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 23, 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 apply 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. This is particularly useful in fields such as autonomous driving, where the vehicle must be able to adapt to various changes in the 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

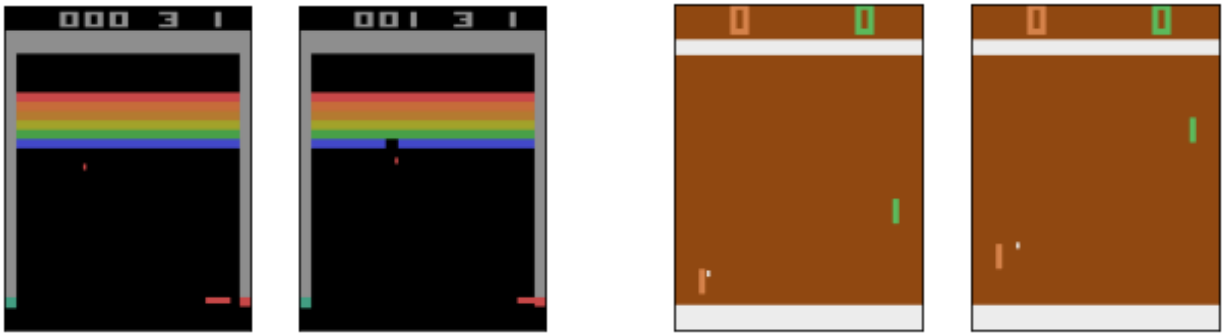
A.1 Reinforcement learning

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 perform successfully in its environment. For an agent to succeed, it must learn a map, $f : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$, that maps from state-action pairs to values, and make informed decisions. Most existing solutions learn this map by either value-based methods or policy gradient methods.

A Value-based methods

The value-based class of reinforcement learning algorithms aims to build a value function, which subsequently lets us define a policy. One of the simplest and most popular value-based algorithms is the Q-learning algorithm (Watkins & Dayan 1992). The basic version of Q-learning keeps a lookup table of values with one entry for every state-action pair. In order to learn the optimal Q-value function, the Q-learning algorithm makes use of the Bellman equation for the Q-value function (Bellman & Dreyfus 1962), which has a unique solution under certain conditions. This idea was expanded to involve approximating the Q-values from experiences gathered (Gordon 1995), and parametrising the Q-values with a neural network whose parameters are updated by stochastic gradient descent (or a variant) by maximising a loss function (Riedmiller 2005). A breakthrough approach used deep convolutional neural networks and a replay memory to play Atari games by learning from pixels and a wide range of past experiences; this approach was the deep Q-network (DQN) algorithm (Mnih et al. 2013). More recent approaches made improvements to the DQN algorithm, such as duelling DQN (Wang et al. 2015) and double DQN (van Hasselt et al. 2015). Rainbow DQN (Hessel et al. 2017) is a combination of the many improvements to the DQN algorithm. For simple environments, a unique solution to the optimal Q-value function may be found; whereas, for complex environments, such as video games, modern approaches use neural networks to approximate the Q-values, which are then used to define a policy.

One limitation with value-based approaches is that these types of algorithms are not well-suited to deal 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 by a human (Mnih et al. 2015).

B Policy gradient methods

Policy gradient methods optimise a performance objective (typically the expected cumulative reward) by finding a good policy thanks to variants of stochastic gradient ascent with respect to the policy parameters. The gradient of the performance objective with respect to the policy parameters is called the policy gradient. The simplest policy gradient estimator is derived by using either the expected finite-horizon undiscounted return or expected infinite-horizon discounted return as the performance objective; this results in the REINFORCE algorithm (Williams 1992). Deep deterministic policy gradient (DDPG) (Lillicrap et al. 2019) extends this approach to deterministic policies to allow DQN algorithms to overcome the restriction of discrete actions. Another common approach is to use an actor-critic architecture that consists of two parts: an actor and a critic (Konda & Tsitsiklis 2001). The actor refers to the policy and the critic to the estimate of a value function (e.g., the Q-value function). In deep RL, both the actor and the critic can be represented by non-linear neural network function approximators (Mnih et al. 2016). More advanced approaches incorporate an advantage function to describe how much better or worse

an action is than other actions on average (relative to the current policy) (Schulman et al. 2018), and use special constraints, expressed in terms of KL-Divergence (Kullback & Leibler 1951), on how close the new and old policies are allowed to be (Schulman et al. 2015). Proximal policy optimisation (PPO) (Schulman et al. 2017) builds on the latter approach by relying on specialised clipping in the objective function. Typically, these approaches rely on tricks to keep new policies close to the old.

