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

April 30, 2021

Abstract **—** **Background**: The recent success of reinforcement learning systems solving visually complex tasks, such as playing video games, means that such systems have potential for real-world applications, such as self-driving cars. To improve these systems, there is ongoing research into developing ways to transfer knowledge gained from previous tasks to solve new tasks, such as from driving in a simulation to driving in the real world.

Aims: The aim of this project is to develop an understanding of reinforcement learning and its intersection with transfer learning by investigating the use of deep neural networks and pre-training. We aim to assess to what extent a current state-of-the-art architecture benefits from pre-training in the context of learning to play Atari video-games.

Method: Classic-control and Atari 2600 environments provided by OpenAI Gym are used to train an initial implementation of the proximal policy optimisation (PPO) algorithm followed by a subsequent implementation which uses a deep convolutional neural network.

Results:

Conclusions: Transfer learning is a promising method to prepare agents for environments where it has limited opportunities to interact with and learn from. By training agents on similar environments, we can build confidence that the agent will perform successfully when evaluated on the test environment.

Keywords **—** Artificial intelligence, machine learning, reinforcement learning, deep learning, transfer learning, game-playing.

I INTRODUCTION

This project is about reinforcement learning (RL) and transfer learning (TL). The project involves developing a RL algorithm to learn to successfully play Atari games. Additionally, the project involves investigating TL approaches in RL to make learning more efficient.

A Background

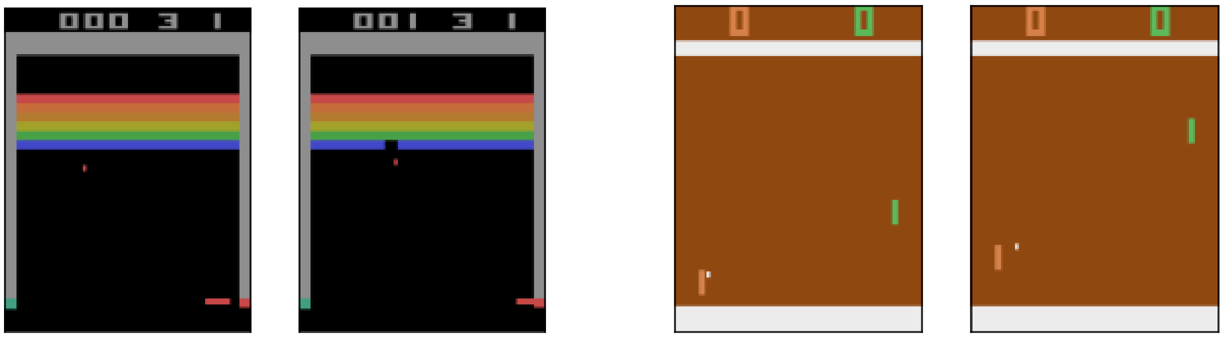
A.1 Reinforcement learning

Reinforcement learning is the class of problems concerned with an agent learning behaviour through trial-and-error interactions with a dynamic environment (Kaelbling et al. 1996). An example of a problem is an aspiring tightrope walker (the agent) learning to maintain balance (the behaviour) while walking along a tightrope that contorts and wobbles under their weight (the dynamic environment). With each attempt and fall (the trial-and-error interactions), the walker learns how better to correct their balance, and adjusts their behaviour slightly for the next attempt. When the walker is able to maintain balance consistently over consecutive attempts, the desired behaviour is achieved, and so the learning task is complete. We say that the problem is solved and the RL agent has learned to perform successfully in the environment.

There are algorithms that act as agents that solve RL problems. These RL algorithms can solve problems in physical environments, such as driving cars, or in virtual environments, such as playing video games. We can treat these algorithms as functions that take as input observations of the environment's *state*, and produce as output *actions*. Examples of states are the pixel values and positions in each frame of a car video stream or video game. Examples of corresponding actions are to brake or to press a controller button. The goal of the algorithm is to learn which actions are the best to take given the observed current state. To measure how good an action is from a state (i.e., a *state-action pair*), we can assign it a *value*. The higher the value, the better the action is considered to be. For example, a self-driving car observing a red traffic light might give the action of braking the greatest value, if the desired behaviour is to drive safely. Similarly, an algorithm playing the game Breakout might give the action of moving in the direction of the ball the greatest value, if the desired behaviour is to get a high score. The algorithm learns a mapping, from states and actions to values, to inform its decision-making. This mapping is initially unknown, but improves the more the algorithm interacts with its environment — the same way one gets better with *experience* at driving or playing video-games. For relatively complex problems, the value of a state-action pair must be estimated from the experience gathered so far. Figuring out a way of valuing actions is a key part of RL models.

