COMPUTING AQA NEA

COMPARING REINFORCEMENT LEARNING ALGORITHMS USING MULTIPLE CONFIGURABLE AGENTS IN UNITY

ZAIN RIZWAN, CANDIDATE NUMBER 1347, CENTRE NUMBER 51411,
LANGLEY GRAMMAR SCHOOL

Analysis

Problem Area

Often, in computer science, it is customary for a topic to be taught beginning with the most basic principles, then moving on to a logical implementation of the concept and ending with a program. Then, a given student is free to implement the given concept in their own program and see the inherent results. However, this approach is proven complicated when teaching complex topics in AI, due to its inherently mathematical nature. Students will often spend hours pondering over mathematical formulae, overwhelmed with their intricacies and unfamiliar to anything else they have encountered. Thus, students often skip to the end product using libraries, without taking time to appreciate the nuances of their development. This diminishes the student's knowledge of the inner workings of such a concept, which can be considered a disadvantage given many industries have massively accelerated their use of AI, and skills interacting with AI are in use in more than simply technical industries.

Despite the recent surge in popularity, I myself have been interested in AI for a longer time, before the launch of ChatGPT. Due to my interests in gaming and chess, I consumed a large amount of content relating to algorithms used to train the neural networks behind the outputs of AI agents used to play as a human would, or to learn to complete a particular goal. One channel in particular, Code Bullet, has produced a multitude of videos upon which algorithms like DQN, PPO, and NEAT were used to teach an agent how to solve a simple problem such as walking. However, there was no convenient way to learn about these algorithms and their nuances without an extreme amount of technical detail, and as such my interest remained pure curiosity. This problem extends to the influx of people that want to learn more about AI, and perhaps in particular different reinforcement learning (RL) algorithms.

Solution Decomposition

I plan to address this problem by creating a program where users can configure two agents to use a given algorithm from the list, along with the ability to change the algorithm's parameters from the defaults. Each element of the program will have easily accessible and human-understandable documentation within the program, to help users understand each part of the program's use without substantial research and understanding of the algorithms themselves. Then, the program will run a game of tag (intended to be an arbitrary task for the two agents to compete in), for a configured number of iterations. There will also be an option for the user to pause the game at any given point to watch the agents' progress and strategies at that point in training, and an option for the user to set the program to pause at given intervals (ie every 1000 iterations). This allows the user to not pay attention to the program while it trains, making it more convenient to use for busy users. At the end of the given number of iterations, the user will be displayed a multitude of statistics to do with the agents' performances at different stages of the game and will have the option to export their data to use in other programs.

Interview

Interview Questions

The following outlines some of the questions I am posing to a stakeholder who may be interested in using the proposed solution as a team lead in a large company.

Q: How useful or important is reinforcement learning to what you do?

This question opens the interview, and ensures that the interviewee knows that this will be specific to reinforcement learning.

Q: How do people currently gain an understanding of how different ML/AI/RL algorithms work?

This question aims to understand the current landscape for how people learn about these topics, and, as such, how the developed program could fit into this space.

Q: Do people who are unfamiliar with ML usually have difficulty in understanding how it works from a beginner level?

Q: Do people who are unfamiliar with AI usually have difficulty in understanding how it works from a beginner level?

Q: Do people who are unfamiliar with RL usually have difficulty in understanding how it works from a beginner level?

These questions aim to clarify the nature of learning for each of these fields.

Q: Is there a need for information about ML/AI/RL that is not connected directly to complex implementation? (ie pseudocode, maths, program code) to be available.

Q: Would it be useful for a program to show you the nuances in how a given reinforcement algorithm may learn a task?

These questions aim to understand whether or not the program would be viable to fill in a gap indicated by the previous group of questions.

Q: If yes, what would be the optimal form that the program would make?

Q: If no, what other programs exist?

Depending on the response, find out more about what the program should be or what already exists for users.

Q: What kind of hardware do you have access to? What kind of hardware would you run such a program upon?

This question aims to find out the scope of the program, as lesser hardware requirements result in a lower scope for the program.

Interview

My interviewee was Tom Merrifield, a Shell employee whose work mainly focuses upon using AI to predict things about a given landscape. While not inherently from an AI field, Tom spends much of his time using it and has as such become proficient enough to teach his peers, delivering a talk while I was at Shell upon the topic.

Q: How useful or important is reinforcement learning to what you do?

For my day to day work, it is not very important. However, we have done some research into RL methods and I believe that in time, it will become more useful. It has not had the breakthroughs that have happened for language and images (yet).

Q: How do people currently gain an understanding of how different ML/AI/RL algorithms work?

Mainly through self-learning, but there are some courses that people can join through work.

For the experts, arxiv is the best place for latest papers & research. Tutorials, blogs and social media are also good places to gain an understanding of the various algorithms.

Q: Do people who are unfamiliar with ML usually have difficulty in understanding how it works from a beginner level?

Yes, some of the concepts are not intuitive to beginners.

Q: Do people who are unfamiliar with AI usually have difficulty in understanding how it works from a beginner level?

I see AI as a superset, of which ML is a subset, so the same answer applied to AI as to ML. However, AI is much broader topic, so at a high level, people do tend to grasp how AI works at a beginner level, especially given how much hype there has been about AI in recent years.

Q: Do people who are unfamiliar with RL usually have difficulty in understanding how it works from a beginner level?

This is a lesser known subset of AI, and while beginners struggle with understanding some of the concepts, it can be a simpler concept to describe, at a high level at least, how RL works. (i.e. same as humans do, essentially).

Q: Is there a need for information about ML/AI/RL that is not connected directly to complex implementation? (ie pseudocode, maths, program code) to be available.

Yes, analogies, simple tutorials and high level overviews that do not go into the details of the algorithms are very useful for people to become accustomed to the terminology and general concepts of what an AI algorithm is attempting to do.

Q: Would it be useful for a program to show you the nuances in how a given reinforcement algorithm may learn a task?

This depends - in my opinion, if (and only if) these nuances have a direct impact on the performance of the RL model, and by changing them, a user can improve the outcome, then these should be exposed with clear descriptions so that people can make informed decisions on how to parameterise the learning process and overall outcome of the product.

If the idea is that people learn how RL works, then I do think the nuances are important for people to grasp.

Q: If yes, what would be the optimal form that the program would make?

I like to see hidden "expert" parameters in programs that are set to sensible defaults, so that beginners can safely leave them untouched. But for those who know what they are doing, exposed so that they can tweak the algorithm accordingly. Provide access to clear descriptions, with embedded documentation (or links to resources) that help explain each parameter. Videos and animations are often very useful for more complex scenes. I recently found 3blue1brown from YouTube has a python library that can be used to generate animations which are very useful to help people understand the inner workings of AI algorithms, including maths for those who are interested.

Google's Colab is a useful place for people to play with code using prepared jupyter notebooks.

Q: If no, what other programs exist?

Many tutorials use colab or similar for people to play with code.

Kaggle provides time-bound jupyter notebooks with access to GPUs, as well as access to useful datasets for testing and courses to learn from.

Some programming languages provide playgrounds to test out ideas, for example Rust: https://play.rust-lang.org/ and Go: Go Playground - The Go Programming Language

Q: What kind of hardware do you have access to? What kind of hardware would you run such a program upon?

I have a laptop provided by work that has an integrated GPU, and works only for toy problems.

My main machines for general work are Linux workstations which have nvidia GPU cards, either Quadro M3000 SE with 4GB GPU memory, or A40-12Q with 12GB of GPU memory.

For real work, I have access to an on-premises HPC cluster, which provides access to a range of GPUs:

V100 (between 2-8 cards/machine, 32GB per GPU)

A100 (8 A100 cards/machine, 80GB / GPU)

DGX-2 (16 V100 cards, 23GB / GPU)

DGX A100 (8 A100 cards, 40GB / GPU)

HGX-2 V100 (16 V100 cards, 32GB / GPU)

I also have access to all AWS resources.

Key Takeaways

- Some RL concepts are not intuitive to beginners.
- Currently, people use courses and scientific papers (arxiv) to gain an understanding of how the programs work.
- Configuration and readily available information is a major factor in the success of the program for both beginner and expert levels of knowledge on these algorithms.
- To see the program in action, people use Jupyter notebooks and Google Colab to run simulations upon datasets and see the performance.
- It is said that RL has nuances, and understanding these is important for anyone wishing to apply these algorithms to a given purpose.
- At the workplace, the user has access to very powerful machines that are immediately more powerful that my own machine. The work-given laptop used by the user has similar specification to my development machine, so it can be used as a reasonable benchmark for the program.

Research

Similar Solutions

In order to confirm that reinforcement learning algorithms were worth putting effort into, I proceeded to research more uses of them. In this, I found evidence of multiple industries attempting to apply reinforcement learning, including finance [0], surgery [1], and autonomous driving [2]. Thus, I was able to conclude that reinforcement learning is a worthwhile investment of time for a potential user to teach to their students, given that if they are to go out into an industry where AI is in heavy use, they are likely to encounter these algorithms. Similarly, a team lead may want to educate their team more thoroughly on the techniques that they will be using.

Tag

Reference [3].

One application of RL was the video of tag that inspired this project. The video itself uses a game of tag played by the agents as a demonstration of how agents in general manipulate their environment to maximise reward. The agents were set to train themselves in an environment against one another and were given a reward for winning, with changes to the environment after the agents had made significant progress.

Similarities:

- Agents learn how to play tag.
- Used to show how agents learn.
- Uses Unity to model situation.
- Uses arbitrary situation.
- Keeps track of statistics
- Continuous datasets/environment

Differences:

- Only uses 1 RL algorithm (MA-POCA)
- Does not allow any user configuration.
- Contains no information, only runs the simulations.
- Does not export data.
- Does not graph data.

An example of the Tag environment from the video. [3]

Despite being the inspiration for this project, the scope of the program is different. It was designed primarily to create the content needed for its associated YouTube video, and as such the designated user is the video creator. So, there is not any reason for there to implement functions to allow the user to configure the program or export data (that we know of), as the video creator would have been able to do this simply from code. Similarly, there is no utility in the program itself to view information about the agents' learning processes. Due to the user-friendly intentions of my program, these features will need to be implemented. In like manner, there was only 1 RL algorithm used. Again, in order to allow users to explore different algorithms, the program must have the capability to independently manage and configure each agent, whereas in this example the program only implements a single algorithm.

FinRL

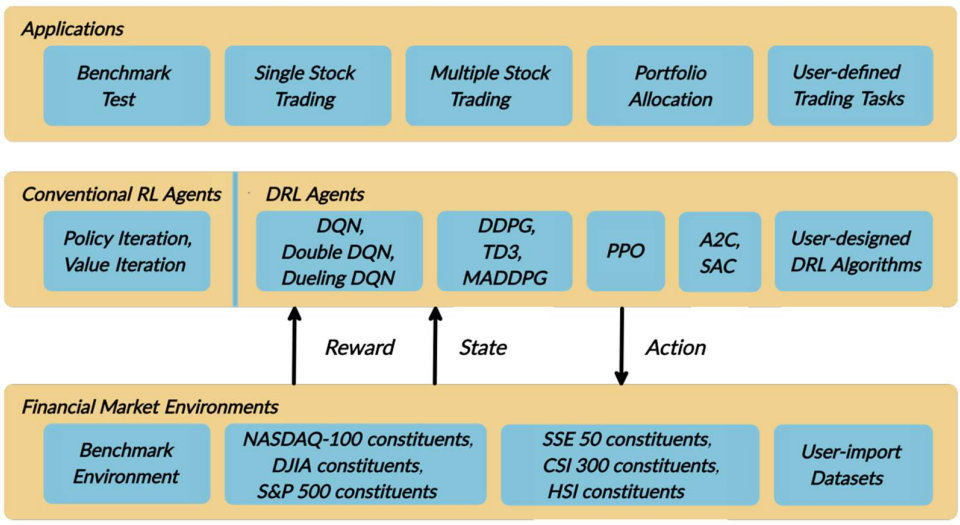
In more detail, the financial application of RL that I discovered was a paper [0] documenting the creation of FinRL, a deep RL library for automated stock trading in quantitative finance. While the paper itself refers simply to the library, it also has showcases of the capabilities of the algorithms implemented using a simulation built into a different layer of the program. As such, the algorithms are shown to work in real-world environments, and the library has surpassed 60,000 downloads on GitHub [4].

Similarities:

- Multiple RL algorithms implemented.
- Allows user configuration.
- Continuous datasets/environment

Differences:

- Provided multiple environments.
- Used in multiple applications.
- Little information on RL with an assumed amount of knowledge
- No UI
- Only models benchmark tests/environment internally.



Different layers of the FinRL Library [0]

Algorithms	Input	Output	Type	State-action spaces support	Finance use cases support	Features and Improvements	Advantages
DQN	States	Q-value	Value based	Discrete only	Single stock trading	Target network, experience replay	Simple and easy to use
Double DQN	States	Q-value	Value based	Discrete only	Single stock trading	Use two identical neural network models to learn	Reduce overestimations
Dueling DQN	States	Q-value	Value based	Discrete only	Single stock trading	Add a specialized dueling Q head	Better differentiate actions, improves the learning
DDPG	State action pair	Q-value	Actor-critic based	Continuous only	Multiple stock trading, portfolio allocation	Being deep Q-learning for continuous action spaces	Better at handling high-dimensional continuous action spaces
A2C	State action pair	Q-value	Actor-critic based	Discrete and continuous	All use cases	Advantage function, parallel gradients updating	Stable, cost-effective, faster and works better with large batch sizes
PPO	State action pair	Q-value	Actor-critic based	Discrete and continuous	All use cases	Clipped surrogate objective function	Improve stability, less variance, simply to implement
SAC	State action pair	Q-value	Actor-critic based	Continuous only	Multiple stock trading, portfolio allocation	Entropy regularization, exploration-exploitation trade-off	Improve stability
TD3	State action pair	Q-value	Actor-critic based	Continuous only	Multiple stock trading, portfolio allocation	Clipped double Q-Learning, delayed policy update, target policy smoothing.	Improve DDPG performance
MADDPG	State action pair	Q-value	Actor-critic based	Continuous only	Multiple stock trading, portfolio allocation	Handle multi-agent RL problem	Improve stability and performance

Information given on RL Algorithms in FinRL [0]

The FinRL library has one of the main parts of my program implemented – the ability to use multiple RL algorithms, and switch between them at will. However, due to its nature as a library, it has no UI for the user to use. Even the paper written upon it has little to no documentation of the actual code or its usage, only the basic structure and its results. Thus, while it is fundamentally quite similar to my project, its execution is extremely specialised to its application and as such it does not align with my goals of teaching the user about these algorithms and allowing them to explore how they may be used. Another way in which this manifests is the fact that the RL algorithms are designed to have multiple different applications once they have been set up. As such, their default configurations have no way to be specialised the application. This is important to my program as users who have not encountered configuring an algorithm such as this may have no idea the common or accepted values for each parameter, and as such having an effective default setting is important to the function of the program.

ChatGPT

Reference [5].

Another application of an RL algorithm used in my program was the training process for ChatGPT, an online LLM-based chatbot that has been become widely known by most people, even those less technically inclined. PPO was used to train ChatGPT-3 to follow instructions, make facts up less often, and overall generate more appropriate outputs. This was done by first collecting a dataset of human-written demonstrations on prompts submitted to the ChatGPT

API, and using this to train the supervised learning baselines. Next, a dataset of human-labelled comparisons between two model outputs on a larger set of API prompts were collected. These were then used to train a reward model on the dataset to predict what output a labeller would prefer. Finally, this model was used as a reward function, and PPO was used to fine-tune ChatGPT to maximise this reward.

Similarities:

- Uses PPO.
- Has a wide range of input parameters.

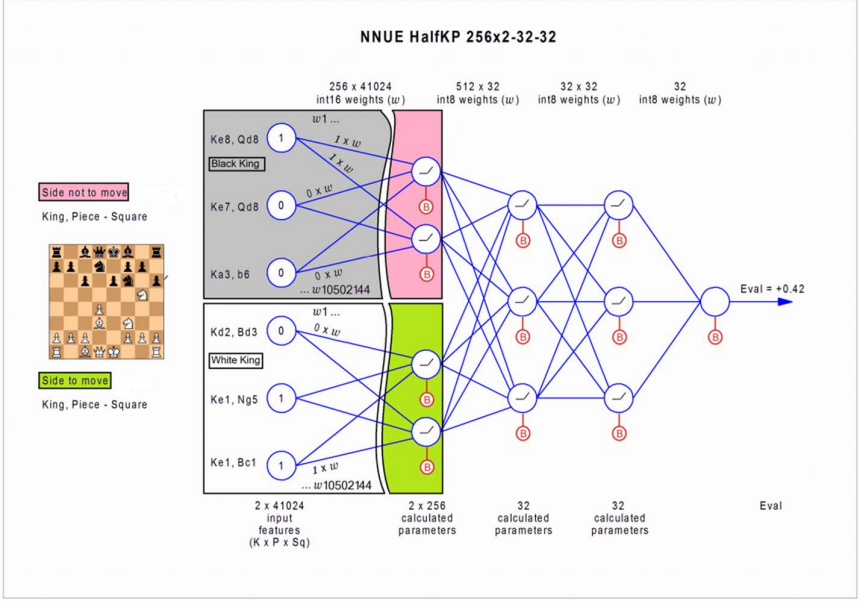
Differences:

- Used in training an already existing system.
- Human-generated dataset.
- A second model was created for the reward function.

This application of PPO was rolled out online for anyone to access, meaning that the algorithm is proven to be reliable in business applications. The wide range of input parameters required for training an LLM suggests that the PPO algorithm is suitable for other RL applications, as most of these occur in continuous sample spaces with a large range of different values. However, it did not learn by performing random actions in an environment. Rather, it used a separately trained reward model and human-generated labelled data to train the model, rather than a simulated situation like my own project.

Neural Networks

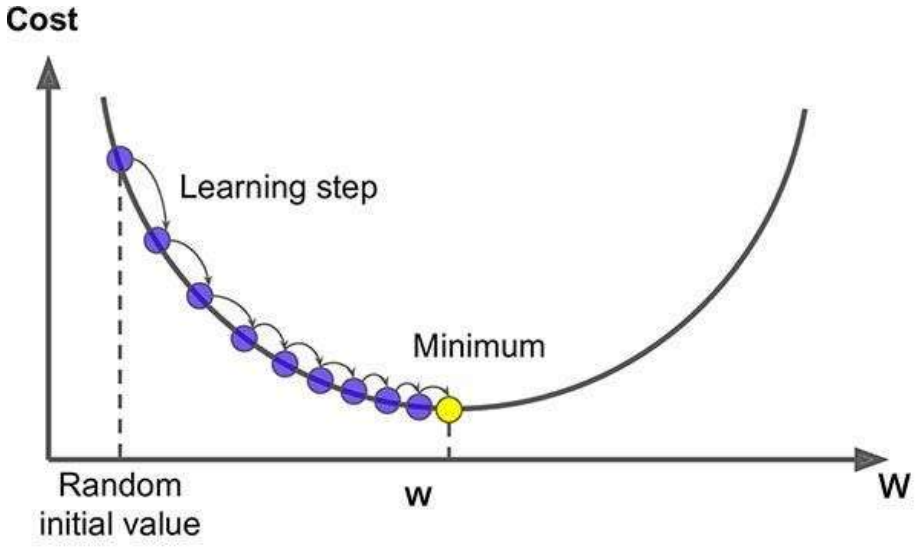
A neural network is a collection of neurones that take a given set of inputs and give an output. They multiply the input along with a value called a "weight", that signifies how much influence one input has on the output of said neuron. After this, a "bias" value is added to the output, allowing the network to account for situation where all input features are 0 or there is no input given. Then, the output is put through an activation function to introduce a non-linear transformation to the network, allowing it to develop more complex learning patterns. In order for a network to "learn", a given piece of input data is given to the network so that it can produce an output. Then, this output is compared to the target output, and the "error" is backpropagated through the network, allowing the network to account for where it went wrong. There are multiple different ways to calculate the "error" of a neural network, or how far away the network is from the desired output.



A neural network used to evaluate a given chess position [6]

Gradient Descent

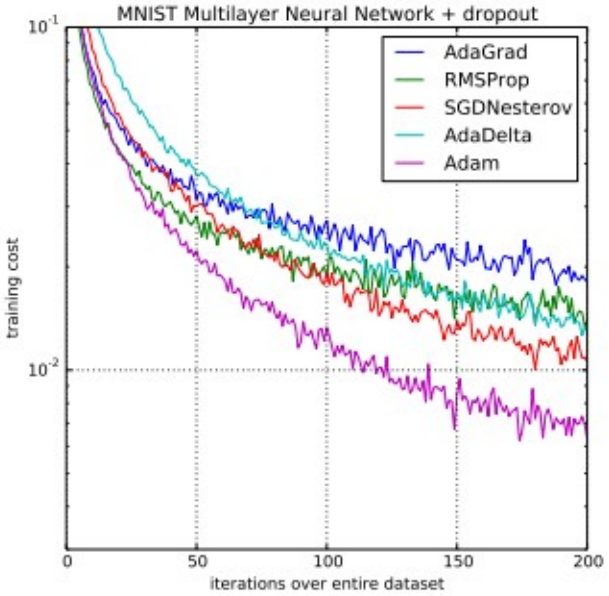
One of the major features of AI is a concept known as gradient descent, which is applied to the learning step of a neural network's development. Essentially, the weights and biases between different nodes of a network are modified until we find the lowest loss possible, and we take steps to move towards those weights and biases. The step taken at each timestep is determined by the "learning rate", and this is often changed dynamically with respect to time in order to allow the network to converge.



Graph showing gradient descent. [7]

Adam Optimisation

Adam (Adaptive Moment Estimation) is an adaptive optimisation algorithm. It is used to update the learning rates for each parameter by calculating the first and second "moments" of each gradient. The first "moment" represents the average gradient, which helps Adam keep track of previous gradients. This referred to as the mean. The second "moment" represents the average of the squared gradients, which helps in scaling the learning rates for each parameter. This is referred to as the "variance." Adam's dynamic updates allow a neural network to update how it changes its parameters during training, ensuring that it can adapt to different situations and events.



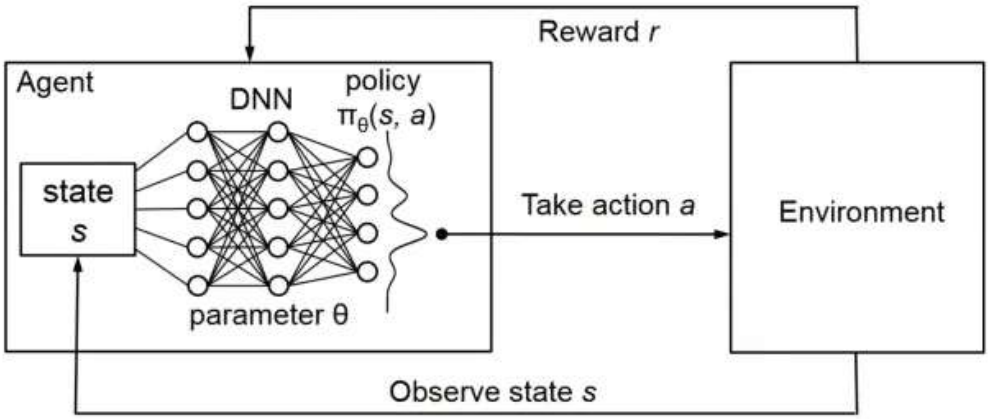
Adam versus other optimisers. [8]

Reinforcement Learning Algorithms

While neural networks can be used to learn a multitude of tasks, it relies on feedback from its actions being immediately available. This is often not the case in real-world situations, so we resort to reinforcement learning to learn from delayed rewards, allowing us to optimise relatively longer-term effects compared to neural networks, which often only optimise for a given decision.

DQN

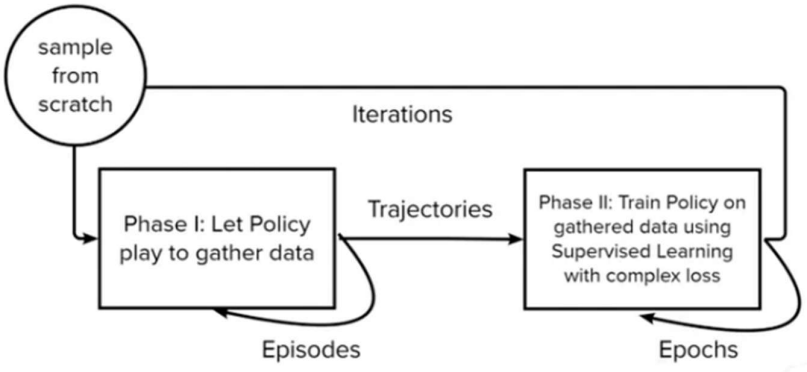
For a given input to a DQN, the algorithm selects either a random action, or forwards the input through a neural network, depending on a probability epsilon (usually referred to as the exploration probability). Then, the program takes in input in the form of the current state, action taken, reward received, and the next state that occurred. This input is added to a buffer of experiences. Then, after the iteration of the program/training session is complete, a sample of the inputs from earlier are taken and are passed through to the neural network as training data.



Form of a DQN agent. [9]

PPO

PPO is a lot harder to understand. The algorithm employs two networks, an "actor" network to output which action is the most promising in the future, and a "critic" network that predicts the reward for the agent at the end of this episode. For the training process, PPO uses the following process:



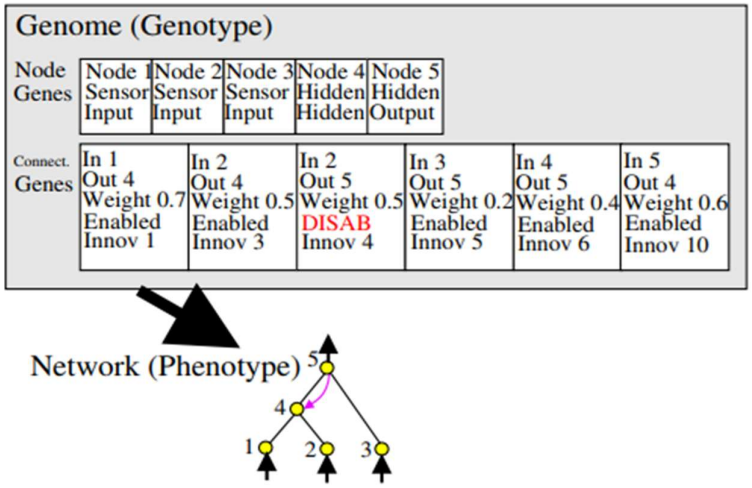
Phase 1 and 2 of the PPO training system. [10]

Phase 1 is covered in the start of the program, where it specifies to collect a set of trajectories, rewards-to-go and advantage estimates using the two networks. Phase 2 is comprised of the processes of fitting the value function and updating the policy. Updating the policy (the "actor" network) is recommended to be done using stochastic gradient ascent with Adam in order to update the program less every epoch, meaning that the program will not have drastic changes in policy per epoch. This also allows the program to be applied for multiple agents, as then the larger number of agents will not drastically affect the policy, as that will often lead to the agent taking the wrong actions in different situations (as we have encouraged that action excessively). Note that stochastic gradient descent refers to the usage of

a given equation to model the update of the parameters of the network alongside the regular backpropagation algorithm.

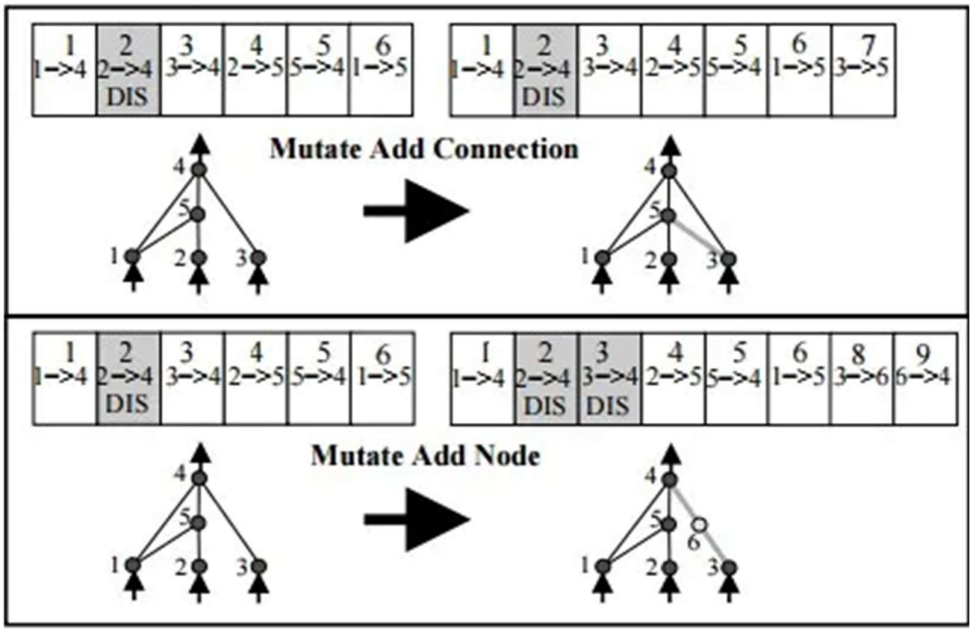
NEAT

The fourth algorithm is not particularly a reinforcement learning algorithm. Instead, it is an evolutionary strategy inspired by nature, with the implementation of genes, a genome and a phenotype.



The genotype and phenotype of a NEAT agent. [11]

The genes of an agent evolve throughout the training process of the agent, as each iteration a "mutation" will occur by random chance to give different agents different attributes (genotype), which can then be tested to see which yields a better result (phenotype). The diagram below shows two of the three possible results a mutation can be, with the third being the ability to change the weight of a connection. In this way, the network can change its shape and effective size depending on the problem given to it, and if the problem changes. This also means that a NEAT agent does not use a traditional neural network, but instead builds its own by instituting its own nodes, connections, and weights through mutations.



2 of the 3 different mutations that can occur to a NEAT agent. [11]

Modelling

The program will forward information once every 1/24th of a second to each agent running the program, and then allow the agent to make its input to the program. Rounds will last 90 seconds, meaning that there will be 90 x 24 = 2160 inputs and outputs per round. If NEAT is being used, one iteration will be one of a given member of the population of genotypes. One round will consist of individual sub-rounds, where each sub-round has a different member of the NEAT population playing.

The program will use Unity physics and UI operations to show the players' actions – in this way Unity merely acts as a representation of the problem that the algorithms are tasked with solving. A different problem, for example Tag using a 2D grid and directionless points rather than a 3D simulation, could be simulated and achieve the same goals. However, it would not be as aesthetically nuanced as the 3D implementation, and as such the user would not be able to see the nuances of the algorithms as clearly as they would with the chosen implementation.

Objectives of the Program

The program has the following objectives:

1. The program will allow the user to view the two players of the game, henceforth referred to as "agents" and controlled by AI.
    - 1a. The program switches between screens as required by the user input
        - 1ai. Backwards
        - 1aii. Forwards
    - 1b. The program will display information about each part of the program.
        - 1bi. Information icon available.
        - 1bii. Button spawns information panel
        - 1biii. User can move information panel around & close at will

2. The program will allow the user to change the settings of the simulation.
    - 2a. The program will allow the user to change parameters on the selected agent.
        - 2ai. Algorithm selection changes algorithm on agent.
        - 2aii. Parameter selection changes parameters on agent.
    - 2b. The program will allow the user to configure the game.
        - 2bi. Total number of rounds to be played.
        - 2bii. Automatic round pauses after a given number of rounds.
        - 2biii. Round interval
        - 2biv. Directory of stats output.
    - 2c. Reset all fields when user requires
    - 2d. The program will only start the program once the settings are all valid
    - 2e. The program should tell the user when an input is invalid

3. The program will allow a given agent to use the DQN algorithm
    - 3a. Receive input from program.
    - 3b. Store an "episode"
        - 3bi. Previous state
        - 3bii. Action taken
        - 3biii. Next state
        - 3biv. Reward received
    - 3c. Take action
        - 3ci. Exploration probability
        - 3cii. Neural network action
        - 3ciii. Random action

3d. Update agent between iterations
- 3di. Select random batch of episodes
- 3dii. Train neural network using episodes
- 3diii. Update exploration probability

4. The program will allow the agent to use the PPO algorithm
   - 4a. Receive input from the program.
   - 4b. Store an "episode"
     - 4bi. Previous state
     - 4bii. Action taken
     - 4biii. Next state
     - 4biv. Reward received
   - 4c. Take action
     - 4ci. Compute action using actor network
     - 4cii. Sample action from multivariate normal distribution
     - 4ciii. Compute log probability of action
   - 4d. Update agent between iterations
     - 4di. Calculate advantage estimates using baseline value (Generalised Advantage Estimate)
     - 4dii. Optimise actor network using clipped objective function
     - 4diii. Update critic network to minimise the value loss

5. The program will allow the agent to use the NEAT algorithm.
   - 5a. Initialise population of neural network genomes
   - 5b. For each genome, take inputs and give a required output.
     - 5bi. Store fitness
     - 5bii. Take action
       1. Action from neural network
   - 5c. Organise Genomes into Species
   - 5d. Update the agent between iterations
     - 5di. Apply genetic operations to create new offspring
       1. Mutations
          - 5a. Add new node
          - 5b. Add new connection
          - 5c. Change edge weight
          - 5d. Change edge bias
       2. Crossover
          - 5a. Select a subset of neural networks to become the next generation
          - 5b. Combine individual properties of multiple networks into one for the next generation

6. The program will perform neural network functions where the algorithm needs them.
   - 6a. Take an input and return an output
     - 6ai. Network Level:
       1. Forward input through each layer
     - 6aii. Layer Level:
       1. Output calculation
       2. Activation function
   - 6b. Train the neural network using a given data set of inputs, expected outputs, and real outputs
     - 6bi. Network Level:
       1. Calculate output
       2. Calculate output gradient

3. Update learning rate
4. Backpropagate loss through each layer

6bii. Layer Level:

1. Apply derivative of activation function
2. Calculate and clip derivatives of weights and biases
3. Calculate the derivative of the inputs
4. Update weights and biases using learning rate and derivatives
5. Update weights and biases using Adam
6. Return derivative of inputs for next layer to use

6c. Create a set of activation functions for the program to use, one set for neural networks and one set for the NEAT implementation

6ci. Neural Networks

1. ReLU
2. LReLU
3. Softmax

6cii. NEAT

1. ReLU
2. LReLU
3. Softmax
4. Sigmoid
5. Tanh

7. The program will run the rounds of the game without user input, as per their configuration.
   - 7a. The program will manage the agents.
     - 7ai. Inputs are passed along to the agents.
     - 7aii. Outputs from the agents are replicated in the game.
     - 7aiii. Agents learn at the end of each round
   - 7b. The program will manage a round of the game.
     - 7bi. Round ends when chaser touches runner.
     - 7bii. Round ends when timer ends.
   - 7c. The program will run another round.
     - 7ci. Another round starts after one ends.
   - 7d. The program will stop the rounds upon configured intervals.
     - 7di. After the round number reaches one that satisfies the condition given by the user, the program pauses the next round.
     - 7dii. Display a prompt to allow the user to continue the rounds.
     - 7diii. Allow the user to continue the rounds.
   - 7e. The program will manage the NEAT agents each getting a turn to play a round.

8. The program will collect data about the game, draw graphs & export to .csv file.
   - 8a. The program will collect data about the game as per each round.
     - 8ai. When a round ends, information about the round is recorded in a .csv file.
   - 8b. The information collected about the last round will be displayed on screen.
   - 8c. The program will visualise the data at the end of the game.
     - 8ci. The program will draw graphs of a given statistic and time.
     - 8cii. The user will be able to switch between each graph
   - 8d. The program will output the statistics in .csv format to a given directory.

Design

This section articulates how key features of the system are to be implemented. including both code design and UI.

System Overview

The system being developed will enable users to configure and run a realistic simulation in which two agents compete in a game of tag, utilising a variety of machine learning algorithms, with a particular focus on reinforcement learning. In this game, one agent will act as the “tagger” and another as the “runner”. The user will have the ability to assign each of the given agents one of four algorithms: a Deep Q-Network (DQN), Proximal Policy Optimization (PPO), and NeuroEvolution of Augmenting Topologies (NEAT). These are widely used and well-documented in machine learning research, each offering unique approaches in decision making and strategy development. The user will be able to configure these agents further, as many of the algorithms employ hyperparameters to control how the algorithm learns. In order to assist users in learning how these algorithms work, comprehensive documentation will be provided to explain their mechanics and how they function within the context of the simulation. Each configurable element will have something attached that allows a user to quickly read a blurb about its effects on the agent. This makes the system accessible to users with varying degrees of experience with such algorithms, allowing them to understand the concepts alongside observing the algorithms in action.

