.4.2 Game Flow
- 5.4.3 Markov Property
- 5.4.4 From Games to Environments
- 5.4.5 Example: Multi-Agent Grid Foraging
- **5.5 Partially Observable Stochastic Games (POSGs)**
 - 5.5.1 POSG Components
 - 5.5.2 Game Flow (with Partial Observability)
 - 5.5.3 Observability Variants
 - 5.5.4 Decentralized POMDPs (Dec-POMDPs)
 - 5.5.5 Example: Grid World with Local Vision
- **5.6 Modeling Communication in Multi-Agent Systems**
 - 5.6.1 Joint Action Space with Communication
 - 5.6.2 Communication in POSGs
 - 5.6.3 Learned Semantics
 - 5.6.4 Example: Grid-World Foraging with Message Passing
 - 5.6.5 Limitations
- **5.7 Knowledge Assumptions in Multi-Agent Games**
 - 5.7.1 Complete Knowledge Games
 - 5.7.2 Incomplete Knowledge in MARL
 - 5.7.3 From Simulator to Knowledge
 - 5.7.4 Symmetry, Asymmetry, and Common Knowledge
 - 5.7.5 Open Multi-Agent Systems
- **5.8 Dictionary: Reinforcement Learning ↔ Game Theory**

✓ 5.1 Hierarchy of Game Models

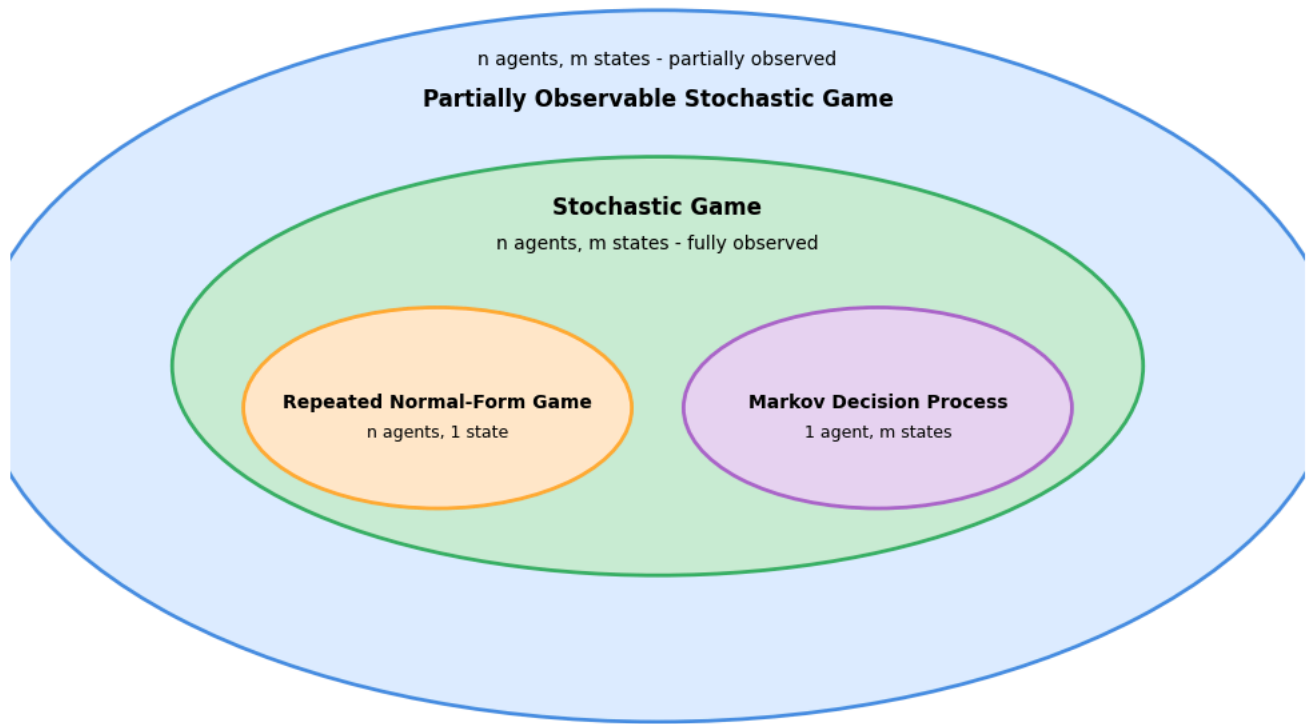
At the simplest level, we consider **normal-form games**, where agents act simultaneously in a single round of interaction, with no state transitions or memory. Moving up, we introduce **stochastic games**, where the environment has multiple states and evolves based on the agents' joint actions. At the top of the hierarchy are **partially observable stochastic games**, where agents do not have access to the full state of the environment and must act based on incomplete or noisy observations.

✓ 5.1.1 Necessary Imports

```
import matplotlib.pyplot as plt
import matplotlib.animation as animation
import matplotlib.patches as patches
import numpy as np
import random
from IPython.display import HTML, display, clear_output
import ipywidgets as widgets
```

> Hierarchy of Game Models

[Show code](#)



✓ 5.2 Normal-Form Games

A **normal-form game** models a one-shot interaction between two or more agents. Each agent selects an action from a finite set, and all actions are executed **simultaneously**. The outcome - a **joint action** - determines the **reward** each agent receives.

Let:

- $I = \{1, \dots, n\}$ be the set of agents
- A_i be the action set for agent i
- $A = A_1 \times A_2 \times \dots \times A_n$ be the joint action space
- $R_i : A \rightarrow \mathbb{R}$ be the reward function for agent i

Each agent selects a **stochastic policy** $\pi_i : A_i \rightarrow [0, 1]$ that defines a distribution over actions. After sampling actions, the resulting tuple $a = (a_1, \dots, a_n)$ is evaluated by each R_i to produce the reward r_i .

✓ 5.2.1 Rock-Paper-Scissors (Zero-Sum Matrix Game)

Each player chooses one of three actions. One player's win is the other's loss.

Agent 1 / Agent 2	Rock	Paper	Scissors
Rock	0, 0	-1, 1	1, -1
Paper	1, -1	0, 0	-1, 1
Scissors	-1, 1	1, -1	0, 0

- The game is **zero-sum**: the total reward is always 0.

```
actions = ['Rock', 'Paper', 'Scissors']
rps_payoff = {
    ('Rock', 'Rock'): (0, 0),
    ('Rock', 'Paper'): (-1, 1),
    ('Rock', 'Scissors'): (1, -1),
    ('Paper', 'Rock'): (1, -1),
    ('Paper', 'Paper'): (0, 0),
    ('Paper', 'Scissors'): (-1, 1),
    ('Scissors', 'Rock'): (-1, 1),
    ('Scissors', 'Paper'): (1, -1),
    ('Scissors', 'Scissors'): (0, 0),
}
```

```
def play_rps(policy1, policy2):
    a1 = random.choices(actions, weights=policy1)[0]
    a2 = random.choices(actions, weights=policy2)[0]
    r1, r2 = rps_payoff[(a1, a2)]
    print(f"Agent 1: {a1} | Agent 2: {a2} → Rewards: ({r1}, {r2})")
    return r1, r2

uniform_policy = [1/3, 1/3, 1/3]
play_rps(uniform_policy, uniform_policy)
```

➡ Agent 1: Paper | Agent 2: Scissors → Rewards: (-1, 1)

5.2.2 Coordination Game (Common-Reward)

Agents are rewarded only when they make the **same** choice.

Agent 1 / Agent 2	A	B
A	10, 10	0, 0
B	0, 0	10, 10

- **Both** players benefit from mutual agreement.
- This is a **common-reward** game: agents share the same payoff.

```
coordination_payoff = {
    ('A', 'A'): (10, 10),
    ('A', 'B'): (0, 0),
    ('B', 'A'): (0, 0),
    ('B', 'B'): (10, 10),
}

def play_coordination(policy1, policy2):
    actions = ['A', 'B']
    a1 = random.choices(actions, weights=policy1)[0]
    a2 = random.choices(actions, weights=policy2)[0]
    r1, r2 = coordination_payoff[(a1, a2)]
    print(f"Agent 1: {a1} | Agent 2: {a2} → Rewards: ({r1}, {r2})")
    return r1, r2

play_coordination([0.4, 0.6], [0.6, 0.4])
```

➡ Agent 1: B | Agent 2: A → Rewards: (0, 0)

5.2.3 Prisoner's Dilemma (General-Sum)

Each agent chooses to **cooperate** or **defect**. Defection is individually optimal, but mutual cooperation yields a better outcome for both.

Agent 1 / Agent 2	Cooperate (C)	Defect (D)
Cooperate (C)	-1, -1	-5, 0
Defect (D)	0, -5	-3, -3

- This is a **general-sum** game: there is no fixed relationship between the rewards.

5.2.4 Types of Normal-Form Games

Normal-form games can be classified by how agent rewards relate to each other:

Type	Mathematical Condition	Description
Zero-Sum	$\sum_i R_i(a) = 0$	One agent's gain is another's loss
Common-Reward	$R_i(a) = R_j(a) \forall i, j$	Fully cooperative: same reward shared
General-Sum	No constraint	Rewards may align or conflict

These distinctions help us reason about strategic incentives: whether agents should compete, cooperate, or mix both behaviors.

5.3 Repeated Normal-Form Games

A **repeated normal-form game** models sequential interaction between agents by replaying the same normal-form game over multiple time steps. This structure allows agents to adapt their actions based on past behavior, enabling strategic reasoning over time.

Unlike one-shot games, repeated interactions let agents build **trust, retaliation, cooperation, and revenge**, giving rise to rich dynamics in multi-agent environments.

5.3.1 Formal Structure

Let $\Gamma = (I, \{A_i\}_{i \in I}, \{R_i\}_{i \in I})$ be a normal-form game. In a repeated normal-form game:

- The same game Γ is played over T time steps: $t = 0, 1, \dots, T - 1$
- Each agent selects action $a_i^t \in A_i$ at time t , based on policy π_i
- Policies can now condition on **interaction history**:
$$\pi_i(a_i^t \mid h^t) \quad \text{where } h^t = (a^0, a^1, \dots, a^{t-1})$$
- Each joint action $a^t = (a_1^t, \dots, a_n^t)$ yields a reward:
$$r_i^t = R_i(a^t)$$

Agents can remember past behavior or use **summary statistics** of the history to guide future decisions. Some famous policies:

Policy	Description
Tit-for-Tat	Start with cooperate, then repeat opponent's last move
Grim Trigger	Cooperate until opponent defects once, then always defect
Always Defect	Defect no matter what
Always Cooperate	Always cooperate regardless of past moves
Random	Choose actions uniformly at random

5.3.2 Finite vs Infinite Repetition

- In **finite repetition**, agents often act differently near the end (e.g., end-game defection).
- In **infinite repetition**, we often use a **discount factor** $\gamma \in [0, 1]$ to weigh future rewards.
 - This introduces a termination probability $1 - \gamma$ at each time step.
 - For $\gamma < 1$, the game is still considered "infinite," since the number of rounds is unbounded in expectation.

✓ 5.3.3 Example: Repeated Prisoner's Dilemma

Agent 1 / Agent 2	Cooperate (C)	Defect (D)
Cooperate (C)	-1, -1	-5, 0
Defect (D)	0, -5	-3, -3

When repeated, this game reveals fascinating dynamics:

- Tit-for-Tat can enforce cooperation over time
- Defectors can exploit naive cooperators
- Strategy adaptation is key to long-term reward

We'll now simulate a **multi-round Axelrod-style tournament** between famous strategies.

```
PAYOFFS = {
    ("C", "C"): (-1, -1),
    ("C", "D"): (-5, 0),
    ("D", "C"): (0, -5),
    ("D", "D"): (-3, -3),
}

def always_cooperate(_, __): return "C"
def always_defect(_, __): return "D"
def tit_for_tat(_, history):
    return "C" if not history else history[-1][1]

def grim_trigger(_, history):
    if any(h[1] == "D" for h in history): return "D"
    return "C"

def random_strategy(_, __): return random.choice(["C", "D"])

STRATEGIES = {
    "Always Cooperate": always_cooperate,
    "Always Defect": always_defect,
    "Tit-for-Tat": tit_for_tat,
    "Grim Trigger": grim_trigger,
    "Random": random_strategy,
}

def play_rounds(strategy1, strategy2, rounds=20):
    history1, history2 = [], []
    rewards1, rewards2 = 0, 0
    for _ in range(rounds):
        move1 = strategy1(1, history1)
        move2 = strategy2(2, history2)
        r1, r2 = PAYOFFS[(move1, move2)]
```

```

    rewards1 += r1
    rewards2 += r2
    history1.append((move1, move2))
    history2.append((move2, move1))
    return rewards1, rewards2

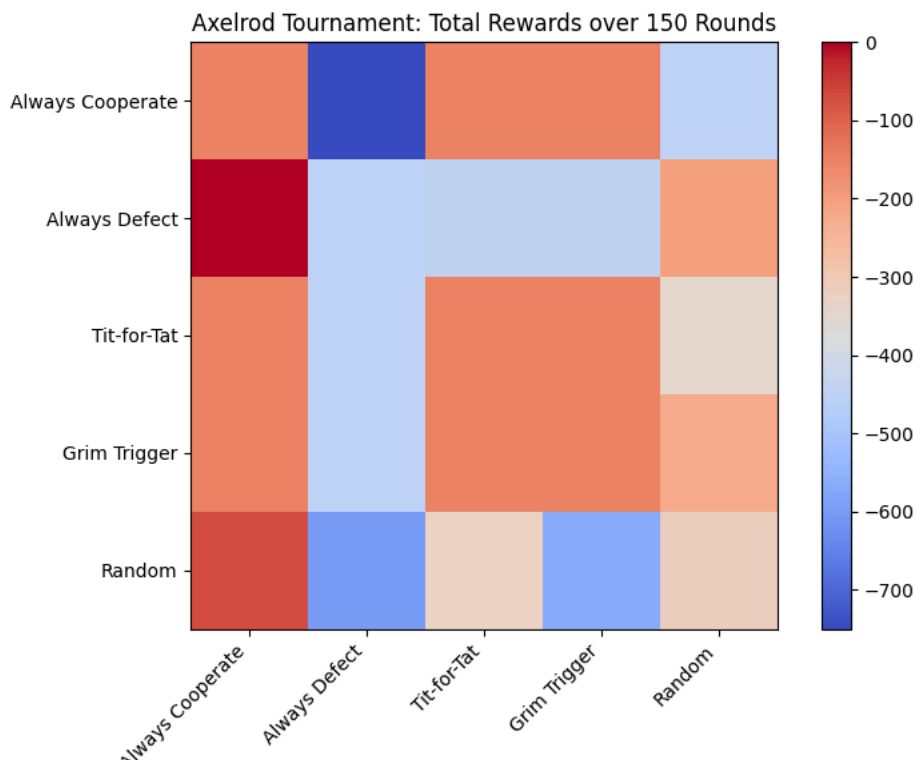
def axelrod_tournament(strategies=STRATEGIES, rounds=20):
    names = list(strategies.keys())
    n = len(names)
    scores = np.zeros((n, n))

    for i in range(n):
        for j in range(n):
            s1, s2 = strategies[names[i]], strategies[names[j]]
            r1, r2 = play_rounds(s1, s2, rounds)
            scores[i, j] = r1

    fig, ax = plt.subplots(figsize=(8, 6))
    im = ax.imshow(scores, cmap="coolwarm", vmin=np.min(scores), vmax=np.max(scores))
    ax.set_xticks(np.arange(n))
    ax.set_yticks(np.arange(n))
    ax.set_xticklabels(names, rotation=45, ha="right")
    ax.set_yticklabels(names)
    plt.title(f"Axelrod Tournament: Total Rewards over {rounds} Rounds")
    plt.colorbar(im, ax=ax)
    plt.tight_layout()
    plt.show()

axelrod_tournament(rounds=150)

```



```

def simulate_game(strategy1, strategy2, rounds=50):
    history1, history2 = [], []
    rewards1, rewards2 = [], []
    total1, total2 = 0, 0

    for _ in range(rounds):
        move1 = strategy1(1, history1)
        move2 = strategy2(2, history2)
        r1, r2 = PAYOFFS[(move1, move2)]
        total1 += r1
        total2 += r2
        rewards1.append(total1)
        rewards2.append(total2)
        history1.append((move1, move2))
        history2.append((move2, move1))

    return rewards1, rewards2

def animate_rewards(strategy_name_1, strategy_name_2, rounds=50):
    strategy1 = STRATEGIES[strategy_name_1]

```

```

strategy2 = STRATEGIES[strategy_name_2]

rewards1, rewards2 = simulate_game(strategy1, strategy2, rounds)

fig, ax = plt.subplots(figsize=(8, 5))
ax.set_xlim(0, rounds)
ax.set_ylim(min(rewards1 + rewards2) - 5, max(rewards1 + rewards2) + 5)
ax.set_xlabel("Round")
ax.set_ylabel("Cumulative Reward")
ax.set_title(f"{strategy_name_1} vs {strategy_name_2}")

line1, = ax.plot([], [], lw=2, color='blue', label=strategy_name_1)
line2, = ax.plot([], [], lw=2, color='red', label=strategy_name_2)
ax.legend()

def init():
    line1.set_data([], [])
    line2.set_data([], [])
    return line1, line2

def update(frame):
    x = list(range(frame + 1))
    line1.set_data(x, rewards1[:frame + 1])
    line2.set_data(x, rewards2[:frame + 1])
    return line1, line2

ani = animation.FuncAnimation(fig, update, frames=rounds, init_func=init,
                              interval=200, blit=True)

plt.close(fig)
return HTML(ani.to_jshtml())

strategy_options = list(STRATEGIES.keys())
dropdown_1 = widgets.Dropdown(options=strategy_options, value='Tit-for-Tat', description='Agent 1')
dropdown_2 = widgets.Dropdown(options=strategy_options, value='Always Defect', description='Agent 2')
output = widgets.Output()

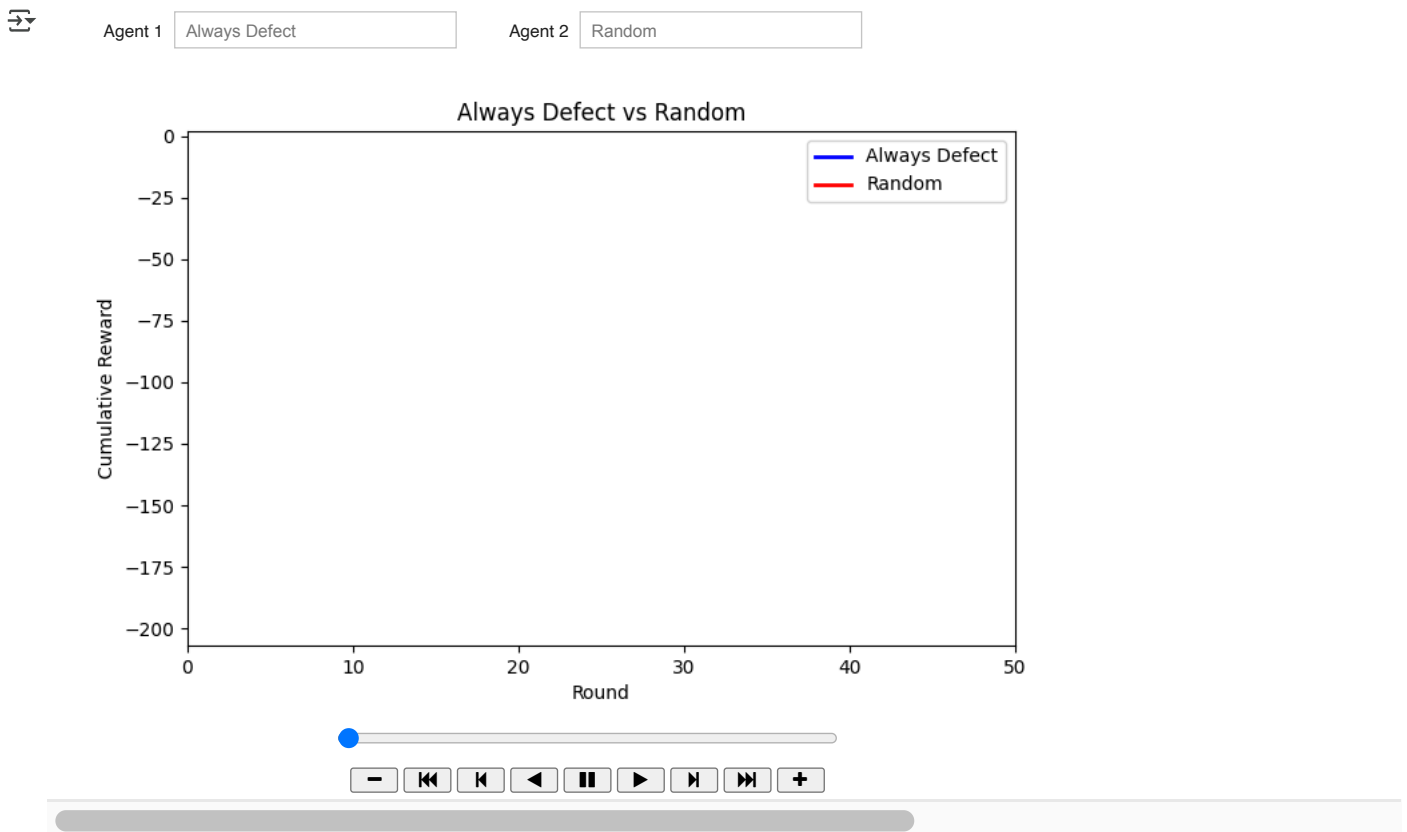
def on_change(change):
    output.clear_output(wait=True)
    with output:
        display(animate_rewards(dropdown_1.value, dropdown_2.value))

dropdown_1.observe(on_change, names='value')
dropdown_2.observe(on_change, names='value')

display(widgets.HBox([dropdown_1, dropdown_2]))
display(output)

on_change(None)

```



✓ 5.4 Stochastic Games (Markov Games)

A **stochastic game** models multi-agent interaction in a dynamic environment where the **state evolves over time** based on the agents' actions and probabilistic transitions.

This structure brings us closer to realistic multi-agent systems, where decisions are not made in isolation but in evolving, context-dependent environments.

5.4.1 Definition

A stochastic game consists of:

- A finite set of **agents**: $I = \{1, \dots, n\}$
- A finite set of **states**: S (with a subset of **terminal states** $\bar{S} \subset S$)
- For each agent $i \in I$:
 - A finite **action set** A_i
 - A **reward function**:

$$R_i : S \times A \times S \rightarrow \mathbb{R}$$

where $A = A_1 \times A_2 \times \dots \times A_n$

- A **transition function**:

$$T : S \times A \times S \rightarrow [0, 1] \quad \text{with} \quad \sum_{s'} T(s, a, s') = 1$$

- An **initial state distribution**:

$$\mu : S \rightarrow [0, 1] \quad \text{with} \quad \sum_{s \in S} \mu(s) = 1$$

5.4.2 Game Flow

At each time step t :

1. The environment is in a state $s_t \in S$
2. Each agent selects action $a_i^t \sim \pi_i(a_i \mid h^t)$
3. The **joint action** $a^t = (a_1^t, \dots, a_n^t)$ is taken
4. The environment transitions to $s_{t+1} \sim T(s_t, a^t, \cdot)$
5. Each agent receives reward $r_i^t = R_i(s_t, a^t, s_{t+1})$
6. The game continues unless:
 - a **terminal state** $s_{t+1} \in \bar{S}$ is reached, or
 - a fixed time horizon is completed

Agents condition their policies on the **state-action history** $h^t = (s_0, a^0, s_1, \dots, s_t)$.

5.4.3 Markov Property

Stochastic games inherit the **Markov property**:

$$P(s_{t+1}, r_t \mid s_t, a^t, \dots, s_0, a^0) = P(s_{t+1}, r_t \mid s_t, a^t)$$

This means future transitions depend **only** on the current state and actions - not on the entire past.

5.4.4 From Games to Environments

Stochastic games generalize:

- **Normal-form games** (single state, no transitions)
- **Repeated games** (same state, deterministic loop)
- **MDPs** (single-agent stochastic games)

They can be classified as:

- **Zero-sum**: $\sum_i R_i = 0$
- **Common-reward**: $R_i = R_j$ for all i, j
- **General-sum**: Arbitrary reward structure

✓ 5.4.5 Example: Multi-Agent Grid Foraging

A simple environment might look like:

- States: Positions of agents and items on a grid
- Actions: Move (up/down/left/right), pick, or noop
- Transitions: Agents move deterministically; items disappear once picked
- Rewards: +1 for picking an item (shared or individual)

```
GRID_SIZE = 5
ITEM_POS = (3, 3)
MAX_STEPS = 10

ACTIONS = ["UP", "DOWN", "LEFT", "RIGHT", "PICK", "NOOP"]
ACTION_TO_DELTA = {
    "UP": (-1, 0), "DOWN": (1, 0),
    "LEFT": (0, -1), "RIGHT": (0, 1),
    "PICK": (0, 0), "NOOP": (0, 0)
}

class ForagingGame:
    def __init__(self):
        self.agent_positions = [(0, 0), (4, 4)]
        self.item_present = True
        self.timestep = 0
        self.history = []

    def step(self, actions):
        rewards = [0, 0]
        new_positions = []

        for i, action in enumerate(actions):
            if action in ACTION_TO_DELTA:
                dx, dy = ACTION_TO_DELTA[action]
                x, y = self.agent_positions[i]
                nx, ny = min(max(x + dx, 0), GRID_SIZE - 1), min(max(y + dy, 0), GRID_SIZE - 1)
                new_positions.append((nx, ny))
            else:
                new_positions.append(self.agent_positions[i])

        self.agent_positions = new_positions

        if self.item_present and "PICK" in actions:
            pickers = [i for i, act in enumerate(actions) if act == "PICK" and self.agent_positions[i] == ITEM_POS]
            if pickers:
                reward = 1.0 / len(pickers)
                for i in pickers:
                    rewards[i] = reward
                self.item_present = False

        self.timestep += 1
        self.history.append((tuple(self.agent_positions), tuple(actions), tuple(rewards)))
        return rewards, self.timestep >= MAX_STEPS or not self.item_present

    def render(self):
        fig, ax = plt.subplots(figsize=(5, 5))
        ax.set_xlim(0, GRID_SIZE)
        ax.set_ylim(0, GRID_SIZE)
        ax.set_xticks(range(GRID_SIZE+1))
        ax.set_yticks(range(GRID_SIZE+1))
        ax.grid(True)

        if self.item_present:
            ax.add_patch(patches.Circle((ITEM_POS[1]+0.5, ITEM_POS[0]+0.5), 0.3, color='orange'))

        for i, (x, y) in enumerate(self.agent_positions):
            ax.add_patch(patches.Rectangle((y+0.1, x+0.1), 0.8, 0.8, color='blue' if i==0 else 'green'))
            ax.text(y + 0.5, x + 0.5, f"A{i}", ha='center', va='center', color='white', weight='bold')

        plt.gca().invert_yaxis()
        plt.title(f"Step {self.timestep}")
        plt.show()

game = ForagingGame()

def agent_policy(agent_id, state, history):
    x, y = state[agent_id]
    if (x, y) == ITEM_POS:
        return "PICK"
    if x < ITEM_POS[0]: return "DOWN"
    if x > ITEM_POS[0]: return "UP"
    if y < ITEM_POS[1]: return "RIGHT"
    if y > ITEM_POS[1]: return "LEFT"
```

```

    if not done:
        return "NOOP"

while True:
    state = game.agent_positions
    actions = [agent_policy(i, state, game.history) for i in range(2)]
    rewards, done = game.step(actions)
    game.render()
    if done:
        print(f"Final rewards: {rewards}")
        break

```

✓ 5.5 Partially Observable Stochastic Games (POSGs)

A **Partially Observable Stochastic Game (POSG)** extends stochastic games by introducing **incomplete information**. Instead of seeing the full environment state and past actions, each agent receives a **noisy, local, or partial observation** about the environment.

This mirrors real-world situations:

- Autonomous vehicles sense the world via imperfect sensors
- Players in card games don't see each other's hands
- Agents in games operate under fog-of-war or limited vision

5.5.1 POSG Components

A POSG includes everything from a stochastic game:

- Finite set of **agents**: $I = \{1, \dots, n\}$
- Finite set of **states**: S and **terminal states** $\bar{S} \subset S$
- **Action sets**: A_i for each agent i
- **Reward functions**:

$$R_i : S \times A \times S \rightarrow \mathbb{R}$$

- **State transitions**:

$$T(s_{t+1} \mid s_t, a_t)$$

Additional POSG elements:

- For each agent i :
 - **Observation space**: O_i
 - **Observation function**:

$$O_i(o_i^t \mid s_t, a_{t-1}) \quad (\text{or } O_i : A \times S \times O_i \rightarrow [0, 1])$$

5.5.2 Game Flow (with Partial Observability)

At each time step t :

1. The environment is in state s_t
2. Each agent i receives an **observation** $o_i^t \sim O_i(o_i^t \mid s_t, a_{t-1})$
3. Based on its **observation history** $h_i^t = (o_i^0, \dots, o_i^t)$, each agent selects action a_i^t
4. The joint action $a^t = (a_1^t, \dots, a_n^t)$ is taken
5. The environment transitions to s_{t+1}
6. Each agent receives reward $r_i^t = R_i(s_t, a^t, s_{t+1})$

The process repeats until a terminal state or time limit is reached.

5.5.3 Observability Variants

Type	Description
Fully Observable	$o_i^t = s_t$ — the agent sees everything
State-Only Local	$o_i^t \subset s_t$ — local grid view or limited FOV
Noisy Observation	O_i is a distribution with randomness
Unobserved Opponent Actions	$o_i^t = (s_t, a_i^t)$ — agent sees state + own action only
Communication via Observation	o_i^t includes nearby agents' messages, if in range

5.5.4 Decentralized POMDPs (Dec-POMDPs)

A POSG with **common reward** (i.e., $R_i = R_j \forall i, j$) is called a **Decentralized POMDP (Dec-POMDP)**. These models are central to **cooperative multi-agent planning** where agents must coordinate using only local observations.

✓ 5.5.5 Example: Grid World with Local Vision

Imagine two agents in a 2D grid tasked with collecting items. Each agent:

- Sees only the 3x3 region around itself (partial view of state)
- Cannot see the other agent's action
- Receives +1 if an item is collected by either agent (common reward)

We'll now simulate this scenario with **local fields of view** and **limited perception**.

```
class PartialObsForagingVis:
    def __init__(self, grid_size=7, vision=2):
        self.grid_size = grid_size
        self.vision = vision
        self.agent_positions = [(0, 0), (6, 6)]
        self.item_pos = (3, 3)
        self.item_collected = False
        self.step_count = 0
        self.max_steps = 15
        self.agent_colors = ['blue', 'green']
        self.terminated = False

    def reset(self):
        self.agent_positions = [(0, 0), (6, 6)]
        self.item_pos = (3, 3)
        self.item_collected = False
        self.step_count = 0
        self.terminated = False

    def step(self, actions):
        new_positions = []
        for i, action in enumerate(actions):
            x, y = self.agent_positions[i]
            dx, dy = {
                "UP": (-1, 0),
                "DOWN": (1, 0),
                "LEFT": (0, -1),
                "RIGHT": (0, 1),
                "NOOP": (0, 0)
            }.get(action, (0, 0))
            nx, ny = min(max(x + dx, 0), self.grid_size - 1), min(max(y + dy, 0), self.grid_size - 1)
            new_positions.append((nx, ny))

        self.agent_positions = new_positions

        if not self.item_collected and self.item_pos in self.agent_positions:
            self.item_collected = True
            print("🍎 Item collected!")

        self.step_count += 1
        if self.step_count >= self.max_steps or self.item_collected:
            self.terminated = True

    def render(self):
        fig, ax = plt.subplots(figsize=(6, 6))
        ax.set_xlim(0, self.grid_size)
        ax.set_ylim(0, self.grid_size)
        ax.set_xticks(range(self.grid_size + 1))
        ax.set_yticks(range(self.grid_size + 1))
        ax.grid(True)
        ax.set_title(f"Step {self.step_count}", fontsize=14)

        if not self.item_collected:
            item_x, item_y = self.item_pos
            ax.add_patch(patches.Circle((item_y + 0.5, item_x + 0.5), 0.25, color='red'))
            ax.text(item_y + 0.5, item_x + 0.5, "🍎", ha='center', va='center', fontsize=16)

        for i, (x, y) in enumerate(self.agent_positions):
            # Agent
            ax.add_patch(patches.Rectangle((y + 0.1, x + 0.1), 0.8, 0.8,
                                           color=self.agent_colors[i], label=f"Agent {i}"))
            ax.text(y + 0.5, x + 0.5, f"A{i}", color='white', weight='bold', ha='center', va='center')

            vis_left = max(y - self.vision, 0)
            vis_right = min(y + self.vision + 1, self.grid_size)
            vis_bottom = max(x - self.vision, 0)
            vis_top = min(x + self.vision + 1, self.grid_size)

            ax.add_patch(patches.Rectangle(
```

```

        (vis_left, vis_bottom),
        vis_right - vis_left,
        vis_top - vis_bottom,
        linewidth=0.5,
        edgecolor=self.agent_colors[i],
        facecolor=self.agent_colors[i],
        alpha=0.15
    ))

```

```

ax.set_aspect('equal')
plt.gca().invert_yaxis()
plt.tight_layout()
plt.show()

```

```
env = PartialObsForagingVis()
```

```

actions_list = [
    ("DOWN", "UP"),
    ("DOWN", "UP"),
    ("DOWN", "UP"),
    ("DOWN", "LEFT"),
    ("RIGHT", "LEFT"),
    ("RIGHT", "NOOP")
]

```

```

for actions in actions_list:
    env.step(actions)
    env.render()
    if env.terminated:
        break

```

✓ 5.6 Modeling Communication in Multi-Agent Systems

Communication in multi-agent environments allows agents to share private information, coordinate actions, or collectively plan toward shared goals. In game-theoretic terms, communication is often modeled as a **separate action channel** that agents can use in parallel with their environment-interacting actions.

This allows stochastic games and POSGs to represent **message exchange protocols**, ranging from:

- Discrete symbols (e.g. "go north")
- Continuous vectors (e.g. location, intent)
- Probabilistic messages (noisy, partial, or lossy)

5.6.1 Joint Action Space with Communication

For each agent i , the **action space** is split into:

$$A_i = X_i \times M_i$$

where:

- X_i = Environment actions (move, collect, attack, etc.)
- M_i = Communication actions (e.g. sending bits or vectors)

A full joint action looks like:

$$a = \langle (x_1, m_1), (x_2, m_2), \dots, (x_n, m_n) \rangle$$

The **key property** of communication:

It does **not affect** the environment state transition directly.

Formally:

$$T(s' \mid s, a) = T(s' \mid s, \langle x_1, \dots, x_n \rangle)$$

Communication is **ephemeral** by default — it only lasts a single timestep — but agents can remember received messages through their internal states.

5.6.2 Communication in POSGs

In partially observable settings, agents may **not see the true state**, but can receive partial observations *and* messages from others.

Each agent i has an **observation function**:

$$O_i(o_i^t \mid s_t, a^{t-1})$$

You can augment the observation vector:

$$o_i^t = [\bar{s}_i, w_1^{t-1}, \dots, w_n^{t-1}]$$

where:

- \bar{s}_i is the partial view of the state
- w_j^{t-1} is the received message from agent j (possibly \emptyset)

This allows modeling:

- **Noisy communication:**
 $w_j^{t-1} = m_j^{t-1} + N(0, \sigma^2)$
- **Message loss:**
 $P(w_j^{t-1} = \emptyset) > 0$
- **Range-limited communication:**
 $w_j^{t-1} = \emptyset$ if $\text{distance}(i, j) > R$

5.6.3 Learned Semantics

In most MARL scenarios, the agents do **not** begin with a shared meaning of communication actions.

Instead, communication is modeled as **abstract actions** in M_i , and agents must:

- Learn **what messages to send**
- Learn **how to interpret received messages**

This process may lead to the emergence of:

- **Implicit protocols** (e.g., “1” means “go left”)
- **Shared vocabularies** (symbols or continuous signals)
- **Multi-agent languages** through reinforcement (Foerster et al. 2016)

5.6.4 Example: Grid-World Foraging with Message Passing

Agents move in a 2D grid, collecting items cooperatively. Each can:

- Take a move action
- Send a message from a fixed set (e.g., intended target location)

They observe:

- Their local field of view
- Past messages from other agents (possibly with noise or loss)

Messages can express:

- Position of observed items
- Intended direction
- Role coordination (e.g., “I’m defender”)

✓ 5.6.5 Limitations

- Messages are **only observed**, never directly influence the environment.
- Agents may require **many episodes** to learn communication semantics.

```
class CommGridWorld:
    def __init__(self, grid_size=7, vision=2):
        self.grid_size = grid_size
        self.vision = vision
        self.agent_positions = [(0, 0), (6, 6)]
        self.item_pos = (3, 3)
        self.item_collected = False
        self.step_count = 0
        self.max_steps = 20
        self.agent_colors = ['blue', 'green']
        self.messages = [None, None]
        self.terminated = False

    def reset(self):
        self.agent_positions = [(0, 0), (6, 6)]
        self.item_pos = (3, 3)
        self.item_collected = False
        self.step_count = 0
        self.messages = [None, None]
```

```

self.terminated = False

def step(self, actions, messages):
    new_positions = []
    for i, action in enumerate(actions):
        x, y = self.agent_positions[i]
        dx, dy = {
            "UP": (-1, 0), "DOWN": (1, 0),
            "LEFT": (0, -1), "RIGHT": (0, 1),
            "NOOP": (0, 0)
        }.get(action, (0, 0))
        nx, ny = min(max(x + dx, 0), self.grid_size - 1), min(max(y + dy, 0), self.grid_size - 1)
        new_positions.append((nx, ny))

    self.agent_positions = new_positions
    self.messages = messages

    if not self.item_collected and self.item_pos in self.agent_positions:
        self.item_collected = True
        print("🍎 Item collected!")

    self.step_count += 1
    if self.step_count >= self.max_steps or self.item_collected:
        self.terminated = True

def render(self):
    fig, ax = plt.subplots(figsize=(6, 6))
    ax.set_xlim(0, self.grid_size)
    ax.set_ylim(0, self.grid_size)
    ax.set_xticks(range(self.grid_size + 1))
    ax.set_yticks(range(self.grid_size + 1))
    ax.grid(True)
    ax.set_title(f"Step {self.step_count}", fontsize=14)

    if not self.item_collected:
        item_x, item_y = self.item_pos
        ax.add_patch(patches.Circle((item_y + 0.5, item_x + 0.5), 0.25, color='red'))
        ax.text(item_y + 0.5, item_x + 0.5, "🍎", ha='center', va='center', fontsize=16)

    for i, (x, y) in enumerate(self.agent_positions):
        ax.add_patch(patches.Rectangle((y + 0.1, x + 0.1), 0.8, 0.8,
                                       color=self.agent_colors[i]))
        ax.text(y + 0.5, x + 0.5, f"A{i}", color='white', weight='bold', ha='center', va='center')

    vis_left = max(y - self.vision, 0)
    vis_right = min(y + self.vision + 1, self.grid_size)
    vis_bottom = max(x - self.vision, 0)
    vis_top = min(x + self.vision + 1, self.grid_size)
    ax.add_patch(patches.Rectangle(
        (vis_left, vis_bottom),
        vis_right - vis_left,
        vis_top - vis_bottom,
        linewidth=0.5,
        edgecolor=self.agent_colors[i],
        facecolor=self.agent_colors[i],
        alpha=0.15
    ))

    msg = self.messages[i]
    if msg and msg != "None":
        ax.text(y + 0.5, x - 0.2, f"💬 {msg}", fontsize=10, ha='center', va='top', color=self.agent_colors[i])

    ax.set_aspect('equal')
    plt.gca().invert_yaxis()
    plt.tight_layout()
    plt.show()

action_options = ["UP", "DOWN", "LEFT", "RIGHT", "NOOP"]
message_options = ["None", "Going to item", "Need help", "I see item", "Target: (3,3)"]

agent0_action = widgets.Dropdown(options=action_options, description='A0 Act:')
agent0_message = widgets.Dropdown(options=message_options, description='A0 Msg:')
agent1_action = widgets.Dropdown(options=action_options, description='A1 Act:')
agent1_message = widgets.Dropdown(options=message_options, description='A1 Msg:')
step_button = widgets.Button(description="Step")

env = CommGridWorld()
env.render()

def on_step_clicked(b):
    clear_output(wait=True)
    env.step([agent0_action.value, agent1_action.value],

```


```

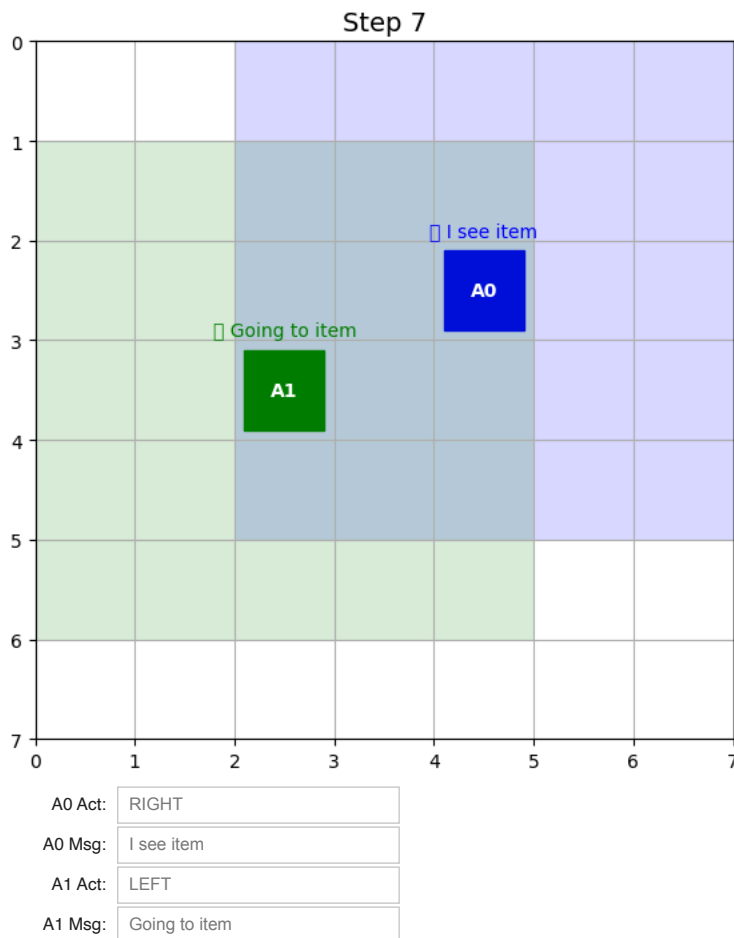
[agent0_message.value, agent1_message.value])
env.render()
display(agent0_action, agent0_message, agent1_action, agent1_message, step_button)

step_button.on_click(on_step_clicked)

display(agent0_action, agent0_message, agent1_action, agent1_message, step_button)

```

 <ipython-input-27-430ad5be29c8>:94: UserWarning: Glyph 128172 (\N{SPEECH BALLOON}) missing from font(s) DejaVu Sans.
plt.tight_layout()



✓ 5.7 Knowledge Assumptions in Multi-Agent Games

Understanding what agents **know** about their environment and one another is fundamental in both classical game theory and MARL. The assumptions made about agents' knowledge define the **epistemic structure** of the game and directly impact learning strategies, reasoning, and coordination.

5.7.1 Complete Knowledge Games

In **classical game theory**, it is typically assumed that agents have **complete knowledge** of the game they are playing. That is, each agent knows:

- The **action spaces** A_i of all agents
- The **reward functions** R_i of all agents
- For state-based games:
 - The full **state space** S
 - The **state transition function** T
 - (In POSGs) the **observation functions** O_i of all agents

This setup is referred to as a **complete knowledge game** (Owen, 2013).

Such knowledge enables agents to:

- Anticipate others' **best responses**
- Perform **multi-step planning** (when T is known)
- Use **policy reconstruction** and **equilibrium computation**

5.7.2 Incomplete Knowledge in MARL

In real-world MARL settings, **complete game knowledge is rarely available**.

Agents typically:

- **Do not know** their own reward function explicitly
- **Do not observe** the reward or actions of others
- **Cannot access** T , R_i , or O_i
- Only observe:
 - Their own reward r_i^t
 - The next state s^{t+1} (in SGs)
 - Or their partial observation o_i^{t+1} (in POSGs)

This places MARL settings under **incomplete information games** (Harsanyi, 1967).

To interact effectively, agents must **learn from experience** by:

- Estimating unknown game components (model-based learning)
- Inferring other agents' policies (agent modeling)
- Adapting to non-stationary behaviors (meta-learning)

5.7.3 From Simulator to Knowledge

Even when game dynamics are not known, MARL often assumes access to a **black-box simulator** T_b , which can be queried as:

$$(r, s') \sim T_b(s, a)$$

This enables empirical learning of transitions and rewards:

$$\Pr(T_b(s, a) = (r, s')) \approx T(s' \mid s, a), \quad R_i(s, a, s') = r_i$$

5.7.4 Symmetry, Asymmetry, and Common Knowledge

Knowledge assumptions can also vary across agents:

- **Symmetric knowledge**: All agents have the same knowledge.
- **Asymmetric knowledge**: Some agents have privileged information.
- **Common knowledge**: Not only does everyone know X , but everyone knows that everyone knows it — ad infinitum.

While these concepts are foundational in game theory and **epistemic logic**, they've seen limited use in MARL so far, except in:

- **Zero-sum games**: where knowledge of opponent payoff structure is exploited.
- **Common-reward games**: where a shared reward function can be assumed.

For example, knowing R_j may allow agent i to **simulate** agent j 's policy and respond optimally.

5.7.5 Open Multi-Agent Systems

Most game models assume:

- A **fixed number of agents** n
- That this number is **known to all agents**

However, emerging MARL research considers **open environments** where:

- Agents may **enter or leave** dynamically
- Systems must support **scalable, decentralized** reasoning

Such settings challenge the very idea of fixed game definitions and lead toward **lifelong, adaptive, and open-agent systems**.

✓ 5.8 Dictionary: Reinforcement Learning ↔ Game Theory

RL Term	GT Term	Description
environment	game	Model specifying the possible actions, observations, and rewards of agents, and the dynamics of how the state evolves.
agent	player	Entity which makes decisions. "Player" may also denote a specific role in the game (e.g., row player, white player).
reward	payoff, utility	Scalar value received by an agent/player after taking an action or upon outcome resolution.
policy	strategy	Function that maps observations (or histories) to a probability distribution over actions.
deterministic policy	pure strategy	Chooses a single action with probability 1.
probabilistic policy	mixed strategy	Assigns a probability distribution over possible actions.
joint policy	profile	Tuple of strategies or actions, one per agent/player.

5.7.2 Incomplete Knowledge in MARL

In real-world MARL settings, **complete game knowledge is rarely available**.

Agents typically:

- **Do not know** their own reward function explicitly
- **Do not observe** the reward or actions of others
- **Cannot access** T , R_i , or O_i
- Only observe:
 - Their own reward r_i^t
 - The next state s^{t+1} (in SGs)
 - Or their partial observation o_i^{t+1} (in POSGs)

This places MARL settings under **incomplete information games** (Harsanyi, 1967).

To interact effectively, agents must **learn from experience** by:

- Estimating unknown game components (model-based learning)
- Inferring other agents' policies (agent modeling)
- Adapting to non-stationary behaviors (meta-learning)

5.7.3 From Simulator to Knowledge

Even when game dynamics are not known, MARL often assumes access to a **black-box simulator** T_b , which can be queried as:

$$(r, s') \sim T_b(s, a)$$

This enables empirical learning of transitions and rewards:

$$\Pr(T_b(s, a) = (r, s')) \approx T(s' \mid s, a), \quad R_i(s, a, s') = r_i$$

5.7.4 Symmetry, Asymmetry, and Common Knowledge

Knowledge assumptions can also vary across agents:

- **Symmetric knowledge**: All agents have the same knowledge.
- **Asymmetric knowledge**: Some agents have privileged information.
- **Common knowledge**: Not only does everyone know X , but everyone knows that everyone knows it — ad infinitum.

While these concepts are foundational in game theory and **epistemic logic**, they've seen limited use in MARL so far, except in:

- **Zero-sum games**: where knowledge of opponent payoff structure is exploited.
- **Common-reward games**: where a shared reward function can be assumed.

For example, knowing R_j may allow agent i to **simulate** agent j 's policy and respond optimally.

5.7.5 Open Multi-Agent Systems

Most game models assume:

- A **fixed number of agents** n
- That this number is **known to all agents**

However, emerging MARL research considers **open environments** where:

- Agents may **enter or leave** dynamically
- Systems must support **scalable, decentralized** reasoning

Such settings challenge the very idea of fixed game definitions and lead toward **lifelong, adaptive, and open-agent systems**.

✓ **5.8 Dictionary: Reinforcement Learning ↔ Game Theory**

RL Term	GT Term	Descript
environment	game	Model specifying the possible actions, observations, and rewards of agents, an
agent	player	Entity which makes decisions. "Player" may also denote a specific role in the ga
reward	payoff, utility	Scalar value received by an agent/player after taking an action or upon outcom
policy	strategy	Function that maps observations (or histories) to a probability distribution over
deterministic policy	pure strategy	Chooses a single action with probability 1.
probabilistic policy	mixed strategy	Assigns a probability distribution over possible actions.
joint policy	profile	Tuple of strategies or actions, one per agent/player.
joint reward	payoff profile	Rewards (payoffs) received by all agents for a joint action.
deterministic joint policy	pure strategy profile	All agents choose actions with probability 1.
stochastic joint policy	mixed strategy profile	All agents may act probabilistically.

Licensed under [CC BY-NC-ND 4.0](#). © Tamás Takács, 2025.

✓ 6. Practice - Solution Concepts for Games

👤 Tamás Takács, PhD student, Department of Artificial Intelligence

🕒 90 min read

📅 January 22, 2025

🏆 Collective Intelligence



ELTE | IK
INFORMATIKAI KAR



DEPARTMENT
OF ARTIFICIAL
INTELLIGENCE



Game Theory

['gām 'thē-ə-rē]

A theoretical framework
for conceiving social
situations among
competing players.

In previous practices, we explored models that characterize how agents interact and make decisions in shared environments. This notebook addresses a deeper and fundamental question:

What does it mean for agents to act optimally in these settings?

Game theory provides a rigorous answer through solution concepts—formal criteria that specify which joint behaviors (policies) are stable, rational, or desirable in multi-agent systems. These concepts, including best response, Nash equilibrium, minimax, correlated equilibrium, and others, serve as benchmarks for evaluating and designing learning algorithms in multi-agent reinforcement learning (MARL).

Through analytical explanations, worked examples, and simulation exercises, this practice guides students in understanding, applying, and comparing key solution concepts. By the end of this practice, students will be able to identify the appropriate solution concepts for various game models and critically assess the strengths and limitations of each within the context of multi-agent learning and decision making.

