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 subfield 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 success and gained significant medial and research attention. Algorithms have been shown to outperform the best human players on a wide range of Atari games [**?**], chess, Shogi and the strategy board game Go [**?**], and even Starcraft [**?**]. Super Mario as a domain is not as popular as the Atari games amongst researchers, though it shares many similarities with the Atari suite that make applying the vast Reinforcement Literature straightforward. The Mario domain is high-dimensional with relatively low gameplay complexity, i.e. the number of possible scenarios for Mario to be in is large, but the game mechanics are uncomplicated.

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 still does not know what a ladder is and will have to relearn how to climb it in a new situation. The agent struggles with transferring knowledge between situations because they fail to see the similarities between them. For this reason, contemporary algorithms that play at a superhuman level require absurd amounts of training episodes on powerful hardware. Agents need to learn a vast array of different situations to play at human or superman level. For example, DeepMind's Rainbow architecture [**?**] attains superhuman performance after experiencing 18 million frames, corresponding to 83 hours of Atari gameplay. Distributional RL [**?**] surpasses human players after 70 million frames in Atari games. If agents were able to sequentially build up a pool of knowledge by abstracting away behaviour when learning and reuse already learned information when faced with new situations, the training time could be reduced.

Hierarchical Reinforcement Learning is an area of Reinforcement Learning that works on introducing the abstraction of behaviour. Broadly speaking, abstracting behaviour means to create functions which solve some subtask in the environment. 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. Playing the game, they would most likely not think in terms of button combinations, but in terms of jumping ravines, collecting coins and killing Goombas. The functions required to finish a level could be 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. *Temporally abstract* actions that do not exist on the game controller, they are aggregates of the lower level button combinations. Recent research [**?**] 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 and is a good choice for this project.

1.2 Objectives

There are two main objectives this research project set out to achieve. First, compare the performance of sequentially and non-sequentially trained agents to measure their bootstrapping abilities. Two sets of agents are placed in learning scenarios where low-complexity Super Mario levels are to be solved. The set of non-sequential agents are trained from scratch in each level - they have never interacted with the Super Mario environment before. The set of sequential agents learns the levels one after the other and have history of interactions with the Super Mario environment. The second objective is to compare the performance of sequential traditional and sequential hierarchical Reinforcement Learning

algorithms. Both objectives serve to establish whether learning can be sped up by sequentially training agents and using hierarchical Reinforcement Learning.

2 Literature Review

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 when building Reinforcement Learning agents. This section will outline the basic concepts of Reinforcement Learning and explain the improvements made by neural networks. In a reinforcement learning problem, agents are tasked with executing a sequence of actions to maximise a score. In order to maximise that score, they need to optimise their behaviour. Agents solve Reinforcement Learning problems by interacting with the environment they live in and receiving feedback on which actions are good and bad. Essentially, it is trial and error until the agent found actions that lead to a high score. The environment in an RL-problem is typically modelled as a Markov-Decision-Process (MDP), a discrete stochastic sequential decision process [**?**] 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 possible 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. This is the score the agent wants to maximise. The reward function used in this project is described in section 4.3.
- γ is a discount factor to prevent infinite rewards.

Agents in the MDP take actions to move between states in the state space in discrete timesteps. At a given point in time t, the agent performs an action a_t in state s_t, 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. A policy is a function that maps the state space to the action space. When provided with a state, it will tell the agent which action to take. Not all policies are equally good however. They are measured on how much reward they produce when the agent decides to use a certain policy. Because the agent wants to maximise the reward signal, policies producing more rewards are better than policies that do not return as much reward. An optimal policy π^* will always prescribe the action which gives the highest expected return of reward for every state in the state space. Goal of the agent is to figure out the best policy for the MDP.

One of the most popular algorithms for learning optimal policies is Q-learning [**?**]. Q-learning is an off policy model free algorithm which means that the agent does not learn transition and reward functions but instead learns so called Q-values from which the best possible action can be deduced by taking random actions in the environment. 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. This is because the reward function returns negative rewards for dying at the hands of a Goomba and positive rewards for killing it. The optimal Q-value is described with the *Bellman Optimality Equation* [**?**]. It states that the Q-value of a state-action pair (s, a) at timestep t is the

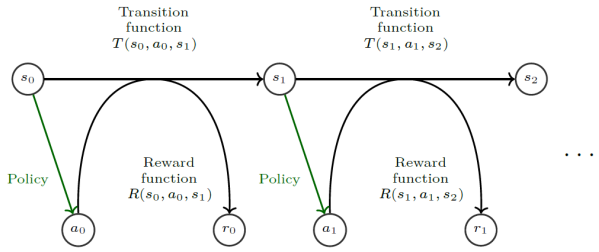


Figure 1: Agent takes an action to move to the next state and observe a reward [**?**]

expected reward of that pair R_{t+1} plus the maximum discounted return of the next state-action pair $Q(s', a')$:

$$Q^*(s, a) = \mathbb{E}\left[R_{t+1} + \gamma \max_{a'} Q(s', a')\right] \tag{1}$$

Because an optimal policy will choose the action which returns the highest reward and Q-values tell us how much reward to expect from an action, an optimal policy chooses the action associated with the largest Q-value. Hence, 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 iteratively update the Q-values until they converge. Observations are the states and rewards associated with taking actions in the state space. 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 are suitable for low-dimensional problems with small state and action spaces, however they quickly become infeasible for complex problems. For instance, assuming an image size of 84×84, the RGB Super Mario state space has a size of $(84 \times 84 \times 3)^{256}$. Storing the Q-values for a state space of that size is impossible. The solution is to replace the lookup table with a neural network called a Deep Q Network (DQN) or policy network. This works because neural networks are universal function approximators and can be used to learn any mapping. Convolutional Neural Networks (CNN) have become especially popular within the Reinforcement Learning community because they allow end to end learning. CNNs have made it possible to identify features in images and map them to outputs, thus removing the process of manually extracting features from game frames. Furthermore, CNNs reduce the need for a large memory footprint and allow for 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 a vector of Q-values which can then be mapped to actions to be taken.

