Reinforcement Learning and Transfer Learning Between Games

Student Name: Matthew Chapman

Supervisor Name: Dr Lawrence Mitchell

Submitted as part of the degree of BSc Natural Sciences to the

Board of Examiners in the Department of Computer Sciences, Durham University

May 5, 2021

Abstract — **Context**: In recent years, researchers have developed algorithms that perform better than humans at solving certain reinforcement learning (RL) problems. As more RL algorithms are used in the real world, it is important to ensure that these algorithms are robust, which may be achieved by transfer learning (TL).

Aims: The project aims to develop a RL algorithm to learn to successfully play the Atari game Breakout. Additionally, the project aims to investigate TL approaches to make learning to play other Atari games more efficient — although this was not able to be achieved.

Method: The method was to study implementations of existing RL algorithms from the literature and use it to solve the cart-pole problem. Then, the algorithm would be adapted by introducing a deep convolutional neural network architecture to learn from pixel inputs.

Results: The best agent our algorithm trained achieved a mean score of 10.24 and a high score of 26 after 15,060 episodes of training (or 2,082,627 steps and 72,347 policy updates). As is, this falls short of human performance, but our agent shows signs of surpassing this with more training.

Conclusions: We found that our implementation was sensitive to hyperparameters, and required many experiments to be conducted to find the best hyperparameters for training. Our implementation shows that the proximal policy optimisation algorithm may be simplified while still keeping convergence properties.

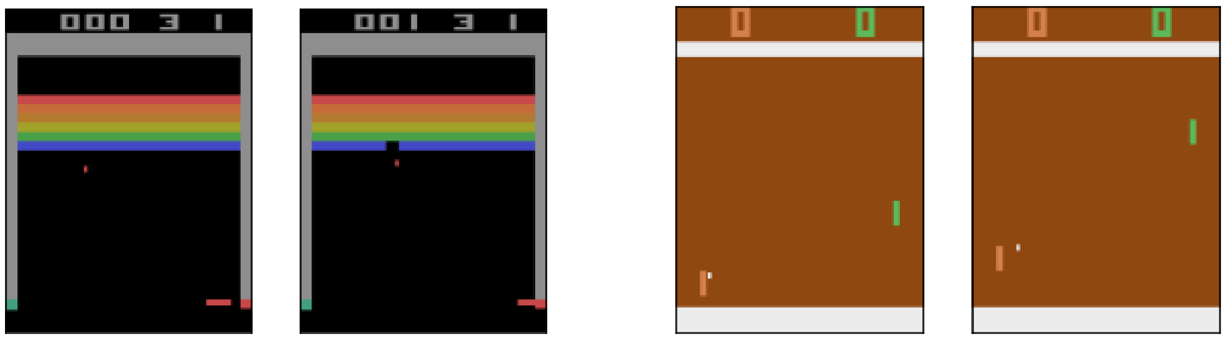
Keywords — Reinforcement learning, deep learning, transfer learning, game-playing.

I INTRODUCTION

A Background

Reinforcement learning (RL) studies how agents learn behaviour in order to maximise a numerical reward signal (Sutton & Barto 2018). An example RL problem is a player (the agent) learning the best strategy (the behaviour) to get a high score (the numerical reward signal) in a game. Often, RL requires the agent to learn through trial-and-error interactions with a dynamic environment (Kaelbling et al. 1996). As in the example, the player has to learn by trying different strategies (the trial-and-error interactions) and observing how the game changes (the dynamic environment). We consider the agent to be successful at solving a RL problem once the agent has learned behaviour that maximises the reward signal consistently over consecutive attempts. In the game scenario, this is when the player gets a high score on every playthrough.

Transfer learning (TL) studies how knowledge gained while solving one problem can be applied to solve a different, but related, problem (Sammut & Webb 2010). Two RL problems that are related are an agent learning to play the game Breakout and an agent learning to play the game Pong. In Breakout, the player must repeatedly bounce a ball off a paddle to destroy bricks; whereas, in Pong, the player must hit a ball back and forth with a paddle to beat an opponent. It should be possible to apply knowledge gained while learning one game to learn the other, since both games require controlling a paddle to intercept a moving ball. In this case, we consider knowledge to have been successfully transferred if an agent with TL learns to play the same game more efficiently than an agent without TL.



(a) Breakout. (b) Pong.

Figure 1: Frames from two *Atari 2600* games.

B Context

In recent years, researchers have developed algorithms that perform better than humans at solving certain RL problems. Perhaps the most famous example is AlphaZero, an algorithm developed by DeepMind that beat world champions at the games of chess, shogi, and Go (Silver et al. 2018). The RL algorithm learns by playing millions of games to try to maximise the number of games won. This is possible to do in far less time than it takes for a human thanks to the algorithm's architecture, which is a deep neural network (DNN). The numerous parameters of a DNN make it so that learning is equivalent to training the network, which is to update its parameters based on interactions had so far. Currently, most state-of-the-art RL algorithms use DNN architectures; these are called deep reinforcement learning (DRL) algorithms.

As RL algorithms are applied to solve problems that are more real-world, such as driving (You et al. 2017), it is important to ensure that these algorithms are robust, which is to say that they are likely to be successful. TL can be useful for achieving robustness in cases where it is expensive for the RL agent to learn directly from the real world. For example, training a self-driving car from scratch will likely lead to wreckage, thus preventing further training. Instead, the car should be trained in a simulator, and then the knowledge gained should be transferred so that the car performs well when tested in the real world. Some TL approaches have been developed that make use of the RL agent's existing architecture (Parisotto et al. 2016), while others propose entirely new architectures (Rusu et al. 2016).

C Aims

The research question proposed is as follows: *How much more efficiently does a reinforcement learning agent with transfer learning learn to play a different Atari game than an agent* *without transfer learning?*. To address this research question, the objectives for the project were divided into three categories: minimum, intermediate, and advanced.

The minimum objectives were to identify and implement a DRL algorithm from the literature which will learn to successfully play Atari games. The algorithm should be used to train an agent to play Breakout, and is expected to perform better than a human by achieving an average score of at least 31.8 (Mnih et al. 2015). The purpose of this objective was to learn about existing RL algorithms and build a foundational model for the remainder of the project.

The intermediate objective was to investigate the approach of TL via transferring network parameters. The implemented DRL algorithm should be used to train two agents to play Pong. However, the network parameters of one of the agents should be initialised to be the same as the parameters that were learned by the Breakout agent. The agent with TL is expected to perform better than the agent without TL by achieving a higher average score after the same amount of training time, as was the case in (Parisotto et al. 2016). The purpose of this objective was to assess the TL capability of our implemented DRL agent's existing architecture.

