Evaluation of deep reinforcement learning and its application through a case study in computer games

by

John Gregg-Webster  
202106626

In partial fulfilment of the   
requirements for the degree of   
Bachelor of Science

in Computer Science

CS408: Individual Project

Marking Scheme: Experimentation-based with Significant Software Development.

“Except where explicitly stated all the work in this report, including appendices, is my own and was carried out during my final year. It has not been submitted for assessment in any other context.”



Department of

Computer and Information Sciences

March, 2023

Contents

[1 Introduction 3](#_Toc162302941)

[2 Background Literature 4](#_Toc162302942)

[2.1 Decision Problems 4](#_Toc162302943)

[2.2 Value Functions and Decision-Making Approaches 5](#_Toc162302944)

[2.3 Sampling Methods 6](#_Toc162302945)

[2.4 Details of Targeted Algorithms 7](#_Toc162302946)

[2.4.1 Q-Value Methods 7](#_Toc162302947)

[2.4.2 Pure Policy-Based Methods 8](#_Toc162302948)

[2.4.3 Actor-Critic Methods 9](#_Toc162302949)

[3 Specification and Design 10](#_Toc162302950)

[3.1 Methodology 10](#_Toc162302951)

[3.2 Analysis 12](#_Toc162302952)

[3.3 Requirements 14](#_Toc162302953)

[3.3.1 Functional Requirements 14](#_Toc162302954)

[3.3.2 Non-Functional Requirements 15](#_Toc162302955)

[3.3.3 Design 15](#_Toc162302956)

[3.3.4 Interface Design 15](#_Toc162302957)

[3.4 System Design 16](#_Toc162302958)

[3.4.1 Maze 16](#_Toc162302959)

[3.4.2 Tag 16](#_Toc162302960)

[3.4.3 Tic-Tac-Toe 17](#_Toc162302961)

[3.4.4 Reinforcement Learning Agents 17](#_Toc162302962)

[3.4.5 Training System 18](#_Toc162302963)

[4 Product 18](#_Toc162302964)

[4.1 Implementation 18](#_Toc162302965)

[4.1.1 Reinforcement Learning Agents 18](#_Toc162302966)

[4.1.2 Environments 19](#_Toc162302967)

[4.1.3 Training & Evaluation System 20](#_Toc162302968)

[4.2 Verification & Validation 21](#_Toc162302969)

[4.2.1 Unit Testing 21](#_Toc162302970)

[4.2.2 Manual Testing Procedures 22](#_Toc162302971)

[5 Results & Evaluation 22](#_Toc162302972)

[5.1.1 Evaluation Process 22](#_Toc162302973)

[5.2 Results of Evaluation 23](#_Toc162302974)

[5.2.1 Environments 23](#_Toc162302975)

[5.2.2 Agents, Training & Evaluation System 24](#_Toc162302976)

[5.3 Returning to the Research Questions 27](#_Toc162302977)

[5.3.1 Effectiveness of Algorithms. 27](#_Toc162302978)

[5.3.2 Strengths & Weaknesses. 27](#_Toc162302979)

[6 Discussion & Reflection 27](#_Toc162302980)

[6.1 Interpreting the Results 27](#_Toc162302981)

[6.2 Reflection 28](#_Toc162302982)

[6.3 Challenges 28](#_Toc162302983)

[6.4 Future Work 29](#_Toc162302984)

[6.5 Limitations 29](#_Toc162302985)

[7 Conclusion 30](#_Toc162302986)

[A Detailed Specification And Design 33](#_Toc162302987)

**Abstract**

Deep reinforcement learning is a branch of machine learning, it is one method for producing decision-making neural networks. Where supervised machine learning methods train neural networks to copy the decisions of humans or traditional algorithms, reinforcement learning models learn their own decision-making strategies by interacting directly with their environment.

During the project, a number of reinforcement learning methods were investigated. Each method was applied to create AI agents, which were tasked with playing a series of computer games. The agents’ scores were documented as they learned to play the games. Finally, those scores were compared to determine the effectiveness, strengths, and weaknesses of each method.

The experimental findings were consistent with the existing literature. The agents learned to play each of the games at a basic level, though the learning process was slow and inefficient.

Acknowledgements

I would like to thank my supervisor, Feng Dong, as well as my second marker Keiren Egan, for their useful feedback on the project.

# Introduction

The aim of this project is to measure the effectiveness of several different deep reinforcement learning methods relative to human baselines, and to compare their relative strength and weaknesses. The targeted algorithms are; Deep Q-Learning, REINFORCE, Actor-Critic, Advantage Actor-Critic, and Proximal Policy Optimization.

Reinforcement learning is a method for training systems to perform an input-output mapping. These systems are usually neural networks. This project focuses specifically on *deep* reinforcement learning; reinforcement learning where the model is a neural network with hidden layers. But it is important to keep in mind that there are other approaches. Some reinforcement learning methods use single layer networks, or do not use networks at all i.e. tabular methods [1].

Reinforcement learning is especially interesting because it can be used to create solutions that are not based on prior human work. The goal is defined by the operator, but the model is capable of adopting any strategy in pursuit of it. The Ur-example is AlphaGo’s development of novel Go moves, but there are also cases that have produced real economic value. In 2022, reinforcement learning was used to develop more efficient matrix multiplication algorithms [2].

First, research was performed on the current state of the reinforcement learning field. The initial goal was to move from the broad question posed in the aim to a set of clearly defined criteria, measurable by experiment. During the course of this research, it became clear that the targets of comparison should be algorithms for training neural networks, and that there are many such algorithms, each with slight differences. From there, the focus of the research shifted to understanding the generally-accepted classification systems for algorithms, and selecting representative examples for each. In order to gather empirical evidence of the performance and limitations of each method, it was necessary to implement and test them. This produced another goal, to understand the details of each algorithm, in order to implement them.

The next objective was to produce a software implementation of each method, and virtual environments for the methods to be applied to. The three environments are Tag, Tic-Tac-Toe, Maze. They are described in more detail in section 3.3.

Once all the methods and environments were implemented, the methods were applied to the environments, and the results were gathered and analysed.

# Background Literature

## Decision Problems

Suppose we have designed an advanced new robot, and want to make it walk. It has the latest sensors, and expensive motors. There’s one problem, the engineers have no idea how to program it. They might try to apply supervised learning, copying an existing solution such as the movements of a human. However, this sort of data is not always available. Perhaps the robot has an unfamiliar body plan, or is walking on a planet with higher gravity. Reinforcement learning gives us the tools to solve this problem.

Reinforcement learning problems are typically framed as Markov decision problems [3, 4, 5]. An agent is given some information, and must make a decision. Once it decides, it receives some feedback on its decision, some new information, and the situation repeats. In the case of our robot, the information would be the sensor inputs, and the decision would be what output strength to assign to each motor.

An MDP is a special case of decision process that has the Markov property [6]. A full specification of the Markov property is outside of the scope of this report, but for the purposes of this report, a process is considered Markov if every state of the process provides sufficient information to predict the expected future reward of all possible actions in that state. Before deploying the notation used by the algorithms in this report, some attempt at formally defining an MDP must be made.

In an MDP, there exists a set of states , a set of terminal states (where ), and a set of actions . Time progresses in discrete steps. At each time-step , the agent receives a tuple containing a state and reward , then takes an action . The process then transitions to a new tuple , chosen stochastically with a probability dependent on and . The process continues until it reaches a point where , then halts.

Note that in the context of reinforcement learning, a single pass through a decision problem, from the initial state at to a terminal state, is called an “episode” [1].

## Value Functions and Decision-Making Approaches

A policy is a mapping . Policies can be either deterministic; suggesting the same action every time for a given state, or stochastic; choosing from a probability spread of actions [7]. The problem of decision making can be framed as finding a policy that causes the agent to receive the greatest possible reward during the process, or over some other period of the process, i.e. the first ten steps. Due to the random element of policies and MDPs, it’s more precise to say that the agent is trying to maximise the *expected* reward. A particular series of actions may produce the most reward on average, yet lead to lesser reward in a particular instance due to bad luck.

If one had access to an oracle giving the true expected future reward from a decision, the problem would be rendered trivial (simply make the decisions with the greatest reward). It is obvious that the ability to accurately predict reward is very useful, and many of the methods investigated frequently attempt to do so, making estimates of expected reward using empirical observations [8, 4, 5, 9]. In the sources that were reviewed, there are two views from which future reward is measured; V, and Q. They will be referenced frequently in this section.

