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

**Abstract**

Deep reinforcement learning is a branch of machine learning, it is one method for producing decision-making neural networks. Where supervised ML 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, each method was applied to a series of computer games. The agents’ scores were documented as they learned to play the games. Finally, those scores were compared to determine how effectively each method learned.

The experimental findings were (in? TODO)/consistent with the existing literature. TODO explain how.

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 was to compare the effectiveness and limitations of several different deep reinforcement learning methods; 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 RL 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, RL was used to develop more efficient matrix multiplication algorithms.

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 under the high gravity of the planet Vulcan. Reinforcement learning gives us the tools to solve this problem.

Reinforcement learning problems are typically framed as decision problems. 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 what output strength to assign to each motor.

In order for reinforcement learning to be applied, the problem must be a Markov decision process. An MDP is a special case of decision process that possesses the Markov property. 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 the future evolution of the process can be predicted using only on the current 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 state and reward , then takes an action . The process then transitions to a new , 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, the manner in which a decision problem transitions between states is often called its “transition model”. A single pass through a decision problem, from the initial state at to a terminal state, is called an “episode”.

## 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. 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 maximize 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 RL methods frequently attempt to do so, making estimates of expected reward using empirical observations. In the literature, 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. This alone is not useful in constructing a policy. However if the transition model of the process is also known, V-values can be used to construct policies i.e. by calculating the expected value of the successor states reached by each action. However, these methods have limited utility as the transition models of real processes are often unknown or difficult 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 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.

**Policy-Based.** The model produces a probability spread of actions for a given state, this can be sampled from to enact a policy. 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 or Q-values can be used, 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. This means that 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. 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. 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, 2] |

Where is the discount rate (a hyperparameter).

The methods differ in that SARSA is on-policy, whereas Q-learning is off-policy. That is, 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 (following a greedy policy with respect to the current approximation of ). Even if these policies are initially identical, they may diverge as weight updates are made.

### Pure Policy-Based Methods

The algorithms detailed in this section are policy-gradient methods. Definitionally, a policy-gradient performs 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 maximize 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. The entropy of a policy with respect to a particular state is given by:

|  |  |
| --- | --- |
|  | [3] |

Where is the function that gives the probability that policy will prescribe in . Note that for a softmax distribution (where ), the function is maximised when is equal for all actions.

#### REINFORCE

This is one of the earliest policy-based methods to be discussed. It is Monte-Carlo; the entire episode is observed, and the sum of rewards received during the episode is used as. The variant used in this report includes an entropy term. At the end of each episode, the network parameters are updated to minimise:

|  |  |  |
| --- | --- | --- |
|  | [4, 3] |  |

Some limitations of this approach; no discounting is applied; it assigns equal credit to all actions.

### Actor-Critic Methods

The key feature of actor-critic methods is their use of two networks; the actor and the critic. The actor learns in the same manner as policy-gradient methods; however the performance measure is calculated using the output of the critic.

The critic learns a value function using the same method as the value-based techniques discussed earlier. Note that as the transitions are being produced according to the policy of the actor, critic learning is necessarily off-policy.

There are multiple ways to produce a performance measure, and these methods are described in more detail in the three implementations below. The actor and critic can be two separate networks, or a single network. In the latter, 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:

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

Where is the critic’s approximation of .

#### Advantage Actor-Critic (A2C)

This n-step algorithm is a synchronous version of Asynchronous Advantage Actor-Critic [6].Advantage refers to the advantage function, which isolates the action’s contribution to the reward from the inherent value of the state.

|  |  |
| --- | --- |
|  | [6] |

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

|  |  |
| --- | --- |
|  | [6] |

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

|  |  |
| --- | --- |
|  | [6] |

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

|  |  |
| --- | --- |
|  | [6] |

#### 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:

|  |  |
| --- | --- |
|  | [7] |

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” [7]. 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.

In actuality this process was not as cleanly delineated, and components of the project were revisited and adapted based on insights gained during the pursuit of other goals.

## Analysis

It is 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 follow 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. It is also important for historical value as the 2017 paper using it marks the start of the current burst of interest in the field.

REINFORCE/Basic Policy Gradient 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 include tuneable parameters that can be used to reduce their difficulty.

A total of three environments were designed, with diverse properties. The intent was to create a variety of environments to 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 implementing an environment with a large image percept for use with a convolutional neural network. However, after discussing the idea with my supervisor I determined that training such a network would be computationally infeasible.