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 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. Each action is represented by a neuron in the output layer in the network. Ranks in this case are the Q-values of the state that was passed through the network, 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 the agent can learn the optimal Q-values of the MDP. First, the loss of the network is calculated using

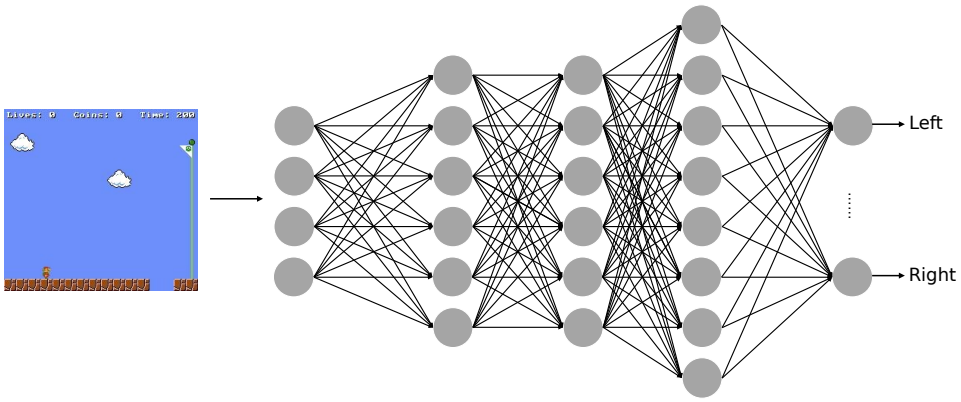


Figure 2: Policy Network for Super Mario (not to scale)

a loss function such as the huber loss [**?**], which is then differentiated with respect to the weights of the network before taking a gradient step. The network is fully trained when the output Q-values have sufficiently approximated the true 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. The loss is computed as the difference between the outputted Q-values of the current state-action pair and the true Q-values.

$$loss = Q^*(s, a) - Q(s, a) \tag{3}$$

Calculating the optimal Q-value of a state $Q(s, a)$ requires the maximum Q-value $Q(s', a')$ of the next state (see equation 1). However, 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 optimal targeted Q-value. Yet 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 trap you 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* [**?**]. The replay memory is a buffer of size N where the agent stores experience tuples $e_t = (s_t, a_t, r_{t+1}, s_{t+1})$. The agent uses samples from the replay memory to train the DQN, not the live frame it currently sees. 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 [**?**] 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 those features to other doors. While Deep Reinforcement Learning partially alleviates the issue of generalisation [**?**], it remains a substantial issue [**?**] to date. Reinforcement Learning agents are prone to overspecialise and overfit because they are trained and tested within the same environment [**?**]. 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* [**?**] 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. 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}. In Figure 3, 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.

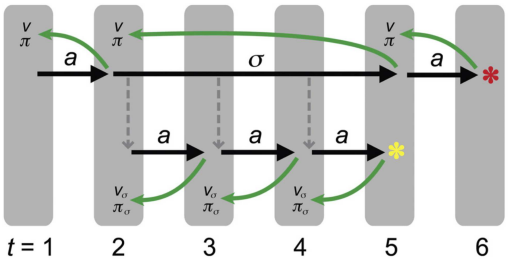


Figure 3: The Hierarchical RL Semi Markov Decision Process [**?**]

2.2.1 Option-Critic

The following Hierarchical algorithm builds upon the Markov-Options framework [**?**] 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.
- $\pi : S \times A \rightarrow [0, 1]$ 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 [**?**] 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 for the Q-value 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. The algorithm can be found in more detail in *The Option-Critic Architecture* [**?**].

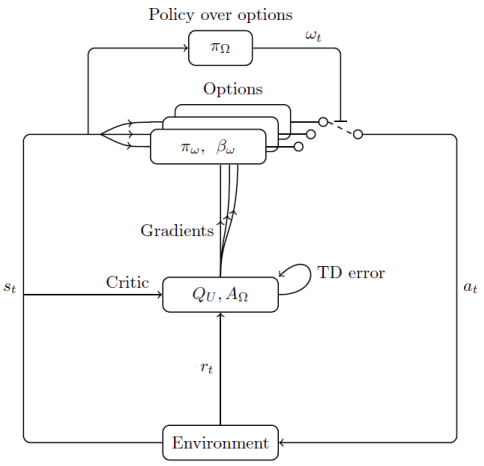


Figure 4: Option-Critic Architecture [**?**]

2.2.2 Feudal Networks (FuN)

Another recent Hierarchical Reinforcement Learning algorithm is DeepMind's FeUdal Network (FuN) [**?**]. Like the Option-Critic architecture, is is an improvement on an older system, called Feudal Reinforcement Learning [**?**]. 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 is theoretically unbounded. 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. The latent space is a compressed version of the data. 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 [**?**] 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 which means that FuN produces a stochastic policy

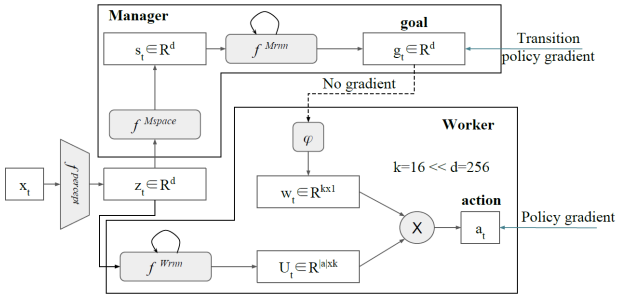


Figure 5: FuN Architecure [**?**]

unlike the deterministic DQN and Option Critic. The FeUdal Network learns by a *transition policy gradient* which boilds down to training the manager to predict goals which return large rewards in the state space.