**V-Value.**

This function gives the expected sum of discounted reward received after starting in state , then following the policy

|  |  |
| --- | --- |
|  | [1] |

The V-Value of a state depends on the policy used because it determines which states will be visited in the future, and those states determine the reward received.

**Q-Value.**

This function gives the expected sum of discounted reward after taking action in state , then following the policy thereafter.

|  |  |
| --- | --- |
|  | [1] |

There are a variety of ways to classify reinforcement learning methods.

**V-Value-Based.**

The model learns to estimate the V-value of states. V-values can be used to construct policies. However, this is only possible if the manner in which the process transitions between states is known [10]. This can be done i.e. by calculating the expected value of the successor states reached by each action, then choosing the highest. However the utility of this approach is limited as the manner in which physical processes transition between states is often unknown or expensive to compute. No pure V-value based methods were implemented over the course of the project.

**Q-Value-Based.**

The model learns to approximate the Q-value of state-action pairs. These values are used to enact a policy [8, 11] i.e. by selecting the action with the highest Q-Value for the current state. Unlike a V-value-based approach, we only need the current state, and the action, so Q-values can be used to construct a policy even if the transition model is not known [12].

**Policy-Based.**

The model produces a probability spread of actions for a given state, this can be sampled from to enact a policy [1, 3]. During training, the model learns to output the action probabilities that produce the optimal policy. Estimations of future return are used to guide the learning process.

**Actor-Critic.**

The model has two components; the actor, and the critic. The actor is policy-based, the critic is value based. Either V-values [5, 9] or Q-values can be used [4], depending on the algorithm. The critic’s output is used to direct the actor during the learning process, and is used for nothing else, so only the actor’s output is necessary to enact the learned policy.

## Sampling Methods

The method by which training data is gathered is also an important factor in the final behaviour of agents. Note that these classifications are orthogonal to the output-structural methods, any combination of the two is possible.

**Monte-Carlo.**

The agent interacts with the environment until it reaches a terminal state (a full episode), no learning occurs during the episode [1]. This means that by definition, Monte-Carlo methods can only be applied to problems with episodes of finite length.

**Temporal Difference.**

The agent is allowed to interact with the environment. Model parameters are updated according to the reward received during each transition between states [1, 8]. This allows for learning over the course of a single episode, which can quicken convergence.

**N-Step.**

This term describes a middle-point between Monte-Carlo and Temporal-Difference methods. Updates are performed according to multiple transitions, but not the entire episode [1, 5]. i.e. training on the first 50 steps of a 100-step episode.

## Details of Targeted Algorithms

Table 1: Overview of Targeted RL Methods

|  |  |  |
| --- | --- | --- |
| Name | Sampling Method | Class |
| Deep Q-Learning. | TD | Q-Value |
| REINFORCE | Monte-Carlo | Policy-Gradient |
| Basic Actor-Critic | TD | Actor-Critic |
| Advantage Actor-Critic (A2C) | N-Step |
| Proximal Policy Optimisation |

### Q-Value Methods

Deep Q-Learning and SARSA are two closely related temporal-difference methods. Each transition is a tuple . Two consecutive states and actions, and the reward received after the first action. If the network’s approximation of is accurate, then by the definition of , the equality should hold. If the approximation is imperfect, then the equality will have some error, which can be used as the target of gradient descent.

|  |  |
| --- | --- |
|  | [1, 11, 8] |
|  | [1] |

Where is the discount rate (a hyperparameter).

The methods differ in that SARSA is on-policy, whereas Q-learning is off-policy [1]. Under SARSA, is the action prescribed by the policy that produced the transition. Whereas under Q-Learning is the action with the highest Q-value. Even if these policies are initially identical, they may diverge as weight updates are made.

### Pure Policy-Based Methods

The rest of the methods are policy-based, they perform network weight updates using a rule with the form:

Where is some measure of the quality of the agent’s action, usually an estimate of , or an estimate of . And where is the function that gives the probability of the network taking action in state .

If is positive, it is desirable to reinforce this behaviour, so the weights are updated to maximise the probability mass function, making the action more likely to be taken in the future. If it is negative, they are updated in the opposite direction.

**Entropy.**

The entropy of a policy captures the variance in the actions it prescribes. Several of the algorithms take into account the entropy of the agent’s policy, usually by adding it to the reward. This discourages the agent from adopting policies that assign high probabilities to a single action. Such a policy makes the agent more likely to take unusual actions and explore new states [13]. The entropy of a policy with respect to a particular state is given by:

|  |  |
| --- | --- |
|  | [13] |

Where is the function that gives the probability that policy will prescribe in .

#### REINFORCE

This is one of the earliest policy-based methods to be discussed. It is Monte-Carlo; the entire episode is observed before weight updates are performed [3]. The variant used in this report includes an entropy term. At the end of each episode, the network parameters are updated to minimise:

|  |  |  |
| --- | --- | --- |
|  | [3, 13] |  |

In the version used in this report, is the discounted sum of rewards received after an action.

### Actor-Critic Methods

The key feature of actor-critic methods is their use of two components; the actor and the critic. The actor learns in a policy-gradient manner; however the performance measure is calculated using the output of the critic [4, 5].

There are multiple ways to produce a performance measure, and these methods are described in more detail in the three implementations below. The methods discussed in this report use a single network for both the actor and critic; the network has outputs for both action probabilities and the critic outputs.

#### Basic Actor-Critic

In this algorithm, the critic learns to approximate , using TD error in the same manner as Q-learning. The actor learns by minimising the loss function:

|  |  |
| --- | --- |
|  | [4] |

Where is the critic’s approximation of .

#### Advantage Actor-Critic (A2C)

This n-step algorithm is a synchronous version of Asynchronous Advantage Actor-Critic [5].Advantage refers to the advantage function, which isolates the action’s contribution to the reward from the inherent value of the state.

|  |  |
| --- | --- |
|  | [5] |

In A2C, the critic learns to approximate [5], and its output is used at each step to calculate an estimate of ; according to the equation:

|  |  |
| --- | --- |
|  | [5] |

The critic learns in an n-step manner. The advantage estimate is squared and serves as an n-step error for :

|  |  |
| --- | --- |
|  | [5] |

The actor learns in the same n-step manner as the critic, according to:

|  |  |
| --- | --- |
|  | [5] |

#### Proximal Policy Optimisation

This algorithm is similar to A2C, and is calculated in the same manner. The key difference is that in PPO, the actor loss is limited within a range, according to:

|  |  |
| --- | --- |
|  | [9] |

Where is a hyperparameter of the network, such that . The definition of is left unclear in the original paper, it is simply referred to as the “entropy bonus” [9]. The notation was interpreted as , incorporating the entropy in a similar manner to REINFORCE and A2C.

# Specification and Design

## Methodology

The original plan was to proceed with development in a cyclical manner, in accordance with Agile principles. It was clear that it would be impossible to plan the project in the linear manner dictated by

waterfall approaches because the developer began the project with little knowledge of the topic. To develop that knowledge it was necessary for the process to allow for experimentation with practical implementations of the concepts involved. Using Agile sprints allowed this process to be perform in structured manner. Each method would be researched, then implemented, before proceeding to the next method. This same pattern would be used for the development of the environments.

It became apparent that this process was not as suitable as it initially seemed. The full research and development of an agent was too much work for a single sprint. The difficulty also varied depending on the algorithm, so it was difficult to set a reasonable time span for sprints. Sprints were originally planned, but the structure was not really adhered to and was eventually abandoned in favour of a text-based TODO list.

Throughout the process, the objectives of the project were revisited and adapted based on insights gained during the pursuit of other goals.