The advanced objective was to investigate the approach of TL via training a generative model, which requires developing a new architecture. The implemented RL algorithm should be used to train 11 agents for 11 Atari games, and gameplay from 10 of these agents should be recorded and used to train the model. Then, the model should be used to predict the actions of the 11th agent from gameplay. The more actions the model correctly predicts, the better it is considered to be. The purpose of this objective was to extend the current state of the art of TL approaches, by providing a solution to one of OpenAI's *unsolved problems*[1].

D Achievements

We achieved the minimum objective of the project, which was the anticipated outcome of the original project proposal, *Reinforcement Learning*[2]. We started by identifying a number of RL algorithms from the literature and developing a theoretical understanding of the different methods. We studied implementations of the Q-learning, deep Q-network, REINFORCE, and proximal policy optimisation (PPO) algorithms (all of which we discuss in the next section), and used them to solve the simple cart-pole problem (Sutton & Barto 2018), where the agent is required to move a cart to keep a pole hinged to the cart from falling over. We then implemented our own PPO algorithm which uses a deep convolutional neural network (DCNN) architecture to learn to play Breakout from pixel inputs.

II RELATED WORK

Many academic papers have been published introducing RL algorithms that learn to perform successfully in some environment. The two categories of algorithms we focus on are value-based methods and policy gradient methods. Before discussing these methods, we introduce the terminology, as written by (Achiam 2018), commonly used in the literature to abstract RL problems in a way that can be solved algorithmically.

[1]openai.com/blog/requests-for-research-2/

[2](proposed by Professor Magnus Bordewich)

A Terminology

As explained in the introduction, RL algorithms are agents that interact with an environment and learn behaviour to maximise a reward signal. This process can be broken down into steps of interaction, where each interaction consists of an observation and an action. At time step t, the agent makes an observation of the environment's current *state* s_t, which is a complete description of the environment. An example of s_t is a game frame (such as in Figure 1) — or, more precisely, the matrix of numbers that represent each pixel and its red, green, and blue (RGB) values. Based on s_t, the agent takes an *action* a_t in the environment. The set of actions allowed in an environment is called the *action space*. An example of a_t for a Breakout agent is 'LEFT', which moves the paddle left. The set of rules that determine how an agent selects a_t from s_t is called its *policy* π. After a_t is taken, the next state s_{t+1} must be observed and action a_{t+1} taken, and so on.

At the beginning of each step t, the agent also receives an immediate *reward* r_t from the environment. For example, $r_t = +1$ in the step immediately after a brick is destroyed in Breakout, since this represents a point earned in the game. After many steps, the total reward received is known as the *return*. One form of return is the *finite-horizon undiscounted return* $R(\tau) = \sum_{t=0}^{T} r_t$, which is the sum of rewards obtained after T steps, where τ is the sequence of states and actions called the *trajectory*. Another form of return is the *infinite-horizon discounted return* $R(\tau) = \sum_{t=0}^{\infty} \gamma^t r_t$, which is the sum of all rewards ever obtained by the agent, discounted by how far off in the future they are obtained. The *discount factor* $\gamma \in (0, 1)$ determines how important future rewards are to the agent. The goal in RL is this: to find the policy which maximises expected return, known as the *optimal* policy.

B Value-based methods

Value-based methods aim to find the optimal policy by building value functions. The *value function*, $V^\pi(s)$, gives the expected return starting from state s and following policy π thereafter, which is the map $V^\pi(s) : \mathcal{S} \to \mathbb{R}$. And, the *action-value function*, $Q^\pi(s, a)$, gives the expected return starting from state s, taking action a, and following policy π thereafter, which is the map $Q^\pi(s, a) : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$. From this latter function, also known as the *Q-value function*, we can derive one of the foundational value-based algorithms, the Q-learning algorithm (Watkins & Dayan 1992).

The goal of Q-learning is to learn an optimal Q-value function, $Q^*(s, a)$, which is the same as $Q^\pi(s, a)$ except the policy is optimal. Given $Q^*(s, a)$, the optimal policy is to always select the action with the highest Q-value at each state. The basic version of Q-learning does this by keeping a lookup table of Q-values with one entry for every *state-action pair* (s, a) (François-Lavet et al. 2018). These Q-values can be found optimally by solving the Bellman equation (Bellman & Dreyfus 1962)

$$Q^*(s, a) = \mathbb{E}\left[r(s, a) + \gamma \max_{a'} Q^*(s', a')\right]$$

which recursively relates each state-action pair, where r is the immediate reward, γ the discount factor, s' the next state, and a' the action from s'. For some environments, however, solving the Bellman equation is infeasible. Instead, the optimal Q-values must be estimated.

Fitted Q-learning (Gordon 1995) estimates the optimal Q-values based on experience gathered from interactions so far, and iteratively updates them. Neural fitted Q-learning (NFQ)

(Riedmiller 2005) improves on this by parametrising the Q-values with a neural network, which computes more efficiently. The Q-network takes as input a state and outputs different Q-values for each of the possible actions. The breakthrough deep Q-network (DQN) algorithm (Mnih et al. 2015) uses a DCNN architecture to take pixels as input to learn to play Atari games. More recent approaches made improvements to DQN: duelling DQN (Wang et al. 2015) helps generalise learning across actions; double DQN (van Hasselt et al. 2015) reduces overestimating action values; and, rainbow DQN (Hessel et al. 2017) combines these improvements to achieve even better performance.

Despite these improvements, there remain limitations with value-based methods. Primarily, these methods poorly handle environments with large action spaces. In the Atari game Gravitar which has 18 actions, a DQN agent achieves a mean score of 306.7 compared to 2672 achieved by a human (Mnih et al. 2015). Another key limitation is that these methods cannot explicitly learn stochastic policies, which assign a probability of being selected to each action from a state, while policy gradient methods can.

C Policy gradient methods

Policy gradient methods aim to find the optimal policy by defining the policy with an objective function and maximising this function. This is done by calculating the gradient — known as the policy gradient — and using it to update the policy. Given the objective function

$$J(\pi_\theta) = \mathop{\mathbb{E}}_{\tau\sim\pi_\theta}[R(\tau)]$$

to define the policy π_θ with parameters θ, where $\tau \sim \pi_\theta$ implies a trajectory obtained by following π_θ, differentiating the function yields the following fundamental result:

$$\nabla_\theta J(\pi_\theta) = \mathop{\mathbb{E}}_{\tau\sim\pi_\theta}\left[\sum_{t=0}^{T} \nabla_\theta \log \pi_\theta(a_t|s_t) R(\tau)\right].$$

To estimate this policy gradient, we must gather a trajectory, or experience, by following the current policy in the environment.