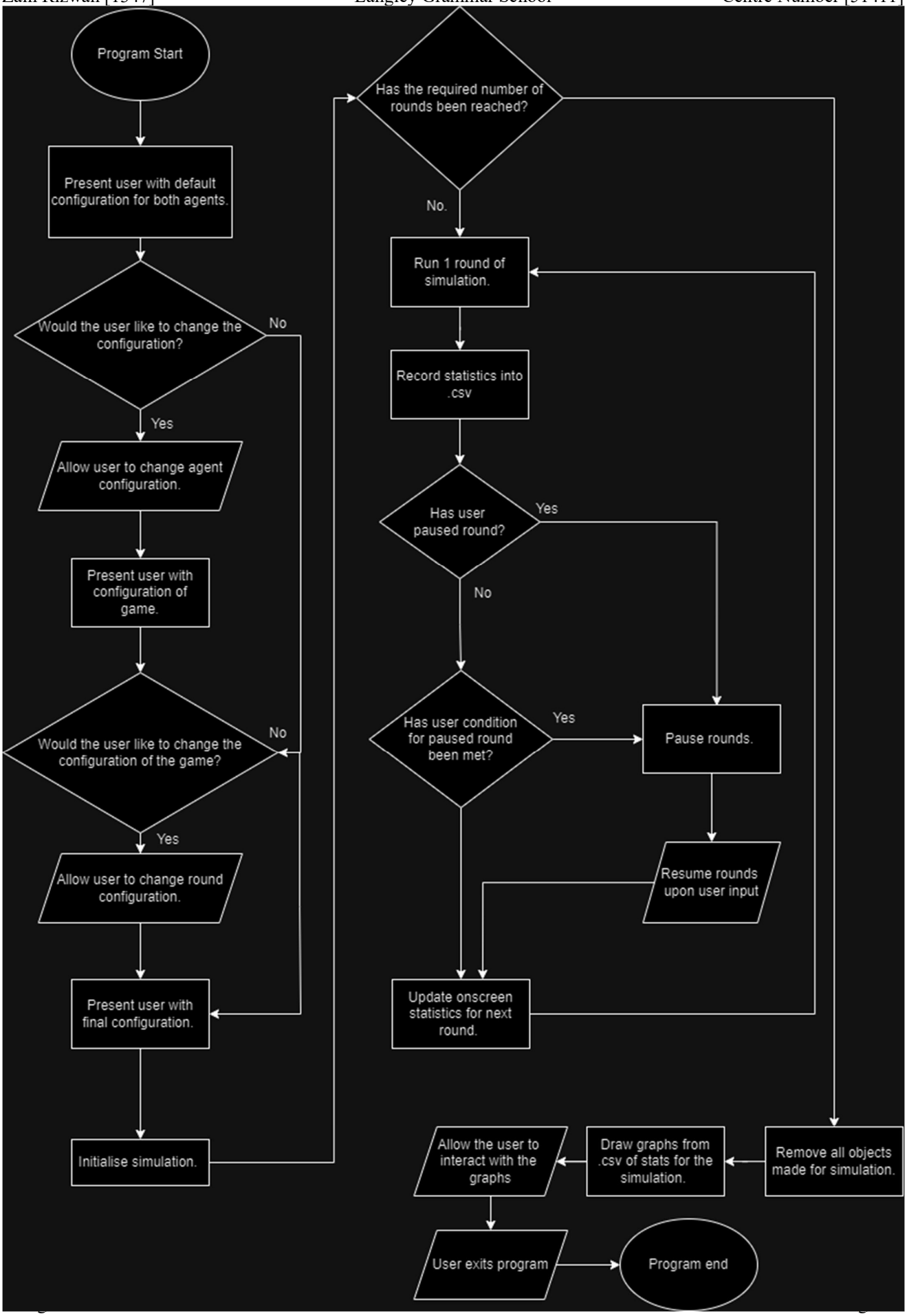
Once the agents are configured, the program will simulate the game of tag, where the tagger and runner will interact with a Unity environment based on the selected algorithms and parameters. The simulation will provide this environment such that users can set the program to pause after a given iteration to observe the strategies and progress of the agents, offering valuable insights into how their chosen algorithm impact behaviour in real time. Additionally, users can set the program to pause automatically ager a specified number of iterations, such as every 1,000 steps, making it easier for users to track progress periodically without needing to monitor the simulation continuously, Throughout the simulation, the program will display detailed statistics, both including cumulative data from the entire simulation and, current performance indicators. The program will also keep track of win/loss rates and loss function data over time.

Once the program has finished running the simulation, it will generate a comprehensive set of graphs on the performance and behaviour of the agents throughout the game. These will detail the progression of each agents’ progress in its competition against its opponent in the form of several different statistics, offering insights into how different algorithms perform under various conditions. Users will be able to export this data in a .csv format for further analysis, making it compatible with external software for deeper exploration. However, the program itself will also be able to graph the data, allowing users to visualise important trends and patterns directly within the system. This makes the program more friendly to use for those who are not familiar with handling and analysing .csv data, further reinforcing the goal of the program being allowing less experienced users to learn the actual effects of these algorithms in a given environment.

Flowcharts/Diagrams

High-Level System Flowchart

This flowchart covers the overall usage pattern of the program in natural language, or at a very high-level. The user first opens the program, after which they are presented with the current configuration of the two agents. Then, they are able to change them within the constraints of the program, with information being readily available as to what they are actually changing. Then, they move on to the game configuration stage, where they are shown a screen allowing them to configure how the game functions, how many iterations there are in the game, after how many iterations to pause the game, and where to output statistics. Then, the program will run the simulation for the given number of iterations, pausing the next round if the user has input, or pausing the next round if it has reached the required threshold to be paused at. Then, once the program has received an input to resume the simulation, it should continue and run another round. This continues until the given number of iterations completes, and the statistics from the program are displayed on screen. The program will have to draw multiple graphs, allow the user to interact with them, and also output the data to the given directory.


Program Start
Present user with default configuration for both agents.
Would the user like to change the configuration?
No
Yes
Allow user to change agent configuration.
Present user with configuration of game.
Would the user like to change the configuration of the game?
No
Yes
Allow user to change round configuration.
Present user with final configuration.
Initialise simulation.
Has the required number of rounds been reached?
No.
Run 1 round of simulation.
Record statistics into .csv
Has user paused round?
Yes
No
Has user condition for paused round been met?
Yes
Pause rounds.
Resume rounds upon user input
Update onscreen statistics for next round.
Remove all objects made for simulation.
Draw graphs from .csv of stats for the simulation.
Allow the user to interact with the graphs
User exits program
Program end

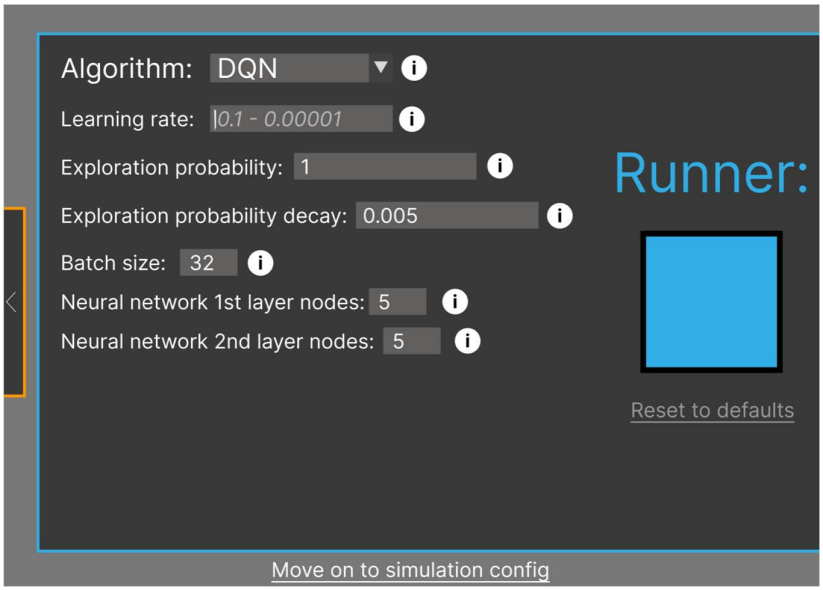
Human Computer Interface

This section will cover the Human Computer Interface of the program, which refers to the on-screen interface between the system and the user. This is especially important for the proposed system as the system puts a large emphasis on the accessibility of information about the given parts. All wireframes/mock-ups are made in Figma, a widely used graphic design software. These are in 2D, but keep in mind that the final product should be in 3D.

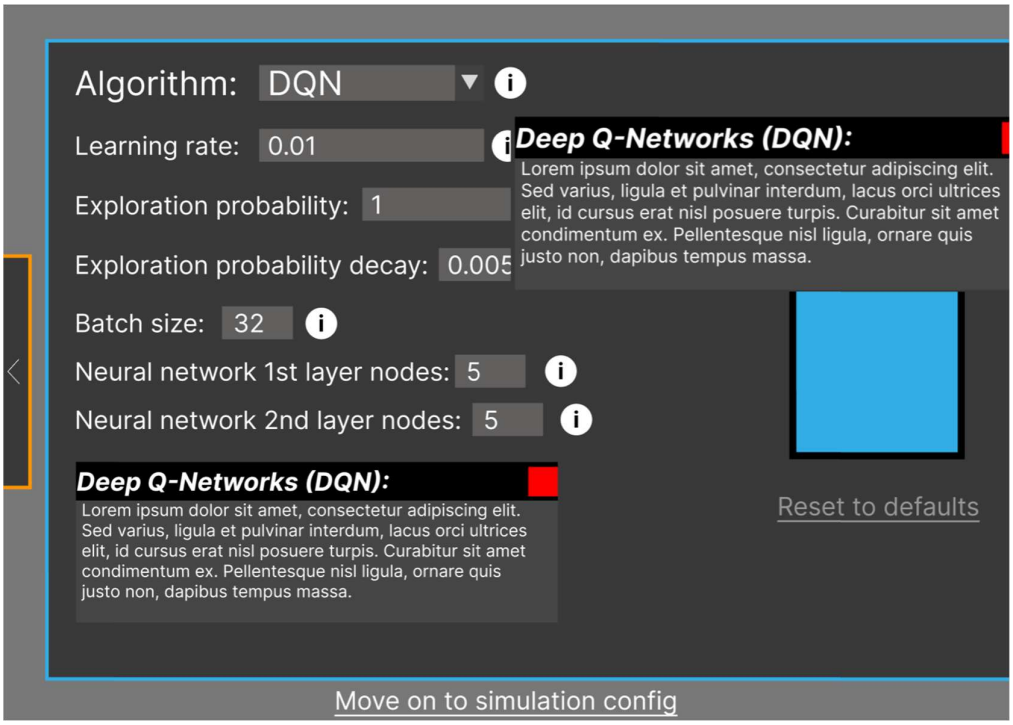
Agent and Simulation Configuration Screens

[Objectives 1, 2]

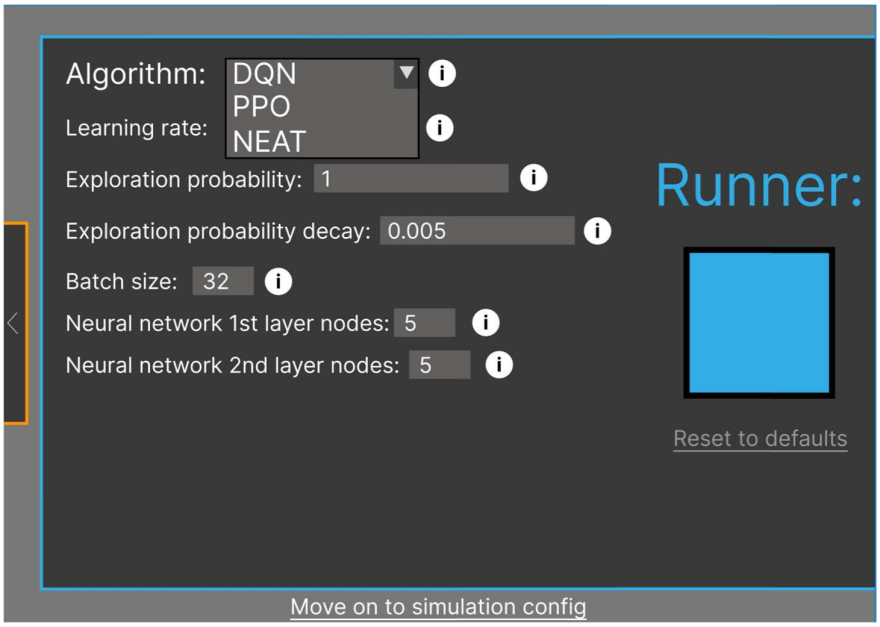
The first screen will be the configuration for the runner, followed by the tagger. They will both have similar looking interfaces, with different accent colours. The selected algorithm's different parameters will be shown for the user to adjust, along with information icons that can be clicked to show more information about a given parameter. This is shown in the below image.



When the aforementioned information icon is clicked, it should spawn a small window containing the term and its definition, that the user can drag around and delete when needed, as shown here



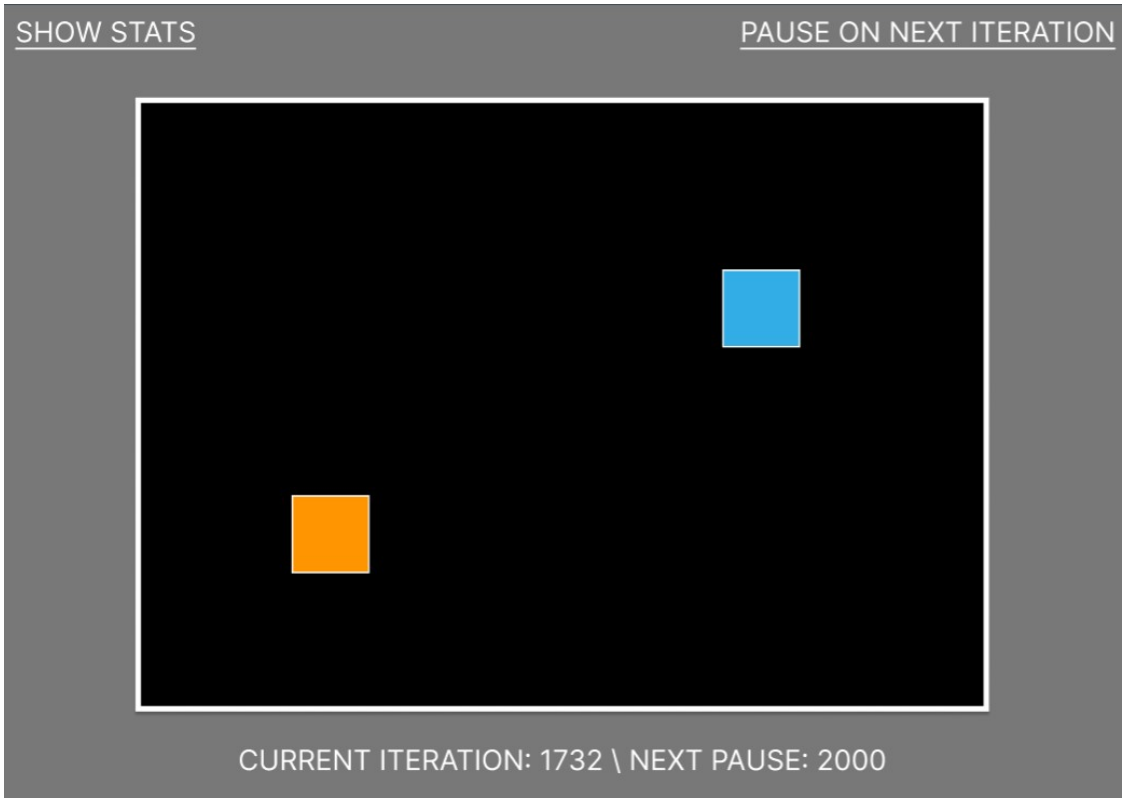
And furthermore, the user should be able to select different algorithms using the Algorithm dropdown.



Simulation Screens

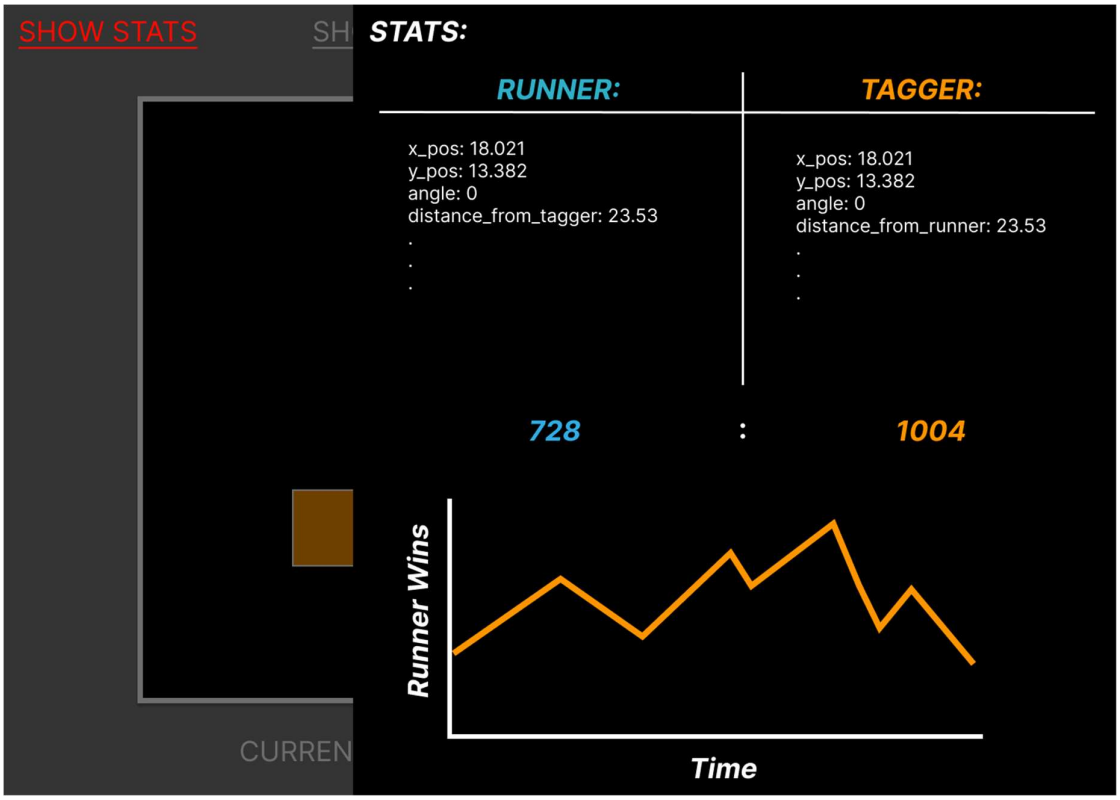
[Objectives 3, 4, 5, 6, 7, 8]

After the user presses the “START SIMULATION” button on this screen, the program enters the simulation, showing the user the current landscape of the game of tag. The following screen shows the simulation after an arbitrary number of iterations of the simulation.

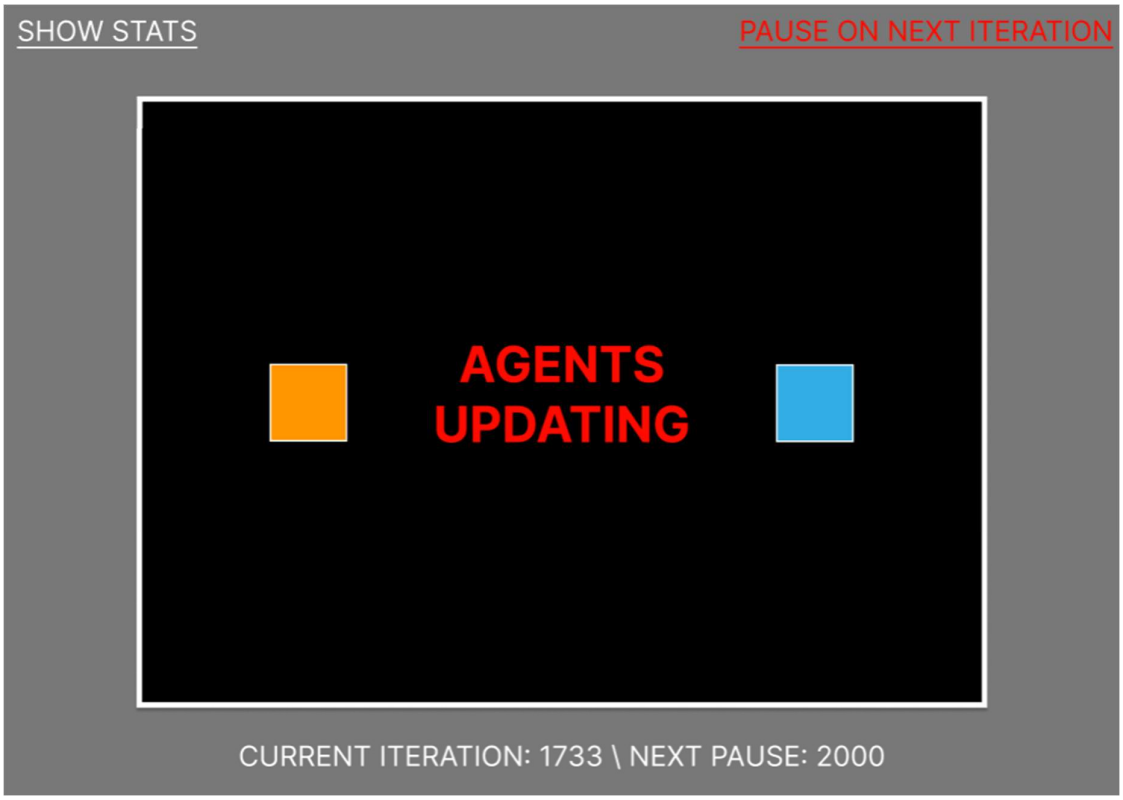


The user can pause the simulation upon the next iteration using the button in the top left. When this happens, the following screen will appear when the next iteration starts.

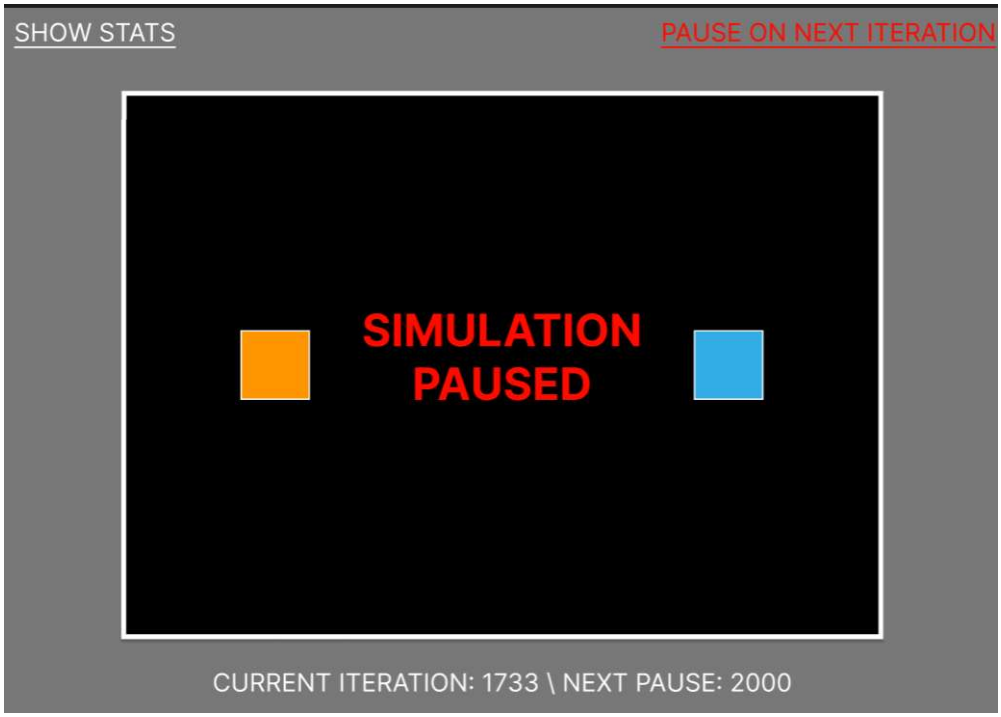
Using the other “show stats” button at the top of the screen should yield this:



And in between rounds, when the agents are training, the program should show this:



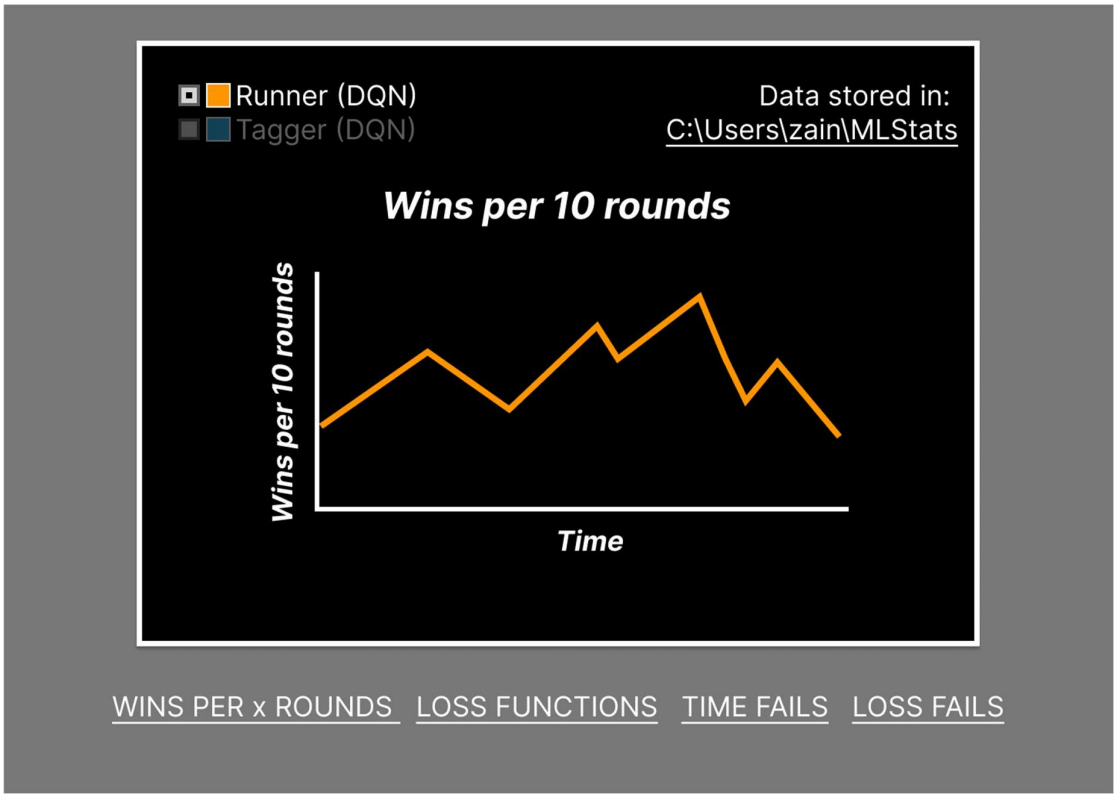
And finally, if the current round has ben paused, the simulation should show this:



Post-Simulation Result Screens

[Objective 9]

After the simulation is over, the program will show the statistics for the whole session, and display a message to the user informing them that the data is in their specified directory.



Neural Network

[Objective 6]

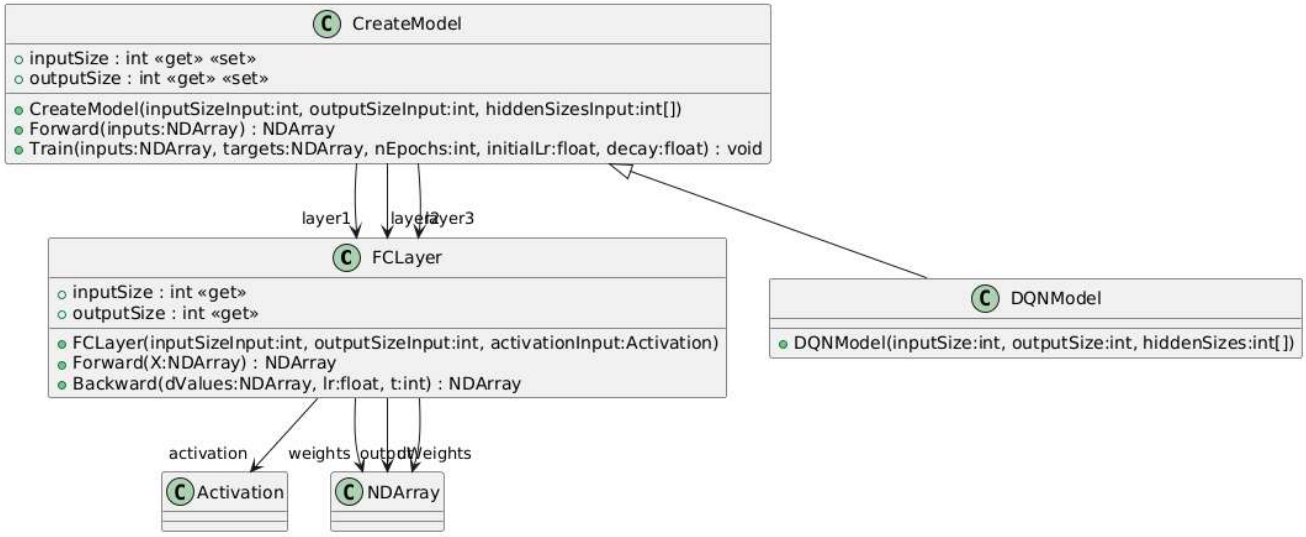
The structure of the neural network comes from a guide [7]. This defines a class for a layer fully connected to that before and after it in the program, and another for a model that combines these layers into a fully connected neural network. The neural network is primarily used in the **DQN** implementation, as the "model" referred to in its processes.

Why didn't you use this neural network in your PPO implementation?

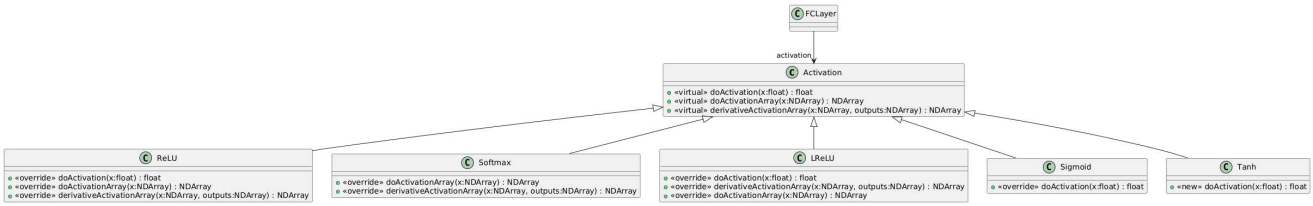
The neural network does not implement the backwards gradient function that PPO utilises to perform its calculations. PPO, when implemented, calculates losses in such a way that there is no clear input, output, or target values for the neural network to use in its training function as implemented in the guide. As such, other methods were used to create the networks for PPO.

File Structure

The file should include both the FCLayer class for the fully connected layer, and the CreateModel class as a parent class for the DQNModel to be created.



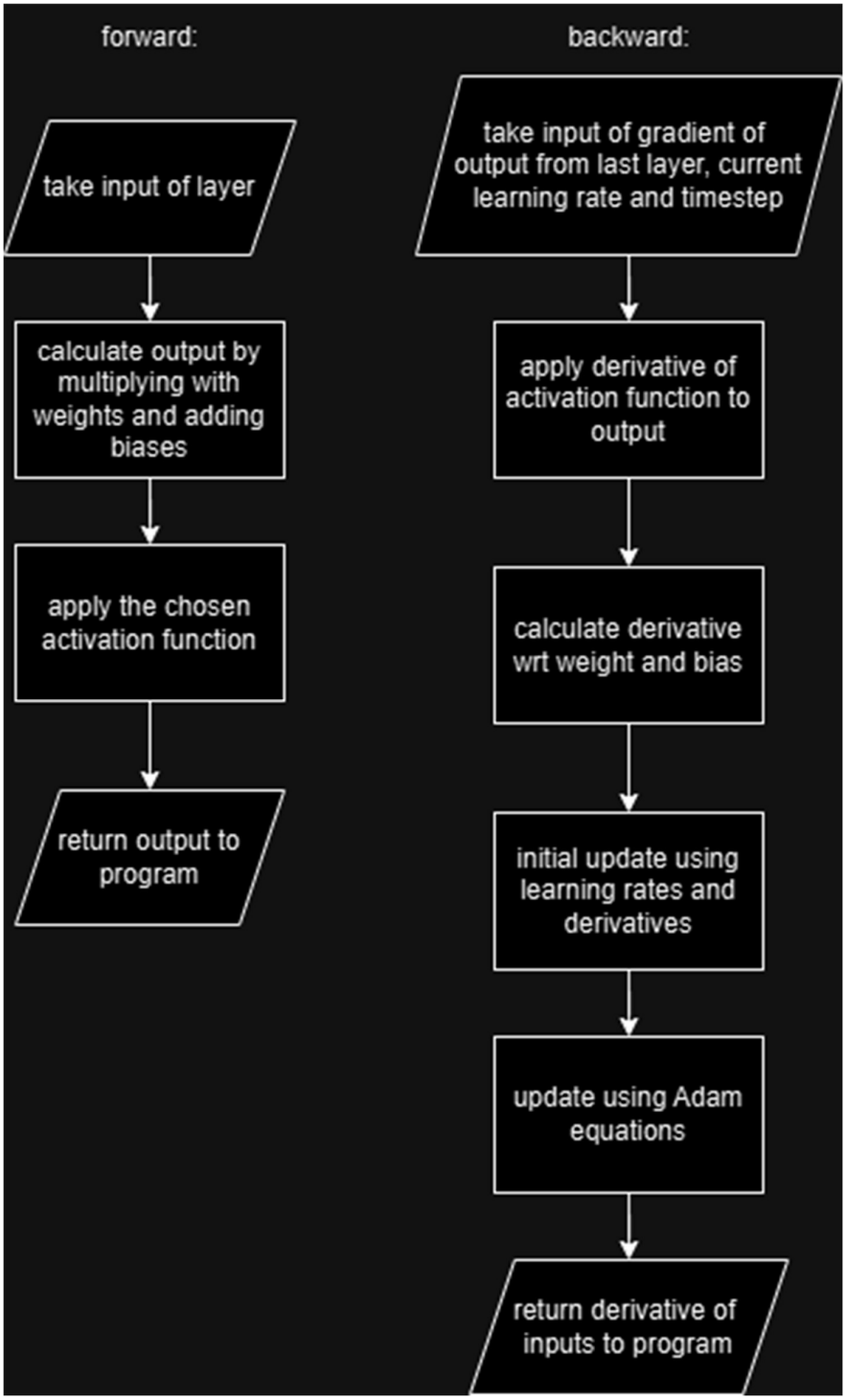
Note that the FCLayer classes refers to another class, the Activation class. This class holds the various activation functions that can be used within not only this class but also the NEAT class for its implementation of a neural network. The structure of the activation classes is shown in the image below.



Note that some activation classes have all three of the original parent class's methods implemented, whereas others only have one. This is because they are meant only for use in the NEAT file, whereas the other three are suitable for use in both the class for the neural network and the class for the layer (FCLayer).

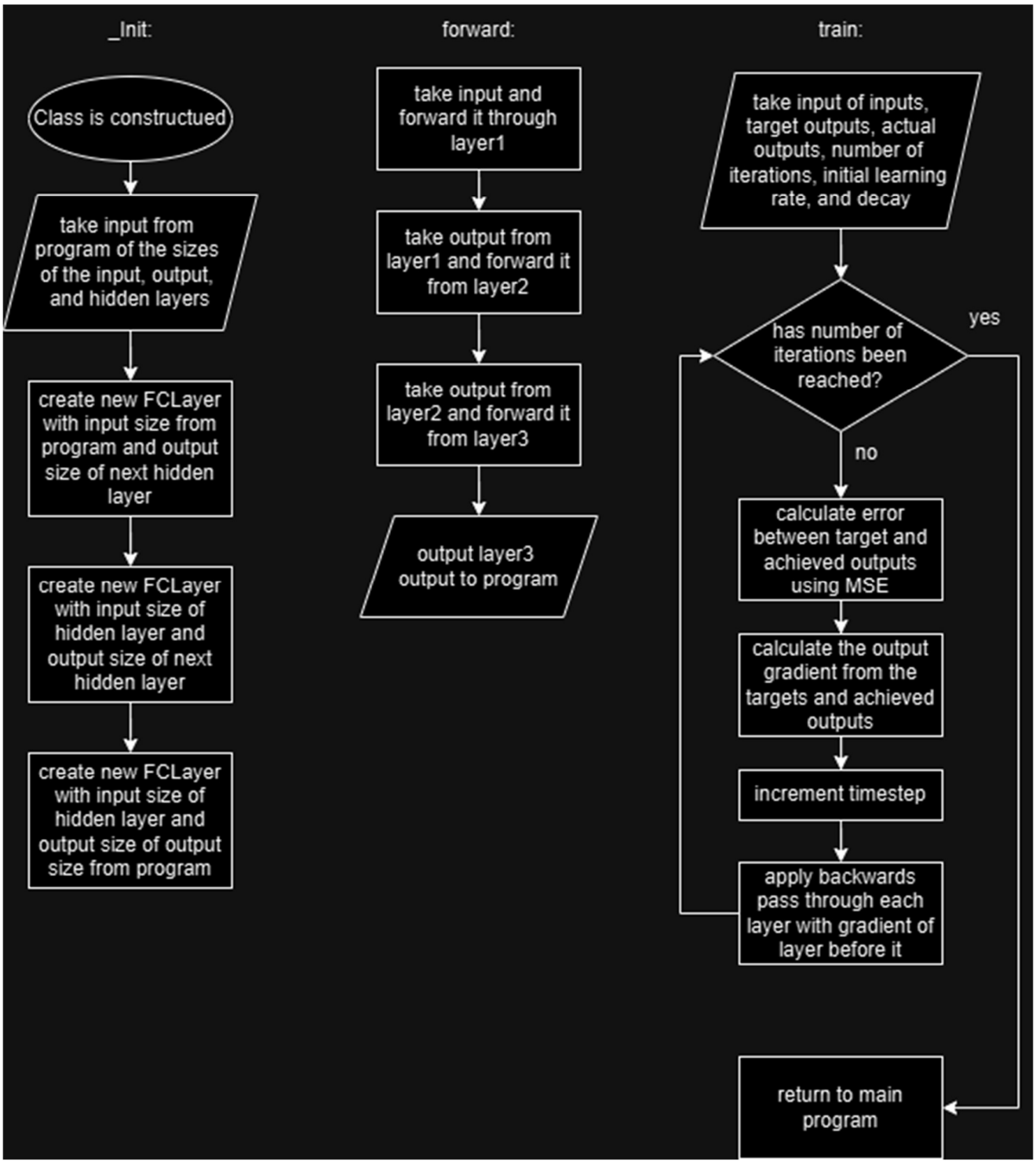
FCLayer Class

The FCLayer Class creates a layer of the network, taking in a given number of network or program inputs and outputting the values for the next layer or the output layer after applying the activation function. This class also takes a set of inputs, outputs, along with information about the neural network to readjust the weights in a backwards pass. When the class is instantiated, it creates all its necessary NDArrays, as well as initialises the weights of the neural network. These functions are shown in the flowcharts below.