A.2 Transfer learning

Transfer learning is the application of knowledge gained while solving one problem to solve a different but related problem (Sammut & Webb 2010). An example is an agent who has learned to walk a tightrope (solving one problem) applying their balancing ability (the knowledge gained) to learn to surf (a different but related problem). By reusing knowledge gained from solving past problems, it is expected that solving a different but related problem will be more efficient than it would be without the prior knowledge. As in the example, a tightrope walker should learn to balance on a surfboard more easily and more quickly, due to their knowledge of balancing on a rope, than someone without the same acquired knowledge of balancing.



(a) Breakout. (b) Pong.

Figure 1: Frames from two Atari 2600 games. Breakout requires you to break bricks, and Pong requires you to beat an opponent at 2D table tennis. You control a paddle to hit a ball in both.

For TL in RL, the knowledge gained that can be applied is the agent's *policy*. The policy is the set of rules that determine an agent's behaviour. An example of a poor policy is to take actions randomly. An example of a better policy might be to always take the action with greatest value. The policy of interest is one that the RL algorithm developed itself while learning. Depending on the architecture of the algorithm, the policy could be a neural network, so that developing the policy equates to training the network. For similar games, such as Breakout and Pong, the hope is that the network learns abstractions of one game's mechanics, which can be used to make learning the other game more efficient.

B Context

RL programs have succeeded at performing better than humans in certain tasks. Historically, TD-Gammon achieved a level of play just slightly below that of the top human backgammon players of the time (). More recently, AlphaZero is arguably currently the best Go player in history (). In the future, an example might be self-driving cars.

Existing RL algorithms have some drawbacks. One is that they are most effective when there are no limits to the trial-and-error interactions an agent can make with its environment while learning. This is not an issue in virtual settings, but could cause problems in physical settings. For example, a Breakout agent could compute games indefinitely. On the other hand, an algorithm learning from scratch to drive a car will likely crash on its first run. For reasons such as not wanting to incur a repair cost or wasting time, it is preferable to be confident that a self-driving car has a reasonable ability to drive before testing it. To build confidence that a RL algorithm will perform reasonably successfully in a real-world environment, we could first train the algorithm in a similar virtual environment, such as a simulation. We refer to this process as *pre-training*. Transfer learning is a key component of this process, since the algorithm's subsequent success depends critically on its ability to transfer the knowledge gained from the virtual domain and apply it to the real domain. For reasons such as this, transfer learning has become a crucial technique to build better RL systems ().

C Aims

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

The minimum objectives were to establish a candidate RL algorithm from the literature which would likely learn good policies for playing Atari games, and implement the algorithm minimally. This minimal algorithm should be used to train an agent in a relatively simple environment, such as CartPole (). In CartPole, the agent is required to prevent an upright pendulum attached to a cart from falling over, and is expected to *solve* the environment by getting an average reward of 195.0 over 100 consecutive trials. The purpose of this objective was to establish a strong understanding of RL algorithms, and build a foundational model for the remainder of the project.

The intermediate objectives were to adapt the implemented RL algorithm with a deep convolutional neural network and evaluate the benefit of pre-training by policy transfer. The adapted algorithm should be used to train two good agents for two Atari games, Breakout and Pong. Solving either environment requires maximising the final score. For Breakout and Pong, the trained agents should attain an average score that at least matches the popular benchmark of 401.2 and 18.9 respectively (). Then, the network weights from the trained Breakout agent should be used to initialise the network weights of a third agent learning Pong, or vice versa, and its performance evaluated against the other trained agent. The purpose of this objective was to understand how RL algorithms can process pixels, and whether knowledge can be directly transferred across neural networks.

