Final Report

Hierarchical Deep Reinforcement Learning in Super Mario

660050748

April 29th, 2020

Abstract

I certify that all material in this dissertation which is not my own work has been identified.

Contents

1 Introduction

Reinforcement Learning is a field of Machine Learning where autonomous agents learn how to act optimally and maximise a reward within a given environment. Throughout the last decade, the field has achieved remarkable successes and gained significant medial and research attention. Algorithms have been shown to outperform the best human players on a wide range of Atari games [1], chess, Shogi and the strategy board game Go [2], and even Starcraft [3]. Super Mario as a domain is not as popular as the Atari games amongst researchers, making it an interesting choice for testing Reinforcement Learning algorithms. The goal of this project is to build a system to investigate, test, evaluate and compare a number of Reinforcement Learning algorithms on the domain of NES Super Mario.

1.1 Motivation

Reinforcement Learning algorithms come in many flavours, however most do so called *tabula-rasa* (blank-slate) learning. Humans are experts at reusing prior knowledge in new situations, e.g. if you see a ladder in a video game you have never played before, you know that you can climb that ladder because you know what a ladder is. A *tabula-rasa* agent on the other hand will first have to learn concepts a human player already brings to the table. Moreover, even if the agent learns to climb the ladder, it has no concept thereof and will have to relearn it in a new situation. For this reason, contemporary algorithms that play at a superhuman level require absurd amounts of training episodes on powerful hardware. DeepMind's Rainbow architecture [4] attains superhuman performance after experiencing 18 million frames, corresponding to 83 hours of Atari gameplay. Distributional RL [5] surpasses human players after 70 million frames in Atari games. If agents were able to abstract away behaviour when learning and reuse already learned information when faced with new situations, the training time could be reduced. Additionally, the agent would be more capable of generalisation.

Introducing the abstraction of behaviour is the research area of Hierarchical Reinforcement Learning. Instead of treating the solution to a Super Mario level purely as a sequence of primitive actions, e.g. pressing left, right, jump etc, Hierarchical Reinforcement Learning breaks down the solution into more abstract learnable chunks. These chunks operate at a higher *temporal abstraction* and act as subgoals for an agent to achieve in the game. Intuitively, this is how a human player would approach the game as well. They would most likely not think in terms of button combinations however, but on a higher level.To finish a level, Mario has to jump ravines, kill Goombas, avoid Fire Bars and climb steps - all of which are further decomposable into arrangements of primitive left, right, jump actions. Recent research [6] has shown that using an Hierarchical Approach to learning, results in up to 3 - 5 times fewer interactions with the environment. Thus, Hierarchical Reinforcement Learning comes with potential to alleviate the problem of long training times and generalisation.

1.2 Objectives

The main objective of this project was to compare

2 Theoretical Background

The following section will provide the theoretical underbody for the different Reinforcement Learning approaches used in this project.

2.1 Deep Reinforcement Learning

Deep Reinforcement Learning combines neural networks with the concepts of traditional Reinforcement Learning and is the contemporary go-to approach. This section will outline the basic concepts of Reinforcement Learning and explain the improvements made by neural networks. The environment in an RL-problem is typically modelled as a Markov-Decision-Process (MDP). It is a discrete stochastic sequential decision process [7] formalised as a 5-part tuple $\langle S, A, T, R, \gamma \rangle$ where:

- S is the state space, i.e. the set of all possible states. For Mario, each pixel configuration is a state.
- A is the action space, i.e. the set of all possible actions. For Mario, these are the different left, right, jump etc. actions.
- $T : S \times A \times S \to [0, 1]$ is a transition function. It gives conditional probabilities for state traversal.
- $R : S \times A \times S \to \mathbb{R}$ is a reward function. It gives a real number for state traversal. The reward function used in this project is described in section 4.2.
- γ is a discount factor to prevent infinite rewards.

At a given point in time t, the agent performs an action a_t in state s_t and observes a reward r_t and a new state s_{t+1}. Actions only last a single timestep, e.g. it is not possible to take action a_t and move from state s_t to state s_{t+5}. Moving from state s_t to state s_{t+1} is noisy and happens with probability $P(s_t, a_t|s_{t+1})$ as defined by the transition function T. While deterministic MDPs exist where the transition function always returns 1, they are neither applicable to real-world scenarios nor to this project. Which action to perform in a given state is dictated by a policy π. An optimal policy π^* will always prescribe the action which gives the highest expected return of reward for every state in the state space.

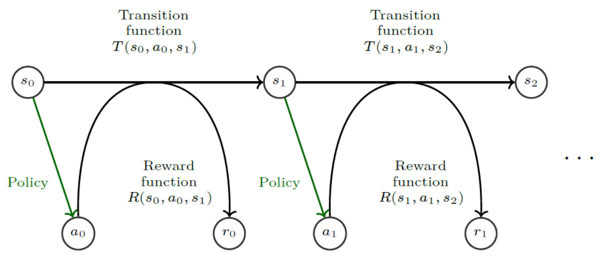

Transition function $T(s_0, a_0, s_1)$
Transition function $T(s_1, a_1, s_2)$
s_0
s_1
s_2
Policy
Reward function $R(s_0, a_0, s_1)$
Policy
Reward function $R(s_1, a_1, s_2)$
...
a_0
r_0
a_1
r_1

Figure 1: Agent takes an action to move to the next state and observe a reward [8]

One of the most popular algorithms for learning optimal policies remains Q-learning [9]. Each state in the state space has state-action pairs called Q-values. The Q-value $Q(s, a)$ describes the rewards an agent can expect from taking taking action a in state s. For instance, if Mario encounters a Goomba in his current state, the Q-value for taking an action running into the Goomba would be lower than the Q-value for taking an action killing the Goomba. The Q-value under a policy π is formalised as the expected sum of discounted rewards **given** state s and action a:

$$Q^\pi(s, a) = \mathbb{E}\left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \middle| S_t = s, A_t = a\right] \tag{1}$$

An optimal policy is therefore one which for each state chooses the action associated with the largest Q-value. To learn an optimal policy, an agent needs to learn the Q function: $Q : S \times A \to \mathbb{R}$ of the problem. Learning the Q-function is a process of continuously experiencing the environment and using the observations to update the Q-values until they converge. The following shows the Q-value update:

$$Q(s, a) \leftarrow Q(s, a) + \alpha \left[R + \gamma \max_{a'} Q(S', a') - Q(s, a)\right] \tag{2}$$

Whenever an agent takes an action in a state and experiences a reward, the old Q-value needs to be updated to reflect that observation. This is done by adding the experienced reward R for taking action a in state s to the discounted largest Q-value of the next state $Q(S', a')$ (γ is the discount factor), subtracting the old Q-value, multiplying everything by the learning rate α and adding it to the old Q-value. The learning rate specifies the importance of new information *vis-à-vis* old information. In traditional RL, the Q-function is a lookup table where each entry corresponds to a Q-value. Q-tables quickly become infeasible for complex problems however. For instance, assuming an image size of 84×84, the RGB Super Mario state space has a size of $(84 \times 84 \times 3)^{256}$. The solution is to replace the lookup table with a neural network - Deep Q Network (DQN) or policy network. This works because neural networks in essence are universal function approximators. Especially Convolutional Neural Networks (CNN) have made it possible to identify features in images and map them to outputs, thus reducing the need for a large memory footprint and allowing the application of Reinforcement Learning to high-dimensional problems like video games. A video game Deep Reinforcement Learning system usually has a *policy network*. The policy network is a CNN that receives a game frame as input and outputs an action to be taken in the game.

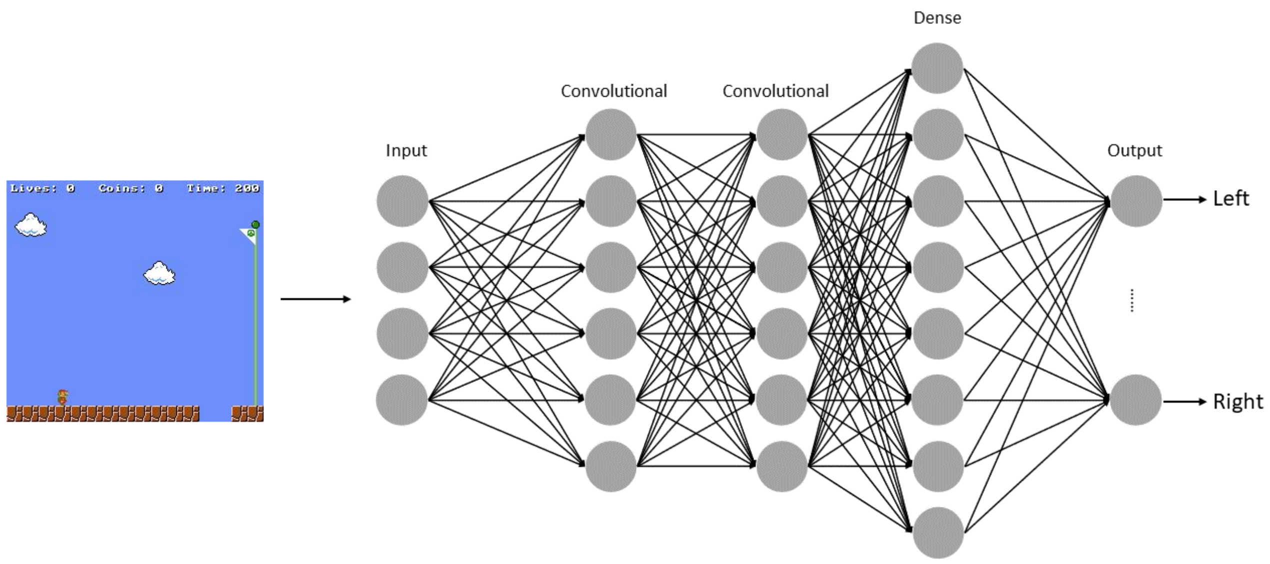


Figure 2: Policy Network for Super Mario (not to scale)