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 [**?**] 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 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 [**?**] and JPype [**?**] frameworks allow for Python code to be executed in a Java environment by being directly embedded in the JVM. This reduces latency, however they have a steep learning curve and in the case of Jython only support Python 2.7. The Py4J [**?**] 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. 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 periods lasting 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 against failures and accidents. Google Cloud provides APIs specifically designed for Machine Learning and comes with free credits for first time users.

4 System Implementation

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

4.1 Super Mario Game

Super Mario Bros game was released by Nintendo in 1985 and is the most iconic Super Mario game to date. It was the first Mario game to have a sidescrolling feature, i.e. where the game is viewed from the side and movement to the left or ride gradually reveals more of the level via scrolling, and introduced many of the now commonly known features such as Goombas and power ups. The goal for Mario is to get to the flag at the end of a level without dying or timing out in the process. Each level contains obstacles such as gaps, pipes, blocks and enemies like Goombas or Koopas.

4.2 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 observes states and rewards in return. The client first initiates the environment via the game interface and receives back the start state of the MDP (note that this process is not displayed in the game server diagram in Figure **??**). 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. $\begin{bmatrix} false & true & false & true & false \end{bmatrix}$. When the action vector is received by the Game Server via Py4J, the action is executed in the environment k times (see section 4.6 for frame skipping). Having executed the acition k times, the final frame of the action sequence is cropped and resized (see section 4.5) and the reward is calculated using the environment's reward function (see section 4.3). 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.

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* [**?**] framework which had to be . extended and rewritten to fit the requirements of the system. Various methods and classes for calculating the reward, preprocessing the game frame and exposing the observations of the environment to an outside agent were added to the existing codebase. To allow for remote training, both client and server are running on a single 4CPU 16GB Google Cloud

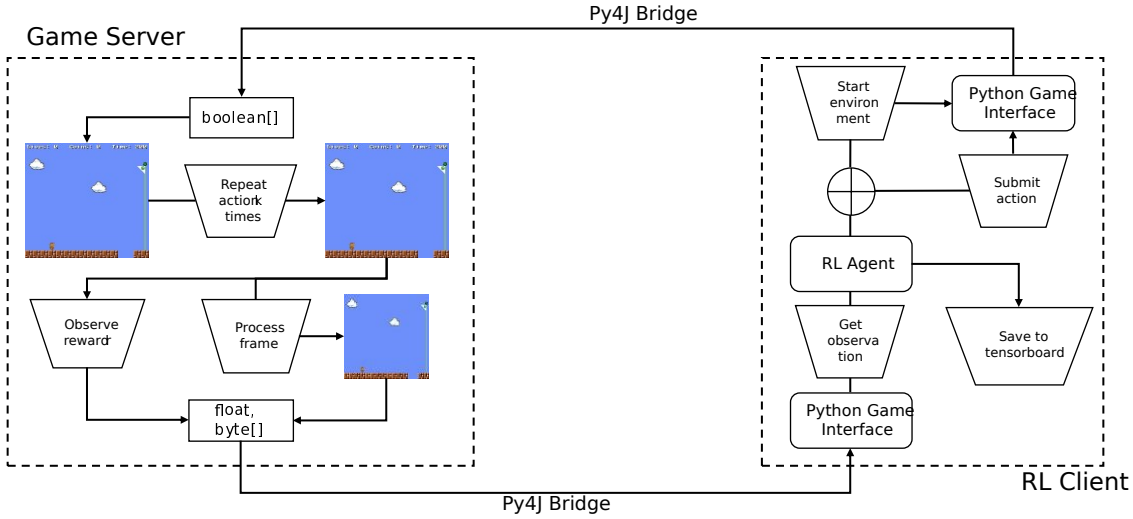


Figure 6: System Architecture

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.

4.3 Reward Function

A Reinforcement Learning agent learns to distinguish between desirable and undesirable behaviour by observing the reward signal associated with actions in 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 $\sum_{i=1}^{t} R_i$. The agent needs to receive rewards such that when maximised achieve the goal or solve the problem that 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 [**?**]. 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, known as *reward hacking* [**?**]. 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 because they are constantly losing points. With this in mind, 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. The reward function used in this project is inspired by the *openai gym* Super Mario Bros framework [**?**]. 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. 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. In the original game, Mario receives more points the higher up he hits the flag at the end of the level. This was not modelled in the reward function to reduce its complexity.
- 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 $(-100, 100)$.

4.4 Exploitation vs Exploration

A major challenge within Reinforcement Learning is to strike a balance between gathering enough information about the environment (exploration) and obtaining as much reward as possible (exploitation) [**?**]. 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 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 [**?**]. 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 [**?**], exponential [**?**] and sinusoidal [**?**]. The algorithms that use an $\epsilon - Greedy$ policy in this project use a linear decay.