The state representation of each environment is designed to provide sufficient information to predict the optimal action. If this were not the case, no agent will be able to do so, and the final metrics would be dominated by random chance. Each environment includes a random element. This makes the task more difficult, as the agent must 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 is necessary; it allows for testing of the environment, it allows a human baseline score to be collected, 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. reward hacking, 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.

## Requirements

### Functional Requirements

**Agents**.

* User can request an action from the agent.
* User can train the agent’s 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.
* 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

**Agents.**

* Training must be replicable.
* Each agent should be able to improve its expected reward for an environment, given sufficient training experience.
* Each fully trained agent should be able to achieve a typical score of at least 50% of the environment baseline in the default configuration. These baselines are.
  + Maze: 4767 (human average over 20 epochs).
  + Tag: 565 (human average over 20 epochs).
  + Tic-Tac-Toe: 1000 (optimal policy).

**Environments.**

* Good performance.
* 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. As such, the visual presentation of the GUIs was not considered particularly important. Screenshots of the final UI are included on the next page.

## System Design

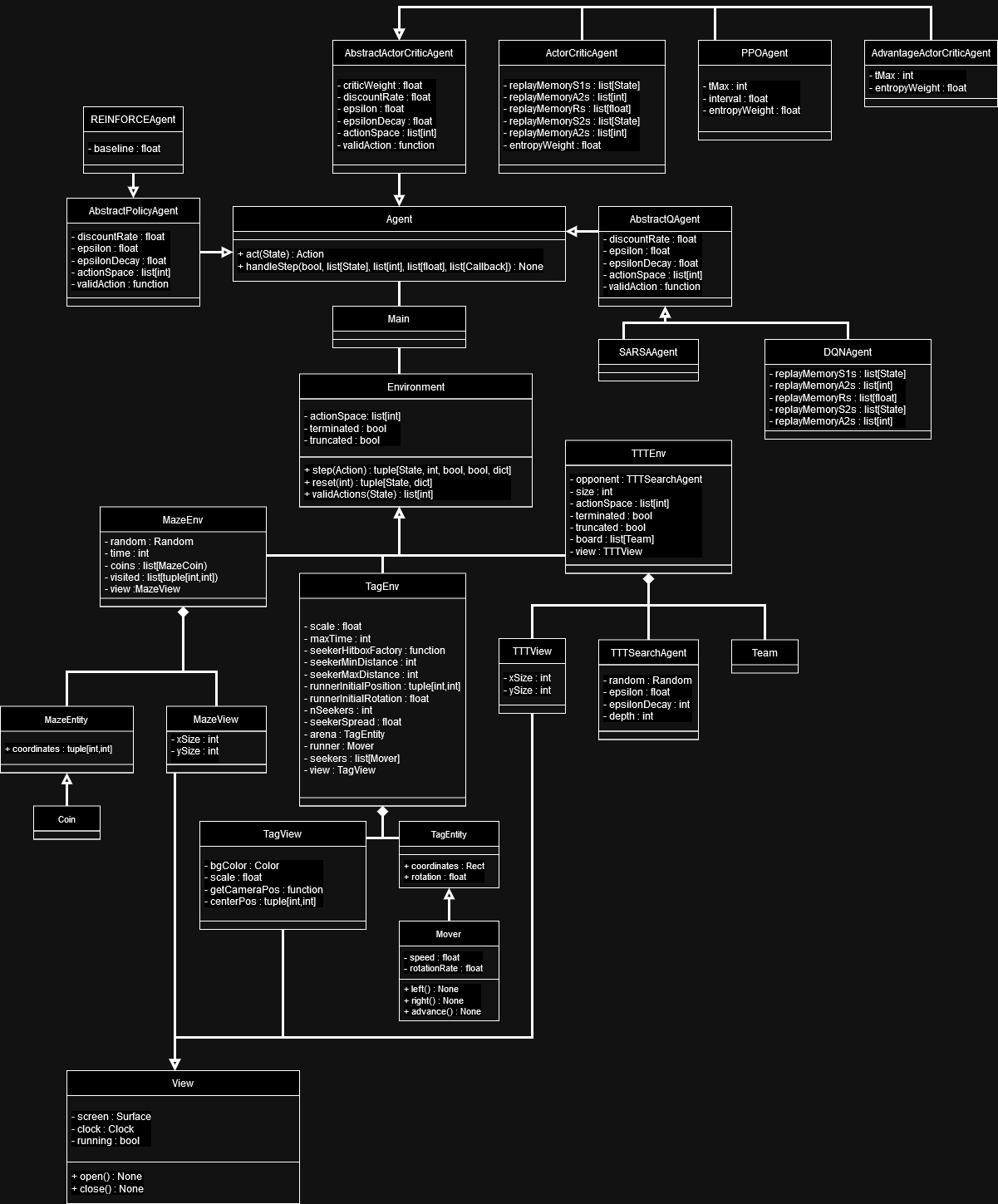


Figure 1. UML diagram of the Python codebase. The observer-observable classes and their relationships were omitted to avoid visual confusion.

### 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.
* +10 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 is placed at the center of the arena, facing in a random direction, and the seekers are placed at a semi-random location nearby. Each seeker has a randomly chosen distance and angle. At each step, the seekers move directly towards the runner. The episode ends when the runner is caught 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).

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.

### 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 acts optimally with 75% chance, otherwise taking a random action.

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; {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.

### Agents

Each agent implements one of the algorithms described in section 2.

# Product

## Implementation

### Agents

The implementation of the agents was by far the more difficult task. The main difficulty was in interpreting the papers describing each algorithm, and the implementations were reworked numerous times due to errors. The use of obscure and confusing notation made the earlier papers [4, 5] particularly difficult.

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. 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 optimizer. However, these classes implement no training logic. Training logic is provided by the various subclasses, each of which implements a single algorithm.