The input first passes through a couple of convolutional layers designed to detect shapes and features such as edges and high level objects. Then follow two fully connected layers to produce the output. For Super Mario, the state is represented as a pixel matrix of dimensions $(width \times height \times channels)$ where channels is the number of colour channels in the frame - 3 for RGB, 1 for grayscale. Given a state, the CNN performs a forward pass and assigns ranks to the possible actions Mario can take, e.g. left, right, jump etc. The network has an output neuron for each possible action Mario can take in the game. Ranks in this case are the Q-values of that state and so the best action is the output neuron with the highest Q-value. Now all that remains is to train the network so that we can learn the optimal Q-values of the MDP. The principle remains the same as in the Q-update equation 2 with the difference that the Q function is a CNN instead of a lookup-table. First, the loss of the network is calculated using a loss function, which is then differentiated with respect to the weights of the network before taking a gradient step. Ultimate goal of the learning process is for the CNN to output Q-values to approximate the optimal Q-values of the MDP.

2.1.1 Target Networks

The improvement made by replacing the Q-table with a neural network is significant, however an issue arises when calculating the network loss in the Q-value update (see **??**). The loss is computed as the difference between the targeted Q-values of the next state, i.e. $Q(s', a')$, and the Q-values outputted by the network for the current state, i.e. $Q(s, a)$. Goal is for $Q(s, a)$ to approximate $Q(s', a')$, though because the policy network is used to calculate both values, any weight updates to the network will shift both $Q(s, a)$ and $Q(s', a')$. At each iteration, $Q(s, a)$ will move closer to the targeted Q-value., However the targeted $Q(s', a')$ will move as well and we end up chasing a moving target. Ultimately, this causes overestimation of the Q-values and leads to learning instabilities resulting in poor policies. The solution is to use a separate target network with the frozen weights of the policy network to calculate the target Q-values. Therefore, target Q-values remain stationary when being approximated and the network does not overestimate Q-values. Periodically, the policy network's weights are copied to the target network so that they accurately reflect the learning process.

2.1.2 Experience Replay

In traditional Reinforcement Learning, observations of the environment are immediately discarded after the Q-values have been updated. Experiencing the environment as a consecutive stream of temporally sequential observations becomes problematic because samples are highly correlated and training on correlated data may get you stuck in a local minimum. Furthermore, because observations are not kept, each one of them can only be used in a single weight update which makes learning less efficient. Therefore, most Deep Q Learning algorithms attempt to break correlation between samples and reuse them for weight updates with a *replay memory* [10]. The replay memory is a buffer of size N where the agent stores experience tuples $e_t = (s_t, a_t, r_{t+1}, s_{t+1})$. Until the agent has collected a sufficient number of samples, it will execute random actions. Once the memory has been filled with enough samples, the agent randomly samples a batch and uses it to train the policy network.

2.2 Hierarchical Reinforcement Learning

Arguably, one of humans' most remarkable abilities is the ability to generalise knowledge [11] across various contexts withstanding input distribution changes. For example, you are able to recognise a door as a door and open it regardless of its colour or material because you abstract away the features which make up the door and generalise it to other doors. While Deep Reinforcement Learning partially alleviates the issue of generalisation [12], it remains a substantial issue [13] to date. Reinforcement Learning agents are prone to overspecialise and overfit because they are trained and tested within the same environment [14]. If Mario is trained to solve level *World 1-1*, he only becomes an expert on that particular level. When placed in the next level, he would not perform as well because Mario did not **learn** any concepts of the world. Instead, he **remembers** optimal action sequences only applicable to a distinct problem. This is also known as *flat* Reinforcement Learning.

Learning reusable skills is the essence of Hierarchical Reinforcement Learning. It is inspired by the fact that human decision making is *temporally abstract* [16] and hierarchically composed. When leaving work to go home, you need to exit the office, however first you need to get up from your chair and take the lift downstairs. The parent task of going home contains multiple subtasks which themselves involve subtasks of their own. By decomposing the overall problem into smaller subproblems, learning is sped up and actions can be reused across the domain. This requires some modification to the MDP used in tradiational Reinforcement Learning. Remember that in *flat* learning, each action a_t only lasts a single timestep. Now, the agent can execute actions lasting multiple timesteps. An action a_t in state s_t can transport the agent to state s'_{t+k}. The policy π is made up of multiple sub-policies, each of which are catering to a specific subset of the state space. At timestep t_1, the primitive action a is selected and lasts one timestep. At t_2, the policy π invokes a subpolicy or subtask σ which is in charge of selecting primitive actions until t_5. Upon termination, the subpolicy returns a reward (yellow asterisk) back to the master policy which incorporates it into the value of the state from where the policy was invoked. What convolutional neural networks did for vision, figuring out behavioural hierarchies are to RL.

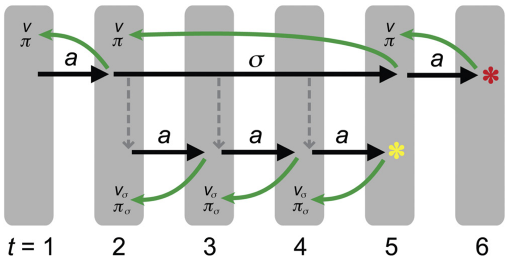


Figure 3: The Hierarchical RL Semi Markov Decision Process [15]

2.2.1 Option-Critic

The following Hierarchical algorithm is an advancement on the Markov-Options framework [16] which first introduced a framework for extending the classical one-timestep-action MDP to a multi-timestep-actions Semi MDP (SMDP). The SMDP contains *Markov Options*, a layer of temporal abstraction on top of the primitive one-timestep actions of the agent. An option is defined as a triplet $\langle I, \pi, \beta \rangle$ where:

- $I \subseteq S$ is the subset of states available to the option.
- $\pi : S \times A \to [0, 1]$ is the policy for the option.

- $\beta : S^+ \rightarrow [0, 1]$ is the termination criterion for the option.

An option can only be chosen by the agent if the current state is in the subset of states available to the option, $s_t \in I$. This restricts options to a specific portion of the state space. For every subsequent state reached in the option, it will terminate with probability $\beta(s_{t+k})$. In Sutton et. al's work, the options/subpolicies had to be handcrafted for the agent to learn and use them. Additionally, learning these subpolicies came at a great temporal and spatial expense because each option is solved as its own MDP. The Option-Critic architecture [17] improves upon this by not only discovering subpolicies autonomously, but also simultaneously learning the policy over options/master policy as well as subpolicies. The number of hierarchies is fixed at two. Furthermore, it assumes that options are available to the agent everywhere in the state space. Just like vanilla Deep Reinforcement Learning, Option-Critic utilises Deep Q Networks to approximate the Q value function.

At each timestep, the master policy π_Ω chooses an option/subpolicy ω_t. We follow the regular MDP structure where the option/subpolicy executes an action a in the environment and receives a reward r and new state s' in return. The critic then evaluates the option by using the Q-value Q_U of executing a subpolicy ω from state s to obtain the new Q update. If the next state is a terminal state of an option, we obtain the update value δ by subtracting the observed reward r from Q_U. Otherwise, we compute the update value like we did for a terminal state and add the discounted Q-value Q_Ω of the entire option as well as the max Q-value of the **next** option. After evaluating, the option is improved using Stochastic Gradient Descent. The option terminates when the criterion β_ω is fulfilled after which the policy over options/master policy chooses the next option.