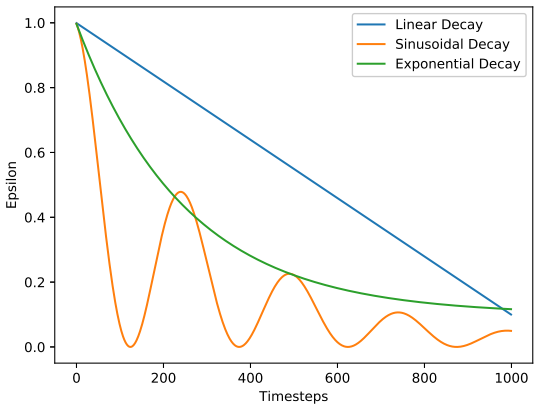


Figure 7: Variants of ϵ Decay

4.5 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, it needs to go through a series of preprocessing steps aimed at reducing its dimensionality to make 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 [**?**] to 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 [**?**]. Converting the image to grayscale is often applied preprocessing step, though grayscale models tend to perform lower [**?**].

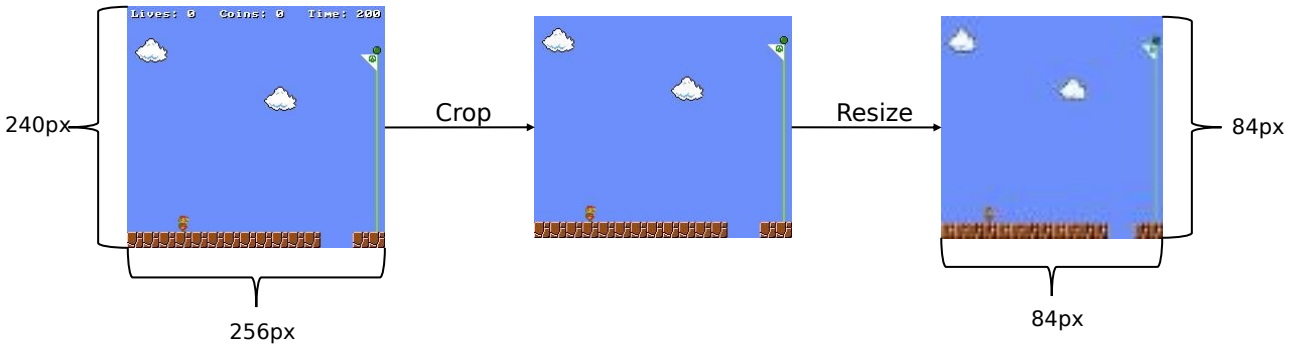


Figure 8: Frame Preprocessing Pipeline

4.6 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 during training. Using 4 frames was again pioneered by DeepMind [**?**].

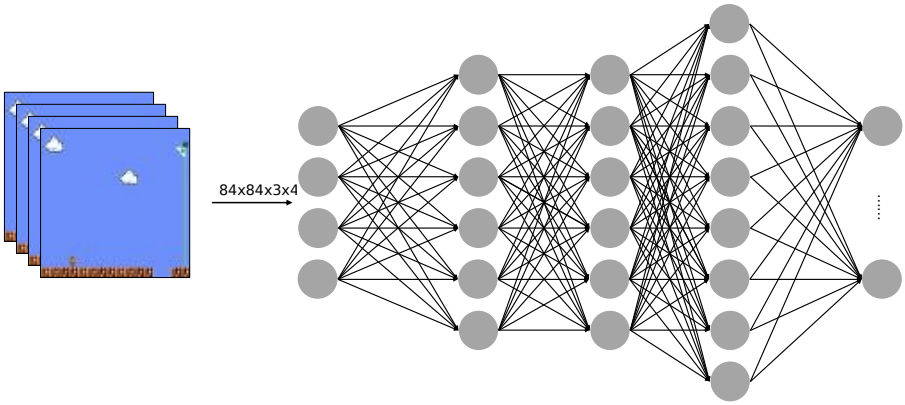


Figure 9: Stacked frames are given as input to the neural network. The dimension of the input thus becomes $84 \times 84 \times 3 \times 4$

.

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 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 doing a forward pass through the policy network to decide on 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 because the emulator is doing more work than the policy network. However, the agent might miss out on opportunities in the game because the same action is repeated too many times [**?**]. 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. This project uses the value $k = 4$.

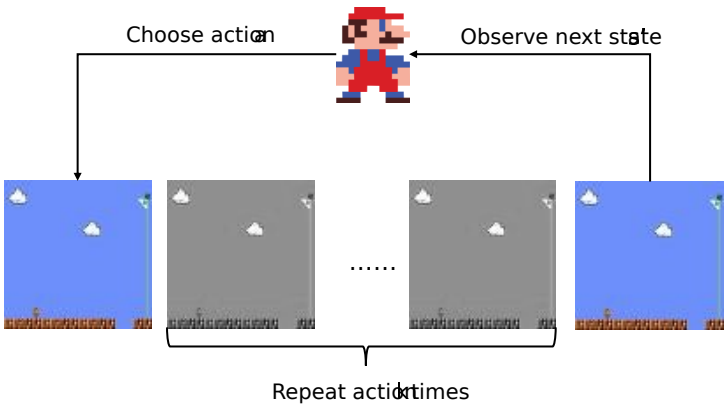


Figure 10: The agent selects action a to be executed k times in the emulator and observes the k^{th} frame

.

