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

by

John Gregg-Webster

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

in Computer Science

CS408: Individual Project



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.

The goal of this project was to compare the effectiveness of several different reinforcement learning methods; [TODO list].

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

# Introduction

In the context of this report, a model is a system that creates an input-output mapping. To learn, models must have parameters, some set of modifiable features that determine the model’s behaviour. In practice, machine learning models are usually neural networks, and their parameters are the network’s weights. 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.

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.

The aim of the project was to investigate the performance and limitations of reinforcement learning methods.

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.

A final set of six algorithms were selected for comparison. They are; Basic Policy Gradient, Policy Gradient with Entropy, Deep Q-Learning, SARSA, Actor-Critic, Advantage Actor Critic, PPO.

The metrics on which they would be compared would be the rewards attained by the resulting models during training, as well as the rewards achieved by the final models over TODO epochs.

The next objective was to produce a software implementation of each method, and virtual environments for the methods to be applied to. First, common interfaces for agents, and for environments, were constructed. This made it possible to proceed with the development of agents and environments in an arbitrary order. The design and implementation of environments was performed concurrently with the research & implementation of methods. The three environments are Tag, Tic-Tac-Toe, Maze. They are described in more detail in section 3.3.

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

# Background Literature

TODO describe a decision problem in natural language.

It is assumed that the reader is already familiar with basic machine learning concepts; neural networks & gradient descent.

Reinforcement learning problems are typically viewed as Markov decision processes, these are a special case of decision processes that possess 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.

There exists a set of states , a set of terminal states 𝕋 (where ), and a set of actions. Time progresses in discrete steps. At each time-step t, 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 an MDP transitions between states is often called the “transition model”.

The problem of training an agent with the best possible performance on a control problem can be framed as finding a policy that maximises the accumulated reward as t → ∞

Stationary problems VS non-stationary problems (harder).

A non-stationary problem is one in which the transition model changes as the process progresses (significantly harder).

**V-Value.**

This function gives the expected sum of discounted reward after the state , under the policy . Sutton & Barto define the V-value as:

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

**Q-Value.**

This function gives the expected sum of discounted reward after taking action in state , then following the policy afterwards. Sutton & Barto defines the Q-value as:

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

**Advantage.**

The advantage of an action isolates the action’s contribution to the reward from the inherent value of the state. Sutton & Barto defines the Advantage function as:

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

**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 produce the action probabilities that map to the optimal policy.

**V-Value-Based.** The model learns to estimate the V-Values of states. This alone is not useful in constructing a policy. However if the transition model of the process is also known, state-values can be used to construct policies i.e. by calculating the weighted average value of the successor states reached by each action. 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 state-value-based approach, this can be used to construct a policy even if the transition model is not known.

**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 approximate the advantage function, and the approximation is used to direct the actor during the learning process. Only the actor’s output is necessary to enact the learned policy.

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 epoch), no learning occurs during the epoch. Once the epoch is over, the model parameters are updated according to the total reward received during the epoch. This means that Monte-Carlo methods can only be applied to problems with epochs 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 epoch, which can quicken convergence.

**N-Step.**

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

**~~Dynamic Programming.~~**

~~In contrast to the three prior strategies, in dynamic programming the agent does not interact with the environment. Instead data is gathered by iterating the state space (in an arbitrary order), using the transition model to explore all possible successor states, and updating the model parameters according to the reward obtained. DP requires a finite state and action space, as well as knowledge of the transition model, so it cannot be applied to many real-world problems.~~

There are a number of non-deep approaches; tabular methods, linear regression.

Tabular approaches achieve the desired input-output mapping by constructing a look-up table containing every possible input state, which is held in memory. Tabular methods are backed by theoretical proofs of their ability to converge to optimal policies [3, 4]. Despite this useful property, such methods are not typically practical due to the large state-spaces of real-world machine-learning problems. The associated table would be so large that storing it and updating it during training would be computationally infeasible.

Modern approaches typically use neural networks to approximate TODO. While perceptrons have been proven to converge to the optimal policy over time [5], neural networks in general are not guaranteed to converge, or even to improve. Though in practice they often do.

Methods based on gradient descent can find only the local optimum.

The use of neural networks with hidden layers to solve reinforcement learning problems is known as deep reinforcement learning, and in the last decade it has been applied to achieve human-level performance in complex control problems [6, 7].

Modern RL encompasses a diverse variety of approaches, and there are a variety of overlapping categories by which reinforcement learning algorithms can be classified. The taxonomy of those approaches is of interest if they are to be meaningfully compared.