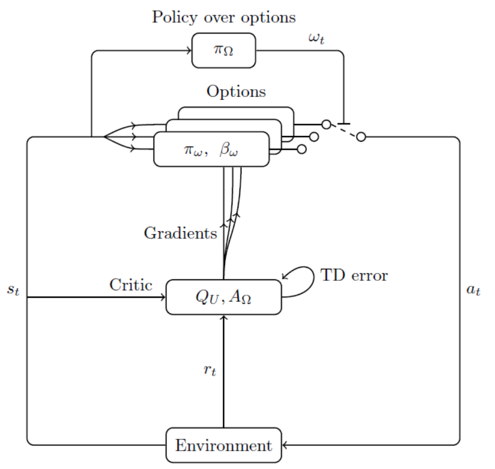
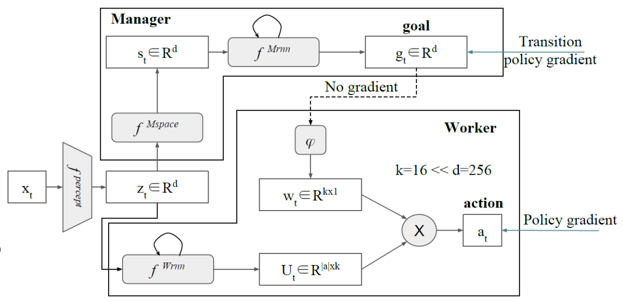


Figure 4: Option-Critic Architecture [15]

2.2.2 Feudal Networks

Another recent Hierarchical Reinforcement Learning algorithm is DeepMind's FeUdal Network (FuN) [18]. Like the Option-Critic architecture, is is an improvement on an older system, called Feudal Reinforcement Learning [19]. Feudal Reinforcement Learning follows the general Hierarchical Reinforcement Learning architecture where different levels of temporal abstraction work together to solve the overall task. Taking after the questionable medieval feudal system, managers set goals for the sub-managers to solve in the state space and receive rewards once completed. Communication is restricted to a single hierarchical step, meaning that each level can only give goals to the sub-manager below and receive goals from the manager above. Additionally, the number of hierarchies are not fixed. Though a great improvement on standard Q-learning, traditional Feudal Reinforcement Learning is not general enough to work on multiple domains and has inherent convergence issues. DeepMind's FeUdal Network provides a fixed two-level manager-worker hierarchy encapsulating multiple neural networks. At the top is the manager, setting goals for the worker in the latent space. Working like a compass, the manager figures out **where** the worker should go. At the bottom, the worker has a high temporal resolution and is responsible for choosing primitive actions in the environment. Given a goal by the manager, the worker decides **how** to achieve it.

FuN has a more complex architecture than the regular DQN and Option-Critic systems. First, the game frame is passed through a CNN modelled after DeepMind's Atari [1] network to compute a separate intermediate representation z_t of the state x_t to be shared between the manager and worker. After further compressing the state z_t into



s_t, the manager utilises a Long short-term memory (LSTM) Recurrent Neural Network (RNN) to compute a goal g_t for the worker to solve. The worker is given the intermediate state representation z_t and uses an LSTM to produce an action embedding matrix U where each row corresponds to a possible action within the game. To incorporate the manager's goal, the worker then embeds it into a vector w_t. By applying a dot product to the action matrix U_t and the goal embedding w_t, we get a probability distribution over the actions, making FuN a stochastic architecture.

3 System Requirements

The following section specifies the requirements to be fulfilled by the system:

- **Super Mario Game Emulator**
  The Reinforcement Learning agent within the system must have the ability to interact with the Super Mario Bros game environment to obtain information about states, rewards and to perform actions. The Python OpenAI Gym library provides a plugin to play Super Mario, however this plugin does not allow the creation of custom levels (more on custom levels in the next point). Instead, the sytem will use the Mario AI Tenth Anniversary Edition [20] framework. Although written in Java, it provides the user with the ability to create their own levels using ASCII art.

- **Custom Levels**
  Using custom levels means that we can better investigate Mario's generalisation abilities. The agent can learn specific skills in specially designed levels and slowly expand their skillset. This is also the main reason for choosing the Mario AI Tenth Anniversary Edition framework over the OpenAI Gym library. OpenAI Gym only provides the original Super Mario levels and therefore makes the skill stacking with Hierarchical Reinforcement Learning algorithms less transparent.

- **Reward Function**
  Every MDP has a reward function. Because the chosen Super Mario framework does not have a reward function, it will first need to be designed. The function should incentivise the intended behaviour of the agent - to move as far right as possible in the level without dying.

- **Java-Python Bridge**
  Because the chosen Super Mario game framework is written in Java and Python is the de facto lingua franca of Reinforcement Learning, the system needs to have a communication bridge between the languages. The system needs to train for longer periods of time, so avoiding unnecessary overhead whereever possible is important. Getting the two languages "as close" as possible to each other is therefore vital. A client-server architecture using HTTP to send JSON is easy to implement, however comes with too much overhead. The Jython [21] and JPype [22] frameworks are directly embedded in the JVM to reduce latency, however they are not as out of the box user friendly. The Py4J [23] framework on the other hand uses sockets to communicate with the Java API which leads to a slight performance decrease. Though it is nearly effortless to setup and easy to use which is why it is the best choice for the system.

- **Deep Learning Libraries**
  Contemporary Reinforcement Learning algorithms rely on Neural Networks to learn behaviour. The system needs to be able to initialise and train neural networks by making use of existing libraries. PyTorch is the most suitable option as it offers a good balance between low-level granularity and easy to use high-level API. Additionally, PyTorch has their own Reinforcement Learning tutorials.

- **Python Game Interface**
  The different Reinforcement Learning agents of this project all need to interact with the Super

Mario Java game emulator. Writing a Python interface that bridges to the emulator and provides standard operations for the agents to use will reduce code duplication on the agents' part and decouple them from the emulator logic.

- **Preprocessing**
  The environment as observed by the Reinforcement Learning agent is a pixel matrix of the current frame of the game. Commonly, frames are cropped and scaled before being passed to the neural network. Cropping removes unnecessary information such as the score and timer counters at the top of the screen. Scaling reduces the complexity of the image and the number of neurons needed in the network. For to reduce latency, cropping and scaling will need to be done on the Java side of the system.

- **Monitor Training**
  Reinforcement Learning algorithms need to train for anything from several hours to several days. Monitoring this process while it is going on is important because bugs can be caught early on and hyperparameters are more easily compared. Tensorboard provides tools for visualising various metrics such as rewards and loss. Graphs are updated live as the training is happening and the data can be downloaded as JSON or CSV for further analysis.

- **Remote Training**
  The entire system needs to be able to run locally as well as remotely in the cloud. Hardware in the cloud is more powerful and can be scaled up and down as needed. Additionally, it is robust. Google Cloud provides APIs specifically designed for Machine Learning and comes with free credits for first time users. Hence it was used for training the Reinforcement Learning models.

4 System Implementation

The following section will describe the system architecture in detail as it was implemented from the requirements section.

4.1 Overall Architecture

The system follows a client-server model where the Reinforcement Learning client uses the Py4J bridge to execute actions on the game emulator server and observe states and rewards. The client first initiates the environment via the game interface and receiving back the start state of the MDP (note that this process is not displayed in the game server diagram). Having chosen an action according to their policy, the agent passes a *tensor* with said action to the game interface which encodes it as a *boolean action vector*, e.g. $\begin{bmatrix} false & true & false & true & false \end{bmatrix}$ for going right. When the action vector is received by the Game Server via Py4J, the action is executed in the environment k times (see section 4.5 for frame skipping). The final frame of the action sequence is cropped and resized (see section 4.4) and the reward is calculated using the environment's reward function (see section 4.2). The reward and the frame are then marshalled back to the client using Py4J and extracted by the Reinforcement Learning Agent using the game interface. At every timestep and after every completed episode, the agent writes the observed reward and the calculated loss into tensorboard. Once training has finished, the models and hyperparameters are saved for demonstration, evaluation and further training. For the

The game server is a *jar* file built using *maven* exposing a port for the client to interact with. It is based on the *marioai* [20] framework which had to be partially rewritten. Various methods and classes for calculating the reward, preprocessing the game frame and exposing the observations of the environment to an outside agent. To allow for remote training, both client and server are running on a 4CPU 16GB Google Cloud virtual machine. Although only relying on CPU power to the run the system is prolonging training times, the expense of adding GPUs was considered too much for this project.