A MARL problem can be fully specified as the combination of:

- A **game model** (e.g., normal-form game, stochastic game, POSG), and
- A **solution concept** (e.g., Nash equilibrium, social welfare optimum, no-regret learning)

Table of Contents

- 6.1 Game Model + Solution Concept = MARL Problem
 - 6.1.1 Necessary Imports

- **6.2 Joint Policy and Expected Return**
 - 6.2.1 Histories and Observation Function
 - 6.2.2 Expected Return via History Enumeration
 - 6.2.3 Recursive Expected Return (Bellman Form)
 - 6.2.4 POSG Environment Assumptions
 - 6.2.5 Simulation Approach
- **6.3 Best Response**
 - 6.3.1 Best Response in Matrix (Normal-Form) Games
 - 6.3.2 Applications of Best Response
 - 6.3.3 Multiple Best Responses
- **6.4 Minimax**
 - 6.4.1 Definition (Minimax Solution)
 - 6.4.2 Interpretation
 - 6.4.3 Best Response Interpretation
 - 6.4.4 Example: Rock-Paper-Scissors
- **6.5 Minimax Solution via Linear Programming**
 - 6.5.1 Linear Program for Agent
 - 6.5.2 Example: Rock-Paper-Scissors
- **6.6 Nash Equilibrium**
 - 6.6.1 Definition
 - 6.6.2 Properties
 - 6.6.3 Examples
 - 6.6.4 Check for Nash Equilibrium
 - 6.6.5 Computing Equilibria
- **6.7 ϵ -Nash Equilibrium**
 - 6.7.1 Definition
 - 6.7.2 Important Notes
 - 6.7.3 Example: Coordination Game
 - 6.7.4 Checking ϵ -Nash Equilibria
- **6.8 Correlated and Coarse Correlated Equilibrium**
 - 6.8.1 Correlated Equilibrium (CE)
 - 6.8.2 Example: Chicken Game
 - 6.8.3 Coarse Correlated Equilibrium (CCE)
 - 6.8.4 Hierarchy
 - 6.8.5 Extension to Sequential Games
- **6.9 Conceptual Limitations of Equilibrium Solutions**
 - 6.9.1 Sub-optimality
 - 6.9.2 Non-uniqueness
 - 6.9.3 Incompleteness
- **6.10 Pareto Optimality**
 - 6.10.1 Definition
 - 6.10.2 Intuition
 - 6.10.3 Example: Chicken Game (Matrix Form)
 - 6.10.4 Pareto Frontier Visualization

- **6.11 Social Welfare and Fairness**

- 6.11.1 Social Welfare
- 6.11.2 Social Fairness (Nash Social Welfare)
- 6.11.3 Example: Battle of the Sexes
- 6.11.4 Visualization: Welfare and Fairness

- **6.12 No-Regret**

- 6.12.1 Definition: Regret
- 6.12.2 No-Regret Condition
- 6.12.3 Example: Prisoner's Dilemma (Empirical Regret)
- 6.12.4 No-Regret in General Game Models
- 6.12.5 Limitations
- 6.12.6 Relation to Equilibria
- 6.12.7 Summary Table

✓ **6.1 Game Model + Solution Concept = MARL Problem**

The game model defines the mechanics of multi-agent interaction. The solution concept defines the desired outcome - i.e., what the agents are trying to achieve through learning.

For example:

- In **common-reward games**, a natural solution concept is to **maximize expected return** across all agents.
- In **general-sum games**, the solution becomes more nuanced, and different concepts (e.g., Nash equilibrium, Pareto optimality) offer different perspectives.

✓ **6.1.1 Necessary Imports**

```
!pip install nashpy
```



Collecting nashpy

Downloading nashpy-0.0.41-py3-none-any.whl.metadata (6.6 kB)

Requirement already satisfied: numpy>=1.21.0 in /usr/local/lib/python3.11/dist-packages (from nashpy)

Requirement already satisfied: scipy>=0.19.0 in /usr/local/lib/python3.11/dist-packages (from nashpy)

Requirement already satisfied: networkx>=3.0.0 in /usr/local/lib/python3.11/dist-packages (from nashpy)

Requirement already satisfied: deprecated>=1.2.14 in /usr/local/lib/python3.11/dist-packages (from nashpy)

Requirement already satisfied: wrapt<2,>=1.10 in /usr/local/lib/python3.11/dist-packages (from nashpy)

Downloading nashpy-0.0.41-py3-none-any.whl (27 kB)

Installing collected packages: nashpy

Successfully installed nashpy-0.0.41

```
import matplotlib.pyplot as plt
import matplotlib.animation as animation
import matplotlib.patches as patches
import numpy as np
import random
from IPython.display import HTML, display, clear_output
import ipywidgets as widgets
from collections import defaultdict
from scipy.optimize import linprog
import nashpy as nash
```

✓ **6.2 Joint Policy and Expected Return**

A solution to a multi-agent game is defined by a **joint policy**:

$$\pi = (\pi_1, \dots, \pi_n)$$

Each agent π_i selects actions based on its observations and history. To evaluate the performance of such a joint policy, we define the **expected return**:

$$U_i(\pi)$$

This quantifies the utility that agent i expects to receive when all agents act according to π .

To make this definition general, we define $U_i(\pi)$ within the most expressive model used in this book – the **Partially Observable Stochastic Game (POSG)** – which subsumes stochastic games and normal-form games.

6.2.1 Histories and Observation Function

Let $\hat{h}_t = \{(s_\tau, o_\tau, a_\tau)_{\tau=0}^{t-1}, s_t, o_t\}$ be the full history up to time t .

Define $\sigma(\hat{h}_t) = (o_0, \dots, o_t)$ as the observation history. The joint observation probability is:

$$O(o_t \mid a_{t-1}, s_t) = \prod_{i \in I} O_i(o_t^i \mid a_{t-1}, s_t)$$

We assume **discounted returns** with absorbing terminal states.

6.2.2 Expected Return via History Enumeration

Let H be the set of all full histories. The expected return is:

$$U_i(\pi) = \sum_{\hat{h}_t \in H} \Pr(\hat{h}_t \mid \pi) \cdot u_i(\hat{h}_t)$$

Where:

- $\Pr(\hat{h}_t \mid \pi)$ is the probability of history under the model
- $u_i(\hat{h}_t) = \sum_{\tau=0}^{t-1} \gamma^\tau R_i(s_\tau, a_\tau, s_{\tau+1})$ is the discounted reward

If agents act independently, the joint action probability factorizes:

$$\pi(a_\tau \mid h_\tau) = \prod_{j \in I} \pi_j(a_\tau^j \mid h_\tau^j)$$

6.2.3 Recursive Expected Return (Bellman Form)

We define two value functions:

Value function:

$$V_i^\pi(\hat{h}) = \sum_{a \in A} \pi(a \mid \sigma(\hat{h})) Q_i^\pi(\hat{h}, a)$$

Action-value function:

$$Q_i^\pi(\hat{h}, a) = \sum_{s'} T(s' \mid s(\hat{h}), a) \left[R_i(s(\hat{h}), a, s') + \gamma \sum_{o'} O(o' \mid a, s') V_i^\pi(\langle \hat{h}, a, s', o' \rangle) \right]$$

The overall expected return becomes:

$$U_i(\pi) = \mathbb{E}_{s_0 \sim \mu, o_0 \sim O(\cdot \mid \emptyset, s_0)} [V_i^\pi(\langle s_0, o_0 \rangle)]$$

This recursive form forms the basis for learning algorithms like value iteration and actor-critic methods in MARL.

6.2.4 POSG Environment Assumptions

- Finite state space S
- Per-agent action sets A_i and observation sets O_i
- Observation function: $O_i(o_i | a_{t-1}, s_t)$
- Transition function: $T(s' | s, a)$
- Reward function: $R_i(s, a, s')$
- Policies: $\pi_i(a_i | h_i^t)$ depend on the observation history $h_i^t = (o_i^0, \dots, o_i^t)$

✓ 6.2.5 Simulation Approach

We simulate multiple episodes (trajectories), each consisting of:

1. Sampling initial state $s_0 \sim \mu$
2. Sampling observations $o_i^0 \sim O_i(\cdot | \emptyset, s_0)$
3. Iteratively choosing actions using policies $\pi_i(a_i | h_i^t)$
4. Sampling next states, rewards, and observations
5. Accumulating discounted rewards per agent

Finally, we compute the average return $U_i(\pi)$ over episodes.

```
S = ['s0', 's1', 's2']
A1 = ['a1', 'a2']
A2 = ['b1', 'b2']
O1 = ['x', 'y']
O2 = ['u', 'v']
gamma = 0.95

T = {
    ('s0', ('a1', 'b1')): {'s1': 0.7, 's2': 0.3},
    ('s0', ('a2', 'b1')): {'s1': 1.0},
    ('s1', ('a1', 'b2')): {'s2': 1.0},
    ('s2', ('a2', 'b2')): {'s0': 1.0},
}
default_T = lambda s, a: {s: 1.0}

R1 = defaultdict(lambda: 0, {
    ('s0', ('a1', 'b1'), 's1'): 1,
    ('s0', ('a2', 'b1'), 's1'): 0.5,
    ('s1', ('a1', 'b2'), 's2'): 2,
})
R2 = defaultdict(lambda: 0, {
    ('s0', ('a1', 'b1'), 's1'): 1,
    ('s0', ('a2', 'b1'), 's1'): 0,
    ('s1', ('a1', 'b2'), 's2'): 3,
})

def obs_fn_i(agent, s, a_prev):
    if agent == 1:
        return np.random.choice(O1, p=[0.8, 0.2]) if s == 's0' else np.random.choice(O1, p=[0.5, 0.5])
    else:
        return np.random.choice(O2, p=[0.6, 0.4]) if s == 's1' else np.random.choice(O2, p=[0.3, 0.7])

def policy1(history):
    return 'a1' if history[-1] == 'x' else 'a2'

def policy2(history):
    return 'b1' if history[-1] == 'u' else 'b2'

def sample_initial_state():
    return 's0'
```

```

def run_episode(max_steps=10):
    s = sample_initial_state()
    h1, h2 = [], []
    total_r1, total_r2 = 0, 0
    gamma_t = 1.0
    a_prev = None

    for t in range(max_steps):
        o1 = obs_fn_i(1, s, a_prev)
        o2 = obs_fn_i(2, s, a_prev)
        h1.append(o1)
        h2.append(o2)

        a1 = policy1(h1)
        a2 = policy2(h2)
        joint_a = (a1, a2)

        trans_probs = T.get((s, joint_a), default_T(s, joint_a))
        s_next = random.choices(list(trans_probs.keys()), list(trans_probs.values()))[0]

        r1 = R1[(s, joint_a, s_next)]
        r2 = R2[(s, joint_a, s_next)]
        total_r1 += gamma_t * r1
        total_r2 += gamma_t * r2

        gamma_t *= gamma
        s = s_next
        a_prev = joint_a

    return total_r1, total_r2

def estimate_expected_returns(n_episodes=1000):
    rewards1, rewards2 = [], []
    for _ in range(n_episodes):
        r1, r2 = run_episode()
        rewards1.append(r1)
        rewards2.append(r2)
    return np.mean(rewards1), np.mean(rewards2)

U1_pi, U2_pi = estimate_expected_returns(1000)
print(f"Estimated U1(π): {U1_pi:.3f}")
print(f"Estimated U2(π): {U2_pi:.3f}")

```

➡ Estimated U1(π): 1.768
Estimated U2(π): 2.156

✓ 6.3 Best Response

In multi-agent games, the **best response** defines the optimal behavior for a single agent assuming that the strategies of all **other agents** are fixed.

Let the joint policy be denoted as:

$$\pi = (\pi_1, \dots, \pi_i, \dots, \pi_n)$$

Then, the set of policies for all agents except agent i is:

$$\pi_{-i} = (\pi_1, \dots, \pi_{i-1}, \pi_{i+1}, \dots, \pi_n)$$

The **best response** of agent i to π_{-i} is the policy π_i that **maximizes** the expected return of agent i under the joint policy:

$$\text{BR}_i(\pi_{-i}) = \arg \max_{\pi_i} U_i(\pi_i, \pi_{-i})$$

6.3.1 Best Response in Matrix (Normal-Form) Games

In normal-form games, the best response often reduces to picking the highest-reward action based on known or observed opponent strategies.

For example, in **Rock-Paper-Scissors**, if agent 2 always plays *Rock*, then agent 1's best response is to always play *Paper*.

6.3.2 Applications of Best Response

Best responses are central to many game-theoretic and learning algorithms:

- **Nash Equilibrium**: Every agent plays a best response to the others
- **Fictitious Play**: Agents iteratively respond to the empirical action history
- **Policy Optimization**: Many MARL algorithms aim to approximate a best response under partial information

✓ 6.3.3 Multiple Best Responses

Note: there may be multiple best-response policies. For instance, if two or more actions yield the same expected return, then any stochastic mixture over them is also a valid best response.

```
payoff_matrix = np.array([
    [0, -1, 1], # Rock
    [1, 0, -1], # Paper
    [-1, 1, 0]  # Scissors
])

actions = ['Rock', 'Paper', 'Scissors']

pi_2 = np.array([0.5, 0.25, 0.25])

expected_returns = payoff_matrix @ pi_2

for i, action in enumerate(actions):
    print(f"Expected return of playing {action}: {expected_returns[i]:.2f}")

best_indices = np.argwhere(expected_returns == np.max(expected_returns)).flatten()
best_responses = [actions[i] for i in best_indices]
print(f"\nBest response(s): {best_responses}")
```

↗ Expected return of playing Rock: 0.00
Expected return of playing Paper: 0.25
Expected return of playing Scissors: -0.25

Best response(s): ['Paper']

✓ 6.4 Minimax

The **minimax** solution concept applies to **two-agent zero-sum games**, where one agent's gain is exactly the other's loss. A canonical example is the **Rock-Paper-Scissors** game.

In such games, agents aim to **maximize their expected return** while assuming that their opponent is acting to **minimize** it. The result is a strategy that performs best against the **worst-case opponent**.

6.4.1 Definition (Minimax Solution)

Let agent i and agent j be players in a two-agent zero-sum game. A **minimax solution** is a joint policy $\pi = (\pi_i, \pi_j)$ that satisfies:

$$U_i(\pi) = \max_{\pi'_i} \min_{\pi'_j} U_i(\pi'_i, \pi'_j) = \min_{\pi'_j} \max_{\pi'_i} U_i(\pi'_i, \pi'_j)$$

Where:

- $U_i(\pi)$ is the expected return for agent i under joint policy π
- $U_j(\pi) = -U_i(\pi)$ (by zero-sum assumption)

6.4.2 Interpretation

- **Maxmin:** Agent i chooses a policy that maximizes the minimum expected return they can guarantee, no matter what the opponent does.
- **Minmax:** Agent j chooses a policy that minimizes the maximum payoff agent i can achieve.

In a **minimax solution**, these two values are equal. Neither agent can benefit by unilaterally changing their policy.

6.4.3 Best Response Interpretation

A joint policy (π_i, π_j) is a minimax solution if and only if:

- $\pi_i \in \text{BR}_i(\pi_j)$
- $\pi_j \in \text{BR}_j(\pi_i)$

That is, both are best responses to each other.

✓ 6.4.4 Example: Rock-Paper-Scissors

In Rock-Paper-Scissors, the minimax solution is for **both agents to play uniformly at random**:

$$\pi_i = \pi_j = \left[\frac{1}{3}, \frac{1}{3}, \frac{1}{3}\right]$$

This yields an **expected return of 0** for both agents. Any deviation from the uniform strategy can be exploited by the opponent.

```

payoff_matrix = np.array([
    [0, -1, 1], # Rock
    [1, 0, -1], # Paper
    [-1, 1, 0]  # Scissors
])

uniform_policy = np.array([1/3, 1/3, 1/3])

expected_return = uniform_policy @ payoff_matrix @ uniform_policy.T

print(f"Expected return for agent 1 (uniform vs uniform): {expected_return:.2f}")

actions = ['Rock', 'Paper', 'Scissors']
for i, action in enumerate(actions):
    opponent_policy = np.zeros(3)
    opponent_policy[i] = 1.0
    reward = uniform_policy @ payoff_matrix @ opponent_policy
    print(f"Agent 1 expected return vs Agent 2 always playing {action}: {reward:.2f}")

➡ Expected return for agent 1 (uniform vs uniform): 0.00
Agent 1 expected return vs Agent 2 always playing Rock: 0.00
Agent 1 expected return vs Agent 2 always playing Paper: 0.00
Agent 1 expected return vs Agent 2 always playing Scissors: 0.00

```

✓ 6.5 Minimax Solution via Linear Programming

For **non-repeated two-agent zero-sum normal-form games**, the **minimax solution** can be found by solving a **linear program (LP)**. Each agent solves a linear program that **minimizes the opponent's expected reward** under the assumption of worst-case behavior.

6.5.1 Linear Program for Agent i

Let agent j be the opponent. Agent i solves the following LP:

Variables:

- x_{a_i} : Probability of selecting action $a_i \in A_i$
- U_j^* : The **expected return of agent j** , which is minimized

Objective:

$$\min_x U_j^*$$

Constraints:

1. For each opponent action $a_j \in A_j$:

$$\sum_{a_i \in A_i} R_j(a_i, a_j) \cdot x_{a_i} \leq U_j^*$$

This ensures no pure strategy of the opponent yields more than U_j^*

2. Valid probability distribution:

$$x_{a_i} \geq 0 \quad \forall a_i \in A_i$$
$$\sum_{a_i \in A_i} x_{a_i} = 1$$

This process yields the **minimax policy** for agent i , and a similar LP can be constructed for agent j by swapping indices.

✓ 6.5.2 Example: Rock-Paper-Scissors

We solve the minimax policy for the **row player (agent 1)** using the **column player's reward matrix** $R_2 = -R_1$.

```
R1 = np.array([
    [0, -1, 1],
    [1, 0, -1],
    [-1, 1, 0]
])

R2 = -R1
n_actions = R1.shape[0]
c = [0] * n_actions + [1]

A_ub = []
b_ub = []

for col in range(n_actions):
    constraint = list(R2[:, col]) + [-1]
    A_ub.append(constraint)
    b_ub.append(0)

A_eq = [[1] * n_actions + [0]]
b_eq = [1]

bounds = [(0, 1)] * n_actions + [(None, None)]
result = linprog(c=c, A_ub=A_ub, b_ub=b_ub, A_eq=A_eq, b_eq=b_eq, bounds=bounds, method='highs')
```

```

if result.success:
    policy = result.x[:-1]
    game_value = result.x[-1]
    print("Minimax Policy for Agent 1 (Row Player):", np.round(policy, 4))
    print("Game Value (Expected reward to Agent 2):", round(game_value, 4))
else:
    print("Linear program failed:", result.message)

```

➡ Minimax Policy for Agent 1 (Row Player): [0.3333 0.3333 0.3333]
Game Value (Expected reward to Agent 2): -0.0

✓ 6.6 Nash Equilibrium

The **Nash equilibrium** is a foundational solution concept in game theory. It defines a **joint policy** in which no agent can unilaterally change their strategy to increase their expected return, assuming the policies of others remain fixed.

6.6.1 Definition

A joint policy $\pi = (\pi_1, \dots, \pi_n)$ is a **Nash equilibrium** if:

$$\forall i, \pi'_i \quad U_i(\pi'_i, \pi_{-i}) \leq U_i(\pi)$$

In other words, for every agent i , the policy π_i is a **best response** to the policies of all other agents π_{-i} .

6.6.2 Properties

- Every **finite** normal-form game has at least one **Nash equilibrium** (Nash, 1950).
- A Nash equilibrium can be **deterministic (pure)** or **probabilistic (mixed)**.
- **Generalization**: Minimax equilibria in zero-sum games are special cases of Nash equilibria.
- **Multiple equilibria** may exist, each yielding **different expected returns**.

6.6.3 Examples

- **Prisoner's Dilemma**:
 - Only equilibrium: Both defect
- **Coordination Game**:
 - Three equilibria: (A, A), (B, B), or 50-50 mixed
- **Rock-Paper-Scissors**:
 - Only equilibrium: Each action with probability $\frac{1}{3}$

6.6.4 Check for Nash Equilibrium

To verify if a given joint policy π is a Nash equilibrium:

1. Fix the policies π_{-i} of all other agents.
2. Compute the best response π'_i for agent i .
3. Check whether $U_i(\pi'_i, \pi_{-i}) \leq U_i(\pi)$.

If this is true for all agents i , then π is a Nash equilibrium.

✓ 6.6.5 Computing Equilibria

Nash equilibria can be computed using:

- **Support enumeration** (for small matrix games)
- **Lemke-Howson algorithm** (for 2-player games)
- **Linear programming** (in zero-sum cases)
- **Reinforcement learning** (for sequential/stochastic settings)

```
A = np.array([
    [10, 0],
    [0, 10]
])

B = np.array([
    [10, 0],
    [0, 10]
])

game = nash.Game(A, B)

equilibria = list(game.support_enumeration())

print("Nash Equilibria (mixed strategies):")
for eq in equilibria:
    print(f"Agent 1: {np.round(eq[0], 3)} \t Agent 2: {np.round(eq[1], 3)}")

σ1, σ2 = equilibria[0]
u1 = game[σ1, σ2][0]
u2 = game[σ1, σ2][1]
print(f"\nExpected payoff: Agent 1 = {u1}, Agent 2 = {u2}")
```

⇒ Nash Equilibria (mixed strategies):

Agent 1: [1. 0.]	Agent 2: [1. 0.]
Agent 1: [0. 1.]	Agent 2: [0. 1.]
Agent 1: [0.5 0.5]	Agent 2: [0.5 0.5]

Expected payoff: Agent 1 = 10.0, Agent 2 = 10.0

✓ 6.7 ϵ -Nash Equilibrium

A strict **Nash equilibrium** requires that **no agent can unilaterally improve** their expected return by deviating from their strategy.

However, in practical settings, this condition may be:

- **Impossible to represent:** Exact equilibria can require **irrational numbers**
- **Too costly:** Computing strict equilibria is often **computationally expensive**

To address this, we consider the ϵ -**Nash equilibrium**, which relaxes the definition by tolerating **small deviations**.

6.7.1 Definition

A joint policy $\pi = (\pi_1, \dots, \pi_n)$ is an ϵ -**Nash equilibrium** if for all agents i and all alternative strategies π'_i :

$$U_i(\pi'_i, \pi_{-i}) - \epsilon \leq U_i(\pi)$$

This means no agent can gain **more than ϵ** in expected return by deviating.

- When $\epsilon = 0$, we recover the classic **Nash equilibrium**.
- For $\epsilon > 0$, the agents are **almost optimal**, but minor improvements are possible.

6.7.2 Important Notes

- An ϵ -Nash equilibrium is **not necessarily close** to a real Nash equilibrium in terms of expected return.
- There may be **no nearby exact Nash equilibrium**, even when ϵ is small.
- These equilibria are useful for **approximations**, especially in large or continuous games.

6.7.3 Example: Coordination Game

Consider this matrix game:

	C	D
A	100,100	0,0
B	1,2	1,1

- The **only Nash equilibrium** is (A, C), with high rewards.
- But (B, D) is a **1-Nash equilibrium**: agent 2 could deviate to C and gain **at most 1**.

Thus, **(B, D)** satisfies the relaxed condition but is **not close** to the actual equilibrium in payoff terms.

✓ 6.7.4 Checking ϵ -Nash Equilibria

To check whether a joint policy π is an ϵ -Nash equilibrium:

1. Fix all other agents' policies: π_{-i}
2. Compute agent i 's best response π'_i
3. Check if:

$$U_i(\pi'_i, \pi_{-i}) - \epsilon \leq U_i(\pi)$$

If true for all agents i , then π is an ϵ -**Nash equilibrium**.

```
A = np.array([
    [100, 0], # A
    [1, 1]   # B
])

B = np.array([
    [100, 0], # A
    [2, 1]   # B
])

epsilon = 1.0

row_strategy = np.array([0.0, 1.0])
col_strategy = np.array([0.0, 1.0])

u1 = row_strategy @ A @ col_strategy.T
u2 = row_strategy @ B @ col_strategy.T

best_response_row = np.argmax(A @ col_strategy.T)
best_response_col = np.argmax(B.T @ row_strategy.T)

br1_util = A[best_response_row] @ col_strategy.T
br2_util = row_strategy @ B[:, best_response_col]

print(f"Agent 1: u1 = {u1}, BR = {br1_util}")
print(f"Agent 2: u2 = {u2}, BR = {br2_util}")

is_epsilon_nash = (br1_util - u1 <= epsilon) and (br2_util - u2 <= epsilon)

print(f"\nIs (B, D) a {epsilon}-Nash equilibrium? {'Yes' if is_epsilon_nash else 'No'}")

➡ Agent 1: u1 = 1.0, BR = 1.0
Agent 2: u2 = 1.0, BR = 2.0
```

Is (B, D) a 1.0-Nash equilibrium? Yes

✓ 6.8 Correlated and Coarse Correlated Equilibrium

A **Nash equilibrium** requires that agents act **independently**. However, agents may achieve **better coordination and higher rewards** by **correlating their actions**.

Correlated equilibrium (Aumann, 1974) extends Nash equilibrium by allowing **action recommendations** drawn from a **joint distribution**. Agents are then expected to follow their **recommended action**, assuming others will do the same.

6.8.1 Correlated Equilibrium (CE)

Let $\pi_c(a)$ be a **joint policy** over actions $a \in A$. Then, π_c is a **correlated equilibrium** if for every agent $i \in I$ and every **action modification function** $\xi_i : A_i \rightarrow A_i$:

$$\sum_{a \in A} \pi_c(a) R_i(\xi_i(a_i), a_{-i}) \leq \sum_{a \in A} \pi_c(a) R_i(a)$$

Interpretation:

- Each agent **knows its own recommendation** a_i from π_c .
- It assumes others **follow their recommendations** a_{-i} .
- Agent i checks if it could improve by deviating to another action $\xi_i(a_i)$.
- If **no beneficial deviation exists**, it's a CE.

6.8.2 Example: Chicken Game

	S	L
S	0,0	7,2
L	2,7	6,6

Joint policy:

- $\pi_c(S, L) = \pi_c(L, S) = \pi_c(L, L) = \frac{1}{3}$
- $\pi_c(S, S) = 0$

Expected return for both agents:

$$7 \cdot \frac{1}{3} + 2 \cdot \frac{1}{3} + 6 \cdot \frac{1}{3} = 5$$

Neither agent benefits from deviating, so this is a **correlated equilibrium**.

6.8.3 Coarse Correlated Equilibrium (CCE)

CCE is a weaker version of CE. The difference:

- In CE: agent sees its **recommended action** and considers deviations.
- In CCE: agent **decides whether to follow π_c or use a fixed action, before** seeing its recommendation.

Formally, π_c is a **CCE** if for all i and constant actions a'_i :

$$\sum_{a \in A} \pi_c(a) R_i(a) \geq \sum_{a \in A} \pi_c(a) R_i(a'_i, a_{-i})$$

6.8.4 Hierarchy

- **Nash equilibrium** \subseteq **Correlated equilibrium** \subseteq **Coarse correlated equilibrium**

Correlated equilibrium allows **shared signals** (e.g., traffic light coordination). Coarse correlated equilibrium only requires agents to **commit ahead of time**.

✓ 6.8.5 Extension to Sequential Games

In **sequential settings**, CE definitions can:

- Reveal **entire policies** or **partial instructions**
- Depend on **state, observations, or history**
- Include **commitment mechanisms**
- Handle **deviations** with penalties or policy revocation

These more advanced CE forms appear in **Dec-POMDPs** and advanced **multi-agent planning**.

```
R1 = np.array([
    [0, 7], # Agent 1: S, L
    [2, 6]
])

R2 = np.array([
    [0, 2], # Agent 2: S, L
    [7, 6]
])

pi_c = np.array([
    [0.0, 1/3], # (S, S), (S, L)
    [1/3, 1/3] # (L, S), (L, L)
])

U1 = np.sum(pi_c * R1)
U2 = np.sum(pi_c * R2)

p_S = pi_c[1, 0] / (pi_c[1, 0] + pi_c[1, 1])
p_L = pi_c[1, 1] / (pi_c[1, 0] + pi_c[1, 1])

L_payoff = 2 * p_S + 6 * p_L
S_payoff = 0 * p_S + 7 * p_L

print(f"Expected payoff for agent 1 if following L: {L_payoff}")
print(f"Expected payoff if deviating to S instead: {S_payoff}")
print(f"Correlated equilibrium condition satisfied? {L_payoff >= S_payoff}")

↔ Expected payoff for agent 1 if following L: 4.0
Expected payoff if deviating to S instead: 3.5
Correlated equilibrium condition satisfied? True
```

✓ 6.9 Conceptual Limitations of Equilibrium Solutions

While equilibrium-based approaches—particularly **Nash equilibrium**—are widely used in multi-agent reinforcement learning (MARL), they come with important **conceptual limitations** that affect both their applicability and performance in practical settings.

6.9.1 Sub-optimality

An equilibrium solution only ensures that **each agent is playing a best response** to the others' strategies. It does **not guarantee** that the **joint outcome** is **optimal** in terms of expected return.

- **Example: Prisoner's Dilemma**
 - The Nash equilibrium is (D, D) with returns -3 for both agents.

- The joint action (C, C) gives -1 each, which is strictly better.
- However, (C, C) is not a Nash equilibrium because each agent has an incentive to deviate.

- **Example: Chicken Game**

- The correlated equilibrium yields a return of **5** to each agent.
- The joint action (L, L) gives **6** to both agents.
- However, (L, L) is **not** a correlated or Nash equilibrium.

6.9.2 Non-uniqueness

Equilibrium solutions are often **not unique**—multiple equilibria can exist with **varying implications** for agents' payoffs.

- In the **Chicken game**, there are:
 - Two **deterministic** Nash equilibria (asymmetric payoffs: (7, 2) and (2, 7))
 - One **probabilistic** equilibrium with equal payoffs ($\approx 4.66, 4.66$)
 - A correlated equilibrium with payoffs (5, 5)

This raises the question:

Which equilibrium should agents adopt?

This is known as the **equilibrium selection problem**.

- In self-play MARL, agents learn **simultaneously** and often **converge to different equilibria** depending on initialization, learning rates, or update rules.
- Equilibrium selection is a major topic in **game theory**, and in MARL it often requires:
 - Coordination strategies
 - Communication protocols
 - Social welfare metrics (see Sections 4.8 & 4.9)

6.9.3 Incompleteness

In **sequential games**, a joint policy π does **not define behavior** on paths that are **off-equilibrium**, i.e., paths that occur with zero probability under π .

- Such paths may still **arise due to noise, exploration, or unexpected deviations**.
- Equilibrium solutions provide **no guidance** for returning to the equilibrium trajectory.
- This problem is especially acute in **long-horizon or continuous control** tasks.

To resolve this, game theory offers **refinements**:

- **Subgame Perfect Equilibrium**: Ensures optimal behavior in

✓ 6.10 Pareto Optimality

In multi-agent settings, equilibrium solutions may not be *socially desirable*. For instance, agents may settle in a stable joint policy (equilibrium) that yields poor returns for all involved. **Pareto optimality** is a powerful concept used to **refine** such equilibria by eliminating joint policies that could be improved for at least one agent **without harming others**.

6.10.1 Definition

A joint policy π is **Pareto-dominated** by another joint policy π' if:

- $\forall i: U_i(\pi') \geq U_i(\pi)$
- $\exists i: U_i(\pi') > U_i(\pi)$

A joint policy is **Pareto-optimal** if it is **not** Pareto-dominated by any other policy.

6.10.2 Intuition

- **Pareto-optimal** policies **maximize returns for some agents** without reducing the returns of others.
- Every game has **at least one** Pareto-optimal policy.
- In **common-reward games**, all Pareto-optimal policies yield the **same** maximum return.
- In **zero-sum games**, all joint policies are Pareto-optimal by construction (since improving one agent necessarily worsens the other).

6.10.3 Example: Chicken Game (Matrix Form)

	S (Stay)	L (Leave)
S	(0, 0)	(7, 2)
L	(2, 7)	(6, 6)

6.10.4 Pareto Frontier Visualization

We now visualize the **space of feasible expected joint rewards** and the **Pareto frontier**.

```
payoffs = {
    ("S", "S"): (0, 0),
    ("S", "L"): (7, 2),
    ("L", "S"): (2, 7),
    ("L", "L"): (6, 6),
}
actions = ["S", "L"]

steps = 30
policy_range = np.linspace(0, 1, steps)
joint_rewards = []

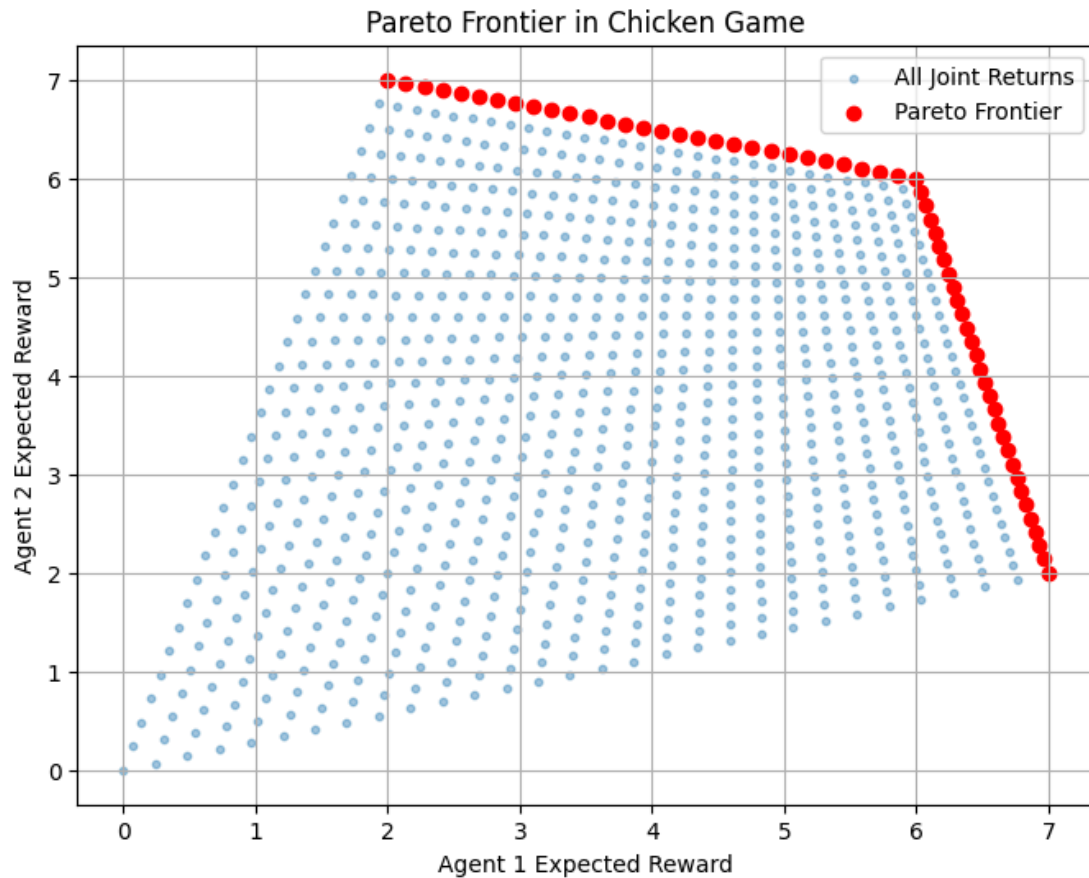
for p1 in policy_range:
    for p2 in policy_range:
        pi1 = {"S": p1, "L": 1 - p1}
        pi2 = {"S": p2, "L": 1 - p2}
        expected_r1 = 0
        expected_r2 = 0
        for a1 in actions:
            for a2 in actions:
                prob = pi1[a1] * pi2[a2]
                r1, r2 = payoffs[(a1, a2)]
                expected_r1 += prob * r1
                expected_r2 += prob * r2
        joint_rewards.append((expected_r1, expected_r2))

def is_pareto_efficient(points):
    points = np.array(points)
    is_efficient = np.ones(points.shape[0], dtype=bool)
    for i, c in enumerate(points):
        if is_efficient[i]:
            is_efficient[is_efficient] = np.any(points[is_efficient] > c, axis=1)
            is_efficient[i] = True
    return is_efficient

joint_rewards = np.array(joint_rewards)
pareto_mask = is_pareto_efficient(joint_rewards)

plt.figure(figsize=(8, 6))
plt.scatter(joint_rewards[:, 0], joint_rewards[:, 1], s=10, alpha=0.4, label="All Joint Returns")
plt.scatter(joint_rewards[pareto_mask][:, 0], joint_rewards[pareto_mask][:, 1], color="red", label="")
```

```
plt.xlabel("Agent 1 Expected Reward")
plt.ylabel("Agent 2 Expected Reward")
plt.title("Pareto Frontier in Chicken Game")
plt.legend()
plt.grid(True)
plt.show()
```



✓ 6.11 Social Welfare and Fairness

Pareto optimality ensures no agent can be made better off without making another worse off. However, it does not account for **how well off** the agents are **together**, or how **evenly** their rewards are distributed. This motivates **social welfare** and **fairness** as additional solution criteria in general-sum games.

6.11.1 Social Welfare

The **social welfare** of a joint policy π is the **sum of all agents' expected returns**:

$$W(\pi) = \sum_{i \in I} U_i(\pi)$$

A policy π is **welfare-optimal** if:

$$\pi \in \arg \max_{\pi'} W(\pi')$$

6.11.2 Social Fairness (Nash Social Welfare)

The **fairness** of a joint policy π is the **product of all agents' expected returns**:

$$F(\pi) = \prod_{i \in I} U_i(\pi)$$

A policy π is **fairness-optimal** if:

$$\pi \in \arg \max_{\pi'} F(\pi')$$

This formulation promotes **equity**: maximizing fairness encourages agents to have **similar expected returns**, especially when total reward (welfare) is held constant.

6.11.3 Example: Battle of the Sexes

	A	B
A	(10, 7)	(2, 2)
B	(0, 0)	(7, 10)

This game represents **coordination with asymmetric preferences**: both agents want to meet, but have different favorite locations.

✓ 6.11.4 Visualization: Welfare and Fairness

The following visualization shows:

- **Feasible joint rewards** as dots.
- **Pareto-optimal** policies as red squares.
- **Fairness-optimal** policies as purple diamonds.
- **Nash equilibria** as blue triangles.

```

payoffs = {
    ("A", "A"): (10, 7),
    ("A", "B"): (2, 2),
    ("B", "A"): (0, 0),
    ("B", "B"): (7, 10),
}
actions = ["A", "B"]

steps = 50
policy_range = np.linspace(0, 1, steps)
joint_rewards = []

for p1 in policy_range:
    for p2 in policy_range:
        pi1 = {"A": p1, "B": 1 - p1}
        pi2 = {"A": p2, "B": 1 - p2}
        r1, r2 = 0, 0
        for a1 in actions:
            for a2 in actions:
                prob = pi1[a1] * pi2[a2]
                u1, u2 = payoffs[(a1, a2)]
                r1 += prob * u1
                r2 += prob * u2
        welfare = r1 + r2
        fairness = r1 * r2
        joint_rewards.append((r1, r2, welfare, fairness))

joint_rewards = np.array(joint_rewards)
welfare_vals = joint_rewards[:, 2]
fairness_vals = joint_rewards[:, 3]

def is_pareto(points):
    points = np.array(points)
    is_efficient = np.ones(points.shape[0], dtype=bool)
    for i, c in enumerate(points):
        if is_efficient[i]:
            is_efficient[is_efficient] = np.any(points[is_efficient] > c, axis=1)
            is_efficient[i] = True

```

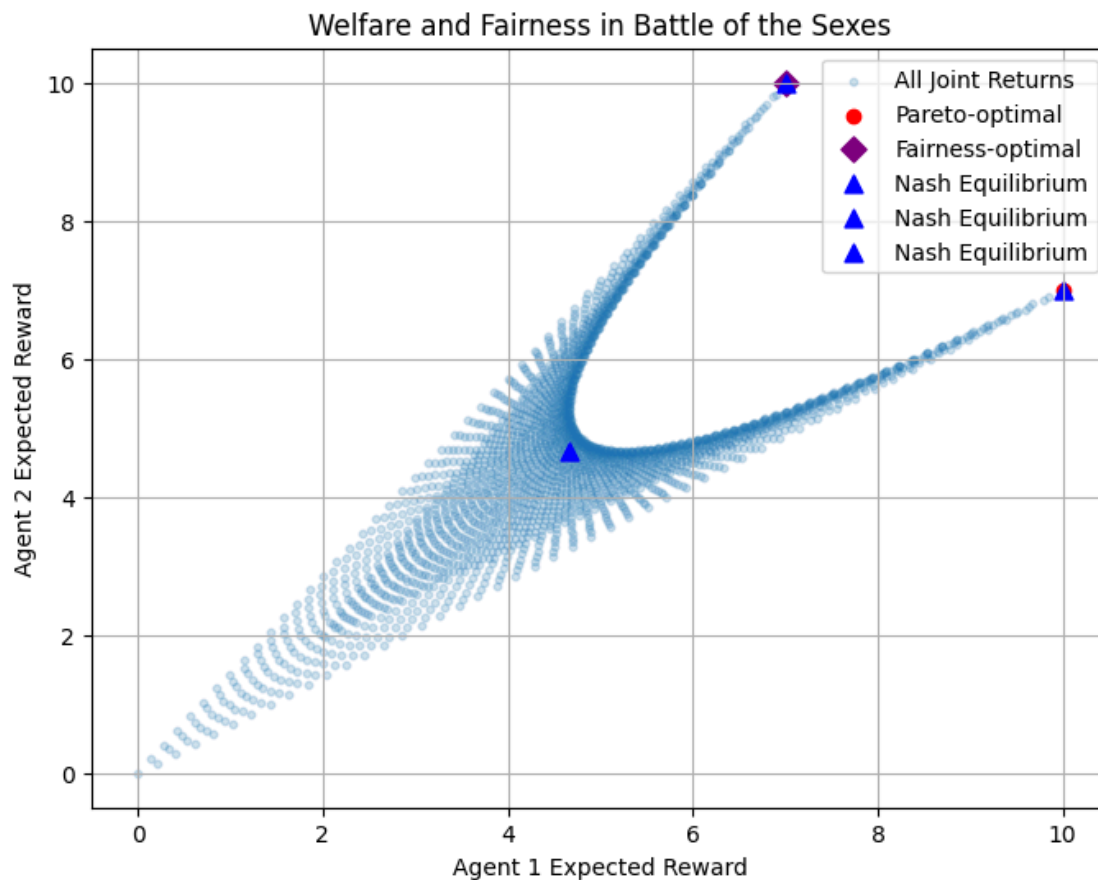
```

return is_efficient

pareto_mask = is_pareto(joint_rewards[:, :2])
fairness_best = np.argmax(fairness_vals)
nash_points = [(10, 7), (7, 10), (4.66, 4.66)] # Theoretical Nash outcomes

plt.figure(figsize=(8, 6))
plt.scatter(joint_rewards[:, 0], joint_rewards[:, 1], alpha=0.2, s=10, label="All Joint Returns")
plt.scatter(joint_rewards[pareto_mask, 0], joint_rewards[pareto_mask, 1], color="red", label="Pareto")
plt.scatter(joint_rewards[fairness_best, 0], joint_rewards[fairness_best, 1], color="purple", marker
for x, y in nash_points:
    plt.scatter(x, y, color="blue", marker="^", s=60, label="Nash Equilibrium")
plt.xlabel("Agent 1 Expected Reward")
plt.ylabel("Agent 2 Expected Reward")
plt.title("Welfare and Fairness in Battle of the Sexes")
plt.legend()
plt.grid(True)
plt.show()

```



6.12 No-Regret

All previous solution concepts in this chapter — **Nash**, **minimax**, **correlated equilibrium**, etc. — are **static**, in the sense that they characterize a fixed joint policy. In contrast, the **no-regret** solution concept is **dynamic**: it evaluates how well agents perform over time during **learning**.

6.12.1 Definition: Regret

Let a^e be the joint action in episode e for $e = 1, \dots, z$. The **regret** of agent i over z episodes is:

$$\text{Regret}_i^z = \max_{a_i \in A_i} \sum_{e=1}^z [R_i(\langle a_i, a_{-i}^e \rangle) - R_i(a^e)]$$

Agent i compares its **actual rewards** to the rewards it **could have gotten** by consistently playing the best fixed action in hindsight.

6.12.2 No-Regret Condition

Agent i has **no regret** if the **average regret** tends to zero:

$$\lim_{Z \rightarrow \infty} \frac{1}{Z} \text{Regret}_i^Z \leq 0$$

This definition can be relaxed to ϵ -**no-regret** by replacing 0 with a small $\epsilon > 0$.

6.12.3 Example: Prisoner's Dilemma (Empirical Regret)

Episode e	a_1^e a_2^e		R_1 (a_1^e , a_2^e)	R_1 (C, D)	R_1 (D, D)
	a_1^e	a_2^e			
1	C	C	-1	-1	0
2	C	D	-5	-5	-3
3	D	C	0	-1	0
4	C	D	-5	-5	-3
5	D	D	-3	-5	-3
6	D	D	-3	-5	-3
7	C	C	-1	-1	0
8	D	C	0	-1	0
9	D	D	-3	-5	-3
10	D	C	0	-1	0

- Agent 1's actual reward sum: **-21**
- Best fixed action in hindsight: **Always D**, which yields: **-15**
- So, $\text{Regret}_1^{10} = -15 - (-21) = 6 \Rightarrow \text{average regret} = \mathbf{0.6}$

6.12.4 No-Regret in General Game Models

For **sequential games**, the regret is computed **over policies** rather than actions:

Let Π_i be the set of policies for agent i , and let π^e be the joint policy in episode e . Then,

$$\text{Regret}_i^Z = \max_{\pi_i \in \Pi_i} \sum_{e=1}^Z [U_i(\langle \pi_i, \pi_{-i}^e \rangle) - U_i(\pi^e)]$$

6.12.5 Limitations

- Assumes **fixed opponents**: Regret compares against counterfactuals assuming others do not change their behavior — which may not be valid in practice.
- **Minimizing regret is not equivalent to maximizing returns.**
 - In Prisoner's Dilemma, always defecting (D) has no regret — but both players end up with low returns.
- Different **types of regret** exist:
 - **External Regret**: Replace all past actions with one best alternative.
 - **Internal (Conditional) Regret**: Swap individual action occurrences (more fine-grained).

6.12.6 Relation to Equilibria

- **No external regret** \Rightarrow joint action distributions converge to **coarse correlated equilibria**.
- **No internal regret** \Rightarrow convergence to **correlated equilibria** (Hart & Mas-Colell, 2000).

✓ 6.12.7 Summary Table

Concept	Definition	Target Behavior
External Regret	Best constant action in hindsight	Coarse Correlated Equilibrium
Internal Regret	Conditional swap of individual actions	Correlated Equilibrium
No-Regret	$\text{Regret} \rightarrow 0$ over time	Decentralized Learning Stability
ϵ -No-Regret	$\text{Regret} \leq \epsilon$	Approximate Learning Convergence

No-regret learning captures the **long-term rationality** of agents across repeated interactions - a cornerstone of **adaptive MARL algorithms**.

```
payoff_matrix = np.array([
    [-1, -5], # Agent 1 plays C
    [0, -3]  # Agent 1 plays D
])

agent1_actions = [0, 0, 1, 0, 1, 1, 0, 1, 1, 1] # C=0, D=1
agent2_actions = [0, 1, 0, 1, 1, 1, 0, 0, 1, 0]

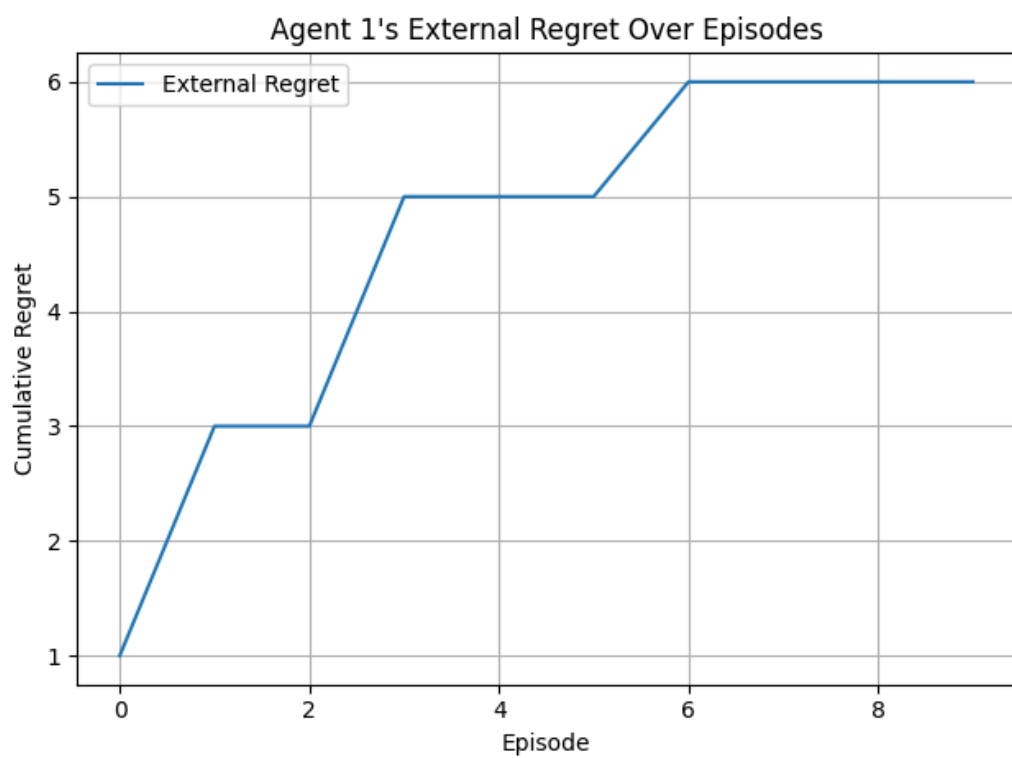
actual_rewards = [payoff_matrix[a1, a2] for a1, a2 in zip(agent1_actions, agent2_actions)]
cumulative_actual_reward = np.cumsum(actual_rewards)

counterfactual_C = [payoff_matrix[0, a2] for a2 in agent2_actions]
counterfactual_D = [payoff_matrix[1, a2] for a2 in agent2_actions]

regret_C = np.cumsum(counterfactual_C) - cumulative_actual_reward
regret_D = np.cumsum(counterfactual_D) - cumulative_actual_reward

external_regret = np.maximum(regret_C, regret_D)

# Plotting
plt.plot(external_regret, label="External Regret")
plt.xlabel("Episode")
plt.ylabel("Cumulative Regret")
plt.title("Agent 1's External Regret Over Episodes")
plt.grid(True)
plt.legend()
plt.tight_layout()
plt.show()
```



✓ 7. Practice - Introduction to Single-Agent Reinforcement Learning

 Zoltán Barta, PhD student, Department of Artificial Intelligence

 90 min read

 January 22, 2025

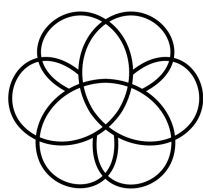
 Collective Intelligence



ELTE | IK
INFORMATIKAI KAR



DEPARTMENT
OF ARTIFICIAL
INTELLIGENCE



Gymnasium

This practice notebook provides a comprehensive introduction to single-agent reinforcement learning (RL), focusing on the fundamental algorithms and principles that underpin modern RL research and applications. Students will begin by exploring the basic components of RL, including agents, environments, and the mathematical framework of Markov Decision Processes. Through hands-on implementation, the notebook guides students from classic Q-learning to advanced deep RL techniques, such as Deep Q-Networks (DQN) and Proximal Policy Optimization (PPO). Practical coding exercises and simulation environments reinforce theoretical concepts and demonstrate how RL agents learn to solve sequential decision problems autonomously. By the end of this practice, students will have acquired both the intuition and technical skills necessary to design, implement, and evaluate single-agent reinforcement learning solutions.