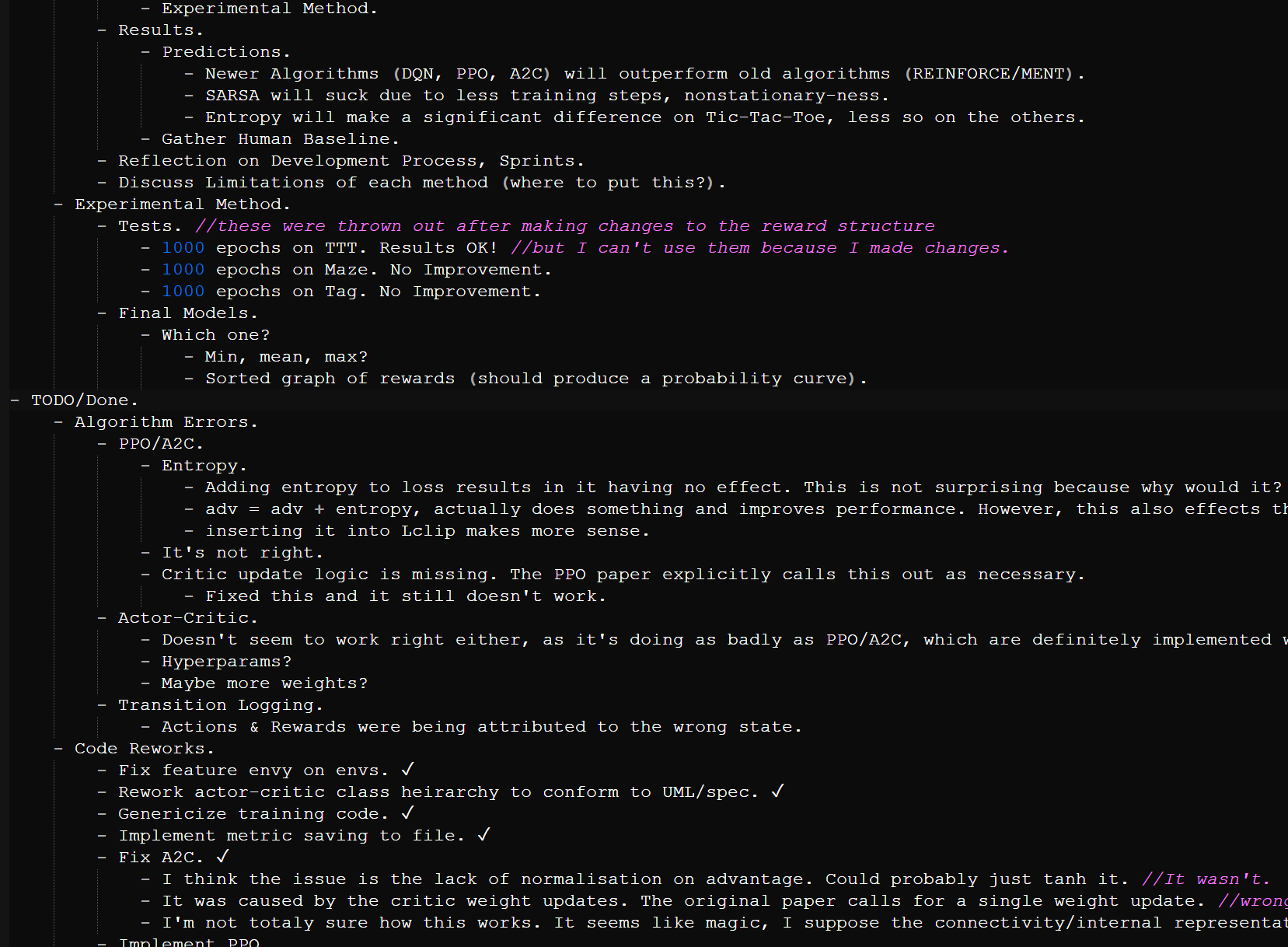


Figure 1. Excerpt from the TODO list.

## Analysis

It was clear from the research detailed in section 2, as well as from discussions with my project supervisor, that the problem of reinforcement learning is typically viewed through an agent-environment dichotomy. The agent and its environment each map naturally to a software element.

Two properties of the agent followed directly from the definition established in section 2. The agent must be capable of making decisions, following a particular policy. The agent must also be able to improve its policy using the data gathered from its interactions with its environment.

Why were these particular algorithms chosen?

* DQN was chosen to include a pure value-based algorithm.
* REINFORCE was chosen as it represents the most basic form of the policy gradient family of algorithms, this fact makes it a useful baseline for the rest of the algorithms, which ought to outperform it.
* Basic Actor critic was chosen on a similar basis, as the simplest of the three actor-critic algorithms.
* A2C and PPO were chosen because they are more recent, and it was felt that placing too much focus on older algorithms would prevent the project from accurately reflecting the current state of the field.

Two additional algorithms were originally planned for comparison; SARSA, and REINFORCE without entropy. They were cut. The decision was made after experimenting with preliminary comparison graphs, as it became clear that it would be difficult to visualise a comparison between so many algorithms. SARSA and REINFORCE were chosen for removal as both are similar to another targeted algorithm – DQN and REINFORCE with entropy, respectively – that was expected to demonstrate superior performance.

Prior to the final training and measurement step, it was unclear how quickly the agents would improve. It could be that the environments are too generally challenging for learning to occur under the chosen algorithms. If this were the case for all algorithms, no meaningful comparison of their performance could be made. For this reason, the environment specifications were written to include tuneable parameters that could be used to reduce their difficulty.

Based on the research performed during the project, it was decided that each environment should present a Markov decision problem. The environments were designed such that their state representation provides sufficient information to predict the optimal action. If this were not the case, no agent would be able to do so, and the final metrics would be dominated by random chance.

A total of three environments were designed, with diverse properties. The intent was to create a variety of environments, as it was hoped that this would help highlight the strengths and weaknesses of each algorithm. Tag has a continuous state space. Tic-Tac-Toe is turn-based and adversarial, there is an opponent agent that also makes decisions. Maze has a random element and rewards planning. I briefly considered – and partially implemented – an environment with a large image percept for use with convolutional neural networks. However, after discussing the idea with my supervisor I determined that training such a network would be computationally infeasible.

Each environment includes a random element. This is one way to increase the difficulty of the task. Randomisation forces the agent to learn a general policy for the problem, rather than memorising one specific sequence of actions.

Each environment has a GUI and is playable by humans. There are several reasons why this was deemed necessary: it allows for testing of the environment, and it allows a human operator to observe the behaviour of agents. The ability to observe agents will be used during the final project demonstration, and allows for testing for undesirable behaviour i.e. exploitation of bugs in the environment.

Manually measuring the performance of agents would take an unfeasibly long time, so the software must be able to save these measurements automatically. As with any scientific process it is important that the measurements are replicable, so it should be possible to run the system repeatedly and produce identical measurements each time.

The details of the evaluation process were chosen based on the research performed in section 2. The metrics were graphed by epoch rather than by real-time to achieve reproducibility, as real-time metrics would be influenced by uncontrollable factors i.e. other processes on the host machine. Reward was chosen as the metric as it is strong measure of the agent’s performance, so long as reward is distributed appropriately. One other possible choice worth mentioning is loss, as it is the typical measure of a model’s quality in supervised learning. Reinforcement learning loss in not an effective measure of the quality of an agent’s choices during. This became clear early in the project, after experimentation with the agents. Instead, loss represents something closer to the current rate at which the agent is learning. A low loss or stable loss indicates that the agent has converged to some policy, at least for the observed data, but not that it is a good one. Consider a simple problem bandit problem where one action provides -1 reward, and the other provides +1. Q-learning could achieve zero loss by only ever taking the -1 action, and correctly predicting its reward.

## Requirements

### Functional Requirements

**Training and Evaluation System.**

* User can train a set of agents.
* User can save training metrics.
* User can graph training metrics.

**Reinforcement Learning Agents**.

* User can request an action from the agent.
* User can request agent to update its policy.

**Maze Environment.**

* Player can take actions.
* Up.
* Down.
* Left.
* Right.
* User can configure environment.
  + Initial placement of agent avatar (set, or random).
  + Number of coins.
  + Length of game.
  + Enable or disable exploration reward.
  + Enable or disable distance reward.
* User can reset environment.

**Tag Environment.**

* Player can take actions.
  + Turn Left.
  + Continue Straight.
  + Turn Right.
* User can configure environment.
  + Number of time steps until episode termination.
  + Number of seekers.
  + Speed ratio between runner and seeker.
  + Maximum deflection of seeker spawn angle from directly behind runner.
  + Min distance seekers can spawn from runner.
  + Max distance seekers can spawn from runner.
  + Height of arena.
  + Width of arena.
* User can reset environment.