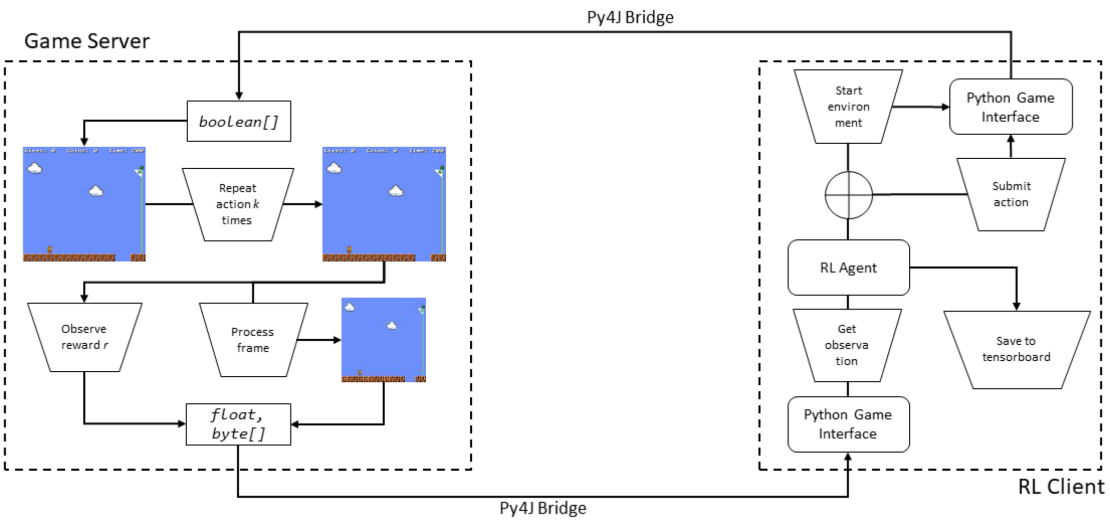


Figure 6: System Architecture

4.2 Reward Function

A Reinforcement Learning agent learns to distinguish between desirable and undesirable behaviour depending on the reward they illicit from the environment. At each time step t, the agent receives a scalar reward $R_t \in \mathbb{R}$ as a result of an action taken at time step $t-1$. The agents goal is to maximise the total reward it receives $\sum_{i=1}^{t} R_i$ and thus solve the task it is given. If we want to the agent to do something, it needs to receive rewards which when maximised achieve the goal we set out. It is important to note that the reward tells the agent *what* it should achieve but not *how* it should be achieved. Designing a reward function improperly might lead to unintended behaviour, also known as the *cobra effect*. During British colonial rule in India, the government offered a reward for dead cobras as a way to reduce their population. Eventually, people began breeding cobras for income to milk the system and the program was scrapped. Breeders then set their snakes free, leading to an increase in the cobra population [24]. This is a pertinent example of how a flawed reward function can produce unwanted behaviour. It is important to remember that in the end, you always get the behaviour you incentivised, not always the behaviour you intended. A well-crafted reward function aligns the **incentivised** behaviour with the **intended** behaviour.

Luckily, simple video games such as Super Mario often have clear criteria for success and failure, meaning that reward function design is of relatively low complexity. We want to incentivise Mario to reach the flag at the end of the level (success) and avoid anything that results in death (failure). Giving Mario positive rewards for moving right in the level and negative rewards for moving left forces Mario to keep moving right. Additionally, providing large negative rewards in terminal failure states and large positive rewards in terminal success states further forces Mario to stay alive and move towards the flag. Positive rewards encourage the agent to keep playing and accumulating those rewards. However, an agent may potentially avoid a terminal state to keep racking up rewards. Therefore, it is important to have positive terminal state values which make the agent choose the terminal state over reward hacking. Negative rewards on the other hand encourage the agent to finish the level as quickly as possible. Besides death, the agent is also penalised each time step t to avoid him standing still. Thus, the agent will want to reach a terminal state as quickly as possible. Rewards are usually bounded to avoid exploding gradients in the neural networks. The reward function used in this project is inspired by the *openai gym* Super Mario Bros framework [25]. It is composed of three variables v, c, d:

- Velocity: $v = x_t - x_{t-1}$. The difference in the agent's x values in between time steps t and $t-1$. Moving right will increase v.
- Clock: $c = c_t - c_{t-1}$. The difference in the game's clock values in between time steps t and $t-1$.

For each action committed in game, c will decrease by 1. Using frame skipping 4.5, each action performed by the Reinforcement Learning agent will yield a c value of -4. This is to prevent the agent from standing still.

- Death/Win:

$$d = \begin{cases} 100 & \text{Mario wins} \\ -100 & \text{Mario dies} \\ -100 & \text{Game times out} \end{cases}$$

  The death and timeout penalty encourage to avoid death and finish the level as quickly as possible. The win reward is to provide an extra incentive to visit the winning terminal state.

- The reward function r is then the summation of the three variables $r(v, c, d) = v + c + d$. Additionally it is clipped in the range of $(-100, 100)$.

4.3 Exploitation vs Exploration

A major challenge within Reinforcement Learning is to strike a balance between gathering enough information about the environment (exploration) and to obtain as much reward as possible (exploitation) [26]. A greedy agent will choose the action which according to their current knowledge returns the largest reward - *exploitation*. Though for an agent to exploit an action they must previously have explored it enough times to have a reliable estimate on the action's reward. Being greedy produces large short term rewards, however the agent might ignore other options with potentially even larger rewards. Performing exploration instead could uncover those options and therefore lead to more reward in the long run. The dilemma arises when at each time step, the agent must decide whether to exploit or explore the environment. Exploring too much makes the agent a slow but steady learner. Exploiting too much means that the agent will quickly converge on their choice of actions but ultimately not act optimally.

A popular policy for choosing between exploitation and exploration is the ϵ-greedy algorithm [27]. At each timestep, the agent chooses the best action to exploit the environment with $1 - \epsilon$ probability. Exploration is more desirable at the beginning of the learning process because the agent does not possess enough information about the environment yet. Hence, the value for ϵ is usually decayed over time to reflect this. Methods for decaying ϵ can be linear [28], exponential [29] and sinusoidal [30]. The algorithms that use an $\epsilon - \textit{Greedy}$ policy in this project use a linear decay.

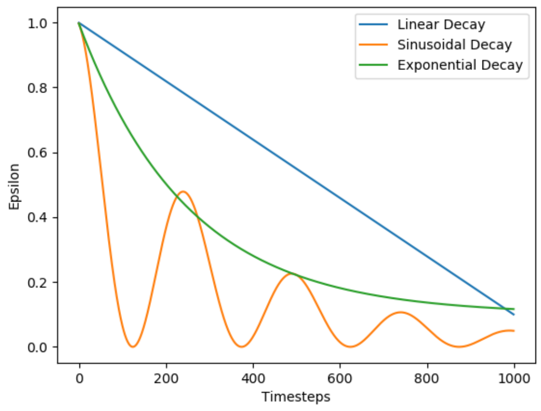


Figure 7: Variants of ϵ Decay

4.4 Frame Preprocessing

Preprocessing data is a step all Machine Learning systems go through prior to starting the actual learning. For a Reinforcement Learning network, the input is a pixel matrix representing a game frame. For Super Mario, the dimensions of the raw image as returned by the game are $240 \times 256 \times 3$. Before the raw image can be passed to the network however, it needs to go through a series of preprocessing steps aimed at reducing its dimensionality and making learning less computationally demanding. First, unnecessary information such as the score, lives, and timer and cropped out because they do not provide relevant clues to the agent. Second, the image is resized to $84 \times 84 \times 3$ using bilinear interpolation [31] to further reduce the size of the neural network needed for training. The dimensions 84×84 are taken from DeepMind's *Playing Atari with Reinforcement Learning* paper [1]. Converting the image to grayscale is another often applied preprocessing step. Though models tend to perform lower on grayscale [32].

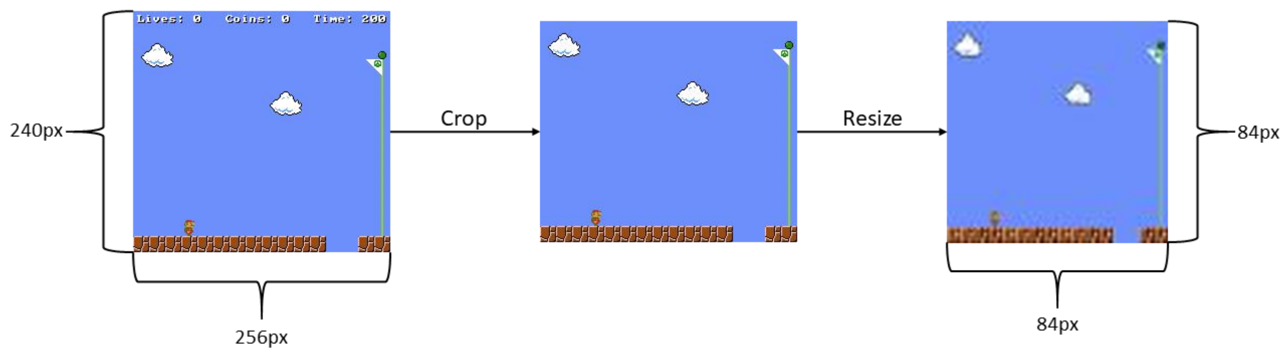


Figure 8: Frame Preprocessing Pipeline

4.5 Frame Stacking and Frame Skipping

When playing a video game, it is important for the player to have a sense of how the objects in the environment are moving. Looking at a single frame of Super Mario does not tell you whether Mario is moving left, right, up or down. Only when presented with consecutive frames could you actually determine Mario's movement. Hence, to give the policy network of our system an idea of where Mario is going, it is fed a stack of 4 frames as input. Using 4 frames was again pioneered by DeepMind [1].