4.7 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 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` files 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

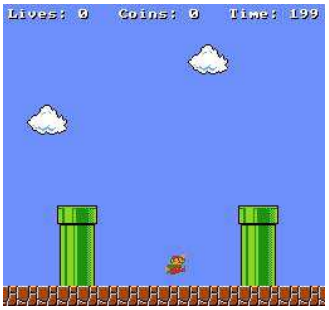


Figure 12: Two Pipes Level

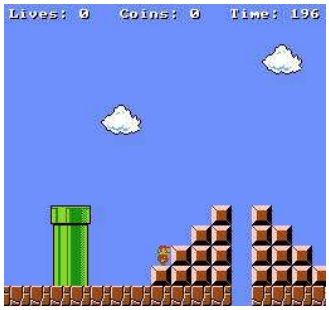


Figure 13: Steps with Gap + Two Pipes

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 Experiment Design

The goal of this project is to investigate whether introducing hierarchies, temporal abstraction and sequential learning 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 is trained sequentially on three levels. The first two levels are designed for Mario learn specific skills - navigating steps and jumping over pipes respectively (section 4.7). The third level is a combination of the two previous levels and requires 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, which should show in an initially steeper reward curve. To judge whether learning *sequentially* gives a learning advantage, the agents will also be compared against *non-sequential* versions of themselves. These are models which have not been trained on previous levels and come without any prior knowledge. The purpose of this comparison is to investigate whether training agents *sequentially* is beneficial over learning from scratch.

Model	**Level**	**Sequential**	**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. sequential indicates whether the model has been pretrained on another level already.

Determining the stopping criterion for the experiments, i.e. when the model should stop learning, was grounds to some experimentation. Letting agents play a set number of episodes 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. Therefore, there would be no guarantee that agents get the same number of experiences. It was decided to let all agents experience the same number of frames of the game to better predict training times and to also use the number of completed episodes as an indicator of performance. 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. 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.

6 Results

6.1 DQN Agent

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 sequential and non-sequential 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) [**?**], which over the past years established itself as the most popular activation function for training neural networks [**?**]. The agent also has an experience replay buffer of size 100,000.

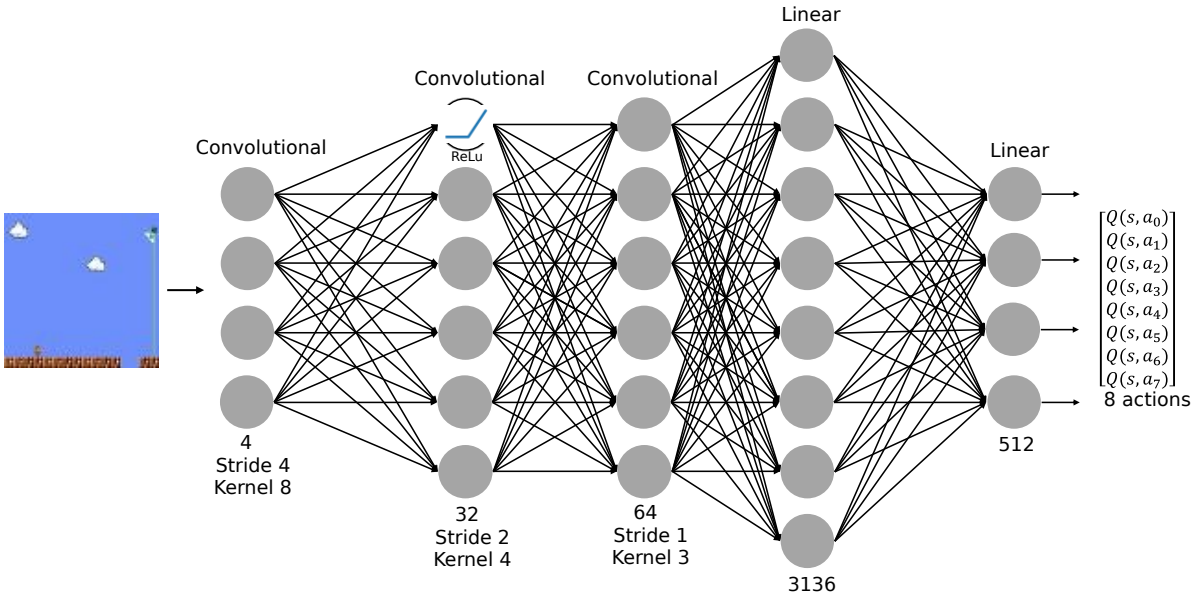


Figure 14: Illustration of the CNN used by the DQN Agent (not to scale). Stride is the number of pixels the convolutional filter moves. Kernel is the size of the filter.

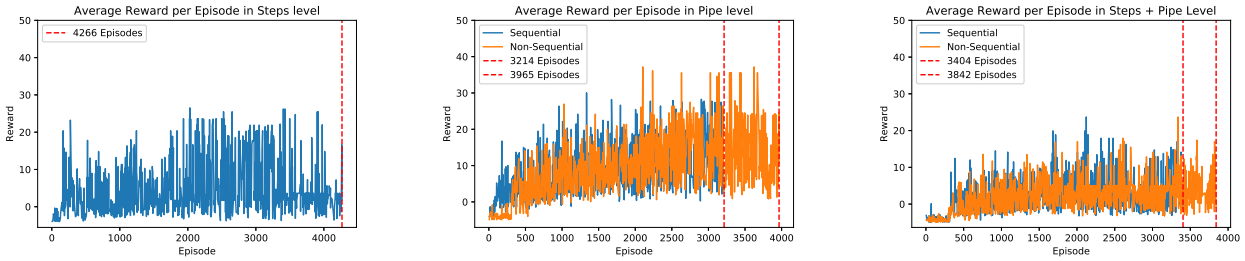
Choosing the hyperparameters for a model is an important aspect. Performing a grid search **??** for optimising parameters would have been to costly considering the time it took to run each experiment. Therefore, parameters values commonly found in the Reinforcement Learning literature and various implementations 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 non-sequential Steps + Pipe level. A successful policy 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 sequential 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-sequential 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-sequential model accumulated more in total as well. In the final level where the

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.