**Tic-Tac-Toe Environment.**

* Player can take actions.
  + Place Symbol (one action for each cell).
* User can configure environment.
  + Dimensions of board.
  + Opponent strategy.
* User can reset environment.

### Non-Functional Requirements

**Training and Evaluation System.**

* Training must be replicable.

**Reinforcement Learning Agents.**

* Efficient use of computational resources.
* Effective at learning to solve decision problems.

**Environments.**

* Efficient use of computational resources.
* Human-readable interface.
* Configurable difficulty.
* The problem posed by the environment is Markov.

### Design

### Interface Design

The focus of the project was on the AI agents, which do not use the GUI in their decision-making process. The GUI is primarily intended as a debugging tool for the developer. As such, the visual presentation of the GUIs was not considered to be as important as typical software, aimed at end users. The UI was very simple, due to several contributing factors;

1. Due to the limited computational resources available, the agents required simple percepts, which mapped to equally simple visual depictions.
2. As all the methods focus on discrete action spaces, the controls necessary for a human are also simple. Any environment necessitating complex UI design — multiple pages, input forms, many simultaneous inputs — was unlikely to be solvable using the RL methods.
3. The nature of the project requires the end user to interact with its code directly. This meant that it was reasonable to expect the user to alter the environment configuration in code, rather than providing a GUI to change it at runtime i.e. a settings menu.

This simplicity allowed traditional approaches such as wireframes to be eschewed in favour of rapid prototyping. Screenshots of the final UI are included in section 5.2.1.

## System Design

### Maze

In this environment, the world is a grid of squares. The agent controls an avatar, and is tasked with moving it around the grid to collect coins. There are empty squares that the agent can move through, and solid walls that block its movement. The location of walls is preset but the location of coins is randomly generated. The agent’s starting position can be randomly chosen or preset. The episode ends after a set number of steps, configurable with a default of 50.

The percept is the contents of each square, from the set {Empty, Wall, Agent, Coin, Visited, Agent+Coin, Agent+Visited, Agent+Coin+Visited}.

**Reward scheme.**

* +500 for collecting a coin.
* +50 for exploring a new square.
* -[Manhattan distance to nearest coin] at each step.

The latter two rewards were added after testing demonstrated that most agents failed to improve, and was intended to ameliorate the sparse nature of the coin-based reward.

### Tag

The agent is tasked with controlling an avatar in a 2D environment. The goal is to prevent the agent avatar (the runner) from contacting hostile agents (the seekers). At the beginning of each episode the runner starts at the center of the arena, facing in a random direction. The seekers start behind the runner, with a slight random deflection in angle relative to the seeker, and a randomised distance. At each step, the seekers move directly towards the runner. The episode ends when the runner collides with a seeker, or moves off the edge of the game area. The episode also ends if the runner successfully evades the seekers for a set number of steps, configurable with a default of 1000.

The percept contains the position of the agent, the rotation of the agent, and the positions of each seeker.

**Reward scheme.**

* +1 per time step.
* -50 upon colliding with a seeker, or leaving the game area.

### Tic-Tac-Toe

This is an arbitrary-size version of the game. The environment is an N⨯N grid (where N is a configurable parameter of the environment). At each time step the agent marks an empty cell with its symbol, then the opponent marks an empty cell with its symbol. The winner is the first player to construct a line of N symbols. By default the opponent has a 75% chance of taking the optimal action, and takes a random action the rest of the time. The player always makes the first move.

The percept is the state of the game board. Rather than absolute values indicating whether a cell is empty, nought, or cross, the values are relative to the current player, from the set {Empty, Player, Opponent}.

**Reward scheme.**

* +1 per time step.
* +2n for a length n line of symbols.
* +10n for winning a game (where n is the length of line required to win).
* -100 for taking an invalid move.

### Reinforcement Learning Agents

Each agent represents one of the algorithms described in section 2. Each agent is responsible for all behaviour that is unique to its particular algorithms. This includes; holding an internal policy, updating its policy in accordance with the definition of its algorithm, and producing actions based on that policy in response to a particular state.

### Training System

The training system is responsible for mediating the interaction of the agents and environments, and for all behaviour that applies to reinforcement learning agents generally. It is also responsible for storing and visualising data gathered during training (metrics).

# Product

## Implementation

### Reinforcement Learning Agents

TensorFlow was used to handle the implementation of neural networks and automatic differentiation, and was chosen due to the developer’s personal familiarity with the library. This choice determined the language used for the rest of the project (Python 3.11.3).

Each agent class corresponds to a training algorithm. Each agent class is intended to be as generic as possible, using composition to create variation in network architecture and input/output format. Variation in the training method is implemented using inheritance. The base classes; AbstractQAgent, AbstractPolicyAgent, and AbstractActorCriticAgent, are responsible for deciding actions, but not for training. Training logic is handled by their subclasses, each of which implements one of the targeted algorithms. AbstractQAgent and AbstractPolicyAgent both handle initialisation of layers, whereas AbstractActorCriticAgent does not. The critic architecture differs between Actor-Critic and A2C/PPO, and each implementations uses a single network for both the actor and critic. This means that there is no shared initialisation logic.

The design of the agents follows guidelines set out in the TensorFlow documentation [14]. Those guidelines recommend that custom models be declared as a subclass of the TensorFlow Model class, with custom training logic implemented in the fit & train\_step methods. Features necessary to interface with a particular environment; the number of outputs, and the structure of hidden layers, are variable and implemented using composition. There are three abstract agent classes, which are responsible for the implementation of the act function, as well as the initialisation of the neural network and its optimiser. However, these classes implement no training logic. Training logic is provided by the various subclasses, each of which implements a single algorithm.

At the outset of the project, the developer’s understanding of the algorithms was incomplete, which made it impossible to construct a class hierarchy of agents. It was decided that a flat class hierarchy should be used, with each agent inheriting only from Tensorflow’s Model class. As the algorithms were implemented, this was revealed as a bad decision; code duplication made it even more complicated. Consequently, the class hierarchy was reworked to the current design, using the newly developed understanding of the workings of each algorithm.

The two cut algorithms (SARSA and REINFORCE) were fully implemented before being cut, though due to their extreme similarity to other algorithms, this did not waste a significant amount of development time.

### Environments

Aside from several bugs caused by poor testing during the early stages of the project, development of the environments was quite simple, and took a few days at most. PyGame was selected to handle the implementation of complex rendering logic for the environments. It was felt that implementing custom rendering logic would take significant effort and distract from the focus of the project: the reinforcement learning agents. PyGame was also used for the collision detection in the Tag environment. The GUI code was based on the template provided in the PyGame documentation [15].

The environments conform to the OpenAI gym environment interface. This consists of the step and reset methods, as well as the manner in which the human-readable environment GUI is enabled. Before the three aforementioned environments were finished, one of the OpenAI gym environments (cartpole) was used to test agent implementations. The interface was retained due to the existing code, and for cross-compatibility with other machine learning projects.

Each environment class represents an environment that the agent interacts with. The environment is responsible for all game logic, while the view is responsible for handling all logic required to render the GUI. A corresponding view is only created if the human-readable mode of the environment is requested by the user. The observer pattern is used to update the view each time the model changes. Each view inherits from the Observer class, and each environment inherits from Observable.

#### Maze

The percept is one-hot encoded. This choice was made because the environment was planned to be used in conjunction with convolutional neural networks. Tensorflow’s Conv2D layer expects a 3D percept, where the last dimension represents multiple channels of data, and one-hot coding conveniently fits this structure.

#### Tag

This is the arguably the most complex environment in terms of behaviour, due to the collision and moving camera. However, PyGame essentially handles the entire process, considerably simplifying the work needed. As with any 2D game, much of the manually implemented code is simple trigonometric maths for positioning and rotating objects.

#### Tic-Tac-Toe

The default opponent is implemented using a mini-max game tree search, it attempts to win the game with the least number of moves. Alpha-beta pruning was implemented to improve performance, in accordance with the non-functional requirements for environments. Though the game itself is not particularly complex, this search logic was likely the most complex element of any of the environments.