The simplest estimator is a sample mean over the experience, since the expression for the policy gradient is an expectation. It turns out that using either the finite-horizon undiscounted return or the infinite-horizon discounted return yields similar performance. This is known as the REINFORCE algorithm (Williams 1992). A variant of the algorithm shows that the policy gradient can also be expressed using *reward-to-go* $\hat{R}_t = \sum_{t'=t}^{T} R(s_{t'}, a_{t'}, s_{t'+1})$ instead of $R(\tau)$. This variant is an improvement because it reduces variance while keeping the expectation the same (Achiam 2018).

Actor-critic algorithms (Konda & Tsitsiklis 2001) build on the discovery that the policy gradient can be re-expressed to improve performance, and learn a value function (the critic) alongside the policy (the actor). Asynchronous advantage actor-critic (A3C) (Mnih et al. 2016) calculates an advantage function, which describes how much better or worse an action is than others on average, and trains actors and critics in parallel to stabilise training. Trust region policy optimisation (TRPO) (Schulman et al. 2015) similarly tries to stabilise policy performance by taking the largest possible update that satisfies a KL-divergence constraint (Kullback & Leibler 1951). Due to the complexity of TRPO, however, the simpler proximal policy optimisation (PPO) (Schulman et al. 2017) was developed, which performs just as well without having to make second-order calculations. As with DQN, the more recent methods use DNNs as function approximators for the policy and value functions.

Most of the limitations of policy gradient methods concern performance collapse, which is when the algorithm suddenly performs worse during training or ceases to learn. This occurs when the objective function gets trapped in a local optimum, and further updates cannot increase its value. This equates to a sub-optimal policy. Performance collapse occurs because the policy becomes less random after lots of training, to *exploit* rewards that have already been found, so it becomes improbable for the agent to *explore* new behaviour that might lead to more rewards.

III SOLUTION

We implemented the proximal policy optimisation (PPO) algorithm (Schulman et al. 2017) and base our design on an implementation by Seungeun Rho[3]. PPO is a policy gradient method that tries to improve the policy by as much as possible during each policy update without collapsing performance.

A Specification

To develop a solution that lets a RL algorithm learn to play Atari games successfully, the solution must satisfy certain requirements. First, the solution should be able to load an Atari game and reset the environment and agent to an initial game state. Second, the solution should change the game state only when the agent takes an action — i.e. at each step. At time step 0, the game state is the initial game state; and, at time step 1, the game state is the result of the first action, and so on. Third, each step should give the new game state alongside additional information, including the reward from taking the previous action, whether the game has finished, and the number of lives the agent has left. Fourth, the game states and additional information should be able to be collected, pre-processed, and used by a RL algorithm to control the agent and iteratively improve its behaviour.

B Tools used

Due to the need for a common benchmarking system in RL research, the first three requirements have already been implemented by Gym (Brockman et al. 2016), an open-source library by OpenAI for developing and comparing RL algorithms. Gym provides a collection of environments, including control theory problems, physics simulations, and Atari 2600 games. This lets us focus on the fourth requirement, which is to design and implement our algorithm and pre-processing pipeline.

For the solution, we used the Python programming language. This is because Python has numerous open-source machine learning libraries with support for deep learning and reinforcement learning. Examples of libraries are Keras, PyTorch, and TensorFlow. Of these, we used PyTorch (Paszke et al. 2019), since this is currently what most researchers are using to implement their state-of-the-art papers. We also used the popular OpenCV (Bradski 2000) open-source computer vision library for image pre-processing.

[3]https://github.com/seungeunrho/minimalRL

Since much computing power is needed for agent-training, we make use of Colab (Bisong 2019). Colab is a development environment by Google that allows Python code to be written and executed in the browser. More importantly, Colab provides free access to GPUs and has support for CUDA, which we can use to speed up training. There was the option to use Durham University's NVIDIA CUDA Centre (NCC); although, we chose not to due to having no prior remote server experience or job queuing system experience (such as SLURM). Some training was also carried out on our local machine's CPUs, which has an Apple M1 chip and 8GB of RAM.

C Design

C.1 Proximal policy optimisation (PPO)

We implemented the most widely used version of PPO, PPO-Clip, which utilises clipping in the objective function to keep the new policy close to the old. Given policy π_{θ_k} with parameters θ_k after k policy updates, the algorithm updates its policy parameters in the next iteration by

$$\theta_{k+1} = \arg\max_{\theta} \underset{s,a\sim\pi_{\theta_k}}{\mathbb{E}} \left[L(s,a,\theta_k,\theta) \right],$$

where L denotes the objective function, $s, a \sim \pi_{\theta_k}$ implies states and actions obtained by following policy π_{θ_k}, and θ is the new set of policy parameters. The objective function L is the minimum between two terms, and is calculated by

$$L(s,a,\theta_k,\theta) = \min\left(\frac{\pi_\theta(a|s)}{\pi_{\theta_k}(a|s)} A^{\pi_{\theta_k}}(s,a),\ \ \text{clip}\left(\frac{\pi_\theta(a|s)}{\pi_{\theta_k}(a|s)}, 1-\epsilon, 1+\epsilon \right) A^{\pi_{\theta_k}}(s,a) \right),$$

for small ϵ, where π_θ is the new policy and $A^{\pi_{\theta_k}}$ is the advantage function with respect to the old policy π_{θ_k}. The ratio $\frac{\pi_\theta(a|s)}{\pi_{\theta_k}(a|s)}$ is the probability ratio between new and old policies, and is kept in the range $[1-\epsilon, 1+\epsilon]$ by the 'clip' term to try to prevent performance collapse. If the ratio is too low, it is replaced by $1-\epsilon$; and, if the ratio is too large, it is replaced by $1+\epsilon$.

For better intuition, when the advantage for the state-action pair is positive, the objective becomes

$$L(s,a,\theta_k,\theta) = \min\left(\frac{\pi_\theta(a|s)}{\pi_{\theta_k}(a|s)}, (1+\epsilon) \right) A^{\pi_{\theta_k}}(s,a).$$

Whereas, when the advantage for the state-action pair is negative, the objective becomes

$$L(s,a,\theta_k,\theta) = \max\left(\frac{\pi_\theta(a|s)}{\pi_{\theta_k}(a|s)}, (1-\epsilon) \right) A^{\pi_{\theta_k}}(s,a).$$

A positive advantage will increase the objective function if the action becomes more likely — i.e., $\pi_\theta(a|s)$ increases — and a negative advantage will increase the objective function if the action becomes less likely — i.e., $\pi_\theta(a|s)$ decreases (Achiam 2018).

C.2 Generalised advantage estimation (GAE)

To estimate the advantages for the objective function, we use generalised advantage estimation (GAE) (Schulman et al. 2018). To get the estimate $\hat{A}_t$ of $A^{\pi_{\theta_k}}$ at step t, where $t = 0, 1, \ldots, T$ are the steps leading up to a policy update after time step T, we use the following equation:

$$\hat{A}_t = \delta_t + (\gamma\lambda)\delta_{t+1} + \cdots + \cdots + (\gamma\lambda)^{T-t-1}\delta_{T-1},$$

where $\delta_t = r_t + \gamma V(s_{t+1}) - V(s_t)$, γ is the discount factor we need to set, λ the GAE parameter we also need to set, and V is the most recent estimate of the value function.