One limitation of policy gradient methods is that 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. This may cause the policy to get trapped in local optima. Additionally, 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 Application of transfer learning

D Generative models

III SOLUTION

A Specification

To develop a reinforcement learning system that can successfully operate in its environment, the system must satisfy certain requirements. First, the system would initialise an environment and the agent in it. Second, a reinforcement learning algorithm would control the agent and interact with the environment by making observations and taking actions. Third, the environment would respond with a reward for each action taken by the agent. Finally, the algorithm would process the observations, actions, and rewards to iteratively improve the agent's behaviour.

Additionally, there is another set of requirements for developing a system to pre-train reinforcement learning agents. First, the trajectories from one or more reinforcement learning agents operating in different environments would be generated. Second, a generative model would learn the agents' behaviour by predicting the actions they took. Finally, the generative model would control the agent in the new environment.

B Tools used

We used the Python programming language for our implementation, since Python has numerous libraries for machine learning, with support for deep learning and reinforcement learning. Examples of libraries are Keras, PyTorch, and TensorFlow. Of these, we used the PyTorch library, since this is currently what most researchers are using to implement their state-of-the-art papers. This library allowed us to implement existing architectures and easily make modifications to them, without the additional complexity of programming in low-level PyTorch. Additionally, we used OpenCV and Scikit-learn for image manipulation in our pre-processing steps.

So we wouldn't need to create our own API for Atari emulations, we used Gym (Brockman et al. 2016), a toolkit developed by OpenAI for developing and comparing reinforcement learning algorithms. This allowed us to focus on implementing our reinforcement learning algorithm. Gym provides a collection of environments, including control theory problems, physics simulations, and Atari games from the Arcade Learning Environment (ALE) (Bellemare et al. 2013). The environments have a shared interface, so we can write general algorithms that will operate in any type of environment.

For network training, we used Google Colab, which allows Python code to be written and executed in the browser. Google Colab does not require configuration and has free access to GPUs (including CUDA). There was the option to use Durham University's NVIDIA CUDA Centre (NCC), although we chose not to use it due to having no prior remote server experience or job queuing system experience (such as SLURM).

C Proximal Policy Optimisation (PPO) algorithm

(At the moment mostly notes that were copied. Will reword.)

The proximal policy optimisation (PPO) algorithm was chosen to be implemented due to its relative simplicity and strong performance on the Atari benchmark (Schulman et al. 2017). PPO is an on-policy algorithm which can be used for environments with either discrete or continuous action spaces. PPO is motivated by the same question as TRPO: how can we take the biggest possible improvement step on a policy using the data we currently have, without stepping so

far that we accidentally cause performance collapse? Where TRPO tries to solve this problem with a complex second-order method, PPO is a family of first-order methods that use a few other tricks to keep new policies close to old. There are two primary variants of PPO: PPO-Penalty and PPO-Clip.

PPO-Penalty approximately solves a KL-constrained update like TRPO, but penalizes the KL-divergence in the objective function instead of making it a hard constraint, and automatically adjusts the penalty coefficient over the course of training so that it's scaled appropriately.

PPO-Clip doesn't have a KL-divergence term in the objective and doesn't have a constraint at all. Instead relies on specialized clipping in the objective function to remove incentives for the new policy to get far from the old policy.

Our implementation is of PPO-clip, which updates policies via

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

takes multiple step of batch gradient descent to maximise the objective, L. L is given by

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

in which ϵ is a (small) hyperparameter which roughly says how far away the new policy is allowed to go from the old. Gradient clipping is a technique to prevent exploding gradients, such as rescaling gradients so that their norm is at most a particular value.