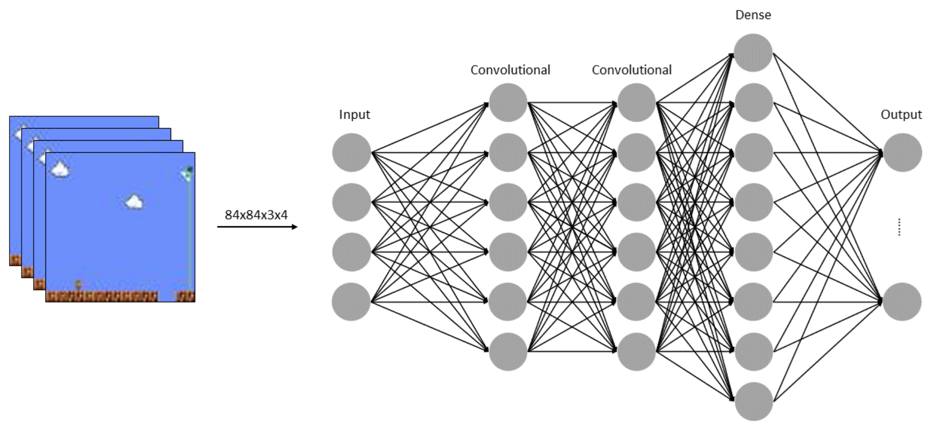


Figure 9: Stacked frames are given as input to the neural network. The dimension of the input thus becomes $84 \times 84 \times 3 \times 4$

.

Another trick DeepMind used in their paper is to only show the agent every k^{th} frame of the game. Instead of taking an action for every frame in the game, the agent now decides on an action every k^{th} frame. The chosen action is then repeated for $k - 1$ frames. This idea is partially motivated by how humans would play the game. A human player makes decisions on how to proceed in the game not for every frame they observe, but at intervals depending on the game environment (some games require faster reactions than others). Using frame-skipping also has the benefit of speedier learning because executing actions in the emulator is faster than making the agent choose an action. Choosing a value for k, one must factor in the desired resolution granularity. A large value for k means that episodes can be played faster, however the agent might miss out on opportunities in the game because the same action is repeated too many times [33]. On the other hand, agents may learn associations between temporally distant states and actions when skipping a large amount of frames. A small k gives the agent finer granularity, though at the cost of longer episodes.

4.6 Custom Levels

Investigating whether agents can learn skills and reuse is best done with custom levels. A custom level represents a skill the agent ought to learn and therefore provides a closed off space where learning can

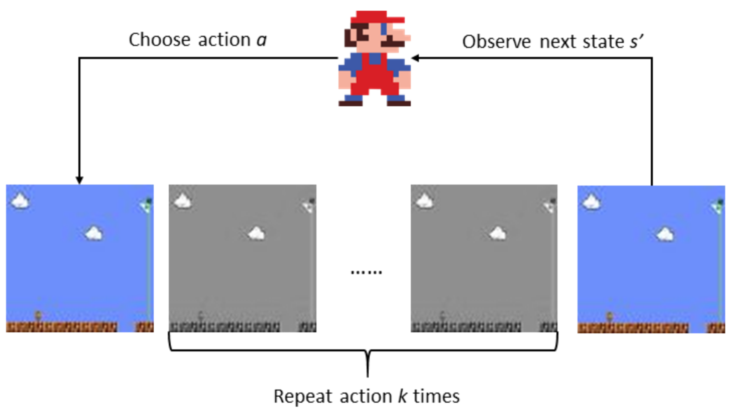


Figure 10: The agent selects action a to be executed k times in the emulator and observes the k^{th} frame

.

take place. The project used three custom built levels - two where agents were meant to learn a basic skill, as well as a third one which is a combination of the two basic levels. Regardless of the skill that agents are supposed to learn, the goal is still to reach the flag at the end. In the system, levels are `.txt` file with ASCII art and can rapidly be created and changed if needed. The three levels created for this project were:

1. **Steps with Gap**
   In this level, the skill to be learned is to jump up the steps and across the gap to reach the flag on the other side.

2. **Two Pipes**
   In this level, the agent needs to learn how to jump over pipes. Though it also involves jumping, overcoming a pipe requires a different set of primitive actions than crossing steps with a gap.

3. **Steps with Gap + Two Pipes**
   A combination of the previous two levels, the agent needs to overcome pipes as well as steps with a gap in the middle before reaching the flag.

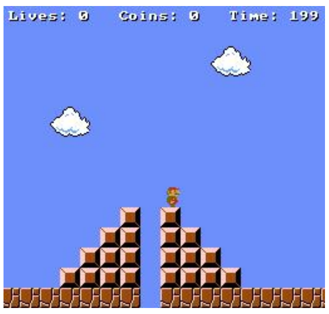


Figure 11: Steps with Gap Level

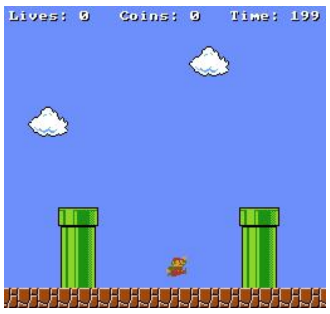


Figure 12: Two Pipes Level

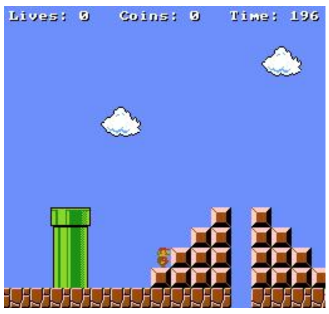


Figure 13: Steps with Gap + Two Pipes

The project is confined to only teaching Mario navigation tasks, hence why none of the levels include any enemies. The reasoning behind this is the state space associated with dynamic and static objects. Pipes, gaps, and steps are static objects that do not move around in the world. A Goomba or Koopa on the other hand is dynamic and forces Mario to learn more states before knowing how to act around it.

5 Experiments and Results

5.1 Experiment Design

The goal of this project is to investigate whether introducing hierarchies and temporal abstraction into Reinforcement Learning speeds up the learning process. A good indicator of a model's learning process is how quickly it is able to accumulate rewards. Models that learn quickly, will have discovered actions which return large rewards sooner than slow learning models. Therefore, the main metric used to compare the different Reinforcement Learning models will be the average reward per episode. To judge whether learning hierarchies is beneficial to learning speed, the same Mario model first learns specific skills in specially designed levels (section 4.6). The model is then placed in a level which requires all the previously learned skills to be solved. An agent which has learned reusable hierarchies should in theory adopt to the new environment faster than a flat-learning agent. Adopting in this sense means to acquire rewards faster. In addition to comparing the different learning agents against each other in the new environment, the agents will also be compared against versions of themselves which have not been trained on previous levels. The purpose of this comparison is to investigate whether training agents *progressively* is beneficial over flat learning.

Determining the stopping criterion, i.e. when the model should stop learning, was grounds to some experimentation. Using the number of episodes played the agent is common, however episodes vary in length across different levels as well as in the same level. This variation means that agents will have had different amount of playtime as better agents would finish sooner and there is no guarantee that agents get the same experience. The goal was to let all agents experience the same number of frames of the game to better predict training times and to use the number of episodes as a metric of success instead. A fast-learning agent should be able to play through more episodes using the same number of timesteps than a slow-learning agent. Thus, the completed number of episodes after t timesteps can also be used to compare the different Reinforcement Learning models. To summarise, an agent which learns quickly should achieve more rewards in fewer episodes.

Model	**Level**	**Progressive**	**Frames**	**Runtime**
DQN	Steps with Gap	False	2×10^5	1d 3h 54m 59s
DQN	Two Pipes	False	2×10^5	1d 5h 31m 54s
DQN	Steps with Gap + Two Pipes	False	2×10^5	1d 5h 46m 34s
DQN	Two Pipes	True	2×10^5	1d 6h 58m 31s
DQN	Steps with Gap + Two Pipes	True	2×10^5	1d 5h 59m 9s
Option Critic	Steps with Gap	False	2×10^5	1d 8h 33m 34s
Option Critic	Two Pipes	False	2×10^5	1d 17h 35m 37s
Option Critic	Steps with Gap + Two Pipes	False	2×10^5	1d 9h 32m 13s
Option Critic	Two Pipes	True	2×10^5	1d 19h 33m 13s
Option Critic	Steps with Gap + Two Pipes	True	2×10^5	1d 13h 7m 29s
FuN	Steps with Gap	False	2×10^5	1d 14h 40m 8s
FuN	Two Pipes	False	2×10^5	1d 13h 58m 52s
FuN	Steps with Gap + Two Pipes	False	2×10^5	1d 12h 13m 18s
FuN	Two Pipes	True	2×10^5	1d 17h 14m 25s
FuN	Steps with Gap + Two Pipes	True	2×10^5	1d 14h 4m 24s

Table 1: Experiments carried out for the project. Progressive indicates whether the model has been pretrained on another level already.