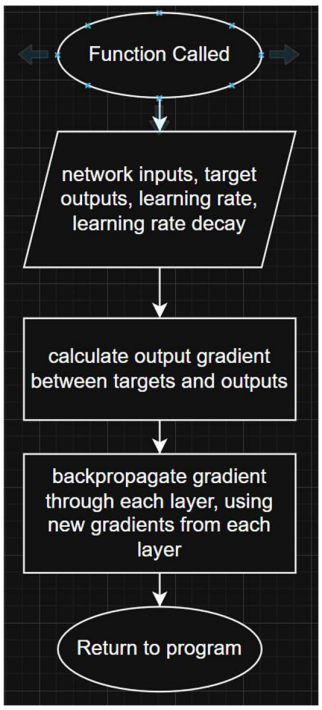
CreateModel Class

The CreateModel Class uses the FCLayer Class to create a neural network of 3 layers. It takes in an input of a given input size, output size, layer weights, and other neural network parameters to create the network. It performs a forward pass through the network by taking an input, passing it through the first layer, then passing that through the next layer, and so on until the output is reached. It also has a training function, taking in a set of inputs, outputs, expected outputs, and other training parameters. It then calculates the loss for each of the outputs and backpropagates this error through the network, by taking its gradient, then using the backwards function for each layer until every layer has had the error backpropagated through it. The different functions of the neural network are shown in the flowcharts below.



DQNModel Class

This is a child class of CreateModel. It is created primarily to modify the original learning function of the program, as DQN does not use the losses, instead using their gradients . Also, DQN only performs one backwards pass through the network each time it calls the function, so the structure of the Train function changes. The new train function works as follows:



Hyperparameters

One of the main aims of the program is to allow the user to edit the settings of each algorithm. These are called Hyperparameters, and as such each algorithm has a unique set of hyperparameters. Therefore, there needs to be accomodations for different inputs for the same algorithm, to ensure that the program stays modular to an extent.

Hyperparameters Class

To achieve the above goal of allowing more algorithms to potentially be added to the program, a base class should be added, with attributes that every preceding class needs to have. The following are in the Hyperparameters class:

Hyperparameter	Data Type	Purpose
obsDim	Integer	The observation dimensions of the simulation – how many values are passed as an observation.
actDim	Integer	The action dimensions of the simulation – how many actions the agent can take.
activationThreshhold	Float	The minimum value an agent output must take for the action to be sent to the simulation.

DQNHyperparameters

Hyperparameter	Data Type	Purpose
Learning rate (lr)	Float	The learning rate to be used by the network each time it updates.
Gamma	Float	The discount factor used in Bellman-Optimality
explorationProbInit	Float	The starting value for the agent's exploration probability.
explorationProbDecay	Float	The decay value for the Epsilon-Greedy Algorithm that updates the exploration probability.
hiddenLayersSizes	Integer Array	The sizes of both of the agents' networks' hidden layers.
batchSize	Integer	The size of the batch used to train the network.
memoryBufferSize	Integer	The size of the memory buffer that the network keeps.

PPOHyperparameters

Hyperparameter	Data Type	Purpose
maxTimestepsPerEpisodes	Integer	Maximum number of timesteps per episode of the program being run
nUpdatesPerIteration	Integer	Number of times to update networks per iteration
Learning rate (lr)	Float	Learning rate of optimisers
Gamma	Float	Discount factor to be applied when calculating rewards-to-go
Lambda (lam)	Float	Parameter for Generalised Advantage Estimation
Clip	Float	Used to define threshold to clip ratio during calculation of surrogate loss
numMinibatches	Integer	Number of minibatches for minibatch update
Entropy Coefficient (entCoef)	Float	Entropy coefficient for entropy regularisation
Maximum Gradient Normal (maxGradNorm)	Float	Gradient clipping threshold
actorNetworkSizes	Integer Array	Size of hidden layers for actor network
criticNetworkSizes	Integer Array	Size of hidden layers for critic network
totalTimesteps	Integer	Timesteps of the program so far
maxEpisodesInBatch	Integer	The maximum episodes that can be in the batch at any given time

NEATHyperparameters

Hyperparameter	Data Type	Purpose
deltaThreshold	Float	Determines the threshold for measuring the compatibility of genomes. If the distance between two genomes exceeds this, they are considered separate species.
maximumFitnessHistory	Integer	Number of past fitness values to record before removing.
populationSize	Integer	Defines the number of individual genomes in the population for each generation.
defaultActivation	Activation	Specifies the default activation function for neurons in the neural network.
distanceWeights	Dictionary (string, float)	Contains weight factors for different components when calculation genome distance.
breedProbabilites	Dictionary (string, float)	Contains probabilities for different reproduction methods for genomes (asexual/sexual)
mutationProbabilities	Dictionary (string, float)	Contains probabilities for various mutations occurring in a genome.

Algorithm Implementation

DQN

[Objective 3]

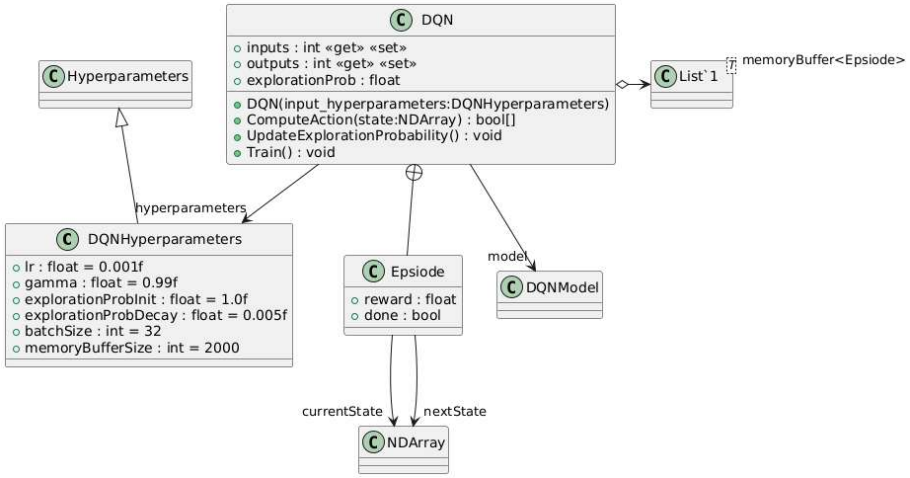
This implementation of DQN is designed such that a program could query an action from the algorithm, and trigger its learning at a given time when required. Based off of [12]

Pseudocode

Algorithm 1: deep Q-learning with experience replay.
Initialize replay memory D to capacity N
Initialize action-value function Q with random weights θ
Initialize target action-value function $\hat{Q}$ with weights $\theta^{-} = \theta$
For episode = 1, M **do**
  Initialize sequence $s_1 = \{x_1\}$ and preprocessed sequence $\phi_1 = \phi(s_1)$
  For t = 1,T **do**
    With probability ε select a random action a_t
    otherwise select $a_t = \text{argmax}_a Q(\phi(s_t), a; \theta)$
    Execute action a_t in emulator and observe reward r_t and image x_{t+1}
    Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$
    Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in D
    Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from D
    Set $y_j = \begin{cases} r_j & \text{if episode terminates at step j+1} \\ r_j + \gamma \max_{a'} \hat{Q}(\phi_{j+1}, a'; \theta^{-}) & \text{otherwise} \end{cases}$
    Perform a gradient descent step on $(y_j - Q(\phi_j, a_j; \theta))^2$ with respect to the network parameters θ
    Every C steps reset $\hat{Q} = Q$
  End For
End For

In more simple terms, the algorithm selects either a random action, or forwards the input through a neural network, depending on a probability epsilon (usually referred to as the exploration probability). Then, the program takes in input in the form of the current state, action taken, reward received, and the next state that occurred. This input is added to a buffer of experiences. Then, after the iteration of the program/training session is complete, a sample of the inputs from earlier are taken and are passed through to the neural network as training data.

Structure of DQN file



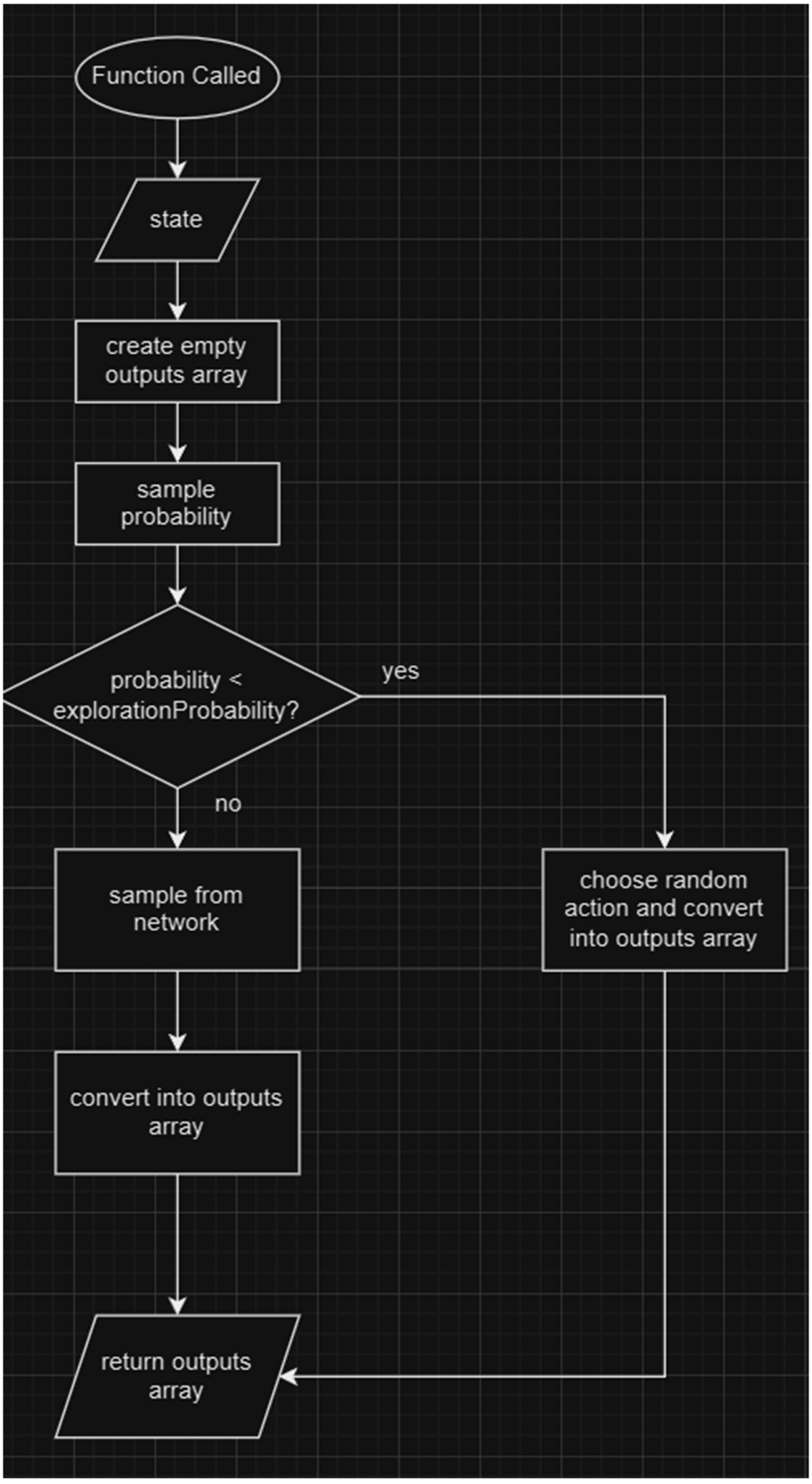
From this, the following are key algorithms, classes, and subroutines required to implement DQN.

Episode

For DQN, an episode is defined as a single timestep of the simulation. As such, the Episode class stores the reward, if the simulation is done or not, the current and next states of the algorithm, and the action taken by the algorithm.

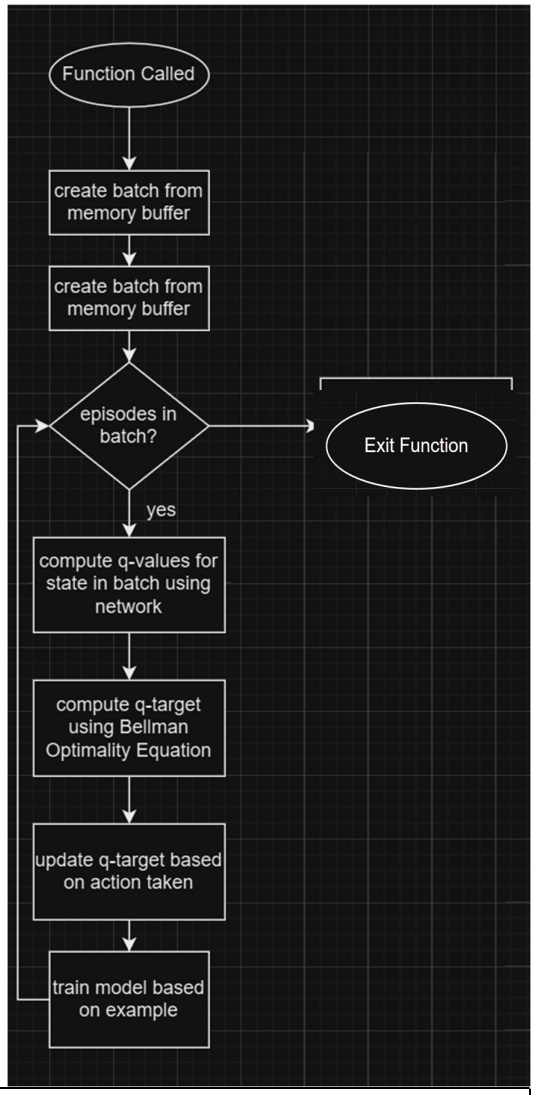
ComputeAction

This function takes an input of a state, and returns the algorithm's decided outputs.



Train

This function takes no input, but it assumes that the buffer of memories in the agent has been updated for each new experience that the neural network has gone through. It selects a batch of memories, and performs actions upon them to prepare them to be used to train the neural network.




```
SUBROUTINE Train()
        Inds ← Array of indices FROM 0 TO memoryBuffer,count - 1
        SHUFFLE inds randomly
        batchSample ← empty list
        FOR EACH i IN INDS DO
                ADD memoryBuffer[i] TO batchSample
        ENDFOR
        FOR EACH ep IN batchSample DO
                COMPUTE qValue using DQNModel
                qValueTarget ← ep.reward
                COMPUTE qValueTarget using Bellman-Optimality
                index ← 0
                i ← 0
                FOR EACH action IN ep.action DO
                        IF action = True THEN
                                index ← i
                        ENDIF
                i ← i + 1
                ENDFOR
                CALL DQNModel.Train(currentState, qValue, lr)
        ENDFOR
ENDSUBROUTINE
```

UpdateExplorationProbability

$$explorationProbability = explorationProbability * e^{-probabilityDecay}$$

This the implementation of the Epsilon – Greedy algorithm to update the exploration probability for the agent after every iteration of the program.

PPO

[Objective 4]

Pseudocode

Coded using [13], [14], [15], [16], [17], [18]

Algorithm 1 PPO-Clip

1: Input: initial policy parameters θ_0, initial value function parameters ϕ_0
2: **for** $k = 0, 1, 2, \ldots$ **do**
3: Collect set of trajectories $\mathcal{D}_k = \{\tau_i\}$ by running policy $\pi_k = \pi(\theta_k)$ in the environment.
4: Compute rewards-to-go $\hat{R}_t$.
5: Compute advantage estimates, $\hat{A}_t$ (using any method of advantage estimation) based on the current value function V_{ϕ_k}.
6: Update the policy by maximizing the PPO-Clip objective:

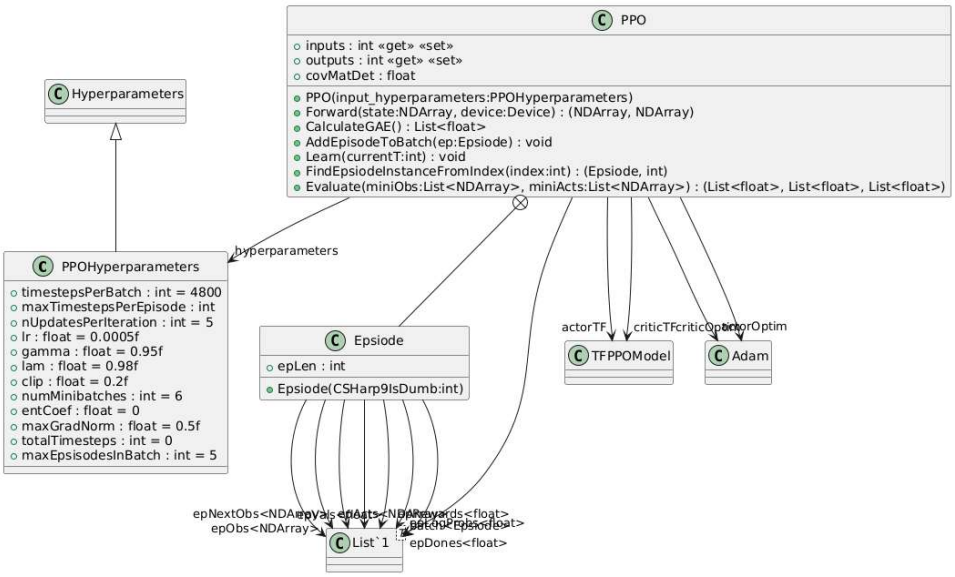
$$\theta_{k+1} = \arg\max_{\theta} \frac{1}{|\mathcal{D}_k|T} \sum_{\tau \in \mathcal{D}_k} \sum_{t=0}^{T} \min\left(\frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_k}(a_t|s_t)} A^{\pi_{\theta_k}}(s_t, a_t),\ \ g(\epsilon, A^{\pi_{\theta_k}}(s_t, a_t)) \right),$$

typically via stochastic gradient ascent with Adam.
7: Fit value function by regression on mean-squared error:

$$\phi_{k+1} = \arg\min_{\phi} \frac{1}{|\mathcal{D}_k|T} \sum_{\tau \in \mathcal{D}_k} \sum_{t=0}^{T} \left(V_\phi(s_t) - \hat{R}_t \right)^2,$$

typically via some gradient descent algorithm.
8: **end for**

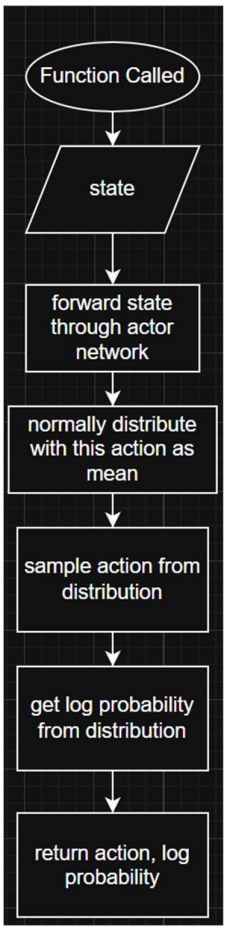
Structure of PPO File



From this, the following are key algorithms, classes, and subroutines required to implement PPO.

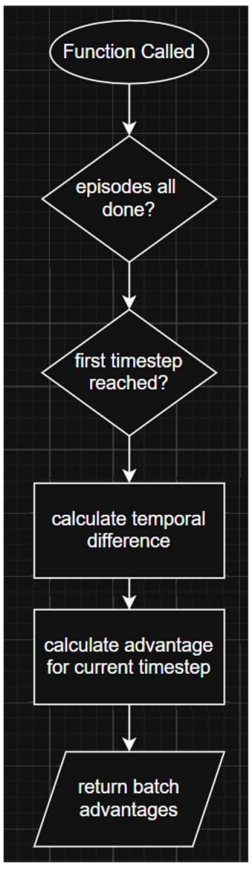
Forward

This function takes the current state from the algorithm's simulation, and forwards it through the actor network . This sampled action is then used as the mean for a normal distribution, and another action is sampled from that. The log probability of this action is also queried from the distribution, before both are returned to the program.



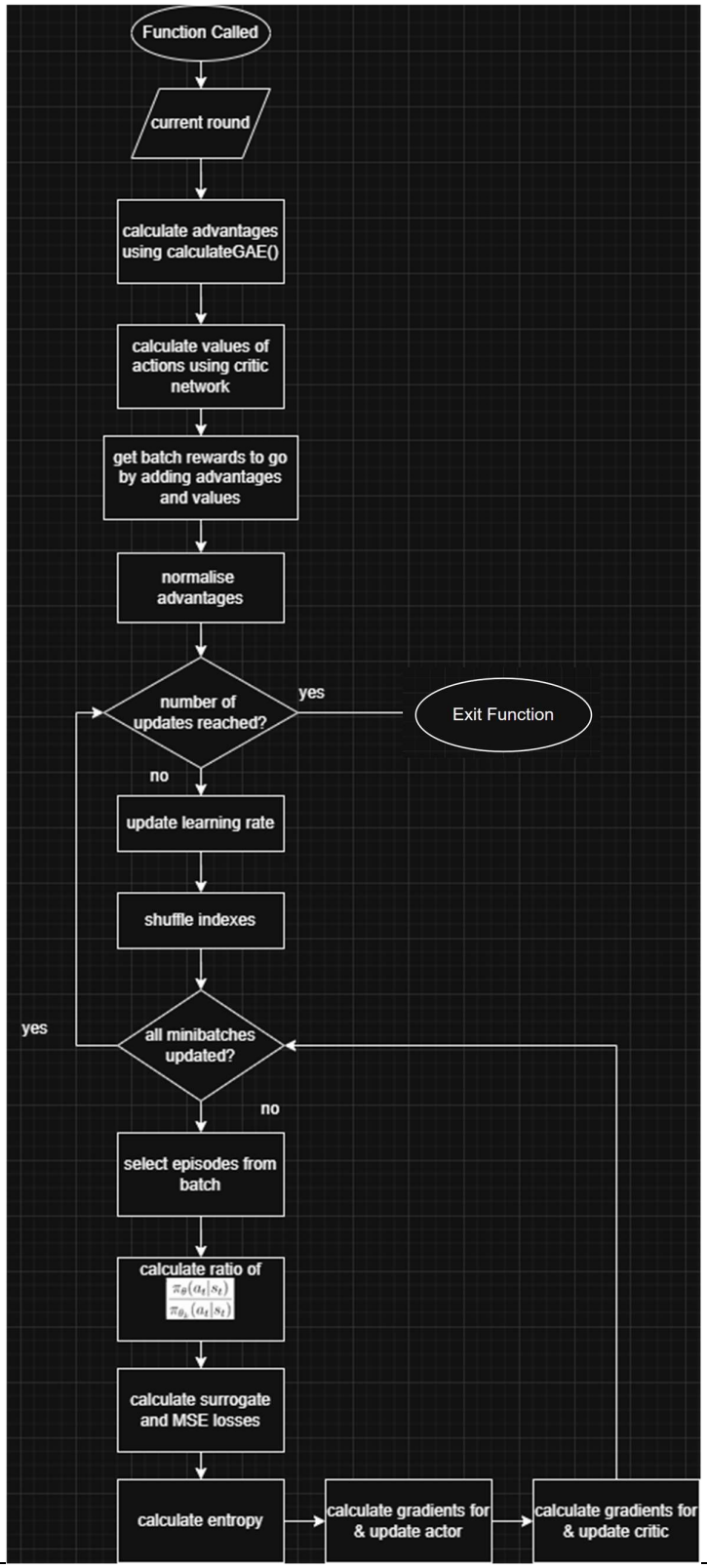
CalculateGAE

This function takes each episode from the batch, and computes its temporal difference. Then it uses these differences to compute Generalised Advantage Estimation.



Learn

This subroutine also assumes that the memory buffer has been updated in between rounds. It takes the current round number as an input. Calculates GAE and other values for the whole batch, and then trains networks for a given number of updates using minibatches.



SUBROUTINE Learn(currentT)

```
    // Calculate advantage using GAE
    A_k ← CalculateGAE()

    // Compute value function estimates
    V ← empty list
    FOR EACH episode IN batch DO
        FOR EACH obs IN episode.epObs DO
            ADD criticTF.forward(obs) TO V
        ENDFOR
    ENDFOR

    // Compute batch return-to-go (RTG)
    batchRTGs ← empty list
    FOR i FROM 0 TO V.Count - 1 DO
        ADD (A_k[i] + V[i]) TO batchRTGs
    ENDFOR

    // Normalize advantages
    sigx2 ← 0
    sigX ← 0
    n ← A_k.Count
    FOR i FROM 0 TO n - 1 DO
        sigx2 ← sigx2 + (A_k[i])²
        sigX ← sigX + A_k[i]
    ENDFOR
    A_k_mean ← sigX / n
    A_k_std ← sqrt((sigx2 - n * (A_k_mean)²) / (n - 1 + 1e-10))

    FOR i FROM 0 TO A_k.Count - 1 DO
        A_k[i] ← (A_k[i] - A_k_mean) / (A_k_std + 1e-10)
    ENDFOR

    // Compute total steps in batch
    step ← 0
    FOR EACH episode IN batch DO
        step ← step + episode.epLen
    ENDFOR

    // Update network for a given number of epochs
    inds ← Array of indices FROM 0 TO step - 1
    minibatchSize ← step / hyperparameters.numMinibatches

    FOR i FROM 0 TO hyperparameters.nUpdatesPerIteration - 1 DO
        // Learning rate annealing
        frac ← (currentT - 1) / hyperparameters.totalTimesteps
        newLr ← hyperparameters.lr * (1 - frac)
        newLr ← max(newLr, 0)

        // Set learning rates
        actorOptim ← Adam(actorTF.parameters(), newLr)
        criticOptim ← Adam(criticTF.parameters(), newLr)

        // Shuffle indices
        SHUFFLE inds randomly

        // Mini-batch updates
        FOR j FROM 0 TO step - 1 STEP minibatchSize DO
            minibatchObs ← empty list
```

```
minibatchActs ← empty list
minibatchLogProbs ← empty list
minibatchAdvantage ← empty list
minibatchRTGs ← empty list

// Define indexes for minibatch
end ← j + minibatchSize
idx ← empty list
FOR k FROM j TO end - 1 DO
   ADD inds[k] TO idx
ENDFOR

// Extract data from selected indices
FOR EACH k IN inds DO
   (miniEp, index) ← FindEpsiodeInstanceFromIndex(k)
   ADD miniEp.epObs[index] TO minibatchObs
   ADD miniEp.epActs[index] TO minibatchActs
   ADD miniEp.epLogProbs[index] TO minibatchLogProbs
   ADD A_k[k] TO minibatchAdvantage
   ADD batchRTGs[k] TO minibatchRTGs
ENDFOR

// Compute V_phi, log probabilities, and entropy
(V, currentLogProbs, entropy) ← Evaluate(minibatchObs, minibatchActs)

// Compute policy ratios
logRatios ← empty list
ratios ← empty list
approxKlList ← empty list
FOR k FROM 0 TO currentLogProbs.Count - 1 DO
   logRatios[k] ← currentLogProbs[k] - minibatchLogProbs[k]
   ratios[k] ← exp(logRatios[k])
   approxKlList[k] ← ratios[k] - 1 - logRatios[k]
ENDFOR

// Compute surrogate losses
surr1 ← empty list
surr2 ← empty list
FOR k FROM 0 TO ratios.Count - 1 DO
   surr1[k] ← ratios[k] * minibatchAdvantage[k]
   surr2[k] ← clamp(ratios[k], 1 - hyperparameters.clip, 1 + hyperparameters.clip) * minibatchAdvantage[k]
ENDFOR
actorLosses ← (-min(surr1, surr2)).mean()

// Compute entropy loss
entropyLoss ← mean(entropy)
actorLosses ← actorLosses - hyperparameters.entCoef * entropyLoss

// Update actor network
actorOptim.zero_grad()
actorLosses.backward()
clip_grad_norm_(actorTF.parameters(), hyperparameters.maxGradNorm)
actorOptim.step()

// Compute critic loss
cumMSE ← 0
FOR k FROM 0 TO V.Count - 1 DO
   cumMSE ← cumMSE + (V[k] - minibatchRTGs[k])²
```

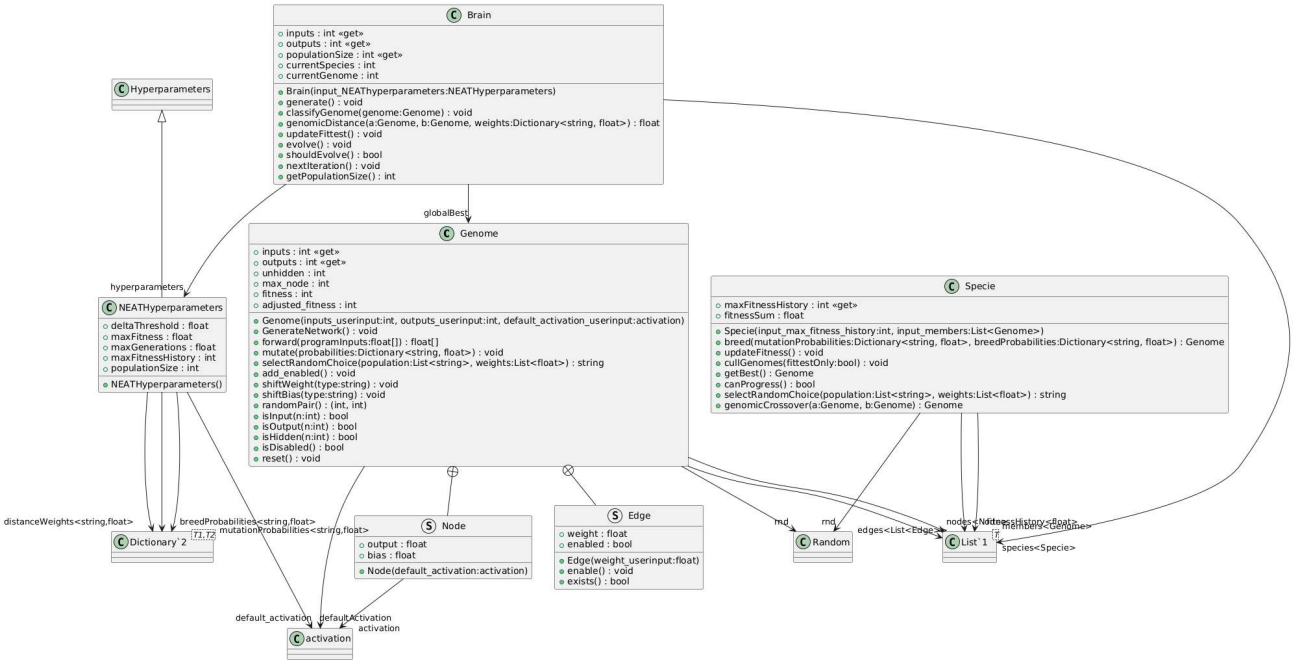
```
        ENDFOR
        criticLosses ← (cumMSE / V.Count)

        // Update critic network
        criticOptim.zero_grad()
        criticLosses.backward()
        clip_grad_norm_(criticTF.parameters(), hyperparameters.maxGradNorm)
        criticOptim.step()
    ENDFOR
  ENDFOR
ENDSUBROUTINE
```

NEAT

[Objective 5]

As NEAT has no official pseudocode, instead opting to have the paper [11] describing the methods used. As such, implementation will be derived from this implementation in Python [19] and the research.

Structure of NEAT File



The Brain contains Species, which contain Genomes. As each of these three classes contain different methods, they will be split into separate sections henceforth. Note that the Genome class refrences the Activation class, which refers to the same set of activations that are used in the neural network's class.

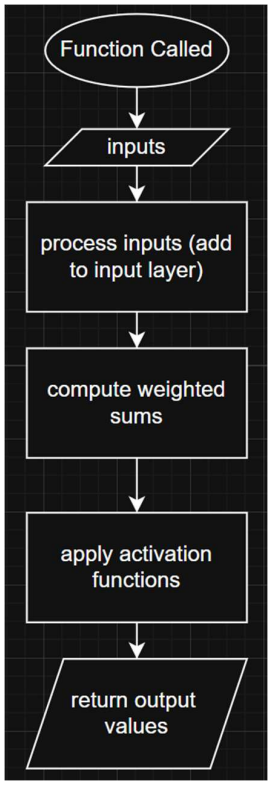
From this, the following are key algorithms, classes, and subroutines required to implement NEAT.

Genome

This class defines a single neural network within a species of the population. The following methods are the most important, with others being defined purely used to make code more readable and understandable.

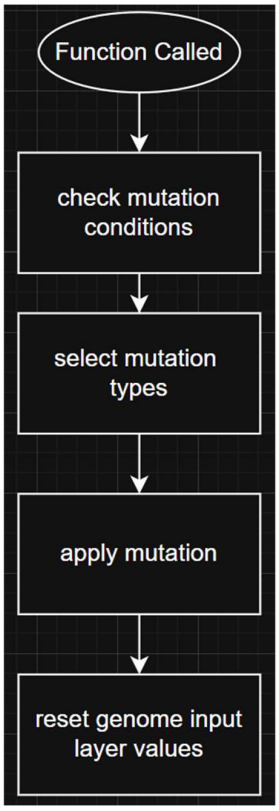
Forward

This function takes an input from the program, and forwards it through the Genome's neural network to produce an output. Note that the activation function here is derived from the same activation functions as those used in the neural network file.



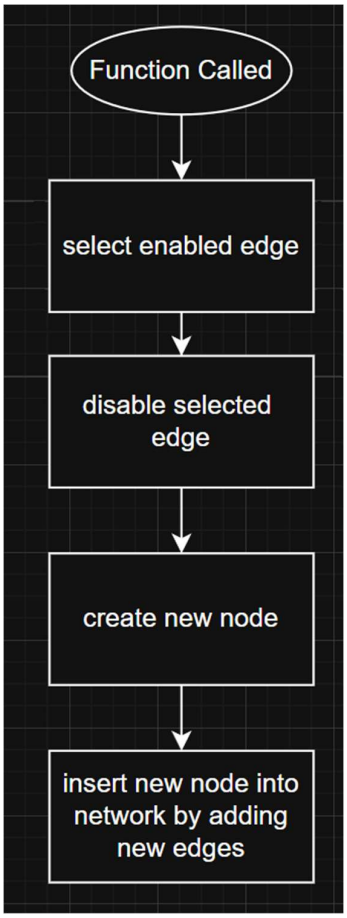
Mutate

This function checks mutation conditions as specified by the probabilities in the hyperparameters, and then applies the given mutation. After this, it resets the output values of every node in the neural network.



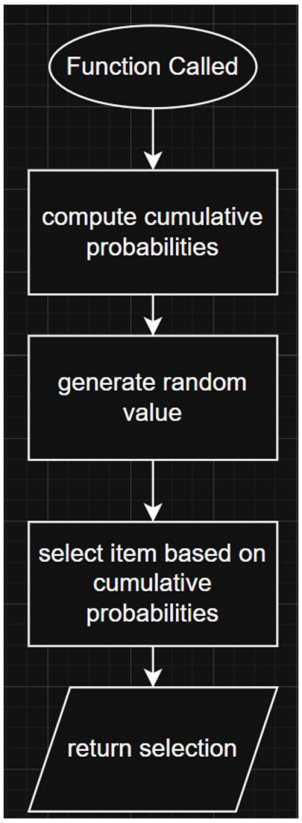
AddNode

This function is used to improve organisation of the program. It edits the sets of nodes and edges to fit a new node onto the connection of two other nodes by disabling that connection and establishing two new ones.