A simplified version, which we implement in our code, is as follows:

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

where

$$g(\epsilon, A) = \begin{cases} (1+\epsilon)A & \text{if } A \geq 0 \\ (1-\epsilon)A & \text{if } A < 0. \end{cases}$$

When the advantage for the state-action pair is positive, the objective reduces to

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

Because the advantage is positive, the objective will increase if the action becomes more likely — that is, if $\pi_\theta(a|s)$ increases. But the min in this term puts a limit to how *much* the objective can increase. Once $\pi_\theta(a|s) > (1+\epsilon)\pi_{\theta_k}(a|s)$, the min kicks in and this term hits a ceiling of $(1+\epsilon)A^{\pi_{\theta_k}}(s,a)$. Thus: the new policy does not benefit by going far away from the old policy.

On the other hand, when the advantage for the state-action pair is negative, the objective reduces to

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

Because the advantage is negative, the objective will increase if the action becomes less likely. Similar to before, the max in this term puts a limit to how much the objective can increase. Once

$\pi_\theta(a|s) < (1-\epsilon)\pi\theta_k(a|s)$, the max kicks in and this term hits a ceiling of $(1-\epsilon)A^{\pi_{\theta_k}}(s,a)$. Thus, again: the new policy does not benefit by going far away from the old policy.

Clipping serves as a regulariser by removing incentives for the policy to change dramatically, and the hyperparameter ϵ corresponds to how far away the new policy can go from the old while still profiting the objective.

Algorithm 1 PPO-Clip pseudocode

1: Input: initial policy parameters θ_0, initial value function parameters ϕ_0
2: **for** $k = 0, 1, 2, ...$ **do**
3: Collect set of trajectories $\mathcal{D}_k = \{\tau_i\}$ by running policy $\pi_k = \pi(\theta_k)$ in the environment.
4: Compute rewards-to-go $\hat{R}_t$.
5: Compute advantage estimates, $\hat{A}_t$ (using any method of advantage estimation) based on the current value function V_{ϕ_k}.
6: Update the policy by maximizing the PPO-Clip objective:

$$\theta_{k+1} = \arg\max_\theta \frac{1}{|\mathcal{D}_k|T} \sum_{\tau \in \mathcal{D}_k} \sum_{t=0}^{T} \min\left(\frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_k}(a_t|s_t)} A^{\pi_{\theta_k}}(s_t,a_t), \quad g(\epsilon, A^{\pi_{\theta_k}}(s_t,a_t)) \right),$$

typically via stochastic gradient ascent with Adam.
7: Fit value function by regression on mean-squared error:

$$\phi_{k+1} = \arg\min_\phi \frac{1}{|\mathcal{D}_k|T} \sum_{\tau \in \mathcal{D}_k} \sum_{t=0}^{T} \left(V_\phi(s_t) - \hat{R}_t \right)^2,$$

typically via some gradient descent algorithm.
8: **end for**

D Design

D.1 Main training loop

Our PPO implementation is based on Seungeun Rho's implementation with minimal lines of code[1]. The main training loop consists of one agent alternating between collecting experience over t time-steps and performing k steps of gradient descent to update network parameters.

When collecting experience, the agent samples an action from a state with probability distribution given by the current policy network. Since we are working with environments with a discrete number of possible actions, the probability distribution the policy returns is also discrete. In PyTorch, the distribution takes the form of a tensor, with the number of elements equal to the number of actions available. Each value corresponds to the probability of selecting the action with corresponding index. For example, if the value of the element at index 0 were 0.25, then the probability of selecting action 0, which could be to move left, is 0.25. Necessarily, the probabilities must sum up to 1 each time.

When updating the network parameters, tuples of state, action, reward, and next state from the last t time-steps are used. First, advantage estimates are calculated by the following generalised

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

advantage estimation equation:

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

where $\delta_t = r_t + \gamma V(s_t + 1) - V(S_t)$. This is used to calculate the PPO-Clip objective, which we subtract from the smooth L1-loss of the value function. Smooth L1-loss creates a criterion that uses a squared term if the absolute element-wise error falls below beta, and an L1 term otherwise. This 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). The loss between x and y is as follows:

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

where z_i is given by the following:

$$z_i = \begin{cases} 0.5(x_i - y_i)^2/beta, & \text{if } |x_i - y_i| < beta \\ |x_i - y_i| - 0.5 * beta, & \text{otherwise} \end{cases}$$

(Beta is an optional parameter that defaults to 1.)