C.3 Stochastic gradient ascent/descent (SGA/SGD)

As shown, to use GAE and hence perform the policy update, experience must be collected. Our algorithm does this by playing the game for T steps using the current policy. The result is a *batch* of experience where each item in the batch is a tuple of information representing the step. Each tuple consists of the following: the state before an action, the action taken, the reward from taking the action, the state after the action, the probability of taking the action given the policy, and whether or not the game is done.

With this batch, we can estimate the policy gradient and update the policy by stochastic gradient ascent (SGA) — since our goal is to maximise the policy. If the whole batch is used to calculate the gradient estimate, then this is *batch* SGA, and only one update is performed. If the batch is divided into equal-sized portions and each is used to calculate a gradient estimate, then this is *mini-batch* SGA, and the number of updates performed is equal to the number of mini-batches. We must also update the value function in a similar manner. Although, instead of SGA, we perform stochastic gradient descent (SGD) since we are aiming to minimise the loss of the value function.

In our implementation, we perform batch SGA and SGD to optimise the policy using the Adam algorithm (Kingma & Ba 2017). Additionally, we repeat the update for a specified number of epochs, K, which results in bigger updates.

C.4 Deep convolutional neural network (DCNN)

So that our algorithm can learn from pixel inputs, we use a DCNN architecture based on (Mnih et al. 2015) to estimate the policy and value function. The policy takes as input the current state and returns a categorical distribution, which is a discrete probability distribution. The number of categories is equal to the size of the action space, and each category has a probability and corresponding action. We must sample from the distribution to obtain an action, such that actions with high probabilities are more likely to be sampled than actions with low probabilities. Necessarily, the probabilities must sum up to 1 overall. On the other hand, the value function takes as input the current state, and outputs the estimated value of that state.

DCNN have convolutional layers followed by fully-connected layers. The convolutional layers apply convolutions to an input, often an image, which is when a filter is applied which gives an activation. Applying the filter many times across the input creates a feature map, which can highlight features such as shapes and edges. The fully-connected layers further process the flattened output of the convolutional layers by combining information across the filters.

In our architecture, the policy network and value network share convolutional layers and have separated fully-connected layers. The reason for this is because the features identified that are useful to improve the policy should also be useful to improve the value function, or vice versa. The fully-connected layers are there to perform further processing in a manner specific to estimating the policy or value function.

The DCNN consists of 3 blocks of alternating convolutional layers and ReLU activation functions. The first block has kernel size 8, stride 4, and takes as input a stack of 4 frames of dimension 84×84 and outputs 32 20×20 frames. The second block has kernel size 4, stride 2, and outputs 64 9×9 frames. The third block has kernel size 3, stride 1, and outputs 64 7×7 frames. The fully-connected layers each consist of 2 blocks of linear layers and ReLU activation functions. The first block takes as input the flattened output of the convolutional layers, of dimension 3136, and the second block takes as input 512. For the policy network, the outputs of the second block are passed through a softmax function to obtain action probabilities.

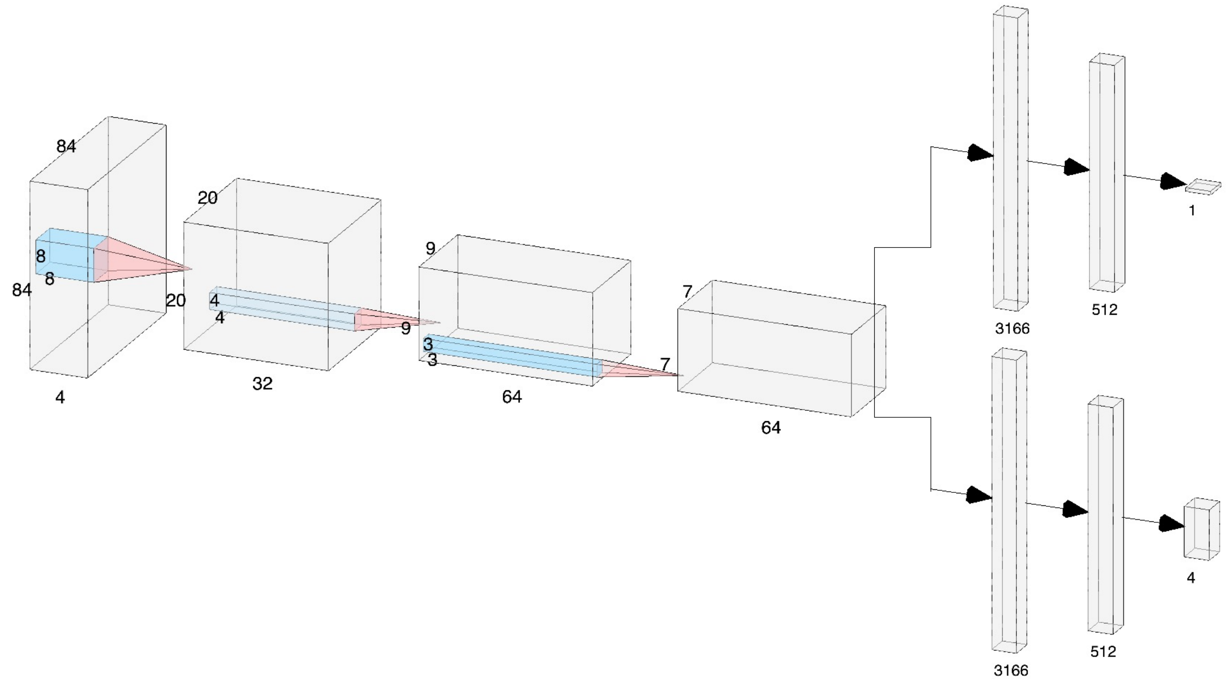


Figure 2: Diagram of our implementation's DCNN architecture[4]. The top branch belongs to the value network, while the lower branch belongs to the policy network. Here, the policy is shown to have 4 outputs, which is the size of Breakout's action space. This number depends on the environment.

D Implementation issues

D.1 State definition

The first implementation issue considers how to define the state to be used as input to the DCNN at each step. Simply using the game state is insufficient. A game state by itself is a singular, static image from which we cannot infer any information about what is moving. Conveying motion to the neural network is important, since it must understand the velocity of the ball and predict its trajectory for the agent to perform successfully. To tackle this issue, we define the state at each step to be a stack of the last 4 game states, or *frames*. The fourth frame is the most recent — the current game state — while the first is the game state 3 actions (or steps) earlier. From this, we expect the DCNN to learn filters that detect the changed positions between frames to infer motion, and take actions accordingly.