The advanced objectives were to develop a novel architecture and workflow for pre-training an agent with multiple policies, and evaluate the effectiveness of this approach. The architecture would be a generative model, and would be required to learn from 10 agents trained on 10 different Atari games using the rRL algorithm implemented earlier. The generative model would then be used to train an agent to play an 11th Atari game, and its performance evaluated against an agent without pre-training. The purpose of this objective was to extend the current state of the art of transfer learning in RL, and provide a solution to one of OpenAI's *unsolved problems* ().

D Achievements

II RELATED WORK

Many academic papers have been published introducing RL algorithms that learn to succeed in an environment. For these algorithms, learning to succeed is framed as finding the policy which maximises expected return — this is the goal in RL. Most existing solutions find this optimal policy by either value-based methods or policy gradient methods.

A Value-based methods

Value-based methods aim to build value functions which help define a policy. 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 (François-Lavet et al. 2018). These Q-values can be found optimally by solving the Bellman equation (Bellman & Dreyfus 1962)

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

which recursively relates each state-action pair. 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 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 deep convolutional neural network (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, while policy gradient methods can.

B Policy gradient methods

Policy gradient methods define a stochastic policy by an objective function, such as $V^\pi(s)$, and aim to optimise this function by calculating the gradient — known as the policy gradient.

Given $V^{\pi_\theta}(s)$ to define the policy with parameters θ, differentiating the policy yields the following fundamental result:

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

To estimate this policy gradient, we must gather expereince by following the current policy. Afterwards, we can use the calcualted estimate to update the policy parameters θ.

The simplest estimator is a sample mean over the experience, since the expression for the policy gradient is an expectation. When calculating the sum, it has been shown that using either the finite-horizon undiscounted return, $R(\tau) = \sum_{t=0}^{T} r_t$, or the infinite-horizon discounted return, $R(\tau) = \sum_{t=0}^{\infty} \gamma^t r_t$ has the same 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', $\hat{R}_t \doteq \sum_{t'=t}^{T} R(s_{t'}, a_{t'}, s_{t'+1})$, which says that only rewards that come after an action matter (Achiam 2018).

Actor-critic algorithms (Konda & Tsitsiklis 2001) build on the discovery that the policy gradient can be re-expressed, to reduce variance whilst keeping the expectation the same, by learning 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 prevent the policy performance collapsing, and does so 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, these recent methods use deep neural networks (DNNs) as function approximators for the policy and value functions.

Most of the limitations of policy gradient methods concern performance collapse — i.e., when the algorithm ceases to learn. This occurs when the policy, or objective function, gets stuck at a local optimum. This happens because the policy becomes less random after lots of training, due to exploiting rewards that have already been found. Nonetheless, policy gradient methods are methods of learning that more closely resemble learning in nature than value-based methods.

III SOLUTION

A Specification

To develop a solution that lets an 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, to make designing an algorithm to control the agent and interact with the environment easier, the solution should change the game state only when the agent takes an action. We call this a *step*. (At time step 0, the game state is the initial game state; and, at time step 1, the game state is that after the first action, and so on.) Third, each step should return the new game state alongside additional information, such as the reward from taking the 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 reinforcement learning algorithm to control the agent and iteratively improve its behaviour.

B Tools used

Fortunately, the first three requirements are already implemented in Gym (Brockman et al. 2016), an open-source library by OpenAI for developing and comparing reinforcement learning 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. Additionally, we used the popular OpenCV (Bradski 2000) open-source computer vision library for image pre-processing.

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).

C Design

We implemented the proximal policy optimisation (PPO) algorithm (Schulman et al. 2017) and base our design on an implementation by Seungeun Rho[1]. PPO is a policy gradient method that tries to improve the policy by as much as possible during each learning update without collapsing performance.

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 update iterations, the algorithm updates its policy parameters in the next iteration by

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

where L denotes the objective function, which uses states s and actions a (gathered from experience) as well as new policy θ in its calculation. The objective function L is simply the minimum between two terms, and is given by

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

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

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. This ratio is kept in the range $[1-\epsilon, 1+\epsilon]$ by the 'clip' term to prevent performance collapse. If the ratio is too low, it is replaced by $1-\epsilon$; and, if the ratio is too large, it becomes $1+\epsilon$.