Table of Contents

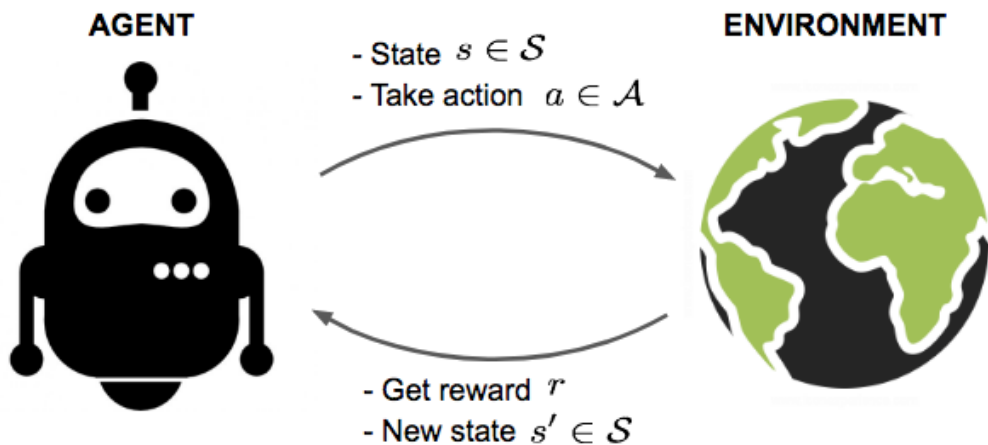
- **7.1 Overview of Reinforcement Learning**
 - 7.1.1 Exploration vs. Exploitation in Reinforcement Learning
- **7.2 Q-Learning**
 - 7.2.1 How It Works
 - 7.2.2 Algorithm Steps
- **7.3 Deep Q-Learning**
 - 7.3.1 How It Works
 - 7.3.2 Algorithm Steps
 - 7.3.3 Implementation Tips
- **7.4 Proximal Policy Optimization (PPO)**
 - 7.4.1 How It Works
 - 7.4.2 Algorithm Steps

✓ 7.1 Overview of Reinforcement Learning

A Markov Decision Process (MDP) is a mathematical framework used to model decision-making in situations where outcomes are partly random and partly under the control of an agent. An MDP is defined by the tuple $((S, A, P, R, \gamma))$:

- S : A set of states representing all possible configurations of the environment.
- A : A set of actions available to the agent.
- $P(s' | s, a)$: The state transition probability, defining the probability of moving to state s' after taking action a in state s .
- $R(s, a)$: A reward function, which assigns a scalar value for taking action a in state s .
- γ : A discount factor ($0 \leq \gamma \leq 1$) that determines the importance of future rewards.

In an MDP, the Markov property assumes that the next state depends only on the current state and action, not on the sequence of past states. MDPs are widely used in reinforcement learning to model environments and solve sequential decision-making problems.



7.1.1 Exploration vs. Exploitation in Reinforcement Learning

Exploration and exploitation are two fundamental concepts in reinforcement learning that define how an agent interacts with its environment to maximize rewards.

- **Exploration:** Refers to the process of trying out new actions to gather more information about the environment. This helps the agent discover better strategies and avoid local optima. For example, taking a random action can help the agent learn about areas of the state space it hasn't visited yet.
- **Exploitation:** Involves selecting the best-known action based on the agent's current knowledge to maximize immediate rewards. This focuses on leveraging what the agent has already learned to achieve higher performance.

A balance between exploration and exploitation is crucial for effective learning. Too much exploration can lead to inefficiency, as the agent spends excessive time on suboptimal actions. On the other hand, over-exploitation might cause the agent to miss out on discovering better strategies. Techniques like ϵ -greedy policies, upper confidence bound (UCB), and Thompson sampling are commonly used to manage the exploration-exploitation tradeoff.

✓ 7.2 Q-Learning

Q-Learning is a model-free reinforcement learning algorithm that learns the optimal action-value function $Q(s, a)$ by iteratively updating estimates based on observed rewards and transitions. Unlike policy-based methods, Q-learning focuses on learning the value of actions and deriving a policy indirectly through action selection.

7.2.1 How It Works

1. Q-Table Initialization:

- The agent maintains a table $Q(s, a)$ that stores the estimated value of taking action a in state s .
- Initially, all Q-values are set arbitrarily (e.g., to 0).

2. Action Selection (ϵ -greedy):

- At each time step, the agent selects an action using an ϵ -greedy strategy:
 - With probability ϵ , a random action is chosen (exploration).
 - Otherwise, it selects the action with the highest Q-value in the current state:
 $\arg \max_a Q(s, a)$ (exploitation).

3. Transition and Update:

- After taking action a_t in state s_t , the agent observes the reward r_t and the next state s_{t+1} .
- The Q-value is updated using the Bellman equation:

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha \left[r_t + \gamma \max_{a'} Q(s_{t+1}, a') - Q(s_t, a_t) \right]$$

4. Policy Derivation:

- Once the Q-table has been sufficiently updated, the optimal policy is derived by selecting the action with the highest Q-value in each state.

✓ 7.2.2 Algorithm Steps

1. Initialize:

- Q-table $Q(s, a)$ with arbitrary values.
- Set hyperparameters: learning rate α , discount factor γ , and exploration rate ϵ .

2. For each episode:

- Initialize the starting state s_0 .

3. For each step in the episode:

- Choose action a_t using ϵ -greedy strategy.
- Execute action, observe reward r_t and next state s_{t+1} .
- Update Q-value

4. Repeat:

- Continue until the episode ends or convergence is reached.

```
import gymnasium
import numpy as np
env = gymnasium.make("FrozenLake-v1", is_slippery=False)
```

```

# Create environment

# Initialize Q-table
q_table = np.zeros((env.observation_space.n, env.action_space.n))

# Hyperparameters
learning_rate = 0.01
discount_factor = 0.99
epsilon = 1.0

epsilon_decay = 0.99
num_episodes = 100000

# Training loop
for episode in range(num_episodes):
    state, info = env.reset()
    done = False
    while not done:
        # Epsilon-greedy action selection
        if np.random.random() < epsilon:
            action = env.action_space.sample()
        else:
            action = np.argmax(q_table[state])
        # Step through the environment
        next_state, reward, terminated, truncated, info = env.step(action)

        # Check if the episode is done
        done = terminated or truncated

        # Q-learning update
        q_table[state, action] = q_table[state, action] + learning_rate * (
            reward + discount_factor * np.max(q_table[next_state]) - q_table[state, action]
        )

        state = next_state

    epsilon *= epsilon_decay

print("Training completed.")
print("Final Q-Table:")
print(q_table)

```



Training completed.

Final Q-Table:

```

[[5.16911799e-04 9.50990050e-01 7.21691201e-08 4.83236228e-03]
 [1.45125457e-03 0.00000000e+00 0.00000000e+00 0.00000000e+00]
 [0.00000000e+00 0.00000000e+00 0.00000000e+00 0.00000000e+00]
 [0.00000000e+00 0.00000000e+00 0.00000000e+00 0.00000000e+00]
 [2.59405358e-06 9.60596010e-01 0.00000000e+00 1.47077574e-05]
 [0.00000000e+00 0.00000000e+00 0.00000000e+00 0.00000000e+00]
 [0.00000000e+00 0.00000000e+00 0.00000000e+00 0.00000000e+00]
 [0.00000000e+00 0.00000000e+00 0.00000000e+00 0.00000000e+00]
 [1.02579243e-05 0.00000000e+00 9.70299000e-01 9.29295941e-06]
 [9.59501299e-03 0.00000000e+00 9.80100000e-01 0.00000000e+00]
 [4.02572628e-03 9.90000000e-01 0.00000000e+00 0.00000000e+00]
 [0.00000000e+00 0.00000000e+00 0.00000000e+00 0.00000000e+00]
 [0.00000000e+00 0.00000000e+00 0.00000000e+00 0.00000000e+00]
 [0.00000000e+00 0.00000000e+00 0.00000000e+00 0.00000000e+00]
 [0.00000000e+00 1.91958854e-02 1.00000000e+00 2.49938696e-03]
 [0.00000000e+00 0.00000000e+00 0.00000000e+00 0.00000000e+00]]

```

```

# Evaluation loop
import time
env = gymnasium.make("FrozenLake-v1", is_slippery=False, render_mode="human")

```

```

state, info = env.reset()
done = False
total_reward = 0

print("Starting evaluation episode...")

while not done:
    # Choose the best action from Q-table
    action = np.argmax(q_table[state])

    # Display the action and the state
    print(f"State: {state}, Action: {action}")

    # Step through the environment
    next_state, reward, terminated, truncated, info = env.step(action)

    # Accumulate reward
    total_reward += reward

    # Render the environment for visualization (optional)
    env.render()
    time.sleep(0.1)
    # Check if the episode is done
    done = terminated or truncated
    state = next_state

print("Evaluation episode completed.")
print(f"Total Reward: {total_reward}")

```

```

➡ Starting evaluation episode...
State: 0, Action: 1
State: 4, Action: 1
State: 8, Action: 2
State: 9, Action: 2
State: 10, Action: 1
State: 14, Action: 2
Evaluation episode completed.
Total Reward: 1.0

```

✓ 7.3 Deep Q-Learning

Deep Q-Learning (DQN) is a powerful reinforcement learning algorithm that extends **Q-learning** by using deep neural networks to approximate the action-value function $Q(s, a)$. It enables agents to learn effective policies in high-dimensional or continuous state spaces, such as images or sensor inputs.

7.3.1 How It Works

1. Q-Function Approximation:

- A neural network $Q_{\theta}(s, a)$ approximates the Q-function, which estimates the expected return of taking action a in state s and following the current policy thereafter.

2. Experience Replay:

- The agent stores transitions (s, a, r, s') in a replay buffer.
- During training, it samples random minibatches to break correlation between sequential data and stabilize learning.

3. Target Network:

- A separate **target network** Q_{θ^-} is used to generate stable targets for training.

- It is periodically updated to match the current Q-network:

$$\theta^- \leftarrow \theta$$

4. Bellman Update:

- The target for each transition is computed using the Bellman equation:

$$y = r + \gamma \max_{a'} Q_{\theta^-}(s', a')$$

- The network minimizes the loss:

$$L(\theta) = (y - Q_{\theta}(s, a))^2$$

5. Action Selection (ϵ -greedy):

- The agent selects actions using an ϵ -greedy strategy:
 - With probability ϵ , choose a random action (exploration).
 - Otherwise, choose $\arg \max_a Q_{\theta}(s, a)$ (exploitation).

7.3.2 Algorithm Steps

1. Initialize:

- Q-network Q_{θ} , target network Q_{θ^-}
- Replay buffer, ϵ -greedy parameters, learning rate, discount factor γ

2. For each episode:

- Initialize environment and get initial state s_0

3. For each step in the episode:

- Select action a_t using ϵ -greedy strategy.
- Execute action, observe reward r_t and next state s_{t+1}
- Store transition (s_t, a_t, r_t, s_{t+1}) in replay buffer.
- Sample minibatch of transitions from the buffer.
- Compute targets
- Update Q-network by minimizing the loss

4. Update Target Network:

- Periodically set $\theta^- \leftarrow \theta$

5. Repeat:

- Continue until convergence or maximum number of episodes is reached.

✓ 7.3.3 Implementation Tips

• Replay Buffer Size:

- Use a large buffer to capture diverse experiences.

• Target Network Update Frequency:

- Update less frequently (e.g., every few hundred steps) to improve stability.

• Reward Clipping:

- Clip rewards (e.g., to $[-1, 1]$) to stabilize training in environments with large reward magnitudes.

• Frame Stacking:

- For visual input (like in Atari), stack several recent frames as input to capture motion.

```

import gymnasium as gym
import math
import random
import matplotlib
import matplotlib.pyplot as plt
from collections import namedtuple, deque
from itertools import count

import torch
import torch.nn as nn
import torch.optim as optim
import torch.nn.functional as F

env = gym.make("CartPole-v1")

# set up matplotlib
is_ipython = 'inline' in matplotlib.get_backend()
if is_ipython:
    from IPython import display

plt.ion()

# if GPU is to be used
device = torch.device(
    "cuda" if torch.cuda.is_available() else
    "mps" if torch.backends.mps.is_available() else
    "cpu"
)
print(device)

🔄 mps

Transition = namedtuple('Transition',
                        ('state', 'action', 'next_state', 'reward'))

class ReplayMemory(object):
    def __init__(self, capacity):
        self.memory = deque([], maxlen=capacity)

    def push(self, *args):
        """Save a transition"""
        self.memory.append(Transition(*args))

    def sample(self, batch_size):
        return random.sample(self.memory, batch_size)

    def __len__(self):
        return len(self.memory)

class DQN(nn.Module):
    def __init__(self, n_observations, n_actions):
        super(DQN, self).__init__()
        self.layer1 = nn.Linear(n_observations, 128)
        self.layer2 = nn.Linear(128, 128)
        self.layer3 = nn.Linear(128, n_actions)

    def forward(self, x):
        x = F.relu(self.layer1(x))
        x = F.relu(self.layer2(x))
        return self.layer3(x)

```

- **BATCH_SIZE** : The number of transitions sampled from the replay buffer at each learning step, determining how many experiences are used to compute the loss.
- **GAMMA** : The discount factor that weighs future rewards; a value closer to 1 emphasizes long-term rewards.
- **EPS_START** : The initial value of epsilon in the ϵ -greedy policy, representing the probability of choosing a random action at the beginning of training.
- **EPS_END** : The minimum value epsilon can decay to, ensuring continued (but limited) exploration throughout training.
- **EPS_DECAY** : The rate at which epsilon decays exponentially; higher values result in slower decay, preserving exploration for a longer time.
- **TAU** : The soft update factor used to slowly blend parameters from the main network into the target network, ensuring stable learning.
- **LR** : The learning rate used by the AdamW optimizer to update the network parameters during training.

```
BATCH_SIZE = 128
GAMMA = 0.99
EPS_START = 0.9
EPS_END = 0.05
EPS_DECAY = 1000
TAU = 0.005
LR = 1e-4
```

```
# Get number of actions from gym action space
n_actions = env.action_space.n
# Get the number of state observations
state, info = env.reset()
n_observations = len(state)
```

```
policy_net = DQN(n_observations, n_actions).to(device)
target_net = DQN(n_observations, n_actions).to(device)
target_net.load_state_dict(policy_net.state_dict())
```

```
optimizer = optim.AdamW(policy_net.parameters(), lr=LR, amsgrad=True)
memory = ReplayMemory(10000)
```

```
steps_done = 0
```

```
def select_action(state):
    global steps_done
    sample = random.random()
    eps_threshold = EPS_END + (EPS_START - EPS_END) * \
        math.exp(-1. * steps_done / EPS_DECAY)
    steps_done += 1
    if sample > eps_threshold:
        with torch.no_grad():
            # t.max(1) will return the largest column value of each row.
            # second column on max result is index of where max element was
            # found, so we pick action with the larger expected reward.
            return policy_net(state).max(1).indices.view(1, 1)
    else:
        return torch.tensor([env.action_space.sample()], device=device, dtype=torch.long)
```

```
episode_durations = []
```

```
def plot_durations(show_result=False):
    plt.figure(1)
    durations_t = torch.tensor(episode_durations, dtype=torch.float)
    if show_result:
        plt.title('Result')
    else:
```

```

        plt.clf()
        plt.title('Training...')
    plt.xlabel('Episode')
    plt.ylabel('Duration')
    plt.plot(durations_t.numpy())
    # Take 100 episode averages and plot them too
    if len(durations_t) >= 100:
        means = durations_t.unfold(0, 100, 1).mean(1).view(-1)
        means = torch.cat((torch.zeros(99), means))
        plt.plot(means.numpy())

plt.pause(0.001) # pause a bit so that plots are updated
if is_ipython:
    if not show_result:
        display.display(plt.gcf())
        display.clear_output(wait=True)
    else:
        display.display(plt.gcf())

def optimize_model():
    if len(memory) < BATCH_SIZE:
        return
    transitions = memory.sample(BATCH_SIZE)
    batch = Transition(*zip(*transitions))

    non_final_mask = torch.tensor(tuple(map(lambda s: s is not None,
                                             batch.next_state)), device=device, dtype=torch.bool)
    non_final_next_states = torch.cat([s for s in batch.next_state
                                       if s is not None])
    state_batch = torch.cat(batch.state)
    action_batch = torch.cat(batch.action)
    reward_batch = torch.cat(batch.reward)

    state_action_values = policy_net(state_batch).gather(1, action_batch)

    next_state_values = torch.zeros(BATCH_SIZE, device=device)
    with torch.no_grad():
        next_state_values[non_final_mask] = target_net(non_final_next_states).max(1).values
    expected_state_action_values = (next_state_values * GAMMA) + reward_batch

    criterion = nn.SmoothL1Loss()
    loss = criterion(state_action_values, expected_state_action_values.unsqueeze(1))

    optimizer.zero_grad()
    loss.backward()

    torch.nn.utils.clip_grad_value_(policy_net.parameters(), 100)
    optimizer.step()

if torch.cuda.is_available() or torch.backends.mps.is_available():
    num_episodes = 600
else:
    num_episodes = 50

for i_episode in range(num_episodes):
    # Initialize the environment and get its state
    state, info = env.reset()
    state = torch.tensor(state, dtype=torch.float32, device=device).unsqueeze(0)
    for t in count():
        action = select_action(state)
        observation, reward, terminated, truncated, _ = env.step(action.item())

```

```

reward = torch.tensor([reward], device=device)
done = terminated or truncated

if terminated:
    next_state = None
else:
    next_state = torch.tensor(observation, dtype=torch.float32, device=device).unsqueeze(1)

# Store the transition in memory
memory.push(state, action, next_state, reward)

# Move to the next state
state = next_state

# Perform one step of the optimization (on the policy network)
optimize_model()

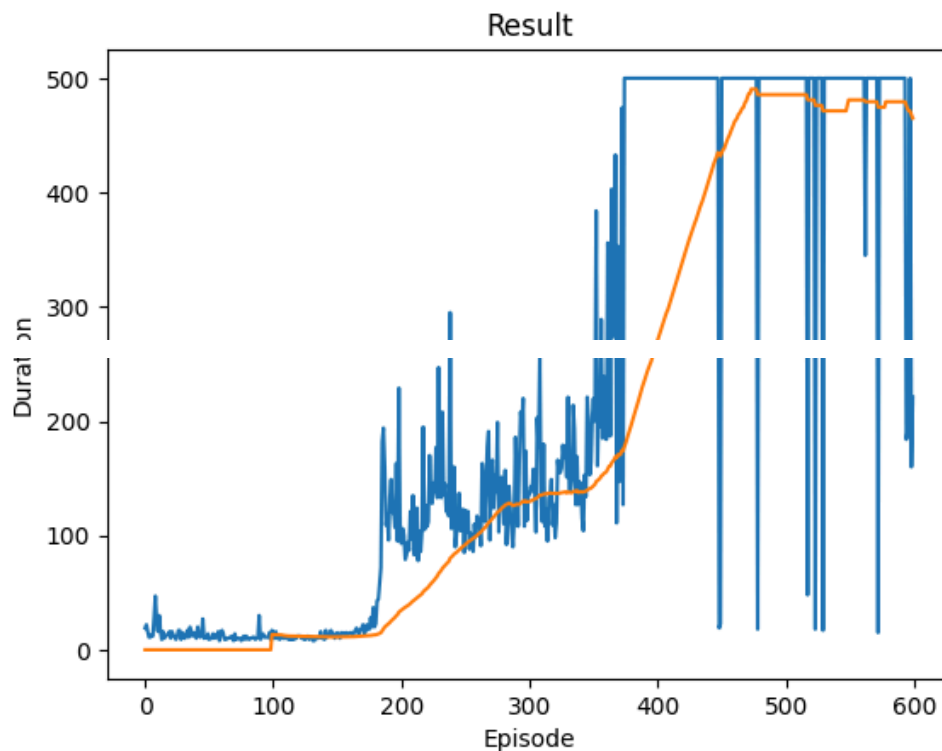
# Soft update of the target network's weights
#  $\theta' \leftarrow \tau \theta + (1 - \tau) \theta'$ 
target_net_state_dict = target_net.state_dict()
policy_net_state_dict = policy_net.state_dict()
for key in policy_net_state_dict:
    target_net_state_dict[key] = policy_net_state_dict[key]*TAU + target_net_state_dict[key]*(1-TAU)
target_net.load_state_dict(target_net_state_dict)

if done:
    episode_durations.append(t + 1)
    plot_durations()
    break

print('Complete')
plot_durations(show_result=True)
plt.ioff()
plt.show()

```

➡ Complete



<Figure size 640x480 with 0 Axes>
 <Figure size 640x480 with 0 Axes>

✓ 7.4 Proximal Policy Optimization (PPO)

Proximal Policy Optimization (PPO) is a state-of-the-art policy gradient method in reinforcement learning that strikes a balance between performance and implementation simplicity. It improves stability during training by constraining policy updates, making it more robust than standard policy gradient or REINFORCE methods.

7.4.1 How It Works

1. Policy Network:

- The agent uses a neural network $\pi_{\theta}(a|s)$ to parameterize the policy, which outputs a probability distribution over actions given the current state.

2. Trajectory Collection:

- The agent interacts with the environment using the current policy to collect trajectories consisting of states, actions, rewards, and log probabilities of actions.

3. Advantage Estimation:

- The agent estimates the **advantage** A_t of taking an action using methods like Generalized Advantage Estimation (GAE) to reduce variance and improve learning efficiency.

4. Clipped Objective:

- PPO introduces a **clipped surrogate objective** to limit the size of policy updates and avoid large destructive changes: $L^{CLIP}(\theta) = \mathbb{E}_t \left[\min(r_t(\theta)A_t, \text{clip}(r_t(\theta), 1 - \epsilon, 1 + \epsilon)A_t) \right]$ where $r_t(\theta) = \frac{\pi_{\theta}(a_t|s_t)}{\pi_{\theta_{\text{old}}}(a_t|s_t)}$ is the probability ratio.

5. Optimization:

- The policy parameters are updated by maximizing the clipped objective, typically using stochastic gradient ascent.
- The value function is trained alongside the policy using mean squared error loss.

7.4.2 Algorithm Steps

1. Initialize:

- Policy network π_{θ} and value network V_{ϕ}
- Set hyperparameters: learning rate, discount factor γ , clipping parameter ϵ , number of epochs, batch size

2. For each iteration:

- Collect trajectories by running the current policy in the environment
- Compute rewards-to-go and advantage estimates A_t
- Compute the ratio $r_t(\theta)$ between new and old policies
- Optimize the clipped objective over multiple epochs

3. Update Value Function:

- Minimize the value loss

4. Repeat:

- Continue the process for a predefined number of iterations or until performance converges.

✓ 7.4.3 Implementation Tips

- **Normalize Advantages:**

- Normalize A_t to have mean 0 and standard deviation 1 to stabilize updates.

- **Entropy Bonus:**

- Add an entropy term to the loss to encourage exploration:

$$L^{ENTROPY} = \beta \cdot \mathbb{E}_t [\mathbb{E}[\pi_\theta(\cdot | s_t)]]$$

- **Mini-batch Updates:**

- Perform multiple epochs of mini-batch updates over the collected data to improve sample efficiency.

- **Value Function Clipping:**

- Optionally clip value estimates similarly to policy updates to further stabilize learning.

```
from collections import defaultdict

import matplotlib.pyplot as plt
import torch
from tensordict.nn import TensorDictModule
from tensordict.nn.distributions import NormalParamExtractor
from torch import nn

from torchrl.collectors import SyncDataCollector
from torchrl.data.replay_buffers import ReplayBuffer
from torchrl.data.replay_buffers.samplers import SamplerWithoutReplacement
from torchrl.data.replay_buffers.storages import LazyTensorStorage
from torchrl.envs import (
    Compose,
    DoubleToFloat,
    ObservationNorm,
    StepCounter,
    TransformedEnv,
)
from torchrl.envs.libs.gym import GymEnv, GymLikeEnv
from torchrl.envs import PendulumEnv
from torchrl.envs.utils import check_env_specs, ExplorationType, set_exploration_type
from torchrl.modules import ProbabilisticActor, TanhNormal, ValueOperator
from torchrl.objectives import ClipPPOLoss
from torchrl.objectives.value import GAE
from tqdm import tqdm
device = torch.device(
    "cuda" if torch.cuda.is_available() else
    #"mps" if torch.backends.mps.is_available() else
    "cpu"
)
print(device)

➡ cpu

num_cells = 256 # number of cells in each layer i.e. output dim.
lr = 3e-4
max_grad_norm = 1.0

frames_per_batch = 1000
total_frames = 100_000
```

```

sub_batch_size = 64
num_epochs = 10
clip_epsilon = (
    0.2
)
gamma = 0.99
lmbda = 0.95
entropy_eps = 1e-4

from torchrl.envs import GymEnv
from torchrl.envs import set_gym_backend
import torch
import torch
import torch
torch.manual_seed(0)
class AutoResettingGymEnv(GymEnv):
    def _step(self, tensordict):
        tensordict = super()._step(tensordict)
        if tensordict["done"].any():
            td_reset = super().reset()
            tensordict.update(td_reset.exclude(*self.done_keys))
        return tensordict
    def _reset(self, tensordict=None):
        if tensordict is not None and "_reset" in tensordict:
            return tensordict.copy()
        return super()._reset(tensordict)
with set_gym_backend("gym"):
    env = AutoResettingGymEnv("Pendulum-v1", auto_reset=True, auto_reset_replace=True, device="mp

base_env = GymEnv("MountainCarContinuous-v0", device=device)

env = TransformedEnv(
    base_env,
    Compose(
        # normalize observations
        ObservationNorm(in_keys=["observation"]),
        DoubleToFloat(),
        StepCounter(),
    ),
)

```

Implementation Note: We usually want normalized values for observations, but we have no prior info about the distribution of the state space, that's why we run the env with a num of random steps, and create a gaussian distro and compute the summary of statistics on these observations.

```
env.transform[0].init_stats(num_iter=1000, reduce_dim=0, cat_dim=0)
```

```
print("normalization constant shape:", env.transform[0].loc.shape)
```

```
⇒ normalization constant shape: torch.Size([2])
```

```

print("observation_spec:", env.observation_spec)
print("reward_spec:", env.reward_spec)
print("input_spec:", env.input_spec)
print("action_spec (as defined by input_spec):", env.action_spec)

```

```

⇒ observation_spec: Composite(
  observation: BoundedContinuous(
    shape=torch.Size([2]),

```



```

        space=ContinuousBox(
            low=Tensor(shape=torch.Size([2]), device=cpu, dtype=torch.float32, contiguous=True),
            high=Tensor(shape=torch.Size([2]), device=cpu, dtype=torch.float32, contiguous=True),
            device=cpu,
            dtype=torch.float32,
            domain=continuous),
    step_count: BoundedDiscrete(
        shape=torch.Size([1]),
        space=ContinuousBox(
            low=Tensor(shape=torch.Size([1]), device=cpu, dtype=torch.int64, contiguous=True),
            high=Tensor(shape=torch.Size([1]), device=cpu, dtype=torch.int64, contiguous=True),
            device=cpu,
            dtype=torch.int64,
            domain=discrete),
        device=cpu,
        shape=torch.Size([1]))
    reward_spec: UnboundedContinuous(
        shape=torch.Size([1]),
        space=ContinuousBox(
            low=Tensor(shape=torch.Size([1]), device=cpu, dtype=torch.float32, contiguous=True),
            high=Tensor(shape=torch.Size([1]), device=cpu, dtype=torch.float32, contiguous=True),
            device=cpu,
            dtype=torch.float32,
            domain=continuous)
    input_spec: Composite(
        full_state_spec: Composite(
            step_count: BoundedDiscrete(
                shape=torch.Size([1]),
                space=ContinuousBox(
                    low=Tensor(shape=torch.Size([1]), device=cpu, dtype=torch.int64, contiguous=True),
                    high=Tensor(shape=torch.Size([1]), device=cpu, dtype=torch.int64, contiguous=True),
                    device=cpu,
                    dtype=torch.int64,
                    domain=discrete),
                device=cpu,
                shape=torch.Size([1])),
            full_action_spec: Composite(
                action: BoundedContinuous(
                    shape=torch.Size([1]),
                    space=ContinuousBox(
                        low=Tensor(shape=torch.Size([1]), device=cpu, dtype=torch.float32, contiguous=True),
                        high=Tensor(shape=torch.Size([1]), device=cpu, dtype=torch.float32, contiguous=True),
                        device=cpu,
                        dtype=torch.float32,
                        domain=continuous),
                    device=cpu,
                    shape=torch.Size([1])),
                device=cpu,
                shape=torch.Size([1]))
    action_spec (as defined by input_spec): BoundedContinuous(
        shape=torch.Size([1]),
        space=ContinuousBox(
            low=Tensor(shape=torch.Size([1]), device=cpu, dtype=torch.float32, contiguous=True),
            high=Tensor(shape=torch.Size([1]), device=cpu, dtype=torch.float32, contiguous=True)

```

```
check_env_specs(env)
```

```
➡ 2025-05-12 11:01:21,468 [torchrl][INFO] check_env_specs succeeded!
```

```

rollout = env.rollout(3)
print("rollout of three steps:", rollout)
print("Shape of the rollout TensorDict:", rollout.batch_size)

```

```

➡ rollout of three steps: TensorDict(
  fields={
    action: Tensor(shape=torch.Size([3, 1]), device=cpu, dtype=torch.float32, is_shared=False),
    done: Tensor(shape=torch.Size([3, 1]), device=cpu, dtype=torch.bool, is_shared=False),
    next: TensorDict(

```

```

        fields={
            done: Tensor(shape=torch.Size([3, 1]), device=cpu, dtype=torch.bool, is_shared=False),
            observation: Tensor(shape=torch.Size([3, 2]), device=cpu, dtype=torch.float32, is_shared=False),
            reward: Tensor(shape=torch.Size([3, 1]), device=cpu, dtype=torch.float32, is_shared=False),
            step_count: Tensor(shape=torch.Size([3, 1]), device=cpu, dtype=torch.int64, is_shared=False),
            terminated: Tensor(shape=torch.Size([3, 1]), device=cpu, dtype=torch.bool, is_shared=False),
            truncated: Tensor(shape=torch.Size([3, 1]), device=cpu, dtype=torch.bool, is_shared=False),
            batch_size: torch.Size([3]),
            device=cpu,
            is_shared=False),
        observation: Tensor(shape=torch.Size([3, 2]), device=cpu, dtype=torch.float32, is_shared=False),
        step_count: Tensor(shape=torch.Size([3, 1]), device=cpu, dtype=torch.int64, is_shared=False),
        terminated: Tensor(shape=torch.Size([3, 1]), device=cpu, dtype=torch.bool, is_shared=False),
        truncated: Tensor(shape=torch.Size([3, 1]), device=cpu, dtype=torch.bool, is_shared=False),
        batch_size: torch.Size([3]),
        device=cpu,
        is_shared=False)
    Shape of the rollout TensorDict: torch.Size([3])

```

```

actor_net = nn.Sequential(
    nn.LazyLinear(num_cells, device=device),
    nn.Tanh(),
    nn.LazyLinear(num_cells, device=device),
    nn.Tanh(),
    nn.LazyLinear(num_cells, device=device),
    nn.Tanh(),
    nn.LazyLinear(2 * env.action_spec.shape[-1], device=device),
    NormalParamExtractor(),
)

```

```

policy_module = TensorDictModule(
    actor_net, in_keys=["observation"], out_keys=["loc", "scale"]
)

```

```

policy_module = ProbabilisticActor(
    module=policy_module,
    spec=env.action_spec,
    in_keys=["loc", "scale"],
    distribution_class=TanhNormal,
    distribution_kwargs={
        "low": env.action_spec.space.low,
        "high": env.action_spec.space.high,
    },
    return_log_prob=True,
)

```

```

value_net = nn.Sequential(
    nn.LazyLinear(num_cells, device=device),
    nn.Tanh(),
    nn.LazyLinear(num_cells, device=device),
    nn.Tanh(),
    nn.LazyLinear(num_cells, device=device),
    nn.Tanh(),
    nn.LazyLinear(1, device=device),
)

```

```

value_module = ValueOperator(
    module=value_net,
    in_keys=["observation"],
)

```

```
print("Running policy:", policy_module(env.reset()))
print("Running value:", value_module(env.reset()))
```

```
➡ Running policy: TensorDict(
  fields={
    action: Tensor(shape=torch.Size([1]), device=cpu, dtype=torch.float32, is_shared=False),
    done: Tensor(shape=torch.Size([1]), device=cpu, dtype=torch.bool, is_shared=False),
    loc: Tensor(shape=torch.Size([1]), device=cpu, dtype=torch.float32, is_shared=False),
    observation: Tensor(shape=torch.Size([2]), device=cpu, dtype=torch.float32, is_shared=False),
    sample_log_prob: Tensor(shape=torch.Size([1]), device=cpu, dtype=torch.float32, is_shared=False),
    scale: Tensor(shape=torch.Size([1]), device=cpu, dtype=torch.float32, is_shared=False),
    step_count: Tensor(shape=torch.Size([1]), device=cpu, dtype=torch.int64, is_shared=False),
    terminated: Tensor(shape=torch.Size([1]), device=cpu, dtype=torch.bool, is_shared=False),
    truncated: Tensor(shape=torch.Size([1]), device=cpu, dtype=torch.bool, is_shared=False),
    batch_size=torch.Size([1]),
    device=cpu,
    is_shared=False)
Running value: TensorDict(
  fields={
    done: Tensor(shape=torch.Size([1]), device=cpu, dtype=torch.bool, is_shared=False),
    observation: Tensor(shape=torch.Size([2]), device=cpu, dtype=torch.float32, is_shared=False),
    state_value: Tensor(shape=torch.Size([1]), device=cpu, dtype=torch.float32, is_shared=False),
    step_count: Tensor(shape=torch.Size([1]), device=cpu, dtype=torch.int64, is_shared=False),
    terminated: Tensor(shape=torch.Size([1]), device=cpu, dtype=torch.bool, is_shared=False),
    truncated: Tensor(shape=torch.Size([1]), device=cpu, dtype=torch.bool, is_shared=False),
    batch_size=torch.Size([1]),
    device=cpu,
    is_shared=False)
```

```
collector = SyncDataCollector(
    env,
    policy_module,
    frames_per_batch=frames_per_batch,
    total_frames=total_frames,
    split_trajs=False,
    device=device,
)

replay_buffer = ReplayBuffer(
    storage=LazyTensorStorage(max_size=frames_per_batch),
    sampler=SamplerWithoutReplacement(),
)

advantage_module = GAE(
    gamma=gamma, lambda=lambda, value_network=value_module, average_gae=True
)

loss_module = ClipPPOLoss(
    actor_network=policy_module,
    critic_network=value_module,
    clip_epsilon=clip_epsilon,
    entropy_bonus=bool(entropy_eps),
    entropy_coef=entropy_eps,
    # these keys match by default but we set this for completeness
    critic_coef=1.0,
    loss_critic_type="smooth_l1",
)

optim = torch.optim.Adam(loss_module.parameters(), lr)
scheduler = torch.optim.lr_scheduler.CosineAnnealingLR(
    optim, total_frames // frames_per_batch, 0.0
)
```

```

logs = defaultdict(list)
pbar = tqdm(total=total_frames)
eval_str = ""

for i, tensordict_data in enumerate(collector):

    for _ in range(num_epochs):

        advantage_module(tensordict_data)
        data_view = tensordict_data.reshape(-1)
        replay_buffer.extend(data_view.cpu())
        for _ in range(frames_per_batch // sub_batch_size):
            subdata = replay_buffer.sample(sub_batch_size)
            loss_vals = loss_module(subdata.to(device))
            loss_value = (
                loss_vals["loss_objective"]
                + loss_vals["loss_critic"]
                + loss_vals["loss_entropy"]
            )

            loss_value.backward()

            torch.nn.utils.clip_grad_norm_(loss_module.parameters(), max_grad_norm)
            optim.step()
            optim.zero_grad()

        logs["reward"].append(tensordict_data["next", "reward"].mean().item())
        pbar.update(tensordict_data.numel())
        cum_reward_str = (
            f"average reward={logs['reward'][-1]: 4.4f} (init={logs['reward'][0]: 4.4f})"
        )
        logs["step_count"].append(tensordict_data["step_count"].max().item())
        stepcount_str = f"step count (max): {logs['step_count'][-1]}"
        logs["lr"].append(optim.param_groups[0]["lr"])
        lr_str = f"lr policy: {logs['lr'][-1]: 4.4f}"
        if i % 10 == 0:
            with set_exploration_type(ExplorationType.DETERMINISTIC), torch.no_grad():
                # execute a rollout with the trained policy
                eval_rollout = env.rollout(1000, policy_module)
                logs["eval reward"].append(eval_rollout["next", "reward"].mean().item())
                logs["eval reward (sum)"].append(
                    eval_rollout["next", "reward"].sum().item()
                )
                logs["eval step_count"].append(eval_rollout["step_count"].max().item())
                eval_str = (
                    f"eval cumulative reward: {logs['eval reward (sum)'][-1]: 4.4f} "
                    f"(init: {logs['eval reward (sum)'][0]: 4.4f}), "
                    f"eval step-count: {logs['eval step_count'][-1]}"
                )
                del eval_rollout
            pbar.set_description(", ".join([eval_str, cum_reward_str, stepcount_str, lr_str]))

        scheduler.step()

➡ eval cumulative reward: -1.9249 (init: -1.0513), eval step-count: 998, average reward=-0.0016

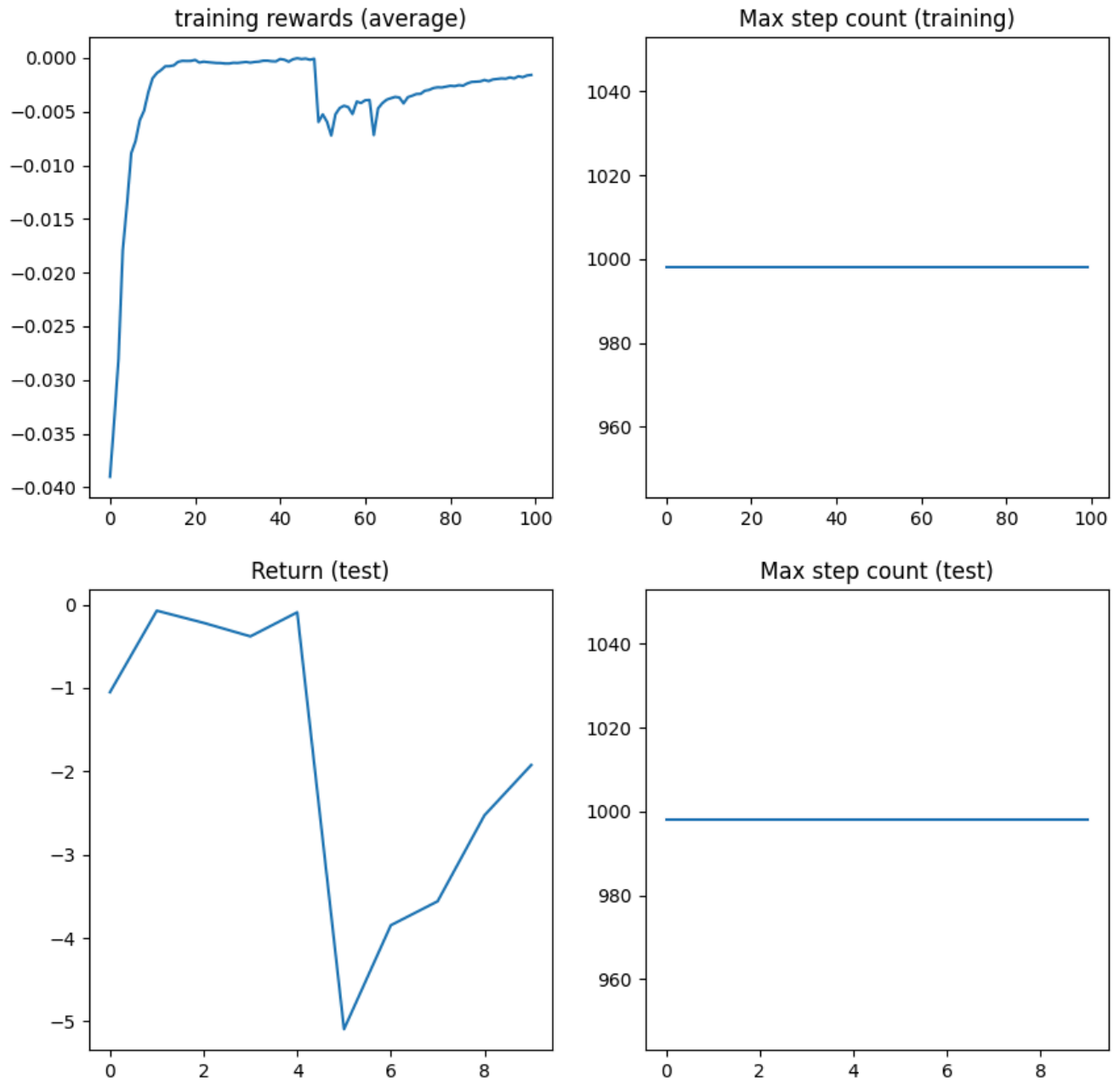
```

```

plt.figure(figsize=(10, 10))
plt.subplot(2, 2, 1)
plt.plot(logs["reward"])
plt.title("training rewards (average)")
plt.subplot(2, 2, 2)

```

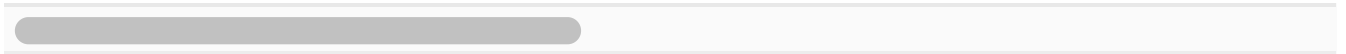
```
plt.plot(logs["step_count"])
plt.title("Max step count (training)")
plt.subplot(2, 2, 3)
plt.plot(logs["eval reward (sum)"])
plt.title("Return (test)")
plt.subplot(2, 2, 4)
plt.plot(logs["eval step_count"])
plt.title("Max step count (test)")
plt.show()
```



```
base_env = GymEnv("MountainCarContinuous-v0", device=device,render_mode="human")
```

```
env = TransformedEnv(
    base_env,
    Compose(
        ObservationNorm(in_keys=["observation"]),
        DoubleToFloat(),
        StepCounter(),
```

```
    ),  
    )  
env.transform[0].init_stats(num_iter=1000, reduce_dim=0, cat_dim=0)  
  
⇒ eval cumulative reward: -1.9249 (init: -1.0513), eval step-count: 998, average reward=-0.0016
```



```
with torch.no_grad():  
    env.rollout(  
        max_steps=10000,  
        policy=policy_module,  
        callback=lambda env, _ : env.render(),  
        auto_cast_to_device=True,  
        break_when_any_done=False,  
    )
```

Source for the PPO implementation [Vincent Moens](#)

✓ References

Implementation code of DQN and PPO are from [Vincent Moens](#) and [Adam Paszke](#), and are appropriately marked after the corresponding cells.

Licensed under [CC BY-NC-ND 4.0](#). © Zoltán Barta, 2025.

✓ 8. Practice - Introduction to Multi-Agent Reinforcement Learning

👤 Zoltán Barta, PhD student, Department of Artificial Intelligence

🕒 90 min read

📅 January 22, 2025

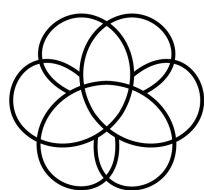
📚 Collective Intelligence



ELTE | IK
INFORMATIKAI KAR



DEPARTMENT
OF ARTIFICIAL
INTELLIGENCE



Gymnasium

This practice notebook provides a structured introduction to multi-agent reinforcement learning (MARL), focusing on the challenges and design principles that arise when multiple agents interact within a shared environment. Beginning with the foundational elements of MARL, including agent definitions, state and observation spaces, and reward structures, the notebook builds conceptual links between classical game theory and contemporary MARL frameworks.

Students will explore both centralized and independent learning paradigms, implementing and evaluating Q-learning algorithms in cooperative multi-agent environments. Practical exercises are designed to highlight key issues such as credit assignment, non-stationarity, coordination, and scalability. By the end of this module, students will be able to formalize multi-agent problems, select appropriate learning approaches, and critically analyze the trade-offs involved in designing and training MARL systems.

Table of Contents

- **8.1 Main Elements of MARL**
 - 8.1.1 Elements of a POMDP
 - 8.1.2 From Game Theory to MARL
 - 8.1.3 Cooperative
 - 8.1.4 Competitive
 - 8.1.5 Mixed
 - 8.1.6 From Single-Agent RL to MARL
- **8.2 Centralized Learning**
- **8.3 Implementing a Centralized Q-Learning Training Loop for a Multi-Agent Foraging Environment**
 - 8.3.1 Objectives
 - 8.3.2 Assumptions
 - 8.3.3 Challenges of Centralized Learning

- **8.4 Independent Learning**
- **8.5 Implementing an Independent Q-Learning Training Loop for a Multi-Agent Foraging Environment**
 - 8.5.1 Objectives
 - 8.5.2 Assumptions
 - 8.5.3 Challenges of Independent Learning
- **References**

✓ 8.1 Main elements of MARL

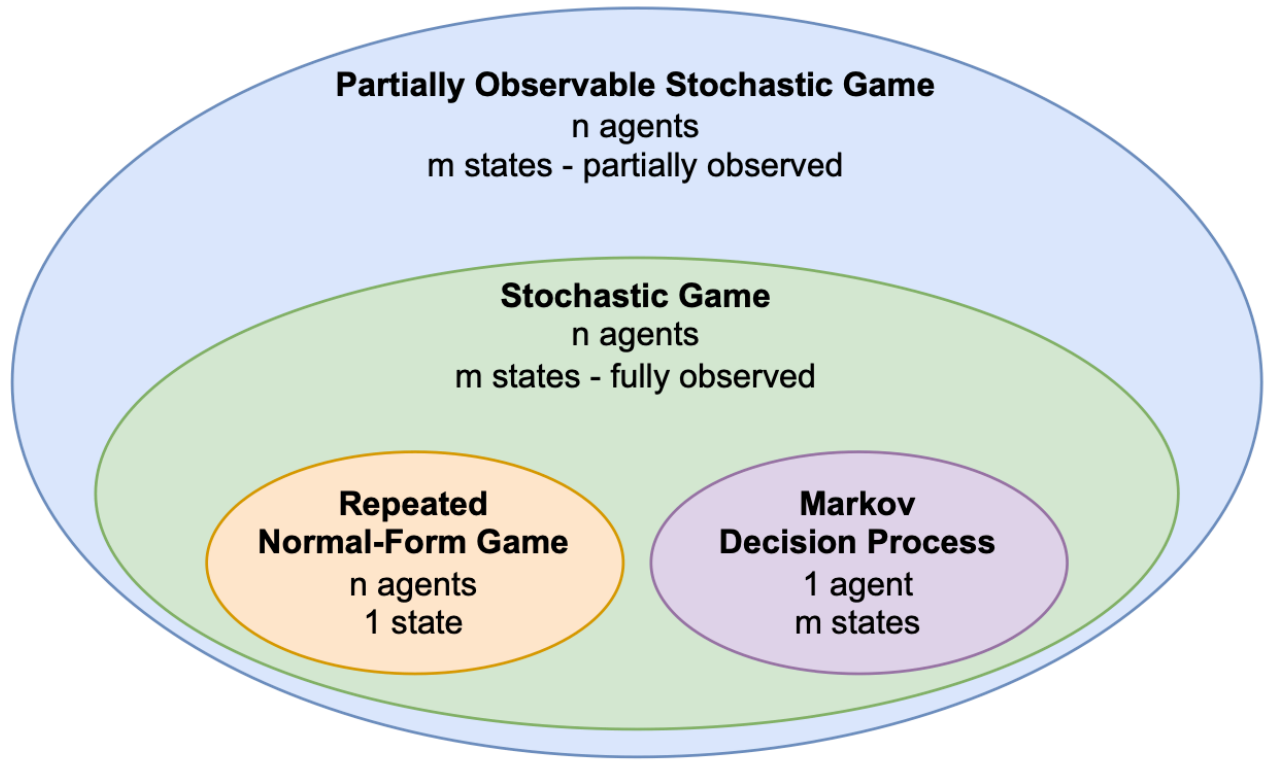
Reinforcement Learning (RL) as discussed so far operates under the framework of **Markov Decision Processes (MDPs)**, where a single agent interacts with an environment to maximize its cumulative rewards. In this setting, the agent has access to a well-defined state and takes actions that directly influence state transitions and rewards.

In contrast, multi-agent scenarios like **Repeated Normal-Form Games (RNFGs)** introduce interactions between multiple agents. These games involve repeated interactions, where agents adapt their strategies based on the outcomes of previous rounds. A well-known example is [Axelrod's Tournament](#), where strategies like "Tit-for-Tat" were tested in an iterated Prisoner's Dilemma, highlighting how agents can develop cooperative or competitive behaviors over time. ([What Game Theory Reveals About Life, The Universe, and Everything](#))

While standard RL focuses on a single agent operating under a **Markov Decision Process (MDP)**, Multi-Agent Reinforcement Learning (MARL) extends this to environments involving multiple agents, often modeled as **Partially Observable Markov Decision Processes (POMDPs)** or **Partially Observable Multi-Agent Decision Processes (POMDPs)**. Here, agents must make decisions based on limited and possibly noisy observations, as they typically lack full knowledge of the environment or the states of other agents.

This framework aligns with real-world scenarios like autonomous driving or collaborative robotics, where agents operate with incomplete information and must learn to coordinate actions or compete effectively. These complex settings will form the basis for the MARL algorithms and methods we explore.

- **Normal-form games:** Represent single-shot interactions where agents choose actions simultaneously, and outcomes are determined immediately.
- **Repeated games:** Extend normal-form games across multiple rounds, allowing agents to adjust strategies over time.
- **Stochastic games:** Model sequential interactions where agents' actions influence the state of the environment dynamically.
- **Partially observable stochastic games:** Add uncertainty to stochastic games by limiting what agents can observe about the environment or each other.



8.1.1 Elements of a POMDP

1. State Space:

- The set of all possible global states of the environment is denoted as:

$$S$$

2. Agent Set:

- The set of all agents in the environment, indexed by i , is:

$$I = \{1, 2, \dots, n\}$$

3. Action Space:

- Each agent i has its own action space A_i .
- The joint action space is represented as:

$$A = A_1 \times A_2 \times \dots \times A_{|I|}$$

4. Observation Space:

- Each agent i has an observation space O_i .
- The probability of an agent receiving a specific observation given the state and joint action is:

$$O(o_i \mid s, a) = P(o_i \mid s, a)$$

where $o_i \in O_i$, $s \in S$, and $a \in A$.

5. Transition Function:

- The probability of transitioning from one state to another given a joint action is:

$$T(s' \mid s, a) = P(s' \mid s, a)$$

where $s, s' \in S$ and $a \in A$.

6. Reward Function:

- The reward for agent i based on the state and joint action is defined as:

$$R_i(s, a) : S \times A \rightarrow \mathbb{R}$$

7. Policy:

- The policy of agent i maps its observations to a probability distribution over actions:

$$\pi_i(a_i | o_i) = P(a_i | o_i)$$

where $a_i \in A_i$.

8. Discount Factor:

- The weighting of future rewards is determined by the discount factor:

$$\gamma \in [0, 1]$$

The cumulative reward for agent i is:

$$U_i = \sum_{t=0}^{\infty} \gamma^t r_i^t$$

where r_i^t is the reward received by agent i at time t .

8.1.2 From Game Theory to MARL

MARL	Game Theory	Description
environment	game	Model specifying actions, observations, rewards, and state dynamics.
agent	player	An entity making decisions; also refers to roles, e.g., "row player" in a matrix game, "white player" in chess.
reward	payoff, utility	Scalar value received after taking an action.
policy	strategy	Function assigning probabilities to actions; "(pure) strategy" sometimes refers to specific actions.
deterministic X	pure X	X assigns probability 1 to one option, e.g., deterministic policy, pure strategy, pure Nash equilibrium.
probabilistic X	mixed X	X assigns probabilities ≤ 1 to options, e.g., probabilistic policy, mixed strategy, mixed Nash equilibrium.
joint X	X profile	X is a tuple with one element per agent/player, e.g., joint reward, joint policy, payoff profile, pure strategy profile.