[4]made using the online tool 'NN-SVG' (github.com/alexlenail/NN-SVG)

An adjustment we made to make conveying motion more obvious, which is used in (Mnih et al. 2015) is to perform *frame-skipping*. Instead of taking 1 action between each frame, we take 4 of the same actions. This means that the fourth frame is actually 12 actions ahead of the first.

D.2 Pre-processing

The second implementation issue considers how to speed up network training without sacrificing performance. Applying convolutions is computationally expensive, more so the larger the image, and the time spent doing so adds up. The game state is an image of height, width, channel $210 \times 160 \times 3$; however, some of this information can be discarded or compressed. Since colour plays no important role in predicting the movement of the ball, we can convert the image to greyscale and obtain a new image with dimension 210×160. Similarly, the visual indicators for the score and number of lives left are redundant since we also keep track of these values in the algorithm. We can crop these indicators out, producing a new image with dimension 160×160. Additionally, if the ball and paddle can both be identified at lower resolutions, then there should be no loss of information, and in theory the performance of the DCNN should be the same. As a result, we down sample the image, which changes the number of pixels, to obtain a new 84×84 image.

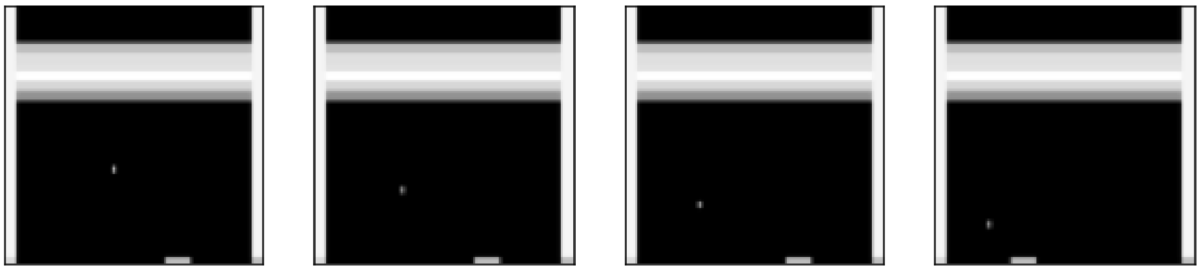


Figure 3: A stack of 4 pre-processed frames from Breakout.

D.3 Huber loss

The third implementation issues consider how to update shared network parameters during an update step. Due to the policy and value networks sharing convolutional layers, we must add an additional term to the objective function which accounts for the value function loss (or error). The additional term we use is the Huber loss function, which calculates the loss between batches (size n) of x and y by

$$\text{loss}(x, y) = \frac{1}{n}\sum_i z_i,$$

where z_i is given by the following:

$$z_i = \begin{cases} \frac{1}{2}(x_i - y_i)^2, & \text{if } |x_i - y_i| < 1 \\ |x_i - y_i| - \frac{1}{2}, & \text{otherwise .} \end{cases}$$

In the implementation, x is the value of the current state, and y the immediate reward plus the discounted value of the next state. This loss must be subtracted from the policy gradient in the objective function, since we are trying to minimise it.

E Verification and Validation

Verification was done on the solution to check that the specification was met. During developmental stages, verification methods consisted of running individual components of the system followed by checking the results from the run. We ran each frame pre-processing step, the convolutional layer, and both fully-connected layers, and verified the outputs were as expected. These runs ensured that the specification continued to be met once the system was formed.

Validation was done to ensure that the solution would meet the project objectives. Validation methods involved discussing with the project advisor and predicting barriers that could lead to insufficient completion of the solution. This led to the updating of project objectives which were used as guidance for planning. Another validation method was to refer back to the original project proposal, *Reinforcement Learning*, from which this project has evolved. By doing validation, we establish evidence that the solution achieved at least the intended minimum objectives.

F Testing

Unit testing, integration testing, and system testing were all done on the solution. Unit testing was conducted by breaking up each component of the system into its individual functions and testing them. For example, we tested that the output probabilities of the softmax function should sum to 1. We then conducted integration testing by running functions together. For example, we tested that the output probabilities of the policy network were the same when the input was kept the same. Finally, we conducted system testing by running the system as a whole. For example, we checked that the total reward printed at the end of a playthrough, called an *episode*, matched the score that was shown at the end of the corresponding recorded video.

IV RESULTS

A Evaluation method

The evaluation method adopted was to produce and analyse the following statistics during agent training: episode score, episode score mean and standard deviation — calculated over the last 100 episodes — objective function loss mean and standard deviation, episode length, as well as total number of steps and learning iterations. Additionally, we record a video of an episode at regular intervals to directly observe and better understand agent behaviour.

To determine to what extent our implementation of the PPO algorithm learns to successfully play Atari games, we trained agents in the Breakout environment provided by Gym. By training each agent for a large number of episodes, we can assess whether the agent will converge to a high score and how much luck is involved. We planned to train each agent for 1,000 episodes initially and evaluate its performance at every 100th episode, by the evaluation method described. If the agent showed signs of improving score, we continued training the agent for an additional 1,000 steps, and the process repeated.

B Experimental settings

We discovered that our implementation was susceptible to performance loss, even early in training, and as a result conducted several experiments using different settings in an effort to improve performance. We started by using hyperparameters that were identical, where possible, to those stated in the paper by OpenAI introducing the PPO algorithm (Schulman et al. 2017).

Table 1: PPO hyperparameters used by OpenAI and in our initial training. Differences are highlighted in bold. α is linearly annealed from 1 to 0 over the course of learning. The horizon T is the number of steps to take before performing a policy update. The Adam step size affects how the Adam algorithm optimises our objective function by SGA/SGD.

(a) OpenAI's hyperparameters.

Hyperparameter	Value
Horizon T	128
Adam step size	$\mathbf{2.5 \times 10^{-4}} \times \alpha$
Num. epochs	3
Mini-batch size	$\mathbf{32 \times 8}$
Discount γ	0.99
GAE parameter λ	0.95
Number of actors	$\mathbf{8}$
Clipping parameter ϵ	$\mathbf{0.1} \times \alpha$
VF coeff. c_1	1
Entropy coeff. c_2	$\mathbf{0.01}$

(b) Our hyperparameters.