All experiments were run once on a Google Cloud 4CPU 15GB RAM virtual machine. While it is desirable to run experiments multiple times to ensure that the observations do not represent noise, financial limitations and long training times only allowed a single run per experiment.

α	**Batch Size**	γ	**Buffer Size**	**Target Update Freq**	**Learn Freq**	**Learn Start**
0.0001	32	0.99	100,000	200	4	50,000

Table 2: DQN Hyperparameters where:

- α: Learning Rate.
- **Batch Size**: The number of samples used to train the network each training epoch.
- γ: Discount Factor.
- **Buffer Size**: Size of the replay buffer.
- **Target Update Freq**: Frequency in timesteps the weights from the policy net are copied to the target net.
- **Learn Freq**: Frequency in timesteps the network is trained, e.g. every 4th frame.
- **Learn Start**: When the agent should start sampling from the replay memory and train the network.

5.2 DQN Agent

Using a DQN as a comparison baseline is common in the Reinforcement Learning literature [17, 34, 4, 35], thus this project will follow the trend. This section will show how the Deep Reinforcement Learning theory from section 2.1 was used to construct a DQN agent and compare the average reward per episode for progressive and non-progressive models. The DQN agent uses a policy and target network, both of which are CNNs with three convolutional layers followed by two linear layers. The activation function used between the layers is a Rectified Linear Unit (ReLu) [36], which over the past years

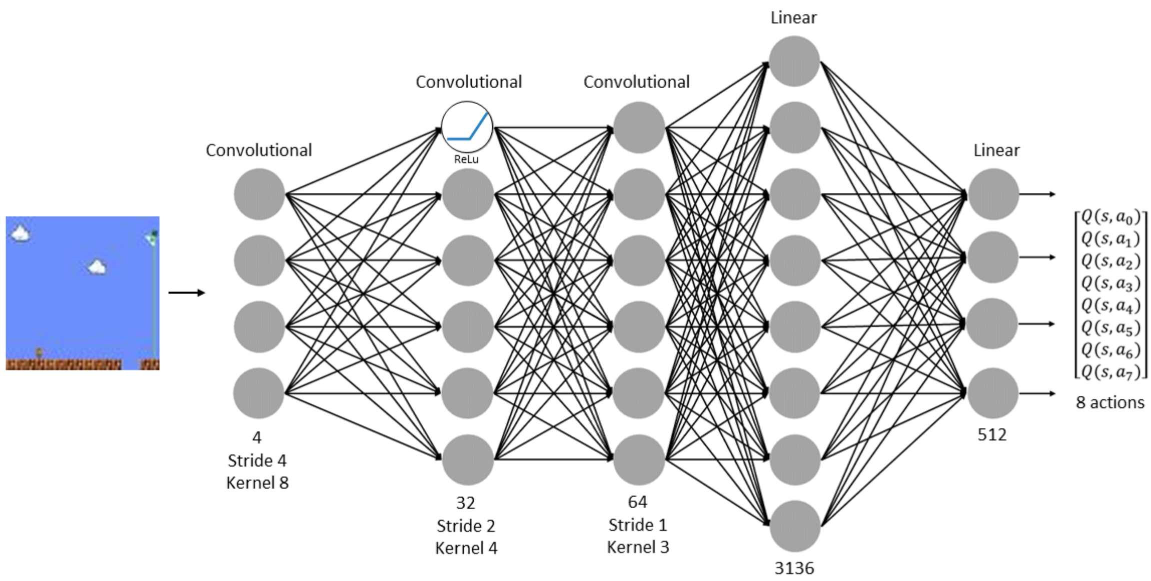


Figure 14: Illustration of the CNN used by the DQN Agent (not to scale). Stride is the number of pixels the convolutional filter moves. Kernel is the size of the filter.

Choosing the hyperparameters for a model is an important aspect. Performing a grid search (source) for optimising parameters would have been to costly considering the time it takes to run each experiment. Therefore, parameters values commonly appearing in the Reinforcement Learning literature and various implementations of the DQN algorithm were chosen.

The DQN agent was able to mostly find successful policies for all three levels, accumulating mostly positive rewards in all except the Steps + Pipe level when trained from scratch. A successful policy

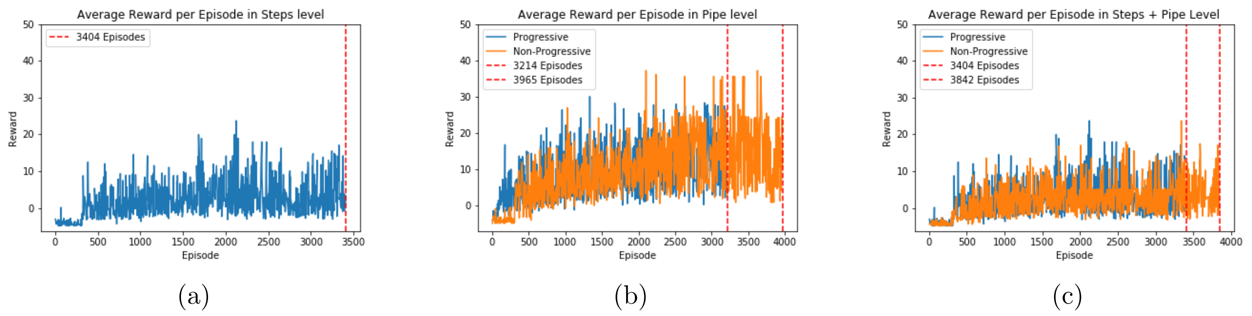


Figure 15: DQN Performance on the three Super Mario levels measured in the average reward per played episode.

Levels	Steps	Progressive Pipe	Pipe	Steps + Pipe	Progressive Steps + Pipe
Final Rewards	242.34	225.76	322.24	-25.0	242.34
Total Rewards	388.12	2485.56	4483.92	-228.09	388.12

Table 3: Rewards accumulated by the DQN agent in each level. Final rewards are the total rewards acquired in the final episode of the run. Total rewards is the sum of all rewards of the run.

should lead to a net positive reward for the agent in the very last episode of the run. In the last episode, the agent is fully exploiting the environment by only choosing optimal actions. Hence, if the agent has learned a successful policy it is expected that the total reward in the last episode is positive. The results also show that a progressive agent who has been trained on the steps level is able to accumulate rewards faster at first in the pipe level when compared to an agent who has to learn the level from scratch (see the first 250 episodes in 15b). Thereafter, the two models have nearly identical reward curves however. The non-progressive agent manages to accumulate a higher reward in the final episode of the run, suggesting that the model learned a better policy. Looking at the total rewards per run, it can be confirmed that the non-progressive model accumulated more in total as well. In the final level where the steps and pipes were combined to test Mario's abilities to reuse information (15c), both the progressive and non-progressive model learned at about equal speeds. Although here the progressive model appears to have found a more successful policy as it has more rewards in the final episode as well as over the entire run.

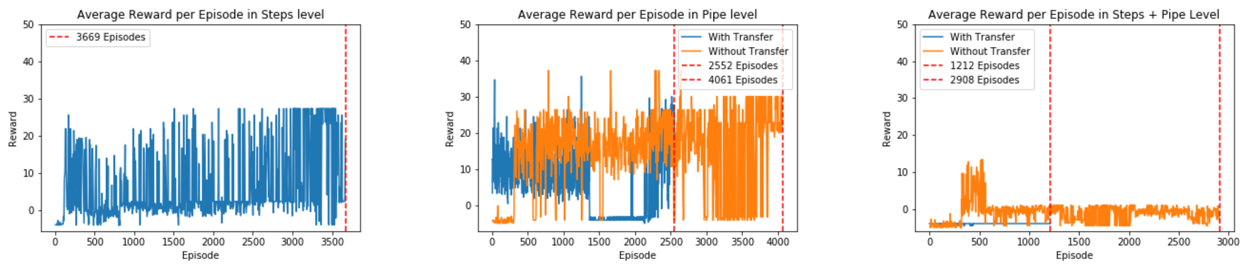
5.3 Option-Critic Agent

The Option Critic implementation follows that of a DQN with the addition of the options framework on top. It is an added layer of complexity, which shows in the experiment runtimes - learning the additional subpolicies takes extra time. Option Critic uses a CNN with the same structure as the DQN (see 14) to learn policies and utilises a policy and target network to avoid overestimating Q-values. Additionally, it also uses the concept of experience replay with a replay buffer of size 100,000 to store observation tuples in and utilises the DQN values for the rest of the hyperparameters as well (see 2). Lastly, the number of options to be learned was specified at 2, taking after the authors of the Option Critic paper [17].