SelectRandomChoice

The select random choice function takes in a given set of probabilities and options, and selects one random choice. It does this by first computing the cumulative probabilities of the options given. Then, a random value is generated between 0 and the maximum of the cumulative probabilities, and the class that the value falls into is selected as the function's selected choice. Then, this is returned to the program

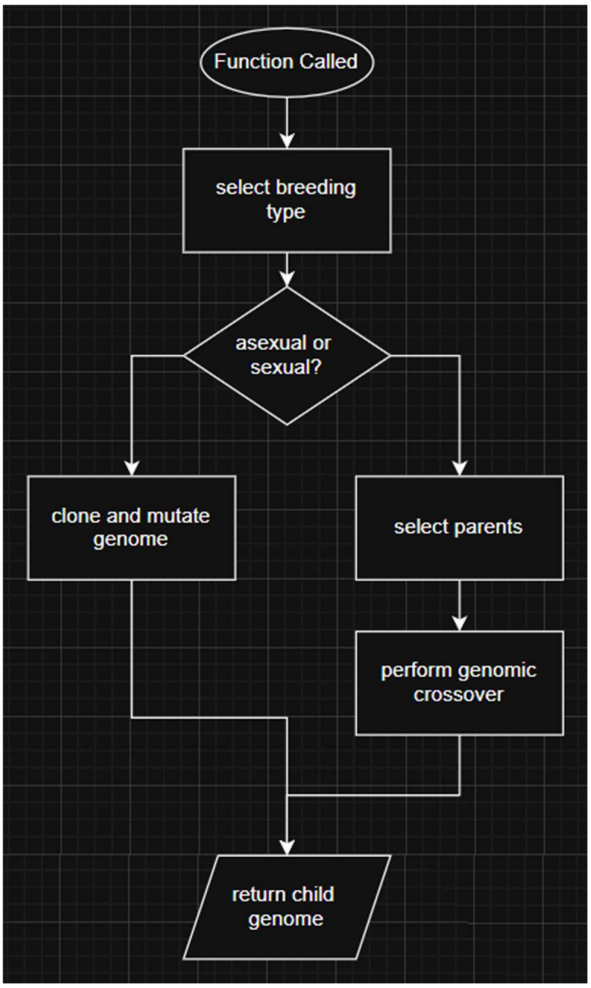


Specie

This class represents a group of genomes that have been deemed similar to each other based on the Genomic Distance algorithm and the delta threshold as set by the user.

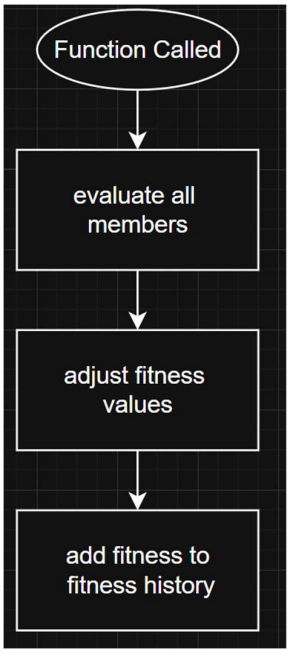
Breed

This function selects the breeding type using the input breed probabilities and the SelectRandomChoice function. Then, depending on its choice, it will either select 2 parents and undergo Genomic Crossover to perform sexual breeding, or clone and mutate the genome to undergo asexual breeding. After either of these two processes, the child genome is returned as an output to the program.



UpdateFitness

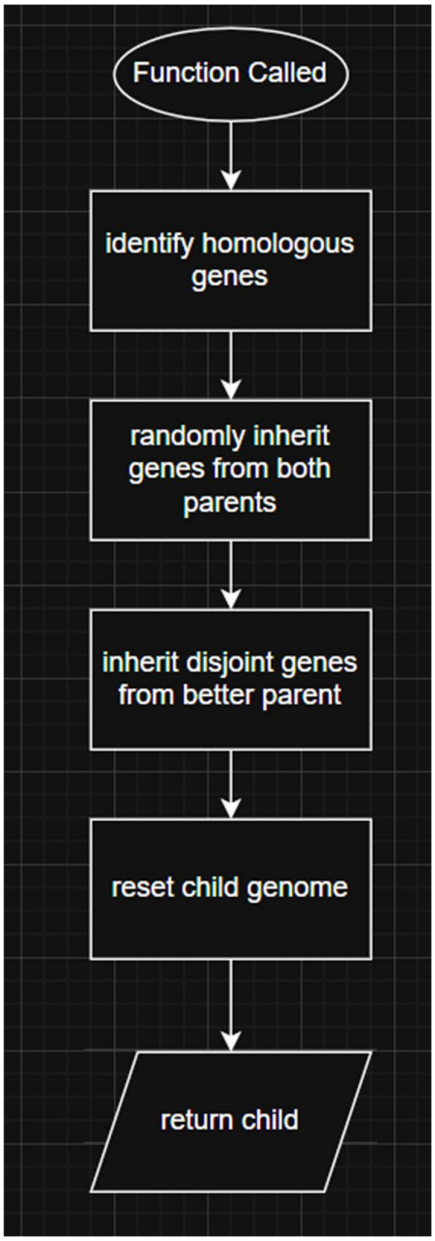
This function updates the fitness values of every genome within a given species and add it to the record of the fitness history. This fitness value is adjusted to fit the size of the population, allowing the program to compare the fitness scores of different populations.



GenomicCrossover

```
SUBROUNTINE GenomicCrossover(a, b)
    // Create child genome with same structure as parent a
    child ← Genome(a.inputs, a.outputs, a.default_activation)

    // Determine which parent has more edges
    aIn ← IF a.edges.Count > b.edges.Count THEN a.edges ELSE
b.edges
    bIn ← IF a.edges.Count > b.edges.Count THEN b.edges ELSE
a.edges

    combinedIns ← empty list

    // Create list of homologous genes
    FOR i FROM 0 TO aIn.Count - 1 DO
        ADD empty list TO combinedIns
        tempList ← IF aIn[i].Count < bIn[i].Count THEN aIn[i] ELSE
bIn[i]
        FOR j FROM 0 TO tempList.Count - 1 DO
            ADD (aIn[i][j], bIn[i][j]) TO combinedIns[i]
        ENDFOR
    ENDFOR

    // Child randomly inherits homologous genes from either parent
    FOR i FROM 0 TO combinedIns.Count - 1 DO
        edgePairList ← combinedIns[i]
        FOR EACH edgePair IN edgePairList DO
            childEdge ← IF rnd.Next(1) = 0 THEN edgePair.Item1 ELSE
edgePair.Item2
            ADD childEdge TO child.edges[i]
        ENDFOR
    ENDFOR

    // Determine fitter parent
    fitterParent ← IF a.fitness > b.fitness THEN a ELSE b

    // Inherit disjoint genes from fitter parent
    FOR i FROM 0 TO fitterParent.edges.Count - 1 DO
        FOR j FROM 0 TO fitterParent.edges[i].Count - 1 DO
            TRY
                child.edges[i][j].exists()
            CATCH
                ADD aIn[i][j] TO child.edges[i]
            ENDTRY
        ENDFOR
    ENDFOR

    // Update maxNode value
    FOR i FROM 0 TO child.edges.Count - 1 DO
        FOR j FROM 0 TO child.edges[i].Count - 1 DO
            currentMax ← IF i > j THEN i ELSE j
            child.maxNode ← IF currentMax > child.maxNode THEN
currentMax ELSE child.maxNode
        ENDFOR
    ENDFOR
    child.maxNode ← child.maxNode + 1

    // Inherit nodes from fitter parent
```

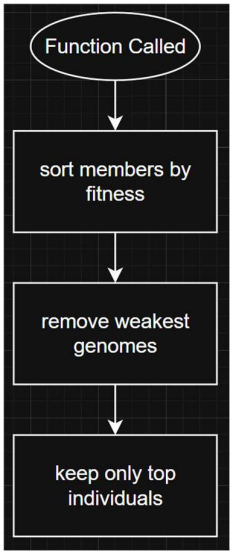
This function performs genomic crossover between two parents, and is used for sexual reproduction in the preceding Breed function. It does this by creating lists of homologous genes, with thte child randomly inheriting homologous genes from either parent, and inheriting disjoint genes from the fitter parents. The child then inherits all nodes from the fitter parents. The output of every node in the chlid is reset and its returned to the program.



```
    FOR i FROM 0 TO child.maxNode - 1 DO
        ADD fitterParent.nodes[i] TO child.nodes
    ENDFOR

    child.Reset()
    RETURN child
ENDSUBROUNTINE
```

CullGenome

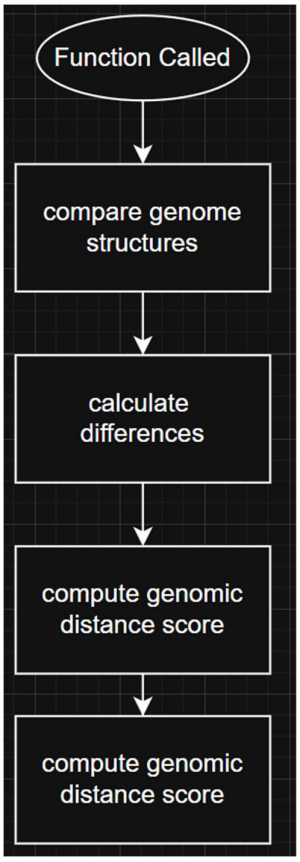
This function is responsible for reducing the number of genomes within a species by eliminating the weakest performers. This function is a key part of the evolutionary process, ensuring that less fit genomes do not progress to the next generation.



Brain

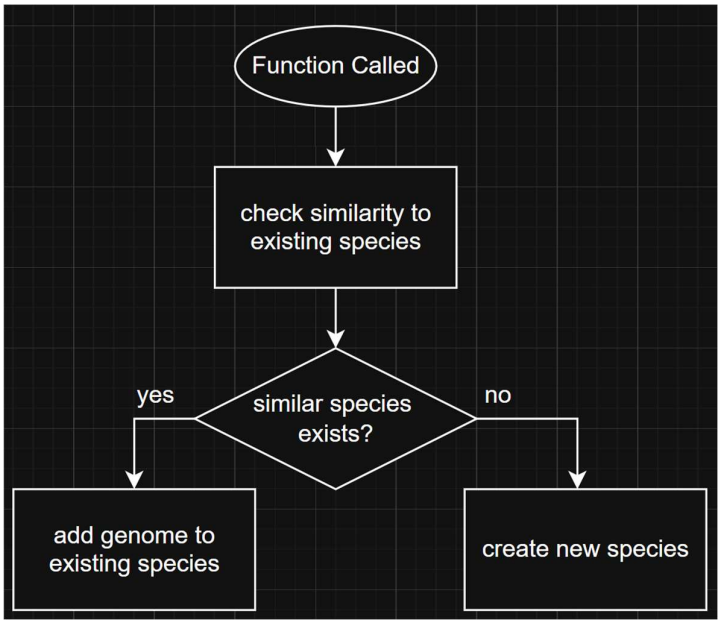
GenomicDistance

This function compares the structural and parametric differences between the genomes to determine how far apart they are from one another in terms of their genetic makeup. This is done by identifying matching and disjoint edges, calculating differences in weights and biases, and combining these differences to compute a single distance score.



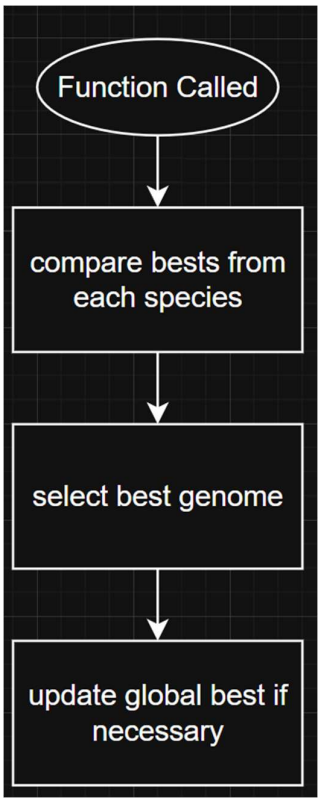
ClassifyGenome

This functions acts a dynamic classifier, assigning genomes to species based on their similarity to existing species' representatives. Genomes that are sufficiently similar to existing species are grouped together, whereas those that are different from any existing species are added to their own new species. This process is crucial to maintaining genetic diversity within the population, allowing different network topologies to evolve seperately, and preventing the entire population from converging to a single solution.



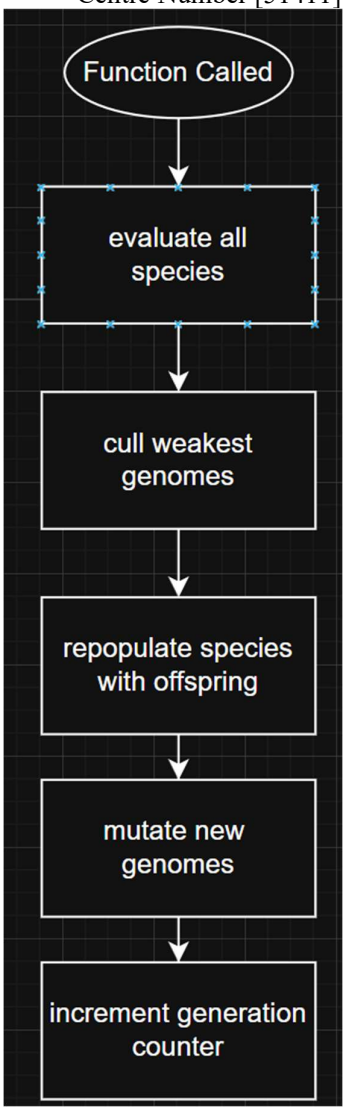
UpdateFittest

Updates the fittest example that the population currently has to offer across all of its species.



Evolve

This function is the most important part of the NEAT algorithm. It is responsible for advancing the population of genomes from one generation to the next by culling less fit genomes (CullGenome) and repopulating the species with new offspring (Breed). The evolve function is called after a whole generation of genomes have played a round of the game, through the NEATAlgorithm class. The fitness of each species is updated using UpdateFitness and the globalFitnessSum is calculated. If the sum of the global fitness is 0, all of the genomes are mutated using Mutate. Otherwise, it determines which species should survive to the next generation, and removes those that should not. The CullGenomes function is called to remove the weakest members of each surviving species. The function then calculates how many offspring should be created for each species based on its fitness relative to the global fitness, and the difference between the desired population size and the current population size. Afte this, the Breed function is used to create new offspring for each species, and ClassifyGenome is called to add the offspring to the appropriate species. If no species survive by this point, the population is repopulated with mutated copies of the global best genome, and new, minimally structured genomes. Finally, the generation counter is incremented.



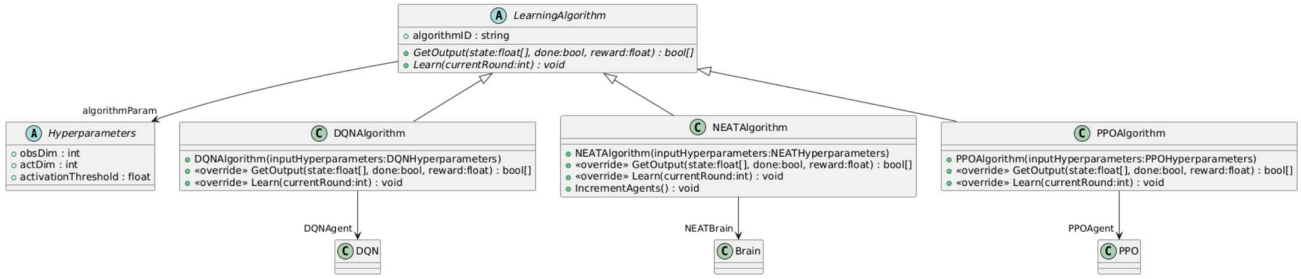

```
SUBROUTINE Evolve()
    // Reset genome and species indices for new training round
    currentGenome ← 0
    currentSpecies ← 0

    // Compute global fitness sum
    globalFitnessSum ← 0
    FOR EACH specie IN species DO
        specie.UpdateFitness()
        globalFitnessSum ← globalFitnessSum + specie.fitnessSum
    ENDFOR

    IF globalFitnessSum = 0 THEN
        // No progress, mutate all genomes
        FOR EACH specie IN species DO
            FOR EACH genome IN specie.members DO
                genome.Mutate(hyperparameters.mutationProbabilities)
            ENDFOR
        ENDFOR
    ELSE
        // Keep only species with improvement potential
        survivingSpecies ← empty list
        FOR EACH specie IN species DO
            IF specie.CanProgress() THEN
                ADD specie TO survivingSpecies
            ENDIF
        ENDFOR
        species ← survivingSpecies

        // Cull weakest genomes in each species
        FOR EACH specie IN species DO
            specie.CullGenomes(False)
        ENDFOR

        // Repopulate species with new offspring
        FOR EACH specie IN species DO
            ratio ← specie.fitnessSum / globalFitnessSum
            diff ← populationSize - GetPopulationSize()
```

```
        offspring ← ROUND(ratio * diff)
        FOR i FROM 0 TO offspring - 1 DO
          child ← specie.Breed(hyperparameters.mutationProbabilities,
hyperparameters.breedProbabilities)
          ClassifyGenome(child)
        ENDFOR
      ENDFOR

      // If no species survived, repopulate with mutations and global best
      IF species.Count = 0 THEN
        FOR i FROM 0 TO populationSize - 1 DO
          IF i % 3 = 0 THEN
            g ← globalBest
          ELSE
            g ← Genome(inputs, outputs, hyperparameters.defaultActivation)
            g.GenerateNetwork()
            g.Mutate(hyperparameters.mutationProbabilities)
            ClassifyGenome(g)
          ENDIF
        ENDFOR
      ENDIF
    ENDIF
    generation ← generation + 1
ENDSUBROUTINE
```

Learning Algorithms

Learning Algorithms Class

The LearningAlgorithm class should be a parent class that provides methods and attributes for the child classes to override. The methods should be: GetOutput(), taking in a current state, whether the program is done or not, and the reward value for the agent. And, Learn(), taking in the current round the simulation is on.



Class Diagram for LearningAlgorithm and its child classes

The purpose of a LearningAlgorithm is to take in inputs from the Unity simulation, passes it through the algorithm, and returns the output. Alongside this, the LearningAlgorithm should be able to trigger the algorithm to learn or evolve at the end of each iteration of the simulation, in order to get closer to the algorithm's goal of maximising the reward function.

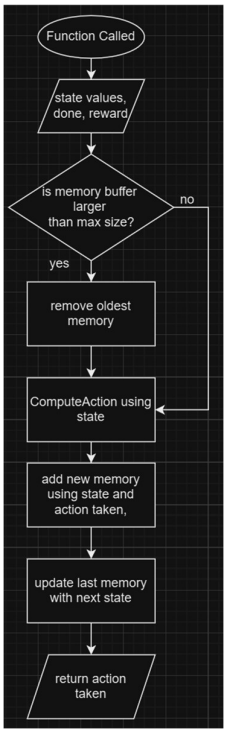
DQNAlgorithm

[Objective 3]

These three methods are not enough to create DQN on their own. For the rest of the logic required to make the DQN algorithm, the DQNAlgorithm class should be used. These implementations begin on the next page.

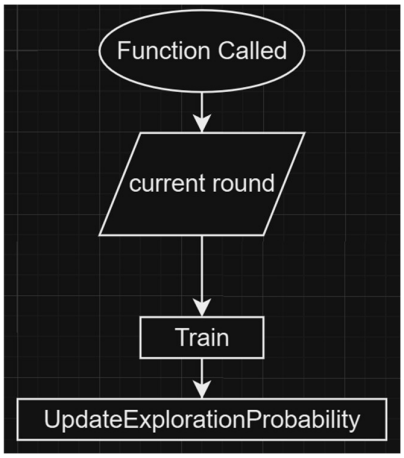
GetOutput

This function first checks if the memory buffer is larger than the defined maximum size, and if so it removes the oldest memory. Then, it computes the action using the ComputeAction function of the DQN implementation. It adds this action to the new episode and then adds it to the memory buffer. Finally, it updates the next state of the previous memory before returning the action taken to the program.



Learn

This function triggers the DQN agent to Train itself, and updates the exploration probability due to the round change.



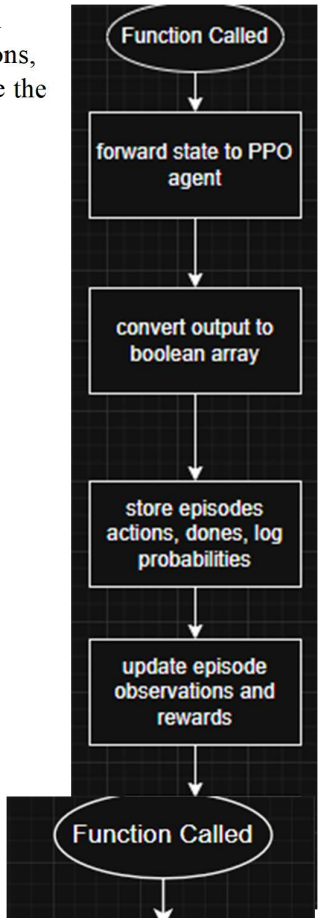
PPOAlgorithm

[Objective 4]

Similarly, the PPO algorithm also needs some extra logic in order to be added to the program.

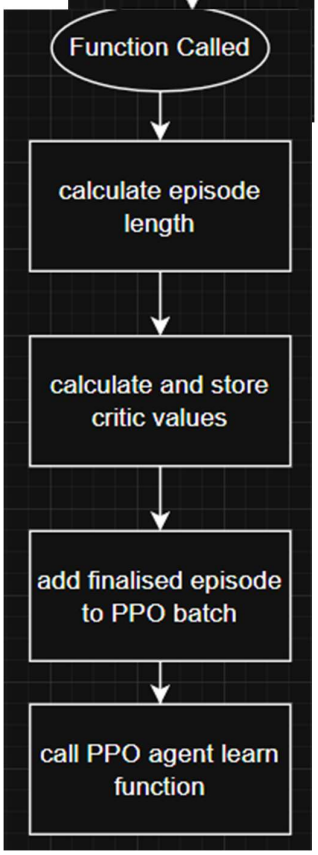
GetOutput

This function takes the input state from the simulation, forwards it through the PPO agent and converts it to an array. Then, the current episode is update with the current frame's state, actions, dones, and log probabilities. Similarly, the previous episode's observations are updated before the output is returned so it can be used.



Learn

This function calculates the length of the current episode, and stores it as part of the episode itself. The finalised episode is then added to the PPO batch, and the agent's Learn function is called.

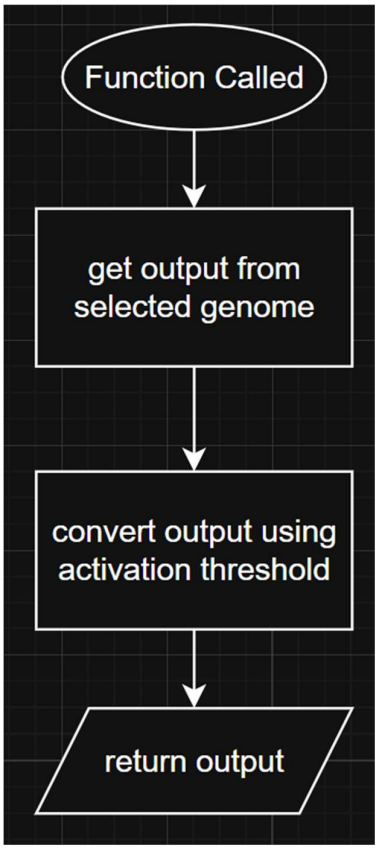


NEATAlgorithm

NEATAlgorithm is the implementation of NEAT into the LearningAlgorithms class and thus the whole program. As there is an added functionality in having to increment the current genome playing the game, the NEATAlgorithm class also includes a function to increment the current agent each iteration.

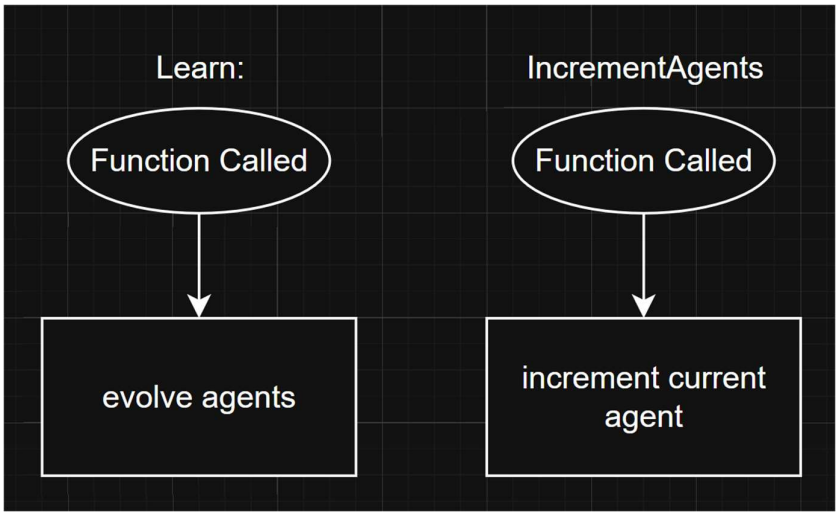
GetOutput

This function takes the current genome and gets it output for the input state to the function. Then, the output is turned into a Boolean using the activation threshold specified in the algorithm's parameters.



Learn & IncrementAgents

Both of these functions simply call their equivalents from the NEAT population's definition. The IncrementAgents function increments the agent such that if there is another agent in the specie, that is the next agent. Then, if there is no other agent in the specie, it moves onto the next specie. Then, if there are no species left, it resets both to start again from the beginning.



Unity Integration

Game Logic

[Objective 7]

The program will forward information once every 1/30th of a second to each agent running the program, and then allow the agent to make its input to the program. Rounds will last 30 seconds, meaning that there will be 30 x 30 = 900 inputs and outputs per round. If NEAT is being used, one iteration will correspond to one of the members of the NEAT population. As such, multiple iterations will be played for each round of tag, to allow for every NEAT agent to have a turn and update their fitness scores. The program will also take the input from the UI as specified below, and format it into the form of a Hyperparamter object to be used in the agent's definitions. An example of this implementation could be like this:

```
SUBROUTINE Start()
    // Set the frame rate to the target display's FPS
    Application.targetFrameRate ← 240

    // Get required components for runner and tagger objects
    runnerTransform ← GetComponent(runnerObject, Transform)
    taggerTransform ← GetComponent(taggerObject, Transform)
    runnerCollider ← GetComponent(runnerObject, Collider)
    taggerCollider ← GetComponent(taggerObject, Collider)

    // Initialize rewards
    runnerReward ← 0
    taggerReward ← 0
ENDSUBROUTINE

SUBROUTINE InitaliseSimulation(simulationParams, taggerParams, runnerParams)
    // Initialize tagger and runner hyperparameters based on parameters passed
    SWITCH taggerParams[0]
        CASE "NEAT"
            InitializeTaggerNEAT(taggerParams)
        CASE "DQN"
            InitializeTaggerDQN(taggerParams)
        CASE "PPO"
            InitializeTaggerPPO(taggerParams)
    END SWITCH

    SWITCH runnerParams[0]
        CASE "NEAT"
            InitializeRunnerNEAT(runnerParams)
        CASE "DQN"
            InitializeRunnerDQN(runnerParams)
        CASE "PPO"
            InitializeRunnerPPO(runnerParams)
    END SWITCH

    // Set simulation parameters
    totalRounds ← PARSE_INT(simulationParams[1])
    pausingEnabled ← simulationParams[4] == "true"
    pauseInterval ← PARSE_INT(simulationParams[2])
    filepath ← CONCATENATE(simulationParams[3], "stats.txt")
ENDSUBROUTINE

SUBROUTINE UpdateRound()
```

```
    // Increment the frame counter
    currentFrame ← currentFrame + 1

    // Calculate rewards
    taggerReward ← CalculateTaggerReward()
    runnerReward ← CalculateRunnerReward()

    // Get input and output for the agents
    taggerSimInput ← TakeTaggerInput()
    runnerSimInput ← TakeRunnerInput()
    taggerOutput ← taggerAlgorithm.GetOutput(taggerSimInput, False, taggerReward)
    runnerOutput ← runnerAlgorithm.GetOutput(runnerSimInput, False, runnerReward)

    // Update the simulation distance
    avgDistance ← (distance - avgDistance) / (currentFrame + 1e-10f)

    // Update stats panel if active
    IF simStatsManager.isActive THEN
        UpdateStatsPanel(taggerSimInput, runnerSimInput, taggerOutput, runnerOutput, taggerReward,
runnerReward)
    END IF

    // Check if the round is over
    IF currentFrame ≥ 30 * 30 THEN
        // Time up, runner has won
        taggerWin ← False
        runnerWin ← True
        lastRoundWinner ← "Runner"
        roundPausedText.text ← "ROUND PAUSED FOR TRAINING"
        StoreStats(taggerSubRoundsWon, runnerSubRoundsWon)
    ELSE IF runnerCollider.bounds.Intersects(taggerCollider.bounds) THEN
        // Collision detected, tagger has won
        runnerWin ← False
        taggerWin ← True
        lastRoundWinner ← "Tagger"
        roundPausedText.text ← "ROUND PAUSED FOR TRAINING"
        StoreStats(taggerSubRoundsWon, runnerSubRoundsWon)
    END IF

    // Round reset operations
    IF taggerWin OR runnerWin THEN
        ResetRound()
        currentFrame ← 0
        currentRound ← currentRound + 1
    END IF
ENDSUBROUTINE

SUBROUTINE FixedUpdate()
    distance ← CalculateDistance(taggerTransform.position, runnerTransform.position)
    maxDistance ← MAX(distance, maxDistance)

    IF NOT gameRunning THEN
        RETURN
    ELSE IF NOT isLearningDone THEN
        AgentsLearn()
        gameStatusText.text ← "ROUND RUNNING"
    ELSE
        UpdateRound()
```

```
        IF currentRound ≥ totalRounds THEN
            gameRunning ← False
        END IF
    END IF
ENDSUBROUTINE

SUBROUTINE CalculateRunnerReward() RETURNS float
    reward ← distance / maxDistance + currentFrame * 2
    RETURN reward
ENDSUBROUTINE

SUBROUTINE CalculateTaggerReward() RETURNS float
    reward ← -distance / maxDistance - currentFrame * 2
    RETURN reward
ENDSUBROUTINE

SUBROUTINE TakeRunnerInput() RETURNS float[]
    DECLARE inputs[7]
    inputs[0] ← runnerTransform.position.x
    inputs[1] ← runnerTransform.position.y
    inputs[2] ← runnerTransform.position.z
    inputs[3] ← CalculateOrientation(runnerTransform, forward)
    inputs[4] ← CalculateDistance(runnerTransform.position, taggerTransform.position)
    inputs[5] ← CalculateOrientation(runnerTransform, taggerTransform.position)
    inputs[6] ← CheckIfGrounded(runnerFoot)
    RETURN inputs
ENDSUBROUTINE

SUBROUTINE TakeTaggerInput() RETURNS float[]
    DECLARE inputs[7]
    inputs[0] ← taggerTransform.position.x
    inputs[1] ← taggerTransform.position.y
    inputs[2] ← taggerTransform.position.z
    inputs[3] ← CalculateOrientation(taggerTransform, forward)
    inputs[4] ← CalculateDistance(taggerTransform.position, runnerTransform.position)
    inputs[5] ← CalculateOrientation(taggerTransform, runnerTransform.position)
    inputs[6] ← CheckIfGrounded(taggerFoot)
    RETURN inputs
ENDSUBROUTINE

SUBROUTINE ResetRound()
    currentFrame ← 0
    runnerObject.transform.position ← runnerSpawn.position
    taggerObject.transform.position ← taggerSpawn.position
    taggerWin ← False
    runnerWin ← False
    avgDistance ← 0
    roundText.text ← FormatRoundText(currentRound)
    roundPaused ← CheckIfRoundPaused()
ENDSUBROUTINE

SUBROUTINE OutputsToProgram(movement, movingObject, footTransform)
    rb ← GetComponent(movingObject, Rigidbody)
    tf ← movingObject.transform
    isGrounded ← CheckIfGrounded(footTransform)
    IF movement[0] THEN
        rb.AddForce(tf.forward * forceMultiplier)
    END IF
```

```
    IF movement[1] THEN
      rb.AddForce(tf.forward * -forceMultiplier)
    END IF
    IF movement[2] THEN
      tf.Rotate(0, rotationMultiplier, 0)
    END IF
    IF movement[3] THEN
      tf.Rotate(0, -rotationMultiplier, 0)
    END IF
    IF movement[4] AND isGrounded THEN
      rb.AddForce(tf.up * jumpMultiplier)
    END IF
  ENDSUBROUTINE
SUBROUTINE StoreStats(taggerRounds, runnerRounds)
  runnerRoundsWon ← runnerRoundsWon + (runnerRounds / (taggerRounds + runnerRounds))
  taggerRoundsWon ← taggerRoundsWon + (taggerRounds / (taggerRounds + runnerRounds))
  statsString ← FormatStats(currentRound, runnerReward, taggerReward, avgDistance, runnerRoundsWon,
taggerRoundsWon)
  WriteToFile(filepath, statsString)
ENDSUBROUTINE
```

The program will use Unity physics and UI operations to show the players' actions – in this way Unity merely acts as a representation of the problem that the algorithms are tasked with solving. A different problem, for example Tag using a 2D grid and directionless points rather than a 3D simulation, could be simulated and achieve the same goals. However, it would not be able to display the agents' actions as well as the 3D implementation, and as such the user would not be able to see the nuances of the algorithms as clearly as they would with the chosen implementation.

UI Logic

Panels

The UI system should work as follows. Different parts of the program are organised into their own panels, each of which contain the necessary UI elements to complete their designated function. For panels that take user input in, a hierarchy has been defined. If the panel contains a dropdown, it should contain the dropdown handler for it. If the panel takes input without this, it should have the input validator upon the panel. This could be implemented as such:

```
SUBROUTINE InitializeFieldRangeValidator()
  // Get the input field component
  inputField ← GetComponent(TMP_InputField)
  // Add listener to validate input when editing ends
  ADD ValidateInput TO inputField.onEndEdit
ENDSUBROUTINE


SUBROUTINE ValidateInput(input)
  IF validatorType = 0 THEN
    // Convert input to float
    val ← PARSE_FLOAT(input)

    // Check and output suitable message
    IF val < minValue THEN
      UpdateText("value too small.", RED)
      isValid ← FALSE
    ELSE IF val > maxValue THEN
      UpdateText("value too large", RED)
      isValid ← FALSE
    ELSE
      isValid ← TRUE
```

```
    ENDIF
  ELSE IF validatorType = 1 THEN
    isValid ← MATCH_REGEX(input, "^[a-zA-Z]+$")
    IF NOT isValid THEN
      UpdateText("Should be a file path.", RED)
    ENDIF
  ENDIF
ENDSUBROUTINE

SUBROUTINE UpdateText(feedback, colour)
  IF feedbackText ≠ NULL THEN
    feedbackText.text ← feedback
    feedbackText.color ← colour
  ENDIF
ENDSUBROUTINE
```

Note that the program referes to the validatorType. This should be set when the module is added to an input field.

Inputs from a single field

To handle inputs from a single field, there should be a defined validator. This should be able to switch between both text validation for a file path and range validation for a numerical value, after the value has stopped being entered. The Unity UI's function can be used to restrict the input fields to integers, floats, or strings as required. This is done to ensure that the user finds out immediately if their attempted input is of an invalid type, rather than having to wait until they type it out. The value of the field's validity should be stored, and if the field is invalid, it should change the text of a given text object near the field to tell the user that the input is invalid.

Inputs from sets of fields

To handle input from a group of these fields, there should be something defined that takes in every field in the set to be validated. Then, this object of fields can be checked to see if they are all valid, and a message can be outputted if otherwise.