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 calculate the advantage estimates in our 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 time step t, where $t = 0, 1, \ldots, T$ are the steps leading up to a policy update, 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 $\delta_t = r_t + \gamma V(s_{t+1}) - V(S_t)$, r_t is the immediate reward at time step t, γ the discount value we need to choose, λ the GAE parameter we also need to choose, and V is the most recent estimate of the value function.

C.3 Stochastic gradient descent (SGD)

As shown, to use GAE and hence perform the policy update, experience must be collected for T steps. Our algorithm does this by playing the game and following 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 the 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. This time, 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. Additionally we repeat the update for a specified number of epochs with the aim of speeding up training.

C.4 Deep convolutional neural network (DCNN)

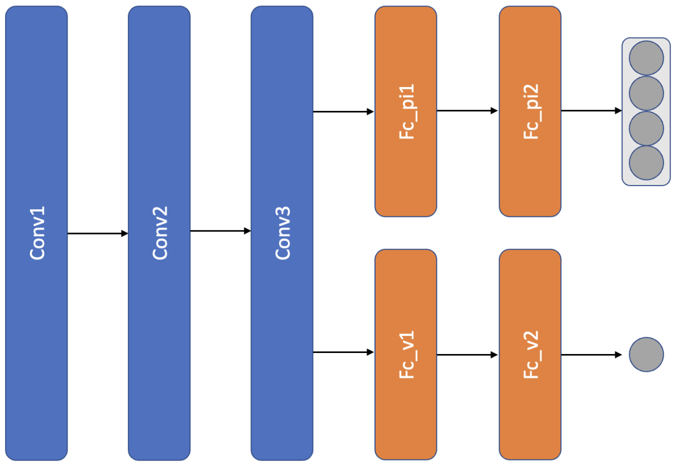
So that we can learn from pixel inputs, we use a DCNN architecture based on (Mnih et al. 2015) to estimate the policy and value function. The policy network takes as input the current state and returns a categorical distribution, which is a discrete probability distribution. The number of categories is equal to the number of actions available, 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 network also 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.

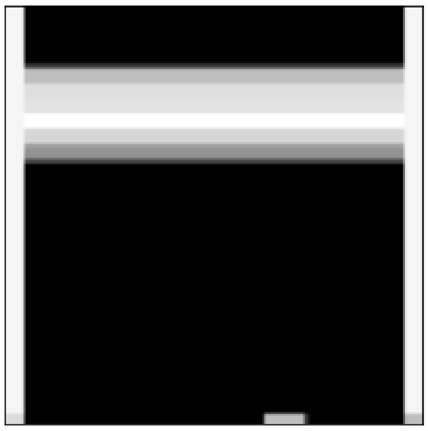
In our architecture, the policy network and value network share convolutional layers and have separated fully-connected layers. The reason for this is because the features identified that are useful to improve the policy should also be useful to improve the value function, or vice versa. The fully-connected 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. The second block has kernel 4, stride 2, and has 32 inputs. The third block has kernel size 3, stride 1, and has 64 inputs.

The fully-connected layers consist of 2 blocks of alternating linear layers and ReLU activation functions. The first block takes as input the flattened output of the convolutional layers, of dimension 3166, and the second block takes as input 512. For the policy network, the output of this are logits that are passed through a softmax function to obtain action probabilities. For the value network, the output is the value.



(a) Network architecture.



(b) Preprocessed frame.

Figure 2

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.

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.

For Pong, there is an additional step, which is to rotate the image $90°$ clockwise. This is so that the orientation of the paddle would be the same as in Breakout. We expect this to have been useful had transfer learning approaches been implemented for learning between the two games.

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 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. Although we did not implement verification methods for after the developmental stages, the prior runs ensured that the specification continued to be met once the system was formed. Nonetheless, an improvement would be to implement additional methods, such as to check action probabilities and content of the frame stack during agent-training.