(a) (b) (c)

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

Levels	Steps	sequential Pipe	Pipe	Steps + Pipe	sequential Steps + Pipe
Final Rewards	128.73	225.76	322.24	-25.0	242.34
Total Rewards	409.92	2485.56	4483.92	-228.09	388.12

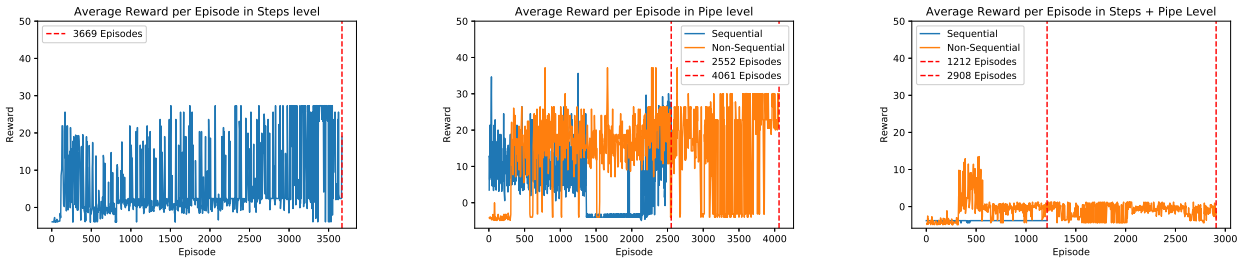
Table 3: Rewards accumulated by the DQN agent in each level. Final rewards are the total rewards acquired in the final episode of the run. Total rewards is the sum of all rewards of the run.

steps and pipes were combined to test Mario's abilities to reuse information (15c), both the sequential and non-sequential model learned at about equal speeds. Although here the sequential 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.

6.2 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 [**?**]. The code was sourced from public github repositories [**?**] and adapted to fit the specifications of the project.

Overall, the performance of the Option Critic is very erratic and unstable in all levels. The average



(a) (b) (c)

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

Levels	Steps	sequential Pipe	Pipe	Steps + Pipe	sequential 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.

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 showing minimal improvement over the duration of a run, its final policies actually outperform the DQN on the Steps, sequential Pipe, and Pipe runs (see Final Rewards in Table 4). Like the DQN, the results show that a sequential agent who has been trained on a previous level is to quicker to adapt to the new environment (see the first 250 episodes in 16b). Adapting quickly means to require fewer episodes to finding actions which return larger rewards. However, the non-sequential model outperforms the sequential model in the late stages of the run for both the Pipe and Steps + Pipe levels (see final and total rewards in Table 4). Occasionally, the Option Critic gets stuck in local minima which can be seen as the flat sections with low reward values in the reward curve (see sequential 16b and sequential 16c). It indicates that Mario has gotten stuck somewhere in the level and keeps accumulating the negative reward for letting the time pass. The agent manages to leave the local minimum in the Pipe level but not in the Steps + Pipe level.

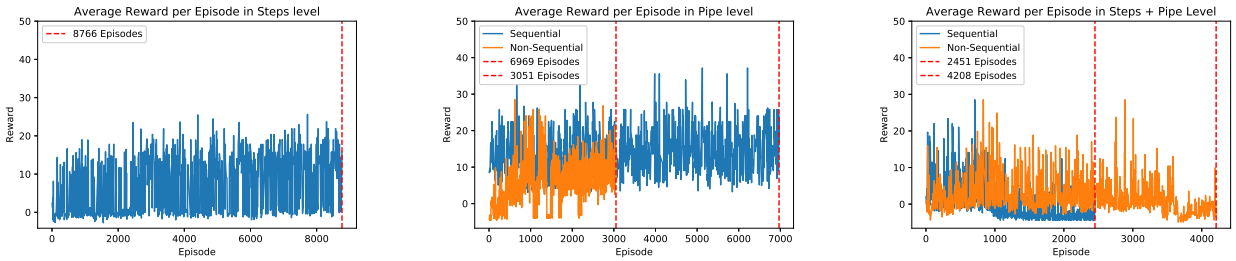
6.3 FuN Agent

The implementation of the FuN agent differs from that of the DQN and Option Critic. Though it also has a CNN with ReLu activation functions, the number of layers as well as values for stride and kernel are different. This is because the CNN does not compute the Q-values but an intermediate state representation that is to be used further down the processing pipeline. Because FuN uses recurrent networks in the form of LSTMs to compute an action distribution for the agent, the architecture can do without the concept of memory replay and target networks. Hence, FuN has a lower memory footprint than both the DQN and Option Critic which might have to store millions of experiences for more complex problems. The intricate design of the FuN agent is more computationally expensive however as can be seen in the experiment runtimes (see Table 1). As with the Option Critic and DQN agent, the code for the FuN agent was taken from a public github repository [**?**] and altered to fit the project specifications.

Levels	Steps	sequential Pipe	Pipe	Steps + Pipe	sequential 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.