Levels	Steps	Progressive Pipe	Pipe	Steps + Pipe	Progressive Steps + Pipe
Final Rewards	342.47	336.71	420.62	-309.0	-629.0
Total Rewards	-471.95	186.54	4492.79	-2687.05	-3870.95

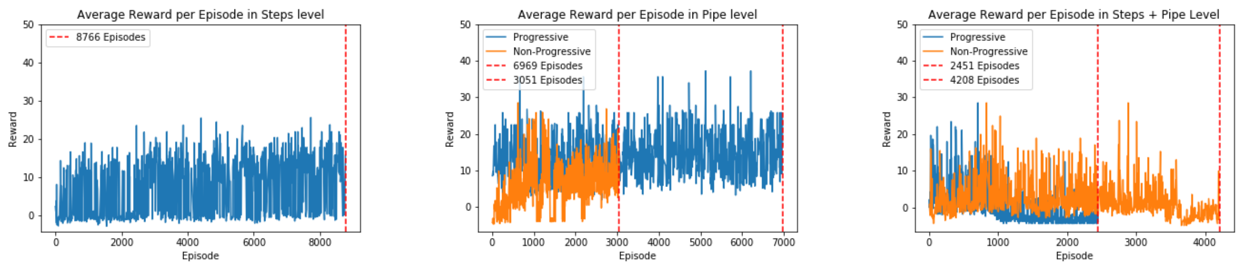
Table 4: The total sum of credits accumulated in each level by the Option Critic agent.

Overall, the performance of the Option Critic is very erratic and unstable in all levels. The average reward obtained per episode goes through extreme fluctuations from values just below zero up to values of thirty. As the episode count goes up the agent is meant to slowly converge on a master policy which returns more rewards than a policy earlier in the run. Though Option Critic agent appears to only be



(a) (b) (c)

Figure 16: Option Critic Performance on the three Super Mario levels measured in the average reward per played episode.



(a) (b) (c)

Figure 17: FuN Performance on the three Super Mario levels measured in the average reward per played episode.

showing minimal improvement over the duration of a run, its final policies actually outperform the DQN on the Steps, Progressive Pipe, and Pipe runs (see Final Rewards in Table 4). Like the DQN, the results show that a progressive agent who has been trained on a previous level is to quicker to adapt to the new environment (see the first 250 episodes 16b). Adapting quickly means to require fewer episodes to finding actions which return larger rewards. Though again, the non-progressive model outperforms the progressive model in the late stages of the run for both the Pipe and Steps + Pipe levels (see final and total rewards in Table 4).

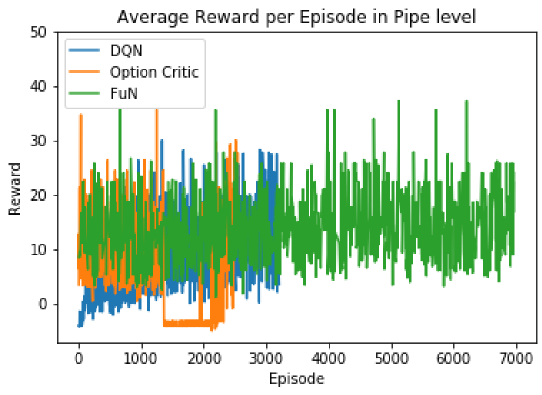
Levels	Steps	Progressive Pipe	Pipe	Steps + Pipe	Progressive Steps + Pipe
Final Reward	330.47	408.62	283.04	-409.73	-629.0
Total Rewards	6595.49	11032.36	2166.36	694.201	-2541.22

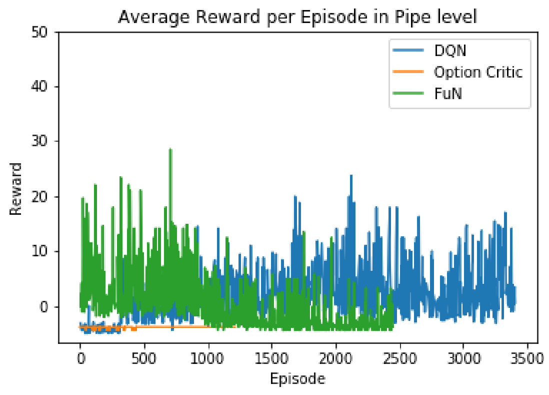
Table 5: The total sum of credits accumulated in each level by the FuN agent.

6 Evaluation of Results

7 Critical Assessment of Project

8 Conclusion





References

[1] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. A. Riedmiller, "Playing atari with deep reinforcement learning," *CoRR*, vol. abs/1312.5602, 2013. [Online]. Available: http://arxiv.org/abs/1312.5602

[2] D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. van den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot, S. Dieleman, D. Grewe, J. Nham, N. Kalchbrenner, I. Sutskever, T. Lillicrap, M. Leach, K. Kavukcuoglu, T. Graepel, and D. Hassabis, "Mastering the game of go with deep neural networks and tree search," *Nature*, vol. 529, pp. 484–503, 2016. [Online]. Available: http://www.nature.com/nature/journal/v529/n7587/full/nature16961.html

[3] K. Arulkumaran, A. Cully, and J. Togelius, "Alphastar: An evolutionary computation perspective," in *Proceedings of the Genetic and Evolutionary Computation Conference Companion*, 2019, pp. 314–315.

[4] M. Hessel, J. Modayil, H. Van Hasselt, T. Schaul, G. Ostrovski, W. Dabney, D. Horgan, B. Piot, M. Azar, and D. Silver, "Rainbow: Combining improvements in deep reinforcement learning," in *Thirty-Second AAAI Conference on Artificial Intelligence*, 2018.

[5] M. G. Bellemare, W. Dabney, and R. Munos, "A distributional perspective on reinforcement learning," in *Proceedings of the 34th International Conference on Machine Learning-Volume 70*. JMLR. org, 2017, pp. 449–458.

[6] O. Nachum, H. Tang, X. Lu, S. Gu, H. Lee, and S. Levine, "Why does hierarchy (sometimes) work so well in reinforcement learning?" *arXiv preprint arXiv:1909.10618*, 2019.

[7] M. L. Puterman, *Markov decision processes: discrete stochastic dynamic programming*. John Wiley & Sons, 2014.

[8] V. François-Lavet, P. Henderson, R. Islam, M. G. Bellemare, J. Pineau *et al.*, "An introduction to deep reinforcement learning," *Foundations and Trends® in Machine Learning*, vol. 11, no. 3-4, pp. 219–354, 2018.

[9] C. J. Watkins and P. Dayan, "Q-learning," *Machine learning*, vol. 8, no. 3-4, pp. 279–292, 1992.

[10] L.-J. Lin, "Reinforcement learning for robots using neural networks," Carnegie-Mellon Univ Pittsburgh PA School of Computer Science, Tech. Rep., 1993.

[11] R. Geirhos, C. R. Temme, J. Rauber, H. H. Schütt, M. Bethge, and F. A. Wichmann, "Generalisation in humans and deep neural networks," in *Advances in Neural Information Processing Systems*, 2018, pp. 7538–7550.

[12] R. S. Sutton, "Generalization in reinforcement learning: Successful examples using sparse coarse coding," in *Advances in neural information processing systems*, 1996, pp. 1038–1044.

[13] H. Van Seijen, M. Fatemi, J. Romoff, R. Laroche, T. Barnes, and J. Tsang, "Hybrid reward architecture for reinforcement learning," in *Advances in Neural Information Processing Systems*, 2017, pp. 5392–5402.

[14] K. Cobbe, O. Klimov, C. Hesse, T. Kim, and J. Schulman, "Quantifying generalization in reinforcement learning," *arXiv preprint arXiv:1812.02341*, 2018.

[15] J. J. Ribas-Fernandes, A. Solway, C. Diuk, J. T. McGuire, A. G. Barto, Y. Niv, and M. M. Botvinick, "A neural signature of hierarchical reinforcement learning," *Neuron*, vol. 71, no. 2, pp. 370–379, 2011.

[16] R. S. Sutton, D. Precup, and S. Singh, "Between mdps and semi-mdps: A framework for temporal abstraction in reinforcement learning," *Artificial intelligence*, vol. 112, no. 1-2, pp. 181–211, 1999.

[17] P.-L. Bacon, J. Harb, and D. Precup, "The option-critic architecture," in *Thirty-First AAAI Conference on Artificial Intelligence*, 2017.

[18] A. S. Vezhnevets, S. Osindero, T. Schaul, N. Heess, M. Jaderberg, D. Silver, and K. Kavukcuoglu, "Feudal networks for hierarchical reinforcement learning," in *Proceedings of the 34th International Conference on Machine Learning-Volume 70*. JMLR. org, 2017, pp. 3540–3549.

[19] P. Dayan and G. E. Hinton, "Feudal reinforcement learning," in *Advances in neural information processing systems*, 1993, pp. 271–278.