Dropdowns

Dropdowns should have a set of groups of GameObjects to be hidden or shown depending on the dropdown's chosen value. For the algorithms, there should be an input purpose that can be accessed, so another script accesing this can know what the selected value of the dropdown actually means in terms of which algorithm is selected. An example of this being implemented could be:

```
SUBROUTINE InitializeDropdownHandler()
  // Add listener to dropdown
  ADD OnDropdownValueChanged TO dropdown.onValueChanged
  // Set initial field visibility based on default dropdown selection
  UpdateFieldVisibility(dropdown.value)
ENDSUBROUTINE

SUBROUTINE OnDropdownValueChanged(index)
  UpdateFieldVisibility(index)
ENDSUBROUTINE

SUBROUTINE UpdateFieldVisibility(index)
  FOR i FROM 0 TO LENGTH(fieldGroups) - 1 DO
    fieldGroups[i].SetActive(i = index)
  ENDFOR
  chosenValidator ← fieldGroups[index].GetComponent(TakeFieldsInput)

  SWITCH index DO
    CASE 0:
      chosenValidator.inputPurpose ← "DQN"
    CASE 1:
      chosenValidator.inputPurpose ← "PPO"
```

```
        CASE 2:
            chosenValidator.inputPurpose ← "NEAT"
    ENDSWITCH
ENDSUBROUTINE
```

Data Structures

Array

In C#, an array is a fundamental data structure used for storing a fixed-size sequential collection of elements of the same type. Unlike lists or more specialized data structures like the NDArray, C# arrays can be single-dimensional, multi-dimensional (such as rectangular arrays), or jagged (arrays of arrays). They are homogenous, meaning that all elements must be of the same data type. While arrays can be indexed, iterated over, and provide basic storage and retrieval functions, they lack advanced features such as broadcasting or shape attributes found in more sophisticated data structures. Although C# arrays can have multiple dimensions, they do not inherently support high-level mathematical operations for element-wise computation or data manipulation. This makes them suitable for simpler data storage and iteration tasks, but less ideal for complex mathematical computations common in fields like machine learning or data science.

NDArray

Most machine learning tutorials use Python, and in particular they use a library called NumPy to do maths operations. However, as Unity uses C#, the program will use a NumPy alternative known as NumSharp. The main data type that NumSharp uses is an NDArray, upon which it performs all of its mathematical operations. These can take any number of dimensions, and are homogenous, meaning each element must be the same data type. NDArrays also have a lot of attributes that make them easier to use than regular arrays, in particular their shape and axis attributes allowing developers to see exactly how big an NDArray has been defined or to access items along a given dimension of the NDArray rather than having to do so manually. Alongside this, NDArrays support broadcasting, so arrays of different sizes can still be used in operations together. As such, they are ideal for neural network calculations.

List

A list is a general data structure, with a (theoretically) unlimited size. This makes them ideal for storing data or statistics, where mathematical operations are not needed. Lists support standard operations like appending, indexing, and slicing but lack specialized mathematical functions for element-wise operations or broadcasting across large datasets. As such, this differentiates them from NDArrays, along with the fact that lists are strictly one-dimensional, with no limit as to how many items can be added in that single dimension.

Struct

A struct is a value type used to encapsulate related variables. Structs are allocated on the stack, offering better performance for small data types by avoiding heap allocation. They support fields, properties, methods, and constructors but cannot have a parameterless constructor or a destructor. Passing a struct to a method creates a copy of the value, rather than modifying the original instance, and as such they cannot be edited during runtime.

Class

A class in C# is a reference type used as a blueprint for creating objects with properties, methods, and events. Objects created from a class are allocated on the heap, supporting dynamic memory allocation. Classes support inheritance, polymorphism, constructors, destructors, and access modifiers. Passing a class instance to a method passes a reference, meaning that changes within the method to affect the original object. One of the main uses of classes in this project was the creation of parent classes, followed by the creation of child classes that inherit, override and add to the parent class. Also, classes were referenced in other code files, in particular in Unity where different scripts could be referenced within one another to call functions, access variables etc.

Dictionary

A dictionary in C# is a data structure that stores key-value pairs, allowing for fast lookups, insertions, and deletions based on keys. Unlike arrays or lists, which are indexed numerically, a dictionary uses unique keys to reference values. The Dictionary<TKey, TValue> class in C# is implemented using a hash table, making lookups highly efficient, with an average time complexity of O(1) for retrieval operations. Keys in a dictionary must be unique, but values can be duplicated. Dictionaries are particularly useful when data needs to be accessed quickly without iterating through an entire collection, such as mapping player scores to usernames or storing game settings. Compared to lists, dictionaries offer a more structured way of storing and accessing data when a key-value relationship is needed, though they require slightly more memory due to their underlying hashing mechanism.

Unity UI Elements

A system in Unity used for creating and managing graphical user interfaces. Includes components such as Text, Buttons, Dropdowns, and Panels, which can be arranges in a Canvas to design user interfaces. They are also part of Unity's GameObject hierarchy, allowing them to also be manipulated like any other GameObject in the scene. This allows for realtime updates, and user-driven interactions.

Software Requirements/Libraries

NumSharp

Reference [20]

Due to many mathematical options being done using numpy, the decision has been made that, since this project is being implemented using NumSharp's maths functions. This is mainly used in the implementation of the neural network, however some functions from numpy are not fully implemented in NumSharp, so they should be implemented in C# within the program instead.

TensorSharp

Reference [21]

Used for advanced linear algebra options on the actor and critic networks in the PPO implementation. As such, the implementations of the networks are also done using TensorSharp, and cannot be done using the C# implementation in the current scope of the program, so TensorSharp is used.

Accord

Reference [22], [23]

Used for generating a Multivariate Normal Distribution, also for use in the PPO implementation.

Unity UI

Used to take inputs from the user into the program for the agents' hyperparameters. Also used to display values from the program for the user to see as the program runs, and after the program runs to see the results of the whole program.

StatsHandler and StatsDrawer

Reference [24], [25]

The StatsDrawer and StatsHandler are two modules developed by PingSharp. References include this YouTube video and this associated GitHub repo. These modules are used to draw graphs in Unity, as other modules could not handle changing between multiple graphs as compared to only drawing one.

Unity 3D Simulation

This is used to simulate the environment that the agents learn in. This is because the focus of the project is upon the agents' themselves, which means that they should be able to function in any given environment. So, Unity's Physics Engine was used to simulate the environment the agents would learn in, with the game itself and the agents' interactions with it handled in code.

Hardware Requirements

User specifications:

From the interview:

"My main machines for general work are Linux workstations which have nvidia GPU cards, either Quadro M3000 SE with 4GB GPU memory, or A40-12Q with 12GB of GPU memory.

For real work, I have access to an on-premises HPC cluster, which provides access to a range of GPUs:

V100 (between 2-8 cards/machine, 32GB per GPU)

A100 (8 A100 cards/machine, 80GB / GPU)

DGX-2 (16 V100 cards, 23GB / GPU)

DGX A100 (8 A100 cards, 40GB / GPU)

HGX-2 V100 (16 V100 cards, 32GB / GPU)

I also have access to all AWS resources."

Developer specifications:

- Windows 11 Home
- Unity 6000.0.28f1
- Visual Studio Code
- AMD Ryzen 9 7940HS w/ Radeon 780M Graphics
- NVIDIA GeForce RTX 4060 Laptop GPU
- 16.0 GB RAM
- 1TB SSD Storage

From these, we can see that the hardware specifications of machines used in industries where users would be using the application far exceed what is available to me and thus, if a program is able to run reasonably well upon my machine, then it should be able to run exceptionally upon the hardware of a given user. As such, my machine is given as a reasonable test benchmark for the performance of the program.

Technical Solution

Technical Overview

This section contains the code for the implementation of the program as set forth in the Design section, using the structures, algorithms and libraries to achieve the objectives. The Techniques Used table shows various techniques used throughout the program and page numbers where evidence of such techniques can be found. Note that page numbers are according to all sections being open, and may be off by 1 or 2 pages. At the beginning of each file, the Objectives that the file is used to meet and the techniques used are in red. Throughout the code, techniques used and objectives are also commented in red. Code formatted using Easy Code Formatter.

Techniques Used

Technique	Page Numbers
OOP	Throughout
Classes	Throughout
Struct	pg.65 – pg.66
Enum	pg.67
Inheritance	pg.57, pg.59, pg.78 -pg.79, pg. 81-84, pg.86
Polymorphism	pg.57, pg.59, pg.78 -pg.79, pg. 81-84, pg.86
Procedures and Functions	Throughout
Unity 3D Physics	pg.99 – pg.102
Unity UI Handling	pg.102 – pg.106
Arrays	Throughout
Neural Networks	pg.57 – pg.78, pg. 81-88,
Lists	pg.65 – pg.78
Sets	pg.75
Vectors	pg.83
Dictionary	pg.96 – pg.98
Read/write to file	pg.88 , pg.102
Try/Catch	pg.69, pg.73
Regular Expressions	pg.105
String Formatting	pg.104
Algorithms	
Neural Network	pg.81 – pg.87, pg. 61
Backpropagation	-
HE-Initalisation	
Adam Optimiser	-
Adjacency Matrix	-
DQN	pg.57 – pg.59
Bellman – Optimality	-
Epsilon – Greedy	-
PPO	pg.59 – pg.65
Generalised Advantage Estimation	-
Multivariate Normal Distribution	-
NEAT	pg.65 – pg.78
Genomic Distance	-
Genomic Crossover	-

Program Code

DQN.cs

```
1. //Objective 3
2. //Class
3. //Neural Network
4. //Procedures & Functions
5. //DQN
6. //Bellman-Optimality
7. //Epsilon-Greedy
8. //Neural Network
9.
10. using FCNN;
11.
12. //math
13. using NumSharp;
14.
15. //system usings
16. using System.Collections.Generic;
17. using UnityEngine;
18.
19. //OOP
20. using Hyperparamters;
21.
22. namespace DQNImplementaion
23. {
24.     public class DQNHyperparameters : Hyperparameters //Polymorphism, Inheritance
25.     {
26.         public float lr = 0.001f; //learning rate
27.         public float gamma = 0.99f; //discounted fctor
28.         public float explorationProbInit = 1.0f; //starting exploration probability
29.         public float explorationProbDecay = 0.005f; //exploration probability decay
30.         public int[] hiddenLayersSizes = new int[] { 24, 24 }; //sizes of hidden layers of network
31.         public int batchSize = 32; //size of experiences we sample to train the neural network
32.         public int memoryBufferSize = 2000; //size of memory buffer (duh)
33.     }
34.
35.     public class DQN
36.     {
37.
38.         public class Epsiode
39.         {
40.             public NDArray currentState;
41.             public bool[] action;
42.             public float reward;
43.             public NDArray nextState;
44.             public bool done;
45.         }
46.
47.         public int inputs { get; set; }
48.         public int outputs { get; set; }
49.         public DQNHyperparameters hyperparameters { get; set; }
50.         public DQNModel model;
51.         public float explorationProb;
52.         public System.Random rnd = new();
53.         public List<Epsiode> memoryBuffer = new();
54.
55.         public DQN(DQNHyperparameters input_hyperparameters)
56.         {
57.             inputs = input_hyperparameters.obsDim;
58.             outputs = input_hyperparameters.actDim;
59.             hyperparameters = input_hyperparameters;
60.             model = new DQNModel(inputs, outputs, hyperparameters.hiddenLayersSizes);
61.             explorationProb = hyperparameters.explorationProbInit;
62.         }
63.
64.         //Objective 3c
65.         //Neural Network
66.         public bool[] ComputeAction(NDArray state)
```

```
67.         {
68.             //we sample a variable uniformly over [0, 1]
69.             //if the variable is less than the exploration probability
70.             //  we chose an action randomly
71.             //else
72.             //  we forward the state through the network and choose the action with the highest Q-value
73.
74.             bool[] outputsArray = new bool[5];
75.             for (int i = 0; i < 5; i++)
76.             {
77.                 outputsArray[i] = false;
78.             }
79.
80.             float prob = (float)rnd.NextDouble();
81.             if (prob < explorationProb) //Objective 3ci
82.             {
83.                 //choose random action
84.                 outputsArray[rnd.Next(0, 5)] = true; //Objective 3ciii
85.             }
86.             else
87.             {
88.                 Debug.Log("neural network step");
89.                 //choose highest q-value
90.                 NDArray qValues = model.Forward(state.reshape(1, hyperparameters.obsDim));
91.                 int index = 0;
92.                 for (int i = 0; i < qValues.shape[1]; i++)
93.                 {
94.                     index = (qValues[0, i] > qValues[0, index]) ? i : index; //if value is greater, change
index. otherwise, keep the same
95.                 }
96.                 outputsArray[index] = true; //Objective 3cii
97.             }
98.
99.             return outputsArray;
100.         }
101.
102.         //Objective 3diii
103.         //Epsilon-Greedy
104.         public void UpdateExplorationProbability()
105.         {
106.             //update exploration probability using epsilon-greedy algorithm
107.             explorationProb *= Mathf.Exp(-1 * hyperparameters.explorationProbDecay);
108.         }
109.
110.         //Objective 3d
111.         //Bellman-Optimality
112.         //Neural Network
113.         public void Train()
114.         {
115.             //Objective 3di
116.             //choose set of indexes of episodes to be used
117.             NDArray inds = np.arange(memoryBuffer.Count);
118.             np.random.shuffle(inds);
119.             List<Epsiode> batchSample = new();
120.
121.             //create sample from buffer
122.             foreach (int i in inds)
123.             {
124.                 batchSample.Add(memoryBuffer[i]);
125.             }
126.
127.             //Objective 3dii
128.             //iterate over selected episodes
129.             foreach (Epsiode ep in batchSample)
130.             {
131.                 //compute q-values of s_t
132.                 NDArray qValueCurrentState = model.Forward(ep.currentState.reshape(1,
hyperparameters.obsDim));
133.                 //compute q-target using Bellman-Optimality equation
134.                 float qValueTarget = ep.reward;
135.                 if (!ep.done)
136.                 {
137.                     qValueTarget += hyperparameters.gamma * np.max(model.Forward(ep.nextState.reshape(1,
hyperparameters.obsDim)));
```

```
                }
                int index = 0;
                int i = 0;
                foreach (bool action in ep.action)
                {
                    index = (action) ? i : index; //if the action is the one taken, that is the index we
need to update
                    i++; //otherwise, move on to next index
                }
                //update q value for action taken
                qValueCurrentState[0, index] = qValueTarget;

                //Objective 3dii
                //train the model
                model.Train(ep.currentState.reshape(1, hyperparameters.obsDim),
qValueCurrentState.reshape(1, hyperparameters.actDim), 1, hyperparameters.lr, 0);

            }
        }
    }
}

```

PPO.cs

```
//Objective 4
//PPO
//Generalised Advantage Estimation
//Multivariate Normal Distribution
//Lists
//Procedures & Functions
//OOP
//Neural Network

//neural network
using FCNN;
using TorchSharp;
using static TorchSharp.torch;
using static TorchSharp.torch.optim;
using TorchSharp.Modules;

//math
using Accord.Statistics.Distributions.Multivariate;
using NumSharp;
using Accord.Math;

//system usings
using System.Collections.Generic;
using System;

//OOP
using Hyperparamters;

namespace PPOImplementation
{
    public class PPOHyperparameters : Hyperparameters //Polymorphism, Inheritance
    {
        public int timestepsPerBatch = 4800; //number of timesteps to run per batch
        public int maxTimestepsPerEpisode = 30 * 30; //maximum number of timesteps per episode
        public int nUpdatesPerIteration = 5; //number of times to update networks per iteration
        public float lr = 0.0005f; //learning rate of actor optimiser
        public float gamma = 0.95f; //discount factor to be applied when calculating rewards to go
        public float lam = 0.98f; //lambda paramter for Generalised Advantage Estimation
        public float clip = 0.2f; //recommended 0.2. helps define threshold to clip ratio during surrogate
loss calculation
        public int numMinibatches = 6; //number of mini-batches for mini-batch update
        public float entCoef = 0; //entropy coeffecient for entropy regularisation
        public float maxGradNorm = 0.5f; //gradient clipping threshold
        //sizes for neural networks
        public int[] actorNetworkSizes = new int[] { 32, 32 };
        public int[] criticNetworkSizes = new int[] { 32, 32 };
        public int totalTimesteps = 0; //total timesteps for the whole program to run for
        public int maxEpsisodesInBatch = 5; //max episodes in a batch

```

```
49.     }
50.
51.     public class PPO
52.     {
53.
54.         public class Epsiode
55.         {
56.             public List<NDArray> epObs, epActs, epNextObs;
57.             public List<float> epLogProbs, epRewards, epVals, epDones;
58.             public int epLen;
59.             //Objective 4b
60.             public Epsiode()
61.             {
62.                 epObs = new();
63.                 epActs = new();
64.                 epLogProbs = new();
65.                 epRewards = new();
66.                 epVals = new();
67.                 epDones = new();
68.                 epNextObs = new();
69.                 epLen = 0;
70.             }
71.         }
72.
73.         public int inputs { get; set; }
74.         public int outputs { get; set; }
75.         public PPOHyperparameters hyperparameters { get; set; }
76.         public PPOModel actor;
77.         public PPOModel critic;
78.         public TFPPOModel actorTF;
79.         public TFPPOModel criticTF;
80.         public Adam actorOptim;
81.         public Adam criticOptim;
82.         public System.Random rnd = new();
83.         public double[,] covMat;
84.         public float covMatDet;
85.         public List<Epsiode> batch;
86.         public PPO(PPOHyperparameters input_hyperparameters)
87.         {
88.             hyperparameters = input_hyperparameters;
89.             inputs = hyperparameters.obsDim;
90.             outputs = hyperparameters.actDim;
91.             covMat = new double[outputs, outputs];
92.
93.             for (int i = 0; i < outputs; i++)
94.             {
95.                 for (int j = 0; j < outputs; j++)
96.                 {
97.                     covMat[i, j] = 0.5;
98.                 }
99.             }
100.
101.             covMatDet = (float)covMat.Determinant();
102.             actorTF = new TFPPOModel(inputs, outputs, hyperparameters.actorNetworkSizes);
103.             criticTF = new TFPPOModel(inputs, 1, hyperparameters.criticNetworkSizes);
104.             actorOptim = Adam(actorTF.parameters(), hyperparameters.lr);
105.             criticOptim = Adam(criticTF.parameters(), hyperparameters.lr);
106.
107.             batch = new();
108.         }
109.
110.         //Objective 4c
111.         //Neural Network
112.         //Multivariate Normal Distribution
113.         public (NDArray, NDArray) Forward(NDArray state, Device device)
114.         {
115.             using var scope = torch.no_grad();
116.             //get actor output
117.             NDArray mean = actorTF.forward(state); //Objective 4bi
118.             //convert mean to array for distribution
119.             double[] meanArray = new double[mean.shape[0]];
120.             for (int i = 0; i < mean.shape[0]; i++)
121.             {
122.                 meanArray[i] = mean[i];
```

```
123.             }
124.
125.             //Objective 4bii
126.             var dist = new MultivariateNormalDistribution(meanArray, covMat);
127.             var action = dist.Generate();
128.
129.             //Objective 4biii
130.             //calculate log probs of actions
131.             var logProbs = dist.LogProbabilityDensityFunction(action);
132.
133.             //output as NDArrays
134.             return (np.array(action), np.array(logProbs));
135.         }
136.
137.         //Objective 4diii
138.         //Generalised Advantage Estimation
139.         public List<float> CalculateGAE()
140.         {
141.             List<float> batchRewards = new();
142.             foreach (Epsiode epsiode in batch)
143.             {
144.                 List<float> advantages = new(); //stores advantages for current episode
145.                 float lastAdvantage = 0;
146.                 float delta;
147.
148.                 for (int t = epsiode.epLen; t > 0; t--)
149.                 {
150.                     if (t + 1 < epsiode.epLen)
151.                     {
152.                         //calculate temporal difference error for the current timestep
153.                         delta = epsiode.epRewards[t] + hyperparameters.gamma * epsiode.epVals[t + 1] * (1
- epsiode.epDones[t + 1]) - epsiode.epVals[t];
154.                     }
155.                     else
156.                     {
157.                         //special case at last timestep
158.                         delta = epsiode.epRewards[t] - epsiode.epVals[t];
159.                     }
160.
161.                     //calculate Generalised Advantage Estimation (GAE) for current timestep
162.                     float advantage = delta + hyperparameters.gamma * hyperparameters.lam * (1 -
epsiode.epDones[t]) * lastAdvantage;
163.                     lastAdvantage = advantage; //update lastAdvantage
164.                     advantages.Insert(0, advantage); //insert at beginning of list
165.                 }
166.
167.                 batchRewards.AddRange(advantages); //add onto end of already existing list
168.             }
169.
170.             return batchRewards;
171.         }
172.
173.         //Objective 4a
174.         public void AddEpisodeToBatch(Epsiode ep)
175.         {
176.             if (batch.Count >= hyperparameters.maxEpsisodesInBatch)
177.             {
178.                 batch.RemoveAt(0);
179.             }
180.
181.             batch.Add(ep);
182.         }
183.
184.         //Objective 4d
185.         //Neural Network
186.         public void Learn(int currentT)
187.         {
188.             //calculate advantage using GAE
189.             var A_k = CalculateGAE(); //Objective 4di
190.
191.             //Objectives 4dii, 4diii
192.
193.             List<float> V = new();
194.             foreach (Epsiode epsiode in batch)
```

```
            {
                foreach (NDArray obs in epsiode.epObs)
                {
                    V.Add(criticTF.forward(obs));
                }
            }

            List<float> batchRTGs = new();
            for (int i = 0; i < V.Count; i++)
            {
                batchRTGs.Add(A_k[i] + V[i]);
            }

            //normalise advantages to decrease variance of advantages
            //+ makes convergence more stable and faster
            float sigx2 = 0;
            float sigX = 0;
            int n = A_k.Count;

            for (int i = 0; i < n; i++)
            {
                sigx2 += (float)Math.Pow((double)A_k[i], 2);
                sigX += A_k[i];
            }

            float A_k_mean = sigX / n;
            float A_k_std = (float)Math.Sqrt((sigx2 - n * Math.Pow((double)A_k_mean, 2)) / (n - 1 + 1e-
10f));

            for (int i = 0; i < A_k.Count; i++)
            {
                A_k[i] = (A_k[i] - A_k_mean) / (A_k_std + 1e-10f);
            }

            //calculate step for loop updating networks
            int step = 0;
            foreach (var episode in batch)
            {
                step += episode.epLen;
            }

            //update the network for a given number of epochs
            NDArray inds = np.arange(step);
            int minibatchSize = step / hyperparameters.numMinibatches;
            for (int i = 0; i < hyperparameters.nUpdatesPerIteration; i++)
            {
                //learning rate annealing
                float frac = (float)(currentT - 1) / hyperparameters.totalTimesteps;
                float newLr = hyperparameters.lr * (1 - frac);

                //make sure learning rate does not go below 0
                newLr = (newLr > 0) ? newLr : 0;

                //set learning rates
                actorOptim = Adam(actorTF.parameters(), newLr);
                criticOptim = Adam(criticTF.parameters(), newLr);

                np.random.shuffle(inds); //shuffling

                //mini batch update
                for (int j = 0; j < step; j += minibatchSize)
                {
                    //init
                    List<NDArray> minibatchObs = new();
                    List<NDArray> minibatchActs = new();
                    List<float> minibatchLogProbs = new();
                    List<float> minibatchAdvantage = new();
                    List<float> minibatchRTGs = new();

                    //define what parts of ind to use
                    int end = j + minibatchSize; //want to use indexes from j to end
                    List<int> idx = new();

                    //extract desired indexes
```

```
268.                     for (int k = 0; k < end; k++)
269.                     {
270.                         idx.Add(inds[k]);
271.                     }
272.
273.                     //extract observation data for given index
274.                     foreach (int k in inds)
275.                     {
276.                         //get episode & index to extract data
277.                         (Epsiode, int) output = FindEpsiodeInstanceFromIndex(k);
278.                         Epsiode miniEp = output.Item1;
279.                         int index = output.Item2;
280.                         //extract data for a given minibatch
281.                         minibatchObs.Add(miniEp.epObs[index]);
282.                         minibatchActs.Add(miniEp.epActs[index]);
283.                         minibatchLogProbs.Add(miniEp.epLogProbs[index]);
284.                         minibatchAdvantage.Add(A_k[k]);
285.                         minibatchRTGs.Add(batchRTGs[k]);
286.                     }
287.
288.                     //calculate V_phi, pi_theta(a_t | s_t) and entropy
289.                     var eval = Evaluate(minibatchObs, minibatchActs);
290.                     V = eval.Item1;
291.                     var currentLogProbs = eval.Item2;
292.                     var entropy = eval.Item3;
293.
294.                     //calculate the ratio pi_theta(a_t | s_t) / pi_theta_k(a_t | s_t)
295.                     List<float> logRatios = new();
296.                     List<float> ratios = new();
297.                     List<float> approxKlList = new();
298.                     for (int k = 0; k < currentLogProbs.Count; k++)
299.                     {
300.                         //note that subtracting log == dividiing
301.                         logRatios.Add(currentLogProbs[k] - minibatchLogProbs[k]);
302.                         ratios.Add(MathF.Exp(logRatios[k]));
303.                         approxKlList.Add(ratios[k] - 1 - logRatios[k]);
304.                     }
305.
306.                     //calculate surrogate losses
307.                     List<float> surr1 = new();
308.                     List<float> surr2 = new();
309.                     for (int k = 0; k < ratios.Count; k++)
310.                     {
311.                         surr1.Add(ratios[k] * minibatchAdvantage[k]);
312.                         surr2.Add((float)(clamp(ratios[k], 1 - hyperparameters.clip, 1 +
hyperparameters.clip) * minibatchAdvantage[k]));
313.                     }
314.                     Tensor surr1T = tensor(surr1, ScalarType.Float64, null, false);
315.                     Tensor surr2T = tensor(surr2, ScalarType.Float64, null, false);
316.                     Tensor actorLosses = (-min(surr1T, surr2T)).mean();
317.
318.                     //entropy calculation
319.                     float total = 0;
320.                     n = 0;
321.                     foreach (float val in entropy)
322.                     {
323.                         total += val;
324.                         n++;
325.                     }
326.
327.                     //entropy regularisation
328.                     float entropyLoss = total / n;
329.                     actorLosses -= hyperparameters.entCoef * entropyLoss;
330.                     //calculate gradients and perform backward propogation for actor network
331.                     actorOptim.zero_grad();
332.                     actorLosses.backward(); //Objective 4dii
333.                     torch.nn.utils.clip_grad_norm_(actorTF.parameters(), hyperparameters.maxGradNorm);
//gradient clipping
334.                     actorOptim.step();
335.
336.                     //calculate MSE between V and minibatchRTGs
337.                     float cumMSE = 0;
338.                     n = 0;
339.                     for (int k = 0; k < V.Count; k++)
```

```
                    {
                        cumMSE = MathF.Pow(V[k] - minibatchRTGs[k], 2);
                        n++;
                    }
                    Tensor criticLosses = tensor(cumMSE / n, ScalarType.Float64, null, false);

                    //calculate gradients and perform backward propogation for critic network
                    criticOptim.zero_grad();
                    criticLosses.backward(); //Objective 4diii
                    torch.nn.utils.clip_grad_norm_(criticTF.parameters(), hyperparameters.maxGradNorm);
//gradient clipping
                    criticOptim.step();

                }

            }

        }

        public (Epsiode, int) FindEpsiodeInstanceFromIndex(int index)
        {

            //create set of lengths
            List<int> lens = new();
            foreach (var ep in batch)
            {
                lens.Add(ep.epLen);
            }

            //create set of cumulative lengths
            List<int> cumLens = new();
            int currentTotal = 0;
            foreach (int num in lens)
            {
                currentTotal += num;
                cumLens.Add(currentTotal);
            }

            //find episode index
            int epIndex = 0;
            while (cumLens[epIndex] < index)
            {
                epIndex++;
            }
            //find observation index within given episode
            int obsIndex = index - (epIndex > 0 ? cumLens[epIndex - 1] : 0); //if at first episode, should
be 0


            //return values
            return (batch[epIndex], obsIndex);
        }

        //Multivariate Normal Distribution
        public (List<float>, List<float>, List<float>) Evaluate(List<NDArray> miniObs, List<NDArray>
miniActs)
        {
            //Estimate the values of each observation, and the log probs of each action.

            //query critic network for a value V for each miniObs
            List<float> V_ = new();
            List<float> logProbs = new();
            List<float> entropy = new();

            for (int i = 0; i < miniObs.Count; i++)
            {

                //query critic network for a value for given obs and add to list
                V_.Add(criticTF.forward(miniObs[i]));

                //query actor network for mean action for given obs
                NDArray meanND = actorTF.forward(miniObs[i]);
                double[] mean = new double[meanND.size];
                for (int j = 0; j < meanND.size; j++)
                {
```

```
411.                     mean[j] = meanND[j];
412.                 }
413.
414.                 //create distribution
415.                 var dist = new MultivariateNormalDistribution(mean, covMat);
416.
417.                 //query for log probs and add to list
418.                 logProbs.Add((float)dist.LogProbabilityDensityFunction(miniActs[i]));
419.
420.                 //calculate entropy and add to list
421.                 float k = dist.Dimension;
422.                 entropy.Add((float)(0.5 * Math.Log(covMatDet) + 0.5 * k * (1 + Math.Log(2 + Math.PI))));
423.             }
424.
425.             return (V_, logProbs, entropy);
426.         }
427.     }
428. }
429.
```