Finally, back-propagation is performed by the Adam optimisation algorithm (Kingma & Ba 2017), which can handle sparse gradients on noisy problems.

D.2 Network architecture

We use a deep convolutional neural network as the main component of our policy gradient and value function estimators. "There are typically 3 main types of layers that are used to build DCNN, which will be used throughout our solution. Convolutional layers perform convolution by sliding a filter across the image and computing the dot product of the filter with the image values at each point, this produces a two-dimensional activation map which gives the response of the filter at each point in the image. Many of these filters can be present in a single layer, each producing a separate activation map. As the network is trained, it will learn filters that activate when they see specific features within the image, such as circular patterns or vertical edges. Convolutional layers are the backbone of the convolutional neural network. Fully-connected layers typically occur after the convolutional layers, and have full connections to all activations in the previous layer, similar to a layer in a basic neural network. They allow us to shape the output of the network to fit the number of classes in the problem."

- Our network architecture consists of a deep convolutional neural network connected to two fully-connected layers on two branches.
- The motivation behind this is that the DCNN learns edges and features, which the fully-connected layers then use to each estimate the value function and policy gradient respectively.
- Our DCNN consists of 3 blocks. 1) 4 in (of 84x84 image), 32 out, kernel size 8, stride 4. 2) 32 in, 64 out, kernel size 4, stride 2. 3) 64 in, 64 out, kernel size 3, stride 1.
- This is the architecture we used.

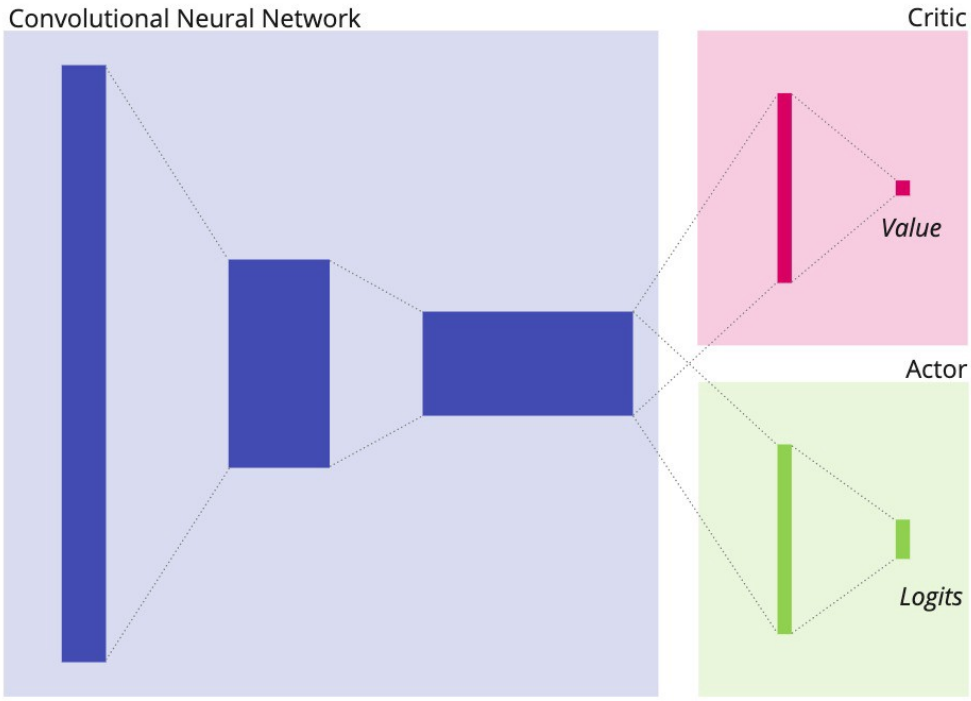


Figure 2: PPO network architecture (will create my own diagram)

E Implementation features

- Our implementation differs from the main paper in a number of ways.
- We used an architecture that shares parameters. We don't have an entropy term.
- 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.
- We did multiple steps of batch SGD rather than mini-batch SGD to maximise the objective
- We used our own convolutional neural network.
- We tried frame skipping.
- We did preprocessing of frames
- We tried resetting after end of each life.
- The original algorithm anneals the adam step size and clipping parameter which we don't.