8.1.3 Cooperative

A cooperative setting refers to a scenario where all agents share a common goal and work together to optimize a joint objective. In this context, agents collaborate to maximize a shared reward function, meaning their success is interconnected and dependent on coordinated efforts. In cooperative MARL, reward design is critical for fostering effective teamwork among agents. By aligning the agents' rewards with the overall system objectives, collaborative behavior can be encouraged. A popular method is to use shared rewards, where all agents receive the same reward signal based on the team's collective performance. This approach simplifies coordination but introduces challenges such as credit assignment, where identifying which agents contributed to the reward is difficult. On the other hand, using individual rewards tailored to specific agent roles can help clarify contributions but risks misaligning incentives, potentially leading to conflicting behaviors. Effective reward design often requires balancing these approaches, sometimes incorporating mechanisms like difference rewards to isolate individual contributions while maintaining a shared objective.

8.1.4 Competitive

In competitive settings, reward design in MARL focuses on encouraging agents to maximize their individual objectives, often in opposition to others. Rewards are typically structured as individual payoffs, reflecting each

agent's success relative to its competitors. For example, in zero-sum games, one agent's gain is exactly offset by the losses of others, directly incentivizing adversarial strategies.

The design of rewards in these settings must ensure that the competition remains meaningful and drives learning toward optimal strategies. However, challenges arise in balancing exploration and exploitation since agents must learn to anticipate and counteract the evolving strategies of opponents. In some cases, shaping rewards to emphasize strategic depth or penalize overly aggressive tactics can lead to more robust and adaptive agent behaviors. Ultimately, reward design in competitive MARL aims to refine agents' abilities to perform well under adversarial conditions while maintaining a stable learning process.

8.1.5 Mixed

In mixed settings, reward design in MARL must address the dual nature of cooperation and competition among agents. Agents may have partially aligned goals, leading to situations where they must balance collaborative behaviors with self-interested actions. Rewards in such environments often combine shared components, encouraging cooperation, with individual incentives that promote competitive behavior when necessary.

A key challenge in mixed settings is avoiding reward structures that overly favor one aspect, such as cooperation, at the expense of competitive dynamics, or vice versa. For example, in trading markets or resource allocation tasks, agents might need to collaborate to stabilize the system while competing for individual gains. Designing rewards that reflect both local contributions and global objectives helps agents navigate this duality. Advanced techniques, such as shaping rewards to emphasize synergies or penalize excessive self-interest, can ensure that the system operates effectively without biasing toward purely cooperative or competitive outcomes. Mixed settings require nuanced reward structures to balance collaboration and competition effectively.

✓ 8.1.6 From single agent RL to MARL

```
# Import some stuffs
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
from lbforaging.foraging import ForagingEnv
```

Q-Learning Class for utility

```
import numpy as np
import random
from collections.abc import Iterable

class QLearning:
    def __init__(
        self,
        state_shape,
        action_size,
        num_agents,
        alpha=0.1,
        gamma=0.99,
        epsilon=1.0,
        epsilon_decay=0.999,
        epsilon_min=0.1,
    ):
        """
        Initialize the Central Q-Learning algorithm.
        """
        self.state_shape = state_shape
```

```

self.action_size = action_size
self.num_agents = num_agents
self.alpha = alpha
self.gamma = gamma
self.epsilon = epsilon
self.epsilon_decay = epsilon_decay
self.epsilon_min = epsilon_min

# Initialize Q-table with zeros
self.q_table = {}

def get_state_key(self, state):
    """
    Convert the state into a string to use it as a key in the Q-table.
    """

    return tuple(state)

def choose_action(self, state):
    """
    Choose a joint action using an epsilon-greedy policy.
    """
    state_key = self.get_state_key(state)

    if random.random() < self.epsilon:
        # Random action for exploration
        actions = [random.randint(0, self.action_size - 1) for _ in range(self.num_agents)]
    else:
        # Choose the best joint action
        if state_key not in self.q_table:
            self.q_table[state_key] = np.zeros(self.action_size**self.num_agents)
        joint_action_index = np.argmax(self.q_table[state_key])
        actions = self.get_actions_from_index(joint_action_index)

    return actions

def get_actions_from_index(self, joint_action_index):
    """
    Convert a joint action index into individual actions.
    """
    actions = []
    for i in range(self.num_agents):
        actions.append(joint_action_index % self.action_size)
        joint_action_index //= self.action_size
    return actions

def get_joint_action_index(self, actions):
    """
    Convert a list of individual actions into a single joint action index.
    """
    if not isinstance(actions, Iterable):
        actions = [actions]
    joint_action_index = 0
    for i, action in enumerate(actions):
        joint_action_index += action * (self.action_size**i)
    return joint_action_index

def update_q_table(self, state, actions, reward, next_state):
    """
    Update the Q-table using the Q-learning update rule.
    """
    state_key = self.get_state_key(state)
    next_state_key = self.get_state_key(next_state)

```

```

if state_key not in self.q_table:
    self.q_table[state_key] = np.zeros(self.action_size**self.num_agents)
if next_state_key not in self.q_table:
    self.q_table[next_state_key] = np.zeros(self.action_size**self.num_agents)

joint_action_index = self.get_joint_action_index(actions)
best_next_action = np.argmax(self.q_table[next_state_key])

td_target = reward + self.gamma * self.q_table[next_state_key][best_next_action]
td_error = td_target - self.q_table[state_key][joint_action_index]

self.q_table[state_key][joint_action_index] += self.alpha * td_error

def decay_epsilon(self):
    """Decay the exploration rate."""
    if self.epsilon > self.epsilon_min:
        self.epsilon *= self.epsilon_decay

```

✓ 8.2 Centralized Learning

Centralized learning in MARL involves training a single central policy π_c , which considers the observations and actions of all agents in the system. This approach effectively reduces the multi-agent problem to a single-agent RL problem by treating the joint action space $A = A_1 \times A_2 \times \dots \times A_n$ as the action space of a single central agent. Reward transformation is essential to simplify the multi-agent reward structure. In common-reward games (fully cooperative settings), all agents receive the same reward $r = r_i$ for any agent i , making the aggregation straightforward. However, in general-sum games (mixed-motive settings), where agents may have conflicting or individual rewards, combining rewards into a single scalar often depends on the desired outcome. For example, one approach is to maximize social welfare, defined as the sum of all agents' rewards, encouraging collective performance while balancing individual goals.

Let's implement Central Q-Learning:

Algorithm 4 Central Q-learning (CQL) for stochastic games

- 1: Initialize: $Q(s, a) = 0$ for all $s \in S$ and $a \in A = A_1 \times \dots \times A_n$
 - 2: Repeat for every episode:
 - 3: **for** $t = 0, 1, 2, \dots$ **do**
 - 4: Observe current state s^t
 - 5: With probability ϵ : choose random joint action $a^t \in A$
 - 6: Otherwise: choose joint action $a^t \in \arg \max_a Q(s^t, a)$
 - 7: Apply joint action a^t , observe rewards r_1^t, \dots, r_n^t and next state s^{t+1}
 - 8: Transform r_1^t, \dots, r_n^t into scalar reward r^t
 - 9: $Q(s^t, a^t) \leftarrow Q(s^t, a^t) + \alpha [r^t + \gamma \max_{a'} Q(s^{t+1}, a') - Q(s^t, a^t)]$
-

```
n_agents = 2
```

```

env = ForagingEnv(
    players=n_agents,
    min_player_level=1,
    max_player_level=1,

```

```

min_food_level=2,
max_food_level=2,
field_size=(4,4),
max_episode_steps=10,
max_num_food=2,
sight=1,
force_coop=True,
)

```

8.3 Implementing a Centralized Q-Learning Training Loop for a Multi-Agent Foraging Environment

In this task, your goal is to implement a **centralized Q-learning** training loop for a multi-agent reinforcement learning environment called `ForagingEnv`. The environment consists of multiple agents who need to learn how to forage cooperatively.

Here a **centralized Q-table** is used to handle the **joint state and joint action space** across all agents.

8.3.1 Objectives

You need to write a function named `train_central_q_learning` that does the following:

1. **Initialize** the centralized Q-learning agent using:

- The shape of the joint observation space
- The number of discrete actions per agent
- The total number of agents

2. **Train over multiple episodes:**

- Reset the environment at the start of each episode
- Concatenate observations from all agents into a single joint state
- Use the centralized Q-agent to choose joint actions
- Perform the chosen actions in the environment
- Receive joint rewards and the next joint state
- Update the centralized Q-table based on the experience
- Accumulate the total reward for the episode
- Apply epsilon decay to reduce exploration over time

3. **Track performance:**

- Store the total reward per episode
- Print logging information every 5000 episodes, including:
 - The moving average of total rewards over the last 1000 episodes
 - The current value of epsilon

4. **Return** the trained Q-agent and the list of episode rewards.

✓ 8.3.2 Assumptions

- The environment follows the `gymnasium` (or `OpenAI Gym`) interface.
- States and actions for all agents are concatenated into joint representations.
- `QLearning` is a provided class that handles centralized Q-table logic.

```

def train_central_q_learning(env, episodes=500):
    state_shape = env.observation_space[0].shape
    action_size = env.action_space[0].n # Assuming discrete actions for each agent
    num_agents = n_agents

    central_q = QLearning(state_shape, action_size, num_agents)
    rewards = []
    for episode in range(episodes):
        state, info = env.reset()
        state = np.concatenate(state).tolist()
        total_reward = 0

        for step in range(env._max_episode_steps):
            # Choose a joint action

            actions = central_q.choose_action(state)

            next_state, reward, done, truncated, info = env.step(actions)
            reward = sum(reward) # Sum of rewards for all agents
            total_reward += reward

            # Update the Q-table

            next_state = np.concatenate(next_state).tolist()
            central_q.update_q_table(state, actions, reward, next_state)

            state = next_state

            if done:
                break

        # Decay epsilon
        central_q.decay_epsilon()
        rewards.append(total_reward)
        # Log progress
        if episode % 5000 == 0:
            print(f"Episode {episode + 1}/{episodes}, Moving Average Reward (1000 episodes): {sum(rewards[-1000:])/1000}")

    return central_q, rewards

```

```

algo, rewards = train_central_q_learning(env, episodes=20000)

```

```

⇒ Episode 1/20000, Moving Average Reward (1000 episodes): 0.0, Epsilon: 0.999
   Episode 5001/20000, Moving Average Reward (1000 episodes): 1.853, Epsilon: 0.100
   Episode 10001/20000, Moving Average Reward (1000 episodes): 5.053, Epsilon: 0.100
   Episode 15001/20000, Moving Average Reward (1000 episodes): 8.316, Epsilon: 0.100

```

```

import time
import random
def evaluate_central_q_learning(env, central_q, render=True):
    """
    Evaluate the trained Central Q-Learning agent by rendering an episode.
    """
    state, info = env.reset() # Reset the environment to the initial state
    state = np.concatenate(state).tolist()
    total_reward = 0
    done = False

    while not done:
        # Choose the best action based on the Q-table
        state_key = central_q.get_state_key(state)

        # If the state is unknown, take a random action

```

```

if state_key not in central_q.q_table:
    actions = [random.randint(0, len(env.action_set)-1) for _ in range(central_q.num_agents)]
else:
    joint_action_index = np.argmax(central_q.q_table[state_key])
    actions = central_q.get_actions_from_index(joint_action_index)

# Take the action in the environments
next_state, reward, done, truncated, info = env.step(actions)
next_state = np.concatenate(next_state).tolist()
reward = sum(reward)
total_reward += reward

# Render the environment after each step
if render:
    env.render()
    time.sleep(0.5) # Adjust pause duration for visualization

# Move to the next state
state = next_state

```

```
print(f"Total Reward in Evaluation Episode: {total_reward}")
```

```
evaluate_central_q_learning(env, algo, render=True)
```

➡ Total Reward in Evaluation Episode: 1.0

```
import pandas as pd
import matplotlib.pyplot as plt
```

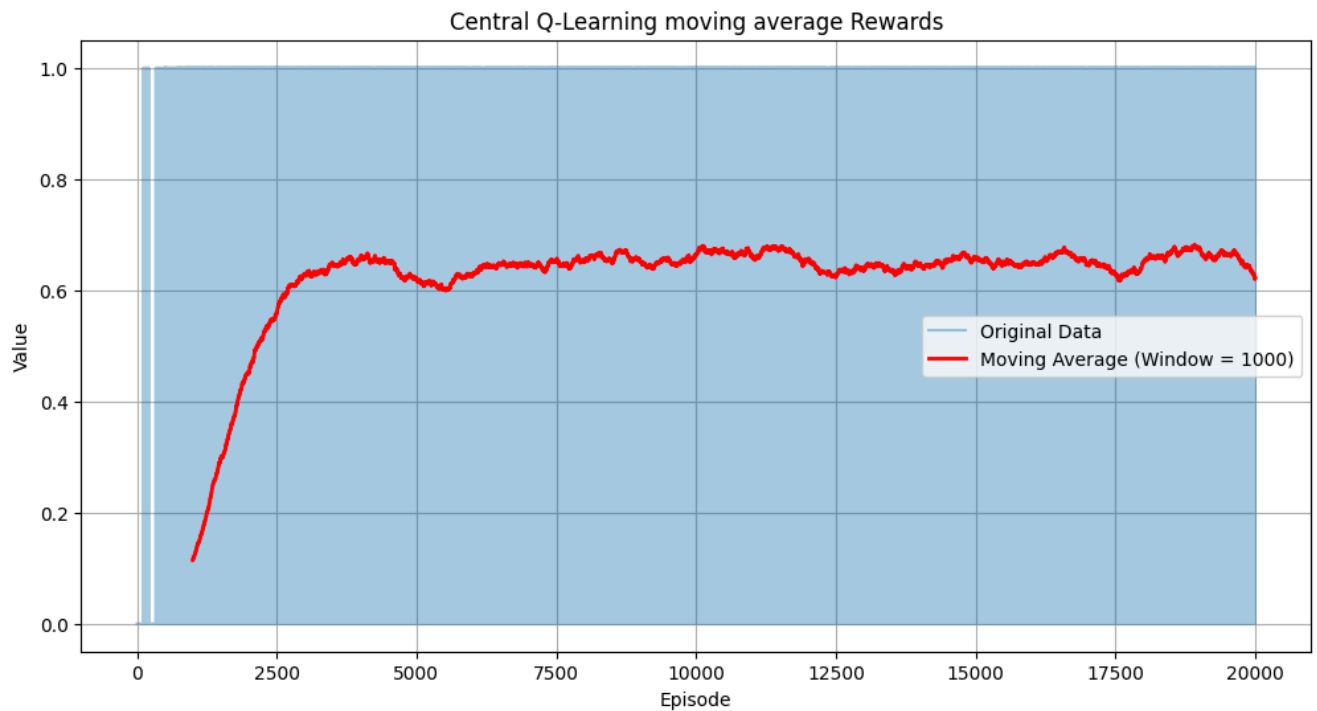
```
def plot_moving_average(data, window_size=1000, title="Moving Average of Data", xlabel="Index", ylabel="Value"):
    """
    Plot the original data and its moving average.

    Parameters:
        data (list or array-like): The input data series (e.g., 0s and 1s).
        window_size (int): The size of the moving average window.
        title (str): The title of the plot.
        xlabel (str): Label for the x-axis.
        ylabel (str): Label for the y-axis.
    """
    # Calculate the moving average
    moving_avg = pd.Series(data).rolling(window=window_size).mean()

    # Plot the original data and the moving average
    plt.figure(figsize=(12, 6))
    plt.plot(data, alpha=0.4, label="Original Data")
    plt.plot(moving_avg, label=f"Moving Average (Window = {window_size})", linewidth=2, color='r')
    plt.xlabel(xlabel)
    plt.ylabel(ylabel)
    plt.title(title)
    plt.legend()
    plt.grid()
    plt.show()

```

```
plot_moving_average(rewards, window_size=1000, title="Central Q-Learning moving average Rewards")
```



8.3.3 Challenges of Centralized Learning

1. Joint-Action Space:

- The size of the joint action space grows exponentially with the number of agents, making centralized learning computationally expensive.

2. Scalability:

- For environments with many agents, learning a centralized policy becomes infeasible due to the scale and complexity of the action space.

3. Distributed Agents:

- In real-world systems, agents are often physically or virtually distributed, limiting the feasibility of centralized control during execution.

✓ 8.4 Independent Learning

Independent Learning in MARL simplifies the multi-agent problem by allowing each agent to learn its own policy independently, treating other agents' actions as part of the environment dynamics. This decentralized approach leverages single-agent RL algorithms, enabling each agent to update its policy based solely on its local observations, actions, and rewards. By avoiding the need for centralized coordination, IL reduces computational complexity and eliminates the need for managing joint action spaces, making it scalable for large systems. Additionally, it is particularly suited to distributed environments where agents operate with limited communication or autonomy. This framework ensures that agents can independently adapt to their surroundings without requiring global synchronization or shared information.

The simplicity and flexibility of independent learning make it an attractive approach in MARL. It is easy to implement using existing single-agent RL frameworks, requiring no complex reward aggregation or global state observation. IL works well in scenarios where explicit coordination is unnecessary, offering a practical and scalable solution for multi-agent problems. By allowing agents to focus on local decision-making, it provides a robust foundation for environments where communication between agents is infeasible or restricted. These benefits often make IL a strong baseline in MARL research, balancing simplicity and performance effectively.

Let's implement Independent Q-Learning:

Algorithm 5 Independent Q-learning (IQL) for stochastic games

```

// Algorithm controls agent i
1: Initialize:  $Q_i(s, a_i) = 0$  for all  $s \in S, a_i \in A_i$ 
2: Repeat for every episode:
3:   for  $t = 0, 1, 2, \dots$  do
4:     Observe current state  $s^t$ 
5:     With probability  $\epsilon$ : choose random action  $a_i^t \in A_i$ 
6:     Otherwise: choose action  $a_i^t \in \arg \max_{a_i} Q_i(s^t, a_i)$ 
7:     (meanwhile, other agents  $j \neq i$  choose their actions  $a_j^t$ )
8:     Observe own reward  $r_i^t$  and next state  $s^{t+1}$ 
9:      $Q_i(s^t, a_i^t) \leftarrow Q_i(s^t, a_i^t) + \alpha [r_i^t + \gamma \max_{a_i'} Q_i(s^{t+1}, a_i') - Q_i(s^t, a_i^t)]$ 

```

```
n_agents = 2
```

```

env = ForagingEnv(
    players=n_agents,
    min_player_level=1,
    max_player_level=1,
    min_food_level=2,
    max_food_level=2,
    field_size=(4,4),
    max_episode_steps=10,
    max_num_food=2,
    sight=1,
    force_coop=True,
)

```

8.5 Implementing an Independent Q-Learning Training Loop for a Multi-Agent Foraging Environment

In this task, your goal is to implement an **independent Q-learning** training loop for a multi-agent reinforcement learning environment called `ForagingEnv`. The environment consists of multiple agents who must learn to forage, but in this case, each agent learns **independently**, using its own Q-table and local observations.

This decentralized approach allows each agent to act and learn on its own, without knowledge of the other agents' internal policies or Q-values.

8.5.1 Objectives

You need to write a function named `train_independent_q_learning` that does the following:

1. **Initialize** a separate Q-learning agent for each environment agent using:

- The shape of the individual observation space
- The number of discrete actions per agent
- One Q-table per agent (independent learning)

2. **Train over multiple episodes:**

- Reset the environment at the start of each episode
- Each agent selects an action based on its own current state
- Perform the joint actions in the environment
- Each agent receives its own reward and next state
- Each agent updates its own Q-table based on its experience
- Accumulate the total reward for the episode
- Apply epsilon decay individually to each agent to reduce exploration over time

3. **Track performance:**

- Store the total reward per episode
- Print logging information every 5000 episodes, including:
 - The moving average of total rewards over the last 1000 episodes

4. **Return** the list of trained Q-agents and the list of episode rewards.

✓ 8.5.2 Assumptions

- The environment follows the `gymnasium` (or `OpenAI Gym`) interface.
- Each agent uses only its own observation and reward signal.
- `QLearning` is a provided class that handles Q-table logic and epsilon-greedy action selection.

```
# Training Loop for ForagingEnv
def train_independent_q_learning(env, episodes=500):
    state_shape = env.observation_space[0].shape
    action_size = env.action_space[0].n # Assuming discrete actions for each agent
    num_agents = n_agents

    agents = [QLearning(state_shape, action_size, 1) for _ in range(num_agents)]
    rewards = []
    for episode in range(episodes):
        state, inf = env.reset()
        total_reward = 0

        for step in range(env._max_episode_steps):
            # Choose a joint action

            actions = []
            for idx, agent in enumerate(agents):
                action = agent.choose_action(state[idx])
                actions.append(action[0])

            next_state, reward, done, truncated, info = env.step(actions)

            # Update the Q-table

            for idx, agent in enumerate(agents):
                agent.update_q_table(state[idx], actions[idx], reward[idx], next_state[idx])
```

```

        reward = sum(reward) # Sum of rewards for all agents

        total_reward += reward

        state = next_state

        if done:
            break

    # Decay epsilon
    for agent in agents:
        agent.decay_epsilon()
    rewards.append(total_reward)
    # Log progress
    if episode % 5000 == 0:
        print(f"Episode {episode + 1}/{episodes}, Moving Average Reward (1000 episodes): {sum(rewards)/1000}")

    return agents, rewards

```

```
agents, rewards = train_independent_q_learning(env, 20000)
```

```

➡ Episode 1/20000, Moving Average Reward (1000 episodes): 0.0
  Episode 5001/20000, Moving Average Reward (1000 episodes): 1.728
  Episode 10001/20000, Moving Average Reward (1000 episodes): 3.989
  Episode 15001/20000, Moving Average Reward (1000 episodes): 6.023

```

```

import time
import random

```

```

def evaluate_independent_q_learning(env, agents, render=True):
    """
    Evaluate the trained Independent Q-Learning agents by rendering an episode.
    """
    state, info = env.reset() # Reset the environment to the initial state
    total_reward = 0
    done = False
    while not done:
        actions = []

        # Each agent selects its action based on its own Q-table
        for idx, agent in enumerate(agents):
            state_key = agent.get_state_key(state[idx])

            # If the state is unknown to the agent, take a random action
            if state_key not in agent.q_table:
                action = random.randint(0, len(env.action_set) - 1)
            else:
                action = np.argmax(agent.q_table[state_key])

            actions.append(action)

        # Take the actions in the environment
        next_state, rewards, done, truncated, info = env.step(actions)
        total_reward += sum(rewards) # Aggregate rewards for all agents

        # Render the environment after each step
        if render:
            env.render()
            time.sleep(0.5) # Adjust pause duration for visualization

        # Move to the next state
        state = next_state

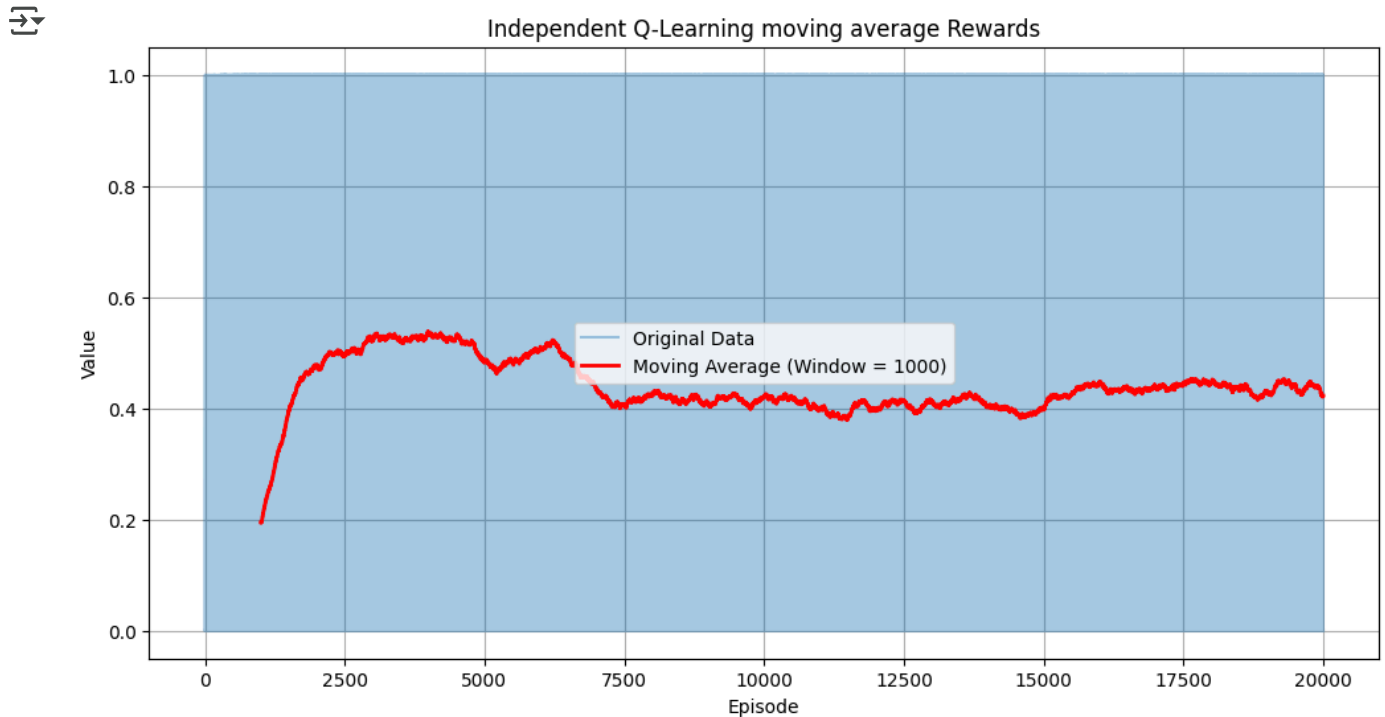
```

```
print(f"Total Reward in Evaluation Episode: {total_reward}")
```

```
evaluate_independent_q_learning(env, agents, render=True)
```

```
➡ Total Reward in Evaluation Episode: 0.0
```

```
plot_moving_average(rewards, window_size=1000, title="Independent Q-Learning moving average Rewa
```



8.5.3 Challenges of Independent Learning

1. Non-Stationarity:

- The environment dynamics appear non-stationary to each agent due to the evolving policies of other agents.
- This can lead to unstable learning or suboptimal convergence.

2. Coordination:

- Independent Learning (IL) struggles in cooperative tasks requiring tight agent coordination since it lacks explicit mechanisms to model or align with others' actions.

3. Ambiguity in Dynamics:

- Agents cannot distinguish between stochastic environment changes and changes caused by other agents' policies, adding to uncertainty during learning.

✓ References

- Multi-Agent Reinforcement Learning: Foundations and Modern Approaches - Stefano V. Albrecht, Filippos Christianos, Lukas Schäfe - <https://www.marl-book.com>
 - Chapter 3 - 4
 - Chapter 5.3
 - Chapter 9
- Level Based Foraging - Filippos Christianos - <https://github.com/semitable/lb-foraging>

Licensed under [CC BY-NC-ND 4.0](https://creativecommons.org/licenses/by-nc-nd/4.0/). © Zoltán Barta, 2025.

✓ 9. Practice - Introduction to TorchRL

👤 Zoltán Barta, PhD student, Department of Artificial Intelligence

🕒 90 min read

📅 January 22, 2025

📖 Collective Intelligence



ELTE | IK
INFORMATIKAI KAR



DEPARTMENT
OF ARTIFICIAL
INTELLIGENCE



TorchRL

This practice notebook provides a hands-on introduction to TorchRL, an advanced reinforcement learning library built on top of PyTorch. TorchRL streamlines the RL workflow by offering modular, GPU-accelerated components for environment handling, data management, and model development. In this notebook, students will learn how to utilize core data structures like `TensorDict`, efficiently manage experience replay buffers, interface with vectorized and transformed environments, and construct composable neural network modules tailored for RL tasks.

[TorchRL Paper](#)

Table of Contents

- **9.1 Necessary Imports**
- **9.2 Data Handling**
 - 9.2.1 TensorDict
 - 9.2.2 Replay Buffers
- **9.3 Environments**
- **9.4 Modules**
 - 9.4.1 Specialized Classes
- **9.5 Combining Environments and Modules**
- **9.6 Objectives**
- **References**

✓ 9.1 Necessary Imports

```
import torch
from torch import nn
from tensordict import TensorDict
from tensordict.nn import TensorDictSequential, TensorDictModule, ProbabilisticTensorDictModule,
from tensordict import from_module

from torchrl.data import PrioritizedReplayBuffer, ReplayBuffer, TensorDictPrioritizedReplayBuffer
from torchrl.modules import ConvNet, MLP, Actor, SafeModule, NormalParamExtractor, TanhNormal
from torchrl.modules.models.utils import SquashDims
from torchrl.envs.utils import step_mdp, ExplorationType, set_exploration_type

try:
    import gymnasium as gym
except ModuleNotFoundError:
    import gym

from torchrl.envs.libs.gym import GymEnv, GymWrapper, set_gym_backend
from torchrl.envs import (
    Compose,
    ObservationNorm,
    ToTensorImage,
    TransformedEnv,
    ParallelEnv,
    Resize,
    RewardScaling,
    CatFrames,
    VecNorm
)

torch.manual_seed(0)

🔗 <torch._C.Generator at 0x111594270>
```

✓ 9.2 Data Handling

9.2.1 TensorDict

The `TensorDict` is a core data structure in **TorchRL**, designed to store and manage batched data from environments in a structured and efficient way. It functions like a dictionary where each key maps to a tensor, all sharing the same leading batch dimensions. `TensorDict` enables seamless integration with TorchRL components and facilitates vectorized operations across complex reinforcement learning pipelines.

We can initialize a `Tensordict` multiple ways:

```
batch_size = 10

data_from_dict = TensorDict({
    'key1': torch.rand(batch_size, 3),
    'key2': torch.zeros(batch_size, 4, 5, dtype=torch.bool),
    'key3': torch.zeros(batch_size, 6, 7, dtype=torch.bool),
},
    batch_size=[batch_size],
)
print('Initilize by using dicts:')
print(data_from_dict)

data = TensorDict(
```

```

    key1=torch.rand(batch_size, 3),
    key2=torch.zeros(batch_size, 4, 5, dtype=torch.bool),
    batch_size=[batch_size],
)
print("Initialize by using keyword arguments:")
print(data)

```

⇒ Initilize by using dicts:

```

TensorDict(
  fields={
    key1: Tensor(shape=torch.Size([10, 3]), device=cpu, dtype=torch.float32, is_shared=False),
    key2: Tensor(shape=torch.Size([10, 4, 5]), device=cpu, dtype=torch.bool, is_shared=False),
    key3: Tensor(shape=torch.Size([10, 6, 7]), device=cpu, dtype=torch.bool, is_shared=False),
    batch_size=torch.Size([10]),
    device=None,
    is_shared=False)
Initialize by using keyword arguments:
TensorDict(
  fields={
    key1: Tensor(shape=torch.Size([10, 3]), device=cpu, dtype=torch.float32, is_shared=False),
    key2: Tensor(shape=torch.Size([10, 4, 5]), device=cpu, dtype=torch.bool, is_shared=False),
    batch_size=torch.Size([10]),
    device=None,
    is_shared=False)

```

Tensordict values are accessible through indicies (of batch_size) and keys as well.

```

print('Accessing data using keys, returns the tensor:')
print(data['key1'])
print('Accessing data using indices, returns a tensordict')
print(data[2])

```

⇒ Accessing data using keys, returns the tensor:

```

tensor([[0.3051, 0.9320, 0.1759],
        [0.2698, 0.1507, 0.0317],
        [0.2081, 0.9298, 0.7231],
        [0.7423, 0.5263, 0.2437],
        [0.5846, 0.0332, 0.1387],
        [0.2422, 0.8155, 0.7932],
        [0.2783, 0.4820, 0.8198],
        [0.9971, 0.6984, 0.5675],
        [0.8352, 0.2056, 0.5932],
        [0.1123, 0.1535, 0.2417]])
Accessing data using indices, returns a tensordict
TensorDict(
  fields={
    key1: Tensor(shape=torch.Size([3]), device=cpu, dtype=torch.float32, is_shared=False),
    key2: Tensor(shape=torch.Size([4, 5]), device=cpu, dtype=torch.bool, is_shared=False),
    batch_size=torch.Size([]),
    device=None,
    is_shared=False)

```

TensorDict often returns views or copies of the data, depending on the operation.

```

print(data['key1'][2]) # This accesses the full tensor under 'key1' and then indexes the tensor
print(data[2]['key1']) # This first extracts a row-slice (like a mini tensordict) at index 2, th
print(data[2]['key1'] is data['key1'][2]) #
print(data[2]['key1'] == data['key1'][2])

```

⇒ tensor([0.2081, 0.9298, 0.7231])
 tensor([0.2081, 0.9298, 0.7231])
 False

```
tensor([True, True, True])
```

Assigning new keys is the same as for dictionaries.

```
data['key3'] = torch.rand(batch_size, 6,7)
print(data)
```

```
⇒ TensorDict(
  fields={
    key1: Tensor(shape=torch.Size([10, 3]), device=cpu, dtype=torch.float32, is_shared=False),
    key2: Tensor(shape=torch.Size([10, 4, 5]), device=cpu, dtype=torch.bool, is_shared=False),
    key3: Tensor(shape=torch.Size([10, 6, 7]), device=cpu, dtype=torch.float32, is_shared=False),
    batch_size=torch.Size([10]),
    device=None,
    is_shared=False)
```

You can use regular torch operations on tensordicts, as for tensors, if their structure is the same. For example: stack, permute, view, expand, to, etc...

```
data = torch.stack([data, data_from_dict], 0)
print(
```

```
    "Data stack:",
    data.batch_size,
    data.get("key1").shape,)
```

```
print(
    "Data view(-1): ",
    data.view(-1).batch_size,
    data.view(-1).get("key1").shape,
)
```

```
print("Data to device: ", data.to("cpu"))
```

```
print(
    "Data permute(1, 0): ",
    data.permute(1, 0).batch_size,
    data.permute(1, 0).get("key1").shape,
)
```

```
print(
    "Data expand: ",
    data.expand(3, *data.batch_size).batch_size,
    data.expand(3, *data.batch_size).get("key1").shape,
)
```

```
⇒ Data stack: torch.Size([2, 10]) torch.Size([2, 10, 3])
Data view(-1): torch.Size([20]) torch.Size([20, 3])
Data to device: TensorDict(
  fields={
    key1: Tensor(shape=torch.Size([2, 10, 3]), device=cpu, dtype=torch.float32, is_shared=False),
    key2: Tensor(shape=torch.Size([2, 10, 4, 5]), device=cpu, dtype=torch.bool, is_shared=False),
    key3: Tensor(shape=torch.Size([2, 10, 6, 7]), device=cpu, dtype=torch.float32, is_shared=False),
    batch_size=torch.Size([2, 10]),
    device=cpu,
    is_shared=False)
Data permute(1, 0): torch.Size([10, 2]) torch.Size([10, 2, 3])
Data expand: torch.Size([3, 2, 10]) torch.Size([3, 2, 10, 3])
```

Source: [TorchRL documentation](#)

✓ 9.2.2 Replay Buffers

The **replay buffer** is a memory module used to store past transitions (s, a, r, s') collected by the agent during interaction with the environment. It enables efficient and stable learning by allowing the agent to sample random minibatches of past experiences, breaking temporal correlations and improving sample efficiency. TorchRL provides a built-in, flexible **ReplayBuffer** module that supports batched sampling, storage limits, and seamless integration with other TorchRL components. The `collate_fn` parameter used to define how individual data samples are combined into batches when sampling.

```
rb = ReplayBuffer(collate_fn=lambda x: x)

rb.add(1)
print('Sampling 1 element from the buffer:', rb.sample(1))
rb.extend([2, 3])
print('Sampling 3 elements from the buffer:', rb.sample(3))
```

```
➡ Sampling 1 element from the buffer: [1]
   Sampling 3 elements from the buffer: [3, 2, 2]
```

```
print('Showing the buffer storage:')
for i in rb.storage:
    print(i)
```

```
➡ Showing the buffer storage:
   1
   2
   3
```

Prioritized Replay Buffers can also be used. Instead of sampling transitions uniformly from the replay buffer, PER samples more important transitions more often — those with higher temporal-difference (TD) error, for instance. This can help your agent learn faster by focusing more on transitions that are surprising or useful.

alpha – *Controls prioritization strength*

- Determines how much prioritization is used when sampling transitions.
- `alpha = 0` means uniform sampling (no prioritization).
- `alpha = 1` means full prioritization based entirely on priority values.
- Typical values are between `0.4` and `0.7`.

beta – *Controls importance-sampling correction*

- Adjusts for the bias introduced by prioritized sampling.
- `beta = 0` applies no correction.
- `beta = 1` applies full correction to make training unbiased.
- Often annealed from `0.4` to `1.0` over training.

These parameters help balance learning speed and stability when using a prioritized replay buffer.

```
rb = PrioritizedReplayBuffer(alpha=0.7, beta=1.0, collate_fn=lambda x: x)
rb.add(1)
rb.sample(1)
rb.update_priority(1, 0.7)
```

Source: [TorchRL documentation](#)

This example demonstrates how to use TorchRL's `TensorDictPrioritizedReplayBuffer` to store samples, update priorities using a TD-error, and inspect the internal sum-tree structure used for prioritized sampling.

```
rb = TensorDictPrioritizedReplayBuffer(alpha=0.7, beta=1.1, priority_key="td_error")
rb.extend(TensorDict({"a": torch.randn(2, 3)}, batch_size=[2]))
data_sample = rb.sample(2).contiguous()
print(data_sample)

print(data_sample["index"])

data_sample["td_error"] = torch.rand(2)
rb.update_tensordict_priority(data_sample)

for i, val in enumerate(rb._sampler._sum_tree):
    print(i, val)
    if i == len(rb):
        break
```



```
TensorDict(
  fields={
    _weight: Tensor(shape=torch.Size([2]), device=cpu, dtype=torch.float32, is_shared=False),
    a: Tensor(shape=torch.Size([2, 3]), device=cpu, dtype=torch.float32, is_shared=False),
    index: Tensor(shape=torch.Size([2]), device=cpu, dtype=torch.int64, is_shared=False),
    batch_size: torch.Size([2]),
    device=None,
    is_shared=False)
  tensor([0, 0])
  0 0.356869637966156
  1 1.0
  2 0.0
```

Source: [TorchRL documentation](#)


✓ 9.3 Environments

TorchRL environments provide a unified and modular interface for interacting with both classic control tasks and complex simulators, fully compatible with PyTorch tensors and RL pipelines. You can find several `EnvWrappers` that you can apply to make your custom env compatible with torchRL. Environments return data in the form of a `TensorDict`, where each key corresponds to a named tensor (e.g., "obs", "action", "reward", "done"). This structure ensures that all data related to a single environment step is stored in a consistent and batch-friendly format. For example, after one step, you might get a `TensorDict` with keys like `{"observation": tensor, "action": tensor, "reward": tensor, "done": tensor}` — making it easy to index and transform.

```
# Initialize an environment by the original gym module and wrap it with the GymWrapper.
gym_env = gym.make("Pendulum-v1")
env = GymWrapper(gym_env)
```

```
# Initialize an environment by the GymEnv module.
env = GymEnv("Pendulum-v1")
data = env.reset()
data = env.rand_step(data)
```

```
print('An example of data returned by the environment:', data)
```



```
An example of data returned by the environment: TensorDict(
  fields={
```

```

action: Tensor(shape=torch.Size([1]), device=cpu, dtype=torch.float32, is_shared=False),
done: Tensor(shape=torch.Size([1]), device=cpu, dtype=torch.bool, is_shared=False),
next: TensorDict(
  fields={
    done: Tensor(shape=torch.Size([1]), device=cpu, dtype=torch.bool, is_shared=False),
    observation: Tensor(shape=torch.Size([3]), device=cpu, dtype=torch.float32, is_shared=False),
    reward: Tensor(shape=torch.Size([1]), device=cpu, dtype=torch.float32, is_shared=False),
    terminated: Tensor(shape=torch.Size([1]), device=cpu, dtype=torch.bool, is_shared=False),
    truncated: Tensor(shape=torch.Size([1]), device=cpu, dtype=torch.bool, is_shared=False),
    batch_size: torch.Size([1]),
    device: None,
    is_shared: False),
  observation: Tensor(shape=torch.Size([3]), device=cpu, dtype=torch.float32, is_shared=False),
  terminated: Tensor(shape=torch.Size([1]), device=cpu, dtype=torch.bool, is_shared=False),
  truncated: Tensor(shape=torch.Size([1]), device=cpu, dtype=torch.bool, is_shared=False),
  batch_size: torch.Size([1]),
  device: None,
  is_shared: False)

```

Source: [TorchRL documentation](#)

Transforms are modular preprocessing components that can be composed to modify observations, rewards, actions, or states before they are passed to the agent. These are especially useful for building data pipelines in a clean and reusable way, and they integrate seamlessly with TorchRL environments via `TransformedEnv`.

Some commonly used transforms include:

- `CatFrames`: stacks multiple consecutive frames to give the agent temporal context (useful in Atari environments).
- `Resize`: resizes image observations to a fixed shape.
- `NormalizeObservation`: normalizes input observations across episodes.
- `RewardScaling`: scales rewards to stabilize learning.
- `ObservationNorm`: tracks mean and std to normalize observations online.
- `ToTensorImage`: converts image observations to PyTorch tensors and permutes channels to match PyTorch's format.

Transforms can be composed in a list and applied as a pipeline, making environment design highly customizable.

```
base_env = GymEnv("CartPole-v1", from_pixels=True, frame_skip=2, pixels_only=False)
```

```
# Apply a sequence of transforms
```

```
env = TransformedEnv(
    base_env,
    Compose(
        Resize(84, 84), # resize pixel input
        ToTensorImage(), # convert image to tensor format
        ObservationNorm(in_keys=["pixels"]), # normalize pixel values
        RewardScaling(0.1, 0.0), # scale rewards
        CatFrames(N=4, in_keys=["pixels"], dim=-1) # stack 4 consecutive frames
    )
)
```

Vectorized environments enable multiple simulations to run in parallel, significantly improving data throughput and making training more efficient and stable. In TorchRL, vectorized environments can be created using `ParallelEnv` or `SyncVectorEnv`, wrapping multiple environment instances into a single batched interface.

```

def make_env():
    return GymEnv("Pendulum-v1", frame_skip=3, from_pixels=True, pixels_only=False)
env = make_env()

print('Before stacking:', env.observation_spec)

env = ParallelEnv(
    4,
    make_env,
)
print('After stacking:', env.observation_spec)
print('Notice that the whole tensordict gained a new dimension of size 4, which is the number of')

➡ Before stacking: Composite(
  observation: BoundedContinuous(
    shape=torch.Size([3]),
    space=ContinuousBox(
      low=Tensor(shape=torch.Size([3]), device=cpu, dtype=torch.float32, contiguous=True),
      high=Tensor(shape=torch.Size([3]), device=cpu, dtype=torch.float32, contiguous=True),
      device=cpu,
      dtype=torch.float32,
      domain=continuous),
  pixels: UnboundedDiscrete(
    shape=torch.Size([500, 500, 3]),
    space=ContinuousBox(
      low=Tensor(shape=torch.Size([500, 500, 3]), device=cpu, dtype=torch.uint8, contiguous=True),
      high=Tensor(shape=torch.Size([500, 500, 3]), device=cpu, dtype=torch.uint8, contiguous=True),
      device=cpu,
      dtype=torch.uint8,
      domain=discrete),
  device=None,
  shape=torch.Size([]))
After stacking: Composite(
  observation: BoundedContinuous(
    shape=torch.Size([4, 3]),
    space=ContinuousBox(
      low=Tensor(shape=torch.Size([4, 3]), device=cpu, dtype=torch.float32, contiguous=True),
      high=Tensor(shape=torch.Size([4, 3]), device=cpu, dtype=torch.float32, contiguous=True),
      device=cpu,
      dtype=torch.float32,
      domain=continuous),
  pixels: UnboundedDiscrete(
    shape=torch.Size([4, 500, 500, 3]),
    space=ContinuousBox(
      low=Tensor(shape=torch.Size([4, 500, 500, 3]), device=cpu, dtype=torch.uint8, contiguous=True),
      high=Tensor(shape=torch.Size([4, 500, 500, 3]), device=cpu, dtype=torch.uint8, contiguous=True),
      device=cpu,
      dtype=torch.uint8,
      domain=discrete),
  device=cpu,
  shape=torch.Size([4]))
Notice that the whole tensordict gained a new dimension of size 4, which is the number of parallel environments

```


Source: [TorchRL documentation](#)

Normalization helps stabilize training by ensuring that input features (like observations or rewards) have consistent statistical properties. While fixed normalization using precomputed statistics (e.g., from a random policy rollout) can be effective, in reinforcement learning, the environment is often unknown or only partially observable, making it difficult to precompute reliable normalization statistics. As a result, we may not know in advance the range or distribution of observations and rewards. This is where adaptive, on-the-fly normalization—such as with `VecNorm`—becomes especially useful, as it updates the normalization parameters based on the data.

collected during training. This allows the model to remain robust even as the environment dynamics or agent performance evolve.

```
env = TransformedEnv(GymEnv("Pendulum-v1"), VecNorm())
data = env.rollout(max_steps=100)

print("mean: :", data.get("observation").mean(0)) # Approx 0
print("std: :", data.get("observation").std(0)) # Approx 1
```



```
mean: : tensor([-0.1146, -0.3184, -0.1428])
std: : tensor([1.0420, 1.0927, 1.1277])
```

Source: [TorchRL documentation](#)

✓ 9.4 Modules


TorchRL provides a wide range of modular components designed to work seamlessly with `TensorDict` objects. These modules are built to accept a `TensorDict` as input, modify it (e.g., by adding value estimates, computing actions, or logging information), and return the updated `TensorDict`, enabling clean and composable workflows. Beyond environment interaction and data processing, TorchRL also includes neural network modules tailored for RL, such as multilayer perceptrons (MLPs) and Convolutional Neural Networks (CNNs), offering flexible tools for policy and value function approximation.

An example of initializing said modules:

```
net = MLP(num_cells=[32, 64], out_features=4, activation_class=nn.ELU)
print(net)
```



```
cnn = ConvNet(
    num_cells=[32, 64],
    kernel_sizes=[8, 4],
    strides=[2, 1],
    aggregator_class=SquashDims,
)
print(cnn)
```



```
MLP(
  (0): LazyLinear(in_features=0, out_features=32, bias=True)
  (1): ELU(alpha=1.0)
  (2): Linear(in_features=32, out_features=64, bias=True)
  (3): ELU(alpha=1.0)
  (4): Linear(in_features=64, out_features=4, bias=True)
)
ConvNet(
  (0): LazyConv2d(0, 32, kernel_size=(8, 8), stride=(2, 2))
  (1): ELU(alpha=1.0)
  (2): Conv2d(32, 64, kernel_size=(4, 4), stride=(1, 1))
  (3): ELU(alpha=1.0)
  (4): SquashDims()
)
```

Source [TorchRL documentaion](#)

The `tensorDictModule` provides a powerful and flexible data container that enables structured, batched storage and manipulation of tensors, serving as the backbone for data flow across components. It operates on `TensorDicts` by reading specific entries using `in_keys`, passing them as inputs to an internal computation (often a neural network), and writing the outputs back under keys specified in `out_keys`. This design allows

seamless composition: for example, a module with `in_keys=["obs"]` and `out_keys=["action"]` will take the "obs" tensor, compute the action, and insert it into "action". For models with multiple outputs (e.g., value and policy), `out_keys` can contain multiple entries like `["action", "log_prob"]`, enabling the module to return and store multiple computed results in a single pass.

```
data = TensorDict({"key1": torch.randn(10, 4)}, batch_size=[10])
module = nn.Linear(4, 5)
td_module = TensorDictModule(module, in_keys=["key1"], out_keys=["key2"])
td_module(data)
print('The input key is "key1" and the output key is "key2". The input tensor is passed to the m
print(data)
```

➡ The input key is "key1" and the output key is "key2". The input tensor is passed to the module.

```
TensorDict(
  fields={
    key1: Tensor(shape=torch.Size([10, 4]), device=cpu, dtype=torch.float32, is_shared=False),
    key2: Tensor(shape=torch.Size([10, 5]), device=cpu, dtype=torch.float32, is_shared=False),
    batch_size=torch.Size([10]),
    device=None,
    is_shared=False)
```

`TensorDictSequential` is a high-level container that chains multiple `TensorDictModule`s into a single, end-to-end pipeline. Each module in the sequence receives a `TensorDict`, reads its required inputs (`in_keys`), computes outputs, and writes the results back using `out_keys`, making it easy to build structured, modular policies or architectures. The modules are executed in order, and later modules can consume the outputs of earlier ones — for example, a backbone network producing "hidden" features that are then used by both an actor and a value head.

```
backbone_module = nn.Linear(5, 3)
backbone = TensorDictModule(
    backbone_module, in_keys=["observation"], out_keys=["hidden"]
)
actor_module = nn.Linear(3, 4)
actor = TensorDictModule(actor_module, in_keys=["hidden"], out_keys=["action"])
value_module = MLP(out_features=1, num_cells=[4, 5])
value = TensorDictModule(value_module, in_keys=["hidden", "action"], out_keys=["value"])

sequence = TensorDictSequential(backbone, actor, value)
print(sequence)

print(sequence.in_keys, sequence.out_keys)

data = TensorDict(
    {"observation": torch.randn(3, 5)},
    [3],
)
backbone(data)
actor(data)
value(data)

data = TensorDict(
    {"observation": torch.randn(3, 5)},
    [3],
)
sequence(data)
print(data)
```

➡ `TensorDictSequential`

```
module=ModuleList(
```

```

(0): TensorDictModule(
  module=Linear(in_features=5, out_features=3, bias=True),
  device=cpu,
  in_keys=['observation'],
  out_keys=['hidden'])
(1): TensorDictModule(
  module=Linear(in_features=3, out_features=4, bias=True),
  device=cpu,
  in_keys=['hidden'],
  out_keys=['action'])
(2): TensorDictModule(
  module=MLP(
    (0): LazyLinear(in_features=0, out_features=4, bias=True)
    (1): Tanh()
    (2): Linear(in_features=4, out_features=5, bias=True)
    (3): Tanh()
    (4): Linear(in_features=5, out_features=1, bias=True)
  ),
  device=cpu,
  in_keys=['hidden', 'action'],
  out_keys=['value'])
),
device=cpu,
in_keys=['observation'],
out_keys=['hidden', 'action', 'value'])
TensorDict(
  fields={
    action: Tensor(shape=torch.Size([3, 4]), device=cpu, dtype=torch.float32, is_shared=False),
    hidden: Tensor(shape=torch.Size([3, 3]), device=cpu, dtype=torch.float32, is_shared=False),
    observation: Tensor(shape=torch.Size([3, 5]), device=cpu, dtype=torch.float32, is_shared=False),
    value: Tensor(shape=torch.Size([3, 1]), device=cpu, dtype=torch.float32, is_shared=False),
  },
  batch_size=torch.Size([3]),
  device=None,
  is_shared=False)

```

Source: [TorchRL documentation](#)

Since `TensorDictModule`s typically wrap neural networks, their parameters need to be optimized during training. TorchRL provides a convenient utility called `from_module`, which allows you to extract all learnable parameters from nested or composed modules. This means you can easily pass the entire model—including all submodules—to an optimizer (e.g., Adam or SGD) without manually collecting parameters, ensuring that every learnable part of the pipeline is updated during training.

```

params = from_module(sequence)
print("extracted params", params)

```

```

➡ extracted params TensorDict(
  fields={
    module: TensorDict(
      fields={
        0: TensorDict(
          fields={
            module: TensorDict(
              fields={
                bias: Parameter(shape=torch.Size([3]), device=cpu, dtype=torch.float32, is_shared=False),
                weight: Parameter(shape=torch.Size([3, 5]), device=cpu, dtype=torch.float32, is_shared=False),
              },
              batch_size=torch.Size([3]),
              device=None,
              is_shared=False),
            batch_size=torch.Size([3]),
            device=None,
            is_shared=False),
          batch_size=torch.Size([3]),
          device=None,
          is_shared=False),
        1: TensorDict(

```

```

fields={
    module: TensorDict(
        fields={
            bias: Parameter(shape=torch.Size([4]), device=cpu, dtype=
            weight: Parameter(shape=torch.Size([4, 3]), device=cpu, d
            batch_size=torch.Size([]),
            device=None,
            is_shared=False)},
    batch_size=torch.Size([]),
    device=None,
    is_shared=False),
2: TensorDict(
    fields={
        module: TensorDict(
            fields={
                0: TensorDict(
                    fields={
                        bias: Parameter(shape=torch.Size([4]), device=cpu
                        weight: Parameter(shape=torch.Size([4, 7]), devic
                        batch_size=torch.Size([]),
                        device=None,
                        is_shared=False),
                2: TensorDict(
                    fields={
                        bias: Parameter(shape=torch.Size([5]), device=cpu
                        weight: Parameter(shape=torch.Size([5, 4]), devic
                        batch_size=torch.Size([]),
                        device=None,
                        is_shared=False),
                4: TensorDict(
                    fields={
                        bias: Parameter(shape=torch.Size([1]), device=cpu
                        weight: Parameter(shape=torch.Size([1, 5]), devic
                        batch_size=torch.Size([]),
                        device=None,
                        is_shared=False)},
            batch_size=torch.Size([]),
            device=None,
            is_shared=False)},
    . . . . .
    . . . . .
    . . . . .