*Table 1: Overview of Targeted RL Methods*

|  |  |
| --- | --- |
| Class | Method |
| Q-Value | Deep Q-Learning. |
| SARSA |
| Pure Policy-Gradient | REINFORCE |
| REINFORCE-MENT |
| Actor-Critic | Basic Actor-Critic |
| Advantage Actor-Critic (A2C) |
| Proximal Policy Optimisation |

**SARSA & Q-Learning**

These are closely related temporal-difference methods.

A transition is a tuple

The network outputs an estimate of If this prediction is accurate, it should be exactly equal to the value of the successor state, plus the observed reward at t. We have access to this observed reward, as well as the network’s prediction of the value of the successor state We can then calculate the squared error and use it as the target of gradient descent.

**Q-Learning Loss Function**

|  |  |
| --- | --- |
|  | [2] |

**SARSA Loss Function**

|  |  |
| --- | --- |
|  |  |

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 ). Even if these policies are initially identical, they will diverge as weight updates are made.

**Entropy**

Many of the remaining algorithms take into account the entropy of the agent’s policy. The entropy of a policy captures the variance in the actions it prescribes. The entropy of a policy with respect to a particular state is given by:

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

Where is the action space, and is the function that gives the probability of the network taking action in state . The policy entropy can be added to the reward at each step of the training process to produce a new target for optimization. Doing so can encourage agents to adopt more complex behaviour, by preventing them from learning locally optimal policies that prescribe the same action in all, or nearly all states.

**Policy Gradient Methods.**

The network’s weights are updated according to a rule.

Where is an approximation of , and is the function that gives the probability of the network taking action in state .

If the advantage 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’s negative, they are updated in the opposite direction.

**REINFORCE.**

This is a policy-based Monte-Carlo method, one of the earliest policy-based methods to be discussed. The entire epoch is observed, and the actual sum of discounted rewards after an action is used to calculate . At the end of each epoch, the network parameters are updated according to the rule:

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

Where and are hyperparameters of the network, is the length of the epoch, and is the cumulative reward received during the epoch, which serves as an estimate of the action advantage.

**REINFORCE-MENT**

This is a variant of REINFORCE, described in [3]. Instead of purely maximising reward, it maximises the sum of reward and policy entropy. It is otherwise identical to REINFORCE.

**Actor-Critic Methods.**

The key feature of actor-critic methods is their use of two networks; the actor & the critic. The actor learns in the same manner as policy-gradient methods, however Â 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 Â, and these methods are described in more detail in the three implementations below. The actor & 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 . The critic’s estimate of the Q-value of the actor’s action is normalised and used as . This algorithm includes an entropy term.

**Advantage Actor-Critic (A2C).**

This n-step algorithm is a synchronous version of Asynchronous Advantage Actor-Critic [5]. The critic learns to approximate , and the estimate is used to calculate Â. Every n steps  
- and at the end of the epoch - weight updates are performed on the new transitions, with Â being calculated according to the equation:

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

Where is the critic’s approximation of , and is the number of steps since the last weight update. This algorithm includes an entropy term.

**PPO**

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

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

Where is a constant such that .

This algorithm includes an entropy term.

# Specification and Design

## Methodology

The original plan was to proceed with development in a cyclical manner, in accordance with Agile principles. 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...

An Agile approach was used because...

I knew that it would be impossible to plan the project in the linear manner dictated by Waterfall approaches, because I began with little knowledge of the topic. To develop that knowledge I would need to experiment with practical implementations of the concepts I was researching. Using Agile sprints allowed me to... It’s common to lack knowledge at the beginning of development, and it’s the reason that Agile was developed in the first place, but the highly technical nature of this project made the issue especially severe. I was not just unfamiliar with the details of this particular case, but with the techniques that I would be applying.

## 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.

These particular algorithms were chosen because...

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 in all cases.

Likewise, REINFORCE-MENT is an example of a simple algorithm that uses entropy. According to the original paper, the entropy term should improve its performance in tasks that require multi-step planning. [TODO PARAPHRASING WARNING, IIRC the paper says this almost verbatim]. This should be noticeable and an interesting point of comparison in the analysis phase.

DQN is important for historical value as the 2017 paper using it marks the start of the current burst of interest in the field. The claimed superior performance in discrete environments ought to make for an interesting comparison.

SARSA was attractive as a target due to its similarity to Q-learning. This meant it would be easy to implement by modifying a Q-learning implementation.

Actor-Critic: The actor-critic family of algorithms [is significant to the field, TODO, clarify], it felt important to include it for this reason. [Discuss some feature of the algorithm itself that makes it suitable].

Advantage Actor-Critic was chosen because...