- The original algorithm runs 8 agents in parallel. We only did one due to hardware limitations.

F Verification

(This might be a description of testing rather than verification.)

Verification was done on the reinforcement learning system to check that the specification was met. During developmental stages, verification methods consisted of conducting individual tests to run components of the system, followed by checking the results from the run. We tested each frame pre-processing step (greyscaling, cropping, resizing, and stacking), the convolutional layer, and both fully-connected layers, by checking the outputs were as expected. Although we did not implement verification methods for after the developmental stages, the prior tests ensured that the specification continued to be met once the system was formed. Nonetheless, an improvement would be to implement additional tests, such as to check action probabilities and replay-buffer content during training.

Were the transfer learning system to have been successfully implemented, verification would also have been done on it. For transfer learning via policy transfer, the verification method would be to check that each network parameter of the newly initialised agent was equal to that of the trained agent. For transfer learning via the generative model, verification methods would consist of conducting tests for each pre-processing step and network layer, as well as during training.

G Validation

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.

H Testing

Unit testing, integration testing, and system testing were done on the solution.

Unit testing was done to test singular components of code to establish if it works correctly. Method, class, test case

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 3000 episodes, and evaluated its performance after every 1000th episode, by the evaluation method described. We discovered that our implementation was susceptible to performance loss, even early in the training procedure, and as a result conducted several experiments using different settings in an effort to improve performance.

B *Initial experiments*

Initial training was done using hyperparameters that were identical, where possible, to those stated in the paper by OpenAI that introduced the PPO algorithm.

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	$2.5 \times 10^{-4} \times \alpha$
Num. epochs	3
Minibatch size	32×8
Discount (γ)	0.99
GAE parameter (λ)	0.95
Number of actors	8
Clipping parameter ϵ	$0.1 \times \alpha$
VF coeff. c_1	1
Entropy coeff. c_2	0.01

(b) Our hyperparameters.

Hyperparameter	Value
Horizon (T)	128
Adam stepsize	2.5×10^{-4}
Num. epochs	3
Minibatch size	—
Discount (γ)	0.99
GAE parameter (λ)	0.95
Number of actors	1
Clipping parameter ϵ	0.1
VF coeff. c_1	—
Entropy coeff. c_2	—

In one run, we found that performance loss occurred immediately after the first training update. Episode 1 had a score of 1, whereas every episode thereafter had a score of 0. Video inspection revealed that the agent moved randomly in the first episode, and in the rest it was stationary. The only action the agent was taking was to initialise the ball. Looking at the mean values of the objective loss, we noticed that the loss peaked in the first episode. This led us to believe that this high loss value caused the training update to push the action probabilities away from the 'move left' and 'move right' actions so much that it was improbable for them to be sampled any time soon.

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.

C Horizon experiments

Thinking that a factor in what is causing our network to converge in local optima is the batch size of the gradient update, we train additional agents with $T = 64$ and $T = 32$. Similar to previous agents, for $T = 64$, our agent converges to a mean score of 2.24 after over 80,000 timesteps and 2,000 learning steps. Videos show that the agent positions itself against the right wall and knows how to initialise the ball; it scores points only when the initialise ball directly lands on the stationary paddle.

D Adam stepsize experiments

V EVALUATION

We discuss in this section the suitability of our approach to this project, the strengths and limitations of our algorithm, and the organisation of the project.

A System strengths

B System limitations

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

Bellemare, M. G., Naddaf, Y., Veness, J. & Bowling, M. (2013), 'The arcade learning environment: An evaluation platform for general agents', *Journal of Artificial Intelligence Research* **47**, 253–279.

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

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

Girshick, R. (2015), 'Fast r-cnn'.

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.

Lillicrap, T. P., Hunt, J. J., Pritzel, A., Heess, N., Erez, T., Tassa, Y., Silver, D. & Wierstra, D. (2019), 'Continuous control with deep reinforcement learning'.

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., Graves, A., Antonoglou, I., Wierstra, D. & Riedmiller, M. A. (2013), 'Playing atari with deep reinforcement learning', *CoRR* **abs/1312.5602**.

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.

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.