```

✓ 9.4.1 Specialized Classes

TorchRL provides a collection of **specialized module wrappers** designed to streamline the use of neural networks in reinforcement learning by automating common behaviors related to input/output handling, safety constraints, and probabilistic action sampling.

These wrappers are typically built on top of `TensorDictModule` and come with **predefined conventions** for input and output keys, along with built-in validation logic that ensures their expected behavior during execution. Some examples:

- The `Actor` module wraps a base neural network and **automatically outputs an "action" key**, unless explicitly overridden. This simplifies policy definition, as it enforces a standard interface across RL components.
- The `SafeModule` ensures that the output of a network stays **within a specified range**, defined by a `TensorSpec` (e.g., `Bounded`). When `safe=True`, it projects the network output to be valid according to this spec, which is especially useful in environments with bounded action spaces.
- The `ProbabilisticTensorDictModule` enables **distribution-based modeling** by generating actions via sampling. It works in tandem with a preceding module that outputs distribution parameters like `"loc"` and `"scale"`, and then uses a specified distribution class (e.g., `TanhNormal`) to sample from it. The sampled result is placed in the designated `out_keys`, such as `"action"`.

- When combined in a `ProbabilisticTensorDictSequential`, these modules form an end-to-end pipeline where a neural network computes distribution parameters, and actions are sampled in a differentiable way, allowing seamless policy learning with stochasticity.

These specialized modules greatly reduce boilerplate code and ensure consistent behavior across agents, while maintaining full compatibility with TorchRL's `TensorDict`-based API.

```
actor_net = nn.Linear(5, 3)
actor = Actor(actor_net, in_keys=["obs"])
data = TensorDict({"obs": torch.randn(5)}, batch_size=[])
actor(data)
print("Actor output:", data)

spec = Bounded(low=-1.0, high=1.0, shape=(3,))
safe_net = nn.Linear(5, 3)
safe_module = SafeModule(safe_net, spec=spec, in_keys=["obs"], out_keys=["action"], safe=True)

data = TensorDict({"obs": torch.randn(5) * 100}, batch_size=[])
safe_action = safe_module(data)["action"]
print("SafeModule output (clipped):", safe_action)

param_net = nn.Sequential(nn.Linear(5, 4), NormalParamExtractor())
param_module = TensorDictModule(param_net, in_keys=["input"], out_keys=["loc", "scale"])

prob_module = ProbabilisticTensorDictModule(
    in_keys=["loc", "scale"],
    out_keys=["action"],
    distribution_class=TanhNormal,
    return_log_prob=False
)

prob_model = ProbabilisticTensorDictSequential(param_module, prob_module)

td = TensorDict({"input": torch.randn(3, 5)}, batch_size=[3])
prob_model(td)
print("Probabilistic output:", td["action"])

⇒ Actor output: TensorDict(
  fields={
    action: Tensor(shape=torch.Size([3]), device=cpu, dtype=torch.float32, is_shared=False),
    obs: Tensor(shape=torch.Size([5]), device=cpu, dtype=torch.float32, is_shared=False),
    batch_size: torch.Size([]),
    device: None,
    is_shared: False)
  SafeModule output (clipped): tensor([ 1., -1.,  1.], grad_fn=<ClampBackward0>)
  Probabilistic output: tensor([[ 0.2397, -0.4166],
    [ 0.2882,  0.4207],
    [ 0.0491, -0.8551]], grad_fn=<_SafeTanhNoEpsBackward>)
```

Randomness and sampling behavior can be controlled using the `set_exploration_type` context manager, which allows you to specify how actions are sampled (e.g., stochastic vs. deterministic) during interaction with environments or policies.

```
td = TensorDict({"input": torch.randn(3, 5)}, [3])

with set_exploration_type(ExplorationType.RANDOM):
    prob_model(td)
    print("random:", td["action"])

with set_exploration_type(ExplorationType.DETERMINISTIC):
```

```

prob_model(td)
print("mode:", td["action"])

random: tensor([[ -0.2115,  0.8018],
               [-0.4305, -0.9352],
               [-0.4585,  0.6259]], grad_fn=<_SafeTanhNoEpsBackward>)
mode: tensor([[ -0.5756, -0.7819],
              [-0.5916, -0.7996],
              [ 0.0282, -0.4533]], grad_fn=<_SafeTanhNoEpsBackward>)

```

Source: [TorchRL documentation](#)

✓ 9.5 Combining Environments and Modules

This example shows how to interact with an environment step-by-step using a custom actor and preallocate storage for the collected data with a `TensorDict`. The `step_mdp` utility is used to transition the "next" fields (e.g., "next_observation") into the standard "observation" keys for the next timestep, streamlining multi-step rollouts.

```

env = GymEnv("Pendulum-v1")

action_spec = env.action_spec
actor_module = nn.Linear(3, 1)
actor = SafeModule(
    actor_module, spec=action_spec, in_keys=["observation"], out_keys=["action"]
)
env.set_seed(0)

max_steps = 100
data = env.reset()
data_stack = TensorDict(batch_size=[max_steps])
for i in range(max_steps):
    actor(data)
    data_stack[i] = env.step(data)
    if data["done"].any():
        break
    data = step_mdp(data) # roughly equivalent to obs = next_obs

tensordicts_prealloc = data_stack.clone()
print("total steps:", i)
print(data_stack)

total steps: 99
TensorDict(
  fields={
    action: Tensor(shape=torch.Size([100, 1]), device=cpu, dtype=torch.float32, is_shared=False),
    done: Tensor(shape=torch.Size([100, 1]), device=cpu, dtype=torch.bool, is_shared=False),
    next: TensorDict(
      fields={
        done: Tensor(shape=torch.Size([100, 1]), device=cpu, dtype=torch.bool, is_shared=False),
        observation: Tensor(shape=torch.Size([100, 3]), device=cpu, dtype=torch.float32, is_shared=False),
        reward: Tensor(shape=torch.Size([100, 1]), device=cpu, dtype=torch.float32, is_shared=False),
        terminated: Tensor(shape=torch.Size([100, 1]), device=cpu, dtype=torch.bool, is_shared=False),
        truncated: Tensor(shape=torch.Size([100, 1]), device=cpu, dtype=torch.bool, is_shared=False),
        batch_size: torch.Size([100]),
        device: None,
        is_shared: False,
      },
      observation: Tensor(shape=torch.Size([100, 3]), device=cpu, dtype=torch.float32, is_shared=False),
      terminated: Tensor(shape=torch.Size([100, 1]), device=cpu, dtype=torch.bool, is_shared=False),
      truncated: Tensor(shape=torch.Size([100, 1]), device=cpu, dtype=torch.bool, is_shared=False),
      batch_size: torch.Size([100]),
      device: None,
    ),
  },
  batch_size=torch.Size([100]),
  device=None,
)

```

```
is_shared=False)
```

Source: [TorchRL documentation](#)

```
# equivalent
env.set_seed(0)
```

```
max_steps = 100
data = env.reset()
data_stack = []
for _ in range(max_steps):
    actor(data)
    data_stack.append(env.step(data))
    if data["done"].any():
        break
    data = step_mdp(data) # roughly equivalent to obs = next_obs
tensordicts_stack = torch.stack(data_stack, 0)
print("total steps:", i)
print(tensordicts_stack)
```

```
➡ total steps: 99
TensorDict(
  fields={
    action: Tensor(shape=torch.Size([100, 1]), device=cpu, dtype=torch.float32, is_shared=False),
    done: Tensor(shape=torch.Size([100, 1]), device=cpu, dtype=torch.bool, is_shared=False),
    next: TensorDict(
      fields={
        done: Tensor(shape=torch.Size([100, 1]), device=cpu, dtype=torch.bool, is_shared=False),
        observation: Tensor(shape=torch.Size([100, 3]), device=cpu, dtype=torch.float32, is_shared=False),
        reward: Tensor(shape=torch.Size([100, 1]), device=cpu, dtype=torch.float32, is_shared=False),
        terminated: Tensor(shape=torch.Size([100, 1]), device=cpu, dtype=torch.bool, is_shared=False),
        truncated: Tensor(shape=torch.Size([100, 1]), device=cpu, dtype=torch.bool, is_shared=False),
        batch_size: torch.Size([100]),
        device=None,
        is_shared=False),
    observation: Tensor(shape=torch.Size([100, 3]), device=cpu, dtype=torch.float32, is_shared=False),
    terminated: Tensor(shape=torch.Size([100, 1]), device=cpu, dtype=torch.bool, is_shared=False),
    truncated: Tensor(shape=torch.Size([100, 1]), device=cpu, dtype=torch.bool, is_shared=False),
    batch_size: torch.Size([100]),
    device=None,
    is_shared=False)
```

Source: [TorchRL documentation](#)

A rollout is a procedure that collects a sequence of interactions between a policy and an environment, and in TorchRL this can be automated using the `rollout` function, which handles resetting, stepping, and stacking transitions into a single `TensorDict`. This method essentially does the same thing as the two cells above, but in a single call.

```
tensordict_rollout = env.rollout(policy=actor, max_steps=max_steps)
print("Tensordict rollout:", tensordict_rollout)
```

```
➡ Tensordict rollout: TensorDict(
  fields={
    action: Tensor(shape=torch.Size([100, 1]), device=cpu, dtype=torch.float32, is_shared=False),
    done: Tensor(shape=torch.Size([100, 1]), device=cpu, dtype=torch.bool, is_shared=False),
    next: TensorDict(
      fields={
```

```

done: Tensor(shape=torch.Size([100, 1]), device=cpu, dtype=torch.bool, is_shar
observation: Tensor(shape=torch.Size([100, 3]), device=cpu, dtype=torch.float
reward: Tensor(shape=torch.Size([100, 1]), device=cpu, dtype=torch.float32, :
terminated: Tensor(shape=torch.Size([100, 1]), device=cpu, dtype=torch.bool, :
truncated: Tensor(shape=torch.Size([100, 1]), device=cpu, dtype=torch.bool, :
batch_size=torch.Size([100]),
device=None,
is_shared=False),
observation: Tensor(shape=torch.Size([100, 3]), device=cpu, dtype=torch.float32, is_
terminated: Tensor(shape=torch.Size([100, 1]), device=cpu, dtype=torch.bool, is_shar
truncated: Tensor(shape=torch.Size([100, 1]), device=cpu, dtype=torch.bool, is_shar
batch_size=torch.Size([100]),
device=None,
is_shared=False)

```

✓ 9.6 Objectives

One of the core components of any reinforcement learning algorithm is the **objective function**, or **loss**, which guides the optimization of the model's parameters. Many commonly used losses are already implemented and readily available as modular components, so there's no need to reimplement them from scratch. These loss modules can be directly integrated into training pipelines like any other PyTorch module. Examples include `DDPGLoss`, `PPOLoss`, `SACLoss`, and others, each encapsulating the full logic of their respective algorithms—handling value estimation, policy updates, entropy regularization, and more. In the following sections, we will look at a practical example using `DDPGLoss`.

```

from torchrl.objectives import DDPGLoss

actor_module = nn.Linear(3, 1)
actor = TensorDictModule(actor_module, in_keys=["observation"], out_keys=["action"])

class ConcatModule(nn.Linear):
    def forward(self, obs, action):
        return super().forward(torch.cat([obs, action], -1))

value_module = ConcatModule(4, 1)
value = TensorDictModule(
    value_module, in_keys=["observation", "action"], out_keys=["state_action_value"]
)

loss_fn = DDPGLoss(actor, value)
loss_fn.make_value_estimator(loss_fn.default_value_estimator, gamma=0.99)

data = TensorDict(
    {
        "observation": torch.randn(10, 3),
        "next": {
            "observation": torch.randn(10, 3),
            "reward": torch.randn(10, 1),
            "done": torch.zeros(10, 1, dtype=torch.bool),
        },
        "action": torch.randn(10, 1),
    },
    batch_size=[10],
    device="cpu",
)
loss_td = loss_fn(data)

print(loss_td)
print(data)

```

Source: [TorchRL documentation](#)

▼ References

Some code snippets are from [Vincent Moens - TorchRL](#) and are appropriately marked after the corresponding cells.

Licensed under [CC BY-NC-ND 4.0](#). © Zoltán Barta, 2025.

✓ 10. Practice - Centralized Training with Decentralized Execution

👤 Zoltán Barta, PhD student, Department of Artificial Intelligence

🕒 90 min read

📅 January 22, 2025

📚 Collective Intelligence



This practice notebook introduces the Centralized Training with Decentralized Execution (CTDE) paradigm, a foundational framework in multi-agent reinforcement learning. CTDE utilizes global information and collaborative learning during training, while enforcing independent, locally observable decision-making at deployment. This approach bridges the gap between theoretical advancements in MARL and practical real-world constraints, where agents often lack access to centralized information at execution time.

The notebook covers both core concepts and hands-on algorithms such as MAPPO, illustrating the architecture, workflow, and training procedures underlying CTDE. By completing this module, students will be equipped to understand, implement, and evaluate multi-agent learning solutions using the CTDE framework.

[CTDE Paper](#)

Table of Contents

- **10.1 Overview of CTDE**
 - 10.1.1 Centralized Training
 - 10.1.2 Decentralized Execution
 - 10.1.3 Advantages of CTDE
 - 10.1.4 Common CTDE Algorithms
 - 10.1.5 Architecture
- **10.2 Necessary Imports**
 - 10.2.1 Hyperparameters
 - 10.2.2 Environment
- **10.3 Multi-Agent Proximal Policy Optimization (MAPPO)**
 - 10.3.1 How It Works
 - 10.3.2 Algorithm Steps
 - 10.3.3 Implementation Tips
 - 10.3.4 Actor (policy) Network
 - 10.3.5 Critic Network
 - 10.3.6 Data Collector
 - 10.3.7 Loss Function
- **10.4 Training Loop**
 - 10.4.1 Results

- **References**

✓ 10.1 Overview of CTDE

In many multi-agent systems, agents have limited perception and cannot access the full global state or the actions of other agents during execution. However, during training, such information is often available.

CTDE allows the use of:

- Centralized information (e.g., full observations, joint actions, global rewards) to stabilize and accelerate training
- Decentralized policies that depend only on local observations for execution, allowing real-world applicability

This paradigm is particularly useful in environments where:

- Communication is restricted or costly
- Agents must act independently
- The environment is partially observable

10.1.1 Centralized Training

During training:

- A centralized critic (or multiple critics) can access the observations and actions of all agents.
- Training algorithms can leverage full state information to estimate joint value functions or shared objectives.
- Agents may communicate or use additional sensors to improve learning.

10.1.2 Decentralized Execution

During execution:

- Each agent acts based solely on its own local observation.
- No access to other agents' observations, actions, or a centralized critic is allowed.
- Policies must be reactive and efficient under partial observability.

This separation ensures that agents are practical to deploy in distributed systems or real-world environments.

10.1.3 Advantages of CTDE

- Improved sample efficiency through centralized critics
- Better coordination among agents during training
- Scalability in deployment due to decentralized policies
- Compatible with both cooperative and competitive multi-agent settings

10.1.4 Common CTDE Algorithms

Several popular multi-agent algorithms are built around the CTDE principle:

- **MADDPG (Multi-Agent DDPG)**
Uses a centralized critic with decentralized actors. The critic takes all agents' actions and observations as input.

- **COMA (Counterfactual Multi-Agent Policy Gradients)**

Employs a centralized critic to compute counterfactual advantages for each agent in cooperative tasks.

- **QMIX**

Trains agents with individual Q-functions combined into a global Q-value via a mixing network, conditioned on the global state.

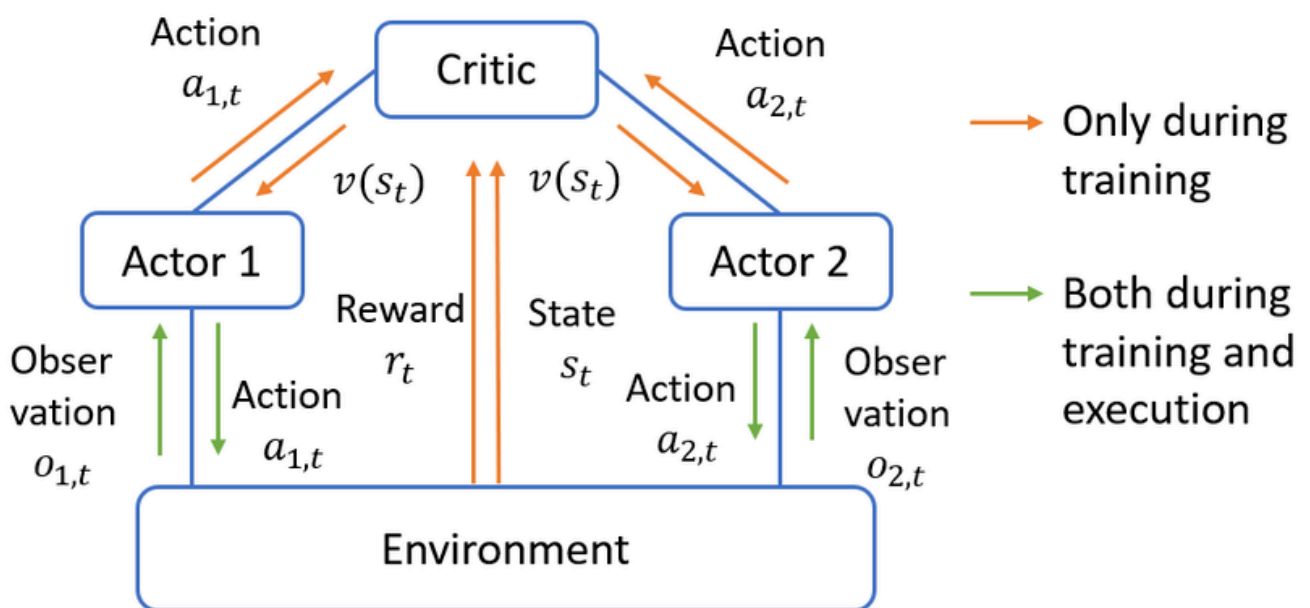
10.1.5 Architecture

A typical CTDE architecture includes:

- **Local policy networks** for each agent
- A **shared or centralized critic** used only during training
- Optional **shared replay buffers** for storing transitions

Each training step may include:

1. Agents take actions based on local observations.
2. The environment returns new states and rewards.
3. Centralized critic(s) use global data to compute value estimates or gradients.
4. Policy networks are updated to maximize performance using local inputs only.



✓ 10.2 Necessary Imports

```
# Torch
import torch

# Tensordict modules
from tensordict.nn import set_composite_lp_aggregate, TensorDictModule
from tensordict.nn.distributions import NormalParamExtractor
from torch import multiprocessing

# Data collection
from torchrl.collectors import SyncDataCollector
from torchrl.data.replay_buffers import ReplayBuffer
from torchrl.data.replay_buffers.samplers import SamplerWithoutReplacement
from torchrl.data.replay_buffers.storages import LazyTensorStorage
```

```

# Env
from torchrl.envs import RewardSum, TransformedEnv
from torchrl.envs.libs.vmas import VmasEnv
from torchrl.envs.utils import check_env_specs
from torchrl.modules import MultiAgentMLP, ProbabilisticActor, TanhNormal
# Loss
from torchrl.objectives import ClipPPOLoss, ValueEstimators

# Utils
torch.manual_seed(0)
from matplotlib import pyplot as plt
from tqdm import tqdm

```

✓ 10.2.1 Hyperparameters

- **frames_per_batch**: Defines how many environment steps are collected before each training phase.
- **n_iters**: Indicates how many full cycles of data collection and training will be performed.
- **total_frames**: The cumulative number of environment interactions across all iterations.
- **num_epochs**: Number of times the collected data is reused for learning in a single iteration.
- **minibatch_size**: Specifies how many samples are processed at once during each gradient update.
- **lr**: Learning rate controlling how much model parameters are adjusted per update step.
- **max_grad_norm**: Sets a threshold to clip gradients and stabilize learning by preventing excessively large updates.
- **clip_epsilon**: Bounds policy updates to prevent overly aggressive shifts, maintaining stable learning.
- **gamma**: Controls the trade-off between short-term and long-term rewards via discounting.
- **lmbda**: Regulates the balance between bias and variance in advantage estimation using GAE.
- **entropy_eps**: Adds randomness to the policy by rewarding higher entropy, promoting better exploration.

```

device = (
    torch.device(0)
    if torch.cuda.is_available()
    else torch.device("cpu")
)

# Sampling
frames_per_batch = 6_000
n_iters = 20
total_frames = frames_per_batch * n_iters

# Training
num_epochs = 30
minibatch_size = 400
lr = 3e-4
max_grad_norm = 1.0

# PPO
clip_epsilon = 0.2
gamma = 0.99
lmbda = 0.9
entropy_eps = 1e-4

# disable log-prob aggregation
set_composite_lp_aggregate(False).set()

```

✓ 10.2.2 Environment

In this tutorial, we use the **VMAS** simulator, a lightweight multi-agent physics engine where each agent operates with its own set of observations, rewards, actions, and auxiliary information. Although agents act independently, they share a common "done" signal, which indicates episode termination. VMAS is built for **vectorized simulation**, meaning that it processes multiple environments in parallel using PyTorch tensors, where the leading dimension corresponds to the number of parallel simulations.

The environment we focus on is the **Navigation** task. In this scenario, several agents are randomly placed in a 2D space and must navigate toward their respective goals, which are also randomly positioned. To avoid collisions, agents rely on LIDAR-based perception (represented by radial sensor readings). Each agent produces 2D continuous actions that correspond to force vectors controlling their movement. The reward function combines three components: a penalty for collisions, a shaping reward based on progress toward the goal, and a shared reward when all agents successfully reach their targets. Observations include each agent's position, velocity, goal direction, and LIDAR readings.

```
max_steps = 300 # Episode steps before done
num_vmas_envs = (
    frames_per_batch // max_steps
) # Number of vectorized envs. frames_per_batch should be divisible by this number
print(num_vmas_envs)
scenario_name = "navigation"

n_agents = 3

env = VmasEnv(
    scenario=scenario_name,
    num_envs=num_vmas_envs,
    continuous_actions=True, # VMAS supports both continuous and discrete actions
    max_steps=max_steps,
    device=device,
    # Scenario kwargs
    n_agents=n_agents, # These are custom kwargs that change for each VMAS scenario, see the VM
)
env = VmasEnv("navigation", num_envs=1, continuous_actions=True, n_agents = 10, agent_radius= 0.05,
```

➡ 20

In TorchRL, you can easily inspect the environment's various specifications—such as actions, rewards, terminations, and observations—which is extremely helpful for debugging and understanding the data structure your model will interact with.

```
print("action_spec:", env.full_action_spec)
print("reward_spec:", env.full_reward_spec)
print("done_spec:", env.full_done_spec)
print("observation_spec:", env.observation_spec)
```

➡ action_spec: Composite(

```
    agents: Composite(
        action: BoundedContinuous(
            shape=torch.Size([1, 10, 2]),
            space=ContinuousBox(
                low=Tensor(shape=torch.Size([1, 10, 2]), device=cpu, dtype=torch.float32,
                high=Tensor(shape=torch.Size([1, 10, 2]), device=cpu, dtype=torch.float32,
                device=cpu,
                dtype=torch.float32,
                domain=continuous),
            device=None,
            shape=torch.Size([1, 10])),
        device=None,
        shape=torch.Size([1]))
    reward_spec: Composite(
```

```

agents: Composite(
    reward: UnboundedContinuous(
        shape=torch.Size([1, 10, 1]),
        space=ContinuousBox(
            low=Tensor(shape=torch.Size([1, 10, 1]), device=cpu, dtype=torch.float32,
            high=Tensor(shape=torch.Size([1, 10, 1]), device=cpu, dtype=torch.float32,
            device=cpu,
            dtype=torch.float32,
            domain=continuous),
        device=None,
        shape=torch.Size([1, 10])),
    device=None,
    shape=torch.Size([1]))
done_spec: Composite(
    done: Categorical(
        shape=torch.Size([1, 1]),
        space=CategoricalBox(n=2),
        device=cpu,
        dtype=torch.bool,
        domain=discrete),
    terminated: Categorical(
        shape=torch.Size([1, 1]),
        space=CategoricalBox(n=2),
        device=cpu,
        dtype=torch.bool,
        domain=discrete),
    device=None,
    shape=torch.Size([1]))
observation_spec: Composite(
    agents: Composite(
        observation: UnboundedContinuous(
            shape=torch.Size([1, 10, 18]),
            space=ContinuousBox(
                low=Tensor(shape=torch.Size([1, 10, 18]), device=cpu, dtype=torch.float32,
                high=Tensor(shape=torch.Size([1, 10, 18]), device=cpu, dtype=torch.float32,
                device=cpu,
                dtype=torch.float32,
                domain=continuous),
            info: Composite(
                pos_rew: UnboundedContinuous(
                    shape=torch.Size([1, 10, 1]),

```

Using the commands shown above, we can inspect the *specifications* of an environment—such as the action, reward, done, and observation specs—which reveal the expected structure and shape of the data returned during interactions. A key detail in multi-agent settings is that most specs (like observations, rewards, and actions) have a leading shape of `(num_vmas_envs, n_agents)`, indicating that each agent in each parallel environment has its own individual values. The "done" spec, however, typically has a shape of `(num_vmas_envs,)`, showing that episode termination is shared across all agents within a single environment instance.

TorchRL distinguishes between **shared** and **per-agent** specs by organizing them accordingly. Specs that are agent-specific—those with the extra agent dimension—are grouped under an "agents" key. For example, if the action or reward spec is agent-dependent, any related data in the `TensorDict` will be nested under "agents", enabling a clean and structured representation.

Moreover, TorchRL provides utility properties like `env.input_spec.keys()` or `env.output_spec.keys()` to quickly retrieve the relevant keys associated with each type of value. This allows us to understand not just the shapes, but also the **location** of the data in the `TensorDict` hierarchy—greatly simplifying integration with other TorchRL components during training and evaluation.

```

print("action_keys:", env.action_keys)
print("reward_keys:", env.reward_keys)
print("done_keys:", env.done_keys)

```

```

➡ action_keys: [('agents', 'action')]
   reward_keys: [('agents', 'reward')]
   done_keys: ['done', 'terminated']

```

To facilitate goal visualization during evaluation or debugging, we apply a `RewardSum` transform to the environment. This transform adds goal-related reward shaping or tracking features, which can be useful for plotting or monitoring purposes. However, it's important to note that the agents themselves do not use this transform—it's purely for external observation and does not influence the policy learning process.

```

env = TransformedEnv(
    env,
    RewardSum(in_keys=[env.reward_key], out_keys=["agents", "episode_reward"])),
)

```

We can check if every module is compatible with the transforms.

```

check_env_specs(env)

```

```

➡ 2025-05-16 07:38:24,048 [torchrl][INFO] check_env_specs succeeded!

```

```

n_rollout_steps = 5
rollout = env.rollout(n_rollout_steps)
print("Rollout of three steps:", rollout)
print("Shape of the rollout TensorDict:", rollout.batch_size)

```

```

➡ Rollout of three steps: TensorDict(
  fields={
    agents: TensorDict(
      fields={
        action: Tensor(shape=torch.Size([1, 2, 10, 2]), device=cpu, dtype=torch.float32, is_shared=False),
        episode_reward: Tensor(shape=torch.Size([1, 2, 10, 1]), device=cpu, dtype=torch.float32, is_shared=False),
        info: TensorDict(
          fields={
            agent_collisions: Tensor(shape=torch.Size([1, 2, 10, 1]), device=cpu, dtype=torch.float32, is_shared=False),
            final_rew: Tensor(shape=torch.Size([1, 2, 10, 1]), device=cpu, dtype=torch.float32, is_shared=False),
            pos_rew: Tensor(shape=torch.Size([1, 2, 10, 1]), device=cpu, dtype=torch.float32, is_shared=False),
            batch_size: torch.Size([1, 2, 10]),
            device: None,
            is_shared: False),
          observation: Tensor(shape=torch.Size([1, 2, 10, 18]), device=cpu, dtype=torch.float32, is_shared=False),
          batch_size: torch.Size([1, 2, 10]),
          device: None,
          is_shared: False),
        done: Tensor(shape=torch.Size([1, 2, 1]), device=cpu, dtype=torch.bool, is_shared=False),
        next: TensorDict(
          fields={
            agents: TensorDict(
              fields={
                episode_reward: Tensor(shape=torch.Size([1, 2, 10, 1]), device=cpu, dtype=torch.float32, is_shared=False),
                info: TensorDict(
                  fields={
                    agent_collisions: Tensor(shape=torch.Size([1, 2, 10, 1]), device=cpu, dtype=torch.float32, is_shared=False),
                    final_rew: Tensor(shape=torch.Size([1, 2, 10, 1]), device=cpu, dtype=torch.float32, is_shared=False),
                    pos_rew: Tensor(shape=torch.Size([1, 2, 10, 1]), device=cpu, dtype=torch.float32, is_shared=False),
                    batch_size: torch.Size([1, 2, 10]),
                    device: None,
                    is_shared: False),
                  observation: Tensor(shape=torch.Size([1, 2, 10, 18]), device=cpu, dtype=torch.float32, is_shared=False),
                  reward: Tensor(shape=torch.Size([1, 2, 10, 1]), device=cpu, dtype=torch.float32, is_shared=False),
                  batch_size: torch.Size([1, 2, 10]),
                  device: None,
                  is_shared: False),
                done: Tensor(shape=torch.Size([1, 2, 1]), device=cpu, dtype=torch.bool, is_shared=False),

```

```

        terminated: Tensor(shape=torch.Size([1, 2, 1]), device=cpu, dtype=torch.bool,
        batch_size=torch.Size([1, 2]),
        device=None,
        is_shared=False),
        terminated: Tensor(shape=torch.Size([1, 2, 1]), device=cpu, dtype=torch.bool, is_shar
        batch_size=torch.Size([1, 2]),
        device=None,
        is_shared=False)
Shape of the rollout TensorDict: torch.Size([1, 2])

```

A **rollout** generates a sequence of interactions between agents and the environment, returning a `TensorDict` where all relevant data (e.g., observations, actions, rewards) are stored in a batched, time-ordered structure.

A key part of this structure is the "next" key, which holds the results of the *next* timestep for each transition. For example, during a 3-step rollout, at each step t , the "next" entry contains what the state will be at $t+1$. When the rollout proceeds to the next step, the "next" block is **flattened back** into the main structure: its values (like "observation" or "reward") overwrite or extend the top-level entries, allowing the environment state to progress naturally from one step to the next.

This mechanism makes it easy to chain transitions together while keeping each timestep's inputs and outputs clearly separated. In multi-agent settings, nested structures like "agents" and subfields such as "info" or "episode_reward" are preserved, ensuring that per-agent data is maintained consistently across the entire rollout.

```
(rollout[:,1]['next']['agents']['observation'] == rollout[:,2]['agents']['observation']).all()
```

✓ 10.3 Multi-Agent Proximal Policy Optimization (MAPPO)

Multi-Agent Proximal Policy Optimization (MAPPO) ([Paper](#)) extends the PPO algorithm to multi-agent environments by supporting multiple agents that may share or maintain separate policies. It maintains the core benefits of PPO—stability, sample efficiency, and simplicity—while handling the additional complexity of multi-agent interactions. MAPPO is particularly effective when agents are trained with parameter sharing, but it also supports decentralized execution and centralized training.

10.3.1 How It Works

1. Policy Network:

- Each agent, or all agents collectively (if using parameter sharing), is equipped with a neural network $\pi_{\theta}(a_i | o_i)$ that maps local observations o_i to a distribution over actions a_i .

2. Trajectory Collection:

- The environment is rolled out in parallel across multiple agents and multiple environments, collecting observations, actions, rewards, and log-probabilities for each agent over time.

3. Advantage Estimation:

- Advantages A_t^i are computed individually for each agent using Generalized Advantage Estimation (GAE), balancing bias and variance while respecting the temporal nature of the data.

4. Clipped Objective:

- The same clipped surrogate loss from PPO is applied per agent to restrict large updates and stabilize learning: $L_i^{CLIP}(\theta) = \mathbb{E}_t \left[\min \left(r_t^i(\theta) A_t^i, \text{clip}(r_t^i(\theta), 1 - \epsilon, 1 + \epsilon) A_t^i \right) \right]$ where $r_t^i(\theta) = \frac{\pi_\theta(a_t^i | o_t^i)}{\pi_{\theta_{\text{old}}}(a_t^i | o_t^i)}$ is the ratio of current to old policy probabilities for agent i .

5. Optimization:

- The loss is aggregated across agents (if sharing parameters), and the policy is optimized using gradient ascent. The value function is trained per agent or shared, typically with mean squared error loss on the predicted returns.

10.3.2 Algorithm Steps

1. Initialize:

- Policy network(s) π_θ and value network(s) V_ϕ
- Set hyperparameters: learning rate, discount factor γ , clipping parameter ϵ , number of epochs, batch size, and agent count

2. For each iteration:

- Run multi-agent rollouts to collect trajectories across environments
- Compute agent-wise rewards-to-go and advantage estimates A_t^i
- Compute probability ratios for each agent and optimize the clipped loss over several epochs

3. Update Value Function(s):

- Train value network(s) using regression on returns, optionally with clipping for stability

4. Repeat:

- Continue until convergence or the desired number of training iterations is reached

10.3.3 Implementation Tips

• Parameter Sharing:

- Use shared policy networks across agents when behaviors can be similar—this accelerates learning and reduces memory usage.

• Advantage Normalization:

- Normalize each agent's advantages independently to improve training stability.

• Centralized Critic:

- Optionally use centralized value functions that receive the observations or actions of all agents to provide better value estimates.

• Agent-Aware Sampling:

- Ensure batch sampling respects agent dimensions (e.g., using "agents" key in `TensorDict`) for correct policy updates.

✓ 10.3.4 Actor (policy) Network

Here, we define a shared policy network for all agents using `MultiAgentMLP`, a TorchRL module tailored for multi-agent settings. The network maps each agent's local observation to the parameters (mean and standard deviation) of a Tanh-Normal distribution over actions, with `NormalParamExtractor` handling the split. By setting

share_params=True, all agents will use the **same** policy, enabling parameter sharing and more sample-efficient training in cooperative settings.

```
share_parameters_policy = True

policy_net = torch.nn.Sequential(
    MultiAgentMLP(
        n_agent_inputs=env.observation_spec["agents", "observation"].shape[
            -1
        ], # n_obs_per_agent
        n_agent_outputs=env.action_spec.shape[-1] * 2, # 2 * n_actions_per_agents
        n_agents=env.n_agents,
        centralised=False,
        share_params=share_parameters_policy,
        device=device,
        depth=2,
        num_cells=64,
        activation_class=torch.nn.Tanh,
    ),
    NormalParamExtractor(),
)
```

Next, we encapsulate the policy network in a `TensorDictModule`, which takes care of reading agent-specific observations from the input, passing them through the network, and writing the resulting distribution parameters (`loc`, `scale`) back into the correct locations—preserving the "agents" structure that holds per-agent data.

```
policy_module = TensorDictModule(
    policy_net,
    in_keys=[("agents", "observation")],
    out_keys=[("agents", "loc"), ("agents", "scale")],
)
```

In the next step, we build the final policy by using a `ProbabilisticActor`, which transforms the distribution parameters into actual actions sampled from a `TanhNormal` distribution. This module handles action clipping based on the environment's bounds and also computes the log-probabilities of the sampled actions, which are essential for training with PPO. It seamlessly integrates with the rest of the pipeline by reading from the "agents" keys and writing the sampled actions to the correct location in the `TensorDict`.

```
policy = ProbabilisticActor(
    module=policy_module,
    spec=env.action_spec_unbatched,
    in_keys=[("agents", "loc"), ("agents", "scale")],
    out_keys=[env.action_key],
    distribution_class=TanhNormal,
    distribution_kwargs={
        "low": env.full_action_spec_unbatched[env.action_key].space.low,
        "high": env.full_action_spec_unbatched[env.action_key].space.high,
    },
    return_log_prob=True,
)
```

✓ 10.3.5 Critic Network

To estimate state values for each agent, we define a critic network that reads observations and returns value predictions. The structure of this network depends on whether we're using **MAPPO** or **IPPO**, and whether we choose to **share parameters** across agents.

In this setup, we configure the critic with centralized input (`centralised=True`), meaning it can access all agent observations simultaneously. This is appropriate for MAPPO, where centralized training is allowed. The network outputs one value per agent, resulting in a tensor of shape (`..., n_agents, 1`).

By setting `share_params=True`, the same critic is used for all agents—beneficial in cooperative tasks like this, where agents share the same reward function. The critic is wrapped in a `TensorDictModule` to connect it to the RL pipeline: it takes agent observations as input and writes value predictions into the `"state_value"` key under `"agents"`.

```
share_parameters_critic = True
mappo = True # IPP0 if False

critic_net = MultiAgentMLP(
    n_agent_inputs=env.observation_spec["agents", "observation"].shape[-1],
    n_agent_outputs=1, # 1 value per agent
    n_agents=env.n_agents,
    centralised=True,
    share_params=share_parameters_critic,
    device=device,
    depth=2,
    num_cells=256,
    activation_class=torch.nn.Tanh,
)

critic = TensorDictModule(
    module=critic_net,
    in_keys=[("agents", "observation")],
    out_keys=[("agents", "state_value")],
)
```

Let's see how each module modifies the environment:

```
print("Running policy:", policy(env.reset()))
print("Running value:", critic(env.reset()))
```

```
➡ Running policy: TensorDict(
  fields={
    agents: TensorDict(
      fields={
        action: Tensor(shape=torch.Size([1, 10, 2]), device=cpu, dtype=torch.float32, is_shared=False),
        action_log_prob: Tensor(shape=torch.Size([1, 10]), device=cpu, dtype=torch.float32, is_shared=False),
        episode_reward: Tensor(shape=torch.Size([1, 10, 1]), device=cpu, dtype=torch.float32, is_shared=False),
        info: TensorDict(
          fields={
            agent_collisions: Tensor(shape=torch.Size([1, 10, 1]), device=cpu, dtype=torch.float32, is_shared=False),
            final_rew: Tensor(shape=torch.Size([1, 10, 1]), device=cpu, dtype=torch.float32, is_shared=False),
            pos_rew: Tensor(shape=torch.Size([1, 10, 1]), device=cpu, dtype=torch.float32, is_shared=False),
            batch_size: torch.Size([1, 10]),
            device: None,
            is_shared: False),
            loc: Tensor(shape=torch.Size([1, 10, 2]), device=cpu, dtype=torch.float32, is_shared=False),
            observation: Tensor(shape=torch.Size([1, 10, 18]), device=cpu, dtype=torch.float32, is_shared=False),
            scale: Tensor(shape=torch.Size([1, 10, 2]), device=cpu, dtype=torch.float32, is_shared=False),
            batch_size: torch.Size([1, 10]),
            device: None,
            is_shared: False),
            done: Tensor(shape=torch.Size([1, 1]), device=cpu, dtype=torch.bool, is_shared=False),
            terminated: Tensor(shape=torch.Size([1, 1]), device=cpu, dtype=torch.bool, is_shared=False),
            batch_size: torch.Size([1]),
            device: None,
            is_shared: False)
      )
    )
Running value: TensorDict(
  fields={
```

```

agents: TensorDict(
  fields={
    episode_reward: Tensor(shape=torch.Size([1, 10, 1]), device=cpu, dtype=torch.float32),
    info: TensorDict(
      fields={
        agent_collisions: Tensor(shape=torch.Size([1, 10, 1]), device=cpu, dtype=torch.float32),
        final_reward: Tensor(shape=torch.Size([1, 10, 1]), device=cpu, dtype=torch.float32),
        pos_reward: Tensor(shape=torch.Size([1, 10, 1]), device=cpu, dtype=torch.float32),
        batch_size: torch.Size([1, 10]),
        device=None,
        is_shared=False),
    observation: Tensor(shape=torch.Size([1, 10, 18]), device=cpu, dtype=torch.float32),
    state_value: Tensor(shape=torch.Size([1, 10, 1]), device=cpu, dtype=torch.float32),
    batch_size: torch.Size([1, 10]),
    device=None,
    is_shared=False),
  done: Tensor(shape=torch.Size([1, 1]), device=cpu, dtype=torch.bool, is_shared=False),
  terminated: Tensor(shape=torch.Size([1, 1]), device=cpu, dtype=torch.bool, is_shared=False),
  batch_size: torch.Size([1]),
  device=None,
  is_shared=False)

```

✓ 10.3.6 Data Collector

A data collector automates the process of gathering experience by repeatedly interacting with the environment using a policy. It handles resetting environments, computing actions from observations, performing environment steps, and managing episode boundaries (like `done` signals). In this example, we use `SyncDataCollector`, a synchronous collector that ensures a fixed number of environment steps are gathered per batch. It runs in a loop, automatically resetting completed episodes and continuing collection until the specified number of `frames_per_batch` or `total_frames` is reached, delivering data in a format compatible with training pipelines.

```

collector = SyncDataCollector(
    env,
    policy,
    device=device,
    storing_device=device,
    frames_per_batch=frames_per_batch,
    total_frames=total_frames,
)

replay_buffer = ReplayBuffer(
    storage=LazyTensorStorage(
        frames_per_batch, device=device
    ),
    sampler=SamplerWithoutReplacement(),
    batch_size=minibatch_size,
)

```

✓ 10.3.7 Loss Function

The `ClipPPOLoss` module encapsulates the full PPO loss computation, including policy loss, value loss, and entropy regularization, hiding the complex control flow and mathematical details behind a clean interface. To use it, we pass in the policy (`actor_network`) and value (`critic_network`) modules, as well as key hyperparameters like the clipping factor (`clip_epsilon`) and the entropy coefficient (`entropy_eps`). It's important to configure `normalize_advantage=False` when working with per-agent data, to avoid mixing statistics across agents.

The `set_keys` method specifies where in the `TensorDict` the loss function should find the required entries such as rewards, actions, values, and termination flags. These keys follow the multi-agent "agents" structure to align with the environment's output.

Next, we attach a value estimator using `make_value_estimator`, which configures the module to compute **Generalized Advantage Estimation (GAE)**. GAE balances bias and variance in advantage computation and populates the `TensorDict` with "advantage" and "value_target" entries, which are then used by `ClipPPOLoss` to compute gradients during training.

Finally, we initialize an optimizer (here, Adam) with the parameters of the loss module, so both actor and critic networks are trained together.

```
loss_module = ClipPPOLoss(
    actor_network=policy,
    critic_network=critic,
    clip_epsilon=clip_epsilon,
    entropy_coef=entropy_eps,
    normalize_advantage=False,
)
loss_module.set_keys(
    reward=env.reward_key,
    action=env.action_key,
    value=("agents", "state_value"),

    done=("agents", "done"),
    terminated=("agents", "terminated"),
)

loss_module.make_value_estimator(
    ValueEstimators.GAE, gamma=gamma, lmbda=lmbda
)
GAE = loss_module.value_estimator

optim = torch.optim.Adam(loss_module.parameters(), lr)
```

✓ 10.4 Training Loop

This training loop orchestrates the full PPO optimization process across multiple iterations. Here's a breakdown of its structure and purpose:

1. Data Collection:

- The `collector` yields batches of experience (`tensorDict_data`) from the environment using the current policy.
- "done" and "terminated" signals are expanded to match the reward tensor's shape. This step is necessary for the **value estimator** (GAE) to align termination information with per-agent reward entries.

2. Advantage Computation (GAE):

- With gradients turned off (`torch.no_grad()`), **Generalized Advantage Estimation (GAE)** is computed and injected into the data. This augments the `TensorDict` with "advantage" and "value_target" entries needed for loss calculation.

3. Storing Experience:

- The batch is flattened and added to the `replay_buffer`, allowing it to be sampled for mini-batch training.

4. Policy and Value Network Update:

- For a fixed number of epochs (`num_epochs`), several mini-batches are drawn from the replay buffer.
- For each mini-batch:
 - The PPO loss (`loss_module`) is computed, aggregating policy, value, and entropy losses.
 - Backpropagation is performed, and optionally, gradients are clipped to stabilize learning.
 - The optimizer updates model parameters and resets gradients for the next step.

5. Syncing Policy Weights:

- After optimization, the collector's internal policy is updated with the newly trained weights to ensure the next rollout uses the latest model.

6. Logging:

- The mean episode reward is calculated using the rewards of completed episodes.
- This metric is added to a list for later visualization and displayed in the progress bar to provide live feedback on performance.

```
def collect_data_and_compute_gae(collector, env, loss_module, replay_buffer):
    tensordict_data = collector.next()

    # Expand done and terminated to match reward shape
    tensordict_data.set(
        ("next", "agents", "done"),
        tensordict_data.get(("next", "done"))
        .unsqueeze(-1)
        .expand(tensordict_data.get_item_shape(("next", env.reward_key))),
    )
    tensordict_data.set(
        ("next", "agents", "terminated"),
        tensordict_data.get(("next", "terminated"))
        .unsqueeze(-1)
        .expand(tensordict_data.get_item_shape(("next", env.reward_key))),
    )

    # Compute GAE and add it to the data
    with torch.no_grad():
        GAE(
            tensordict_data,
            params=loss_module.critic_network_params,
            target_params=loss_module.target_critic_network_params,
        )

    # Flatten and store in replay buffer
    data_view = tensordict_data.reshape(-1)
    replay_buffer.extend(data_view)

    return tensordict_data

def optimize_model(replay_buffer, loss_module, optim, num_epochs, frames_per_batch, minibatch_size):
    for _ in range(num_epochs):
        for _ in range(frames_per_batch // minibatch_size):
            subdata = replay_buffer.sample()
            loss_vals = loss_module(subdata)

            loss_value = (
                loss_vals["loss_objective"]
                + loss_vals["loss_critic"]
                + loss_vals["loss_entropy"]
            )
```

```

        loss_value.backward()
        torch.nn.utils.clip_grad_norm_(loss_module.parameters(), max_grad_norm)
        optim.step()
        optim.zero_grad()

```

```

pbar = tqdm(total=n_iters, desc="episode_reward_mean = 0")
episode_reward_mean_list = []

for _ in range(n_iters):
    tensordict_data = collect_data_and_compute_gae(collector, env, loss_module, replay_buffer)

    optimize_model(replay_buffer, loss_module, optim, num_epochs, frames_per_batch, minibatch_size)

    collector.update_policy_weights_()

    # Logging
    done = tensordict_data.get(("next", "agents", "done"))
    episode_reward_mean = (
        tensordict_data.get(("next", "agents", "episode_reward"))[done].mean().item()
    )
    episode_reward_mean_list.append(episode_reward_mean)
    pbar.set_description(f"episode_reward_mean = {episode_reward_mean}", refresh=False)
    pbar.update()

```

10.4.1 Results

```

plt.plot(episode_reward_mean_list)
plt.xlabel("Training iterations")
plt.ylabel("Reward")
plt.title("Episode reward mean")
plt.show()

```

Let's see how the agents solve the task:

```

env = VmasEnv("navigation", num_envs=1, continuous_actions=True, n_agents = 10, agent_radius= 0.05,
with torch.no_grad():
    env.rollout(
        max_steps=1000,
        policy=policy,
        callback=lambda env, _: env.render(),
        auto_cast_to_device=True,
        break_when_any_done=False,
    )

```

References

The implementation codes are from [Matteo Bettini - TorchRL](#)

Licensed under [CC BY-NC-ND 4.0](#). © Zoltán Barta, 2025.

11. Practice - Handling Heterogeneous Teams in MARL Using MADDPG

👤 Zoltán Barta, PhD student, Department of Artificial Intelligence

🕒 90 min read

📅 January 22, 2025

📚 Collective Intelligence



ELTE | IK
INFORMATIKAI KAR



DEPARTMENT
OF ARTIFICIAL
INTELLIGENCE



TorchRL

This practice notebook explores methods for training heterogeneous teams in multi-agent reinforcement learning, where agents may have different roles, objectives, or capabilities. Using the MADDPG algorithm, students will learn how to group agents, implement specialized learning strategies for each team, and manage cooperation and competition in complex environments. By the end of this module, students will be able to design and evaluate MARL systems for heterogeneous agent populations.

Table of Contents

- **11.1 Working with Heterogeneous Agent Teams in MARL**
 - 11.1.1 How do teams relate to each other?
- **11.2 Necessary Imports**
 - 11.2.1 Hyperparameters
 - 11.2.2 Environment
 - 11.2.3 Setting up Multi-Agent Environments with Team Separation
- **11.3 Multi-Agent Deep Deterministic Policy Gradient (MADDPG)**
 - 11.3.1 How It Works
 - 11.3.2 Algorithm Steps
 - 11.3.3 Implementation Tips
 - 11.3.4 Actor
 - 11.3.5 Wrapping Team Policies with TanhDelta Distributions

- 11.3.6 Adding Exploration to Deterministic Policies
- 11.3.7 Our Strategy: Additive Gaussian Noise
- 11.3.8 Why Additive Noise Leads to Intelligent Exploration
- 11.3.9 Learning the Environment Before the Noise
- 11.3.10 Learning to Handle Noise Later
- 11.3.11 Stable vs. Unstable Regions
- 11.3.12 Critic Construction for MADDPG
- **11.4 Training Loop**
 - 11.4.1 Stopping One Team Mid-Training
 - 11.4.2 Why do this?
 - 11.4.3 Results
- **References**

✓ 11.1 Working with Heterogeneous Agent Teams in MARL

Diversity poses a fundamental challenge: we can no longer assume a shared optimization target or a common policy architecture for the entire agent population. A single, shared algorithm would fail to capture the nuances of each agent's role, and forcing parameter sharing across incompatible objectives may lead to unstable or suboptimal learning dynamics.

To address this, a common and effective approach is to **group agents into teams** based on their similarity (e.g., agents with the same reward function or role in the system). Each team can then be assigned a separate learning algorithm, possibly even a different type of algorithm entirely (e.g., PPO for one team, Q-learning for another), that is better suited to their specific objective and constraints.