PPO was chosen because it is a more recent algorithm. I was concerned with the lack of representation of contemporary algorithms, as several originate from early research into the field. There was a choice between TRPO & PPO, and PPO was chosen as the paper introducing claims it is simpler, and the paper was less mathematically dense. There were time constraints at play.

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 input 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 cast, 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.

**Tag Environment.**

* Player can take actions.
  + Turn Left.
  + Continue Straight.
  + Turn Right.
* User can configure environment.
  + Number of time steps until epoch 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.

**Maze Environment.**

* Player can take actions.
* Up.
* Down.
* Left.
* Right.
* User can configure environment.
  + Maze configuration.
  + Number of coins.
  + Initial placement of agent avatar (set, or random).
  + Time limit increase per coins.
* User can reset environment.

**Training and Testing Suite.**

* User can train model.
* User can evaluate model (no learning).

### Non-Functional Requirements

**Environments.**

* Good performance.
* Human-readable interface.
* Configurable difficulty.
* The problem posed by the environment is Markov.

**Training and Testing Suite.**

* Training must be replicable.
* Data should be saved periodically, in case of unexpected errors.

## Design

### Interface Design

The focus of the project was on the AI agent, 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.



Figure 1: Maze environment final UI.

Figure 2: Tag environment final UI.

Figure 3: Tic-Tac-Toe environment final UI.

## 3.4.2 System Design

**Agents**

Each agent class corresponds to a training algorithm. Each agent class is intended to be as generic as possible.

AbstractQAgent, AbstractPolicyAgent, and AbstractActorCriticAgent contain logic for deciding actions, but not for training. Training logic is handled by their subclasses, each of which implements one of the targeted algorithms TODO [list them again here | chosen from the list discussed earlier].

AbstractQAgent and AbstractPolicyAgent handle layer initialisation, but AbstractActorCriticAgent does not. This is because the Actor-Critic implementations use a single network for both the actor and critic. The critic architecture differs between Actor-Critic and A2C, so there is no shared initialisation logic to move up to a superclass.

**Environments**

Environments conform to the OpenAI gym environment interface. Initially, one of these environments (cartpole) was used to test agent implementations before the three environments were created. The interface was retained due to the existing code, and for cross-compatibility.

**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 epoch 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 epoch ends when the runner is caught or moves off the edge of the game area. The epoch 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 follows an ε-Greedy policy. This means that it takes a random action ε% of the time (where ε is a configurable parameter of the environment), and the rest of the time it takes the optimal 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.

**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. There is a counter that decays by 1 each step. The counter increases each time the agent collects a coin. The epoch ends when the counter reaches 0, or after a set number of steps.

The percept is the contents of each square; {Empty, Wall, Agent, Coin}.