### Training & Evaluation System

The agents train in parallel for the same set number of episodes. At each step of the episode, the training logic is delegated to the agent, passing it the new state of the process. The agent processes the data, performs weight updates, then returns control. A set RNG seed is used, and used by all components – environments, agents – to determine their random elements. Training is parallelised using Python’s multiprocessing library; each agent has a single process, which executes sequentially. Once an agent is finished training, its final weights are written to a file. The metrics for all agents – currently only the reward – are then collated, graphed, and written to a single file. Model weights are stored in .tf format, Metrics are stored in .npz format. Graphing was performed using Matplotlib.

At an earlier stage of the project, it was planned that .csv files would be used to store metrics. However it became evident that .npz was more appropriate due to the potentially multidimensional nature of the data. Earlier iterations of the software experimented with concurrency, but in a different manner. This earlier implementation used concurrency at the agent level; with several threads accumulating gradients and then updating a target network. However, this introduced nondeterminism into the training process (due to race conditions), and thus failed to meet the software requirements. Solving this issue was not seen as a valuable use of development time, so the approach was abandoned in favour of a sequential approach. Eventually that too was replaced, by the current implementation.

For convenience, the training and evaluation system was split into four separate Python files. The first three files train agents for one of the three environment, the fourth is capable of re-plotting metrics generated for any environment.

As part of the evaluation process, non-reinforcement learning agents were used to provide a baseline. The Tic-Tac-Toe agent is the mini-max search agent described in section 4.1.2. The Maze agent was made specifically for this purpose. It follows the shortest path to the nearest coin, calculated using A\* search, and using the Manhattan distance as the heuristic.

## Verification & Validation

### Unit Testing

The agents were difficult to test for three key reasons;

1. Though easily to decide intuitively, it is hard to construct a generalised automated process to determine whether an algorithm has significantly improved an agent’s policy.
2. Proving that there is an error with an algorithm requires finding evidence of absence, which is notoriously difficult. It could be that the algorithm is implemented correctly, and the lack of improvement is due to other factors; too-short training, an impossible environment, or poorly chosen hyperparameters.
3. The stochastic nature of the environments, and the network weight initialisation, means that agents can fail to learn due to random chance. Though this is highly unlikely.

The agents were validated by training them on an extremely simple environment; a 2-armed bandit where only one action provides reward. If an agent cannot learn the optimal policy for this problem, it strongly suggests the presence of an implementation error. This approach was limited however, as the inverse does not necessarily hold.

During the development of the environments, ad-hoc testing was performed. Once all features were believed to be successfully implemented, unit tests were written, and manual integration was performed according to a formal specification. The unit tests are included in the attached code.

Unit testing was performed using Python’s unittest module.

### Manual Testing Procedures

In addition to unit testing, manual testing of the environments was performed. This was necessary to verify the correctness of the GUI, as it could not be effectively unit tested.

Details of the testing procedure are included in appendix B.

At the time of submission, the software component passes all tests.

# Results & Evaluation

### Evaluation Process

All evaluations in this section were performed using the default parameters for each environment.

The average score under the optimal policy for each environment is listed below. For tag, the nature of the environment is such that the optimal policy will always survive until episode termination, so the expected reward is trivial to determine. The optimal policy for Maze and Tic-Tac-Toe policy is known, so these values were determined by running a near-optimal agent for 100 episodes, calculating the average reward, and rounding it to the nearest integer.

|  |  |  |  |
| --- | --- | --- | --- |
| Environment | Maze | Tag | Tic-Tac-Toe |
| Baseline | 5588 | 1000 | 950 |

All maze agents used one convolutional layers with a 3×3 kernel and stride 1, a flatten layer, then the output layer.

Table 2. Maze Agent Configuration.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | DQN | REINFORCE | Actor-Critic | A2C | PPO |
| learning rate | 0.00015 | 0.00015 | 0.00015 | 0.00015 | 0.00015 |
| epsilon | 0.2 | 0.2 | 0.2 | 0.2 | 0.2 |
| epsilon decay | 1 | 1 | 1 | 1 | 1 |
| discount rate | 0.99 | 0.99 | 0.99 | 0.99 | 0.99 |
| entropy weight | n/a | 5 | 5 | 5 | 5 |
| critic weight | n/a | n/a | 2 | 2 | 2 |
| max n-step size | n/a | n/a | n/a | 10 | 10 |
| interval | n/a | n/a | n/a | n/a | 0.2 |
| replay memory | 1000 | n/a | 1000 | n/a | n/a |
| mini-batch size | 100 | n/a | 100 | n/a | n/a |

All Tag agents used a single dense hidden layer with 4 neurons.

Table 3. Tag Agent Configuration.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | DQN | REINFORCE | Actor-Critic | A2C | PPO |
| learning rate | 0.0001 | 0.001 | 0.0001 | 0.0001 | 0.0001 |
| epsilon | 0.05 | 0.05 | 0.05 | 0.05 | 0.05 |
| epsilon decay | 1 | 1 | 1 | 1 | 1 |
| discount rate | 0.95 | 0.95 | 0.95 | 0.95 | 0.95 |
| entropy weight | n/a | 1 | 1 | 1 | 1 |
| critic weight | n/a | n/a | 2 | 2 | 2 |
| max n-step size | n/a | n/a | n/a | 1000 | 1000 |
| interval | n/a | n/a | n/a | n/a | 0.2 |
| replay memory | 2000 | n/a | 1500 | n/a | n/a |
| mini-batch size | 100 | n/a | 100 | n/a | n/a |

All Tic-Tac-Toe agents used a Flatten layer, then a dense single hidden layer with 8 neurons.

Table 4. Tic-Tac-Toe Agent Configuration.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | DQN | REINFORCE | Actor-Critic | A2C | PPO |
| learning rate | 0.001 | 0.001 | 0.001 | 0.001 | 0.001 |
| epsilon | 0.1 | 0.1 | 0.1 | 0.1 | 0.1 |
| epsilon decay | 1 | 1 | 1 | 1 | 1 |
| discount rate | 0.9 | 0.9 | 0.9 | 0.9 | 0.9 |
| entropy weight | n/a | 0.1 | 0.1 | 0.1 | 0.1 |
| critic weight | n/a | n/a | 2 | 2 | 2 |
| max n-step size | n/a | n/a | n/a | 1000 | 1000 |
| interval | n/a | n/a | n/a | n/a | 0.2 |
| replay memory | 1000 | n/a | 1000 | n/a | n/a |
| mini-batch size | 100 | n/a | 100 | n/a | n/a |

## Results of Evaluation

### Environments, Testing & Evaluation

The environments and evaluation system fulfil the requirements stated in section 3. The evaluation system is capable of training agents concurrently, and graphing the reward per episode. Each environment presents a suitable challenge for reinforcement learning agents. The environments be used in conjunction with the RL agents and training system, or manually interacted with by a human user.



Figure 2. Maze environment final GUI.



Figure 3. Tag environment final GUI.



Figure 4. Tic-Tac-Toe environment final GUI.

### Agents

The agents were successfully implemented according to the specification, the applied to each environment. The following graphs show the reward attained during training.