Within each team, however, agents are often **homogeneous**. In such cases, we can still apply parameter sharing by training a **single model instance** for all agents in that team. This preserves computational efficiency and enables the agents to learn from one another's experiences, without compromising the ability to specialize across teams.

This modular, team-based approach to MARL promotes scalability and flexibility, and aligns well with the structure of complex, multi-agent systems found in real-world applications, from autonomous vehicles and robot swarms to multi-player games and economic simulations.

11.1.1 How do teams relate to each other?

Teams can relate to each other in different ways:

- **Cooperative** (e.g., all agents work toward a shared goal),
- **Competitive** (e.g., zero-sum games between opposing teams),
- Or **mixed**, where partial cooperation and competition exist.

✓ 11.2 Necessary Imports

```
import copy
import tempfile

import torch

from matplotlib import pyplot as plt
```

```

from tensordict import TensorDictBase

from tensordict.nn import TensorDictModule, TensorDictSequential
from torch import multiprocessing

from torchrl.collectors import SyncDataCollector
from torchrl.data import LazyMemmapStorage, RandomSampler, ReplayBuffer

from torchrl.envs import (
    check_env_specs,
    ExplorationType,
    PettingZooEnv,
    RewardSum,
    set_exploration_type,
    TransformedEnv,
    VmasEnv,
)

from torchrl.modules import (
    AdditiveGaussianModule,
    MultiAgentMLP,
    ProbabilisticActor,
    TanhDelta,
)

from torchrl.objectives import DDPGLoss, SoftUpdate, ValueEstimators

from tqdm import tqdm

```

✓ 11.2.1 Hyperparameters

- **n_optimiser_steps**: Number of gradient descent steps taken after each rollout. Typical range: 1–500. Higher values extract more signal from each batch but risk overfitting and slower training.
- **train_batch_size**: Number of transitions passed to the optimizer per step (across all agents). Typical range: 64–2048. Larger batches improve gradient stability and GPU efficiency, while smaller ones reduce memory use and introduce regularizing noise.
- **lr**: Learning rate for the Adam optimizer (applied to both actor and critic networks). Typical range: $1e-5$ – $1e-3$. Start with $3e-4$ and adjust based on learning stability and convergence speed.
- **max_grad_norm**: Maximum ℓ_2 norm for gradient clipping. Typical range: 0.5–10.0. Helps stabilize training, especially in sparse-reward settings.
- **gamma**: Discount factor determining how much future rewards influence current updates. Typical range: 0.9–0.995. Higher values favor long-term planning but increase variance.
- **polyak_tau**: Coefficient for soft-updating target networks using Polyak averaging. Typical range: 0.001–0.02. Lower values lead to more stable target updates, but may lag during rapid policy changes.

```

# Seed
seed = 0
torch.manual_seed(seed)

# Devices
is_fork = multiprocessing.get_start_method() == "fork"
device = (
    torch.device(0)
    if torch.cuda.is_available() and not is_fork
    else torch.device("cpu")
)

```

```

)

# Sampling
frames_per_batch = 1_000
n_iters = 10
total_frames = frames_per_batch * n_iters

iteration_when_stop_training_evaders = n_iters // 2

# Replay buffer
memory_size = 1_000_000

# Training
n_optimiser_steps = 100
train_batch_size = 128
lr = 3e-4
max_grad_norm = 1.0

# DDPG
gamma = 0.99
polyak_tau = 0.005

```

✓ 11.2.2 Environment

In this tutorial, we work with a **multi-agent tagging scenario** where two opposing teams - **chasers** and **evaders** - interact in a shared 2D continuous world. The environment supports multiple agent groups running in parallel, where all agents receive observations and produce actions **synchronously** at each timestep. This setup is compatible with the TorchRL MARL API, which allows agents to be **grouped**, for example, by team, enabling clean separation and structured access to their data within a `TensorDict`.

We use the *simple_tag* environment, where red-circle chasers aim to catch green-circle evaders. Chasers receive a team reward (+10) when any of them touches an evader, while the evader receives a penalty (-10). Evaders are faster and more agile, adding asymmetry to the gameplay. The environment also includes randomly placed static obstacles (black circles) and features drag and elastic collisions to simulate realistic physics. Agent actions are 2D continuous force vectors that control their acceleration. Observations include the agent's position, velocity, distances to other entities (agents and obstacles), and evader velocities.

The environment can be instantiated using either **VMAS** or **PettingZoo**, with slight differences: PettingZoo penalizes evaders for leaving bounds, while VMAS prevents it physically. In VMAS, the reward signals for both teams are symmetric (equal and opposite), while in PettingZoo evader rewards tend to be lower due to boundary penalties.

To manage episode lengths, we cap each episode with a `max_steps` limit. This functionality is built into the simulators, but can also be added manually using TorchRL's `StepCounter` transform.

```

max_steps = 100 # Environment steps before done
n_chasers = 2
n_evaders = 1
n_obstacles = 2

num_vmas_envs = (
    frames_per_batch // max_steps
)
base_env = VmasEnv(
    scenario="navigation",
    num_envs=num_vmas_envs,
    continuous_actions=True,
    max_steps=max_steps,

```

```

device=device,
seed=seed,
# Scenario specific
# num_good_agents=n_evaders,
# num_adversaries=n_chasers,
# num_landmarks=n_obstacles,
)

```

✓ 11.2.3 Setting up multi-agent environments with team separation

After successfully setting up the environment, an important next step is handling scenarios where **multiple distinct teams** are present. In this notebook, we manage teams using a simple **Python dictionary**, where:

- The **key** is the team identifier (group, e.g. "pursuers" or "evaders"),
- The **value** is the component assigned to that team – such as a policy module, replay buffer, or learning algorithm.

To get this structure you can retrieve the agent grouping information of a multi-agent environment using `env.group_map`, which returns a dictionary mapping each group name to the list of agents it contains.

```
print(f"group_map: {base_env.group_map}")
```

```
➞ group_map: {'agents': ['agent_0', 'agent_1', 'agent_2', 'agent_3']}
```

```

print("action_spec:", base_env.full_action_spec)
print("reward_spec:", base_env.full_reward_spec)
print("done_spec:", base_env.full_done_spec)
print("observation_spec:", base_env.observation_spec)

```

```

➞ action_spec: Composite(
  agents: Composite(
    action: BoundedContinuous(
      shape=torch.Size([10, 4, 2]),
      space=ContinuousBox(
        low=Tensor(shape=torch.Size([10, 4, 2]), device=cpu, dtype=torch.float32,
        high=Tensor(shape=torch.Size([10, 4, 2]), device=cpu, dtype=torch.float32,
        device=cpu,
        dtype=torch.float32,
        domain=continuous),
      device=cpu,
      shape=torch.Size([10, 4])),
    device=cpu,
    shape=torch.Size([10]))
  reward_spec: Composite(
    agents: Composite(
      reward: UnboundedContinuous(
        shape=torch.Size([10, 4, 1]),
        space=ContinuousBox(
          low=Tensor(shape=torch.Size([10, 4, 1]), device=cpu, dtype=torch.float32,
          high=Tensor(shape=torch.Size([10, 4, 1]), device=cpu, dtype=torch.float32,
          device=cpu,
          dtype=torch.float32,
          domain=continuous),
        device=cpu,
        shape=torch.Size([10, 4])),
      device=cpu,
      shape=torch.Size([10]))
    done_spec: Composite(
      done: Categorical(
        shape=torch.Size([10, 1]),
        space=CategoricalBox(n=2),
        device=cpu,
        dtype=torch.bool,
        domain=discrete),

```

```

    terminated: Categorical(
        shape=torch.Size([10, 1]),
        space=CategoricalBox(n=2),
        device=cpu,
        dtype=torch.bool,
        domain=discrete),
    device=cpu,
    shape=torch.Size([10]))
observation_spec: Composite(
  agents: Composite(
    observation: UnboundedContinuous(
      shape=torch.Size([10, 4, 18]),
      space=ContinuousBox(
        low=Tensor(shape=torch.Size([10, 4, 18]), device=cpu, dtype=torch.float32),
        high=Tensor(shape=torch.Size([10, 4, 18]), device=cpu, dtype=torch.float32),
        device=cpu,
        dtype=torch.float32,
        domain=continuous),
    info: Composite(
      pos_rew: UnboundedContinuous(
        shape=torch.Size([10, 4, 1]),

```

```

env = TransformedEnv(
    base_env,
    RewardSum(
        in_keys=base_env.reward_keys,
        reset_keys=["_reset"] * len(base_env.group_map.keys()),
    ),
)

```

```
check_env_specs(env)
```

```

🔄 2025-05-16 19:54:29,834 [torchrl][INF0] check_env_specs succeeded!

```

✓ 11.3 Multi-Agent Deep Deterministic Policy Gradient (MADDPG)

Multi-Agent Deep Deterministic Policy Gradient (MADDPG) ([Paper](#)) extends the DDPG algorithm to multi-agent environments by enabling each agent to learn a decentralized policy while being trained with centralized information. It is particularly suited for environments involving both **cooperative** and **competitive** agents, where each agent operates from its own local observation but benefits from global information during training.

11.3.1 How It Works

1. Actor Network:

- Each agent is equipped with a deterministic actor network $\mu_{\theta}(o_i)$ that maps local observations o_i to continuous actions a_i .
- During environment interaction (execution), only this local policy is used—supporting decentralized decision-making.

2. Centralized Critic:

- During training, each agent also has a critic network $Q(s, a)$ that takes as input the **global state** or all agent observations and **joint actions**.
- This critic estimates the expected return for a given multi-agent configuration and is used solely for learning.

3. Actor Objective:

- The actor is trained to maximize the critic's estimate of the expected return, using the agent's own observations: $L_{\text{actor}} = -\mathbb{E}_{o_i} [Q(o_1, \dots, o_n, \mu_1(o_1), \dots, \mu_i(o_i), \dots, \mu_n(o_n))]$
- Gradients are computed through the centralized critic, but only the agent's own actor is updated.

4. Critic Update:

- The critic is trained with the TD error using target networks:

$$L_{\text{critic}} = \mathbb{E}_{(s,a,r,s')} [(Q(s,a) - y)^2], \quad y = r + \gamma Q_{\text{target}}(s', a')$$
- The target Q and actions a' are computed using slowly updated target networks.

5. Target Networks and Stability:

- Both actor and critic networks have corresponding **target networks**, which are updated using Polyak averaging: $\theta_{\text{target}} \leftarrow \tau \theta + (1 - \tau) \theta_{\text{target}}$
- This reduces instability by smoothing target estimates over time.

11.3.2 Algorithm Steps

1. Initialize:

- For each agent: initialize actor μ_θ , critic Q_ϕ , and their target networks.
- Set hyperparameters: learning rate, discount factor γ , soft update rate τ , and replay buffer.

2. For each training step:

- Collect environment interactions using current policies.
- Store transitions (o, a, r, o') in a shared replay buffer.
- Sample mini-batches of transitions and:
 - Update the critic using TD loss and target networks.
 - Update the actor using the critic's gradient signal.
 - Soft-update target networks.

3. Repeat:

- Continue training until convergence or maximum number of iterations is reached.

11.3.3 Implementation Tips

• Centralized Training:

- Ensure that the critic has access to full global information (joint observations and actions) during training.

• Decentralized Execution:

- Actors should rely only on local observations, allowing them to operate independently at inference time.

• Parameter Sharing (Optional):

- In cooperative teams, parameter sharing across agents with identical roles can speed up training and reduce resource usage.

• Replay Buffer:

- Use a shared replay buffer to store full joint experiences, enabling proper credit assignment during critic updates.

• Target Networks:

- Keep τ small (e.g., 0.005) to stabilize target updates and reduce training oscillations.

✓ 11.3.4 Actor

For each agent group, we define a decentralized actor as a `TensorDictModule` that maps local observations to action parameters using a `MultiAgentMLP`, optionally sharing weights among agents within the group.

```
policy_modules = {}
for group, agents in env.group_map.items():
    share_parameters_policy = True # Can change this based on the group

    policy_net = MultiAgentMLP(
        n_agent_inputs=env.observation_spec[group, "observation"].shape[
            -1
        ], # n_obs_per_agent
        n_agent_outputs=env.full_action_spec[group, "action"].shape[
            -1
        ], # n_actions_per_agents
        n_agents=len(agents), # Number of agents in the group
        centralised=False, # the policies are decentralised (i.e., each agent will act from its
        share_params=share_parameters_policy,
        device=device,
        depth=2,
        num_cells=256,
        activation_class=torch.nn.Tanh,
    )

    policy_module = TensorDictModule(
        policy_net,
        in_keys=[(group, "observation")],
        out_keys=[(group, "param")],
    ) # We just name the input and output that the network will read and write to the input ten
    policy_modules[group] = policy_module
```

✓ 11.3.5 Wrapping Team Policies with TanhDelta Distributions

Once we have our policy networks defined for each team, we need to convert their raw outputs (typically unbounded action parameters) into valid actions that match the environment's action space. This is done using a `ProbabilisticActor` wrapper with a `TanhDelta` distribution.

Why `TanhDelta`?

- **Bounded continuous actions:** Many environments define actions within a fixed range (e.g. $[-1, 1]$). The `TanhDelta` distribution applies a `tanh` function to the output of the network to ensure actions stay within these bounds.
- **Deterministic policy:** Unlike `Normal` or other stochastic distributions, `TanhDelta` represents a **deterministic** policy (like in standard DDPG), which is typically preferred in MADDPG to reduce variance during training.

```
policies = {}
for group, _agents in env.group_map.items():
    policy = ProbabilisticActor(
        module=policy_modules[group],
        spec=env.full_action_spec[group, "action"],
        in_keys=[(group, "param")],
        out_keys=[(group, "action")],
        distribution_class=TanhDelta,
        distribution_kwargs={
```

```

        "low": env.full_action_spec_unbatched[group, "action"].space.low,
        "high": env.full_action_spec_unbatched[group, "action"].space.high,
    },
    return_log_prob=False,
)
policies[group] = policy

```

11.3.6 Adding Exploration to Deterministic Policies

Since MADDPG uses **deterministic actors** (via the `TanhDelta` distribution), exploration during training cannot rely on sampling from a stochastic policy like in standard policy gradient methods. Instead, we inject **external noise** into the actions to encourage exploration.

Why not ϵ -greedy?

- **ϵ -greedy** strategies are common in discrete action spaces, where the agent can randomly pick an alternative action with some probability ϵ .
- In **continuous action spaces**, however, ϵ -greedy is not well-defined or effective.
- Instead, we use **additive noise** — a smoother and more natural way to perturb continuous actions during exploration.

11.3.7 Our Strategy: Additive Gaussian Noise

We wrap each policy in a `TensorDictSequential` module that appends an `AdditiveGaussianModule` on top of the original deterministic actor. This:

- Adds **Gaussian noise** to the actions sampled from the actor.
- Uses **annealing** to reduce the noise over time:
 - Starts with a high standard deviation (`sigma_init = 0.9`) to promote exploration.
 - Gradually reduces it to a low value (`sigma_end = 0.1`) over the first half of training (`total_frames // 2`).

11.3.8 Why Additive Noise Leads to Intelligent Exploration

Using **additive Gaussian noise** in a deterministic policy does more than just inject randomness — it creates a natural *learning signal* for how the agent should behave in **familiar vs. unfamiliar situations**.

11.3.9 Learning the Environment Before the Noise

- Early in training, the agent is still learning the **structure of the environment** — how states, actions, and rewards relate.
- At this stage, the **high noise level** encourages broad exploration and prevents premature convergence to suboptimal behaviors.

11.3.10 Learning to Handle Noise Later

- As training progresses, the policy begins to adapt not just to the environment, but also to the **effect of noise itself**.
- Because the actor is deterministic, it "sees" the added noise as part of the environment — and it can learn to **compensate** for or **leverage** it strategically.

✓ 11.3.11 Stable vs. unstable regions

- Over time, this leads to a useful behavioral pattern:
 - In **well-known, stable parts** of the environment, the policy acts **more deterministically** — it has learned what to do there.
 - In **uncertain or novel regions**, the residual noise still present causes more **exploratory behavior**.

This approach gives us a form of **adaptive exploration** — without needing to learn a separate stochastic policy or use tricks like ϵ -greedy. It fits naturally into the deterministic framework of MADDPG, while still allowing for deep, state-sensitive exploration dynamics.

```
exploration_policies = {}
for group, _agents in env.group_map.items():
    exploration_policy = TensorDictSequential(
        policies[group],
        AdditiveGaussianModule(
            spec=policies[group].spec,
            annealing_num_steps=total_frames
            // 2, # Number of frames after which sigma is sigma_end
            action_key=(group, "action"),
            sigma_init=0.9, # Initial value of the sigma
            sigma_end=0.1, # Final value of the sigma
        ),
    )
    exploration_policies[group] = exploration_policy
```

✓ 11.3.12 Critic Construction for MADDPG

A separate **critic** is built for each agent group defined in the environment's `group_map`. Each critic follows the centralized training paradigm of MADDPG, where both the agent's **observations** and **actions** are used as inputs. To facilitate this, a `TensorDictModule` first concatenates each agent's observation and action into a new entry (e.g., `("group", "obs_action")`). This is then passed to a `MultiAgentMLP` that predicts a value estimate `("state_action_value")` for each agent in the group.

Critics can optionally **share parameters** across agents within a group (`share_params=True`), which is suitable when agents have identical roles. By organizing critics in a `TensorDictSequential`, the modules operate in sequence, cleanly encapsulating both the preprocessing and value computation steps.

```
critics = {}
for group, agents in env.group_map.items():
    share_parameters_critic = True # Can change for each group
    MADDPG = True # IDDPG if False, can change for each group

    # This module applies the lambda function: reading the action and observation entries for th
    # and concatenating them in a new `` (group, "obs_action") `` entry
    cat_module = TensorDictModule(
        lambda obs, action: torch.cat([obs, action], dim=-1),
        in_keys=[(group, "observation"), (group, "action")],
        out_keys=[(group, "obs_action")],
    )

    critic_module = TensorDictModule(
        module=MultiAgentMLP(
            n_agent_inputs=env.observation_spec[group, "observation"].shape[-1]
            + env.full_action_spec[group, "action"].shape[-1],
            n_agent_outputs=1, # 1 value per agent
```

```

        n_agents=len(agents),
        centralised=MADDPG,
        share_params=share_parameters_critic,
        device=device,
        depth=2,
        num_cells=256,
        activation_class=torch.nn.Tanh,
    ),
    in_keys=[(group, "obs_action")],
    out_keys=[
        (group, "state_action_value")
    ],
)

critics[group] = TensorDictSequential(
    cat_module, critic_module
)

```

Put all of the actor policies into a single module, because it will apply all contained submod
agents_exploration_policy = TensorDictSequential(*exploration_policies.values())

```

collector = SyncDataCollector(
    env,
    agents_exploration_policy,
    device=device,
    frames_per_batch=frames_per_batch,
    total_frames=total_frames,
)

```

We define different replay buffers and optimizers for all of the teams

```

replay_buffers = {}
for group, _agents in env.group_map.items():
    replay_buffer = ReplayBuffer(
        storage=LazyMemmapStorage(
            memory_size,
        ), # We will store up to memory_size multi-agent transitions
        sampler=RandomSampler(),
        batch_size=train_batch_size, # We will sample batches of this size
    )
    if device.type != "cpu":
        replay_buffer.append_transform(lambda x: x.to(device))
    replay_buffers[group] = replay_buffer

```

TorchRL provides the `DDPGLoss` class to simplify training with DDPG, encapsulating the algorithm's core logic while allowing assigning different policies and critics to each agent group. Also we define separate optimizers for each group as well.

```

losses = {}
for group, _agents in env.group_map.items():
    loss_module = DDPGLoss(
        actor_network=policies[group], # Use the non-explorative policies
        value_network=critics[group],
        delay_value=True, # Whether to use a target network for the value
        loss_function="l2",
    )
    loss_module.set_keys(
        state_action_value=(group, "state_action_value"),
        reward=(group, "reward"),
    )

```

```

        done=(group, "done"),
        terminated=(group, "terminated"),
    )
    loss_module.make_value_estimator(ValueEstimators.TD0, gamma=gamma)

    losses[group] = loss_module

target_updaters = {
    group: SoftUpdate(loss, tau=polyak_tau) for group, loss in losses.items()
}

optimisers = {
    group: {
        "loss_actor": torch.optim.Adam(
            loss.actor_network_params.flatten_keys().values(), lr=lr
        ),
        "loss_value": torch.optim.Adam(
            loss.value_network_params.flatten_keys().values(), lr=lr
        ),
    }
    for group, loss in losses.items()
}

```

✓ 11.4 Training Loop

1. Data Collection:

- Batches of experiences are gathered from the environment using the current policy.
- These batches contain observations, actions, rewards, and termination signals for all agent groups.

2. Group-wise Data Processing:

- The collected data is separated by agent group (e.g., evaders vs. chasers).
- Each group's batch is processed and stored in its dedicated **replay buffer**, ensuring group-specific training.

3. Optimization:

- Each group is trained independently using its own loss function, optimizer, and replay buffer.
- For a fixed number of gradient steps, mini-batches are sampled and passed through the loss module, followed by backpropagation and optional gradient clipping.
- Policy and target networks are updated as needed.

4. Selective Training:

- Training can be selectively disabled for specific agent groups after a certain number of iterations (e.g., freezing evaders).

5. Logging and Monitoring:

- After each iteration, the mean episode reward is computed for each group and logged.
- These metrics are used to update the progress bar and monitor learning dynamics in real time.

A utility function that will shape the data samples.

```

def process_batch(batch: TensorDictBase) -> TensorDictBase:
    """
    If the `(group, "terminated")` and `(group, "done")` keys are not present, create them by ex
    `"terminated"` and `"done"`.
    """

```

This is needed to present them with the same shape as the reward to the loss.
"""

```
for group in env.group_map.keys():
    keys = list(batch.keys(True, True))
    group_shape = batch.get_item_shape(group)
    nested_done_key = ("next", group, "done")
    nested_terminated_key = ("next", group, "terminated")
    if nested_done_key not in keys:
        batch.set(
            nested_done_key,
            batch.get(("next", "done")).unsqueeze(-1).expand((*group_shape, 1)),
        )
    if nested_terminated_key not in keys:
        batch.set(
            nested_terminated_key,
            batch.get(("next", "terminated")).
            .unsqueeze(-1)
            .expand((*group_shape, 1)),
        )
return batch
```

```
def collect_and_store_group_batches(
    batch, env, train_group_map, replay_buffers
):
    current_frames = batch.numel()
    batch = process_batch(batch) # Expands "done"/"terminated" if needed

    for group in train_group_map.keys():
        group_batch = batch.exclude(
            *[
                key
                for _group in env.group_map.keys()
                if _group != group
                for key in [_group, ("next", _group)]
            ]
        )
        group_batch = group_batch.reshape(-1)
        replay_buffers[group].extend(group_batch)

    return batch, current_frames
```

```
def optimize_groups(
    train_group_map,
    replay_buffers,
    losses,
    optimisers,
    target_updaters,
    exploration_policies,
    current_frames,
    n_optimiser_steps,
    max_grad_norm,
):
    for group in train_group_map.keys():
        for _ in range(n_optimiser_steps):
            subdata = replay_buffers[group].sample()
            loss_vals = losses[group](subdata)

            for loss_name in ["loss_actor", "loss_value"]:
                loss = loss_vals[loss_name]
                optimiser = optimisers[group][loss_name]

                loss.backward()
                params = optimiser.param_groups[0]["params"]
```

```

        torch.nn.utils.clip_grad_norm_(params, max_grad_norm)
        optimiser.step()
        optimiser.zero_grad()

    target_updaters[group].step()

    exploration_policies[group][-1].step(current_frames)

def log_episode_rewards(batch, env, episode_reward_mean_map):
    for group in env.group_map.keys():
        episode_reward_mean = (
            batch.get(("next", group, "episode_reward"))[
                batch.get(("next", group, "done"))
            ]
            .mean()
            .item()
        )
        episode_reward_mean_map[group].append(episode_reward_mean)

    return episode_reward_mean_map

```

11.4.1 Stopping One Team Mid-Training

At a predefined training iteration (e.g. halfway through), we **stop updating one of the teams**, usually the evaders or defenders, while the other continues to learn. This creates a dynamic where:

- One team becomes a **static opponent**,
- The other team continues adapting, trying to exploit the frozen team's strategy.

✓ 11.4.2 Why do this?

- The goal is **not to find a perfect equilibrium**, but to **demonstrate how separate training control per team** can lead to interesting multi-agent behaviors.
- It mimics real-world scenarios where one side of the environment is fixed (e.g. a known strategy, a rule-based opponent, or a pre-trained policy).
- It encourages the learning team to continuously adapt, resulting in a kind of **non-stationary game** dynamic.

```

pbar = tqdm(
    total=n_iters,
    desc="", ".join(
        [f"episode_reward_mean_{group} = 0" for group in env.group_map.keys()]
    ),
)

episode_reward_mean_map = {group: [] for group in env.group_map.keys()}
train_group_map = copy.deepcopy(env.group_map)

for iteration, batch in enumerate(collector):
    batch, current_frames = collect_and_store_group_batches(batch, env, train_group_map, replay_

    optimize_groups(
        train_group_map,
        replay_buffers,
        losses,
        optimisers,
        target_updaters,
        exploration_policies,
        current_frames,

```

```

        n_optimiser_steps,
        max_grad_norm,
    )

    episode_reward_mean_map = log_episode_rewards(batch, env, episode_reward_mean_map)

    pbar.set_description(
        ", ".join(
            [
                f"episode_reward_mean_{group} = {episode_reward_mean_map[group][-1]}"
                for group in env.group_map.keys()
            ]
        ),
        refresh=False,
    )
    pbar.update()

```

```

➡ episode_reward_mean_agents = -6.946998596191406: 100%|██████████| 10/10 [00:05<00:00, 1.73it

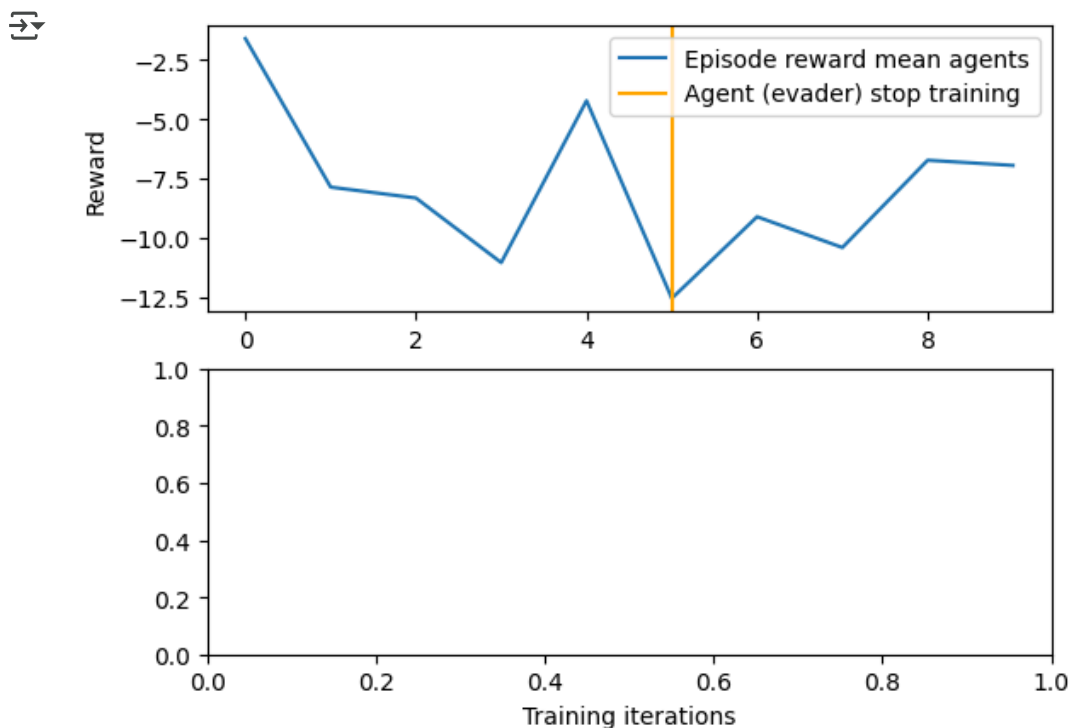
```

11.4.3 Results

```

fig, axs = plt.subplots(2, 1)
for i, group in enumerate(env.group_map.keys()):
    axs[i].plot(episode_reward_mean_map[group], label=f"Episode reward mean {group}")
    axs[i].set_ylabel("Reward")
    axs[i].axvline(
        x=iteration_when_stop_training_evaders,
        label="Agent (evader) stop training",
        color="orange",
    )
    axs[i].legend()
axs[-1].set_xlabel("Training iterations")
plt.show()

```



Let's see how the agents solve the task:

```
with torch.no_grad():
    env.rollout(
        max_steps=1000,
        policy=policies['agents'],s
        callback=lambda env, _: env.render(),
        auto_cast_to_device=True,
        break_when_any_done=False,
    )
```

✓ References

The implementation codes are from [Matteo Bettini - TorchRL](#)

Licensed under [CC BY-NC-ND 4.0](#). © Zoltán Barta, 2025.

✓ 12. Practice - Communication in MARL

👤 Zoltán Barta, PhD student, Department of Artificial Intelligence

🕒 90 min read

📅 January 22, 2025

📖 Collective Intelligence



ELTE | IK
INFORMATIKAI KAR



DEPARTMENT
OF ARTIFICIAL
INTELLIGENCE



TorchRL

This practice notebook introduces the role of communication in multi-agent reinforcement learning (MARL). Students will examine how agents can share information to coordinate actions and overcome partial observability, using both explicit and implicit communication strategies. Through hands-on exercises with MAPPO in environments requiring communication, students will gain practical experience in designing, training, and evaluating MARL systems with communication protocols.

[A Survey of Multi-Agent Deep Reinforcement Learning with Communication](#)

Table of Contents

- **12.1 Communication Overview**
 - 12.1.1 How Can We Model Communication?
 - 12.1.2 Key Challenges
 - 12.1.3 Performance Metrics
 - 12.1.4 Environment: `simple_speaker_listener`
- **12.2 Training MAPPO**
 - 12.2.1 Results
- **References**

✓ 12.1 Communication Overview

In fully observable environments, agents might act independently by reacting only to the environment state. But in **partially observable or cooperative settings**, agents often need to:

- Share **intentions**,
- Signal **observations** others can't access,
- Or coordinate actions to avoid conflict or redundancy.

Communication allows agents to **synchronize** behavior and make decisions based on **shared context**, not just local inputs.

12.1.1 How Can We Model Communication?

In multi-agent reinforcement learning (MARL), **communication** allows agents to share information and coordinate their actions to solve complex tasks. Conceptually, communication can be thought of as a **special kind of action** that does **not directly affect the environment**, but instead influences other agents' behavior by providing them with additional information.

There are two primary approaches to modeling communication in MARL:

1. Explicit Communication

- Agents **output messages**—such as learned vectors, symbolic tokens, or predefined signals—which are passed through structured communication channels to other agents.
- This method gives direct control over the message content and structure, making it useful in scenarios where the communication protocol itself is a key research focus (e.g., emergent language, cooperation protocols).
- These messages are typically included as additional inputs in the recipient agents' policies.
- The challenge is that the agents must **jointly learn** how to construct useful messages, how to interpret received ones, and how to integrate communication into decision-making.

2. Implicit Communication

- Here, communication is **not explicitly defined**. Instead, agents learn to **embed information** in their observable behavior, network activations, or actions, which indirectly influence their peers.
- For example, one agent may move in a certain pattern to signal intent or share state, and another agent learns to pick up on this.
- This form of communication often **emerges as a byproduct** of optimizing shared objectives, especially in partially observable or decentralized settings.
- Although less interpretable, it requires no explicit communication infrastructure and is more biologically and physically plausible in some domains.

12.1.2 Key Challenges

Incorporating communication into MARL introduces several unique challenges:

- **Non-Stationarity**: As each agent's policy—including its messaging strategy—evolves during training, the learning environment becomes highly unstable.
- **Credit Assignment**: It becomes difficult to trace back which messages led to which outcomes, especially over long time horizons and noisy channels.
- **Message Interpretation**: Beyond generating messages, agents must also **learn to interpret** the messages they receive—meaning that learning effective communication becomes a **two-sided problem**.
- **Objective Alignment**: Agents must balance between **learning to solve the task** and **learning how and what to communicate**, often under separate gradients or reward signals.

- **Bandwidth and Latency Constraints:** In real-world scenarios, communication may be limited by cost, time, or energy—forcing agents to **compress** or **prioritize** the information they transmit.
- **Protocol Emergence:** Designing or inducing **emergent, interpretable, and task-relevant communication protocols** is still an open problem, particularly in adversarial or mixed-motive settings.

✓ 12.1.3 Performance Metrics

To properly evaluate communication in MARL, we must measure both **task performance** and **communication efficiency**. Common metrics include:

- **Task Reward:** The overall success in completing the shared objective.
- **Message Entropy / Bits per Message:** Quantifies how much information is being transmitted.
- **Mutual Information:** Measures how informative messages are about observations or actions.
- **Communication Efficiency:** Reward improvement per bit transmitted—higher means better use of bandwidth.
- **Message Sparsity:** Fraction of timesteps where agents actually send non-trivial messages.
- **Robustness to Channel Noise:** Tests how fragile or resilient the learned communication is under perturbation or partial failure.

By combining these metrics, we can assess not just whether agents communicate, but **how well and how meaningfully** they do so.

```
# Torch
import torch
import torch.nn as nn
# Tensordict modules
from tensordict.nn import set_composite_lp_aggregate, TensorDictModule
from tensordict.nn.distributions import NormalParamExtractor

# Data collection
from torchrl.collectors import SyncDataCollector
from torchrl.data.replay_buffers import ReplayBuffer
from torchrl.data.replay_buffers.samplers import SamplerWithoutReplacement
from torchrl.data.replay_buffers.storages import LazyTensorStorage

# Env
from torchrl.envs import RewardSum, TransformedEnv
from torchrl.envs.libs.vmas import VmasEnv
from torchrl.envs.utils import check_env_specs

# Multi-agent network
from torchrl.modules import MultiAgentMLP, ProbabilisticActor, TanhNormal

# Loss
from torchrl.objectives import ClipPPOLoss, ValueEstimators
# Utils
torch.manual_seed(0)
from matplotlib import pyplot as plt
from tqdm import tqdm

device = (
    torch.device(0)
    if torch.cuda.is_available()
    else torch.device("cpu")
)

# Sampling
frames_per_batch = 10_000 # Number of team frames collected per training iteration
```

```

n_iters = 30 # Number of sampling and training iterations
total_frames = frames_per_batch * n_iters

# Training
num_epochs = 30 # Number of optimization steps per training iteration
minibatch_size = 400 # Size of the mini-batches in each optimization step
lr = 3e-4 # Learning rate
max_grad_norm = 1.0 # Maximum norm for the gradients

# PPO
clip_epsilon = 0.2 # clip value for PPO loss
gamma = 0.99 # discount factor
lmbda = 0.9 # lambda for generalised advantage estimation
entropy_eps = 1e-4 # coefficient of the entropy term in the PPO loss

# disable log-prob aggregation
set_composite_lp_aggregate(False).set()

```

✓ 12.1.4 Environment: `simple_speaker_listener`

The `simple_speaker_listener` scenario in VMAS/MPE is a minimal yet powerful setup for testing **explicit communication** in multi-agent systems. In this environment, one agent is designated as the **speaker**, while the others act as **listeners**. The **goal** of the environment is for the listeners to reach a target location – but only the speaker knows where this goal is. Importantly, the speaker itself **cannot move**. Its only role is to **communicate useful information** to the listeners through an explicit message channel, enabling them to navigate toward the goal based on this shared information. Observations are **agent-specific**. The **speaker's observation** includes the true goal location (typically given in coordinates), while the **listeners receive** their own state information (such as position and velocity), along with the **messages sent by the speaker**. In this scenario, all agents receive a shared reward based on how close the listeners are to the goal, encouraging cooperation. Since only the speaker knows the goal, it must learn to communicate effectively to help the listeners navigate. This allows the environment to simulate a realistic setting of **asymmetric information** and communication. The key challenge in this task is learning an **effective communication protocol**: the speaker must learn to encode helpful signals into the message, and the listeners must learn how to interpret those signals and act accordingly. Coordination must emerge from this shared information, despite the partial observability and role asymmetry.

```

device = torch.device("cuda" if torch.cuda.is_available() else "cpu")

env = VmasEnv(
    scenario="simple_speaker_listener",
    num_envs=1,
    continuous_actions=True, # VMAS supports both continuous and discrete actions
    max_steps=500,
    device=device,
    # Scenario kwarg
)

env = TransformedEnv(
    env,
    RewardSum(in_keys=env.reward_keys, out_keys=[("listener", "episode_reward"), ("speaker", "episode_reward")])
)
check_env_specs(env)

```

🔄 2025-05-12 10:43:09,801 [torchrl][INFO] check_env_specs succeeded!

The setup of the two teams is similar to the previous practice, but in this case we are using MAPPO instead of MADDPG.

```

policy_modules = {}
for group, agents in env.group_map.items():
    policy_net = torch.nn.Sequential(
        MultiAgentMLP(
            n_agent_inputs=env.observation_spec[(group, "observation")].shape[-1]
            - 1, # n_obs_per_agent
            n_agent_outputs=env.action_spec[group]['action'].shape[-1] * 2, # 2 * n_actions_per_age
            n_agents=len(agents), # n_agents
            centralised=False, # the policies are decentralised (ie each agent will act from its ob
            share_params=True,
            device=device,
            depth=2,
            num_cells=64,
            activation_class=torch.nn.Tanh,
        ), # this will just separate the last dimension into two outputs: a loc and a non-negative sc
        NormalParamExtractor(),
    )
    policy_module = TensorDictModule(
        policy_net,
        in_keys=[(group, "observation")],
        out_keys=[(group, "loc"), (group, "scale")],
    )
    policy_modules[group] = ProbabilisticActor(
        module=policy_module,
        spec=env.action_spec_unbatched[(group, "action")],
        in_keys=[(group, "loc"), (group, "scale")],
        out_keys=[(group, "action")],
        distribution_class=TanhNormal,
        distribution_kwargs={
            "low": env.full_action_spec_unbatched[(group, 'action')].space.low,
            "high": env.full_action_spec_unbatched[(group, 'action')].space.high,
        },
        return_log_prob=True,
    )

print(policy_modules["listener"](env.reset()))
print("Listener working!")
print(policy_modules["speaker"](env.reset()))
print("Speaker working!")

# TensorDict(
    fields={
        done: Tensor(shape=torch.Size([1, 1]), device=cpu, dtype=torch.bool, is_shared=False),
        listener: TensorDict(
            fields={
                action: Tensor(shape=torch.Size([1, 1, 2]), device=cpu, dtype=torch.float32,
                action_log_prob: Tensor(shape=torch.Size([1, 1]), device=cpu, dtype=torch.fl
                episode_reward: Tensor(shape=torch.Size([1, 1, 1]), device=cpu, dtype=torch.
                loc: Tensor(shape=torch.Size([1, 1, 2]), device=cpu, dtype=torch.float32, is_
                observation: Tensor(shape=torch.Size([1, 1, 11]), device=cpu, dtype=torch.fl
                scale: Tensor(shape=torch.Size([1, 1, 2]), device=cpu, dtype=torch.float32, :
            }, batch_size=torch.Size([1, 1]),
            device=cpu,
            is_shared=False),
        speaker: TensorDict(
            fields={
                episode_reward: Tensor(shape=torch.Size([1, 1, 1]), device=cpu, dtype=torch.
                observation: Tensor(shape=torch.Size([1, 1, 3]), device=cpu, dtype=torch.flo
            }, batch_size=torch.Size([1, 1]),
            device=cpu,
            is_shared=False),
        terminated: Tensor(shape=torch.Size([1, 1]), device=cpu, dtype=torch.bool, is_shared=
    }, batch_size=torch.Size([1]),
    device=cpu,

```

```

        is_shared=False)
Listener working!
TensorDict(
  fields={
    done: Tensor(shape=torch.Size([1, 1]), device=cpu, dtype=torch.bool, is_shared=False),
    listener: TensorDict(
      fields={
        episode_reward: Tensor(shape=torch.Size([1, 1, 1]), device=cpu, dtype=torch.float32, is_shared=False),
        observation: Tensor(shape=torch.Size([1, 1, 11]), device=cpu, dtype=torch.float32, is_shared=False),
        batch_size: torch.Size([1, 1]),
        device=cpu,
        is_shared=False),
    speaker: TensorDict(
      fields={
        action: Tensor(shape=torch.Size([1, 1, 5]), device=cpu, dtype=torch.float32, is_shared=False),
        action_log_prob: Tensor(shape=torch.Size([1, 1]), device=cpu, dtype=torch.float32, is_shared=False),
        episode_reward: Tensor(shape=torch.Size([1, 1, 1]), device=cpu, dtype=torch.float32, is_shared=False),
        loc: Tensor(shape=torch.Size([1, 1, 5]), device=cpu, dtype=torch.float32, is_shared=False),
        observation: Tensor(shape=torch.Size([1, 1, 3]), device=cpu, dtype=torch.float32, is_shared=False),
        scale: Tensor(shape=torch.Size([1, 1, 5]), device=cpu, dtype=torch.float32, is_shared=False),
        batch_size: torch.Size([1, 1]),
        device=cpu,
        is_shared=False),
    terminated: Tensor(shape=torch.Size([1, 1]), device=cpu, dtype=torch.bool, is_shared=False),
    batch_size: torch.Size([1]),
    device=cpu,
    is_shared=False)
Speaker working!

```



```

critics = {}
for group, agents in env.group_map.items():

    critic_net = MultiAgentMLP(
        n_agent_inputs=env.observation_spec[group, "observation"].shape[-1],
        n_agent_outputs=1, # 1 value per agent
        n_agents=len(agents), # n_agents
        centralised=True,
        share_params=True,
        device=device,
        depth=2,
        num_cells=256,
        activation_class=torch.nn.Tanh,
    )

    critic = TensorDictModule(
        module=critic_net,
        in_keys=[(group, "observation")],
        out_keys=[(group, "state_value")],
    )
    critics[group] = critic

print(critics["listener"](env.reset()))
print("Listener critic working!")
print(critics["speaker"](env.reset()))
print("Speaker critic working!")

```



```

TensorDict(
  fields={
    done: Tensor(shape=torch.Size([1, 1]), device=cpu, dtype=torch.bool, is_shared=False),
    listener: TensorDict(
      fields={
        episode_reward: Tensor(shape=torch.Size([1, 1, 1]), device=cpu, dtype=torch.float32, is_shared=False),
        observation: Tensor(shape=torch.Size([1, 1, 11]), device=cpu, dtype=torch.float32, is_shared=False),
        state_value: Tensor(shape=torch.Size([1, 1, 1]), device=cpu, dtype=torch.float32, is_shared=False),
        batch_size: torch.Size([1, 1]),
        device=cpu,
        is_shared=False),
    speaker: TensorDict(
      fields={
        action: Tensor(shape=torch.Size([1, 1, 5]), device=cpu, dtype=torch.float32, is_shared=False),
        action_log_prob: Tensor(shape=torch.Size([1, 1]), device=cpu, dtype=torch.float32, is_shared=False),
        episode_reward: Tensor(shape=torch.Size([1, 1, 1]), device=cpu, dtype=torch.float32, is_shared=False),
        loc: Tensor(shape=torch.Size([1, 1, 5]), device=cpu, dtype=torch.float32, is_shared=False),
        observation: Tensor(shape=torch.Size([1, 1, 3]), device=cpu, dtype=torch.float32, is_shared=False),
        scale: Tensor(shape=torch.Size([1, 1, 5]), device=cpu, dtype=torch.float32, is_shared=False),
        state_value: Tensor(shape=torch.Size([1, 1, 1]), device=cpu, dtype=torch.float32, is_shared=False),
        batch_size: torch.Size([1, 1]),
        device=cpu,
        is_shared=False),
    terminated: Tensor(shape=torch.Size([1, 1]), device=cpu, dtype=torch.bool, is_shared=False),
    batch_size: torch.Size([1]),
    device=cpu,
    is_shared=False)

```

```

        batch_size=torch.Size([1, 1]),
        device=cpu,
        is_shared=False),
    speaker: TensorDict(
        fields={
            episode_reward: Tensor(shape=torch.Size([1, 1, 1]), device=cpu, dtype=torch.float32, is_shared=False),
            observation: Tensor(shape=torch.Size([1, 1, 3]), device=cpu, dtype=torch.float32, is_shared=False),
            batch_size=torch.Size([1, 1]),
            device=cpu,
            is_shared=False),
        terminated: Tensor(shape=torch.Size([1, 1]), device=cpu, dtype=torch.bool, is_shared=False),
        batch_size=torch.Size([1]),
        device=cpu,
        is_shared=False)
Listener critic working!
TensorDict(
  fields={
    done: Tensor(shape=torch.Size([1, 1]), device=cpu, dtype=torch.bool, is_shared=False),
    listener: TensorDict(
      fields={
        episode_reward: Tensor(shape=torch.Size([1, 1, 1]), device=cpu, dtype=torch.float32, is_shared=False),
        observation: Tensor(shape=torch.Size([1, 1, 11]), device=cpu, dtype=torch.float32, is_shared=False),
        batch_size=torch.Size([1, 1]),
        device=cpu,
        is_shared=False),
      speaker: TensorDict(
        fields={
          episode_reward: Tensor(shape=torch.Size([1, 1, 1]), device=cpu, dtype=torch.float32, is_shared=False),
          observation: Tensor(shape=torch.Size([1, 1, 3]), device=cpu, dtype=torch.float32, is_shared=False),
          state_value: Tensor(shape=torch.Size([1, 1, 1]), device=cpu, dtype=torch.float32, is_shared=False),
          batch_size=torch.Size([1, 1]),
          device=cpu,
          is_shared=False),
          terminated: Tensor(shape=torch.Size([1, 1]), device=cpu, dtype=torch.bool, is_shared=False),
          batch_size=torch.Size([1]),
          device=cpu,
          is_shared=False)
    }
  )
Speaker critic working!

```

```

from tensordict.nn import TensorDictSequential

```

```

policies = TensorDictSequential(*policy_modules.values())
collector = SyncDataCollector(
    env,
    policies,
    device=device,
    frames_per_batch=frames_per_batch,
    total_frames=total_frames,
)

```

```

replay_buffers = {}
for group in env.group_map.keys():
    replay_buffer = ReplayBuffer(
        storage=LazyTensorStorage(
            frames_per_batch, device=device
        ), # We store the frames_per_batch collected at each iteration
        sampler=SamplerWithoutReplacement(),
        batch_size=minibatch_size, # We will sample minibatches of this size
    )
    replay_buffers[group] = replay_buffer

```

```

losses = {}
optimizers = {}
gaes = {}

```

```

for group in env.group_map.keys():

    loss_module = ClipPPOLoss(
        actor_network=policy_modules[group],
        critic_network=critics[group],
        clip_epsilon=0.2,
        entropy_coef=0.01,
        normalize_advantage=False, # Important to avoid normalizing across the agent dimension
    )
    loss_module.set_keys( # We have to tell the loss where to find the keys
        reward=(group, 'reward'),
        action=(group, 'action'),
        value=(group, "state_value"),
        # These last 2 keys will be expanded to match the reward shape
        done=(group,"done"),
        terminated=(group,"terminated"),
        advantage=(group, "advantage"), # This is the key where the GAE will write
    )

    loss_module.make_value_estimator(
        ValueEstimators.GAE, gamma=0.99, lmbda=0.9
    ) # We build GAE
    GAE = loss_module.value_estimator
    losses[group] = loss_module
    GAE.set_keys(
        advantage=(group, "advantage"), # This is the key where the GAE will write
    )
    gaes[group] = GAE

optimizers = {x:torch.optim.Adam(losses[x].parameters(), lr=3e-4) for x in env.group_map.keys()}

```

✓ 12.2 Training MAPPO

1. Data Collection:

- A batch of experiences is collected from the environment using the `collector`.
- The batch is passed through a preprocessing step (`process_batch`) that ensures each agent group has its own "done" and "terminated" keys.
- These keys are expanded to match the reward tensor shape, which is required for proper alignment during value estimation.

2. Advantage Computation (GAE):

- For each agent group, **Generalized Advantage Estimation (GAE)** is computed without tracking gradients.
- The computed advantages and value targets are added to the data structure, enabling loss calculation for both the policy and value functions.

3. Storing Experience:

- The batch is reshaped to flatten the dimensions and then added to each group's `replay_buffer`.
- This allows later sampling of randomized mini-batches for training while decoupling collection from optimization.

4. Policy and Value Network Update:

- For each group, a fixed number of training iterations are performed.
- In each iteration, multiple mini-batches are sampled from the group's replay buffer.
- For every mini-batch:

- The PPO loss is computed, including contributions from the policy objective, entropy regularization, and value loss.
- Backpropagation is applied, and gradient clipping may be used to improve training stability.
- Each group's optimizer updates its model parameters and clears gradients afterward.

5. Logging:

- At the end of each training iteration, the mean episode reward is computed separately for each group.
- Rewards are selected using the "done" flag to ensure only completed episodes are evaluated.
- These reward values are stored for later visualization and are displayed live via a progress bar for real-time feedback.

```
def process_batch(batch):
    """
    If the `(group, "terminated")` and `(group, "done")` keys are not present, create them by ex
    `"terminated"` and `"done"`.
    This is needed to present them with the same shape as the reward to the loss.
    """
    for group in env.group_map.keys():
        keys = list(batch.keys(True, True))
        group_shape = batch.get_item_shape(group)
        nested_done_key = ("next", group, "done")
        nested_terminated_key = ("next", group, "terminated")
        if nested_done_key not in keys:
            batch.set(
                nested_done_key,
                batch.get(("next", "done")).unsqueeze(-1).expand((*group_shape, 1)),
            )
        if nested_terminated_key not in keys:
            batch.set(
                nested_terminated_key,
                batch.get(("next", "terminated"))
                .unsqueeze(-1)
                .expand((*group_shape, 1)),
            )
    return batch

def collect_data_and_compute_gae(batch, env, gaes, losses, replay_buffers):
    current_frames = batch.numel()
    batch = process_batch(batch)

    for group in env.group_map.keys():
        with torch.no_grad():
            gaes[group](
                batch,
                params=losses[group].critic_network_params,
                target_params=losses[group].target_critic_network_params,
            )

    data_view = batch.reshape(-1)
    replay_buffers[group].extend(data_view)

    return batch, current_frames

def optimize_all_groups(env, replay_buffers, losses, optimizers, n_iters, frames_per_batch, mini
```

```

        loss_value = (
            loss_vals["loss_objective"]
            + loss_vals["loss_entropy"]
            + loss_vals["loss_critic"]
        )

        loss_value.backward()
        torch.nn.utils.clip_grad_norm_(losses[group].parameters(), 1.0)
        optimizers[group].step()
        optimizers[group].zero_grad()

pbar = tqdm(
    total=n_iters,
    desc=", ".join(
        [f"episode_reward_mean_{group} = 0" for group in env.group_map.keys()]
    ),
)
episode_reward_mean_map = {group: [] for group in env.group_map.keys()}

for iteration, batch in enumerate(collector):
    batch, current_frames = collect_data_and_compute_gae(batch, env, gaes, losses, replay_buffer

    optimize_all_groups(env, replay_buffers, losses, optimizers, n_iters, frames_per_batch, mini

# Logging
for group in env.group_map.keys():
    episode_reward_mean = (
        batch.get(("next", group, "episode_reward"))[
            batch.get(("next", group, "done"))
        ]
        .mean()
        .item()
    )
    episode_reward_mean_map[group].append(episode_reward_mean)

pbar.set_description(
    ", ".join(
        [
            f"episode_reward_mean_{group} = {episode_reward_mean_map[group][-1]}"
            for group in env.group_map.keys()
        ]
    ),
    refresh=False,
)
pbar.update()

➡ episode_reward_mean_speaker = -628.1336669921875, episode_reward_mean_listener = -628.1336669

```



✓ 12.2.1 Results

```

plt.plot(episode_reward_mean_map["listener"], label="Listener")
plt.xlabel("Training iterations")
plt.ylabel("Reward")
plt.title("Episode reward mean")
plt.show()

```



```
with torch.no_grad():
    env.rollout(
        max_steps=1000,
        policy=policies,
        callback=lambda env, _: env.render(),
        auto_cast_to_device=True,
        break_when_any_done=False,
    )
```

✓ References

*This implementation is heavily based on the code from **Matteo Bettini**:*

- [MULTI-AGENT REINFORCEMENT LEARNING \(PPO\) WITH TORCHRL TUTORIAL](#)
- [COMPETITIVE MULTI-AGENT REINFORCEMENT LEARNING \(DDPG\) WITH TORCHRL TUTORIAL](#)

Licensed under [CC BY-NC-ND 4.0](#). © Zoltán Barta, 2025.