The initial struggle to interpreting the algorithms made it impossible to construct a class hierarchy at the outset of the project. 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. The tasks delegated to PyGame were by far the most difficult part of the implementation, so this choice hastened the development considerably.

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. PyGame was selected to handle the implementation of complex rendering logic for the environments, and was also used for the collision detection in the Tag environment.

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 for use with a convolutional neural network.

#### Tag

#### 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.

### Training 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.

## 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.

Due to the simplicity of the environments, testing was performed in an ad-hoc manner during development. 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.

[screenshot of passing unit tests]

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 & screenshots of manual testing process]

# Results & Evaluation

### Evaluation Process

Within each trial, the network architecture was the same for all agents.

[diagrams of network architectures]

[table of maze env config]

[table of maze model parameters]

[table of tag env config]

[table of tag model parameters]

[table of TTT config]

[table of TTT model parameters]

## Results of Evaluation

### Software Component.

The software fulfils the requirements stated in section 3.



Figure 2. Maze environment final UI.



Figure 3. Tag environment final UI.



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

## Returning to the Research Questions

### Were the algorithms able to meet the performance criteria?

### Analysis of Strength & Weaknesses.

# Discussion & Reflection

## Interpreting the Results

Look, it goes up! DQN did really well. This is a discrete environment where it can’t revisit states so that’s unsurprising.

**Maze.** DQN will prob suck because it can control the state distribution.

## Reflection

## Challenges

## Future Work

## Limitations

* Software has some issues.
  + Excessive memory usage. Further attention should have been given to performance, and research into reinforcement learning optimisation strategies should have been performed.
  + Poor testing (was not able to research the SOTA/industry standard testing techniques).

# Conclusion

References

|  |  |
| --- | --- |
| [1] | R. S. Sutton and A. G. Barto, Reinforcement Learning: An Introduction, Second Edition, MIT Press, 2018. |
| [2] | 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. |
| [3] | R. J. Williams and J. Peng, “Function Optimization Using Connectionist Reinforcement Learning Algorithms,” *Connection Science,* no. 3, pp. 241-268, 1991. |
| [4] | R. J. Williams, “Simple Statistical Gradient-Following Algorithms for Connectionist Reinforcement Learning,” *Machine Learning,* vol. 8, p. 229–256, 1992. |
| [5] | V. Konda and J. Tsitsiklis, “Actor-Critic Algorithms,” *Advances in neural information processing systems,* vol. 12, 1999. |
| [6] | 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. |
| [7] | J. Schulman, F. Wolski, P. Dhariwal, A. Radford and O. Klimov, “Proximal Policy Optimization Algorithms,” *arXiv:1707.06347 [cs.LG].* |

1. Appendix