[20] A. Khalifa, "Mario ai framework 10th anniversary edition," 2019. [Online]. Available: https://github.com/amidos2006/Mario-AI-Framework

[21] "Jython." [Online]. Available: https://www.jython.org/

[22] "Jpype." [Online]. Available: https://jpype.readthedocs.io/en/latest/

[23] "Py4j." [Online]. Available: https://www.py4j.org/

[24] H. Siebert, *Der Kobra-Effekt: Wie man Irrwege der Wirtschaftspolitik vermeidet.* Dt. Verlag-Anst., 2001.

[25] "Pypi gym super mario bros." [Online]. Available: https://pypi.org/project/gym-super-mario-bros/

[26] R. S. Sutton and A. G. Barto, "Reinforcement learning: An introduction," 2011.

[27] J. White, *Bandit algorithms for website optimization.* " O'Reilly Media, Inc.", 2012.

[28] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski *et al.*, "Human-level control through deep reinforcement learning," *Nature*, vol. 518, no. 7540, pp. 529–533, 2015.

[29] A. Maroti, "Rbed: Reward based epsilon decay," *arXiv preprint arXiv:1910.13701*, 2019.

[30] R. Chuchro and D. Gupta, "Game playing with deep q-learning using openai gym," *Semantic Scholar*, 2017.

[31] K. T. Gribbon and D. G. Bailey, "A novel approach to real-time bilinear interpolation," in *Proceedings. DELTA 2004. Second IEEE International Workshop on Electronic Design, Test and Applications.* IEEE, 2004, pp. 126–131.

[32] G. Lample and D. S. Chaplot, "Playing fps games with deep reinforcement learning," in *Thirty-First AAAI Conference on Artificial Intelligence*, 2017.

[33] A. Braylan, M. Hollenbeck, E. Meyerson, and R. Miikkulainen, "Frame skip is a powerful parameter for learning to play atari," in *Workshops at the Twenty-Ninth AAAI Conference on Artificial Intelligence*, 2015.

[34] T. D. Kulkarni, K. Narasimhan, A. Saeedi, and J. Tenenbaum, “Hierarchical deep reinforcement learning: Integrating temporal abstraction and intrinsic motivation,” in *Advances in neural information processing systems*, 2016, pp. 3675–3683.

[35] A. Levy, G. Konidaris, R. Platt, and K. Saenko, “Learning multi-level hierarchies with hindsight,” *arXiv preprint arXiv:1712.00948*, 2017.

[36] Y. LeCun, Y. Bengio, and G. Hinton, “Deep learning,” *nature*, vol. 521, no. 7553, pp. 436–444, 2015.

[37] P. Ramachandran, B. Zoph, and Q. V. Le, “Searching for activation functions,” *arXiv preprint arXiv:1710.05941*, 2017.

Abbeel, P. and Schulman, J. (2015). CS 294: Deep Reinforcement Learning, Fall 2015. UC Berkely.

Arulkumaran, K., Deisenroth, M., Brundage, M. and Bharath, A. (2017). Deep Reinforcement Learning: A Brief Survey. IEEE Signal Processing Magazine, 34(6), pp.26-38.

Bellman, R. E. (1957). Dynamic Programming. Princeton University Press, Princeton.

Bellemare, M. G., Dabney,W., and Munos, R. (2017). A distributional perspective on reinforcement learning. In the International Conference on Machine Learning (ICML).

Botvinick, M., Niv, Y. and Barto, A. (2008). Hierarchically organized behavior and its neural foundations: A reinforcement learning perspective. Cognition, 113(3), pp.262-280.

Borges, D. (2018). The Curse of Dimensionality!. [online] Medium. Available at: https://medium.com/diogo-menezes-borges/give-me-the-antidote-for-the-curse-of-dimensionality-b14bce4bf4d2 [Accessed 15 Nov. 2019].

Bradtke, S.J. and Duff, M.O. (1995). Reinforcement learning methods for continuous-time Markov decision problems, in: Advances in Neural Information Processing Systems 7, MIT Press, Cambridge, MA, pp. 393–400.

Cobbe, K., Klimov, O., Hesse, C., Kim, T. and Schulman, J. (2019). Quantifying Generalization in Reinforcement Learning. In: Proceedings of the 36 th International Conference on Machine Learning (ICML).

Dawes, G. (2017). Ancient and Medieval Empiricism (Stanford Encyclopedia of Philosophy/Winter 2017 Edition). [online] Plato.stanford.edu. Available at: https://plato.stanford.edu/archives/win2017/entries/empiricism-ancient-medieval/ [Accessed 14 Nov. 2019].

learning. In: Advances in Neural Information Processing Systems 5, Morgan Kaufmann, San Mateo, CA, pp. 271–278.

Deepmind. (2019). AlphaStar: Mastering the Real-Time Strategy Game StarCraft II. [online] Available at: https://deepmind.com/blog/article/alphastar-mastering-real-time-strategy-game-starcraft-ii [Accessed 18 Nov. 2019].

Diuk, C., Cohen, A. and Littman, M. (2008). An object-oriented representation for efficient reinforcement learning. Proceedings of the 25th international conference on Machine learning - ICML ’08.

Fler-Berliac, Y. (2019). The Promise of Hierarchical Reinforcement Learning. [online] The Gradient. Available at: https://thegradient.pub/the-promise-of-hierarchical-reinforcement-learning/ [Accessed 18

Nov. 2019].

Geirhos, R., Temme, C., Rauber, J., Schütt, H., Bethge, M. and Wichmann, F. (2018). Generalisation in humans and deep neural networks. In: Neural Information Processing Systems (NIPS).

Guestrin, C., Koller, D., Gearhart, C., & Kanodia, N. (2003). Generalizing plans to new environments in relational mdps. IJCAI. pp. 1003–1010.

Hessel, M., Modayil, J., van Hasselt, H., Schaul, T., Ostrovski, G., Dabney, W., Horgan, D., Piot, B., Azar, M., and Silver, D. (2018). Rainbow: Combining Improvements in Deep Reinforcement Learning. In the Thirty-Second AAAI Conference on Artificial Intelligence (AAAI).

Justesen, N., Bontrager, P., Togelius, J. and Risi, S. (2019). Deep Learning for Video Game Playing. IEEE Transactions on Games, pp.1-1.

Keramati, R., Whang, J., Cho, P. and Brunskill, E. (2018). Strategic Exploration in Object-Oriented Reinforcement Learning. Published at the Exploration in Reinforcement Learning Workshop at the 35th International Conference on Machine Learning, Stockholm, Sweden.

Konidaris, G., Kaelbling, L. and Lozano-Perez, T. (2018). From Skills to Symbols: Learning Symbolic Representations for Abstract High-Level Planning. Journal of Artificial Intelligence Research, 61, pp.215-289.

Minsky, M. (1961). Steps towards Artificial Intelligence. Proceedings of the IRE, 49(1), pp.8-30.

Mnih, V., Kavukcuoglu, K., Silver, D., Graves, A., Antonoglou, I., Wierstra, D., and Riedmiller, M. (2013). Playing atari with deep reinforcement learning. In NIPS Deep Learning Workshop.

Peng, X., Chang, M., Zhang, G., Abbeel, P. and Levine, S. (2019). MCP: Learning Composable Hierarchical Control with Multiplicative Compositional Policies. NeurIPS.

Posner, M. I., & Cohen, Y. (1984). Components of Visual Orienting. In H. Bouma, & D. Bowhuis (Eds.), Attention and Performance X (pp. 531-556). Hillsdale, NJ: Erlbaum.

Puterman, M. (2005). Markov Decision Processes: Discrete Stochastic Dynamic Programming. Hoboken, New Jersey: John Wiley & Sons.

Scholz, J., Levihn, M., Isbell, C. L., and Wingate, D. (2014). A Physics-Based Model Prior for Object-Oriented MDPs. In Proceedings of the 31st International Conference on Machine Learning (ICML).

Seijen, H., Fatemi, M., Romoff, J., Laroche, R., Barnes, T. and Tsang, J. (2017). Hybrid Reward Architecture for Reinforcement Learning. In: 31st Conference on Neural Information Processing Systems (NIPS 2017).

Stolle, M., and Precup, D. (2002). Learning options in reinforcement learning. In Abstraction, Reformulation and Approximation, 5th International Symposium, SARA Proceedings, 212–223.

Sutton, R. (1995). Generalization in Reinforcement Learning: Successful Examples Using Sparse Coarse Coding. In: Neural Information Processing Systems (NIPS).

Sutton, R. and Barto, A. (2014). Reinforcement Learning: An Introduction. 2nd ed. Cambridge, Massachusetts: The MIT Press.

Sutton, R., Precup, D. and Singh, S. (1999). Between MDPs and semi-MDPs: A framework for temporal abstraction in reinforcement learning. Artificial Intelligence, 112(1-2), pp.181-211.

Vezhnevets, A. S., Osindero, S., Schaul, T., Heess, N., Jaderberg, M., Silver, D., and Kavukcuoglu, K. (2017). Feudal networks for hierarchical reinforcement learning. Proceedings of the 34th International Conference on Machine Learning - Volume 70. Pages 3540-3549.