NEAT.cs

```
 1. //Objective 5
 2. //OOP
 3. //Classes
 4. //Struct
 5. //Enum
 6. //Arrays
 7. //Neural Networks
 8. //Lists
 9. //Dictionary
10. //Try/Catch
11. //NEAT
12. //Genomic Distance
13. //Genomic Crossover
14.
15. //for activation functions
16. using FCNN;
17. using System.Collections.Generic;
18.
19. //system usings
20. using System;
21. using System.Linq;
22.
23. //for some reason the rest of the program cant find a genome without this
24. using static NEATImplementation.Genome;
25.
26. //OOP
27. using Hyperparamters;
28.
29. namespace NEATImplementation
30. {
31.     //Classes
32.     //genome - a single neural network to be a part of a specie
33.     public class Genome
34.     {
35.         //Structs
36.         //edge struct
37.         public struct Edge
38.         {
39.             public float weight;
40.             public bool enabled;
41.             public Edge(float weight_userinput)
42.             {
43.                 weight = weight_userinput;
44.                 enabled = true;
45.             }
46.             public void enable()
47.             {
48.                 enabled = true;
49.             }
50.             public bool exists()
51.             {
52.                 return true;
```

```
53.             }
54.         }
55.
56.         //node struct
57.         public struct Node
58.         {
59.             public float output;
60.             public float bias;
61.             public Activation activation;
62.             public Node(Activation default_activation)
63.             {
64.                 activation = default_activation;
65.                 output = 0;
66.                 bias = 0;
67.             }
68.         }
69.
70.         public int inputs { get; }
71.         public int outputs { get; }
72.         public Activation default_activation { get; }
73.         public int unhidden, maxNode, fitness, adjustedFitness;
74.         public List<List<Edge>> edges;
75.         public List<Node> nodes;
76.         public Random rnd;
77.
78.         public Genome(int inputs_userinput, int outputs_userinput, Activation
default_activation_userinput)
79.         {
80.             inputs = inputs_userinput;
81.             outputs = outputs_userinput;
82.             default_activation = default_activation_userinput;
83.             unhidden = inputs + outputs;
84.             maxNode = inputs + outputs;
85.             fitness = 0;
86.             adjustedFitness = 0;
87.             edges = new List<List<Edge>>();
88.             nodes = new List<Node>();
89.             rnd = new Random();
90.
91.             Console.WriteLine($"{inputs}, {outputs}, {maxNode}");
92.         }
93.
94.         public void GenerateNetwork()
95.         {
96.             //generate the neural network of this genome with minimal initial topology
97.             //only input and output layers
98.
99.             Node default_node = new();
100.            default_node.activation = default_activation;
101.
102.            for (int i = 0; i < maxNode; i++)
103.            {
104.                nodes.Add(default_node);
105.                Console.WriteLine("Node added");
106.            }
107.
108.            for (int i = 0; i < inputs; i++)
109.            {
110.                for (int j = inputs; j <= unhidden; j++)
111.                {
112.                    AddEdge(i, j, rnd.Next(-1, 1));
113.                    Console.WriteLine("Edge added");
114.                }
115.            }
116.
117.            Console.WriteLine($"{edges.Capacity}, {nodes.Capacity}");
118.        }
119.
120.        //Objective 5bii
121.        //Neural Network
122.        //Lists
123.        //Enum
124.        public float[] Forward(float[] programInputs)
125.        {
```

```
126.             //loop through each input
127.             for (int i = 0; i < programInputs.Length; i++)
128.             {
129.                 //set node outputs to inputs
130.                 var copy = nodes[i];
131.                 copy.output = programInputs[i];
132.                 nodes[i] = copy;
133.             }
134.
135.             //keep track of which nodes are connected to which
136.             List<List<int>> from = new();
137.             //initalise
138.             for (int i = 0; i < maxNode; i++)
139.             {
140.                 from.Add(new List<int>());
141.             }
142.
143.             //loop through every edge
144.             for (int i = 0; i < edges.Count; i++)
145.             {
146.                 for (int j = 0; j < edges[i].Count; j++)
147.                 {
148.                     //if connection enabled, note source index
149.                     if (edges[i][j].enabled)
150.                     {
151.                         from[j].Add(i);
152.                     }
153.                 }
154.             }
155.
156.             //order nodes - first process hidden nodes, then input nodes
157.             var ordered_nodes = Enumerable.Range(unhidden, maxNode -
unhidden).Concat(Enumerable.Range(inputs, unhidden - inputs));
158.             foreach (int j in ordered_nodes)
159.             {
160.                 //weighted sum
161.                 float ax = 0;
162.                 foreach (int i in from[j])
163.                 {
164.                     ax += edges[i][j].weight * nodes[i].output;
165.                 }
166.                 var currentnode = nodes[j];
167.                 //compute final output using activation
168.                 currentnode.output = currentnode.activation.DoActivation(ax + currentnode.bias);
169.             }
170.
171.             //process and return
172.             float[] nnoutputs = new float[outputs];
173.             for (int i = 0; i < unhidden - inputs; i++)
174.             {
175.                 nnoutputs[i] = nodes[i].output;
176.             }
177.             return nnoutputs;
178.         }
179.
180.         //Objective 5dii
181.         //Dictionary
182.         //Lists
183.         public void Mutate(Dictionary<string, float> probabilities)
184.         {
185.             //randomly mutate genome to initiate variation
186.
187.             if (IsDisabled())
188.             { //if everything is disabled, enable some
189.                 AddEnabled();
190.             }
191.
192.             var population = new List<string>(probabilities.Keys); //what happens to the population
193.             var weights = new List<float>(probabilities.Values); //weights of possibilities
194.             var choice = SelectRandomChoice(population, weights); //randomly choose a choice
195.
196.             switch (choice) //perform choice
197.             {
198.                 //Objective 5di\1\5a
```

```
199.                 case "node":
200.                     AddNode();
201.                     break;
202.
203.                 //Objective 5di\1\5b
204.                 case "edge":
205.                     (int, int) pair = RandomPair();
206.                     AddEdge(pair.Item1, pair.Item2, (float)rnd.NextDouble() * 2 - 1); //creates edge with
weight between -1 and 1
207.                     break;
208.
209.                 //Objective 5di\1\5c
210.                 case "weight perturb":
211.                 case "weight set":
212.                     ShiftWeight(choice);
213.                     break;
214.
215.                 //Objective 5di\1\5d
216.                 case "bias perturb":
217.                 case "bias set":
218.                     ShiftBias(choice);
219.                     break;
220.
221.             }
222.
223.             Reset();
224.         }
225.
226.         public string SelectRandomChoice(List<string> population, List<float> weights)
227.         {
228.             float totalWeight = 0;
229.             foreach (float weight in weights)
230.             {
231.                 totalWeight += weight;
232.             }
233.             float[] cumulativeWeights = new float[weights.Count];
234.             cumulativeWeights[0] = weights[0] / totalWeight; //normalised first weight
235.             for (int i = 1; i < weights.Count; i++)
236.             {
237.                 cumulativeWeights[i] = cumulativeWeights[i - 1] + (weights[i] / totalWeight);
//cumulative, normalised weights
238.             }
239.             double randomValue = rnd.NextDouble(); //random number betwwen 0 and 1
240.             for (int i = 0; i < cumulativeWeights.Length; i++)
241.             {
242.                 if (randomValue < cumulativeWeights[i])
243.                 { //select index based on random value
244.                     return population[i];
245.                 }
246.             }
247.
248.             return population[0]; //fallback
249.         }
250.
251.         //Objective 5di\1\5a
252.         //Neural Network
253.         private void AddNode()
254.         {
255.             //store all enabled nodes
256.             List<(int, int)> enabled = new();
257.             //iterate through to fine enabled
258.             for (int i = 0; i < edges.Count; i++)
259.             {
260.                 for (int j = 0; j < edges[i].Count; j++)
261.                 {
262.                     //if enabled, store on list
263.                     if (edges[i][j].enabled)
264.                     {
265.                         enabled.Add((i, j));
266.                     }
267.                 }
268.             }
269.
270.             //randomly select enabled edge to split and inset new node
```

```
            (int, int) selectedEdge = enabled[rnd.Next(0, enabled.Count - 1)];
            //retrieve selected edge
            Edge edge = edges[selectedEdge.Item1][selectedEdge.Item2];

            //disable selected edge
            edge.enabled = false;
            edges[selectedEdge.Item1][selectedEdge.Item2] = edge;

            //create new node
            int newNode = maxNode;
            maxNode++;

            //initalise new node
            Node node = new(default_activation);
            nodes.Add(node);

            //add edges connecting new node to the rest of the graph
            AddEdge(selectedEdge.Item1, newNode, 1);
            AddEdge(newNode, selectedEdge.Item2, edge.weight);
        }

        //Objective 5di\1\5b
        //Try/Catch
        private void AddEdge(int i, int j, float weight)
        {
            try
            {
                edges[i][j].enable(); //try to enable existing edge
            }
            catch
            {
                try { edges[i].Add(new Edge(weight)); } //try to add edge to node
                catch
                {
                    //create new list for node, add edge to that
                    List<Edge> temp = new();
                    edges.Add(temp);
                    edges[i].Add(new Edge(weight));
                }
            }
        }

        public void AddEnabled()
        {
            var disablededges = new List<Edge>();
            // find disabled edges
            foreach (var edgeList in edges)
            {
                disablededges.AddRange(edgeList.Where(e => !e.enabled));
            }

            if (disablededges.Count > 0)
            {
                var randomEdge = disablededges[rnd.Next(disablededges.Count)];
                randomEdge.enabled = true;
            }
        }
        //Objective 5di\1\5c
        public void ShiftWeight(string type)
        {
            //randomly shift/perturb or set one of the edges weights
            int i = rnd.Next(edges.Count);
            int j = rnd.Next(edges[i].Count);
            Edge edge = edges[i][j];
            if (type == "weight perturb")
            {
                edge.weight += (float)(rnd.NextDouble() * 2) - 1f;
            }
            else if (type == "weight set")
            {
                edge.weight = (float)(rnd.NextDouble() * 2) - 1f;
            }
            edges[i][j] = edge;
        }
```

```
345.
346.         //Objective 5di\1\5d
347.         public void ShiftBias(string type)
348.         {
349.             //randomly shift/perturb or set one of the edges weights
350.             int i = rnd.Next(inputs, maxNode);
351.             Node node = nodes[i];
352.             if (type == "bias perturb")
353.             {
354.                 node.bias += (float)(rnd.NextDouble() * 2) - 1f;
355.             }
356.             else if (type == "bias set")
357.             {
358.                 node.bias = (float)(rnd.NextDouble() * 2) - 1f;
359.             }
360.             nodes[i] = node;
361.         }
362.
363.         //Enum
364.         //Lists
365.         public (int, int) RandomPair()
366.         {
367.             /* select a pair of nodes such that
368.              *  i is not an output
369.              *  j is not an input
370.              *  i != j
371.              */
372.
373.             List<int> availableNodes = Enumerable.Range(0, maxNode)
374.                                         .Where(n => !IsOutput(n))
375.                                         .ToList();
376.
377.             int i = availableNodes[rnd.Next(availableNodes.Count)];
378.             int j = 0;
379.
380.             List<int> jList = Enumerable.Range(0, maxNode)
381.                               .Where(n => !IsInput(n) && n != i)
382.                               .ToList();
383.
384.             if (jList.Count == 0)
385.             {
386.                 j = maxNode;
387.                 AddNode();
388.             }
389.             else
390.             {
391.                 j = jList[rnd.Next(jList.Count)];
392.             }
393.
394.             return (i, j);
395.         }
396.
397.         public bool IsInput(int n)
398.         {
399.             return 0 <= n && n < inputs;
400.         }
401.
402.         public bool IsOutput(int n)
403.         {
404.             return inputs <= n && n < unhidden;
405.         }
406.
407.         public bool IsHidden(int n)
408.         {
409.             return unhidden <= n && n < maxNode;
410.         }
411.
412.         public bool IsDisabled()
413.         {
414.             for (int i = 0; i < edges.Count; i++)
415.             {
416.                 for (int j = 0; j < edges[i].Count; j++)
417.                 {
418.                     if (edges[i][j].enabled)
```

```
                    {
                        return false;
                    }
                }
            }

            return false;
        }

        public void Reset()
        {
            for (int i = 0; i < maxNode; i++)
            {
                Node node = nodes[i];
                node.output = 0;
                nodes[i] = node;
            }
            fitness = 0;
        }
    }
    public class Specie
    {
        public Random rnd;
        public int maxFitnessHistory { get; }
        public List<Genome> members { get; set; }
        public float fitnessSum;
        public List<float> fitnessHistory;

        public Specie(int input_max_fitness_history, List<Genome> input_members)
        {
            maxFitnessHistory = input_max_fitness_history;
            members = input_members;
            fitnessSum = 0;
            fitnessHistory = new List<float>();
            rnd = new Random();
        }

        //Objective 5di
        //Genomic Crossover
        public Genome Breed(Dictionary<string, float> mutationProbabilities, Dictionary<string, float>
breedProbabilities)
        {
            //return a child as a result of either a mutated clone or crossover between two parent genomes
            Genome child;
            var population = new List<string>(breedProbabilities.Keys);
            var weights = new List<float>(breedProbabilities.Values);
            var choice = SelectRandomChoice(population, weights);

            if (choice == "asexual" || members.Count == 1)
            {
                //Objective 5di\1
                child = members[rnd.Next(members.Count)];
                child.Mutate(mutationProbabilities);
            }
            else
            { //choice == "sexual"
                //Objective 5di\2\5a
                int num1 = rnd.Next(members.Count);
                int num2 = rnd.Next(members.Count);
                while (num1 == num2)
                {
                    num2 = rnd.Next(members.Count);
                } //find 2 distinct parents in the specie

                //Objective 5di\2\5b
                child = GenomicCrossover(members[num1], members[num2]);
            }

            return child;
        }

        public void UpdateFitness()
        {
            fitnessSum = 0;
```

```
            foreach (var genome in members)
            {
                genome.adjustedFitness = genome.fitness / members.Count;
                fitnessSum += genome.adjustedFitness;
            }

            fitnessHistory.Add(fitnessSum);

            if (fitnessHistory.Count > maxFitnessHistory)
            {
                fitnessHistory.RemoveAt(0);
            }
        }

        public void CullGenomes(bool fittestOnly)
        {
            //externimate the weakest per specie
            if (fittestOnly)
            {
                Genome fittestGenome = members[0];
                foreach (Genome genome in members)
                {
                    fittestGenome = (genome.fitness > fittestGenome.fitness) ? genome : fittestGenome;
                }
                members = new List<Genome> { fittestGenome };
            }
            else
            {
                members = members.OrderByDescending(g => g.fitness).ToList();
                int countToTake = (int)Math.Ceiling(members.Count * 0.25);
                members = members.Take(countToTake).ToList();
            }
        }

        public Genome GetBest()
        {
            //get member with highest fitness
            Genome bestGenome = members[0];
            foreach (Genome genome in members)
            {
                bestGenome = (genome.fitness > bestGenome.fitness) ? genome : bestGenome;
            }
            return bestGenome;
        }

        public bool CanProgress()
        {
            //determines whether species should survive the culling
            int n = fitnessHistory.Count;
            float totalFitness = 0;

            foreach (Genome g in members)
            {
                totalFitness += g.fitness;
            }

            float avgFitness = totalFitness / n;
            return avgFitness > fitnessHistory[0] || n < maxFitnessHistory;
        }

        public string SelectRandomChoice(List<string> population, List<float> weights)
        {
            float totalWeight = 0;
            foreach (float weight in weights)
            {
                totalWeight += weight;
            }
            float[] cumulativeWeights = new float[weights.Count];
            cumulativeWeights[0] = weights[0] / totalWeight; //normalised first weight
            for (int i = 1; i < weights.Count; i++)
            {
                cumulativeWeights[i] = cumulativeWeights[i - 1] + (weights[i] / totalWeight);
//cumulative, normalised weights
            }
```

```
565.             double randomValue = rnd.NextDouble(); //random number betwwen 0 and 1
566.             for (int i = 0; i < cumulativeWeights.Length; i++)
567.             {
568.                 if (randomValue < cumulativeWeights[i])
569.                 { //select index based on random value
570.                     return population[i];
571.                 }
572.             }
573.
574.             return population[0]; //fallback
575.         }
576.
577.         //Objective 5di\2
578.         //Genomic Crossover
579.         //Lists
580.         public Genome GenomicCrossover(Genome a, Genome b)
581.         {
582.             //breed two genomes and return the child. matching genes are inherited randomly, while disjoin
genes are inherited from the fitter parent
583.             Genome child = new(a.inputs, a.outputs, a.default_activation);
584.             List<List<Edge>> aIn = (a.edges.Count > b.edges.Count) ? a.edges : b.edges;
585.             List<List<Edge>> bIn = (a.edges.Count > b.edges.Count) ? b.edges : a.edges;
586.             List<Edge> tempList;
587.             List<List<(Edge, Edge)>> combinedIns = new();
588.             Edge childEdge;
589.
590.             //create a list of edges that match between the two parents "homologous genees"
591.             for (int i = 0; i < aIn.Count; i++)
592.             {
593.                 combinedIns.Add(new List<(Edge, Edge)>());
594.                 tempList = (aIn[i].Count < bIn[i].Count) ? aIn[i] : bIn[i]; //templist is the shorter of
both lists
595.                 for (int j = 0; j < tempList.Count; j++)
596.                 {
597.                     combinedIns[i].Add((aIn[i][j], bIn[i][j]));
598.                 }
599.             }
600.
601.             //child inherits either parent's homologous genes
602.             for (int i = 0; i < combinedIns.Count; i++)
603.             {
604.                 List<(Edge, Edge)> edgePairList = combinedIns[i];
605.                 foreach (var edgePair in edgePairList)
606.                 {
607.                     childEdge = (rnd.Next(1) == 0) ? edgePair.Item1 : edgePair.Item2;
608.                     child.edges[i].Add(childEdge);
609.                 }
610.             }
611.
612.             //get fitter parent
613.             Genome fitterParent = (a.fitness > b.fitness) ? a : b;
614.             //if edge already exists, leave it be (homologous gene)
615.             //if edge does not already exist, add the fitter parent's edge (disjoint gene)
616.             for (int i = 0; i < fitterParent.edges.Count; i++)
617.             {
618.                 for (int j = 0; j < fitterParent.edges[i].Count; j++)
619.                 {
620.                     try
621.                     {
622.                         child.edges[i][j].exists();
623.                     }
624.                     catch
625.                     {
626.                         child.edges[i].Add(aIn[i][j]);
627.                     }
628.                 }
629.             }
630.
631.             //update maxNode
632.             for (int i = 0; i < child.edges.Count; i++)
633.             {
634.                 for (int j = 0; j < child.edges[i].Count; j++)
635.                 {
636.                     int currentMax = (i > j) ? i : j;
```

```
637.                     child.maxNode = (currentMax > child.maxNode) ? currentMax : child.maxNode;
638.                 }
639.             }
640.             child.maxNode++;
641.
642.             //inherit nodes
643.             for (int i = 0; i < child.maxNode; i++)
644.             {
645.                 child.nodes.Add(fitterParent.nodes[i]);
646.             }
647.
648.             child.Reset();
649.             return child;
650.         }
651.     }
652.
653.     public class Brain
654.     {
655.         //base class for a 'brain' that learns through the evolution of a population of genomes
656.         public int inputs { get; }
657.         public int outputs { get; }
658.         public int populationSize { get; }
659.         public NEATHyperparameters hyperparameters { get; }
660.         public List<Specie> species;
661.         public int currentSpecies, currentGenome;
662.         public Genome globalBest;
663.         int generation = 0;
664.         public Brain(NEATHyperparameters input_NEAThyperparameters)
665.         {
666.             hyperparameters = input_NEAThyperparameters;
667.             inputs = hyperparameters.obsDim;
668.             outputs = hyperparameters.actDim;
669.             populationSize = input_NEAThyperparameters.populationSize;
670.             species = new();
671.             generation = 0;
672.             currentSpecies = 0;
673.             currentGenome = 0;
674.             globalBest = new Genome(inputs, outputs, hyperparameters.defaultActivation);
675.         }
676.
677.         public void Generate()
678.         {
679.             //generate iniital population of genomes
680.             for (int i = 0; i < populationSize; i++)
681.             {
682.                 Genome genome = new(inputs, outputs, hyperparameters.defaultActivation);
683.                 genome.GenerateNetwork();
684.                 ClassifyGenome(genome);
685.             }
686.         }
687.
688.         //Objective 5c
689.         //Genomic Distance
690.         public void ClassifyGenome(Genome genome)
691.         {
692.             //classify genome into a species via genomic distance algorithm
693.             //see if genome fits into any existing species
694.             foreach (Specie specie in species)
695.             {
696.                 Genome rep = specie.members[0];
697.                 float distance = GenomicDistance(genome, rep, hyperparameters.distanceWeights);
698.                 if (distance <= hyperparameters.deltaThreshold)
699.                 {
700.                     specie.members.Add(genome);
701.                     return;
702.                 }
703.             }
704.             //doesnt fit into any existing species, create new species
705.             List<Genome> listOfOneGenome = new() { genome };
706.             species.Add(new Specie(hyperparameters.maxFitnessHistory, listOfOneGenome));
707.         }
708.
709.         //Genomic Distance
710.         public float GenomicDistance(Genome a, Genome b, Dictionary<string, float> weights)
```

```
        {
            //calculate genomic distance between two genomes
            List<List<Edge>> aIn = (a.edges.Count > b.edges.Count) ? a.edges : b.edges;
            List<List<Edge>> bIn = (a.edges.Count > b.edges.Count) ? b.edges : a.edges;
            List<List<(Edge, Edge)>> matchingEdges = new();
            int noOfMatchingEdges = 0;
            int noOfDisjointEdges = 0;
            List<Edge> tempList;
            //calculate length of each genome
            int aLength = 0;
            int bLength = 0;
            foreach (var aList in aIn)
            {
                foreach (var item in aList)
                {
                    aLength++;
                }
            }
            foreach (var bList in bIn)
            {
                foreach (var item in bList)
                {
                    bLength++;
                }
            }
            int N_edges = (aLength > bLength) ? aLength : bLength; //choose the greatest length
            int N_nodes = (a.maxNode < b.maxNode) ? a.maxNode : b.maxNode; //choose the lesser length

            //create a list of edges that match between the two parents "homologous genees"
            for (int i = 0; i < aIn.Count; i++)
            {
                matchingEdges.Add(new List<(Edge, Edge)>());
                tempList = (aIn[i].Count < bIn[i].Count) ? aIn[i] : bIn[i]; //templist is the shorter of
both lists
                for (int j = 0; j < tempList.Count; j++)
                {
                    matchingEdges[i].Add((aIn[i][j], bIn[i][j]));
                    noOfMatchingEdges++;
                }
            }

            //find number of disjoint edges
            var aEdgesFlattened = aIn.SelectMany(x => x).ToList();
            var bEdgesFlattened = bIn.SelectMany(x => x).ToList();
            //find edges that are in a but not in b and vice versa
            var aMinusB = aEdgesFlattened.Except(bEdgesFlattened).ToList();
            var bMinusA = bEdgesFlattened.Except(aEdgesFlattened).ToList();
            //combine and take the length
            noOfDisjointEdges = aMinusB.Union(bMinusA).ToList().Count;

            //calculate difference in weights across every matching edge
            float weightDiff = 0;
            foreach (var list in matchingEdges)
            {
                foreach (var edgePair in list)
                {
                    weightDiff += Math.Abs(edgePair.Item1.weight - edgePair.Item2.weight);
                }
            }

            float biasDiff = 0;
            //calulate difference in biases across every matching node
            for (int i = 0; i < N_nodes; i++)
            {
                biasDiff += Math.Abs(a.nodes[i].bias - b.nodes[i].bias);
            }

            float c1 = weights["edge"] * noOfDisjointEdges / N_edges;
            float c2 = weights["weight"] * weightDiff / noOfMatchingEdges;
            float c3 = weights["bias"] * biasDiff / N_nodes;
            return c1 + c2 + c3;
        }

        public void UpdateFittest()
```

```
784.         {
785.             //update the highest performing genome in the population
786.
787.             //get top performers from each species
788.             List<Genome> topPerformers = new();
789.             foreach (Specie specie in species)
790.             {
791.                 topPerformers.Add(specie.GetBest());
792.             }
793.
794.             //if any in top performers better than current best, update
795.             foreach (Genome genome in topPerformers)
796.             {
797.                 globalBest = (genome.fitness > globalBest.fitness) ? genome : globalBest;
798.             }
799.         }
800.
801.         public void Evolve()
802.         {
803.             //the final meat of the program.
804.             //evolve the population of genomes by elimnating the poorest performing genomes
805.             //and repopulating with mutated children
806.             //prioritising the most promising species
807.
808.             //initalise currentGenome and currentSpecies for next round of training
809.             currentGenome = 0;
810.             currentSpecies = 0;
811.
812.             //get fitness sum across all species
813.             float globalFitnessSum = 0;
814.             foreach (Specie specie in species)
815.             {
816.                 specie.UpdateFitness();
817.                 globalFitnessSum += specie.fitnessSum;
818.             }
819.
820.             if (globalFitnessSum == 0)
821.             {
822.                 //no progress, mutate everybody
823.                 foreach (Specie specie in species)
824.                 {
825.                     foreach (Genome genome in specie.members)
826.                     {
827.                         genome.Mutate(hyperparameters.mutationProbabilities);
828.                     }
829.                 }
830.             }
831.             else
832.             {
833.                 //only keep species that have potential to improve
834.                 List<Specie> survivingSpecies = new();
835.                 foreach (Specie specie in species)
836.                 {
837.                     if (specie.CanProgress())
838.                     {
839.                         survivingSpecies.Add(specie);
840.                     }
841.                 }
842.                 species = survivingSpecies;
843.
844.                 //elimate the weakest genomes in each species
845.                 foreach (Specie specie in species)
846.                 {
847.                     specie.CullGenomes(false);
848.                 }
849.
850.                 //repopulate
851.                 foreach (Specie specie in species)
852.                 {
853.                     float ratio = specie.fitnessSum / globalFitnessSum; //species quality with respect to
global quality
854.                     float diff = populationSize - GetPopulationSize(); //how many extra genomes needed
855.                     int offspring = (int)Math.Round(ratio * diff); //how many offspring to create from
this species
```

```
856.                     for (int i = 0; i < offspring; i++)
857.                     {
858.                         Genome child = specie.Breed(hyperparameters.mutationProbabilities,
hyperparameters.breedProbabilities);
859.                         ClassifyGenome(child);
860.                     }
861.                 }
862.
863.                 //no species survived
864.                 //repopulate with mutated minimal structures and global best
865.                 if (species.Count == 0)
866.                 {
867.                     for (int i = 0; i < populationSize; i++)
868.                     {
869.                         Genome g;
870.                         if (i % 3 == 0)
871.                         {
872.                             g = globalBest;
873.                         }
874.                         else
875.                         {
876.                             g = new Genome(inputs, outputs, hyperparameters.defaultActivation);
877.                             g.GenerateNetwork();
878.                             g.Mutate(hyperparameters.mutationProbabilities);
879.                             ClassifyGenome(g);
880.                         }
881.                     }
882.                 }
883.             }
884.             generation++;
885.         }
886.
887.         public bool ShouldEvolve()
888.         {
889.             //determine if the system should continue to evolve
890.             //based on the maximum fitness and generation count
891.             UpdateFittest();
892.             bool fit = globalBest.fitness <= hyperparameters.maxFitness;
893.             bool end = generation != hyperparameters.maxGenerations;
894.
895.             return fit && end;
896.         }
897.
898.         public void NextIteration()
899.         {
900.             //call after every evaluation of individual genome to progress training
901.             Specie s = species[currentSpecies];
902.             if (currentGenome < s.members.Count - 1)
903.             {
904.                 currentGenome++; //move to next genome
905.             }
906.             else if (currentSpecies < species.Count - 1)
907.             {
908.                 currentSpecies++;
909.                 currentGenome = 0; //move onto next species
910.             }
911.             else
912.             {
913.                 currentGenome = 0;
914.                 currentSpecies = 0;
915.             }
916.
917.             //adapted from original -  Evolve() is now called from the program
918.         }
919.
920.         public int GetPopulationSize()
921.         {
922.             //return true (calculated) population size
923.             int size = 0;
924.             foreach (Specie specie in species)
925.             {
926.                 size += specie.members.Count;
927.             }
928.             return size;
```

```
        }

    }

    //Dictionary
    public class NEATHyperparameters : Hyperparameters //Polymorphism, Inheritance
    {
        public float deltaThreshold, maxFitness, maxGenerations;
        public int maxFitnessHistory, populationSize;

        public Activation defaultActivation;
        public Dictionary<string, float> distanceWeights, breedProbabilities, mutationProbabilities;

        public NEATHyperparameters()
        {
            deltaThreshold = 1.5f;
            maxFitnessHistory = 30;
            populationSize = 15;
            defaultActivation = new LReLU();
            maxFitness = 10000000;
            maxGenerations = 100000000;
            distanceWeights = new Dictionary<string, float>{
                {"edge" , 1.0f},
                {"weight" , 1.0f},
                {"bias" , 1.0f}
            };
            breedProbabilities = new Dictionary<string, float>{
                {"asexual" , 0.5f},
                {"sexual" , 0.5f}
            };
            mutationProbabilities = new Dictionary<string, float>{
                {"node" , 0.01f},
                {"edge" , 0.09f},
                {"weight perturb" , 0.4f},
                {"weight set" , 0.1f},
                {"bias perturb", 0.3f},
                {"bias set", 0.1f}
            };
        }
    }
}

```

LearningAlgorithms.cs

```
//Objectives 3, 4, 5
//DQN
//PPO
//NEAT
//OOP
//Polymorphism
//Inheritance
//Procedures & Functions
//Lists

//MY IMPLEMENTATIONS!!
using Hyperparamters;
using DQNImplementaion;
using PPOImplementation;
using NEATImplementation;

//math & conversions
using NumSharp;

namespace Hyperparamters
{
    public abstract class Hyperparameters
    {
        public int obsDim;
        public int actDim;
        public float activationThreshold;
    }
```

```
28. }
29.
30. namespace LearningAlgorithms
31. {
32.
33.     public abstract class LearningAlgorithm
34.     {
35.         public Hyperparameters algorithmParam;
36.         public string algorithmID;
37.         public abstract bool[] GetOutput(float[] state, bool done, float reward);
38.         public abstract void Learn(int currentRound);
39.     }
40.     #region DQN
41.     public class DQNAlgorithm : LearningAlgorithm
42.     {
43.         //inits
44.         public DQN DQNAgent;
45.         public DQNAlgorithm(DQNHyperparameters inputHyperparameters) {
46.             algorithmParam = inputHyperparameters;
47.             algorithmID = "DQN";
48.             DQNAgent = new DQN(inputHyperparameters);
49.         }
50.         //Objective 3a
51.         //Objective 3b
52.         //get dqn action
53.         public override bool[] GetOutput(float[] state, bool done, float reward){
54.             //cast param to DQN type to check if memory buffer is too large
55.             if (algorithmParam is DQNHyperparameters algorithmDQNParam && DQNAgent.memoryBuffer.Count >=
algorithmDQNParam.memoryBufferSize){
56.                 DQNAgent.memoryBuffer.RemoveAt(0);
57.             }
58.
59.             //get outputs
60.             var algorithmsOutputs = DQNAgent.ComputeAction(np.array(state));
61.
62.             //create new episode and store
63.             DQN.Epsiode currentEpsiode = new()
64.             {
65.                 currentState = np.array(state),
66.                 action = algorithmsOutputs
67.             };
68.             DQNAgent.memoryBuffer.Add(currentEpsiode);
69.
70.             //edit past episode
71.             DQNAgent.memoryBuffer[^1].nextState = np.array(state);
72.             return algorithmsOutputs;
73.         }
74.
75.         public override void Learn(int currentRound)
76.         {
77.             DQNAgent.Train();
78.             DQNAgent.UpdateExplorationProbability();
79.         }
80.     }
81.     #endregion
82.     #region NEAT
83.     public class NEATAlgorithm : LearningAlgorithm
84.     {
85.         public Brain NEATBrain;
86.         public NEATAlgorithm(NEATHyperparameters inputHyperparameters){
87.             algorithmParam = inputHyperparameters;
88.             algorithmID = "NEAT";
89.             NEATBrain = new(inputHyperparameters);
90.         }
91.
92.         //Objective 5a
93.         public override bool[] GetOutput(float[] state, bool done, float reward){
94.             //get output from selected member of selected species
95.             var output =
NEATBrain.species[NEATBrain.currentSpecies].members[NEATBrain.currentGenome].Forward(state);
96.
97.             //convert to bool using activation threshold & return
98.             bool[] outputBool = new bool[5];
99.             for(int i = 0; i < 5; i ++){
```

```
100.                 outputBool[i] = output[i] > algorithmParam.activationThreshold;
101.             }
102.             return outputBool;
103.         }
104.
105.         public override void Learn(int currentRound)
106.         {
107.             NEATBrain.Evolve();
108.         }
109.
110.         public void IncrementAgents(){
111.             NEATBrain.NextIteration();
112.         }
113.     }
114.     #endregion
115.
116.     #region PPO
117.     public class PPOAlgorithm : LearningAlgorithm
118.     {
119.         public PPO PPOAgent;
120.         public PPO.Epsiode currentEpisode;
121.         public PPOAlgorithm (PPOHyperparameters inputHyperparameters){
122.             algorithmParam = inputHyperparameters;
123.             algorithmID = "PPO";
124.             PPOAgent = new PPO (inputHyperparameters);
125.         }
126.
127.         //Objective 4a
128.         //Objective 4b
129.         public override bool[] GetOutput(float[] state, bool done, float reward)
130.         {
131.             //get output from PPO
132.             var output = PPOAgent.Forward(np.array(state), null);
133.
134.             //convert to bool
135.             bool[] actionOutput = new bool[5];
136.             int i = 0;
137.             foreach (float outputValue in output.Item1){
138.                 actionOutput[i] = outputValue > algorithmParam.activationThreshold;
139.             }
140.
141.             //episode adds + small amount of admin req for them
142.             currentEpisode.epActs.Add(output.Item1);
143.             float doneFloat = (done) ? 1 : 0;
144.             currentEpisode.epDones.Add(doneFloat);
145.             currentEpisode.epLogProbs.Add(output.Item2);
146.             //replace the one before the current with the current state, and refill current with zero to
be replaced
147.             currentEpisode.epNextObs[^1] = np.array(state);
148.             currentEpisode.epNextObs.Add(np.zeros(1));
149.             currentEpisode.epObs.Add(np.array(state));
150.             currentEpisode.epRewards.Add(reward);
151.             //epVals, epLen to be done in Learn()
152.             return actionOutput;
153.
154.         }
155.
156.         public override void Learn(int currentRound)
157.         {
158.             //calculate epLen and epObs
159.             currentEpisode.epLen = currentEpisode.epActs.Count;
160.             foreach(NDArray obs in currentEpisode.epObs){
161.                 currentEpisode.epVals.Add(obs);
162.             }
163.
164.             //add finalised episode to batch
165.             PPOAgent.AddEpisodeToBatch(currentEpisode);
166.
167.             //learn
168.             PPOAgent.Learn(currentRound);
169.         }
170.     }
171.     #endregion
172. }
```

FCNN.cs

```
//Objective 6
//OOP
//Inheritance
//Polymorphism
//Neural Network
//Vectors
//Arrays

//math
using NumSharp;

//TFPPOModel ONLY!! CreateModel and its child classes DO NOT USE!!
using TorchSharp;
using static TorchSharp.torch.nn;

//system usings
using System;

namespace FCNN
{
    #region activations
    //Objective 6c
    //activation functions
    public class Activation //parent class
    {
        public virtual float DoActivation(float x)
        {
            return x;
        }

        public virtual NDArray DoActivationArray(NDArray x)
        {
            return x;
        }

        public virtual NDArray DerivativeActivationArray(NDArray x, NDArray outputs)
        {
            return x;
        }
    }

    //Objectives 6ci\1, 6cii\1
    //Arrays
    public class ReLU : Activation //child classes
    {
        public override float DoActivation(float x)
        {
            return MathF.Max(0, x);
        }

        public override NDArray DoActivationArray(NDArray x)
        {

            return np.maximum(np.zeros(x.shape), x);
        }

        public override NDArray DerivativeActivationArray(NDArray x, NDArray outputs)
        {
            var mask = np.zeros(outputs.shape);

            for (int i = 0; i < outputs.shape[0]; i++)
                for (int j = 0; j < outputs.shape[1]; j++)
                {
                    float value = outputs[i, j];
                    mask[i, j] = (value > 0) ? 1 : 0;
                }


            return np.multiply(x, mask);
        }
```

```
    }

    //Objectives 6ci\3, 6cii\3
    //Arrays
    //Vectors
    public class Softmax : Activation{
        public override NDArray DoActivationArray(NDArray x)
        {
            NDArray output = np.zeros(x.shape);

            // Exponentiate each element (e^x)
            var expX = np.exp(x);

            // Create an array to store the sum of each row
            var sumExp = new double[expX.shape[0]];

            // Manually calculate the sum of each row
            for (int i = 0; i < expX.shape[0]; i++)
            {
                double rowSum = 0.0;
                for (int j = 0; j < expX.shape[1]; j++)
                {
                    rowSum += expX[i, j];
                }
                sumExp[i] = rowSum;
            }

            // Normalize each element by the row sum
            for (int i = 0; i < expX.shape[0]; i++)
            {
                for (int j = 0; j < expX.shape[1]; j++)
                {
                    output[i, j] = expX[i, j] / sumExp[i];
                }
            }

            return output;
        }

        public override NDArray DerivativeActivationArray(NDArray x, NDArray outputs)
        {
            NDArray derivative = np.zeros(x.shape);
            for (int i = 0; i < derivative.shape[0]; i++){
                NDArray gradient = derivative[i];

                if (gradient.shape.Length == 0){ //for single instance
                    gradient.reshape(-1, 1);
                }
                //create diagflat becaues numsharp does not have diagflat function
                NDArray diagflat = np.zeros(gradient.shape[0], gradient.shape[0]);
                for (int j = 0; j < diagflat.shape[0]; j++){
                    diagflat[j, j] = gradient[j];
                }
                //do dot product manually because numsharp doesnt like it
                //Vectors
                NDArray gradientDotProduct = np.zeros(gradient.shape[0], gradient.shape[0]);
                for (int j = 0; j < gradientDotProduct.shape[0]; j++){
                    for (int k = 0; k < gradientDotProduct.shape[1]; k++){
                        gradientDotProduct[j, k] = gradient[j] * gradient[k];
                    }
                }

                //final calculations
                NDArray jacobian = diagflat - gradientDotProduct;
                derivative[i] = np.dot(jacobian, outputs[i].reshape(outputs[i].shape[0], 1));
            }
            return derivative;
        }
    }

    //Objective 6ci\2, 6cii\2
    //Arrays
    public class LReLU : Activation
    {
```

```
145.         public override float DoActivation(float x)
146.         {
147.             //if greater than 0, return value.
148.             //else, return small amount of value.
149.             if (x >= 0) {
150.                 return x;
151.             } else {
152.                 return 0.01f * x;
153.             }
154.         }
155.
156.         public override NDArray DerivativeActivationArray(NDArray x, NDArray outputs)
157.         {
158.             var mask = np.zeros(outputs.shape);
159.
160.             for (int i = 0; i < outputs.shape[0]; i++)
161.                 for (int j = 0; j < outputs.shape[1]; j++)
162.                 {
163.                     float value = outputs[i, j];
164.                     mask[i, j] = (value > 0) ? 1 : 0.01; //like ReLU, but instead of going to 0 we go to
0.01
165.                 }
166.
167.             return np.multiply(x, mask);
168.         }
169.
170.         public override NDArray DoActivationArray(NDArray x)
171.         {
172.             NDArray returnArray = np.zeros(x.shape);
173.             //for each value in input array, do activation
174.             for(int i = 0; i < x.shape[0]; i++){
175.                 for(int j = 0; j < x.shape[1]; j++){
176.                     returnArray[i, j] = DoActivation(x[i, j]);
177.                 }
178.             }
179.             return returnArray;
180.         }
181.     }
182.
183.     //Objective 6cii\4
184.     public class Sigmoid : Activation
185.     {
186.         public override float DoActivation(float x)
187.         {
188.             float result = 1 / (1 + MathF.Exp(-x));
189.             return result;
190.         }
191.     }
192.
193.     //Objective 6cii\5
194.     public class Tanh : Activation
195.     {
196.         public override float DoActivation(float x)
197.         {
198.             float result = MathF.Tanh(x);
199.             return result;
200.         }
201.     }
202.     #endregion
203.
204.     #region FCLayer
205.
206.     /* Implements MLP with Adam and other cool features, by using one class for a layer and another to
fully construct the network.
207.      * this is useful because we may require multiple different types of NN (ie A2N, PPO strategies*/
208.     public class FCLayer
209.     {
210.         public FCLayer(int inputSizeInput, int outputSizeInput, Activation activationInput)
211.         {
212.             inputSize = inputSizeInput;
213.             outputSize = outputSizeInput;
214.             activation = activationInput;
215.             weights = np.random.randn(inputSize, outputSize) * np.sqrt(2.0 / inputSize); //initialise
weights according to HE-Initialisation
```

```
216.             biases = np.zeros(1, outputSize);
217.             mWeights = np.zeros(inputSize, outputSize);
218.             vWeights = np.zeros(inputSize, outputSize);
219.             mBiases = np.zeros(1, outputSize);
220.             vBiases = np.zeros(1, outputSize);
221.             //initialise hyperparameters for Adam optimiser
222.             beta1 = 0.9f;
223.             beta2 = 0.999f;
224.             epsilon = 1e-8f;
225.         }
226.         public int inputSize { get; }
227.         public int outputSize { get; }
228.         public Activation activation { get; }
229.         //regular weights and biases
230.         public NDArray weights;
231.         private NDArray biases;
232.         //derivatives of weights and biases
233.         public NDArray dWeights;
234.         private NDArray dBiases;
235.         /* Define m & v for weights and biases
236.          * These variables are used in Adam optimisation */
237.         private NDArray mWeights;
238.         private NDArray mBiases;
239.         private NDArray vWeights;
240.         private NDArray vBiases;
241.         private NDArray mHatBiases;
242.         private NDArray vHatBiases;
243.         private NDArray mHatWeights;
244.         private NDArray vHatWeights;
245.         //hyperparameters for Adam - ADD AS PARAMETERS FOR NN?
246.         private readonly float beta1;
247.         private readonly float beta2;
248.         private readonly float epsilon;
249.         public NDArray output;
250.         private NDArray x;
251.         //Objective 6aii
252.         public NDArray Forward(NDArray X) //x is a layer input array to perform the forward pass on
253.         {
254.             x = X;
255.             NDArray z;
256.             //calculate the layer output z
257.             z = biases + np.dot(x, weights); //Objective 6aii\1
258.             //remove linearity using activation function
259.             output = activation.DoActivationArray(z); //Objective 6aii\2
260.             return output;
261.         }
262.
263.         private NDArray dInputs;
264.         //Objective 6bii
265.         //Neural Network
266.         //Adam
267.         public NDArray Backward(NDArray dValues, float lr, int t) //dValues is the derivative of the
output, t is the timestep
268.         {
269.             dValues = activation.DerivativeActivationArray(dValues, output); //Objective 6bii\1
270.
271.             //Objective 6bii\2
272.             //calculte derivatives wrt weight and bias
273.             dWeights = np.dot(x.T, dValues);
274.             dBiases = SumOverAxis0(dValues);
275.             //limit derivatives to avoid really big or really small numbers
276.             dWeights = np.clip(dWeights, -1, 1);
277.             dBiases = np.clip(dBiases, -1, 1);
278.
279.             //Objective 6bii\3
280.             //calculate gradient wrt to inputs
281.             dInputs = np.dot(dValues, weights.T);
282.
283.             //Objective 6bii\4
284.             //update weights and biases using learning rate and derivatives
285.             weights -= lr * dWeights;
286.             biases -= lr * dBiases;
287.
288.             //Objective 6bii\5
```

```
289.             //Adam
290.             //update weights using m and v values (Adam)
291.             mWeights = beta1 * mWeights + (1 - beta1) * dWeights;
292.             vWeights = beta2 * vWeights + (1 - beta2) * np.power(dWeights, 2);
293.             mHatWeights = mWeights / (1 - np.power(beta1, t));
294.             vHatWeights = vWeights / (1 - np.power(beta2, t));
295.             weights -= lr * mHatWeights / (np.sqrt(vHatWeights) + epsilon);
296.
297.             //update biases using m and v values (Adam)
298.             mBiases = beta1 * mBiases + (1 - beta1) * dBiases;
299.             vBiases = beta2 * vBiases + (1 - beta2) * np.power(dBiases, 2);
300.             mHatBiases = mBiases / (1 - np.power(beta1, t));
301.             vHatBiases = vBiases / (1 - np.power(beta2, t));
302.             biases -= lr * mHatBiases / (np.sqrt(vHatBiases) + epsilon);
303.
304.             //Objective 6bii\6
305.             return dInputs;
306.         }
307.         static NDArray SumOverAxis0(NDArray array) //fine... i'll do it myself. implements np.sum(array,
axis:0, keepDims = True).
308.         {
309.             int rows = array.shape[0];
310.             int cols = array.shape[1];
311.             NDArray result = np.zeros(cols);
312.             for (int i = 0; i < rows; i++)
313.             {
314.                 for (int j = 0; j < cols; j++)
315.                 {
316.                     result[j] += array[i, j];
317.                 }
318.             }
319.
320.             return result.reshape(1, cols);
321.         }
322.     }
323.
324.     #endregion
325.
326.     #region CreateModel
327.
328.     public class CreateModel
329.     {
330.
331.         public int inputSize { get; set; }
332.         public int outputSize { get; set;}
333.         public int[] hiddenSizes { get; set;}
334.         public FCLayer layer1 { get; set; }
335.         public FCLayer layer2 { get; set; }
336.         public FCLayer layer3 { get; set; }
337.         public CreateModel(int inputSizeInput, int outputSizeInput, int[] hiddenSizesInput)
338.
339.         {
340.             inputSize = inputSizeInput;
341.             outputSize = outputSizeInput;
342.             hiddenSizes = hiddenSizesInput;
343.
344.             layer1 = new FCLayer(inputSize, hiddenSizes[0], new ReLU());
345.             layer2 = new FCLayer(hiddenSizes[0], hiddenSizes[1], new ReLU());
346.             layer3 = new FCLayer(hiddenSizes[1], outputSize, new Softmax());
347.         }
348.
349.         //Objective 6ai
350.         public NDArray Forward(NDArray inputs)
351.         {
352.             //Objective 6ai\1
353.             NDArray output1 = layer1.Forward(inputs);
354.             NDArray output2 = layer2.Forward(output1);
355.             NDArray output3 = layer3.Forward(output2);
356.
357.             return output3;
358.         }
359.
360.         private int t = 0;
361.         private float lr;
```

```
362.         private NDArray outputGrad;
363.         private NDArray grad3;
364.         private NDArray grad2;
365.         private NDArray grad1;
366.         //Objective 6bi
367.         public void Train(NDArray inputs, NDArray targets, int nEpochs, float initialLr, float decay)
368.         {
369.             for (int epoch = 0; epoch < nEpochs; epoch++)
370.             {
371.                 //forward pass
372.                 NDArray output = Forward(inputs); //Objective 6bi\1
373.
374.                 //backwards pass
375.                 outputGrad = 6 * (output - targets) / output.shape[0]; //Objective 6bi\2
376.                 t++;
377.                 lr = initialLr / (1 + decay * epoch); //Objective 6bi\3
378.                 //Objective 6bi\4
379.                 grad3 = layer3.Backward(outputGrad, lr, t);
380.                 grad2 = layer2.Backward(grad3, lr, t);
381.                 grad1 = layer1.Backward(grad2, lr, t);
382.             }
383.         }
384.     }
385.     #endregion
386.     #region PPOModel
387.     //OOP
388.     public class PPOModel : CreateModel //Polymorphism, Inheritance
389.     {
390.         public PPOModel(int inputSize, int outputSize, int[] hiddenSizes) : base(inputSize, outputSize,
hiddenSizes)
391.         {
392.             base.inputSize = inputSize;
393.             base.outputSize = outputSize;
394.             base.hiddenSizes = hiddenSizes;
395.         }
396.     }
397.     #endregion
398.     #region DQN
399.     //OOP
400.     public class DQNModel : CreateModel //Polymorphism, Inheritance
401.     {
402.         public DQNModel(int inputSize, int outputSize, int[] hiddenSizes) : base(inputSize, outputSize,
hiddenSizes)
403.         {
404.             base.inputSize = inputSize;
405.             base.outputSize = outputSize;
406.             base.hiddenSizes = hiddenSizes;
407.         }
408.
409.     }
410.
411.     #endregion
412.     #region TFPPO
413.
414.     public class TFPPOModel : Module
415.     {
416.         public int inDim;
417.         public int outDim;
418.         public TorchSharp.Modules.Linear layer1, layer2, layer3;
419.         public TFPPOModel(int inDimInputs, int outDimInputs, int[] hiddenDimInputs) : base("TFPPOModel"){
420.             inDim = inDimInputs;
421.             outDim = outDimInputs;
422.             layer1 = torch.nn.Linear(inDim, hiddenDimInputs[0]);
423.             layer2 = torch.nn.Linear(hiddenDimInputs[0], hiddenDimInputs[1]);
424.             layer3 = torch.nn.Linear(hiddenDimInputs[1], outDim);
425.         }
426.         public NDArray forward(NDArray inputs){
427.             //convert to tensor for nn
428.             torch.Tensor inputsTens = torch.tensor((long)inputs, torch.ScalarType.Float64, null, false);
429.             //forward through nn
430.             var activation1 = functional.relu(layer1.forward(inputsTens));
431.             var activation2 = functional.relu(layer2.forward(activation1));
432.             var output = layer3.forward(activation2);
433.             //convert to float
```

```
            float[] netArray = output.data<float>().ToArray();
            //convert to ndarray and return
            return np.array(netArray);
        }
    }

    #endregion

}
```