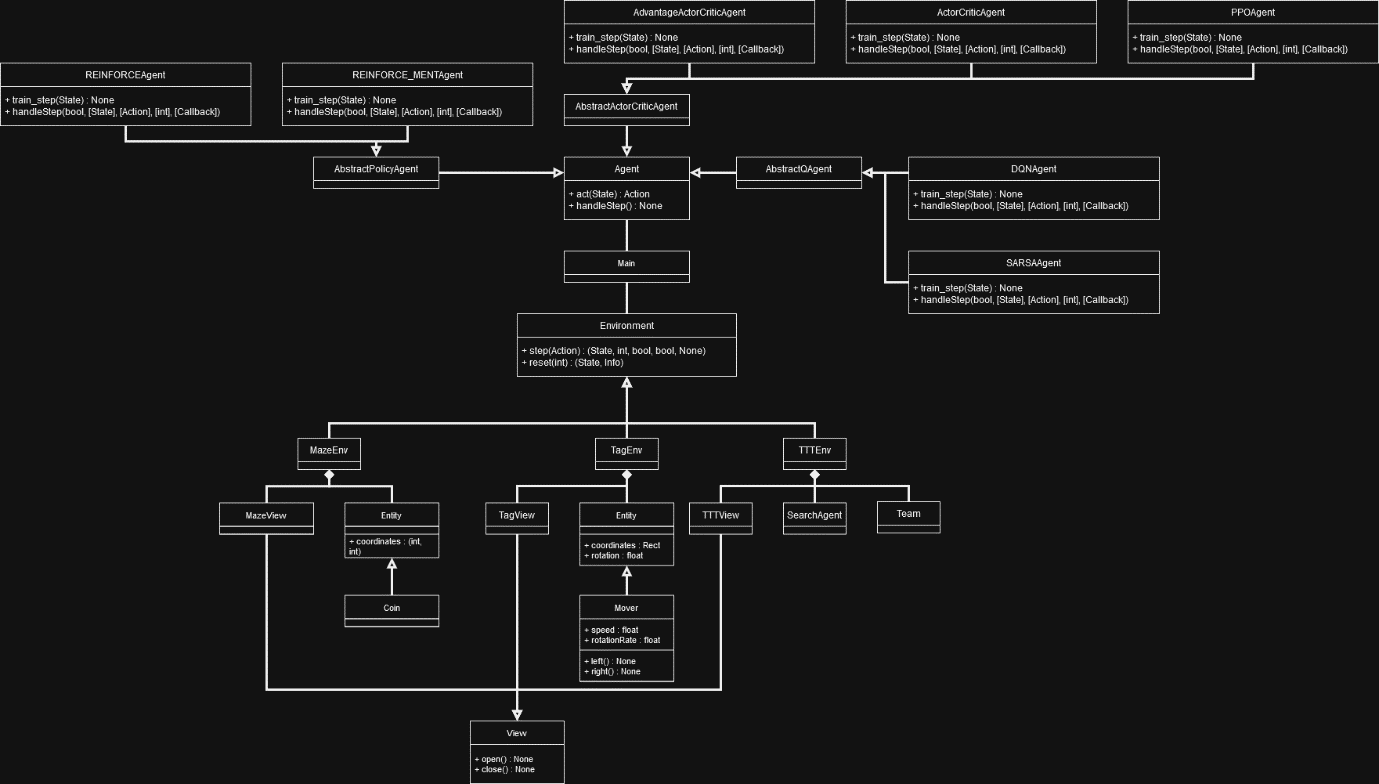
**Reward scheme.**

* +1 for collecting a coin.

**Training/Testing Suite**

A software component is necessary to mediate the interaction between agent and its environment. This is responsible for controlling the duration of the interaction, determining when policy updates are performed, and controlling the data needed for policy updates.

[details of training]. The rewards & model weights are periodically written to disk. Model weights are stored in .tf format. Metrics are stored in .npy format.



# Product

## Implementation

TensorFlow was used to handle the implementation of neural networks and automatic differentiation, and was chosen because of the developer’s personal familiarity with the library. This choice determined the language used for the rest of the project. PyGame was selected to handle the implementation of complex rendering and collision detection 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, and initialisation of the neural network and its optimizer. However they implement no training logic. The training logic is provided by the various subclasses, each of which implements a single algorithm.

The Env class represents the environment that the agent interacts with. It is implemented using the MVC pattern. It is responsible for providing the external interface. The view is responsible for handling all logic related to the human-readable GUI, and is only created if requested by the user. The Env and View are not necessary for the game logic, as that is the responsibility of the Model.

The observer pattern is used to update the view each time the model changes. The view is the observer, and extends the Observer class, the model is the observable and extends Observable.

4.2 Verification & Validation

# Results & Evaluation

**5.1 Evaluation Process**

### 5.2 Results of Evaluation

### 5.3 Returning to the Research Questions

### 5.3.1 RQ1

It is clear from our findings that James Bond was born in Wigtown in Scotland. However, he grew up in Diss, in Norfolk. We know this because ....

### 5.3.2 RQ2

We were not able to answer this question from our studies, although some suggestions were made. These could not be proven.

### 5.3.3 Objective 1

### 5.3.4 Objective 2

# Discussion & Reflection

## Interpreting the Results

## Reflection

## Challenges

## Limitations

* Software has some issues.
  + Poor performance (not time for optimisation).
  + Data kept in memory unnecessarily.
  + Poor testing (was not able to research the SOTA/industry standard testing techniques).

## Future Work

# 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 preprint arXiv:1312.5602,* 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] | 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] | J. Schulman, F. Wolski, P. Dhariwal, A. Radford and O. Klimov, “Proximal Policy Optimization Algorithms,” *arXiv:1707.06347 [cs.LG].* |
| [7] | 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. |

1. Appendix

This is where you can include your documentation.

Remember that the marker is not required to read this, but might well check to ensure that you have included product documentation, and ethical approval, as required.

* 1. Ethical Approval Form

If your project required you to do any evaluation with humans, you MUST include this. It can be downloaded from the Ethics system.

https://local.cis.strath.ac.uk/wp/extras/ethics/index.php

* 1. Participant Information Sheet

If your project required you to do any evaluation with humans, you MUST include this

https://www.strath.ac.uk/ethics/informationsheetandconsentform/

* 1. Consent Form

If your project required you to do any evaluation with humans, you MUST include this.

<https://www.strath.ac.uk/ethics/informationsheetandconsentform/>