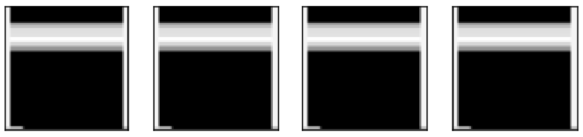
Hyperparameter	Value
Horizon T	128
Adam step size	$\mathbf{2.5 \times 10^{-4}}$
Num. epochs	3
Mini-batch size	**—**
Discount γ	0.99
GAE parameter λ	0.95
Number of actors	$\mathbf{1}$
Clipping parameter ϵ	$\mathbf{0.1}$
VF coeff. c_1	1
Entropy coeff. c_2	**—**

To address the concern about reproducing results, we take steps to control sources of randomness that will cause our agent to learn differently across different runs. The steps we take are to seed the random number generation (RNG) for each module we use. We seed 'torch', 'random', 'numpy', and the environment and action space for 'gym'. The seed value we used was 742.

C Experiments

First, we trained an agent using the hyperparameters stated in Table 1b. Disappointingly, the result was that the agent stopped learning within the first 10 episodes of training. Looking at the statistics, we see that from episode 4 the final score of each episode was 0. Additionally, we see that from episode 7 the mean and standard deviation of the loss at each episode was also 0, which means that policy updates no longer affected the policy. However, each episode length was exactly 2,500 steps. Looking at the latest video, we observe that the agent immediately moves the paddle to the left wall, without firing the ball, and remains there until the episode is forcibly terminated. We confirm this by loading the saved model and inspecting a state input and corresponding action-probabilities produced by the policy network: the 'LEFT' action had a 0.99 probability. We hypothesise that this performance collapse was due to the policy becoming trapped in a local optimum, and conduct further experiments to try and overcome this problem.

Second, we trained using different values for the Adam step size. The reason for this is to assess the hypothesis that smaller gradient updates might prevent performance collapse, since the policy may be less likely to get trapped in a local optimum. For step size 1.5×10^{-4}, the result was that performance collapse occurred by episode 3, which is even earlier than before. And, for step size 0.5×10^{-4}, the result was that performance collapse occurred by episode 29, which is later than before. This time, from looking at the latest video of the latter agent, we see that the action probabilities were heavily weighted towards the 'RIGHT' action.



(a) Input state.

NOOP	FIRE	RIGHT	LEFT
0.0001	0.0003	0.0009	0.9985

(b) Output action-probabilities.

Figure 4: The input to the policy network and corresponding output by the agent in the first experiment. As can be seen, the paddle is by the left wall in all 4 frames. ‘NOOP’ is short for ‘no operation’, and ‘FIRE’ is the action that initialises the ball by firing it towards the paddle.

Third, we trained using different values for the horizon T. The reason for this is to assess the hypothesis that estimating the policy gradient using smaller batch sizes might prevent performance collapse, since there may be less varied experience. For $T = 64$, the result was that the agent achieved a mean score of 2.24 after 1,000 episodes (or 80,496 steps and 2,044 policy updates). However, looking at the statistics, we noted that this mean score was not improving over time even though the mean loss was non-zero. Looking at the latest video, we observed that the agent moves to the right wall and remains there, but knows to fire the ball. The mean score is due to the ball hitting the paddle and destroying bricks by chance. Although this policy is better, it shows no sign of improving if we were to train it for longer. For $T = 32$, the result was that the agent behaved similarly to $T = 64$ to begin with, and then performance collapsed occurred after episode 265 in the same way as it did in the first and second experiments.

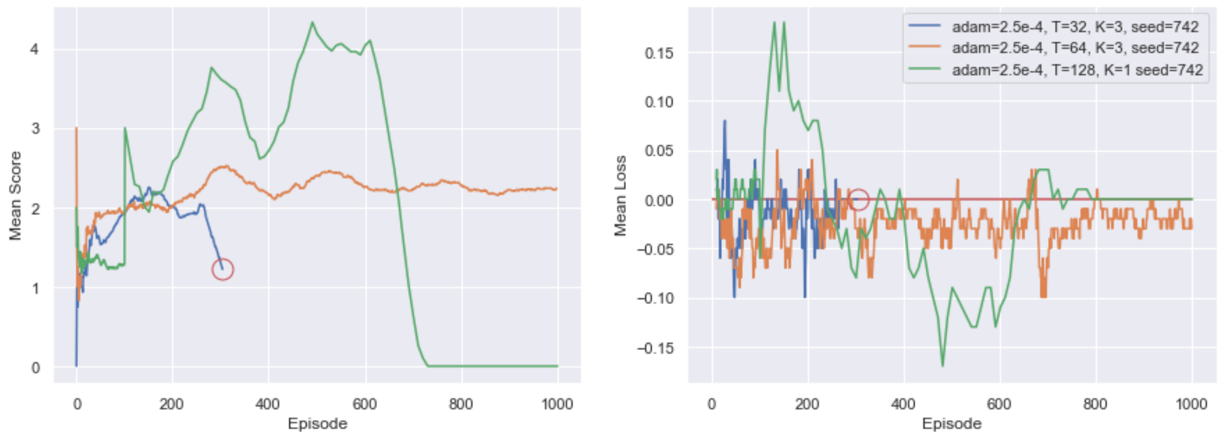


Figure 5: Graphs of mean score and mean loss against episode for some of the experiments. The red circles indicate where training was stopped early due to performance collapse, to save time since each episode was taking a minute to compute. The red line in the second graph is a line at mean loss 0.00.

Fourth we trained using different values for the number of epochs K. The reason for this is to assess the hypothesis that decreasing the number of updates to the policy might help to prevent performance collapse, since, again, the policy may be less likely to get trapped in a local optimum. For $K = 2$, the result was that performance collapse occurred by episode 3. For $K = 1$, the result was that the agent scored a high score of 8, had a mean score of 4.10 by episode 610, and performance collapse occurred by episode 630.

Finally, we trained using a combination of the values that improved performance in the previous experiments. For the combination $K = 1$, $T = 64$, and Adam step size 0.5×10^{-4}, the result was that performance collapse occurred after episode 2. However, for the combination $K = 1$, $T = 32$, and Adam step size 0.5×10^{-4}, the result was that there was no performance collapse and by episode 1,000 (or 95,770 steps and 3,484 policy updates), the agent had achieved a mean score of 4.67 and a high score of 9. Inspecting the videos revealed that the agent was beginning to learn the desired behaviour: to predict the ball's trajectory and follow it in order to intercept it. Because of this, we decided to train the agent for an additional 1,000 episodes. The agent improved again, and so we trained it for another 1,000 episodes and so on. Eventually, after, 15,060 episodes of training (or 2,082,627 steps and 72,347 policy updates) the agent had achieved a mean score of 10.24 and a high score of 26.