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 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 is to produce and analyse the following statistics during agent training: mean episode score, episode score standard deviation, and mean objective loss — each over the most recent 100 episodes — as well as total number of time 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 and Pong environments 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 chose to train each agent for 1000 episodes initially, and evaluated its performance after every 100th episode by the evaluation method described. If the agent showed signs of converging, we continued training the agent. 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.

B Experimental settings

We started by using hyperparameters that were identical, where possible, to those stated in the paper by OpenAI that introduced the PPO algorithm (Schulman et al. 2017).

Table 1: PPO hyperparameters used in Atari experiments. α is linearly annealed from 1 to 0 over the course of learning.

(a) OpenAI's hyperparameters.

Hyperparameter	Value
Horizon (T)	128
Adam stepsize	$\mathbf{2.5 \times 10^{-4}} \times \alpha$
Num. epochs	3
Minibatch 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	$\mathbf{0.01}$

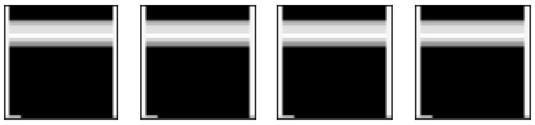
(b) Our hyperparameters.

Hyperparameter	Value
Horizon (T)	128
Adam stepsize	$\mathbf{2.5 \times 10^{-4}}$
Num. epochs	3
Minibatch 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 value of the seed we use is 742.

B.1 Experiments

First, we trained an agent using the hyperparameters stated in Table 1b. 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 the policy is not improving. However, each episode lasted exactly 2500 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 the action-probabilities produced by the policy network: there is a 0.99 probability of selecting the 'LEFT' action. We conclude that this performance collapse was due to the policy becoming trapped in a local optimum due to initial conditions causing lack of exploration early on in training.



(a) Input state.

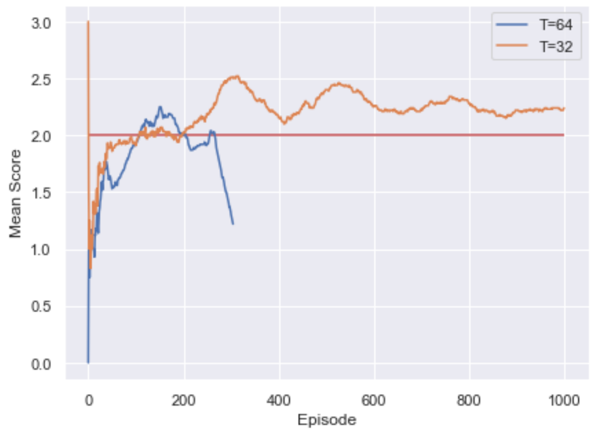
NOOP	FIRE	RIGHT	LEFT
0.0001	0.0003	0.0009	0.9985

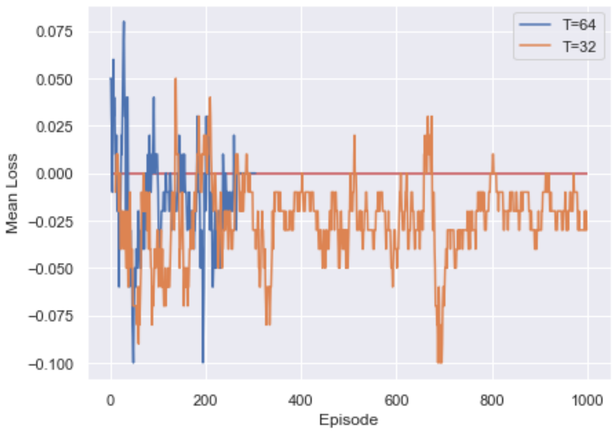
(b) Output action-probabilities.

Figure 3: .