StatsDrawer.cs

```
//Objective 8c
//Unity UI
//Read from File

using System.IO;
using System.Collections.Generic;
using TMPro;
using UnityEngine;
using System.Linq;

public class StatsDrawer : MonoBehaviour
{
    public string filepath;
    public TMP_Dropdown dropdown;
    public GraphHandler graphHandler;
    private List<int> roundNumbers;
    private List<float>[] stats;
    int numLines = 1;

    // Start is called once before the first execution of Update after the MonoBehaviour is created
    void Start()
    {
        //add listener for dropdown
        dropdown.onValueChanged.AddListener(OnDropdownValueChanged);
        //intialise stats array of lists
        List<float> emptyList = new();
        stats = new List<float>[5] { emptyList, emptyList, emptyList, emptyList, emptyList };
    }

    //
    void OnDropdownValueChanged(int index)
    {
        UpdateGraph(index);
    }

    //Objective 8cii
    void UpdateGraph(int index)
    {
        for (int i = 0; i < numLines; i++)
        {
            graphHandler.ChangePoint(i, new Vector2(roundNumbers[i], stats[index][i]));
        }
    }

    //Read from File
    public void ParseStats()
    {
        using StreamReader sr = new(filepath);

        //read first line
        string line = sr.ReadLine();

        //continue to read lines
        while (line != null)
        {
            //split up into individual values
            string[] lineValues = line.Split(",");
            for (int i = 0; i < lineValues.Count(); i++)
            {
                roundNumbers.Add(int.Parse(lineValues[0]));
                stats[i].Add(float.Parse(lineValues[i + 1]));
            }
```

```
63.
64.             //add point to graph to edit when needed
65.             graphHandler.CreatePoint(new Vector2(numLines, 0));
66.
67.             //iterate count and read next line
68.             numLines++;
69.             line = sr.ReadLine();
70.         }
71.     }
72. }
73.
74.
```

DraggablePanel.cs

```
1. //Objective 1ciii
2. //Unity UI
3.
4. using UnityEngine;
5. using UnityEngine.UI;
6. using TMPro;
7. using UnityEngine.EventSystems;
8. using System.Collections.Generic;
9. using Unity.VisualScripting;
10.
11. public class DraggablePanel : MonoBehaviour
12. {
13.     [Header("references")]
14.     [SerializeField] private RectTransform rectTransform;
15.     [SerializeField] private TMP_Text headerText;
16.     [SerializeField] private TMP_Text bodyText;
17.     [SerializeField] private Button closeButton;
18.     [SerializeField] private RectTransform dragHandle;
19.     [SerializeField] private CanvasGroup canvasGroup;
20.
21.     [Header("animation settings")]
22.     [SerializeField] private float animationDuration = 0.3f;
23.     [SerializeField] private AnimationCurve fadeInCurve = AnimationCurve.EaseInOut(0, 0, 1, 1);
24.     [SerializeField] private AnimationCurve fadeOutCurve = AnimationCurve.EaseInOut(0, 1, 1, 0);
25.     [SerializeField] private AnimationCurve scaleCurve = AnimationCurve.EaseInOut(0, 0, 1, 1);
26.
27.     private bool isDragging = false;
28.     private bool isAnimating = false;
29.     private Canvas canvas;
30.     private Vector3 originalScale;
31.
32.     void Awake()
33.     {
34.         //get canvas
35.         canvas = GetComponentInParent<Canvas>();
36.
37.         //add listener to close button
38.         closeButton.onClick.AddListener(() => DestroyPanel());
39.
40.         //set original scale
41.         originalScale = rectTransform.localScale;
42.
43.         //set grabhandle to move
44.         EventTrigger trigger = dragHandle.gameObject.GetComponent<EventTrigger>();
45.
46.         //setup drag handler
47.
48.         //begin drag
49.         EventTrigger.Entry beginDrag = new()
50.         {
51.             eventID = EventTriggerType.BeginDrag
52.         };
53.         beginDrag.callback.AddListener((data) => OnBeginDrag((PointerEventData)data));
54.         trigger.triggers.Add(beginDrag);
55.
56.         //during drag
57.         EventTrigger.Entry drag = new()
58.         {
59.             eventID = EventTriggerType.Drag
```

```
60.         };
61.         drag.callback.AddListener((data) => OnDrag((PointerEventData)data));
62.         trigger.triggers.Add(drag);
63.
64.         //end drag
65.         EventTrigger.Entry endDrag = new()
66.         {
67.             eventID = EventTriggerType.EndDrag
68.         };
69.         endDrag.callback.AddListener((data) => OnEndDrag((PointerEventData)data));
70.         trigger.triggers.Add(endDrag);
71.
72.         //start with panel invisible
73.         canvasGroup.alpha = 0;
74.         rectTransform.localScale = Vector3.zero;
75.
76.         //entrance animation
77.         StartCoroutine(AnimateEntrance());
78.     }
79.
80.     //let other processes set texts
81.     public void SetHeader(string text){
82.         headerText.text = text;
83.     }
84.
85.     public void SetBody(string text){
86.         bodyText.text = text;
87.     }
88.
89.     //destroy
90.     public void DestroyPanel(){
91.         if (!isAnimating){
92.             StartCoroutine(AnimateExit());
93.         }
94.     }
95.
96.     //drag events
97.     private void OnBeginDrag(PointerEventData eventData){
98.         isDragging = true;
99.     }
100.
101.    private void OnDrag(PointerEventData eventData){
102.        if (!isDragging) return;
103.
104.        //change position wrt to start of drag
105.        rectTransform.anchoredPosition += eventData.delta / canvas.scaleFactor;
106.    }
107.
108.    private void OnEndDrag(PointerEventData eventData){
109.        isDragging = false;
110.    }
111.
112.    private IEnumerator<Null> AnimateEntrance(){
113.        isAnimating = true;
114.        float elapsed = 0;
115.
116.        while(elapsed < animationDuration){
117.            //figure out progress through animation
118.            elapsed += Time.deltaTime;
119.            float progress = elapsed/animationDuration;
120.
121.            //animate opacity
122.            canvasGroup.alpha = fadeInCurve.Evaluate(progress);
123.
124.            //animate scale
125.            float scale = scaleCurve.Evaluate(progress);
126.            rectTransform.localScale = scale * originalScale;
127.
128.            yield return null;
129.        }
130.
131.        //ensure final values are set
132.        canvasGroup.alpha = 1;
133.        rectTransform.localScale = originalScale;
```

```
134.
135.         isAnimating = false;
136.     }
137.
138.     private IEnumerator<Null> AnimateExit(){
139.         isAnimating = true;
140.         float elapsed = 0;
141.
142.         while (elapsed < animationDuration){
143.             elapsed += Time.deltaTime;
144.             float progress = elapsed / animationDuration;
145.
146.             //animate opacity
147.             canvasGroup.alpha = fadeOutCurve.Evaluate(progress);
148.
149.             //animate scale
150.             float scale = scaleCurve.Evaluate(1 - progress);
151.             rectTransform.localScale = scale * originalScale;
152.
153.             yield return null;
154.         }
155.
156.         //actually destroy panel
157.         Destroy(gameObject);
158.     }
159. }
160.
```

UInterfaceManager.cs

```
 1. //Objectives 1, 2
 2. //Unity UI
 3. //Arrays
 4. //Coroutines
 5.
 6. using UnityEngine;
 7. using UnityEngine.UI;
 8. using TMPro;
 9. using System.Collections.Generic;
10. using Unity.VisualScripting;
11.
12. public class UInterfaceManager : MonoBehaviour
13. {
14.     [Header("screens")]
15.     public GameObject[] screens; //array of screens to switch between as program progresses
16.     //current screen index
17.     public int screenIndex = 0;
18.     [Header("screen animation settings")]
19.     [SerializeField] private float animationDuration = 0.5f;
20.     [SerializeField] private float moveDistance = 1000f;
21.     [SerializeField] private AnimationCurve easingCurve = AnimationCurve.EaseInOut(0, 0, 1, 1);
22.     [SerializeField] private AnimationCurve fadeCurve = AnimationCurve.EaseInOut(0, 0, 1, 1);
23.     [Header("buttons")] [Space]
24.     //buttons to switch between config screens
25.     public Button nextButton;
26.     public Button backButton;
27.     //button to start simulation
28.     public Button startButton;
29.     [Header("managers & stats drawer")] [Space]
30.     //gameManager to manage simulation
31.     public GameManager gameManager;
32.     //panelsManager for UI panels
33.     public PanelManager panelManager;
34.     //stats drawer to control stats on final screen
35.     public StatsDrawer statsDrawer;
36.     //validators to ensure correct input before simulation started
37.     [Header("other stuff")][Space]
38.     public TakeFieldsInput[] validators;
39.     //dropdownManagers to get validators for agent configs
40.     public DropdownHandler runnerDropdownHandler;
```

```
41.     public DropdownHandler taggerDropdownHandler;
42.     //error text for user
43.     public TMP_Text errorText;
44.     //for config control items
45.     public GameObject configControl;
46.     //button to start stats screen
47.     public Button statsButton;
48.     //button to pause game
49.     public Button pauseButton;
50.     //button to resume game
51.     public Button resumeButton;
52.     //button to toggle stats while simulation is running
53.     public Button statsToggleButton;
54.     private bool isAnimating;
55.     private RectTransform[] screenRects;
56.     private CanvasGroup[] screenGroups;
57.
58.     // Start is called once before the first execution of Update after the MonoBehaviour is created
59.     void Start()
60.     {
61.         //initalise
62.         gameManager.gameRunning = false;
63.         screenRects = new RectTransform[screens.Length];
64.         screenGroups = new CanvasGroup[screens.Length];
65.         int i = 0;
66.         foreach (GameObject screen in screens) {
67.             screen.SetActive(false);
68.             screenRects[i] = screen.GetComponent<RectTransform>();
69.             screenGroups[i] = screen.GetComponent<CanvasGroup>();
70.             i++;
71.         }
72.         screens[0].SetActive(true);
73.         configControl.SetActive(true);
74.         validators = new TakeFieldsInput[3];
75.
76.         //add listeners for buttons
77.         nextButton.onClick.AddListener(NextScreen);
78.         backButton.onClick.AddListener(PreviousScreen);
79.         startButton.onClick.AddListener(StartSimulationChecks);
80.         pauseButton.onClick.AddListener(PauseGame);
81.         resumeButton.onClick.AddListener(ResumeGame);
82.
83.         //add validator for simulation settings will stay the same, so set it now
84.         validators[2] = screens[2].GetComponent<TakeFieldsInput>();
85.     }
86.
87.     //Objective 1aii
88.     //Coroutines
89.     void NextScreen()
90.     {
91.         if (0 <= screenIndex && screenIndex < 2 && !isAnimating) {
92.             //deactivate current screen and activate next screen
93.             StartCoroutine(SwitchScreens(screenIndex, screenIndex + 1, true));
94.             screenIndex++;
95.         }
96.     }
97.
98.     private IEnumerator<Null> SwitchScreens (int oldIndex, int newIndex, bool slideRight) {
99.         isAnimating = true;
100.
101.         //set initial pos
102.         screens[newIndex].SetActive(true);
103.         Vector2 oldStartPos = screenRects[oldIndex].anchoredPosition;
104.         Vector2 newStartPos = oldStartPos;
105.         newStartPos.x = slideRight ? +moveDistance : -moveDistance;
106.         screenRects[newIndex].anchoredPosition = newStartPos;
107.
108.         //set initial alphas
109.         screenGroups[oldIndex].alpha = 1;
110.         screenGroups[newIndex].alpha = 0;
111.
112.         //calculate end positions
113.         Vector2 oldEndPos = oldStartPos;
114.         oldEndPos.x += slideRight ? -moveDistance : +moveDistance;
```

```
115.         Vector2 newEndPos = Vector2.zero;
116.
117.         //move old screen out
118.         float elapsed = 0;
119.         while (elapsed < animationDuration){
120.             //calculate progress and curve values
121.             elapsed += Time.deltaTime;
122.             float progress = elapsed/animationDuration;
123.             float curveValue = easingCurve.Evaluate(progress);
124.             float fadeValue = fadeCurve.Evaluate(progress);
125.
126.             //update positions
127.             screenRects[oldIndex].anchoredPosition = Vector2.Lerp(oldStartPos, oldEndPos, curveValue);
128.
129.             //update fade
130.             screenGroups[oldIndex].alpha = Mathf.Lerp(1f, 0f, fadeValue);
131.             yield return null;
132.         }
133.
134.         //set final values for old screen
135.         screenRects[oldIndex].anchoredPosition = oldEndPos;
136.         screenGroups[oldIndex].alpha = 0f;
137.         screens[oldIndex].SetActive(false);
138.
139.         //move new screen in
140.         elapsed = 0;
141.         while (elapsed < animationDuration){
142.             //calculate progress and curve values
143.             elapsed += Time.deltaTime;
144.             float progress = elapsed/animationDuration;
145.             float curveValue = easingCurve.Evaluate(progress);
146.             float fadeValue = fadeCurve.Evaluate(progress);
147.
148.             //update positions
149.             screenRects[newIndex].anchoredPosition = Vector2.Lerp(newStartPos, newEndPos, curveValue);
150.
151.             //update fade
152.             screenGroups[newIndex].alpha = Mathf.Lerp(0f, 1f, fadeValue);
153.
154.             yield return null;
155.         }
156.
157.         //set final values for new screen
158.         screenRects[newIndex].anchoredPosition = newEndPos;
159.         screenGroups[newIndex].alpha = 1f;
160.
161.         isAnimating = false;
162.     }
163.
164.     //Objective 1aii
165.     //Coroutines
166.     void PreviousScreen()
167.     {
168.         if (2 >= screenIndex && screenIndex > 0 && !isAnimating) {
169.         //deactivate current screen and activate previous screen
170.         //deactivate current screen and activate next screen
171.             StartCoroutine(SwitchScreens(screenIndex, screenIndex - 1, false));
172.             screenIndex--;
173.         }
174.     }
175.
176.     //Objective 2d
177.     void StartSimulationChecks()
178.     {
179.         if (screenIndex == 2) {
180.             //get appropriate validators
181.             validators[0] = runnerDropdownHandler.chosenValidator;
182.             validators[1] = taggerDropdownHandler.chosenValidator;
183.             //check if is all valid
184.             if (validators[0].isAllValid && validators[1].isAllValid && validators[2].isAllValid) {
185.                 Debug.Log("START SIM");
186.                 StartSimulation();
187.             } else {
188.                 //show error message
```

```
                Debug.Log("Invalid input");
                errorText.text = $"Invalid input. For each section:\n " +
                    $"Runner Params: {validators[0].isAllValid} \n" +
                    $"Tagger Params: {validators[1].isAllValid} \n" +
                    $"Simulation Params: {validators[2].isAllValid} \n"
                    ;
            }
        } else {
            errorText.text = "Please complete all sections before starting simulation";
        }
    }

    void StartSimulation(){
        //remove panels
        panelManager.DestroyAllPanels();

        //deactivate current screen and activate next screen
        configControl.SetActive(false);
        screens[screenIndex].SetActive(false);
        screenIndex++;
        screens[screenIndex].SetActive(true);

        //start simulation
        foreach (TakeFieldsInput validator in validators) {
            validator.TakeInputs();
        }
        gameManager.InitaliseSimulation(validators[2].inputs, validators[1].inputs, validators[0].inputs);
        gameManager.gameRunning = true;
    }

    private void PauseGame()
    {
        gameManager.roundPaused  = true;
    }

    private void ResumeGame()
    {
        gameManager.gameRunning = true;
    }

    public void GraphOverview(){
        //deactivate current screen
        screens[screenIndex].SetActive(false);

        //activate final stats panel
        screenIndex++;
        screens[screenIndex].SetActive(true);

        //draw graphs
        statsDrawer.filepath = gameManager.filepath;
        statsDrawer.ParseStats();
    }
}
```

GameManager.cs

```
//Objectives 2, 7, 8
//Procedures & Functions
//Unity 3D Simulation
//Unity UI System
//Arrays
//Dictionary
//Write to File

//unity/system usings
using UnityEngine;
using TMPro;
using System.Collections.Generic;

//OOP usings
```

```
15. using LearningAlgorithms;
16. using NEATImplementation;
17. using DQNImplementaion;
18. using PPOImplementation;
19. using System.IO;
20.
21. public class GameManager : MonoBehaviour
22. {
23.
24.     //assigns
25.     [Header("references")]
26.     public GameObject runnerObject;
27.     public Transform runnerFoot;
28.     public GameObject taggerObject;
29.     public Transform taggerFoot;
30.     public Transform runnerSpawn;
31.     public Transform taggerSpawn;
32.     public LayerMask groundedLayers;
33.     [Header("managers")]
34.     [Space]
35.     public SimStatsManager simStatsManager;
36.     public UInterfaceManager uIController;
37.     [Header("ui references")]
38.     [Space]
39.
40.     //ui elements
41.     public TMP_Text roundText;
42.     public TMP_Text roundPausedText;
43.     public TMP_Text gameStatusText;
44.     [Header("player movement variables")]
45.     [Space]
46.
47.     //player movement variables
48.     public float forceMultiplier = 100;
49.     public float jumpMultiplier = 0.04f;
50.     public float rotationMultiplier = 10;
51.     private bool isGrounded;
52.     [Header("player learning algorithms")]
53.     [Space]
54.
55.     //player algorithms
56.     public LearningAlgorithm taggerAlgorithm;
57.     public LearningAlgorithm runnerAlgorithm;
58.
59.     //internal variables
60.
61.     //rounds
62.     private bool taggerWin = false;
63.     private bool runnerWin = false;
64.     private int currentRound = 0;
65.     private int currentSubRound = 0;
66.
67.     //simulation
68.     private float maxDistance = 0;
69.     private float distance = 0;
70.     [SerializeField] private int currentFrame = 0;
71.     [Header("sim configs + general info")]
72.     [Space]
73.     public bool gameRunning = false;
74.     public bool roundPaused = false;
75.     public bool pausingEnabled = false;
76.     public int pauseInterval = 0;
77.     public int totalRounds;
78.     public string filepath;
79.
80.     //runner/tagger
81.     private Transform runnerTransform;
82.     private Transform taggerTransform;
83.     private Collider runnerCollider;
84.     private Collider taggerCollider;
85.
86.     //ui
87.     private string lastRoundWinner = @"N\A";
88.     private int taggerOverallRoundsWon = 0;
```

```
89.     private int runnerOverallRoundsWon = 0;
90.     private int taggerSubRoundsWon = 0;
91.     private int runnerSubRoundsWon = 0;
92.     private int numSubRounds = 0;
93.
94.     //stats
95.     private float taggerRoundsWon = 0;
96.     private float runnerRoundsWon = 0;
97.     private float avgDistance = 0;
98.     private bool isLearningDone = true;
99.
100.    //stuff i wanna see
101.    [Header("reward functions")]
102.    [Space]
103.    [SerializeField] private float runnerReward;
104.    [SerializeField] private float taggerReward;
105.
106.    // Start is called once before the first execution of Update after the MonoBehaviour is created
107.    void Start()
108.    {
109.        //cap fps at my display fps to prevent unnecessary render
110.
111.        Application.targetFrameRate = 240;
112.        //get components
113.        runnerTransform = runnerObject.GetComponent<Transform>();
114.        taggerTransform = taggerObject.GetComponent<Transform>();
115.        runnerCollider = runnerObject.GetComponent<Collider>();
116.        taggerCollider = taggerObject.GetComponent<Collider>();
117.
118.        //initialise rewards
119.        runnerReward = 0;
120.        taggerReward = 0;
121.    }
122.
123.    //Objective 2a
124.    //Dictionary
125.    //Array
126.    public void InitaliseSimulation(string[] simulationParams, string[] taggerParams, string[]
runnerParams)
127.    {
128.        //assign agents and params as needed
129.        Hyperparamters.Hyperparameters taggerHyperparameters;
130.        Hyperparamters.Hyperparameters runnerHyperparameters;
131.
132.        //tagger hyperparameters
133.        switch (taggerParams[0])
134.        {
135.            case "NEAT":
136.                taggerHyperparameters = new NEATHyperparameters()
137.                {
138.                    deltaThreshold = float.Parse(taggerParams[1]),
139.                    maxFitnessHistory = int.Parse(taggerParams[2]),
140.                    populationSize = int.Parse(taggerParams[3]),
141.                    distanceWeights = new Dictionary<string, float>{
142.                        {"edge" , float.Parse(taggerParams[4])},
143.                        {"weight" , float.Parse(taggerParams[5])},
144.                        {"bias" , float.Parse(taggerParams[6])}
145.                    },
146.                    breedProbabilities = new Dictionary<string, float>{
147.                        {"asexual" , float.Parse(taggerParams[7])},
148.                        {"sexual" , float.Parse(taggerParams[8])}
149.                    },
150.                    mutationProbabilities = new Dictionary<string, float>{
151.                        {"node" , float.Parse(taggerParams[9])},
152.                        {"edge" , float.Parse(taggerParams[10])},
153.                        {"weight perturb" , float.Parse(taggerParams[11])},
154.                        {"weight set" , float.Parse(taggerParams[12])},
155.                        {"bias perturb", float.Parse(taggerParams[13])},
156.                        {"bias set", float.Parse(taggerParams[14])}
157.                    },
158.                };
159.                numSubRounds = int.Parse(taggerParams[3]);
160.                taggerAlgorithm = new NEATAlgorithm((NEATHyperparameters)taggerHyperparameters);
161.                break;
```

```
162.             case "DQN":
163.                 taggerHyperparameters = new DQNHyperparameters()
164.                 {
165.                     lr = float.Parse(taggerParams[1]),
166.                     explorationProbInit = float.Parse(taggerParams[2]),
167.                     explorationProbDecay = float.Parse(taggerParams[3]),
168.                     batchSize = int.Parse(taggerParams[4]),
169.                     memoryBufferSize = int.Parse(taggerParams[5]),
170.                     hiddenLayersSizes = new int[2] { int.Parse(taggerParams[6]),
int.Parse(taggerParams[7]) },
171.                     actDim = 5,
172.                     obsDim = 7
173.                 };
174. 
175.                 taggerAlgorithm = new DQNAlgorithm((DQNHyperparameters)taggerHyperparameters);
176.                 break;
177.             case "PPO":
178.                 taggerHyperparameters = new PPOHyperparameters()
179.                 {
180.                     maxEpsisodesInBatch = int.Parse(taggerParams[1]),
181.                     timestepsPerBatch = int.Parse(taggerParams[2]),
182.                     nUpdatesPerIteration = int.Parse(taggerParams[3]),
183.                     lr = float.Parse(taggerParams[4]),
184.                     gamma = float.Parse(taggerParams[5]),
185.                     lam = float.Parse(taggerParams[6]),
186.                     clip = float.Parse(taggerParams[7]),
187.                     maxGradNorm = float.Parse(taggerParams[8]),
188.                     actorNetworkSizes = new int[2] { int.Parse(taggerParams[9]),
int.Parse(taggerParams[10]) },
189.                     criticNetworkSizes = new int[2] { int.Parse(taggerParams[11]),
int.Parse(taggerParams[12]) },
190.                     actDim = 5,
191.                     obsDim = 7
192.                 };
193. 
194.                 taggerAlgorithm = new PPOAlgorithm((PPOHyperparameters)taggerHyperparameters);
195.                 break;
196.         }
197. 
198.         //runner hyperparameters
199.         switch (runnerParams[0])
200.         {
201.             case "NEAT":
202.                 runnerHyperparameters = new NEATHyperparameters()
203.                 {
204.                     deltaThreshold = float.Parse(runnerParams[1]),
205.                     maxFitnessHistory = int.Parse(runnerParams[2]),
206.                     populationSize = int.Parse(runnerParams[3]),
207.                     distanceWeights = new Dictionary<string, float>{
208.                         {"edge" , float.Parse(runnerParams[4])},
209.                         {"weight" , float.Parse(runnerParams[5])},
210.                         {"bias" , float.Parse(runnerParams[6])}
211.                     },
212.                     breedProbabilities = new Dictionary<string, float>{
213.                         {"asexual" , float.Parse(runnerParams[7])},
214.                         {"sexual" , float.Parse(runnerParams[8])}
215.                     },
216.                     mutationProbabilities = new Dictionary<string, float>{
217.                         {"node" , float.Parse(runnerParams[9])},
218.                         {"edge" , float.Parse(runnerParams[10])},
219.                         {"weight perturb" , float.Parse(runnerParams[11])},
220.                         {"weight set" , float.Parse(runnerParams[12])},
221.                         {"bias perturb", float.Parse(runnerParams[13])},
222.                         {"bias set", float.Parse(runnerParams[14])}
223.                     },
224.                 };
225.                 //update number of subrounds if greater population size
226.                 numSubRounds = (int.Parse(runnerParams[3]) > numSubRounds) ? int.Parse(runnerParams[3]) :
numSubRounds;
227.                 runnerAlgorithm = new NEATAlgorithm((NEATHyperparameters)runnerHyperparameters);
228.                 break;
229.             case "DQN":
230.                 runnerHyperparameters = new DQNHyperparameters()
231.                 {
```

```
                    lr = float.Parse(runnerParams[1]),
                    explorationProbInit = float.Parse(runnerParams[2]),
                    explorationProbDecay = float.Parse(runnerParams[3]),
                    batchSize = int.Parse(runnerParams[4]),
                    memoryBufferSize = int.Parse(runnerParams[5]),
                    hiddenLayersSizes = new int[2] { int.Parse(runnerParams[6]),
int.Parse(runnerParams[7]) },
                    actDim = 5,
                    obsDim = 7
                };

                runnerAlgorithm = new DQNAlgorithm((DQNHyperparameters)runnerHyperparameters);
                break;
            case "PPO":
                runnerHyperparameters = new PPOHyperparameters()
                {
                    maxEpsisodesInBatch = int.Parse(runnerParams[1]),
                    timestepsPerBatch = int.Parse(runnerParams[2]),
                    nUpdatesPerIteration = int.Parse(runnerParams[3]),
                    lr = float.Parse(runnerParams[4]),
                    gamma = float.Parse(runnerParams[5]),
                    lam = float.Parse(runnerParams[6]),
                    clip = float.Parse(runnerParams[7]),
                    maxGradNorm = float.Parse(runnerParams[8]),
                    actorNetworkSizes = new int[2] { int.Parse(runnerParams[9]),
int.Parse(runnerParams[10]) },
                    criticNetworkSizes = new int[2] { int.Parse(runnerParams[11]),
int.Parse(runnerParams[12]) },
                    actDim = 5,
                    obsDim = 7
                };

                runnerAlgorithm = new PPOAlgorithm((PPOHyperparameters)runnerHyperparameters);
                break;
        }

        //simulation parameters
        totalRounds = int.Parse(simulationParams[1]);
        pausingEnabled = simulationParams[4] == "true";
        pauseInterval = int.Parse(simulationParams[2]);
        filepath = Path.Combine(simulationParams[3], "stats.txt");
    }

    //Objective 7aiii
    void AgentsLearn()
    {
        taggerAlgorithm.Learn(currentRound);
        runnerAlgorithm.Learn(currentRound);
        isLearningDone = true;
    }

    //Objectives 7a, 7b, 7c, 7d
    public void UpdateRound()
    {
        //increment frame
        currentFrame++;

        //rewards
        taggerReward = CalculateTaggerReward();
        runnerReward = CalculateRunnerReward();

        //have agents take action
        var taggerSimInput = TakeTaggerInput();
        var runnerSimInput = TakeRunnerInput();
        var taggerOutput = taggerAlgorithm.GetOutput(taggerSimInput, false, taggerReward);
        var runnerOutput = runnerAlgorithm.GetOutput(runnerSimInput, false, taggerReward);
        OutputsToProgram(taggerOutput, taggerObject, taggerFoot);
        OutputsToProgram(runnerOutput, runnerObject, runnerFoot);

        //update average distance
        avgDistance += (distance - avgDistance) / (currentFrame + 1e-10f);

        //update stats panel if active
        if (simStatsManager.isActive)
```

```
303.         {
304.             simStatsManager.taggerSimInput = taggerSimInput;
305.             simStatsManager.runnerSimInput = runnerSimInput;
306.             simStatsManager.taggerSimOutput = taggerOutput;
307.             simStatsManager.runnerSimOutput = runnerOutput;
308.             simStatsManager.taggerReward = taggerReward;
309.             simStatsManager.runnerReward = runnerReward;
310.             simStatsManager.taggerRoundsWon = taggerOverallRoundsWon;
311.             simStatsManager.runnerRoundsWon = runnerOverallRoundsWon;
312.             simStatsManager.lastRoundWinnerName = lastRoundWinner;
313.             simStatsManager.UpdateStatsPanel();
314.         }
315.
316.         //check if round is over (if a given agent has won)
317.         if (currentFrame >= 30 * 30)
318.         { //time up, runner has got away //Objective 7bi
319.             Debug.Log("RUNNER WIN");
320.             taggerWin = false;
321.             lastRoundWinner = "Runner";
322.             runnerSubRoundsWon++;
323.             runnerOverallRoundsWon++;
324.             gameStatusText.text = "ROUND PAUSED FOR TRAINING";
325.             runnerWin = true;
326.         }
327.         else if (runnerCollider.bounds.Intersects(taggerCollider.bounds))
328.         { //collision, tagger has caught runner //Objective 7bii
329.             runnerWin = false;
330.             lastRoundWinner = "Tagger";
331.             taggerOverallRoundsWon++;
332.             taggerSubRoundsWon++;
333.             gameStatusText.text = "ROUND PAUSED FOR TRAINING";
334.             taggerWin = true;
335.         }
336.
337.         //round over operations
338.         if (taggerWin || runnerWin)
339.         {
340.             ResetRound();
341.             currentFrame = 0;
342.             if (currentSubRound == numSubRounds) //note that this defaults to 0 for no NEAT
343.             {
344.                 if (taggerWin) { taggerRoundsWon++; }
345.                 if (runnerWin) { runnerRoundsWon++; }
346.                 Debug.Log("stats");
347.                 StoreStats(taggerSubRoundsWon, runnerSubRoundsWon);
348.                 currentSubRound = 0;
349.                 taggerSubRoundsWon = 0;
350.                 runnerSubRoundsWon = 0;
351.                 currentRound++;
352.                 isLearningDone = false;
353.             }
354.             else
355.             {
356.                 currentSubRound++;
357.                 if (taggerWin) { taggerSubRoundsWon++; }
358.                 if (runnerWin) { runnerSubRoundsWon++; }
359.                 if (taggerAlgorithm is NEATAlgorithm taggerNEAT)
360.                 {
361.                     taggerNEAT.IncrementAgents();
362.                 }
363.                 if (runnerAlgorithm is NEATAlgorithm runnerNEAT)
364.                 {
365.                     runnerNEAT.IncrementAgents();
366.                 }
367.             }
368.
369.             Time.timeScale = 1;
370.         }
371.     }
372.
373.     void FixedUpdate()
374.     {
375.         distance = Vector3.Distance(taggerTransform.position, runnerTransform.position);
376.         maxDistance = (distance > maxDistance) ? distance : maxDistance;
```

```
377.         //skip all if the game isnt running
378.         if (!gameRunning)
379.         {
380.             return;
381.         }
382.         else if (!isLearningDone)
383.         {
384.             AgentsLearn();
385.             gameStatusText.text = "ROUND RUNNING";
386.         }
387.         else
388.         { //if game is running, run the round
389.             UpdateRound();
390.             //end sim if at end
391.             if (currentRound >= totalRounds)
392.             {
393.                 gameRunning = false;
394.             }
395.
396.         }
397.     }
398.
399.     //Objective 7ai
400.     float CalculateRunnerReward()
401.     {
402.         float reward = 0;
403.         reward += distance / maxDistance; //encourage staying away
404.         reward += currentFrame * 2; //encourage staying alive
405.         return reward;
406.     }
407.
408.     //Objective 7ai
409.     float CalculateTaggerReward()
410.     {
411.         float reward = 0;
412.         reward -= distance / maxDistance; //encourage closing distance
413.         reward -= currentFrame * 2; //encourage being quick
414.         return reward;
415.     }
416.
417.     //Objective 7ai
418.     float[] TakeRunnerInput()
419.     {
420.         float[] inputs = new float[7];
421.         Vector2 forward = new(1, 0);
422.
423.         Vector2 runnerForward = new(runnerTransform.forward.x, runnerTransform.forward.z);
424.         Vector2 taggerToRunner = new(taggerTransform.position.x - runnerTransform.position.x,
taggerTransform.position.z - runnerTransform.position.z);
425.
426.         /* xpos */
427.         inputs[0] = runnerTransform.position.x;
428.         /* ypos */
429.         inputs[1] = runnerTransform.position.y;
430.         /* zpos */
431.         inputs[2] = runnerTransform.position.z;
432.         /* orientation wrt forward */
433.         inputs[3] = Vector2.Angle(forward, runnerForward);
434.         /* distance to tagger */
435.         inputs[4] = Vector3.Distance(taggerTransform.position, runnerTransform.position);
436.         /* orientation wrt tagger */
437.         inputs[5] = Vector2.Angle(runnerForward, taggerToRunner);
438.
439.         //isGrounded
440.         if (Physics.CheckSphere(runnerFoot.position, 0.2f, groundedLayers))
441.         {
442.             inputs[6] = 1;
443.         }
444.         else
445.         {
446.             inputs[6] = 0;
447.         }
448.
449.         return inputs;
```

```
450.     }
451.
452.     //Objective 7ai
453.     float[] TakeTaggerInput()
454.     {
455.         float[] inputs = new float[7];
456.         Vector2 forward = new(1, 0);
457.         Vector2 taggerForward = new(taggerTransform.forward.x, taggerTransform.forward.z);
458.         Vector2 taggerToRunner = new(runnerTransform.position.x - taggerTransform.position.x,
runnerTransform.position.z - taggerTransform.position.z);
459.
460.         /* xpos */
461.         inputs[0] = taggerTransform.position.x;
462.         /* ypos */
463.         inputs[1] = taggerTransform.position.y;
464.         /* zpos */
465.         inputs[2] = taggerTransform.position.z;
466.         /* orientation wrt forward */
467.         inputs[3] = Vector2.Angle(forward, taggerForward);
468.         /* distance to tagger */
469.         inputs[4] = Vector3.Distance(taggerTransform.position, runnerTransform.position);
470.         /* orientation wrt tagger */
471.         inputs[5] = Vector2.Angle(taggerForward, taggerToRunner);
472.
473.         //isGrounded
474.         if (Physics.CheckSphere(taggerFoot.position, 0.2f, groundedLayers))
475.         {
476.             inputs[6] = 1;
477.         }
478.         else
479.         {
480.             inputs[6] = 0;
481.         }
482.
483.         return inputs;
484.     }
485.
486.     //Objective 7c
487.     void ResetRound()
488.     {
489.         //reset frame counter
490.         currentFrame = 0;
491.
492.         //reset positions
493.         runnerObject.transform.position = runnerSpawn.position;
494.         taggerObject.transform.position = taggerSpawn.position;
495.
496.         //reset round winners
497.         taggerWin = false;
498.         runnerWin = false;
499.
500.         //reset avg distance
501.         avgDistance = 0;
502.
503.         //set round info text
504.         if (pausingEnabled)
505.         {
506.             roundText.text = $"Current Round: {currentRound}" + @"\" + $"Next Pause: {currentRound +
(pauseInterval - (currentRound % pauseInterval)) % pauseInterval}";
507.             if (currentRound % pauseInterval == 0)
508.             {
509.                 roundPaused = true;
510.             }
511.         }
512.         else
513.         {
514.             roundText.text = $"Current Round: {currentRound}";
515.         }
516.
517.         //check if round is to be paused. if true, then pause round
518.         if (roundPaused)
519.         {
520.             gameRunning = false;
521.             roundPausedText.text = "Round Paused";
```

```
522.         }
523.     }
524.
525.     //Objective 7aii
526.     void OutputsToProgram(bool[] movement, GameObject movingObject, Transform footTransform)
527.     {
528.         //get components
529.         Rigidbody rb = movingObject.GetComponent<Rigidbody>();
530.         Transform tf = movingObject.transform;
531.
532.         //grounded check
533.         isGrounded = Physics.CheckSphere(footTransform.position, 0.2f, groundedLayers);
534.         if (movement[0])
535.         {
536.             rb.AddForce(tf.forward * forceMultiplier); //forward
537.         }
538.
539.         if (movement[1])
540.         {
541.             rb.AddForce(tf.forward * -forceMultiplier); //backward
542.         }
543.
544.         if (movement[2])
545.         {
546.             tf.Rotate(0, rotationMultiplier, 0); //left
547.         }
548.
549.         if (movement[3])
550.         {
551.             tf.Rotate(0, -rotationMultiplier, 0); //right
552.         }
553.
554.         if (movement[4] && isGrounded)
555.         {
556.             rb.AddForce(tf.up * jumpMultiplier); //jump
557.         }
558.
559.     }
560.
561.     //Objective 8ai
562.     //Write to File
563.     void StoreStats(int taggerRounds, int runnerRounds)
564.     {
565.         runnerRoundsWon += (float)runnerRounds / (taggerRounds + runnerRounds);
566.         taggerRoundsWon += (float)taggerRounds / (taggerRounds + runnerRounds);
567.         string statsString =
$"{currentRound},{runnerReward},{taggerReward},{avgDistance},{runnerRoundsWon},{taggerRoundsWon}";
568.         using StreamWriter sw = new(filepath, true);
569.         sw.WriteLine(statsString);
570.     }
571. }
572.
573.
```

Note that in this file, the code formatter I was using to paste code bugged out and didn't render any of the latter lines' colours. Objectives met and techniques used have been highlighted in red nonetheless.