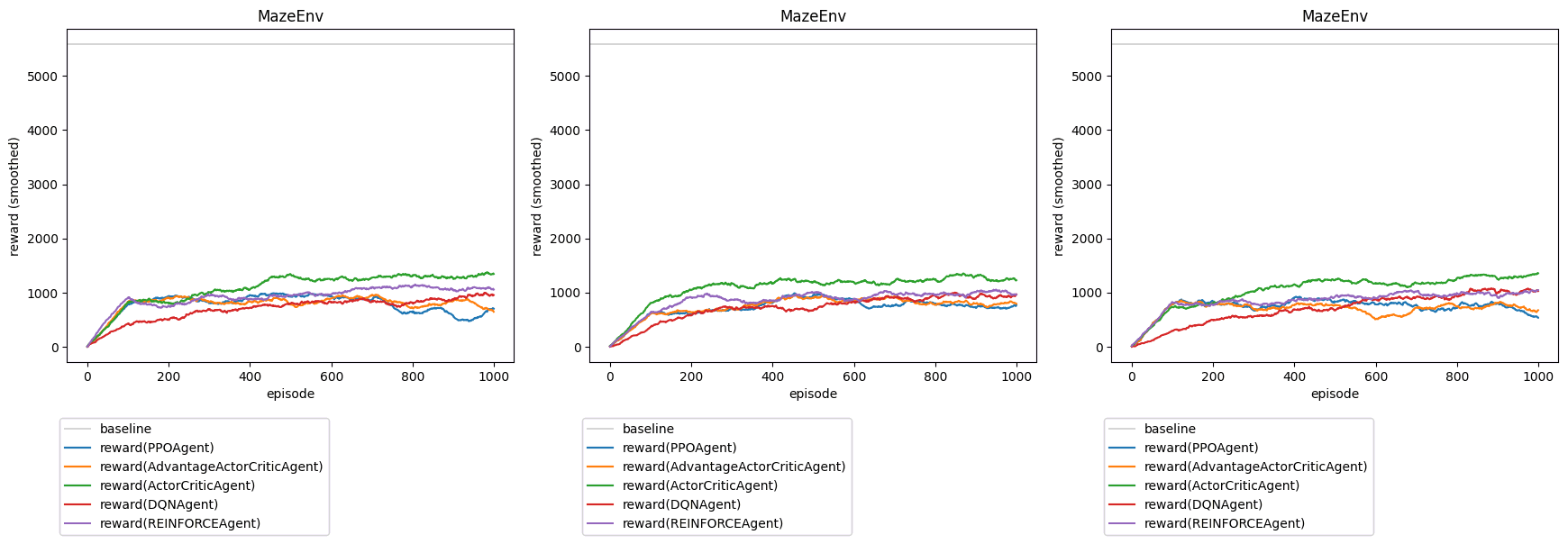


Figure 5. Maze evaluations. From left to right, seeds; 0, 2000, 3000. Size-100 moving average.

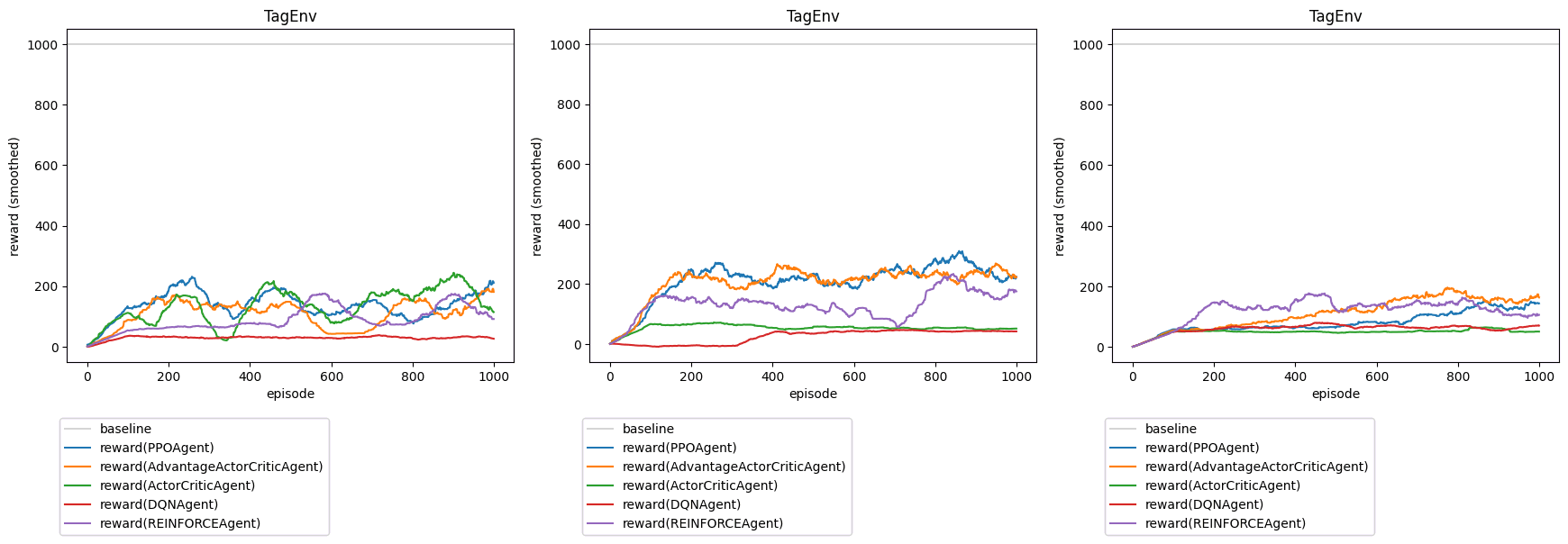


Figure 6. Tag evaluations. From left to right, seeds; 0, 1000, 2000. Size-100 moving average.

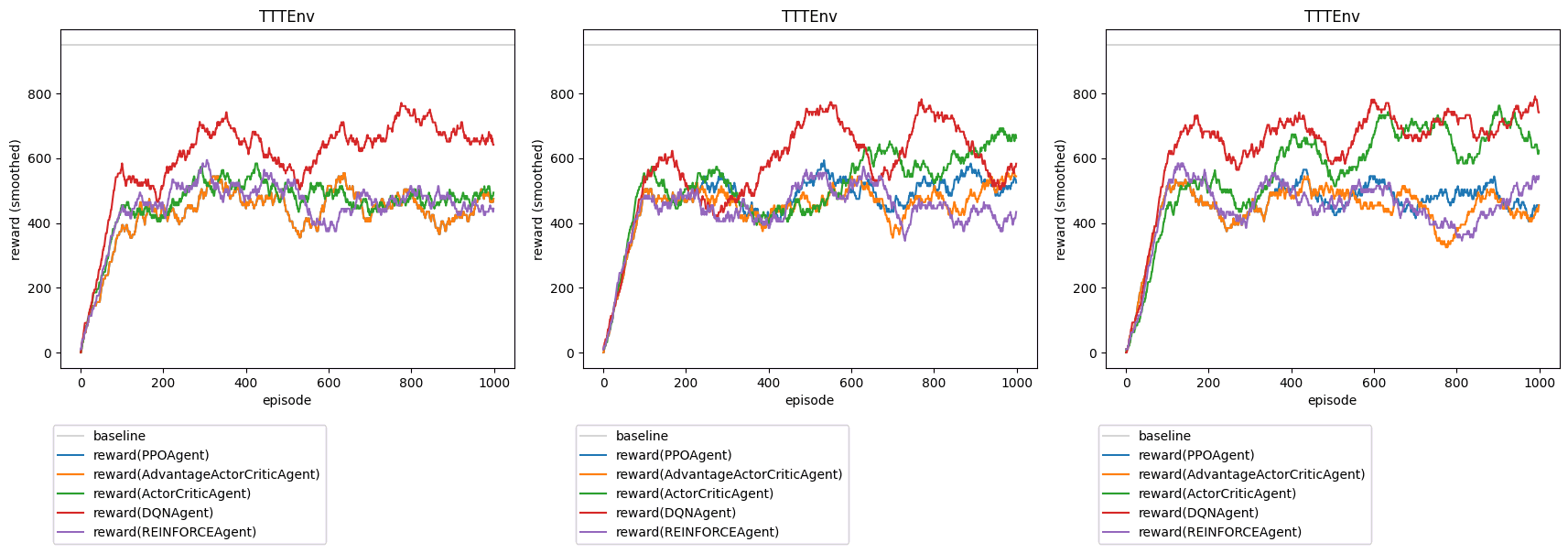


Figure 7. Tic-Tac-Toe evaluations. From left to right, seeds; 0, 1000, 2000. Size 100 moving average.

## Returning to the Research Questions

### Effectiveness of Algorithms.

There is a clear upward trend on most of the trials, with the majority of algorithms being able to generate improvement on all three environments. On each environment, at least one agent demonstrated improvement, and each algorithm demonstrated improvement on at least one environment. Agents performed worst on Tag, with several failing to learn, and the best achieving only ≈20% of the baseline. Agents performed better relative to the baseline on Tic-Tac-Toe than on the other environments, likely due to the simple nature of the game.

### Strengths & Weaknesses.

REINFORCE performed consistently on all three environments, receiving a middling final reward in 7 of 9 trials.

DQNExcelled at Tic-Tac-Toe, but consistently failed to learn on Tag. Though it is not clear from the evaluation which features of the environment are responsible for this effect.

Actor-Critic struggled with the Tag environment. It also demonstrated unstable performance, improving in only one trial. The algorithm performed well on the other environments, so this may represent a hyperparameter selection issue.