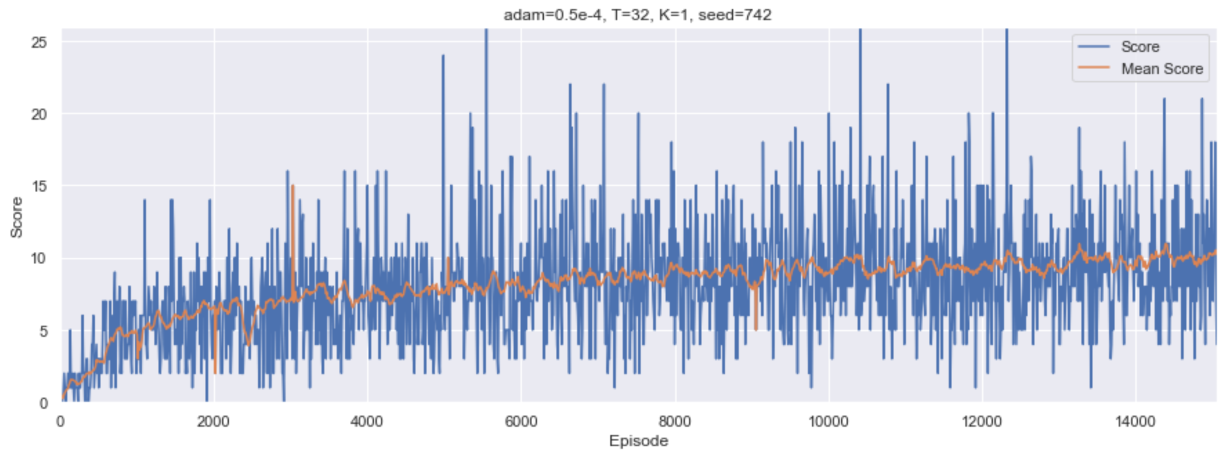


Figure 6: Learning curve of our most successful agent. There are some outliers in the mean score, which occur due to the stopping and starting of training every 1000 episodes. There is a clear upward trend that suggests convergence to a high score, if trained for longer.

V EVALUATION

A System strengths

The first strength of our system is that it is fast. The system is able to process over 200 frames per second, which we calculate by dividing the number of steps per episode by the time taken and multiplying by 4 to take into account the stack of frames. This is made possible due to a number of factors: we used PyTorch, which is optimised for tensor computation on high-dimensional data and supports CUDA, which lets the system take advantage of parallel computing on the CPU and GPU; we incorporated a pre-processing pipeline, that simplifies the raw game state so that the DCNN only processes essential information; and, we implemented a memory-efficient PPO, which discards the batch of experience immediately after the policy update and does not keep a replay-memory of states like DQN. The resulting speed meant that we could identify when performance collapse had occurred, since there would be unusually long episodes, which allowed us to iterate through experiments quicker.

The second strength of our system is that the RL algorithm learns a stochastic policy. This allows our agent to explore states naturally, due to taking actions that are sampled from a probability distribution not necessarily with the highest probability. This is better compared to value-based methods, where we would have had to hard-coded when our agent would take a random action to explore new states.

The third strength of our system is that the design modifies the original PPO algorithm while still showing signs of convergence. Compared to OpenAI's PPO, our system trains 1 actor instead of 8, making it more accessible to run on less powerful hardware. Also, by using Huber loss instead of a mean-squared error loss for the value function, we made the system less sensitive to outliers (Girshick 2015).

B System limitations

One limitation of our system is that it is sensitive to the hyperparameters. As shown by the results, many of the experiments where we varied the hyperparameter values resulted in performance collapse, which is when the agent settles on a sub-optimal policy. The hyperparameters had to be configured in such a way that the policy update rule, which depends on the hyperparameters, would allow the policy to explore enough game-playing strategies without focusing on one too quickly. Although the PPO paper uses the same set of hyperparameters to successfully train agents in other Atari games, it is not entirely unreasonable to suspect that our most successful set of hyperparameters will not lead to performance collapse when tried on another environment. As a result, this makes the system unsuitable as a general purpose system that will learn to play every Atari game.

Another limitation of our system is that the agent is unable to recover once performance collapse has occurred. This is likely caused by the difference between our implementation and OpenAI's. OpenAI's PPO algorithm includes an entropy bonus in the objective function, which measures the entropy of the policy and helps with exploration (Williams & Peng 1991). OpenAI's algorithm also anneals the Adam step size and clipping parameter over time, which we do manually, to encourage exploitation of rewards already found. Moreover, OpenAI's PPO uses early stopping to prevent gradient updates if the mean KL-divergence of the new policy from the old grows beyond a threshold (Schulman et al. 2017). Finally, we did batch updates rather than mini-batch updates. Had these features been implemented, it could be that our system would be able to successfully train agents under a less strict set of parameter values.

C Approach

The approach to the project was an ambitious one. The layout of the objectives — of first developing a RL algorithm, and then investigating TL approaches — was meant to eliminate the risk of having no solution were TL to have performed poorly. What was overlooked was a situation where getting the RL algorithm to work took up the whole project, so that TL could not even be considered. Having the intermediate and advanced objectives depend on the minimum objective in this way also meant that the solution could not be continually improved over time. This was exacerbated by the fact that the advanced objective proposed the development of an architecture completely different to that of the minimum objective. This layout is akin to a waterfall development approach, whereas an agile approach, with some adjustments to the aims, would have been more appropriate.

The reason why developing the RL algorithm took up the whole portion of the project was because of an overestimate in ability to learn the tools and theory involved. There was a steep learning curve, which could have been amplified by a lack of personal background in machine learning, deep learning, and computer vision. Studying reinforcement learning theory, deep neural networks, convolutional neural networks, PyTorch, and Gym, whilst dealing with issues setting up development environments on both Google Colab and locally took up a lot of time. On top of that, there were long testing cycles for algorithms, which can train for days, and neural networks whose behaviour we could not easily predict or fix. If we were to restart the project, we would focus just on RL, since the work required to complete the TL objectives was worth a whole separate project entirely.

D Project organisation

With this project being the first major piece of individual work that we have undertaken, the proper management of it was essential for success. Early into the project, we prepared a plan; then, over time, we monitored progress against it. The plan changed over time, as we became enthusiastic about making objectives more ambitious, and also as we paid more attention to other parts of the course. During the first term, we allocated 12 hours per week to the project. Much of this time, however, was spent reading and building foundational knowledge. Looking back, more of this time should have been spent implementing. During the second term, when there were other deadlines, less time was spent on the project on average, so an effort was made to catch up during the holiday.

In planning our work, we discussed with our supervisor to identify the deliverables intended to fulfil the corresponding objectives in the original project proposal. Due to the results of our initial investigations, we revised our deliverables so that we could investigate the state of the art. This was done in consultation with and approved by our supervisor; however, with the outcome of the project, we were not able to extend the state of the art, and would therefore have benefited more by remaining with the original project proposal.