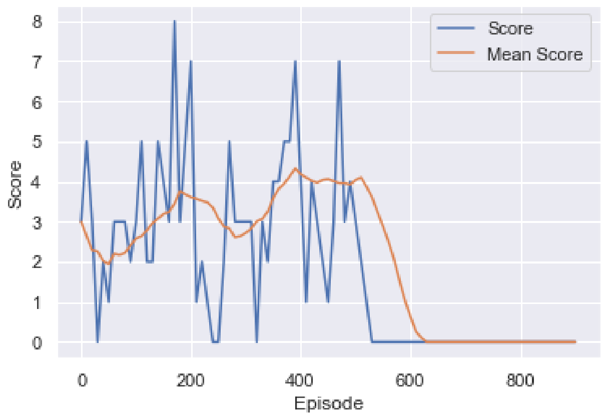
Second, we trained using different values for the Adam step-size. The reason for this is to assess the hypothesis that smaller gradient updates might prevent performance collapse, since the policy may be less likely to get trapped in a local optimum. For step-size 1.5×10^{-4}, the result was that performance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

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 1000 episodes (or 80,000 steps and 2,000 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 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.





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 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.



Fifth, we trained using a combination of the values that improved performance in the previous experiemnts. These were $K = 1$, $T = 64$, and adam step-size 0.5×10^{-4}. We also used $K = 1$, $T = 32$, and adam step-size 0.5×10^{-4}. The results were that there performance loss occurred again early on in training. We did K=1, T=32, and adam = 0.5 for seed 742. Performance collapse after 350 episodes.

We tried starting values on seed 426. Agent just learned to stay still. For seed 999, agent moved to right wall.We tried combination of values in fifth for seed 999, no luck.

Finally, we varied teh seed. In another run, we had more success. After 1,000 episodes, with over 100,000 timesteps and 1,300 learning steps, the agent achieved a mean score of 5.31 and a high score of 11. Again, video inspection revealed interesting behaviour: at 1000 timesteps, the agent had learned to alternate between moving to the left and right walls, as this was where the ball travelled to after it was first initialised. However, at 500 timesteps, the behaviour more closely resembled that of moving in the direction of the ball. We believe that the agent overfitted to the initial conditions when episodes are reset — i.e. the policy gradient is stuck in a local maximum. Upon training the agent further, we observed the agent to start to undo its overfitting behaviour by episode 1,300; however, by episode 1,400, performance had collapsed and the mean score dropped to 0. This time, the action probabilities were heavily weighted towards the 'move right' action. This meant the ball never came into play, and the agent would remain trapped in the episode, which would last over 5 minutes. As a result, training was terminated.

V EVALUATION

A System strengths

The first strength of the system is that it learns a stochastic policy. Policy gradient methods are better than value-based methods. It explores by sampling actions according to the latest version of its stochastic policy. The amount of randomness in action selection depends on both initial conditions and the training procedure. Over the course of training, the policy typically becomes progressively less random, as the update rule encourages it to exploit rewards that it has already found.

The second strength of the system is that it it is fast. It uses CUDA to speed up training and has an efficient architecture. It is memory efficient due to being sample efficient. Talk about architecture. Adam optimisation algorithm (Kingma & Ba 2017) can handle sparse gradients on noisy problems.

The third strength of the system is that it is simpler than other policy gradient methods. PPO has fewer tricks and is easier to understand. Huber loss is used because it is less sensitive to outliers than mean-squared error loss (squared L2 loss), and in some cases prevents exploding gradients (Girshick 2015).

B System limitations

The first limitation is that the Adam step size and clipping parameter are not annealed.

The second limitation is that we run a single agent. The original paper runs 8.

The third limitation is that we don't use early stopping to help prevent new policy being too far from old. Rule is if the mean KL-divergence of the new policy from the old grows beyond a threshold, stop taking gradient steps. The method is susceptible to policies being trapped in local optima.

The fourth limitation is that we did multiple steps of batch SGD rather than mini-batch SGD to maximise the objective. Even seemingly small differences in parameter space can have very large differences in performance — so a single bad step can collapse the policy performance. This has drastic impacts on the algorithms' sample efficiency.

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 it was the RL algorithm that performed poorly, so that TL could not even be considered. Having the intermediate and advanced objectives depend on the minimum objective in this way 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 most of the 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.

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

A *Project overview*

- The project was to …

B *Main findings*

- The main findings were as follows: …
- The conclusions from these findings were …

C *Further work*

- The project can be extended by …

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.

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.

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**.

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.