Out of the three agents, the FuN agent was able to accumulate the most total rewards in the



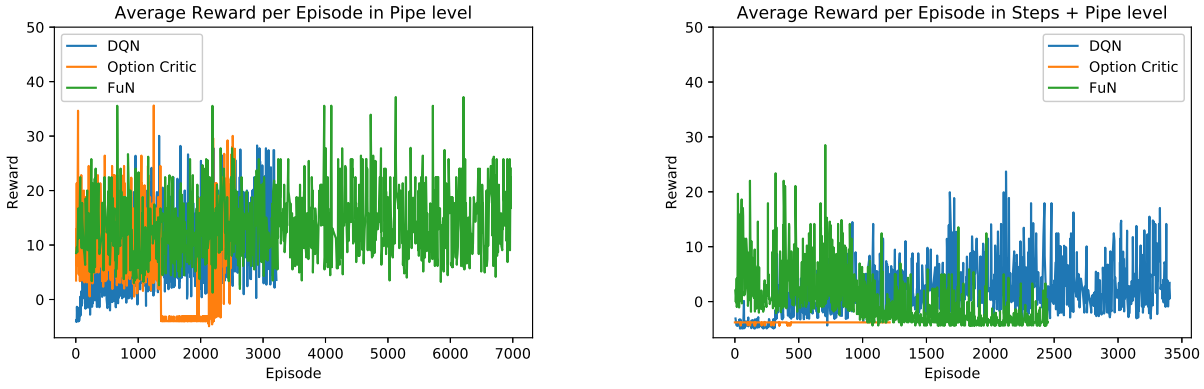
(a) (b) (c)

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

Steps, sequential Pipe and Steps + Pipe level. It also outperformed the DQN on the final rewards in the Steps and the sequential Pipe level. As with the DQN and Option Critic, the sequential model which was trained on the Steps level outperforms that of a model which learns the level from scratch (see the first 1000 episodes in 18b). Whereas the sequential DQN and Option Critic models were outperformed by the non-sequential counterparts later in the run, the sequential FuN does better than the non-sequential FuN in the final reward and total reward. Much like the the Option Critic, FuN has a worse performance than the DQN on the final Steps + Pipe level.

6.4 Evaluation of Results

Comparing the sequential models against non-sequential models showed that sequential models have an initial advantage over non-sequential models when being placed in a new environment. They accumulate rewards quicker in the beginning because the weights of the policy network have already been trained on a somewhat related environment. Thus, the models becomes equipped with previous knowledge they can use to their favour. The next step is to compare how the various sequential models perform when being placed in a novel environment. A flat learning agent like the DQN should accumulate knowledge of lower quality than the the hierarchical agents because it does not have temporally abstract actions available. Quality of knowledge in this case is measured by its reusability. Knowledge which can be transferred to new spaces is highly reusable and of high quality. Likewise, knowledge which is only applicable to a narrow space is of poor reusability and thus low quality. The Option Critic and FuN agent are expected to learn reusable subpolicies which should translate to higher quality knowledge and therefore more rewards in the novel environment.



(a) (b)

Figure 18: Comparing the average rewards per episode for the two levels where models were pretrained.

Comparing the average rewards per episode for all three sequentially trained algorithms shows that the hierarchical models have an initial advantage over the DQN in the Pipe level. Both the FuN and Option Critic agent are able to obtain higher rewards than the DQN in the first 1,000 episodes of

the run. This suggests that they learned reusable hierarchies in the Steps level they were previously trained on. Additionally, the learned hierarchies are superior to the learning achievements of the DQN because they allowed the agents to choose actions which lead to higher rewards. Though the DQN catches up with the hierarchical models later, initially it cannot capitalise on the information from the previous Steps level as much as hierarchical models. The same cannot be said for the the final Steps + Pipe level however as it gives inconclusive results on whether hierarchical models can exploit previously learned knowledge. The FuN agent initially obtains high rewards but converges to a suboptimal policy. The Option Critic is stuck in a suboptimal policy from the start and the DQN starts out with low rewards but is able to find a successful policy in the end. There may be reason to think that training already trained models on new environments causes them to forget what they previously learned. This would explain the inconsistent results for the performance of sequential models compared against non-sequential models for the average rewards per episode for the Steps + Pipe level in the previous sections (6.1, 6.2, 6.3). Whereas the sequential FuN model learned quicker than its counterpart, the opposite was true for the Option Critic whereas the DQNs learned at around the same speed. Therefore, it can only be concluded that hierarchical sequential models have an initial learning advantage going from the Steps to the Pipe level. Adding another level to the sequence appears to cause something known as *catastrophic forgetting*. When neural networks are continuously learning new tasks, the weights that were used to solve task A are overwritten to meet the objective of task B [**?**]. When the network is given task A after learning task B, it has "forgotten" how to solve it. Having learned how to navigate steps and pipes in the first two levels, the various agents fail to reuse that knowledge in the final level which combines the two, because the network weights have been overwritten.

6.5 Further Policy Analysis

Sequentially trained hierarchical agents are faster learners when being shown a new environment (atleast for a task sequence of length 2), but are they able to find better policies? An agent can have a fast learning speed but in the end only find suboptimal policies. It was postulated that an agent who is a fast learner will achieve more rewards faster and play more episodes in the same amount of frames. Using the number of completed episodes shows that the non sequential models were able to play more episodes with the exception of the sequential FuN on the pipe level and the sequential DQN on the steps + pipe level. However, only relying on the number of completed episodes is not a reliable indicator of a good policy. Playing more episodes does not necessarily correlate with finding a better policy, it just means that you are able to finish an episode faster. An agent acting according to a policy which forces them to commit suicide immediately is able to play many episodes, but will accumulate low rewards. Therefore, looking at the number of completed episodes as the only variable variable will give the illusion of a good policy. Only where the number of completed episodes correlates positively with an overall higher reward can it be concluded that the policy is superior. Figure 19 shows a moderate positive correlation between the number of episodes completed by an agent and the averaged average reward per episode per run. Broadly speaking, the higher the number of episodes completed, the higher the average reward. Thus, an agent that plays more episodes generally ends up with a good policy too. Although, it cannot be concluded that the hierarchical models were able to find better policies than the DQN as figure 19 shows no discernible pattern for the different algorithms.