VI CONCLUSIONS

In this project, we developed a reinforcement learning algorithm to learn to play the Atari game Breakout. The best agent our algorithm trained achieved a mean score of 10.24 and a high score of 26 after 15,060 episodes of training (or 2,082,627 steps and 72,347 policy updates). Our agent as it is falls short of human performance, which requires a mean score of 31.8, but may reach this milestone with more training, as suggested by the shape of the learning curve. To achieve this, we implemented our own modified proximal policy optimisation algorithm that uses a deep convolutional neural network architecture to learn from pixels, and conducted several experiments to determine the best set of hyperparameters for training. We found that our implementation was sensitive to hyperparameters; however, our results suggest that the original PPO algorithm may be simplified while retaining convergence properties.

Further work can be done on this project be completing the ambitiously set intermediate and advanced objectives. These require training an agent to play Pong, and developing a generative model, such as a generative adversarial network (GAN), to learn from gameplay of several agents.

References

Achiam, J. (2018), ‘Spinning Up in Deep Reinforcement Learning’.

Bellman, R. E. & Dreyfus, S. E. (1962), *Applied Dynamic Programming*, RAND Corporation, Santa Monica, CA.

Bisong, E. (2019), *Google Colaboratory*, Apress, Berkeley, CA, pp. 59–64.

Bradski, G. (2000), 'The OpenCV Library', *Dr. Dobb's Journal of Software Tools* .

Brockman, G., Cheung, V., Pettersson, L., Schneider, J., Schulman, J., Tang, J. & Zaremba, W. (2016), 'Openai gym'.

François-Lavet, V., Henderson, P., Islam, R., Bellemare, M. G. & Pineau, J. (2018), 'An introduction to deep reinforcement learning', *CoRR* **abs/1811.12560**.

Girshick, R. B. (2015), 'Fast R-CNN', *CoRR* **abs/1504.08083**.

Gordon, G. J. (1995), Stable fitted reinforcement learning, *in* 'Proceedings of the 8th International Conference on Neural Information Processing Systems', NIPS'95, MIT Press, Cambridge, MA, USA, p. 1052–1058.

Hessel, M., Modayil, J., van Hasselt, H., Schaul, T., Ostrovski, G., Dabney, W., Horgan, D., Piot, B., Azar, M. G. & Silver, D. (2017), 'Rainbow: Combining improvements in deep reinforcement learning', *CoRR* **abs/1710.02298**.

Kaelbling, L. P., Littman, M. L. & Moore, A. W. (1996), 'Reinforcement learning: A survey', *Journal of Artificial Intelligence Research* **4**, 237–285.

Kingma, D. P. & Ba, J. (2017), 'Adam: A method for stochastic optimization'.

Konda, V. & Tsitsiklis, J. (2001), 'Actor-critic algorithms', *Society for Industrial and Applied Mathematics* **42**.

Kullback, S. & Leibler, R. A. (1951), 'On information and sufficiency', *The annals of mathematical statistics* **22**(1), 79–86.

Mnih, V., Badia, A. P., Mirza, M., Graves, A., Lillicrap, T. P., Harley, T., Silver, D. & Kavukcuoglu, K. (2016), 'Asynchronous methods for deep reinforcement learning'.

Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A. A., Veness, J., Bellemare, M. G., Graves, A., Riedmiller, M., Fidjeland, A. K., Ostrovski, G., Petersen, S., Beattie, C., Sadik, A., Antonoglou, I., King, H., Kumaran, D., Wierstra, D., Legg, S. & Hassabis, D. (2015), 'Human-level control through deep reinforcement learning', *Nature* **518**(7540), 529–533.

Parisotto, E., Ba, J. L. & Salakhutdinov, R. (2016), 'Actor-mimic: Deep multitask and transfer reinforcement learning'.

Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., Killeen, T., Lin, Z., Gimelshein, N., Antiga, L., Desmaison, A., Kopf, A., Yang, E., DeVito, Z., Raison, M., Tejani, A., Chilamkurthy, S., Steiner, B., Fang, L., Bai, J. & Chintala, S. (2019), Pytorch: An imperative style, high-performance deep learning library, *in* H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox & R. Garnett, eds, 'Advances in Neural Information Processing Systems 32', Curran Associates, Inc., pp. 8024–8035.

Riedmiller, M. (2005), Neural fitted q iteration – first experiences with a data efficient neural reinforcement learning method, *in* J. Gama, R. Camacho, P. B. Brazdil, A. M. Jorge & L. Torgo, eds, 'Machine Learning: ECML 2005', Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 317–328.

Rusu, A. A., Rabinowitz, N. C., Desjardins, G., Soyer, H., Kirkpatrick, J., Kavukcuoglu, K., Pascanu, R. & Hadsell, R. (2016), 'Progressive neural networks', *CoRR* **abs/1606.04671**.

Sammut, C. & Webb, G. I., eds (2010), *Encyclopedia of Machine Learning*, Springer US.

Schulman, J., Levine, S., Moritz, P., Jordan, M. I. & Abbeel, P. (2015), 'Trust region policy optimization', *CoRR* **abs/1502.05477**.

Schulman, J., Moritz, P., Levine, S., Jordan, M. & Abbeel, P. (2018), 'High-dimensional continuous control using generalized advantage estimation'.

Schulman, J., Wolski, F., Dhariwal, P., Radford, A. & Klimov, O. (2017), 'Proximal policy optimization algorithms', *CoRR* **abs/1707.06347**.

Silver, D., Hubert, T., Schrittwieser, J., Antonoglou, I., Lai, M., Guez, A., Lanctot, M., Sifre, L., Kumaran, D., Graepel, T., Lillicrap, T., Simonyan, K. & Hassabis, D. (2018), 'A general reinforcement learning algorithm that masters chess, shogi, and go through self-play', *Science* **362**(6419), 1140–1144.

Sutton, R. S. & Barto, A. G. (2018), *Reinforcement learning: An introduction*, MIT press.

van Hasselt, H., Guez, A. & Silver, D. (2015), 'Deep reinforcement learning with double q-learning', *CoRR* **abs/1509.06461**.

Wang, Z., de Freitas, N. & Lanctot, M. (2015), 'Dueling network architectures for deep reinforcement learning', *CoRR* **abs/1511.06581**.

Watkins, C. J. C. H. & Dayan, P. (1992), 'Q-learning', *Machine Learning* **8**(3-4), 279–292.

Williams, R. J. (1992), 'Simple statistical gradient-following algorithms for connectionist reinforcement learning', *Machine Learning* **8**(3-4), 229–256.

Williams, R. & Peng, J. (1991), 'Function optimization using connectionist reinforcement learning algorithms', *Connection Science* **3**, 241–.

You, Y., Pan, X., Wang, Z. & Lu, C. (2017), 'Virtual to real reinforcement learning for autonomous driving', *CoRR* **abs/1704.03952**.
URL: *http://arxiv.org/abs/1704.03952*