A2C and PPO shared the same strengths and weaknesses, performing similarly in every trial. This is unsurprising, as the algorithms are closely related. They outperformed the other algorithms on Tag, but demonstrated low performance on Maze and Tic-Tac-Toe. This poor performance it surprising, as the algorithms are two of the more recent and sophisticated in the report, however the cause is not clear from the evaluation. It seems unlikely that it is related to the environment, and may be a result of the chosen hyperparameters.

# Discussion & Reflection

## Interpreting the Results

The evaluation demonstrated the ability of these algorithms to create self-improving agents for simple tasks. However, the improvement demonstrated was weak; the agents improved slowly, and within the short training period of the trial they achieved low scores relative to the baseline. This is consistent with the research in [11, 5, 9] which demonstrate improvement, but only over extremely long training periods.

## Reflection

The final aims of the project were not specified in sufficient detail early enough. The algorithm choices were not solidified until late into the process (November 2023), and the final list was only determined several weeks before the deadline. Work proceeded with the vague direction of learning about reinforcement learning, and then of evaluating algorithms, but the specific criteria; comparison relative to baselines, strengths, and weaknesses, were not determined until the final stages of the project. Deciding the final criteria earlier would have made it easier to set objectives. Whether this is realistic is up for debate. While in hindsight it is easy to see that it was an error, at the time the poor understanding of the material made this difficult. Greater time investment during the early stages of the process may have ameliorated this issue.

Multiprocessing should have been implemented earlier. It was avoided due to difficulties with TensorFlow models and concurrency, and only tackled once the deadline loomed. Once it was attempted, it was not particularly difficult. The long training times before this decision was made were demoralising, and could have been avoided.

It would have been wise to proceed with the development of the environments in a test-focused manner, or to otherwise test them earlier. Once the unit tests were implemented, they caught several bugs that had been missed during ad-hoc testing. These bugs likely caused unexpected behaviour when the environments were integrated with other components, and would likely have eased the development of those other components.

Choosing PyGame was not ideal as it required an event handling loop in any file where the GUI of an environment was drawn. This was not a particularly severe issue, but it would probably have been better to pick an alternative library.

## Challenges

Understanding the details of each algorithm was by far the most difficult task, and the implementations were reworked numerous times due to errors. The main difficulty was in interpreting the papers that described each algorithm. The use of obscure and confusing notation made the earlier papers [3, 4] particularly difficult.

The design of the environments was also fairly difficult, much harder than implementing them. It was difficult to come up with games that would be interesting, yet possible for the agents to complete. It was also difficult to decide on reward schemes for the environments, and they had to be reworked several times in response to training failures.

The sensitivity of the process to hyperparameters presented a challenge. Slight changes could cause the agents to fail to learn, or cause the loss the explode, leading to a crash. This slowed down the evaluation process as parameters had to be carefully determined by trial-and-error. A set of hyperparameters might lead to learning with one random seed, but fail on another. Finding hyperparameters that performed well across a range of random seeds was difficult.

## Future Work

The two most obvious starting points are to implement more algorithms, and implement more environments, especially environments with properties not already explored; For instance, all of the environments in this report have finite and short episode lengths.

Running the training process for a longer period would likely produce better agents with little effort.

As addressed in section 6.5, the software was poorly optimised, so optimised implementations could be the topic of future work.

## Limitations

The performance of the project is poor, as little attempt has been made to optimise it. As it was deemed to be unfeasible within the timescale. The training system uses an excessive amount of memory, growing with time, which suggests the presence of a memory leak. As a result, when all five agents are trained simultaneously, the training system is limited to episode counts on the order of 1000.

The games are not particularly compelling. The project would have benefited from more complex environments, but computational limitations meant that the environments had to be simplified to be suitable for the agents. The poor performance of the training system exacerbated this issue, as longer training durations would likely have allowed the agents to approach more complex tasks.

The evaluation could be improved. For instance, more tests could have been performed and averaged, data could have been gathered using a range of hyperparameters. There is insufficient data to make any strong conclusions about specific behaviour of any of the targeted algorithms.

# Conclusion

In conclusion, the objectives of the project were addressed. Three environments were implemented, with a range of different properties. Reinforcement methods were investigated, applied to the environments, and found to be moderately effective at solving several those basic tasks. However there was room for improvement; the training efficiency was poor, and the behaviour varied significantly depending on the random seed.

References

|  |  |
| --- | --- |
| [1] | R. S. Sutton and A. G. Barto, Reinforcement Learning: An Introduction, Second Edition, MIT Press, 2018. |
| [2] | A. Fawzi, M. Balog, A. Huang, T. Hubert, B. Romera-Paredes, M. Barekatain, A. Novikov, F. J. R. Ruiz, J. Schrittwieser, G. Swirszcz, D. Silver and P. Kohli, “Discovering faster matrix multiplication algorithms with reinforcement learning,” *Nature,* no. 610, p. 47–53, 2022. |
| [3] | R. J. Williams, “Simple Statistical Gradient-Following Algorithms for Connectionist Reinforcement Learning,” *Machine Learning,* vol. 8, p. 229–256, 1992. |
| [4] | V. Konda and J. Tsitsiklis, “Actor-Critic Algorithms,” *Advances in neural information processing systems,* vol. 12, 1999. |
| [5] | M. Volodymyr, A. P. Badia, M. Mirza, A. Graves, T. Harley, L. P. Timothy, S. David and K. Kavukcuoglu, “Asynchronous Methods for Deep Reinforcement Learning,” in *Proceedings of The 33rd International Conference on Machine Learning*, 2016. |
| [6] | Stanford Online, “Stanford CS234: Reinforcement Learning | Winter 2019 | Lecture 1 - Introduction - Emma Brunskill,” 30 March 2019. [Online]. Available: https://www.youtube.com/watch?v=FgzM3zpZ55o. [Accessed 27 March 2024]. |
| [7] | Stanford Online, “Stanford CS234: Reinforcement Learning | Winter 2019 | Lecture 8 - Policy Gradient I,” 15 April 2019. [Online]. Available: https://www.youtube.com/watch?v=8LEuyYXGQjU. [Accessed 27 March 2024]. |
| [8] | P. Dayan and C. J. Watkins, “Q-Learning,” *Machine Learning,* vol. 8, pp. 279-292, 1992. |
| [9] | J. Schulman, F. Wolski, P. Dhariwal, A. Radford and O. Klimov, “Proximal Policy Optimization Algorithms,” *arXiv:1707.06347 [cs.LG],* 2018. |
| [10] | Stanford Online, “Stanford CS234: Reinforcement Learning | Winter 2019 | Lecture 2 - Given a Model of the World,” 30 March 2019. [Online]. Available: https://www.youtube.com/watch?v=E3f2Camj0Is. [Accessed 27 March 2024]. |
| [11] | V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra and M. Riedmiller, “Playing Atari with Deep Reinforcement Learning,” *arXiv:1312.5602 [cs.LG],* 2013. |
| [12] | Stanford Online, “Stanford CS234: Reinforcement Learning | Winter 2019 | Lecture 4 - Model Free Control,” 30 March 2019. [Online]. Available: https://www.youtube.com/watch?v=j080VBVGkfQ. [Accessed 27 March 2024]. |
| [13] | R. J. Williams and J. Peng, “Function Optimization Using Connectionist Reinforcement Learning Algorithms,” *Connection Science,* no. 3, pp. 241-268, 1991. |
| [14] | F. Chollet, “Making new layers and models via subclassing,” 24 July 2023. [Online]. Available: https://www.tensorflow.org/guide/keras/making\_new\_layers\_and\_models\_via\_subclassing. [Accessed 25 March 2024]. |
| [15] | PyGame, “Pygame Front Page,” [Online]. Available: https://www.pygame.org/docs/. [Accessed 25 March 2024]. |