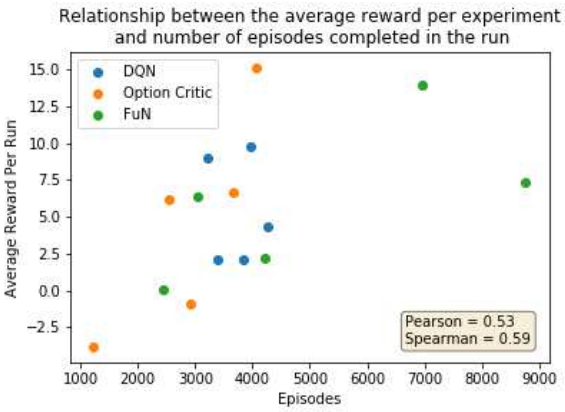


Figure 19: The average reward per run and the number of completed episodes. Pearson is a linear correlation coefficient. Spearman is a nonlinear correlation coefficient.

7 Deviations From Original Specifications

The project mostly adhered to the original specifications written down at the beginning with two exceptions. First, communication between the RL agents and game emulator was supposed to happen with JSON using HTTP requests. After trialling this method it was deemed to slow and exchanged for the faster Py4J bridge. Second, the set of hierarchical algorithms to be tested included the original Options framework and an approach called Multiplicative Compositional Policies (MCP). The Options framework was substituted for the Option Critic model which is a more recent extension of it and was therefore deemed more reflective of the contemporary Reinforcement Learning literature. The MCP was exchanged for DeepMind's Feudal Network because of the lack of information and code available, both of which spoke in DeepMind's favour instead.

8 Critical Assessment of Project

This section will assess various of the project to see whether any changes or expansions could have benefited the project. Moreover, there are additional components which if added could have positively contributed to the project.

- **Experiment Runtimes**
  A big problem with the project was the long runtime of the experiments. The cumulative and average runtime of all experiments was 22 days, 44 minutes, 20 seconds and 1 day, 11 hours, 14 minutes and 57 seconds respectively. As a consequence, each experiment could only be run once which means that very little data could be collected. Averaging the results of multiple runs would have given a more definitive representation of the effect that sequential learning and hierarchical algorithms have on the various agents' learning abilities. With a single run per experiment however, the collected data is less reliable unfortunately. More experiments could potentially have given answers to questions like why the DQN outperforms the Option Critic and FuN on the Steps + Pipe level or why the Option Critic tends to get stuck in local optima. Additionally, the excessively long runtimes prevented discovering good values for hyperparameters which would have required further experiments. There are a number of reasons why experiments ran slow, some of which are outlined below.

- **Emulator Development**
  The Java framework used for the emulator did not meet the project requirements right out of the box. It was missing endpoints for the Python agents, classes for extracting the frames from the game as well as preprocessing, and methods for playing the game in a step by step manner. Although necessary for the success of the project, developing all of these functionalities took away valuable time from running experiments because finishing the emulator was a prerequisite for being able to run the entire system.

- **Slow Game Emulator Speed**
  Choosing the less popular *marioai* Java version over the Python *openaigym* version as a game emulator allowed for the easy creation of custom levels, however came at the cost of decreased emulator speed. First, the communication overhead from the Py4J bridge slowed down the agents because they had to wait for a response to be marshalled back to them. Second, the Java emulator had a ceiling for the maximum number of frames per second (fps). Thus, even with more powerful hardware the agents were constrained to a set speed and learning was slowed down.

- **Additional Levels**
  The experiments showed that sequential hierarchical models are quicker to adopt to a new environments than non-sequential traditional models. To further solidify this notion it would be desirable to train the agents in different levels. This would be to make sure that the difference in performance is not due to the level environment but the agents themselves.

- **Exploration Function**
  All agents used a linear function to calculate the exploration value ϵ (see section 4.4). In multiple experiments, the Option Critic got stuck in local optima. While it is difficult to pin down the exact reason, trialing a non-linear exploration function such as the sinusoidal function [**?**] which continously moves between more exploration to more exploitation over the course of an experiment could have produced different results.

- **Metric for MDP Distance**
  An important thing to know when transferring knowledge from one environment to the next is how closely related they are. Figuring out the difference between the two environments makes it possible to assess the learning abilities in a more scientific manner. Using environments which are too similar might lead to an effect where agents perform well in the second environment because of chance and not because they learned something in the first one. Metrics such as the Hausdorff or Kantorovich metric which can be used to measure the distance between MDPs [**?**] could have been implemented to determine the difference between levels in Super Mario and construct levels of varying degree.

- **Egocentric Learning**
  The allocentric state representation for the MDP as an 84×84 RGB image could have been substituted for a lower-dimensional egocentric view of the game. In allocentric learning, the agent sees the entire environment (screen) whereas an egocentric agent only sees Mario's immediate surroundings. Using an egocentric view of the environment, Mario will only take into account the objects that are close to him when learning [**?**]. This means that many more of the states in the state space are similar and will therefore invoke a similar response from the agent. The state space reduction means fewer states for the agent to learn and therefore a reduced experiment runtime.

- **Visualise Options**
  An interesting addition which would have helped with the analysis of the various algorithms is the visualisation of the hierarchies learned by the agents. The FuN and Option Critic agents both learned temporally abstract behaviour in the various levels, however just looking at the average reward episode does not tell when and where the agent used that behaviour. Additionally, just because the agent learned hierarchies does not mean that these hierarchies are useful. Therefore, showing which hierarchies the agents learned could have been useful in explaining the obtained results.

9 Conclusion

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.