First Assignment

Collective Intelligence

Agent-Based Modeling

2025

Task Description

This assignment requires you to create a **custom NetLogo model** based on a social, biological, or physical phenomenon that interests you or that you wish to explore further. Please read the requirements below carefully to complete the assignment successfully.

1. Objective:

- Develop a unique NetLogo model to simulate a phenomenon of your choice.
- You may use pre-defined models in the NetLogo library for inspiration, but you must **not** use them as a solution.

2. Model Requirements:

- The environment must be a **2D grid map** with horizontal and vertical wrapping enabled.
- **Map size:** Choose between 20×20 and 128×128 .
- The interface must include:
 - At least 5 adjustable hyperparameters (e.g., Sliders, Switches, Choosers, Inputs).
 - Buttons for **go**, **go-once**, and **setup**.
 - At least 3 reporters (via Monitors or Plots) to display model data.
- Your code should incorporate:
 - Agent breeds.
 - At least 3 agent attributes and 3 global variables.
 - Helper functions to improve code readability and structure.
- Provide minimal documentation following the markdown format in the NetLogo Info Tab (Info -> Edit).

3. Experiment Requirements:

- Use the **BehaviorSpace** tool to run an experiment:
 - Vary a chosen hyperparameter across an interval where you predict a phase transition or tipping point may occur.
 - Measure 2 reporters of your choice for the experiment.



- Set repetitions to 10.
- Export experiment results to a CSV file.
- Create visualizations (plots) of the experiment results.

PowerPoint Presentation

While presenting your work is not mandatory, **not presenting will limit your maximum grade to 3**. If you choose to present, please follow these guidelines:

1. **Duration:** The presentation should last 5–6 minutes and include approximately 5–6 slides.
2. **Content:**
 - **Introduction:** Explain your model idea and the motivation for your topic.
 - **Implementation Details:** Highlight key elements of your code, including interface elements (buttons, hyperparameters) and design decisions.
 - **Demonstration:** Include a GIF or short video of your model in action.
 - **Experiment Results:** Present additional runs, experiments, and dynamic changes in your model. Show plots from BehaviorSpace results and explain their significance.
3. **Submission Requirements:**
 - Save all work into a single `.nlogo` file.
 - Include the BehaviorSpace configuration file (`.xml`).
 - Convert your PowerPoint presentation into a PDF and include it in your submission.

Assignment Submission and General Rules

- **Submission Files:**
 - `.nlogo` file (model)
 - `.xml` file (BehaviorSpace experiment configuration)
 - PowerPoint presentation in `.pdf` format
 - Submit a **zipped file** containing the `.nlogo`, `.xml`, and converted `.pdf` presentation to Canvas.
- **Deadline:** March 12th, Wednesday 11:59 PM (strict, no late submission)
- **Important Notes:**
 - Copying others' code results in automatic failure (grade 0)
 - Not submitting anything results in a grade of 0
 - Submitting something, as long as it is not an empty NetLogo project, might result in a minimum grade of 1



By completing this assignment, you will enhance your understanding of NetLogo, gain hands-on experience in modeling complex systems, and improve your analytical skills through experimentation and visualization.

Prepared by: Tamás Takács

Date: 2025

Licensed under [CC BY-NC-ND 4.0](#). © Tamás Takács, 2025.

Mechanism Design

Collective Intelligence

Multi-Agent Reinforcement Learning

2025



Task Description

This assignment focuses on implementing a **Multi-Agent Proximal Policy Optimization (MAPPO)** algorithm using [TorchRL](#) within a dynamic, graph-based MARL environment. Based on the famous



board game *Scotland Yard*, agents operate in asymmetric roles—one as a **target** (**Mr. X**) and others as **cooperative pursuers** (**Policemen**).

Project GitHub Link: <https://github.com/elte-collective-intelligence/Mechanism-Design>

The current project utilizes:

- A **custom graph environment** implemented in the PettingZoo AEC format.
- A centralized value function and shared policy (IDQN) for Policemen.
- A dynamic adversary (Mr. X), controlled by DQN or PPO.
- Meta-learning capabilities that adjust difficulty during training.
- Experimental Graph Neural Networks (GNNs) to process local agent neighborhoods and guide movement strategies.
- **Stable-Baselines3** for initial training, with TensorBoard + WandB logging and trajectory visualizations.

The new assignment goal is to replace the Policemen's current IDQN policy with a **MAPPO-based TorchRL implementation**, supporting Centralized Training with Decentralized Execution (CTDE). Policemen must learn to coordinate efficiently on the graph using only local views, while Mr. X attempts to evade detection for as long as possible. [\[CTDE Reference\]](#)

While the final goal is to use TorchRL's native environment structure (**EnvBase**, **TensorDict**, etc.), you may initially use PettingZoo environments with the official PettingZooWrapper provided by TorchRL, if helpful for bootstrapping.

Example: Consider a team of police drones tracking a stealthy target in an urban environment. Each drone has local neighborhood information but no global view. By learning together during training but acting independently at test time, they must corner and capture the intruder with optimal coverage.

Environment Phases

1. **Spawn Phase:** All agents are randomly placed on graph nodes.
2. **Pursuit Phase:** Policemen must collaborate to locate and catch Mr. X using local views.
3. **Evasion Phase (Mr. X):** Uses stealthy navigation to maximize survival time.

Assignment Directions

You may choose between the following development directions for this assignment:

- **Option 1: Incremental Migration**

Maintain the current implementation based on Stable-Baselines3, PettingZoo, and Supersuit, and gradually migrate to TorchRL. This involves:

- Converting the current environment to TorchRL's **EnvBase** format.
- Replacing IDQN with MAPPO using TorchRL's PPO implementation.
- Retaining logging (TensorBoard + WandB), curriculum integration, and Docker support.
- Adding structured unit tests, visualization tools, and configuration logic.

- **Option 2: Reimplementation Using Native TorchRL**

Build a new version of the environment directly with TorchRL-native components, including:

- MAPPO-based CTDE pipeline.
- GNN-encoded local observations.
- Fully decentralized execution logic.
- Structured logging, evaluation, and visualization tools.

Elements to Preserve:

- **MAPPO-style PPO Algorithm:** Centralized critic, shared policy for Policemen, trained with PPO-Clip. [\[PPO-Clip\]](#)
- Map topology represented as a graph (nodes = positions, edges = moves).
- **Adversarial Setup:** Separate policy for Mr. X (e.g., DQN or PPO).

Elements to Improve or Redesign:

- **GNN-Based State Encoding.** [\[Graph Spaces\]](#)
- Encourage coverage, triangulation, and proximity coordination. Penalize isolated movement or redundant overlaps.
- **Curriculum Learning:** Gradually increase graph size, node degrees, and evasion intelligence. [\[Curriculum Learning\]](#)
- **Evaluation Metrics:**
 - Capture rate
 - Policemen clustering entropy
 - Average distance to Mr. X
 - Coverage heatmaps

A Possible Structured Plan for Reimplementation Using Native TorchRL

0. Possible Directory Structure

```
marl-task/  
|-- configs/  
|   |-- base.yaml  
|   |-- env/  
|   |-- algo/  
|   |-- agent/  
|   '-- experiment/  
|-- docker/  
|   |-- Dockerfile  
|   '-- entrypoint.sh  
|-- logs/
```

```
|-- outputs/
|-- models/
|   '-- ppo/
|-- src/
|   |-- envs/
|   |-- agents/
|   |-- rollout/
|   '-- main.py
|-- test/
|   |-- test_env.py
|   '-- test_metrics.py
|-- .gitignore
|-- requirements.txt
|-- README.md
'-- LICENSE
```

1. Environment Setup

Define a Custom TorchRL-Compatible Environment

Create a class `Env(EnvBase)` in `src/envs/env.py` with the following methods:

- `reset(self) -> TensorDict`
- `step(self, actions: TensorDict) -> TensorDict`

Define:

- `observation_spec`
- `action_spec`
- `reward_spec`
- `done_spec`

Ensure all I/O uses `TensorDict`. Observations should be partial and relative, including distance to the shape center and nearest neighbor. Use `torchrl.envs.transforms` for normalization or preprocessing.

Optional: PettingZoo Wrapper

Use `PettingZooWrapper` from `torchrl.envs.libs.pettingzoo` if adapting from existing environments:

Listing 1: PettingZoo Wrapper Example

```
from torchrl.envs.libs.pettingzoo import PettingZooWrapper
wrapped_env = PettingZooWrapper(pettingzoo_env)
```

2. Agent and Model Definition

Define Policy and Critic Modules

In `src/agents/ppo_agent.py`, implement:

A shared `TensorDictModule` policy:

Listing 2: Shared Policy

```
policy = TensorDictModule(network, in_keys=[...], out_keys=["action"])
```

A centralized critic using ValueOperator:

Listing 3: Centralized Critic

```
critic = ValueOperator(critic_network, in_keys=[...])
```

This supports the CTDE paradigm: centralized critic with decentralized policy execution.

3. PPO Training Setup

Collector Configuration

Use SyncDataCollector or MultiSyncDataCollector:

Listing 4: Collector Configuration

```
collector = SyncDataCollector(  
    create_env_fn=env_fn,  
    policy=policy,  
    frames_per_batch=2048,  
    total_frames=...  
)
```

Loss Function

Use ClipPPOLoss:

Listing 5: PPO Loss Module

```
loss_module = ClipPPOLoss(  
    actor=policy,  
    critic=critic,  
    clip_epsilon=0.2,  
    entropy_coef=0.01  
)
```

4. Training Loop

Training in main.py

Set up the training loop using collector, replay_buffer, loss_module, and optimizer:

Listing 6: Training Loop

```
for batch in collector:  
    for _ in range(ppo_epochs):  
        loss = loss_module(batch)  
        loss.backward()  
        optimizer.step()  
        optimizer.zero_grad()
```

5. Evaluation and Logging

Logging

Use TensorBoard or W&B:

Listing 7: TensorBoard Logging

```
from torch.utils.tensorboard import SummaryWriter  
writer = SummaryWriter(log_dir=...)  
writer.add_scalar("reward/mean", mean_reward, step)
```

Evaluation

Run trained policies with local observations only (CTDE) and export GIFs using `pygame`, `matplotlib`, or `imageio`. Store results in `outputs/`.

6. Configuration Management

Hydra Integration

Use Hydra or structured YAML configs in `configs/`:

- `configs/env/task.yaml`
- `configs/algo/ppo.yaml`
- `configs/experiment/sweep.yaml`

Launch with:

Listing 8: Launch Command

```
python src/main.py +experiment=task +algo=ppo
```

7. Testing

Unit Tests

Place tests in `test/`:

Listing 9: Unit Test Example

```
def test_env_reset():
    env = Env(...)
    td = env.reset()
    assert "observation" in td
```

8. CTDE Framework Details

- The shared policy is trained with access to a centralized value function.
- Execution uses only local observations per agent.
- During inference, policies should operate without access to the global state or other agents' observations.
- Ensure the actor's input keys are restricted to local observations, while the critic receives richer information.

9. Docker for Reproducibility

Add Docker Support

Create a `docker/` folder with the following:

Dockerfile:

Listing 10: Dockerfile Example

```
FROM python:3.12-slim
WORKDIR /app
COPY . /app
RUN pip install -r requirements.txt
CMD ["python", "src/main.py"]
```

entrypoint.sh (optional launcher script)
Build and run:

Listing 11: Docker Build and Run

```
docker build -t marl-task .
docker run --rm marl-task
```

PowerPoint Presentation

While presenting your work is not mandatory, **not presenting will limit your maximum grade to 3**. The presentation serves as a concise overview of your project.

Duration

Aim for a few well-organized slides that complement your documentation.

Suggested Structure

1. **Title & Objective:** Brief objective and project direction.
2. **System Architecture:** High-level overview (environment, agent setup, training loop).
3. **Environment & Task Setup:** Describe environment, agent logic, and dynamics.
4. **Key Design Choices:** Reward shaping, curriculum, metrics, logging.
5. **Results & Visualizations:** GIFs, reward curves, training plots, insights.
6. **Conclusion & Future Work:** Key takeaways.

Important Notes

The core of your submission is your documentation and code, which will be the primary basis for grading. The presentation is your opportunity to highlight contributions and insights.

Assignment Submission and General Rules

- All development must be carried out within a **GitHub repository**.
- For teams:
 - Collaboration strategy (e.g., shared/individual branches) is up to you.
 - **Task division must be clearly defined** in the project's README.



- For individuals: use a separate branch in the repository.
 - Submit a single ZIP file to Canvas with:
 - The entire project repository (excluding large model files/checkpoints).
 - The presentation in PDF format.
 - Collaboration is highly encouraged; this is a large-scale assignment.
-

Prepared by: Zoltán Barta

Date: 2025

Licensed under [CC BY-NC-ND 4.0](#), © Zoltán Barta, 2025.

Cooperative Mingle

Collective Intelligence

Multi-Agent Reinforcement Learning

2025



Task Description

This assignment focuses on implementing a **Multi-Agent Reinforcement Learning (MARL)** system using [TorchRL](#). Inspired by the series *Squid Game*, this project requires agents to learn coordinated decision-making and spatial negotiation in a competitive-cooperative setting.

Agents begin each episode standing on a rotating central platform. Once the platform stops spinning, agents must quickly and cooperatively navigate toward rooms positioned around the arena, each with limited capacity. Their goal is to fill the rooms fairly and efficiently, avoiding overcrowding or exclusion, which results in penalties or failure.

Project GitHub Link:

<https://github.com/elte-collective-intelligence/student-particle-swarm-optimization>

At this stage, the project is in its early conceptual phase, no implementation or starter code has been developed yet. The idea remains exploratory and has not been successfully realized in previous semesters. It represents an original contribution within the context of the class.

This task is best addressed using the **Centralized Training with Decentralized Execution (CTDE)** paradigm. During training, agents can access global critic information to stabilize learning, but during evaluation, each agent must act independently based on partial, local observations. See: <https://arxiv.org/abs/2409.03052>

While the final goal is to use TorchRL’s native environment structure (`EnvBase`, `TensorDict`, etc.), you may initially use PettingZoo environments with the official `PettingZooWrapper` provided by TorchRL, if helpful for bootstrapping.

Environment Phases

1. At the start of each episode, agents are spun into randomized positions on a central circular arena. During this **rotation phase**, agents must observe others, explore the map, and begin to organize themselves into informal teams. These teams are not predefined—agents must learn heuristics or strategies to determine who to align with, how to split up, and what areas to claim.
2. Once the spinning stops, the arena suddenly reveals a fixed set of rooms around its perimeter—each with limited capacity (e.g., 3 agents max). Agents must quickly and fairly occupy these rooms, avoiding overfilling, collisions, or being left out. Rooms are claimed on a first-come, first-serve basis.

This environment is designed to study spontaneous cooperation, social grouping, and negotiation-like behavior.

Assignment Directions

You may choose between the following development directions for this assignment:

- **Option 1: Incremental Migration (not available for Cooperative Mingle)**
Maintain the current implementation based on Stable-Baselines3 (SB3), Supersuit, and PettingZoo, and gradually migrate the system to TorchRL-compatible components while preserving existing functionality. This includes integration of configuration management (e.g., `Hydra` or `YAML`), logging via `Weights & Biases` and `TensorBoard`, `Docker` for reproducibility, structured unit testing, visualization, and a clear README. (*Option not available for this task.*)
- **Option 2: Reimplementation Using Native TorchRL**
Build the project from scratch using TorchRL’s native APIs. Instead of using PettingZoo, start from a TorchRL-compatible environment or adapt an existing one. Design the training pipeline, agent logic, and evaluation entirely within the TorchRL framework. Include CTDE, configuration management, Docker, logging, visualization, testing, and documentation.

Elements to Consider

- Utilize Proximal Policy Optimization, specifically the clipped variant (`PPO-Clip`), as the core learning algorithm. Optionally experiment with `MADDPG`, `QMIX`, or `VDN`.
- Custom Environment: Rotating phase, timed transition, discrete rooms, room capacities, collisions, overshooting penalties.
- `Curriculum Learning`: Start with fewer agents and larger rooms, then increase difficulty.
- `Shape the reward` function to promote desirable behaviors such as occupancy and coordination.
- Design and track custom metrics: room fill rate, completion time, collisions, idle agents.

A Possible Structured Plan for Implementation Using Native TorchRL

0. Possible Directory Structure

```
marl-task/  
|-- configs/  
|   |-- base.yaml  
|   |-- env/  
|   |-- algo/  
|   |-- agent/  
|   '-- experiment/  
|-- docker/  
|   |-- Dockerfile  
|   '-- entrypoint.sh  
|-- logs/  
|-- outputs/  
|-- models/  
|   '-- ppo/  
|-- src/  
|   |-- envs/  
|   |-- agents/  
|   |-- rollout/  
|   '-- main.py  
|-- test/  
|   |-- test_env.py  
|   '-- test_metrics.py  
|-- .gitignore  
|-- requirements.txt  
|-- README.md  
'-- LICENSE
```

A Possible Structured Plan for Implementation Using Native TorchRL

1. Environment Setup

Define a Custom TorchRL-Compatible Environment

Create a class `Env(EnvBase)` in `src/envs/env.py` with the following methods:

- `reset(self) -> TensorDict`
- `step(self, actions: TensorDict) -> TensorDict`

Define:

- `observation_spec`
- `action_spec`
- `reward_spec`
- `done_spec`

Ensure all I/O uses `TensorDict`. Observations should be partial and relative, including distance to the shape center and nearest neighbor. Use `torchrl.envs.transforms` for normalization or preprocessing.

Optional: PettingZoo Wrapper

Use `PettingZooWrapper` from `torchrl.envs.libs.pettingzoo` if adapting from existing environments:

Listing 1: PettingZoo Wrapper Example

```
from torchrl.envs.libs.pettingzoo import PettingZooWrapper
wrapped_env = PettingZooWrapper(pettingzoo_env)
```

2. Agent and Model Definition

Define Policy and Critic Modules

In `src/agents/ppo_agent.py`, implement:

A shared `TensorDictModule` policy:

Listing 2: Shared Policy

```
policy = TensorDictModule(network, in_keys=[...], out_keys=["action"])
```

A centralized critic using `ValueOperator`:

Listing 3: Centralized Critic

```
critic = ValueOperator(critic_network, in_keys=[...])
```

This supports the CTDE paradigm: centralized critic with decentralized policy execution.

3. PPO Training Setup

Collector Configuration

Use `SyncDataCollector` or `MultiSyncDataCollector`:

Listing 4: Collector Configuration

```
collector = SyncDataCollector(
    create_env_fn=env_fn,
    policy=policy,
    frames_per_batch=2048,
    total_frames=...
)
```

Loss Function

Use `ClipPPOLoss`:

Listing 5: PPO Loss Module

```
loss_module = ClipPPOLoss(
    actor=policy,
    critic=critic,
    clip_epsilon=0.2,
    entropy_coef=0.01
)
```

4. Training Loop

Training in main.py

Set up the training loop using `collector`, `replay_buffer`, `loss_module`, and `optimizer`:

Listing 6: Training Loop

```
for batch in collector:
    for _ in range(ppo_epochs):
        loss = loss_module(batch)
        loss.backward()
        optimizer.step()
        optimizer.zero_grad()
```

5. Evaluation and Logging

Logging

Use TensorBoard or W&B:

Listing 7: TensorBoard Logging

```
from torch.utils.tensorboard import SummaryWriter
writer = SummaryWriter(log_dir=...)
writer.add_scalar("reward/mean", mean_reward, step)
```

Evaluation

Run trained policies with local observations only (CTDE) and export GIFs using `pygame`, `matplotlib`, or `imageio`. Store results in `outputs/`.

6. Configuration Management

Hydra Integration

Use Hydra or structured YAML configs in `configs/`:

- `configs/env/task.yaml`
- `configs/algo/ppo.yaml`
- `configs/experiment/sweep.yaml`

Launch with:

Listing 8: Launch Command

```
python src/main.py +experiment=task +algo=ppo
```

7. Testing

Unit Tests

Place tests in `test/`:

Listing 9: Unit Test Example

```
def test_env_reset():
    env = Env(...)
    td = env.reset()
    assert "observation" in td
```

8. CTDE Framework Details

- The shared policy is trained with access to a centralized value function.
- Execution uses only local observations per agent.
- During inference, policies should operate without access to the global state or other agents' observations.
- Ensure the actor's input keys are restricted to local observations, while the critic receives richer information.

9. Docker for Reproducibility

Add Docker Support

Create a `docker/` folder with the following:

Dockerfile:

Listing 10: Dockerfile Example

```
FROM python:3.12-slim
WORKDIR /app
COPY . /app
RUN pip install -r requirements.txt
CMD ["python", "src/main.py"]
```

`entrypoint.sh` (optional launcher script)

Build and run:

Listing 11: Docker Build and Run

```
docker build -t marl-task .
docker run --rm marl-task
```

PowerPoint Presentation

While presenting your work is not mandatory, **not presenting will limit your maximum grade to 3**. The presentation serves as a concise overview of your project.

Duration

Aim for a few well-organized slides that complement your documentation.

Suggested Structure

1. **Title & Objective:** Brief objective and project direction.
2. **System Architecture:** High-level overview (environment, agent setup, training loop).
3. **Environment & Task Setup:** Describe environment, agent logic, and dynamics.
4. **Key Design Choices:** Reward shaping, curriculum, metrics, logging.
5. **Results & Visualizations:** GIFs, reward curves, training plots, insights.
6. **Conclusion & Future Work:** Key takeaways.



Important Notes

The core of your submission is your documentation and code, which will be the primary basis for grading. The presentation is your opportunity to highlight contributions and insights.

Assignment Submission and General Rules

- All development must be carried out within a **GitHub repository**.
- For teams:
 - Collaboration strategy (e.g., shared/individual branches) is up to you.
 - **Task division must be clearly defined** in the project's README.
- For individuals: use a separate branch in the repository.
- Submit a single ZIP file to Canvas with:
 - The entire project repository (excluding large model files/checkpoints).
 - The presentation in PDF format.
- Collaboration is highly encouraged; this is a large-scale assignment.

Prepared by: Tamás Takács

Date: 2025

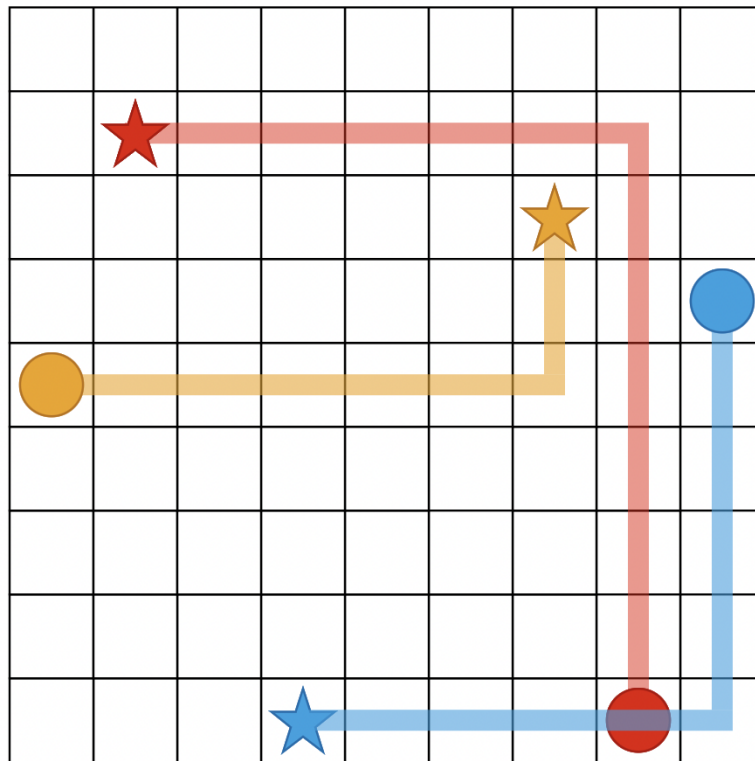
Licensed under [CC BY-NC-ND 4.0](#). © Tamás Takács, 2025.

Pathfinding

Collective Intelligence

Multi-Agent Reinforcement Learning

2025



Task Description

This assignment focuses on implementing a **Multi-Agent Reinforcement Learning (MARL)** system using [TorchRL](#), where agents must collaboratively solve multi-agent pathfinding tasks in complex, obstacle-rich environments. This assignment explores adaptive coordination, where agents must navigate dynamically generated maps, avoid hazards, and reach their targets in cooperation with teammates. These environments mimic real-world challenges such as robot swarm navigation, emergency evacuation, or warehouse logistics, where route planning is made harder by changing layouts, moving goals, and the presence of other agents.

Project GitHub Link: <https://github.com/elte-collective-intelligence/student-pathfinding>
(highly recommend MG's work)

The current project versions utilize:

- A PettingZoo AEC (Agent Environment Cycle) environment, customized to simulate 2D pathfinding with discrete grid or continuous motion. [\[AEC API\]](#)
- The Stable-Baselines3 PPO algorithm for training shared policies. [\[SB3 PPO\]](#)
- Real-time visual rendering using `pygame` and post-processed visualizations.
- Centralized Training and Centralized Execution model. A single shared policy is trained alongside a centralized value function that has access to shared information across agents. During evaluation, this same centralized policy is used by all agents, meaning that each agent's behavior is determined by a common model, rather than acting independently based on purely local observations.
- A simple agent evaluation framework. (SB3 with TensorBoard Integration)

Agents must navigate from their spawn points to assigned targets while avoiding obstacles, minimizing path length, and coordinating to avoid collisions or deadlocks. The environment may be static or dynamic, with changing obstacle layouts or moving goals. This setting requires cooperative behavior, best addressed with a Centralized Training with Decentralized Execution (CTDE) approach, where policies are trained with access to global critic information but executed independently by each agent. [\[CTDE Reference\]](#)

While the final goal is to use TorchRL's native environment structure (`EnvBase`, `TensorDict`, etc.), you may initially use PettingZoo environments with the official `PettingZooWrapper` provided by TorchRL, if helpful for bootstrapping.

Environment Phases

1. **Planning Phase (Warm-Up):** Agents can scan the environment or explore without penalties.
2. **Execution Phase:** A timer starts and agents are evaluated based on path efficiency, collaboration, and avoidance of congestion.

Assignment Directions

You may choose between the following development directions for this assignment:

- **Option 1: Incremental Migration**

Maintain the current implementation based on Stable-Baselines3 (SB3), Supersuit, and PettingZoo, and gradually migrate the system to TorchRL. This approach involves adapting the environment and training loop to TorchRL-compatible components while preserving existing functionality. The migration should also include:

- Integration of a configuration management system (e.g., [Hydra](#) or structured YAML)
- Preservation of logging via both [Weights & Biases](#) (WandB) and TensorBoard
- [Docker](#) for reproducibility and cross-platform compatibility
- Structured unit testing (at least 2 components)
- Visualization outputs (e.g., GIFs, performance plots)

- A clear and well-maintained `README.md` with setup and usage instructions

- **Option 2: Reimplementation Using Native TorchRL**

Build the project from scratch using TorchRL's native APIs. Instead of using PettingZoo, start from a TorchRL-compatible environment (e.g., based on `EnvBase`) or adapt an existing one. Design the training pipeline, agent interaction logic, and evaluation procedures entirely within the TorchRL framework. As with the first option, the final solution should support:

- Centralized Training with Decentralized Execution (CTDE)
- Configuration management
- Docker deployment
- WandB/TensorBoard logging
- Visualization and reproducibility tools
- Testing and documentation

Elements to Preserve:

- PPO Algorithm: Continue using Proximal Policy Optimization, specifically the clipped variant (PPO-Clip), as the core learning algorithm. `[PPO-Clip]` (Optionally, you could experiment with `[MADDPG]`, `[QMIX]`, `[VDN]`)
- MPE Environment (Optional): The Multi-Agent Particle Environment (MPE) can be retained, though you are also encouraged to consider reimplementing a simplified particle-based environment using native TorchRL. `[PettingZoo MPE]`
- Core Objective: The primary task remains, agents must collaboratively solve multi-agent pathfinding tasks in a complex, obstacle-rich environment.
- Multi-Agent Setting

Elements to Improve or Redesign:

- Environmental Complexity: Introduce walls, traps, moving hazards, or multi-room maps with choke points to increase coordination complexity.
- Curriculum Learning: Start with simple mazes or open maps, then gradually increase complexity (e.g., tighter corridors, dynamic targets, agent congestion). `[Curriculum Learning]`
- Reward Design: Develop a reward function that balances path efficiency, obstacle avoidance, goal completion, and collaborative movement (e.g., penalizing blocking teammates or deadlocks). `[Reward Shaping]`
- Your system should encourage behaviors like traffic yielding, lane forming, or goal re-routing when paths are blocked.
- Evaluation Metrics: Add custom metrics for training and evaluation, such as:
 - Average path length
 - Success rate (% agents reaching their goal)
 - Collision rate
 - Completion time variance
 - Congestion/delay penalties

A Possible Structured Plan for Implementation Using Native TorchRL

0. Possible Directory Structure

```
marl-task/  
|-- configs/  
|   |-- base.yaml  
|   |-- env/  
|   |-- algo/  
|   |-- agent/  
|   '-- experiment/  
|-- docker/  
|   |-- Dockerfile  
|   '-- entrypoint.sh  
|-- logs/  
|-- outputs/  
|-- models/  
|   '-- ppo/  
|-- src/  
|   |-- envs/  
|   |-- agents/  
|   |-- rollout/  
|   '-- main.py  
|-- test/  
|   |-- test_env.py  
|   '-- test_metrics.py  
|-- .gitignore  
|-- requirements.txt  
|-- README.md  
'-- LICENSE
```

1. Environment Setup

Define a Custom TorchRL-Compatible Environment

Create a class `Env(EnvBase)` in `src/envs/env.py` with the following methods:

- `reset(self) -> TensorDict`
- `step(self, actions: TensorDict) -> TensorDict`

Define:

- `observation_spec`
- `action_spec`
- `reward_spec`
- `done_spec`

Ensure all I/O uses `TensorDict`. Observations should be partial and relative, including distance to the shape center and nearest neighbor. Use `torchrl.envs.transforms` for normalization or preprocessing.

Optional: PettingZoo Wrapper

Use `PettingZooWrapper` from `torchrl.envs.libs.pettingzoo` if adapting from existing environments:

Listing 1: PettingZoo Wrapper Example

```
from torchrl.envs.libs.pettingzoo import PettingZooWrapper
wrapped_env = PettingZooWrapper(pettingzoo_env)
```

2. Agent and Model Definition

Define Policy and Critic Modules

In `src/agents/ppo_agent.py`, implement:

A shared `TensorDictModule` policy:

Listing 2: Shared Policy

```
policy = TensorDictModule(network, in_keys=[...], out_keys=["action"])
```

A centralized critic using `ValueOperator`:

Listing 3: Centralized Critic

```
critic = ValueOperator(critic_network, in_keys=[...])
```

This supports the CTDE paradigm: centralized critic with decentralized policy execution.

3. PPO Training Setup

Collector Configuration

Use `SyncDataCollector` or `MultiSyncDataCollector`:

Listing 4: Collector Configuration

```
collector = SyncDataCollector(
    create_env_fn=env_fn,
    policy=policy,
    frames_per_batch=2048,
    total_frames=...
)
```

Loss Function

Use `ClipPPOLoss`:

Listing 5: PPO Loss Module

```
loss_module = ClipPPOLoss(
    actor=policy,
    critic=critic,
    clip_epsilon=0.2,
    entropy_coef=0.01
)
```

4. Training Loop

Training in main.py

Set up the training loop using `collector`, `replay_buffer`, `loss_module`, and `optimizer`:

Listing 6: Training Loop

```
for batch in collector:
    for _ in range(ppo_epochs):
        loss = loss_module(batch)
        loss.backward()
        optimizer.step()
        optimizer.zero_grad()
```

5. Evaluation and Logging

Logging

Use TensorBoard or W&B:

Listing 7: TensorBoard Logging

```
from torch.utils.tensorboard import SummaryWriter
writer = SummaryWriter(log_dir=...)
writer.add_scalar("reward/mean", mean_reward, step)
```

Evaluation

Run trained policies with local observations only (CTDE) and export GIFs using `pygame`, `matplotlib`, or `imageio`. Store results in `outputs/`.

6. Configuration Management

Hydra Integration

Use Hydra or structured YAML configs in `configs/`:

- `configs/env/task.yaml`
- `configs/algo/ppo.yaml`
- `configs/experiment/sweep.yaml`

Launch with:

Listing 8: Launch Command

```
python src/main.py +experiment=task +algo=ppo
```

7. Testing

Unit Tests

Place tests in `test/`:

Listing 9: Unit Test Example

```
def test_env_reset():
    env = Env(...)
    td = env.reset()
    assert "observation" in td
```

8. CTDE Framework Details

- The shared policy is trained with access to a centralized value function.
- Execution uses only local observations per agent.
- During inference, policies should operate without access to the global state or other agents' observations.
- Ensure the actor's input keys are restricted to local observations, while the critic receives richer information.

9. Docker for Reproducibility

Add Docker Support

Create a `docker/` folder with the following:

Dockerfile:

Listing 10: Dockerfile Example

```
FROM python:3.12-slim
WORKDIR /app
COPY . /app
RUN pip install -r requirements.txt
CMD ["python", "src/main.py"]
```

`entrypoint.sh` (optional launcher script)

Build and run:

Listing 11: Docker Build and Run

```
docker build -t marl-task .
docker run --rm marl-task
```

PowerPoint Presentation

While presenting your work is not mandatory, **not presenting will limit your maximum grade to 3**. The presentation serves as a concise overview of your project.

Duration

Aim for a few well-organized slides that complement your documentation.

Suggested Structure

1. **Title & Objective:** Brief objective and project direction.
2. **System Architecture:** High-level overview (environment, agent setup, training loop).
3. **Environment & Task Setup:** Describe environment, agent logic, and dynamics.
4. **Key Design Choices:** Reward shaping, curriculum, metrics, logging.
5. **Results & Visualizations:** GIFs, reward curves, training plots, insights.
6. **Conclusion & Future Work:** Key takeaways.



Important Notes

The core of your submission is your documentation and code, which will be the primary basis for grading. The presentation is your opportunity to highlight contributions and insights.

Assignment Submission and General Rules

- All development must be carried out within a **GitHub repository**.
- For teams:
 - Collaboration strategy (e.g., shared/individual branches) is up to you.
 - **Task division must be clearly defined** in the project's README.
- For individuals: use a separate branch in the repository.
- Submit a single ZIP file to Canvas with:
 - The entire project repository (excluding large model files/checkpoints).
 - The presentation in PDF format.
- Collaboration is highly encouraged; this is a large-scale assignment.

Prepared by: Tamás Takács

Date: 2025

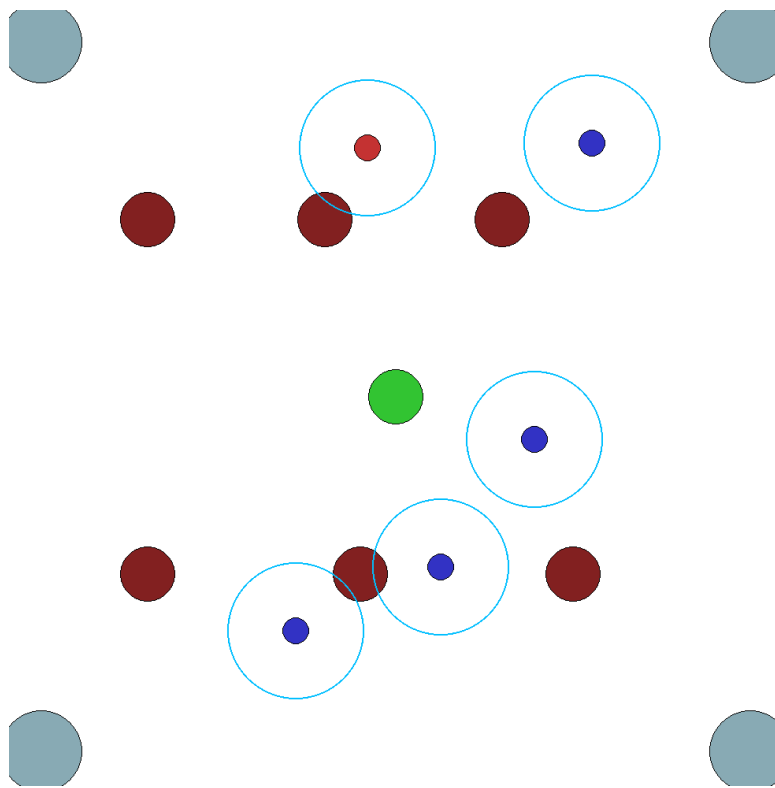
Licensed under [CC BY-NC-ND 4.0](#). © Tamás Takács, 2025.

Patrolling

Collective Intelligence

Multi-Agent Reinforcement Learning

2025



Task Description

This assignment focuses on implementing a **Multi-Agent Reinforcement Learning (MARL)** system using [TorchRL](#). In this project, autonomous drones engage in a strategic game of pursuit and evasion. A team of **patroller drones** is tasked with locating and intercepting **intruder drones**, which aim to reach a target destination without being detected. Patrollers must coordinate their behavior through indirect communication to maximize detection success, while intruders use stealth and strategy to evade them.

The patrolling agents must learn to coordinate movement and coverage based on partial observations and indirect cooperation. Intruders, on the other hand, act as stealthy adversaries seeking to bypass

detection. Both teams operate under realistic observation constraints. This task is best approached using the **Centralized Training with Decentralized Execution (CTDE)** paradigm. During training, patroller agents may access shared critic and team-level information to stabilize learning. During evaluation, however, each agent must act independently based on local observations, reflecting realistic deployment constraints. [\[CTDE Reference\]](#)

While the final goal is to use TorchRL’s native environment structure (`EnvBase`, `TensorDict`, etc.), you may initially use PettingZoo environments with the official `PettingZooWrapper` provided by TorchRL, if helpful for bootstrapping.

Example: Imagine border surveillance drones patrolling a national perimeter, while intruder drones attempt to penetrate the area. Patrollers must coordinate with minimal communication, covering blind spots and reacting to emerging threats.

Environment Phases

1. **Deployment Phase:** Patrollers and intruders are spawned randomly.
2. **Patrolling Phase:** Patrollers move to cover territory and maximize visibility.
3. **Intrusion Phase:** Intruders attempt to cross undetected while patrollers intercept.

Assignment Directions

You may choose between the following development directions for this assignment:

- **Option 1: Incremental Migration**

Maintain the current implementation based on Stable-Baselines3 (SB3), Supersuit, and PettingZoo, and gradually migrate the system to TorchRL. This approach involves adapting the environment and training loop to TorchRL-compatible components while preserving existing functionality. The migration should also include:

- Integration of a configuration management system (e.g., [Hydra](#) or structured YAML)
- Preservation of logging via both [Weights & Biases](#) (WandB) and TensorBoard
- [Docker](#) for reproducibility and cross-platform compatibility
- Structured unit testing (at least 2 components)
- Visualization outputs (e.g., GIFs, performance plots)
- A clear and well-maintained `README.md` with setup and usage instructions

- **Option 2: Reimplementation Using Native TorchRL**

Build the project from scratch using TorchRL’s native APIs. Instead of using PettingZoo, start from a TorchRL-compatible environment (e.g., based on `EnvBase`) or adapt an existing one. Design the training pipeline, agent interaction logic, and evaluation procedures entirely within the TorchRL framework. As with the first option, the final solution should support:

- Centralized Training with Decentralized Execution (CTDE)
- Configuration management
- Docker deployment
- WandB/TensorBoard logging

- Visualization and reproducibility tools
- Testing and documentation

Elements to Preserve:

- PPO Algorithm: Continue using Proximal Policy Optimization, specifically the clipped variant (PPO-Clip), as the core learning algorithm. [PPO-Clip] (Optionally, you could experiment with MADDPG, QMIX, VDN)
- MPE Environment (Optional): The Multi-Agent Particle Environment (MPE) can be retained, though you are also encouraged to consider reimplementing a simplified particle-based environment using native TorchRL. [PettingZoo MPE]
- Core Objective: The primary task remains, agents must collaboratively patrol an area to detect and intercept intruder agents based on partial local observations.
- Adversarial Multi-Agent Setting

Elements to Improve or Redesign:

- Add intruder speed boosts, directional detection cones, hiding zones, and terrain or obstacle elements that affect visibility.
- Curriculum Learning: Gradually increase difficulty by varying the number of intruders, expanding patrol regions, or increasing intruder speed. [Curriculum Learning]
- Design a reward function that promotes effective patrolling strategies, including interception efficiency, area coverage, and minimizing redundant paths. [Reward Shaping]
- Track key indicators such as:
 - Detection rate
 - Coverage entropy
 - Time to detection
 - Agent distribution balance

A Possible Structured Plan for Implementation Using Native TorchRL

0. Possible Directory Structure

```
marl-task/  
|-- configs/  
|   |-- base.yaml  
|   |-- env/  
|   |-- algo/  
|   |-- agent/  
|   '-- experiment/  
|-- docker/  
|   |-- Dockerfile  
|   '-- entrypoint.sh  
|-- logs/
```

```
|-- outputs/
|-- models/
|   '-- ppo/
|-- src/
|   |-- envs/
|   |-- agents/
|   |-- rollout/
|   '-- main.py
|-- test/
|   |-- test_env.py
|   '-- test_metrics.py
|-- .gitignore
|-- requirements.txt
|-- README.md
'-- LICENSE
```

1. Environment Setup

Define a Custom TorchRL-Compatible Environment

Create a class `Env(EnvBase)` in `src/envs/env.py` with the following methods:

- `reset(self) -> TensorDict`
- `step(self, actions: TensorDict) -> TensorDict`

Define:

- `observation_spec`
- `action_spec`
- `reward_spec`
- `done_spec`

Ensure all I/O uses `TensorDict`. Observations should be partial and relative, including distance to the shape center and nearest neighbor. Use `torchrl.envs.transforms` for normalization or preprocessing.

Optional: PettingZoo Wrapper

Use `PettingZooWrapper` from `torchrl.envs.libs.pettingzoo` if adapting from existing environments:

Listing 1: PettingZoo Wrapper Example

```
from torchrl.envs.libs.pettingzoo import PettingZooWrapper
wrapped_env = PettingZooWrapper(pettingzoo_env)
```

2. Agent and Model Definition

Define Policy and Critic Modules

In `src/agents/ppo_agent.py`, implement:

A shared `TensorDictModule` policy:

Listing 2: Shared Policy

```
policy = TensorDictModule(network, in_keys=[...], out_keys=["action"])
```

A centralized critic using ValueOperator:

Listing 3: Centralized Critic

```
critic = ValueOperator(critic_network, in_keys=[...])
```

This supports the CTDE paradigm: centralized critic with decentralized policy execution.

3. PPO Training Setup

Collector Configuration

Use SyncDataCollector or MultiSyncDataCollector:

Listing 4: Collector Configuration

```
collector = SyncDataCollector(  
    create_env_fn=env_fn,  
    policy=policy,  
    frames_per_batch=2048,  
    total_frames=...  
)
```

Loss Function

Use ClipPPOLoss:

Listing 5: PPO Loss Module

```
loss_module = ClipPPOLoss(  
    actor=policy,  
    critic=critic,  
    clip_epsilon=0.2,  
    entropy_coef=0.01  
)
```

4. Training Loop

Training in main.py

Set up the training loop using collector, replay_buffer, loss_module, and optimizer:

Listing 6: Training Loop

```
for batch in collector:  
    for _ in range(ppo_epochs):  
        loss = loss_module(batch)  
        loss.backward()  
        optimizer.step()  
        optimizer.zero_grad()
```

5. Evaluation and Logging

Logging

Use TensorBoard or W&B:

Listing 7: TensorBoard Logging

```
from torch.utils.tensorboard import SummaryWriter  
writer = SummaryWriter(log_dir=...)  
writer.add_scalar("reward/mean", mean_reward, step)
```



Evaluation

Run trained policies with local observations only (CTDE) and export GIFs using `pygame`, `matplotlib`, or `imageio`. Store results in `outputs/`.

6. Configuration Management

Hydra Integration

Use Hydra or structured YAML configs in `configs/`:

- `configs/env/task.yaml`
- `configs/algo/ppo.yaml`
- `configs/experiment/sweep.yaml`

Launch with:

Listing 8: Launch Command

```
python src/main.py +experiment=task +algo=ppo
```

7. Testing

Unit Tests

Place tests in `test/`:

Listing 9: Unit Test Example

```
def test_env_reset():
    env = Env(...)
    td = env.reset()
    assert "observation" in td
```

8. CTDE Framework Details

- The shared policy is trained with access to a centralized value function.
- Execution uses only local observations per agent.
- During inference, policies should operate without access to the global state or other agents' observations.
- Ensure the actor's input keys are restricted to local observations, while the critic receives richer information.

9. Docker for Reproducibility

Add Docker Support

Create a `docker/` folder with the following:

Dockerfile:

Listing 10: Dockerfile Example

```
FROM python:3.12-slim
WORKDIR /app
COPY . /app
RUN pip install -r requirements.txt
CMD ["python", "src/main.py"]
```

entrypoint.sh (optional launcher script)
Build and run:

Listing 11: Docker Build and Run

```
docker build -t marl-task .
docker run --rm marl-task
```

PowerPoint Presentation

While presenting your work is not mandatory, **not presenting will limit your maximum grade to 3**. The presentation serves as a concise overview of your project.

Duration

Aim for a few well-organized slides that complement your documentation.

Suggested Structure

1. **Title & Objective:** Brief objective and project direction.
2. **System Architecture:** High-level overview (environment, agent setup, training loop).
3. **Environment & Task Setup:** Describe environment, agent logic, and dynamics.
4. **Key Design Choices:** Reward shaping, curriculum, metrics, logging.
5. **Results & Visualizations:** GIFs, reward curves, training plots, insights.
6. **Conclusion & Future Work:** Key takeaways.

Important Notes

The core of your submission is your documentation and code, which will be the primary basis for grading. The presentation is your opportunity to highlight contributions and insights.

Assignment Submission and General Rules

- All development must be carried out within a **GitHub repository**.
- For teams:
 - Collaboration strategy (e.g., shared/individual branches) is up to you.
 - **Task division must be clearly defined** in the project's README.



- For individuals: use a separate branch in the repository.
 - Submit a single ZIP file to Canvas with:
 - The entire project repository (excluding large model files/checkpoints).
 - The presentation in PDF format.
 - Collaboration is highly encouraged; this is a large-scale assignment.
-

Prepared by: Zoltán Barta

Date: 2025

Licensed under [CC BY-NC-ND 4.0](#), © Zoltán Barta, 2025.

Formation

Collective Intelligence

Multi-Agent Reinforcement Learning

2025



Task Description

This assignment focuses on implementing a **Multi-Agent Reinforcement Learning (MARL)** system using [TorchRL](#), where agents collaboratively self-organize inside dynamically generated geometric patterns (e.g., circles, squares, convex and non-convex shapes).

Project GitHub Link: <https://github.com/elte-collective-intelligence/student-formation/tree/main>

The current project utilizes:

- A **PettingZoo AEC (Agent Environment Cycle) environment**, customized to simulate physical formations with simple agent dynamics. [\[AEC API\]](#)
- The **Stable-Baselines3 PPO algorithm** for training shared policies. [\[SB3 PPO\]](#)

- Real-time visual rendering using *pygame* and post-processed visualizations with GIF exports using *PIL* and *imageio*. [\[pygame\]](#)
- **Centralized Training and Centralized Execution** model. A single shared policy is trained alongside a centralized value function that has access to shared information across agents. During evaluation, this same centralized policy is used by all agents, meaning that each agent's behavior is determined by a common model, rather than acting independently based on purely local observations.
- A simple agent evaluation framework. (SB3 with TensorBoard Integration, SB3 with WANDB Integration)

Agents must collaboratively enter the shape and space themselves as **evenly as possible**, adapting to both static and dynamically shifting shapes. This setting requires cooperative behavior, best addressed with a **Centralized Training with Decentralized Execution (CTDE)** approach, where policies are trained with access to global critic information but executed independently by each agent. [\[CTDE Reference\]](#)

While the final goal is to use TorchRL's native environment structure (`EnvBase`, `TensorDict`, etc.), you may initially use PettingZoo environments with the official `PettingZooWrapper` provided by TorchRL, if helpful for bootstrapping.

Example: Imagine a swarm of rescue drones dispatched over a disaster zone. They must quickly position themselves into various geometric formations—grids, circles, spirals—to optimize communication coverage. The formations change based on terrain and mission phase.

Environment Phases

1. **Entry phase:** agents approach the shape.
2. **Alignment phase:** agents adjust their positions.
3. **Reconfiguration:** mid-episode shape changes.

Assignment Directions

You may choose between the following development directions for this assignment:

- **Option 1: Incremental Migration**
Maintain the current implementation based on Stable-Baselines3 (SB3), Supersuit, and PettingZoo, and gradually migrate the system to TorchRL-compatible components while preserving existing functionality. The migration should also include:
 - Integration of a configuration management system (e.g., [Hydra](#) or structured YAML)
 - Preservation of logging via both [Weights & Biases](#) (WandB) and [TensorBoard](#)
 - [Docker](#) for reproducibility and cross-platform compatibility
 - Structured unit testing (at least 2 components)
 - Visualization outputs (e.g., GIFs, performance plots)
 - A clear and well-maintained `README.md` with setup and usage instructions

- **Option 2: Reimplementation Using Native TorchRL**

Build the project from scratch using TorchRL's native APIs. Instead of using PettingZoo, start from a TorchRL-compatible environment or adapt an existing one. Design the training pipeline, agent interaction logic, and evaluation procedures entirely within the TorchRL framework. As with the first option, the final solution should support:

- Centralized Training with Decentralized Execution (CTDE)
- Configuration management
- Docker deployment
- WandB/TensorBoard logging
- Visualization and reproducibility tools
- Testing and documentation

Elements to Preserve:

- **PPO Algorithm:** Continue using Proximal Policy Optimization, specifically the clipped variant (PPO-Clip), as the core learning algorithm. [PPO-Clip] (Optionally, you could experiment with [MADDPG], [QMIX], [VDN])
- **MPE Environment (Optional):** The Multi-Agent Particle Environment (MPE) can be retained, though you are also encouraged to consider reimplementing a simplified particle-based environment using native TorchRL. [PettingZoo MPE]
- **Core Objective:** The primary task remains, agents must coordinate to form structured spatial arrangements within designated shapes.
- **Multi-Agent Setting**

Elements to Improve or Redesign:

- **Environmental Complexity:** Introduce static or dynamic obstacles to increase navigation difficulty and promote more strategic coordination.
- **Curriculum Learning:** Implement a training curriculum that gradually increases difficulty, e.g., by varying shape complexity or introducing shape changes mid-episode. [Curriculum Learning]
- **Reward Design:** Develop a more comprehensive reward function that balances formation accuracy, distance to assigned target position, agent spacing, boundary adherence, and obstacle avoidance. [Reward Shaping]
- **Evaluation Metrics:** Add custom metrics for training and evaluation, such as:
 - Formation symmetry
 - Agent spacing variance
 - Obstacle proximity penalties
 - Formation completion time

A Possible Structured Plan for Reimplementation Using Native TorchRL

0. Possible Directory Structure

```
marl-task/  
|-- configs/  
|   |-- base.yaml  
|   |-- env/  
|   |-- algo/  
|   |-- agent/  
|   '-- experiment/  
|-- docker/  
|   |-- Dockerfile  
|   '-- entrypoint.sh  
|-- logs/  
|-- outputs/  
|-- models/  
|   '-- ppo/  
|-- src/  
|   |-- envs/  
|   |-- agents/  
|   |-- rollout/  
|   '-- main.py  
|-- test/  
|   |-- test_env.py  
|   '-- test_metrics.py  
|-- .gitignore  
|-- requirements.txt  
|-- README.md  
 '-- LICENSE
```

1. Environment Setup

Define a Custom TorchRL-Compatible Environment

Create a class `Env(EnvBase)` in `src/envs/env.py` with the following methods:

- `reset(self) -> TensorDict`
- `step(self, actions: TensorDict) -> TensorDict`

Define:

- `observation_spec`
- `action_spec`
- `reward_spec`
- `done_spec`

Ensure all I/O uses `TensorDict`. Observations should be partial and relative, including distance to the shape center and nearest neighbor. Use `torchrl.envs.transforms` for normalization or preprocessing.