1. Detailed Specification And Design

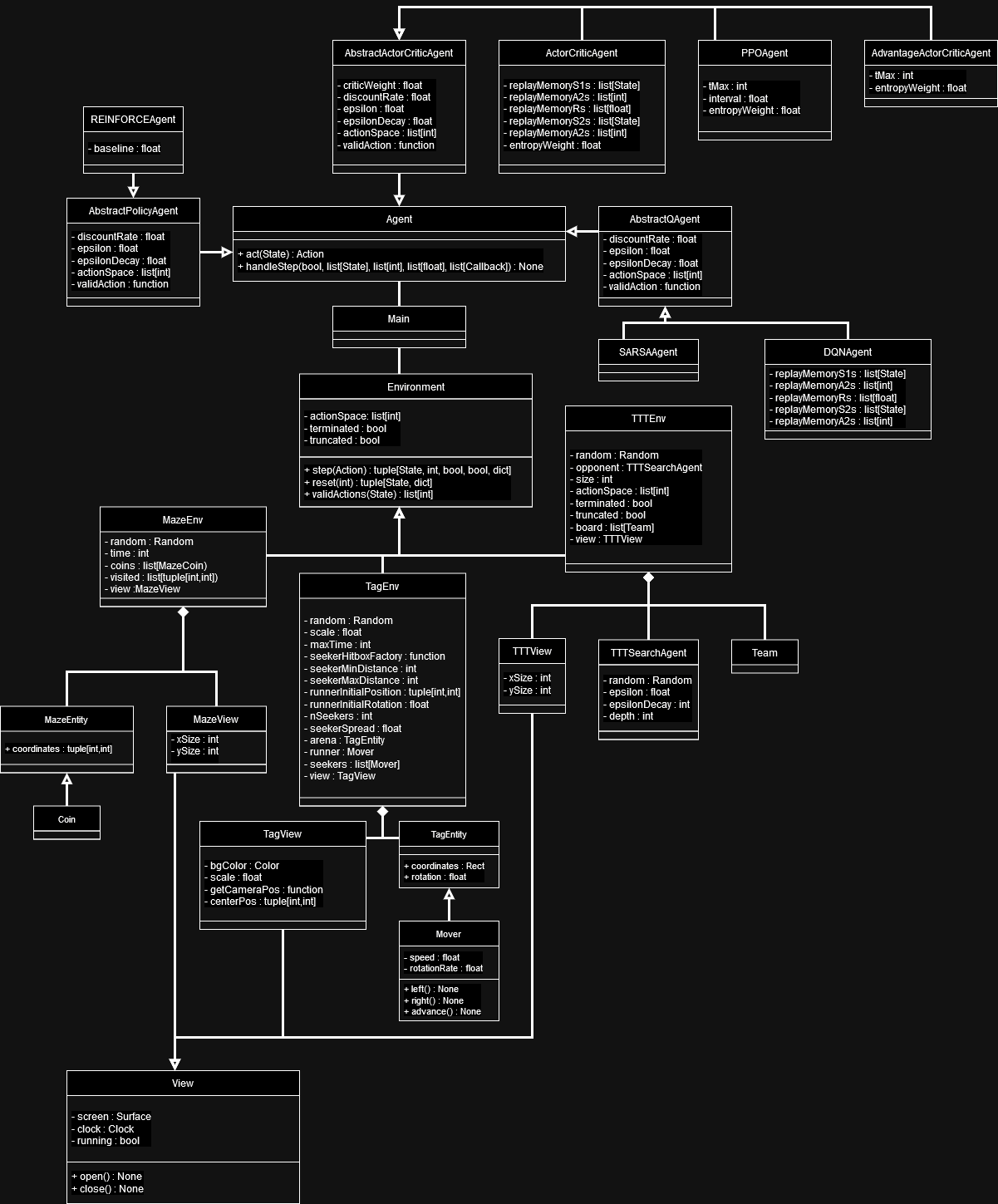


Figure 8. UML diagram of the Python codebase. The observer-observable classes and their relationships were omitted to avoid visual confusion.

1. Detailed Test Strategy & Test Cases

|  |  |  |
| --- | --- | --- |
| Method Tested | Description | Result |
| Agent.act, Agent.handleStep | Each agent is run on a basic 2-arm bandit environment for 100 episodes. Once training is complete, the average score is checked to ensure that the agents have improved sufficiently. There are five such tests, one for each agent. | Pass. |
| MazeEnv.step | Several step operations are called on the same MazeEnv. The state is interrogated each time to ensure that it was updated correctly. | Pass. |
| MazeEnv.reset | An MazeEnv’s state is mutated, then the reset method is called. The state is then interrogated to determine that it has been correctly reset. | Pass. |
| TagEnv.step | Several step operations are called on the same TagEnv. The state is interrogated each time to ensure that it was updated correctly. | Pass. |
| TagEnv.reset | An TagEnv’s state is mutated, then the reset method is called. The state is then interrogated to determine that it has been correctly reset. | Pass. |
| TTTSearchAgent.act | A TTTSearchAgent is provided with several manually defined Tic-Tac-Toe percepts, and the resulting actions are compared with the optimal actions to ensure equality. | Pass. |
| TTTEnv.step | Several step operations are called on the same TTTEnv. The state is interrogated each time to ensure that it was updated correctly. | Pass. |
| TTTEnv.reset | An TTTEnv’s state is mutated, then the reset method is called. The state is then interrogated to determine that it has been correctly reset. | Pass. |

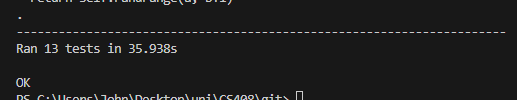


Figure 9. Result of running the unit tests.

Table 5. Maze Manual Tests.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Test | Environment Config | Procedure | Expected Result | Actual Result |
| Controls | Render mode: human, number of coins: 0. | Press each control button at least once. | All control mappings are correct. No visual errors. | Pass. |
| Coins | Render mode: human, reward exploration: false, reward distance: false. | Collect as many coins as possible during the episode. | All collected coins are immediately replaced. Final score is equal to 500 × number of coins collected. No visual errors. | Pass. |
| Exploration | Render mode: human, Reward distance: false, number of coins: 0. | Visit every square at least once. | Final score equal to 50 × number of squares visited - 50. No visual errors. | Pass. |
| Collision | Render mode: human, number of coins: 0. | Collide with all edges, and all solid squares. | Attempting to move off the edge or into solid squares results in no movement. No visual errors. | Pass. |

Table 6. Tag Manual Tests.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Test | Environment Config | Procedure | Expected Result | Actual Result |
| Controls. | Render mode: human, number of seekers: 0. | Press every control button at least once. | Control mapping is correct. No visual errors. | Pass. |
| Lose, collide with seeker. | Render mode: human. | Collide with the seeker as quickly as possible. | Colliding with seeker ends the episode, as truncated. No visual errors. | Pass. |
| Lose, exit arena. | Render mode: human. | Exit the arena as soon as possible. | Colliding with seeker ends the episode, as truncated. No visual errors. | Pass. |
| Win. | Render mode: human, number of seekers: 0. | Survive until end of epoch | Epoch ends after time expires, as terminated. No visual errors. | Pass. |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Test | Environment Config | Procedure | Expected Result | Actual Result |
| Controls. | Render mode: human. | Place a symbol in each cell, restarting the game between moves. | Control mapping is correct, all symbols placed in correct cells. No visual errors. | Pass. |
| Win. | Render mode: human, opponent epsilon: 1. | Play the full match and win. | Match ends in truncation, with correct reward. No visual errors. | Pass. |
| Draw. | Render mode: human. | Play the full match and draw. | Match ends in termination, with correct reward. No visual errors. | Pass. |
| Loss. | Render mode: human. | Play the full match and lose. | Match ends in truncation, with correct reward. No visual errors. | Pass. |

Table 7. TTT Manual Tests.

1. User Guide

The software takes the form of 16 Python files, most of which are intended to be run directly. The project was tested using Python 3.11.3. Ensure that the requirements are installed, they are listed in requirements.txt. For more detail, check the readme. The project contains several components;

* Training Files. These files will automatically train agents for the associated environment. This typically takes a long time (1000 episodes = 30-40 minutes on my machine).
* Plot. This file is capable of re-displaying the data produced by the training files, reproducing the final graph.
* Human Interactive Environments. These files allow the user to play one of the three games directly. The readme contains an explanation of the game mechanics. For an immersive reinforcement learning agent experience, avoid reading these and try to guess them yourself.
* Demo Files. These files run an example agent on one of the environments, with a GUI to observe its actions.
* Baseline files. These were used to produce the baseline values in section 5, and can be used to reproduce them.
* Unit Tests. These are the unit tests described in Appendix B.
* The remaining three files; agents.py, environments.py, observer.py, are not intended to be run directly. They are used by the previously mentioned files.