PanelMaker.cs

```
 1. //Objective 1bi
 2. //Unity UI
 3.
 4. using UnityEngine;
 5. using UnityEngine.UI;
 6.
 7. public class PanelMaker : MonoBehaviour
 8. {
 9.     public string headerText;
10.     [TextArea(3, 10)] //makes text area bigger in inspector
11.     public string bodyText;   // The text to display when the button is clicked
12.
13.     public PanelManager panelManager;
14.     //add listener & panel manager
15.     void Start()
```

```
16.     {
17.         Button btn = GetComponent<Button>();
18.         btn.onClick.AddListener(CreatePanel);
19.     }
20.
21.     // create panel
22.     public void CreatePanel()
23.     {
24.         panelManager.CreatePanel(headerText, bodyText);
25.     }
26. }
27.
28.
```

PanelManager.cs

```
 1. //Objective 1bii
 2.
 3. using UnityEngine;
 4. using System.Collections.Generic;
 5.
 6. public class PanelManager : MonoBehaviour
 7. {
 8.     [SerializeField]
 9.     private GameObject panelPrefab;
10.     [SerializeField]
11.     private Canvas targetCanvas;
12.
13.     private readonly List<DraggablePanel> panels = new();
14.
15.     void Awake()
16.     {
17.         //find canvas
18.         targetCanvas = FindAnyObjectByType<Canvas>();
19.     }
20.
21.     //Objective 1bii
22.     public DraggablePanel CreatePanel(string headerText, string bodyText, Vector2 position = default){
23.         GameObject panelObj = Instantiate(panelPrefab, targetCanvas.transform);
24.
25.         //set position (defaults to center)
26.         RectTransform rectTransform = panelObj.GetComponent<RectTransform>();
27.         if (position == default){
28.             position = Vector2.zero;
29.         }
30.         rectTransform.anchoredPosition = position;
31.
32.         //setup panel content
33.         DraggablePanel panel = panelObj.GetComponent<DraggablePanel>();
34.         panel.SetBody(bodyText);
35.         panel.SetHeader(headerText);
36.
37.         //add to list
38.         panels.Add(panel);
39.
40.         return panel;
41.     }
42.
43.     public void DestroyAllPanels(){
44.         //if panel exists, destroy
45.         foreach (DraggablePanel panel in panels){
46.             if (panel != null){
47.                 panel.DestroyPanel();
48.             }
49.         }
50.
51.         //clear list
52.         panels.Clear();
53.     }
54. }
55.
56.
```

SimStatsManager.cs

```
1. //Objective 8b
2. //Unity UI
3. //String Formatting
4.
5. using UnityEngine;
6. using UnityEngine.UI;
7. using TMPro;
8.
9. public class SimStatsManager : MonoBehaviour
10. {
11.     //stats to be assigned from game manager
12.     public float[] taggerSimInput;
13.     public float[] runnerSimInput;
14.     public bool[] taggerSimOutput;
15.     public bool[] runnerSimOutput;
16.     public float runnerReward;
17.     public float taggerReward;
18.     public int taggerRoundsWon;
19.     public int runnerRoundsWon;
20.     public string lastRoundWinnerName;
21.
22.     //scene assigns
23.     public GameManager gameManager;
24.     public Button toggleButton;
25.     public TMP_Text taggerColumn;
26.     public TMP_Text runnerColumn;
27.     public TMP_Text taggerScore;
28.     public TMP_Text runnerScore;
29.     public TMP_Text lastRoundWinner;
30.
31.     //flag
32.     public bool isActive = false;
33.
34.     //internal array so the program knows what each output means
35.     string[] outputLabels = {"forward", "backward", "turn left", "turn right", "jump"};
36.
37.     // Start is called once before the first execution of Update after the MonoBehaviour is created
38.     void Start()
39.     {
40.         toggleButton.onClick.AddListener(ToggleStatsPanel);
41.         gameObject.SetActive(false);
42.         isActive = false;
43.     }
44.
45.     public void UpdateStatsPanel(){
46.         taggerColumn.text = FormatString(taggerSimInput, taggerSimOutput, taggerReward);
47.         runnerColumn.text = FormatString(runnerSimInput, runnerSimOutput, runnerReward);
48.         taggerScore.text = taggerRoundsWon.ToString();
49.         runnerScore.text = runnerRoundsWon.ToString();
50.         lastRoundWinner.text = lastRoundWinnerName;
51.     }
52.
53.     public void ToggleStatsPanel(){
54.         isActive = !isActive;
55.         gameObject.SetActive(isActive);
56.     }
57.
58.     private string FormatString(float[] inputs, bool[] outputs, float reward){
59.         string outputsString = FormatString(outputs);
60.
61.         return $"x position: {inputs[0]:F3}\n" +
62.                $"y position: {inputs[1]:F3}\n" +
63.                $"z position: {inputs[2]:F3}\n" +
64.                $"direction wrt forward: {inputs[3]:F3}\n" +
65.                $"distance: {inputs[4]:F3}\n" +
66.                $"angle wrt other {inputs[5]:F3}\n" +
67.                $"reward {reward:F3}\n" +
68.                $"actions taken:\n" +
69.                $"{outputsString}";
70.     }
71.
72.     //String Formatting
```

```
    private string FormatString(bool[] outputs){
        System.Text.StringBuilder sb = new();
        //if output is "true", add output to
        for (int i = 0; i < 5; i++){
            if (outputs[i]){
                if (sb.Length > 0) sb.Append(",\n");
                sb.Append(outputLabels[i]);
            }
        }
        return sb.ToString();
    }
}
```

FieldResetter.cs

```
//Objective 2c
//Unity UI

using UnityEngine;
using UnityEngine.UI;
using TMPro;

public class FieldResetter : MonoBehaviour
{
    public Button resetButton;
    public TMP_Dropdown algoDropdown;
    public TMP_InputField[] DQNFields;
    public float[] ResetValues;

    // Start is called once before the first execution of Update after the MonoBehaviour is created
    void Start()
    {
        resetButton.onClick.AddListener(ResetFields);
    }

    public void ResetFields()
    {
        algoDropdown.value = 0; //reset dropdown to DQN

        //reset all DQN values
        for (int i = 0; i < DQNFields.Length; i++)
        {
            DQNFields[i].text = ResetValues[i].ToString();
        }
    }
}
```

FieldRangeValidator.cs

```
//Objective 2e
//Regualar Expressions

using UnityEngine;
using TMPro;
using System.Text.RegularExpressions;

public class FieldRangeValidator : MonoBehaviour
{
    //diffentiatte between different types of validators
    public int validatorType = 0;
    //for type 0
    public float minValue = 0f;
    public float maxValue = 100f;

    private TMP_InputField inputField;
    public TMP_Text feedbackText;
    public bool isValid;

    void Awake()
    {
```

```
22.         //get component of input field
23.         inputField = GetComponent<TMP_InputField>();
24.
25.         //add end edit listener
26.         inputField.onEndEdit.AddListener(ValidateInput);
27.     }
28.
29.     void ValidateInput(string input)
30.     {
31.         if (validatorType == 0) {
32.             //take input
33.             float val = float.Parse(input);
34.
35.             //check & output suitable message
36.             if (val < minValue) {
37.                 UpdateText("value too small.", Color.red);
38.                 isValid = false;
39.             } else if (val > maxValue) {
40.                 UpdateText("value too large", Color.red);
41.                 isValid = false;
42.             } else if (val >= minValue && val <= maxValue) {
43.                 UpdateText("", Color.white);
44.                 Debug.Log("aura gained");
45.                 isValid = true;
46.             }
47.         } else if (validatorType == 1) {
48.             isValid = Regex.IsMatch(input, @"^[a-zA-Z]+$"); // a file path validator //Regular Expressions
49.             if (!isValid){
50.                 UpdateText("Should be a file path.", Color.red);
51.             }
52.         }
53.     }
54.
55.     void UpdateText(string feedback, Color colour)
56.     {
57.         if (feedbackText != null)
58.         {
59.             feedbackText.text = feedback;
60.             feedbackText.color = colour;
61.         }
62.     }
63. }
64.
```

DropdownHandler.cs

```
1. //Objective 2ai
2. //Unity UI
3.
4. using UnityEngine;
5. using TMPro;
6.
7. public class DropdownHandler : MonoBehaviour
8. {
9.     public TMP_Dropdown dropdown; //reference to dropdown for user
10.     public GameObject[] fieldGroups; //array of field groups for different algorithms to store/hide
11.     public TakeFieldsInput chosenValidator;
12.
13.     void Start()
14.     {
15.         //add listener for dropdown
16.         dropdown.onValueChanged.AddListener(OnDropdownValueChanged);
17.         //initalise field visibility based on default selection
18.         UpdateFieldVisibility(dropdown.value);
19.
20.     }
21.
22.     void OnDropdownValueChanged(int index)
23.     {
24.         UpdateFieldVisibility(index);
25.     }
26.
27.     void UpdateFieldVisibility(int index)
28.     {
```

```
29.         for (int i = 0; i < fieldGroups.Length; i++)
30.         {
31.             fieldGroups[i].SetActive(i== index);
32.             chosenValidator = fieldGroups[index].GetComponent<TakeFieldsInput>();
33.         }
34.         switch (index) {
35.             case 0:
36.                 chosenValidator.inputPurpose = "DQN";
37.                 break;
38.             case 1:
39.                 chosenValidator.inputPurpose = "PPO";
40.                 break;
41.             case 2:
42.                 chosenValidator.inputPurpose = "NEAT";
43.                 break;
44.         }
45.     }
46. }
47.
48.
```

Testing

Test Overview

This is a representative sample of tests that would be used to test the whole program. Due to its modular nature, the testing of the program can be conducted by taking parts of the code out of the overall program, and using them in separate C# projects to test them, using test data. The tests are to be conducted in a video, and timestamps for each test are provided in a separate column.

This is the key for the Pass/Fail column of the Test Plan.

Pass	Fail

Test Plan

Test Number	Objectives Tested	Test Description	Test Details	Expected Results	Time-stamp	Pass /Fail
1	Objective 1a: Screen Navigation	Ensure the program allows smooth transitions between different screens	Move between configuration screens, simulation screen, results screen, and back	Screens should switch correctly without errors or delays, windows should be retained between screens.	0:00	
2	Objective 1b: UI Windows Functionality	Ensure that pop-up windows, settings panels, and information panels function correctly	Open various in-game windows, move them, close them, and reopen them	Windows should appear when requested, move smoothly, close properly, and have correct information /headings.	0:39	
3	Objective 2a: Algorithm selection	Verify that selecting different algorithms updates agent behaviour	Select different algorithms for both agents.	Inspector view of UIManager should update to show respective algorithms	1:00	
4	Objective 2e: Input validation	Test invalid input handling for hyperparameters	Enter negative/invalid values for learning rate, exploration probability, etc.	System should show an appropriate error message when user has errooneous input	1:38	
5	Objective 3c: DQN Action Selection (Exploration vs Exploitation)	Ensure DQN balances exploration and exploitation correctly	Run DQN agent with exploration probabilities of 0, 1, and 0.5.	For exploration probability = 0, always neural network. For =1, always random. For =0.5, either.	2:25	
6	Objective 3c: DQN Compute Action Correctness	Verify DQN selects the highest Q-value action during exploitation	Provide states, show Q-values and index of selected actions	Action should match the one with the highest Q-value	3:52	

7	Objective 3d: DQN Epsilon-Greedy Correctness	Ensure Epsilon-Greedy algorithm is applied correctly	Run Epsilon-Greedy implementation and record value of exploration probability before and after.	Updated value should match calculations done by hand.	4:55	
8	Objective 4di: PPO Generalized Advantage Estimation (GAE)	Ensure advantage calculations match theoretical values	Compute expected advantages manually and compare to PPO's computed values	PPO's computed advantages should match manual calculations	5:47	
9	Objective 5a: NEAT Population Initialization	Verify that initial genomes are diverse and correctly structured	Run NEAT and inspect the first population of genomes	Initial genomes should have random weights and varying structures	7:43	
10	Objective 5d: NEAT Mutation Process Accuracy	Ensure mutations correctly modify genomes	Track genome structure before and after mutation step	Mutations should introduce new nodes, connections, or weight adjustments	8:37	
11	Objective 5d: NEAT Crossover Functionality	Verify that offspring inherit correct traits from parents	Cross two known parent genomes with distinct structures, inspect child genome	Child genome should inherit a mix of parent structures and weights	9:27	
12	Objective 6a: Neural Network Forward Pass Correctness	Ensure NN outputs match expected activations	Feed predefined inputs to a layer and compare outputs with expected activations	Outputs should match manually calculated values for the layer	10:50	
13	Objective 6b: Neural Network Backpropagation Accuracy	Ensure gradient calculations and updates are correct	Train NN on the wine dataset [26], compare accuracy to original implementation.	Accuracy should be within 5% after 100 iterations	12:13	
14	Objective 6c: Neural Network Activation Functions	Ensure activation functions produce correct outputs	Input test values into ReLU, Softmax, and Sigmoid, compare to expected values	Outputs should match theoretical activation function outputs	13:58	

15	Objective 7b: Simulation Round Termination	Ensure rounds end correctly based on game rules	Run simulation with algorithms disabled and tester input.	Rounds should end when timeout is reached or chaser catches runner	16:41	
16	Objective 7d: Simulation Pausing at Configured Intervals	Verify that auto-pausing works correctly	Set auto-pause every 2 iterations, check if simulation stops at correct points	Simulation should pause precisely at configured intervals	17:40	
17	Objective 8a: Data Collection Accuracy	Ensure that collected data is accurate and formatted correctly	Run rounds using manual input, and record values in program. Check stats folder after too.	Collected data should match manually recorded game statistics – chosen statistic is rounds won for testing. During program, values should change according to tester input	18:40	
18	Objective 8c: Graph Generation Accuracy	Verify that graphs correctly represent collected data	Compare graph outputs to expected results	Graphs should match expected trends in collected data	20:05	

Evaluation

Evaluation Overview

This evaluation section looks at the project as a whole, and assesses whether each Objective from the Analysis section has been met. These objectives have been copied to this section for convenience, and comments have been made on each and their respective implementations in the Technical Solution. Then, feedback from the interviewee contacted in the Analysis section is taken and also commented upon, and a set of potential improvements is also created to show how to program could progress with further development.

Objective Evaluation

Objectives

The program has the following objectives:

1. The program will allow the user to view the two players of the game, henceforth referred to as "agents" and controlled by AI.
    - 1a. The program switches between screens as required by the user input
        - 1ai. Backwards
        - 1aii. Forwards
    - 1b. The program will display information about each part of the program.
        - 1bi. Information icon available.
        - 1bii. Button spawns information panel
        - 1biii. User can move information panel around & close at will

2. The program will allow the user to change the settings of the simulation.
    - 2a. The program will allow the user to change parameters on the selected agent.
        - 2ai. Algorithm selection changes algorithm on agent.
        - 2aii. Parameter selection changes parameters on agent.
    - 2b. The program will allow the user to configure the game.
        - 2bi. Total number of rounds to be played.
        - 2bii. Automatic round pauses after a given number of rounds.
        - 2biii. Round interval
        - 2biv. Directory of stats output.
    - 2c. Reset all fields when user requires
    - 2d. The program will only start the program once the settings are all valid
    - 2e. The program should tell the user when an input is invalid

3. The program will allow a given agent to use the DQN algorithm
    - 3a. Receive input from program.
    - 3b. Store an "episode"
        - 3bi. Previous state
        - 3bii. Action taken
        - 3biii. Next state
        - 3biv. Reward received
    - 3c. Take action
        - 3ci. Exploration probability
        - 3cii. Neural network action
        - 3ciii. Random action
    - 3d. Update agent between iterations

3di. Select random batch of episodes
3dii. Train neural network using episodes
3diii. Update exploration probability

4. The program will allow the agent to use the PPO algorithm
    4a. Receive input from the program.
    4b. Store an "episode"
        4bi. Previous state
        4bii. Action taken
        4biii. Next state
        4biv. Reward received
    4c. Take action
        4ci. Compute action using actor network
        4cii. Sample action from multivariate normal distribution
        4ciii. Compute log probability of action
    4d. Update agent between iterations
        4di. Calculate advantage estimates using baseline value (Generalised Advantage Estimate)
        4dii. Optimise actor network using clipped objective function
        4diii. Update critic network to minimise the value loss

5. The program will allow the agent to use the NEAT algorithm.
    5a. Initialise population of neural network genomes
    5b. For each genome, take inputs and give a required output.
        5bi. Store fitness
        5bii. Take action
            1. Action from neural network
    5c. Organise Genomes into Species
    5d. Update the agent between iterations
        5di. Apply genetic operations to create new offspring
            1. Mutations
                5a. Add new node
                5b. Add new connection
                5c. Change edge weight
                5d. Change edge bias
            2. Crossover
                5a. Select a subset of neural networks to become the next generation
                5b. Combine individual properties of multiple networks into one for the next generation

6. The program will perform neural network functions where the algorithm needs them.
    6a. Take an input and return an output
        6ai. Network Level:
            1. Forward input through each layer
        6aii. Layer Level:
            1. Output calculation
            2. Activation function
    6b. Train the neural network using a given data set of inputs, expected outputs, and real outputs
        6bi. Network Level:
            1. Calculate output
            2. Calculate output gradient
            3. Update learning rate

4. Backpropagate loss through each layer

6bii. Layer Level:

1. Apply derivative of activation function
2. Calculate and clip derivatives of weights and biases
3. Calculate the derivative of the inputs
4. Update weights and biases using learning rate and derivatives
5. Update weights and biases using Adam
6. Return derivative of inputs for next layer to use

6c. Create a set of activation functions for the program to use, one set for neural networks and one set for the NEAT implementation

6ci. Neural Networks

1. ReLU
2. LReLU
3. Softmax

6cii. NEAT

1. ReLU
2. LReLU
3. Softmax
4. Sigmoid
5. Tanh

7. The program will run the rounds of the game without user input, as per their configuration.

7a. The program will manage the agents.

7ai. Inputs are passed along to the agents.

7aii. Outputs from the agents are replicated in the game.

7aiii. Agents learn at the end of each round

7b. The program will manage a round of the game.

7bi. Round ends when chaser touches runner.

7bii. Round ends when timer ends.

7c. The program will run another round.

7ci. Another round starts after one ends.

7d. The program will stop the rounds upon configured intervals.

7di. After the round number reaches one that satisfies the condition given by the user, the program pauses the next round.

7dii. Display a prompt to allow the user to continue the rounds.

7diii. Allow the user to continue the rounds.

7e. The program will manage the NEAT agents each getting a turn to play a round.

8. The program will collect data about the game, draw graphs & export to .csv file.

8a. The program will collect data about the game as per each round.

8ai. When a round ends, information about the round is recorded in a .csv file.

8b. The information collected about the last round will be displayed on screen.

8c. The program will visualise the data at the end of the game.

8ci. The program will draw graphs of a given statistic and time.

8cii. The user will be able to switch between each graph

8d. The program will output the statistics in .csv format to a given directory.

Comments

Objective 1 is mainly focused on the UI elements of the program. These are implemented using the Unity UI elements, as well as the StatsDrawer element taken from the YouTube tutorial. The program uses a more modular structure, allowing the user to switch between different screens before they start the simulation. As well as this, the program also

has a Panel system that allows the user to spawn different panels with different information about the program. This implementation fully meets the objective, as users can navigate between screens smoothly.

Objective 2 is focused upon allowing the user to change settings of the simulation, and of either agent. This is handled by input managers on each input field, with an associated output text for feedback if the program's data is entered incorrectly. Groups of these fields are used to represent each algorithm for each agent, and the program selects all the data from a given group when the simulation starts. The implementation successfully allows full customization.

Objective 3 aims to implement DQN. This is implemented in the DQN.cs file and the implementation of the DQNAlgorithm class in LearningAlgorithms.cs file. As well as implementing the learn step, DQNAlgorithm also calls the step to update the exploration probability using Bellman-Optimality. This implementation meets the objective.

Objectuve 4 aims to implement PPO. This is done in the PPO.cs file, and the implementation of the PPOAlgorithm class in the LearningAlgorithms.cs file. This implementation meets the objective.

Objective 5 aims to implement the NEAT algorithm. This is done using a collection of classes in the NEAT.cs file, and the LearningAlgorithms.cs file. A particular difference of the NEAT algorithm is that agents have to be changed each sub round, so a new method is introduced to account for that. This implementation meets the objective, and other round logic for this is handled in Objective 7.

Objective 6 refers to a full implementation of a neural network, organised into layers and an overall network. This is implemented fully in the FCNN.cs file, which also contains derivative classes for other neural network structures to be used in other applications (for example, a DQNModel to be used in the DQN algorithm). This implementation meets requirements.

Objective 7 is focused on the round structure of the program. In particular, this refers to managing the inputs and outputs of the agents, and resetting rounds after they end. Similarly, this also stipulates that rounds should be paused either according to given interval or user input. This is all implemented in the GameManager.cs file.

Objective 8 refers to data collection within the program. It means that the user is able to access a file with all of the statistics of the program's duration. This is done by having the user enter a file path, which then contains a text file with all of the program's statistics. Then, during the program, telemetry data for a given round as well as round information for the program's run as a whole should be available, and after the program has finished, the data should be graphed in the program for the user to see. This is implemented in the GameManager.cs, SimStatsManager.cs, and StatsDrawer.cs files. The features have been fully implemented.

Feedback from Interviewee

My interviewee was Tom Merrifield, a Shell employee whose work mainly focuses upon using AI to predict things about a given landscape. While not inherently from an AI field, Tom spends much of his time using it and has as such become proficient enough to teach his peers, delivering a talk while I was at Shell upon the topic.

Feedback

"Zain's program implements each agent in a mathematically sound manner, and he has tested parts of the program to prove this. The user-interface is intuitive, and of particular note were the information windows. Each containing relevant information about their associated headings, with clear definitions and headings allowing users to not only understand the function of a singular input for one algorithm but also compare it with the same input for a different algorithm. Having the program graph the statistics by default is useful to see overall trends, but the fact that the statistics themselves are stored in a file that I can access is the main highlight. It allows me not only to save sets of data for different simulations, but also to use other software to create more curated graphs for reports and presentations, while any of my team actually using the program can still make sense of the data without having to focus on tedious work making graphs. If I had to suggest a potential extra feature for the program, it would be to add more algorithms. Wheras the current set is relevant and well implemented, there are a few other algorithms that could be of use in my specific case, one of which being TRPO. However, in its current scope, the program achieves everything it sets out to do."

Comments

- Agents are implemented correctly and act according to the user's previous experience
- UI clarity allowing users to access information easily was a great focus initially, and as such the user has seen this in their experience using the program
- The allowance of flexibility to the user on the statistics side is useful, this could be expanded to provide more statistics perhaps.
- The program could implement more algorithms, in particular TRPO.

Reflection

Requirements Met

The program meets its set requirements from the Analysis section, by implementing each of its objectives. The program is functionally correct, with each algorithm along with their own indvidual methods being tested as working correctly. The program also displays information in a concise yet aesthetic manner, allowing users to switch screens smothly and configure settings within logical ranges, with clear validation feedback. The addition of information panels allows users to both teach themselves and use the program to teach others in a group setting, as was also set out in the Analysis. The project passed each of its test cases, covering a variety of of different inputs and test types.

Potential Improvements

The first of these improvements would be implementing GPU acceleration. Due to the system performing many mathematical calculations in a short amount of time, using the GPU to perform these calculations would make the program more convenient to use as the user would not have to wait as long in between each round for the agents to be trained. This allows for users to cover more rounds in the same amount of time, which would cause the agents to run for longer, show more of their potential nuances, and overall become more proficient at the game of tag.

Another improvement that could be made is enhancing data visualisation. Currently, although there are a large number of different neural networks used in the program, there is no manner to easily view their structure. For some of the algorithms, such as DQN or PPO, this is of little structure, as the focus is on the usage of the neural network rather than its actual structure. However, for algorithms such as NEAT, where neural network structures are consistently changing due to the evolutionary algorithm, this may be a vital piece of information for users to see how the algorithms' internal structures change as time progresses. Similarly, decision heatmaps could be made for the course of the program. This would allow users to see the relations between certain variables and the actions chosen to implement by the agent, further enabling the effectiveness of the teaching provided.

Furthermore, a simple improvement that can be implemented is the inclusion of more algorithms. This would allow users to witness the growing gaps in the training techniques of different algorithms, facilitating wider learning. Some algorithms in particular that could be implemented could be A2C/A3C, TROP or MA-POCA. A2C/A3C are two versions of the same algorithm, where A2C only accounts for one agent whereas A3C accounts for the possibility of deploying multiple agents to the environment. An agent interacts with the environment, collecting experiences and updating its policy and value functions depending on the inputs. This algorithm does the same for A3C, and is similar to PPO in the manner that two networks are employed – one to take actions and the other to evaluate those actions. TRPO is another algorithm developed by OpenAI, similar to PPO. It utilises a "trust region", restricting any optimisation step to a certain region around the algorithm's current point in the paramter space. In essence, the agent can only update itself to a certain extent every iteration. TRPO also employs a single neural network, as well as a defined value function. MA-POCA was used in the original "tag" game showcased in the Analysis section, and it introduces a framework that allows multiple agents to cooperatively work together towards a common goal. It emphasises the importance of communication and coordination around multiple agents, leading to more efficient learning and improved performance in complex environments. This algorithm would particularly help, if, for example, a team mode was implemented into the game, and further if the game was made more complex. These vastly different methods of learning how to tackle an environment mean that there is even more for a given person to learn to understand them along with the algorithms already implemented. Therefore, by allowing the user to have more choice, they are less likely to have to do extra research or learn without such a visual aid.

Finally, the game itself could be built upon to be more complex. One example of this could be placing cubes, walls, and other environment objects, and allowing the agent to pick up and move them around. Or, as mentioned with MA-POCA, the game could be altered to have multiple agents playing a single game. These increase the complexity of the environment, allowing users to see how agents react to more input data, or significantly less predictable surroundings. Similarly, the game's environment could change throughout its runtime. The size of the arena could change, or walls could suddenly be added and taken away. This would also require altering the program to give information of these new features of the environment, perhaps implementing computer vision to reduce the amount of other inputs needed to allow the user to understand its environment. This would mean that users see how agents can react to sudden changes in their environment, which may be practical for a user hoping to implement one of these algorithms to train a model of their own in such a fluctuating environment.

Biblography

[0] FinRL Paper: Liu, X., Yang, Z., Gao, X., Wang, X., & Zhang, J. (2020). FinRL: A Deep Reinforcement Learning Library for Automated Stock Trading in Quantitative Finance. Retrieved from https://arxiv.org/abs/2011.09607

[1] Surgery Paper: Mou, J., Zhang, Z., Qin, H., & Luo, X. (2020). Reinforcement Learning in Surgery: Current Applications and Future Directions. *Surface Science*, *699*, 121653. Retrieved from https://www.sciencedirect.com/science/article/abs/pii/S0039606020308254

[2] Autonomous Driving: Li, Y., Wang, Y., Cheng, B., & Wang, X. (2020). A Reinforcement Learning Framework for Autonomous Driving Policy Learning. *IEEE Transactions on Intelligent Transportation Systems*, *22*(4), 2353-2364. Retrieved from https://ieeexplore.ieee.org/abstract/document/9351818

[3] Tag Video: YouTube. (n.d.). "Reinforcement Learning Playing Tag." Retrieved from https://www.youtube.com/watch?v=hCmrMOzx5VA&t=15s

[4] FinRL GitHub Repository: AI4Finance-Foundation. (n.d.). FinRL. Retrieved from https://github.com/AI4Finance-Foundation/FinRL

[5] ChatGPT: OpenAI. (n.d.). "ChatGPT: Instruction Following." Retrieved from https://openai.com/index/instruction-following/

[6] Chess Neural Network: Chess Programming Wiki. (n.d.). "Stockfish Neural Network UE Layers." Retrieved from https://www.chessprogramming.org/File:StockfishNNUELayers.png

[7] Neural Network Tutorial: Shafiei, Y. (2020). "Making a Neural Network Fully Connected Layer from Scratch using Only NumPy." *Medium*. Retrieved from https://medium.com/@YasinShafiei/making-a-neural-network-fully-connected-layer-from-scratch-only-numpy-49bd7958b6f3

[8] Adam Optimizer: Kingma, D., & Ba, J. (2014). "Adam: A Method for Stochastic Optimization." Retrieved from https://arxiv.org/abs/1412.6980

[9] DQN Image: Jayasinghe, D. (n.d.). "DQN Explained." Retrieved from https://dilithjay.com/blog/dqn

[10] PPO Image: OpenDataScience. (n.d.). "Reinforcement Learning with PPO." Retrieved from https://opendatascience.com/reinforcement-learning-with-ppo

[11] NEAT Paper: Stanley, K. O., & Miikkulainen, R. (2002). "Evolving Neural Networks through Augmenting Topologies." *IEEE Transactions on Evolutionary Computation*, *10*(2), 99-127. Retrieved from https://nn.cs.utexas.edu/downloads/papers/stanley.cec02.pdf

[12] DQN Python Implementation: Towards Data Science. (n.d.). "Deep Q-Networks: Theory and Implementation." Retrieved from https://towardsdatascience.com/deep-q-networks-theory-and-implementation-37543f60dd67/

[13] PPO Medium Series Part 1: Eyyu, (n.d.). "Coding PPO from Scratch with PyTorch - Part 1/4." *Medium*. Retrieved from https://medium.com/analytics-vidhya/coding-ppo-from-scratch-with-pytorch-part-1-4-613dfc1b14c8

[14] PPO Medium Series Part 2: Eyyu, (n.d.). "Coding PPO from Scratch with PyTorch - Part 2/4." *Medium*. Retrieved from https://medium.com/@eyyu/coding-ppo-from-scratch-with-pytorch-part-2-4-f9d8b8aa938a

[15] PPO Medium Series Part 3: Eyyu, (n.d.). "Coding PPO from Scratch with PyTorch - Part 3/4." *Medium*. Retrieved from https://medium.com/@eyyu/coding-ppo-from-scratch-with-pytorch-part-3-4-82081ea58146

[16] PPO Medium Series Part 4: Xia, Z. (n.d.). "Coding PPO from Scratch with PyTorch - Part 4/4." *Medium*. Retrieved from https://medium.com/@z4xia/4e21f4a63e5c

[17] PPO Python GitHub: Yang, E. (n.d.). "PPO for Beginners." Retrieved from https://github.com/ericyangyu/PPO-for-Beginners/blob/master/part4/ppo_for_beginners/ppo_optimized.py

[18] PPO C# GitHub: Asieradzk. (n.d.). "Discrete PPO Agent in C#." Retrieved from https://github.com/asieradzk/RL_Matrix/blob/master/src/RLMatrix/Agents/PPO/Implementations/DiscretePPOAgent.cs

[19] NEAT Python GitHub: SirBob01. (n.d.). "NEAT-Python." Retrieved from https://github.com/SirBob01/NEAT-Python/tree/master

[20] NumSharp GitHub: SciSharp. (n.d.). "NumSharp." Retrieved from https://github.com/SciSharp/NumSharp

[21] TorchSharp GitHub: .NET Foundation. (n.d.). "TorchSharp." Retrieved from https://github.com/dotnet/TorchSharp

[22] Accord Framework Website: Accord Framework. (n.d.). "Accord.NET Machine Learning Framework." Retrieved from http://accord-framework.net/

[23] Accord GitHub: Accord Framework. (n.d.). "Accord.NET GitHub Repository." Retrieved from https://github.com/accord-net/framework

[24] Statistics YouTube Video: YouTube. (n.d.). "Statistics and Data Science Explained." Retrieved from https://www.youtube.com/watch?v=kGXV4MOVC0Y&

[25] Statistics GitHub Repository: PingSharpCode. (n.d.). "Graph-Based Statistical Analysis." Retrieved from https://github.com/PingSharpCode/Graph

[26] Wine Dataset: UCI Machine Learning Repository. (n.d.). "Wine Dataset." Retrieved from https://archive.ics.uci.edu/dataset/109/wine