Optional: PettingZoo Wrapper

Use PettingZooWrapper from `torchrl.envs.libs.pettingzoo` if adapting from existing environments:

Listing 1: PettingZoo Wrapper Example

```
from torchrl.envs.libs.pettingzoo import PettingZooWrapper
wrapped_env = PettingZooWrapper(pettingzoo_env)
```

2. Agent and Model Definition

Define Policy and Critic Modules

In `src/agents/ppo_agent.py`, implement:

A shared `TensorDictModule` policy:

Listing 2: Shared Policy

```
policy = TensorDictModule(network, in_keys=[...], out_keys=["action"])
```

A centralized critic using `ValueOperator`:

Listing 3: Centralized Critic

```
critic = ValueOperator(critic_network, in_keys=[...])
```

This supports the CTDE paradigm: centralized critic with decentralized policy execution.

3. PPO Training Setup

Collector Configuration

Use `SyncDataCollector` or `MultiSyncDataCollector`:

Listing 4: Collector Configuration

```
collector = SyncDataCollector(
    create_env_fn=env_fn,
    policy=policy,
    frames_per_batch=2048,
    total_frames=...
)
```

Loss Function

Use `ClipPPOLoss`:

Listing 5: PPO Loss Module

```
loss_module = ClipPPOLoss(
    actor=policy,
    critic=critic,
    clip_epsilon=0.2,
    entropy_coef=0.01
)
```

4. Training Loop

Training in main.py

Set up the training loop using `collector`, `replay_buffer`, `loss_module`, and `optimizer`:

Listing 6: Training Loop

```
for batch in collector:
    for _ in range(ppo_epochs):
        loss = loss_module(batch)
        loss.backward()
        optimizer.step()
        optimizer.zero_grad()
```

5. Evaluation and Logging

Logging

Use TensorBoard or W&B:

Listing 7: TensorBoard Logging

```
from torch.utils.tensorboard import SummaryWriter
writer = SummaryWriter(log_dir=...)
writer.add_scalar("reward/mean", mean_reward, step)
```

Evaluation

Run trained policies with local observations only (CTDE) and export GIFs using `pygame`, `matplotlib`, or `imageio`. Store results in `outputs/`.

6. Configuration Management

Hydra Integration

Use Hydra or structured YAML configs in `configs/`:

- `configs/env/task.yaml`
- `configs/algo/ppo.yaml`
- `configs/experiment/sweep.yaml`

Launch with:

Listing 8: Launch Command

```
python src/main.py +experiment=task +algo=ppo
```

7. Testing

Unit Tests

Place tests in `test/`:

Listing 9: Unit Test Example

```
def test_env_reset():
    env = Env(...)
    td = env.reset()
    assert "observation" in td
```

8. CTDE Framework Details

- The shared policy is trained with access to a centralized value function.
- Execution uses only local observations per agent.
- During inference, policies should operate without access to the global state or other agents' observations.
- Ensure the actor's input keys are restricted to local observations, while the critic receives richer information.

9. Docker for Reproducibility

Add Docker Support

Create a `docker/` folder with the following:

Dockerfile:

Listing 10: Dockerfile Example

```
FROM python:3.12-slim
WORKDIR /app
COPY . /app
RUN pip install -r requirements.txt
CMD ["python", "src/main.py"]
```

`entrypoint.sh` (optional launcher script)

Build and run:

Listing 11: Docker Build and Run

```
docker build -t marl-task .
docker run --rm marl-task
```

PowerPoint Presentation

While presenting your work is not mandatory, **not presenting will limit your maximum grade to 3**. The presentation serves as a concise overview of your project.

Duration

Aim for a few well-organized slides that complement your documentation.

Suggested Structure

1. **Title & Objective:** Brief objective and project direction.
2. **System Architecture:** High-level overview (environment, agent setup, training loop).
3. **Environment & Task Setup:** Describe environment, agent logic, and dynamics.
4. **Key Design Choices:** Reward shaping, curriculum, metrics, logging.
5. **Results & Visualizations:** GIFs, reward curves, training plots, insights.
6. **Conclusion & Future Work:** Key takeaways.



Important Notes

The core of your submission is your documentation and code, which will be the primary basis for grading. The presentation is your opportunity to highlight contributions and insights.

Assignment Submission and General Rules

- All development must be carried out within a **GitHub repository**.
- For teams:
 - Collaboration strategy (e.g., shared/individual branches) is up to you.
 - **Task division must be clearly defined** in the project's README.
- For individuals: use a separate branch in the repository.
- Submit a single ZIP file to Canvas with:
 - The entire project repository (excluding large model files/checkpoints).
 - The presentation in PDF format.
- Collaboration is highly encouraged; this is a large-scale assignment.

Prepared by: Tamás Takács

Date: 2025

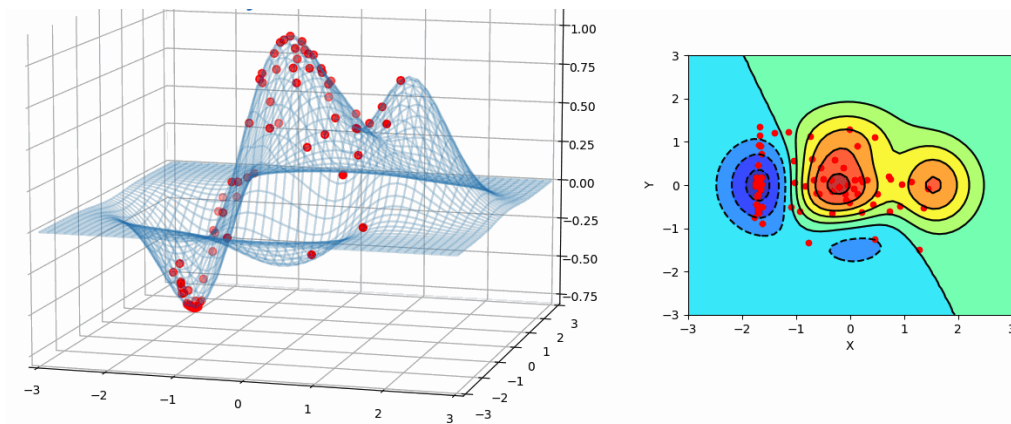
Licensed under [CC BY-NC-ND 4.0](#). © Tamás Takács, 2025.

Particle Swarm Optimization

Collective Intelligence

Multi-Agent Reinforcement Learning

2025



Task Description

This assignment focuses on implementing a **Multi-Agent Reinforcement Learning (MARL)** system using [TorchRL](#). The goal is to design and train a swarm of particles that collaboratively optimize a given objective function in a multi-dimensional search space. While traditionally PSO relies on deterministic position and velocity updates, this assignment explores a new angle: training particle agents using reinforcement learning to learn optimization behavior in a swarm setting. Agents are expected to mimic cooperative PSO-like behavior through interaction and learning, rather than rule-based updates.

Project GitHub Link:

<https://github.com/elte-collective-intelligence/student-particle-swarm-optimization>

At this stage, the project is in its early conceptual phase—no implementation or starter code has been developed yet. The idea remains exploratory and has not been successfully realized in previous semesters. It represents an original contribution within the context of the class.

Agents must coordinate their movements to converge efficiently toward the global optimum, while preserving diversity and even spatial distribution across the search space. The system should adapt to both static and dynamically changing objective landscapes, simulating real-world conditions where optima may shift over time. This setting requires cooperative behavior, best addressed with a Centralized Training with Decentralized Execution (CTDE) approach, where policies are trained with access to global critic information but executed independently by each agent. [\[CTDE Reference\]](#)

While the final goal is to use TorchRL's native environment structure (`EnvBase`, `TensorDict`, etc.), you may initially use PettingZoo environments with the official `PettingZooWrapper` provided by TorchRL, if helpful for bootstrapping.

Assignment Directions

You may choose between the following development directions for this assignment:

- **Option 1: Incremental Migration (not available for PSO)**

Maintain the current implementation based on Stable-Baselines3 (SB3), Supersuit, and PettingZoo, and gradually migrate the system to TorchRL. This approach involves adapting the environment and training loop to TorchRL-compatible components while preserving existing functionality. The migration should also include:

- Integration of a configuration management system (e.g., `Hydra` or structured YAML)
- Preservation of logging via both `Weights & Biases` (WandB) and TensorBoard
- `Docker` for reproducibility and cross-platform compatibility
- Structured unit testing (at least 2 components)
- Visualization outputs (e.g., GIFs, performance plots)
- A clear and well-maintained `README.md` with setup and usage instructions

- **Option 2: Reimplementation Using Native TorchRL**

Build the project from scratch using TorchRL's native APIs. Instead of using PettingZoo, start from a TorchRL-compatible environment (e.g., based on `EnvBase`) or adapt an existing one. Design the training pipeline, agent interaction logic, and evaluation procedures entirely within the TorchRL framework. As with the first option, the final solution should support:

- Centralized Training with Decentralized Execution (CTDE)
- Configuration management
- Docker deployment
- WandB/TensorBoard logging
- Visualization and reproducibility tools
- Testing and documentation

Elements to Consider:

- Utilize Proximal Policy Optimization, specifically the clipped variant (`PPO-Clip`), as the core learning algorithm. Optionally, experiment with `MADDPG`, `QMIX`, or `VDN`.
- Custom Environment (Optional): While benchmark functions (e.g., Sphere, Rastrigin) are suitable, consider implementing a custom dynamic optimization environment, where the objective landscape shifts over time or incorporates constraints.
- Introduce static or time-varying constraints, such as obstacles or restricted zones in the search space.
- Each particle receives a partial observation: its own coordinates, velocity, personal best score, and optionally, a soft neighborhood summary (e.g., average position or fitness of nearby agents within a fixed radius).

- Integrate a curriculum-based optimization schedule, gradually increasing task complexity (e.g., moving from unimodal to multimodal landscapes, increasing dimensionality, or introducing shifting optima). [\[Curriculum Learning\]](#)
- Shape the reward function to promote desirable swarm behaviors—such as spatial dispersion (to maintain diversity) and avoidance of premature convergence (e.g., by penalizing stagnation or collapse into local optima) [\[Reward Shaping\]](#). Can agents specialize into scouts, exploiters, or spreaders without being told to? Design your observation and reward schemes to allow such roles to emerge.
- Design and track custom metrics to evaluate the performance of your swarm, such as:
 - Convergence speed
 - Global vs. local optima ratio
 - Diversity of particles (e.g., average inter-particle distance)
 - Stability in dynamic environments

A Possible Structured Plan for Implementation Using Native TorchRL

0. Possible Directory Structure

```
marl-task/  
|-- configs/  
|   |-- base.yaml  
|   |-- env/  
|   |-- algo/  
|   |-- agent/  
|   '-- experiment/  
|-- docker/  
|   |-- Dockerfile  
|   '-- entrypoint.sh  
|-- logs/  
|-- outputs/  
|-- models/  
|   '-- ppo/  
|-- src/  
|   |-- envs/  
|   |-- agents/  
|   |-- rollout/  
|   '-- main.py  
|-- test/  
|   |-- test_env.py  
|   '-- test_metrics.py  
|-- .gitignore  
|-- requirements.txt  
|-- README.md  
'-- LICENSE
```

1. Environment Setup

Define a Custom TorchRL-Compatible Environment

Create a class `Env(EnvBase)` in `src/envs/env.py` with the following methods:

- `reset(self) -> TensorDict`
- `step(self, actions: TensorDict) -> TensorDict`

Define:

- `observation_spec`
- `action_spec`
- `reward_spec`
- `done_spec`

Ensure all I/O uses `TensorDict`. Observations should be partial and relative, including distance to the shape center and nearest neighbor. Use `torchrl.envs.transforms` for normalization or preprocessing.

Optional: PettingZoo Wrapper

Use `PettingZooWrapper` from `torchrl.envs.libs.pettingzoo` if adapting from existing environments:

Listing 1: PettingZoo Wrapper Example

```
from torchrl.envs.libs.pettingzoo import PettingZooWrapper
wrapped_env = PettingZooWrapper(pettingzoo_env)
```

2. Agent and Model Definition

Define Policy and Critic Modules

In `src/agents/ppo_agent.py`, implement:

A shared `TensorDictModule` policy:

Listing 2: Shared Policy

```
policy = TensorDictModule(network, in_keys=[...], out_keys=["action"])
```

A centralized critic using `ValueOperator`:

Listing 3: Centralized Critic

```
critic = ValueOperator(critic_network, in_keys=[...])
```

This supports the CTDE paradigm: centralized critic with decentralized policy execution.

3. PPO Training Setup

Collector Configuration

Use `SyncDataCollector` or `MultiSyncDataCollector`:

Listing 4: Collector Configuration

```
collector = SyncDataCollector(
    create_env_fn=env_fn,
    policy=policy,
    frames_per_batch=2048,
    total_frames=...
)
```

Loss Function

Use ClipPPOLoss:

Listing 5: PPO Loss Module

```
loss_module = ClipPPOLoss(  
    actor=policy,  
    critic=critic,  
    clip_epsilon=0.2,  
    entropy_coef=0.01  
)
```

4. Training Loop

Training in main.py

Set up the training loop using collector, replay_buffer, loss_module, and optimizer:

Listing 6: Training Loop

```
for batch in collector:  
    for _ in range(ppo_epochs):  
        loss = loss_module(batch)  
        loss.backward()  
        optimizer.step()  
        optimizer.zero_grad()
```

5. Evaluation and Logging

Logging

Use TensorBoard or W&B:

Listing 7: TensorBoard Logging

```
from torch.utils.tensorboard import SummaryWriter  
writer = SummaryWriter(log_dir=...)  
writer.add_scalar("reward/mean", mean_reward, step)
```

Evaluation

Run trained policies with local observations only (CTDE) and export GIFs using pygame, matplotlib, or imageio. Store results in outputs/.

6. Configuration Management

Hydra Integration

Use Hydra or structured YAML configs in configs/:

- configs/env/task.yaml
- configs/algo/ppo.yaml
- configs/experiment/sweep.yaml

Launch with:

Listing 8: Launch Command

```
python src/main.py +experiment=task +algo=ppo
```

7. Testing

Unit Tests

Place tests in `test/`:

Listing 9: Unit Test Example

```
def test_env_reset():
    env = Env(...)
    td = env.reset()
    assert "observation" in td
```

8. CTDE Framework Details

- The shared policy is trained with access to a centralized value function.
- Execution uses only local observations per agent.
- During inference, policies should operate without access to the global state or other agents' observations.
- Ensure the actor's input keys are restricted to local observations, while the critic receives richer information.

9. Docker for Reproducibility

Add Docker Support

Create a `docker/` folder with the following:

Dockerfile:

Listing 10: Dockerfile Example

```
FROM python:3.12-slim
WORKDIR /app
COPY . /app
RUN pip install -r requirements.txt
CMD ["python", "src/main.py"]
```

`entrypoint.sh` (optional launcher script)

Build and run:

Listing 11: Docker Build and Run

```
docker build -t marl-task .
docker run --rm marl-task
```

PowerPoint Presentation

While presenting your work is not mandatory, **not presenting will limit your maximum grade to 3**. The presentation serves as a concise overview of your project.

Duration

Aim for a few well-organized slides that complement your documentation.



Suggested Structure

1. **Title & Objective:** Brief objective and project direction.
2. **System Architecture:** High-level overview (environment, agent setup, training loop).
3. **Environment & Task Setup:** Describe environment, agent logic, and dynamics.
4. **Key Design Choices:** Reward shaping, curriculum, metrics, logging.
5. **Results & Visualizations:** GIFs, reward curves, training plots, insights.
6. **Conclusion & Future Work:** Key takeaways.

Important Notes

The core of your submission is your documentation and code, which will be the primary basis for grading. The presentation is your opportunity to highlight contributions and insights.

Assignment Submission and General Rules

- All development must be carried out within a **GitHub repository**.
- For teams:
 - Collaboration strategy (e.g., shared/individual branches) is up to you.
 - **Task division must be clearly defined** in the project's README.
- For individuals: use a separate branch in the repository.
- Submit a single ZIP file to Canvas with:
 - The entire project repository (excluding large model files/checkpoints).
 - The presentation in PDF format.
- Collaboration is highly encouraged; this is a large-scale assignment.

Prepared by: Tamás Takács

Date: 2025

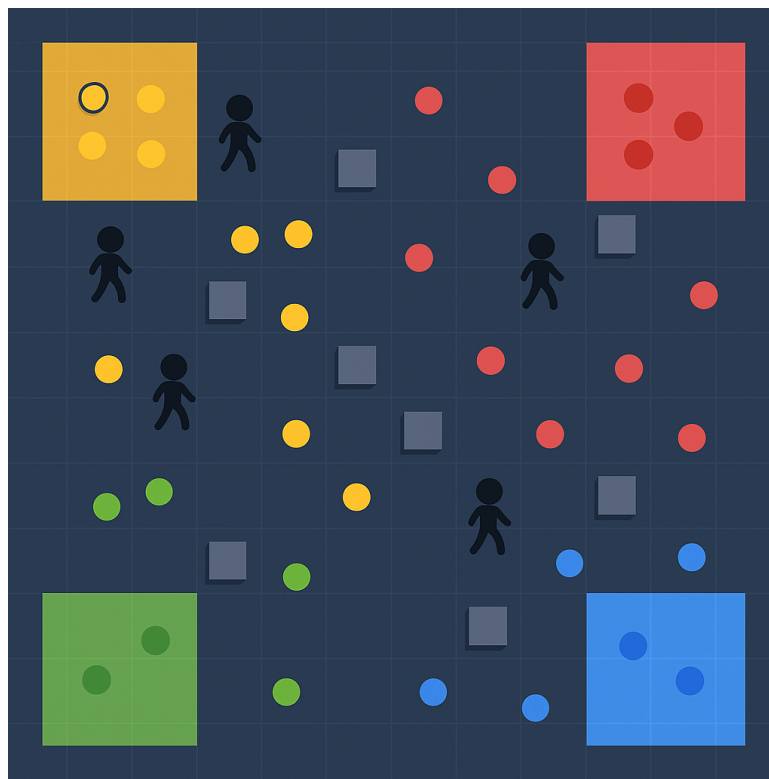
Licensed under [CC BY-NC-ND 4.0](#). © Tamás Takács, 2025.

Search and Rescue

Collective Intelligence

Multi-Agent Reinforcement Learning

2025



Task Description

This assignment focuses on re-implementing a **Multi-Agent Reinforcement Learning (MARL)** system using [TorchRL](#), where a swarm of rescuers must locate and assist stranded victims while navigating an environment with obstacles and dynamic conditions. The task enforces **Centralized Training with Decentralized Execution (CTDE)**—agents are trained with shared information, but deployed with only partial local observations. [\[CTDE Reference\]](#)

Project GitHub Link:

<https://github.com/elte-collective-intelligence/student-search>

The existing project features:

- A PettingZoo AEC-based simulation.
- Victims and rescuers interacting on a dynamic 2D map.
- Landmarks like trees and safe zones.
- Centralized policy training (currently SB3 PPO).
- Visual rendering (PyGame/imageio) and simple evaluation tools.

The **Search and Rescue** scenario involves agents (rescuers) navigating an environment to locate and assist missing persons (victims) while avoiding obstacles. The environment includes various landmarks, such as trees and safe zones, each with specific properties affecting agent behavior.

While the final goal is to use TorchRL's native environment structure (`EnvBase`, `TensorDict`, etc.), you may initially use PettingZoo environments with the official `PettingZooWrapper` provided by TorchRL, if helpful for bootstrapping.

Example: Picture a team of autonomous drones scanning earthquake rubble for survivors. Each drone can see locally, but needs coordinated behavior to cover ground efficiently, rescue victims, and avoid getting stuck.

Environment Phases

1. **Exploration Phase:** Rescuers disperse and search for victims.
2. **Rescue Phase:** Victims are located and guided to safety.
3. **Obstacle Phase:** Agents must avoid trees, walls, or hazards.

Assignment Directions

You may choose between the following development directions for this assignment:

- **Option 1: Incremental Migration**

Maintain the current implementation based on Stable-Baselines3 (SB3), Supersuit, and PettingZoo, and gradually migrate the system to TorchRL-compatible components. The migration should also include:

- Integration of a configuration management system (e.g., `Hydra` or structured YAML)
- Preservation of logging via both `Weights & Biases` (WandB) and TensorBoard
- `Docker` for reproducibility and cross-platform compatibility
- Structured unit testing (at least 2 components)
- Visualization outputs (e.g., GIFs, performance plots)
- A clear and well-maintained `README.md` with setup and usage instructions

- **Option 2: Reimplementation Using Native TorchRL**

Build the project from scratch using TorchRL's native APIs. Instead of using PettingZoo, start from a TorchRL-compatible environment (e.g., based on `EnvBase`) or adapt an existing one. Design the training pipeline, agent interaction logic, and evaluation procedures entirely within the TorchRL framework. As with the first option, the final solution should support:

- Centralized Training with Decentralized Execution (CTDE)
- Configuration management
- Docker deployment
- WandB/TensorBoard logging
- Visualization and reproducibility tools
- Testing and documentation

Elements to Preserve:

- PPO Algorithm: Continue using Proximal Policy Optimization, specifically the clipped variant (PPO-Clip), as the core learning algorithm. [PPO-Clip] (Optionally: [MADDPG], [QMIX], [VDN])
- MPE Environment (Optional): The Multi-Agent Particle Environment (MPE) can be retained, though you are encouraged to consider reimplementing a simplified particle-based environment using native TorchRL. [PettingZoo MPE]
- Core Objective: The primary task remains, agents must locate, reach, and rescue victims in a shared environment, coordinating without full knowledge of each other's positions.
- Multi-Agent Setting

Elements to Improve or Redesign:

- Introduce dynamic obstacles, varied terrain types (e.g., slippery, blocked, hazardous), and randomized victim spawn logic.
- Curriculum Learning: Start with simplified scenarios, then gradually increase map size, obstacle density, or victim mobility to foster generalization. [Curriculum Learning]
- Shape the reward to encourage fast and safe rescues, penalizing delays, collisions, and ineffective exploration patterns. [Reward Shaping]
- Design and track custom metrics such as:
 - Rescue success rate
 - Time taken to complete missions
 - Number of collisions with obstacles

A Possible Structured Plan for Implementation Using Native TorchRL

0. Possible Directory Structure

```
marl-task/  
|-- configs/  
|   |-- base.yaml  
|   |-- env/  
|   |-- algo/  
|   |-- agent/  
|   '-- experiment/  
|-- docker/  
|   |-- Dockerfile
```

```
| '-- entrypoint.sh
|-- logs/
|-- outputs/
|-- models/
| '-- ppo/
|-- src/
|   |-- envs/
|   |-- agents/
|   |-- rollout/
| '-- main.py
|-- test/
|   |-- test_env.py
|   '-- test_metrics.py
|-- .gitignore
|-- requirements.txt
|-- README.md
'-- LICENSE
```

1. Environment Setup

Define a custom TorchRL-compatible environment. Create a class `Env(EnvBase)` in `src/envs/env.py` with the following methods:

- `reset(self) -> TensorDict`
- `step(self, actions: TensorDict) -> TensorDict`

Define:

- `observation_spec`
- `action_spec`
- `reward_spec`
- `done_spec`

Ensure all I/O uses `TensorDict`. Observations should be partial and relative, including distance to the shape center and nearest neighbor. Use `torchrl.envs.transforms` for normalization or preprocessing.

Optional: PettingZoo Wrapper

Use `PettingZooWrapper` from `torchrl.envs.libs.pettingzoo` if adapting from existing environments:

Listing 1: PettingZoo Wrapper Example

```
from torchrl.envs.libs.pettingzoo import PettingZooWrapper
wrapped_env = PettingZooWrapper(pettingzoo_env)
```

2. Agent and Model Definition

Define policy and critic modules in `src/agents/ppo_agent.py`:

A shared `TensorDictModule` policy:

Listing 2: Shared Policy

```
policy = TensorDictModule(network, in_keys=[...], out_keys=["action"])
```

A centralized critic using `ValueOperator`:

Listing 3: Centralized Critic

```
critic = ValueOperator(critic_network, in_keys=[...])
```

This supports the CTDE paradigm: centralized critic with decentralized policy execution.

3. PPO Training Setup

Collector configuration:

Listing 4: Collector Configuration

```
collector = SyncDataCollector(  
    create_env_fn=env_fn,  
    policy=policy,  
    frames_per_batch=2048,  
    total_frames=...  
)
```

Loss function:

Listing 5: PPO Loss Module

```
loss_module = ClipPPOLoss(  
    actor=policy,  
    critic=critic,  
    clip_epsilon=0.2,  
    entropy_coef=0.01  
)
```

4. Training Loop

Set up the training loop using `collector`, `replay_buffer`, `loss_module`, and `optimizer`:

Listing 6: Training Loop

```
for batch in collector:  
    for _ in range(ppo_epochs):  
        loss = loss_module(batch)  
        loss.backward()  
        optimizer.step()  
        optimizer.zero_grad()
```

5. Evaluation and Logging

Logging:

Listing 7: TensorBoard Logging

```
from torch.utils.tensorboard import SummaryWriter  
writer = SummaryWriter(log_dir=...)  
writer.add_scalar("reward/mean", mean_reward, step)
```

Evaluation: Run trained policies with local observations only (CTDE) and export GIFs using `pygame`, `matplotlib`, or `imageio`. Store results in `outputs/`.

6. Configuration Management

Use Hydra or structured YAML configs in `configs/`. Launch with:

Listing 8: Launch Command

```
python src/main.py +experiment=task +algo=ppo
```

7. Testing

Unit tests in `test/`:

Listing 9: Unit Test Example

```
def test_env_reset():
    env = Env(...)
    td = env.reset()
    assert "observation" in td
```

8. CTDE Framework Details

- The shared policy is trained with access to a centralized value function.
- Execution uses only local observations per agent.
- During inference, policies should operate without access to the global state or other agents' observations.
- Ensure the actor's input keys are restricted to local observations, while the critic receives richer information.

9. Docker for Reproducibility

Create a `docker/` folder with the following:

Dockerfile:

Listing 10: Dockerfile Example

```
FROM python:3.12-slim
WORKDIR /app
COPY . /app
RUN pip install -r requirements.txt
CMD ["python", "src/main.py"]
```

Build and run:

Listing 11: Docker Build and Run

```
docker build -t marl-task .
docker run --rm marl-task
```

PowerPoint Presentation

While presenting your work is not mandatory, **not presenting will limit your maximum grade to 3**. The presentation serves as a concise overview of your project.

Duration: Aim for a few well-organized slides that complement your documentation.

Suggested Structure:

1. Title & Objective: Brief objective and project direction.
2. System Architecture: High-level overview (environment, agent setup, training loop).
3. Environment & Task Setup: Describe environment, agent logic, and dynamics.
4. Key Design Choices: Reward shaping, curriculum, metrics, logging.
5. Results & Visualizations: GIFs, reward curves, training plots, insights.
6. Conclusion & Future Work: Key takeaways.

Important Notes: The core of your submission is your documentation and code, which will be the primary basis for grading. The presentation is your opportunity to highlight contributions and insights.



Assignment Submission and General Rules

- All development must be carried out within a **GitHub repository**.
- For teams:
 - Collaboration strategy (e.g., shared/individual branches) is up to you.
 - **Task division must be clearly defined** in the project's README.
- For individuals: use a separate branch in the repository.
- Submit a single ZIP file to Canvas with:
 - The entire project repository (excluding large model files/checkpoints).
 - The presentation in PDF format.
- Collaboration is highly encouraged; this is a large-scale assignment.

Prepared by: Zoltán Barta

Date: 2025

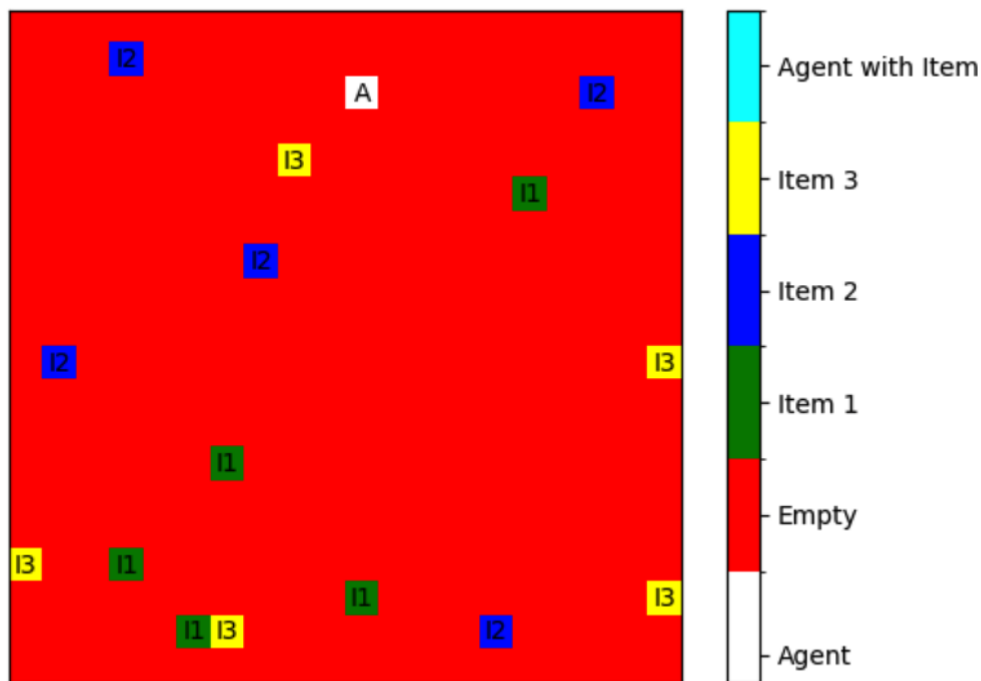
Licensed under [CC BY-NC-ND 4.0](#), © Zoltán Barta, 2025.

Sorting & Clustering

Collective Intelligence

Multi-Agent Reinforcement Learning

2025



Task Description

This assignment focuses on implementing a **Multi-Agent Reinforcement Learning (MARL)** system using [TorchRL](https://pytorch.org/torchrl/), where multiple agents (ants) interact in a shared 2D environment containing scattered items of different categories. Items are scattered randomly at spawn and do not have pre-defined destinations. Agents must dynamically form spatial clusters based on category, rather than placing items at fixed goal locations. The agents must learn to pick up, **sort and cluster these items into coherent spatial groups** across the map, based on their types.

Project GitHub Link:

<https://github.com/elte-collective-intelligence/student-sorting-clustering>

The current project versions utilize:

- A PettingZoo AEC (Agent Environment Cycle) environment, customized to simulate 2D pathfinding with discrete grid or continuous motion.
- A TensorFlow based PPO implementation.
- Real-time visual rendering using pygame and post-processed visualizations.
- Centralized Training and Centralized Execution model. A single shared policy is trained alongside a centralized value function that has access to shared information across agents. During evaluation, this same centralized policy is used by all agents, meaning that each agent's behavior is determined by a common model, rather than acting independently based on purely local observations.
- A simple agent evaluation framework.

Agents must learn to:

- Perceive local item types and neighborhood density.
- Decide which item to move and where to place it.
- Avoid interfering with teammates while cooperatively building category-specific clusters.
- Adapt to varying map layouts, object types, and dynamic item spawning conditions.

This setting requires cooperative behavior, best addressed with a **Centralized Training with Decentralized Execution (CTDE)** approach, where policies are trained with access to global critic information but executed independently by each agent. [\[CTDE Reference\]](#)

While the final goal is to use TorchRL's native environment structure (`EnvBase`, `TensorDict`, etc.), you may initially use PettingZoo environments with the official `PettingZooWrapper` provided by TorchRL, if helpful for bootstrapping.

Note: This task is loosely inspired by **ant-based clustering** models, where agents interact locally with their environment to collectively sort objects into piles without explicit coordination or global knowledge.

Assignment Directions

You may choose between the following development directions for this assignment:

- **Option 1: Incremental Migration**

Maintain the current implementation based on TensorFlow (SB3) and gradually migrate the system to TorchRL. This approach involves adapting the environment and training loop to TorchRL-compatible components while preserving existing functionality. The migration should also include:

- Integration of a configuration management system (e.g., [Hydra](#) or structured YAML)
- Inclusion of logging via both [Weights & Biases](#) (WandB) and TensorBoard
- [Docker](#) for reproducibility and cross-platform compatibility
- Structured unit testing (at least 2 components)
- Visualization outputs (e.g., GIFs, performance plots)
- A clear and well-maintained `README.md` with setup and usage instructions

• Option 2: Reimplementation Using Native TorchRL

Build the project from scratch using TorchRL's native APIs. Start from a TorchRL-compatible environment (e.g., based on **EnvBase**) or adapt an existing one. Design the training pipeline, agent interaction logic, and evaluation procedures entirely within the TorchRL framework. As with the first option, the final solution should support:

- Centralized Training with Decentralized Execution (CTDE)
- Configuration management
- Docker deployment
- WandB/TensorBoard logging
- Visualization and reproducibility tools
- Testing and documentation

Elements to Preserve:

- PPO Algorithm: Continue using Proximal Policy Optimization, specifically the clipped variant (PPO-Clip), as the core learning algorithm. **[PPO-Clip]** (Optionally: **[MADDPG]**, **[QMIX]**, **[VDN]**)
- Core Objective: The primary task remains, agents must learn to pick up, sort and cluster categorized items into coherent spatial groups across the map, based on their types.
- Multi-Agent Setting

Elements to Improve or Redesign:

- Environmental Complexity: Add multiple item types (e.g., red, green, blue blocks) with varying spawn locations. Include distractors, obstacles, or limited pickup range. Optionally allow agents to pick up, carry, or drop items in 2D space. Consider map setups with limited clustering zones or spatial constraints.
- PettingZoo Support (Optional): Leverage existing tools or port environments to TorchRL format.
- Curriculum Learning: Start with simple and small maps, then gradually increase complexity (e.g., more item categories, obstacles). **[Curriculum Learning]**
- Reward Design: Develop a reward function that balances sorting efficiency (sorting items in their respective cluster), obstacle avoidance, time efficiency, and collaborative movement (e.g., penalizing blocking teammates or deadlocks). **[Reward Shaping]** Reward shaping should encourage correct placement (item ends up near similar items), penalize misplacement (e.g., placing a red item in a blue cluster), and incentivize space-efficient clustering.
- Evaluation Metrics: Add custom metrics for training and evaluation, such as:
 - Clustering purity
 - Mean sorting accuracy
 - Conflict rate
 - Idle time
- For clustering purity or accuracy, you can calculate the mean intra-cluster distance or use DBSCAN-style heuristics over item positions after an episode.

A Possible Structured Plan for Implementation Using Native TorchRL

0. Possible Directory Structure

```
marl-task/  
|-- configs/  
|   |-- base.yaml  
|   |-- env/  
|   |-- algo/  
|   |-- agent/  
|   '-- experiment/  
|-- docker/  
|   |-- Dockerfile  
|   '-- entrypoint.sh  
|-- logs/  
|-- outputs/  
|-- models/  
|   '-- ppo/  
|-- src/  
|   |-- envs/  
|   |-- agents/  
|   |-- rollout/  
|   '-- main.py  
|-- test/  
|   |-- test_env.py  
|   '-- test_metrics.py  
|-- .gitignore  
|-- requirements.txt  
|-- README.md  
'-- LICENSE
```

1. Environment Setup

Define a custom TorchRL-compatible environment. Create a class `Env(EnvBase)` in `src/envs/env.py` with the following methods:

- `reset(self) -> TensorDict`
- `step(self, actions: TensorDict) -> TensorDict`

Define:

- `observation_spec`
- `action_spec`
- `reward_spec`
- `done_spec`

Ensure all I/O uses `TensorDict`. Observations should be partial and relative, including distance to the shape center and nearest neighbor. Use `torchrl.envs.transforms` for normalization or preprocessing.

Optional: PettingZoo Wrapper

Use `PettingZooWrapper` from `torchrl.envs.libs.pettingzoo` if adapting from existing environments:

Listing 1: PettingZoo Wrapper Example

```
from torchrl.envs.libs.pettingzoo import PettingZooWrapper
wrapped_env = PettingZooWrapper(pettingzoo_env)
```

2. Agent and Model Definition

Define policy and critic modules in `src/agents/ppo_agent.py`:

A shared `TensorDictModule` policy:

Listing 2: Shared Policy

```
policy = TensorDictModule(network, in_keys=[...], out_keys=["action"])
```

A centralized critic using `ValueOperator`:

Listing 3: Centralized Critic

```
critic = ValueOperator(critic_network, in_keys=[...])
```

This supports the CTDE paradigm: centralized critic with decentralized policy execution.

3. PPO Training Setup

Collector configuration:

Listing 4: Collector Configuration

```
collector = SyncDataCollector(
    create_env_fn=env_fn,
    policy=policy,
    frames_per_batch=2048,
    total_frames=...
)
```

Loss function:

Listing 5: PPO Loss Module

```
loss_module = ClipPPOLoss(
    actor=policy,
    critic=critic,
    clip_epsilon=0.2,
    entropy_coef=0.01
)
```

4. Training Loop

Set up the training loop using `collector`, `replay_buffer`, `loss_module`, and `optimizer`:

Listing 6: Training Loop

```
for batch in collector:
    for _ in range(ppo_epochs):
        loss = loss_module(batch)
        loss.backward()
        optimizer.step()
        optimizer.zero_grad()
```

5. Evaluation and Logging

Logging:

Listing 7: TensorBoard Logging

```
from torch.utils.tensorboard import SummaryWriter
writer = SummaryWriter(log_dir=...)
writer.add_scalar("reward/mean", mean_reward, step)
```



Evaluation: Run trained policies with local observations only (CTDE) and export GIFs using `pygame`, `matplotlib`, or `imageio`. Store results in `outputs/`.

6. Configuration Management

Use Hydra or structured YAML configs in `configs/`. Launch with:

Listing 8: Launch Command

```
python src/main.py +experiment=task +algo=ppo
```

7. Testing

Unit tests in `test/`:

Listing 9: Unit Test Example

```
def test_env_reset():
    env = Env(...)
    td = env.reset()
    assert "observation" in td
```

8. CTDE Framework Details

- The shared policy is trained with access to a centralized value function.
- Execution uses only local observations per agent.
- During inference, policies should operate without access to the global state or other agents' observations.
- Ensure the actor's input keys are restricted to local observations, while the critic receives richer information.

9. Docker for Reproducibility

Create a `docker/` folder with the following:

Dockerfile:

Listing 10: Dockerfile Example

```
FROM python:3.12-slim
WORKDIR /app
COPY . /app
RUN pip install -r requirements.txt
CMD ["python", "src/main.py"]
```

Build and run:

Listing 11: Docker Build and Run

```
docker build -t marl-task .
docker run --rm marl-task
```

PowerPoint Presentation

While presenting your work is not mandatory, **not presenting will limit your maximum grade to 3**. The presentation serves as a concise overview of your project.

Duration: Aim for a few well-organized slides that complement your documentation.

Suggested Structure:

1. Title & Objective: Brief objective and project direction.



2. System Architecture: High-level overview (environment, agent setup, training loop).
3. Environment & Task Setup: Describe environment, agent logic, and dynamics.
4. Key Design Choices: Reward shaping, curriculum, metrics, logging.
5. Results & Visualizations: GIFs, reward curves, training plots, insights.
6. Conclusion & Future Work: Key takeaways.

Important Notes: The core of your submission is your documentation and code, which will be the primary basis for grading. The presentation is your opportunity to highlight contributions and insights.

Assignment Submission and General Rules

- All development must be carried out within a **GitHub repository**.
- For teams:
 - Collaboration strategy (e.g., shared/individual branches) is up to you.
 - **Task division must be clearly defined** in the project's README.
- For individuals: use a separate branch in the repository.
- Submit a single ZIP file to Canvas with:
 - The entire project repository (excluding large model files/checkpoints).
 - The presentation in PDF format.
- Collaboration is highly encouraged; this is a large-scale assignment.

Prepared by: Tamás Takács

Date: 2025

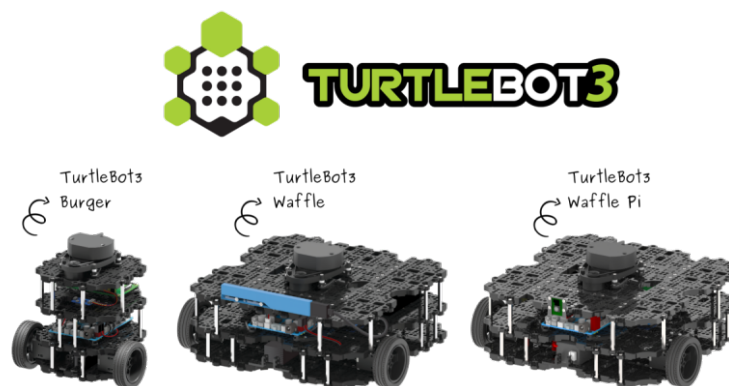
Licensed under [CC BY-NC-ND 4.0](#). © Tamás Takács, 2025.

TurtleBot3s

Collective Intelligence

Multi-Agent Reinforcement Learning

2025



Task Description

This assignment focuses on implementing a **Multi-Agent Reinforcement Learning (MARL)** system using [TorchRL](#). Inspired by real-world cooperative robotics, this challenge requires a team of TurtleBot3 agents to learn **coordinated navigation and obstacle avoidance** in dynamic environments simulated in [ROS2 Humble](#) and [Gazebo](#). Agents must operate in a **shared arena with static and dynamic obstacles**, reaching individual goals without colliding with one another or the environment.

Project GitHub Link:

<https://github.com/elte-collective-intelligence/student-turtlebot3s>

You will start with an existing MARL project built using custom Deep Q-Networks (DQN) integrated with ROS2 nodes. The main task is to **migrate this project to use TorchRL's native ecosystem**.

This task is best addressed using the **Centralized Training with Decentralized Execution (CTDE)** paradigm. During training, agents can access global state information and train with a centralized critic to stabilize learning. During evaluation, each TurtleBot3 agent must act independently based on partial, local sensor data (such as LiDAR, odometry, and goal pose). [\[CTDE Reference\]](#)

While the final goal is to use TorchRL's native environment structure ([EnvBase](#), [TensorDict](#), etc.), you may initially use PettingZoo environments with the official [PettingZooWrapper](#) provided by TorchRL if helpful for bootstrapping.

Assignment Directions

Adapt the Existing TurtleBot3 Environment Using Native TorchRL and MAPPO

- Rebuild the existing ROS2 + Gazebo-based MARL project using TorchRL's native APIs, replacing the current DQN implementation with a **Multi-Agent Proximal Policy Optimization (MAPPO)** architecture.
- Adapt the custom simulation environment to be compatible with **EnvBase** and **TensorDict** (rather than using **PettingZoo**).
- Design a full training pipeline using TorchRL that supports:
 - Centralized Training with Decentralized Execution (CTDE)
 - MAPPO implementation using TorchRL's policy and value networks
 - Integration of a configuration management system (e.g., **Hydra** or structured YAML)
 - Inclusion of logging via both **Weights & Biases** (WandB) and TensorBoard
 - Docker for reproducibility and cross-platform compatibility
 - Structured unit testing (at least 2 components)
 - Visualization outputs (e.g., GIFs, performance plots)
 - A clear and well-maintained **README.md** with setup and usage instructions

Elements to Preserve:

- Existing ROS2 Humble and Gazebo simulation setup, including TurtleBot3 motion primitives, sensor models, and real-time physics-based interactions.
- The core challenge remains *multi-agent goal-reaching without collisions*, where agents must coordinate spatially in a confined shared arena with limited sensing.
- Multi-Agent Setting.

Elements to Improve or Redesign:

- Use Proximal Policy Optimization, specifically the clipped variant (PPO-Clip), as the core learning algorithm. **[PPO-Clip]** (Optionally: **MADDPG**, **QMIX**, **VDN**)
- Extend the task dynamics by introducing tighter corridors, dynamic door states, or shifting obstacles (e.g., mobile barriers or rotating gates) to enforce collision-aware path planning.
- Implement curriculum learning: Start with low-density, obstacle-free arenas and gradually introduce more agents, goal proximity overlaps, and cluttered layouts. **[Curriculum Learning]**
- Refine the reward function to penalize overshooting, deadlocks, or collisions, while positively rewarding coordinated timing, goal occupancy success, and spatial distribution. **[Reward Shaping]**
- Optionally, inject LiDAR noise or add occlusion logic to simulate more realistic sensing and encourage robust policies.
- Design and track custom metrics, such as:
 - Average time-to-goal per agent
 - Number of idle or blocked agents per episode
 - Collision count and type (agent-agent, agent-wall)

A Possible Structured Plan for Implementation Using Native TorchRL

0. Possible Directory Structure

```
marl-task/  
|-- configs/  
|   |-- base.yaml  
|   |-- env/  
|   |-- algo/  
|   |-- agent/  
|   '-- experiment/  
|-- docker/  
|   |-- Dockerfile  
|   '-- entrypoint.sh  
|-- logs/  
|-- outputs/  
|-- models/  
|   '-- ppo/  
|-- src/  
|   |-- envs/  
|   |-- agents/  
|   |-- rollout/  
|   '-- main.py  
|-- test/  
|   |-- test_env.py  
|   '-- test_metrics.py  
|-- .gitignore  
|-- requirements.txt  
|-- README.md  
'-- LICENSE
```

1. Environment Setup

Define a custom TorchRL-compatible environment. Create a class `Env(EnvBase)` in `src/envs/env.py` with the following methods:

- `reset(self) -> TensorDict`
- `step(self, actions: TensorDict) -> TensorDict`

Define:

- `observation_spec`
- `action_spec`
- `reward_spec`
- `done_spec`

Ensure all I/O uses `TensorDict`. Observations should be partial and relative, based on each robot's local sensor data and goal.

Optional: PettingZoo Wrapper

Use `PettingZooWrapper` from `torchrl.envs.libs.pettingzoo` if adapting from existing environments:

Listing 1: PettingZoo Wrapper Example

```
from torchrl.envs.libs.pettingzoo import PettingZooWrapper
wrapped_env = PettingZooWrapper(pettingzoo_env)
```

2. Agent and Model Definition

Define policy and critic modules in `src/agents/ppo_agent.py`:

A shared `TensorDictModule` policy:

Listing 2: Shared Policy

```
policy = TensorDictModule(network, in_keys=[...], out_keys=["action"])
```

A centralized critic using `ValueOperator`:

Listing 3: Centralized Critic

```
critic = ValueOperator(critic_network, in_keys=[...])
```

This supports the CTDE paradigm: centralized critic with decentralized policy execution.

3. PPO Training Setup

Collector configuration:

Listing 4: Collector Configuration

```
collector = SyncDataCollector(
    create_env_fn=env_fn,
    policy=policy,
    frames_per_batch=2048,
    total_frames=...
)
```

Loss function:

Listing 5: PPO Loss Module

```
loss_module = ClipPPOLoss(
    actor=policy,
    critic=critic,
    clip_epsilon=0.2,
    entropy_coef=0.01
)
```

4. Training Loop

Set up the training loop using `collector`, `replay_buffer`, `loss_module`, and `optimizer`:

Listing 6: Training Loop

```
for batch in collector:
    for _ in range(ppo_epochs):
        loss = loss_module(batch)
        loss.backward()
        optimizer.step()
        optimizer.zero_grad()
```

5. Evaluation and Logging

Logging:

Listing 7: TensorBoard Logging

```
from torch.utils.tensorboard import SummaryWriter
writer = SummaryWriter(log_dir=...)
writer.add_scalar("reward/mean", mean_reward, step)
```



Evaluation: Run trained policies with local observations only (CTDE) and export GIFs using `pygame`, `matplotlib`, or `imageio`. Store results in `outputs/`.

6. Configuration Management

Use Hydra or structured YAML configs in `configs/`. Launch with:

Listing 8: Launch Command

```
python src/main.py +experiment=task +algo=ppo
```

7. Testing

Unit tests in `test/`:

Listing 9: Unit Test Example

```
def test_env_reset():
    env = Env(...)
    td = env.reset()
    assert "observation" in td
```

8. CTDE Framework Details

- The shared policy is trained with access to a centralized value function.
- Execution uses only local observations per agent.
- During inference, policies should operate without access to the global state or other agents' observations.
- Ensure the actor's input keys are restricted to local observations, while the critic receives richer information.

9. Docker for Reproducibility

Create a `docker/` folder with the following:

Dockerfile:

Listing 10: Dockerfile Example

```
FROM python:3.12-slim
WORKDIR /app
COPY . /app
RUN pip install -r requirements.txt
CMD ["python", "src/main.py"]
```

Build and run:

Listing 11: Docker Build and Run

```
docker build -t marl-task .
docker run --rm marl-task
```

PowerPoint Presentation

While presenting your work is not mandatory, **not presenting will limit your maximum grade to 3**. The presentation serves as a concise overview of your project.

Duration: Aim for a few well-organized slides that complement your documentation.

Suggested Structure:

1. Title & Objective: Brief objective and project direction.



2. System Architecture: High-level overview (environment, agent setup, training loop).
3. Environment & Task Setup: Describe environment, agent logic, and dynamics.
4. Key Design Choices: Reward shaping, curriculum, metrics, logging.
5. Results & Visualizations: GIFs, reward curves, training plots, insights.
6. Conclusion & Future Work: Key takeaways.

Important Notes: The core of your submission is your documentation and code, which will be the primary basis for grading. The presentation is your opportunity to highlight contributions and insights.

Assignment Submission and General Rules

- All development must be carried out within a **GitHub repository**.
- For teams:
 - Collaboration strategy (e.g., shared/individual branches) is up to you.
 - **Task division must be clearly defined** in the project's README.
- For individuals: use a separate branch in the repository.
- Submit a single ZIP file to Canvas with:
 - The entire project repository (excluding large model files/checkpoints).
 - The presentation in PDF format.
- Collaboration is highly encouraged; this is a large-scale assignment.

Prepared by: Zoltán Barta

Date: 2025

Licensed under [CC BY-NC-ND 4.0](#). © Zoltán Barta, 2025.