Reinforcement Learning: Transfer Learning Between Different 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 9, 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. In addition to this, network weights from an agent trained on Breakout were used as initial weights for another agent training on Pong, after which its performance was evaluated in comparison to one without pre-training.

Results — Pre-training leads to the agent performing better, as measured by an improvement in the following three metrics: jump-start performance, accumulated reward, and final performance. The more pre-training information the agent has, the better it performs.

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 (2-3 PAGES)

This project is about reinforcement learning and transfer learning. The project involves developing a reinforcement learning algorithm to learn to perform successfully in some environment. Additionally, the project involves investigating how transfer learning lets the algorithm store knowledge and apply it — to learn to perform successfully in a different but related environment.

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 reinforcement learning agent has learned to perform successfully in the environment.

There are algorithms that act as agents that solve reinforcement learning problems. These reinforcement learning algorithms can solve problems in physical settings, such as driving cars, or in virtual settings, such as playing games. We can treat these algorithms as functions that takes as input observations and outputs actions. Examples of observations are the video from a camera attached to a self-driving car or the positions of pieces on a chessboard in an online match. Examples of corresponding actions are to turn the steering wheel in one direction or to move a chess piece. The goal of the algorithm is to learn which actions are the best to take given some observations. How good an action given an observation is can be measured by how likely taking the action is to lead to the desired behaviour. For an algorithm that drives cars, the desired behaviour might be to drive safely, and so the algorithm would know to stop at a red traffic light. Whereas, for an algorithm that plays chess, the desired behaviour might be to win, and so the algorithm would know to take the opponent's king. The algorithm is learning a mapping, from actions and observations 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 practice at driving or playing chess. For relatively complex problems, there may not be an optimal solution, such as behaviour that guarantees no accidents or that always wins, and so the best the algorithm can do is to approximate an optimal solution.

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.

In transfer learning for reinforcement learning algorithms, 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 (which can be informed by the mapping mentioned in the previous section). An example of a policy for an agent playing the video game Breakout might be to move the paddle randomly. Another, better policy might be to move the paddle in the direction of the projectile, so as to deflect it. The policy of interest, however, is one that the reinforcement learning algorithm developed itself while learning to play. Depending on the architecture of the algorithm, the policy could be a neural network, so that developing the policy equates to training the network. An example of transfer learning for reinforcement learning algorithms, then, could be to apply the trained neural network, or part of it, that was trained in the agent that learned to play Breakout, to a new agent that is learning to play a different but related game, such as Pong. In both games, players must move a paddle to deflect a projectile. The hope is that an abstraction of this concept has been learned in the network, which gets translated and used to make learning other games with similar features more efficient.

A.3 Context

Reinforcement learning algorithms have proven to be successful at achieving at least human-level performance in some tasks. A historical example is TD-Gammon: a program that achieved a level of play just slightly below that of the top human backgammon players of the time (). Another example that is more recent is AlphaGo: a program that is claimed to be arguably the strongest Go player in history (). In the future, an example might be self-driving cars. As computing power increases and new algorithms continue to be discovered (), reinforcement learning research will only grow in popularity, and so it is important to investigate it.

There is one drawback to existing reinforcement learning algorithms: 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 computer chess program does not feel fatigue and is able to compute millions of games in minutes. 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 reinforcement learning 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. 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 reinforcement learning systems ().

B 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 reinforcement learning 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 reinforcement learning algorithms, and build a foundational model for the remainder of the project.

The intermediate objectives were to adapt the implemented reinforcement learning 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. In Breakout, the agent is required to break bricks with a ball and paddle (); whereas, in Pong, the agent is required to beat an opponent at table tennis with a ball and paddle (). 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 reinforcement learning 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 reinforcement learning 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 reinforcement learning, and provide a solution to one of OpenAI's *unsolved problems* ().

C Achievements

We achieved the minimum and intermediate objectives of the project. We developed a minimal PPO algorithm that learned to perform successfully in the CartPole environment. By introducing a convolutional neural network (CNN) to the policy and value networks, the algorithm was able to learn to perform successfully in the environments Pong and Breakout. By reusing the weights in the network of one of the agents, another agent was trained and its performance analysed. We observed there to indeed be a benefit of pre-training. The greater the amount of data, the greater the benefit there was, as measured by three metrics: jump-start performance, accumulated rewards, and final performance. The advanced objective was not able to be completed due to the challenges brought about by external limiting factors, such as Covid-19.

II RELATED WORK (3 PAGES)

There have been several academic achievements in developing reinforcement learning algorithms that address challenging sequential decision-making problems, with potential for real-world applications. For an agent to learn a good behaviour, the agent has to make decisions in an environment to optimise a given notion of cumulative rewards. We focus on existing solutions that achieve this by value-based methods or policy gradient methods. Other approaches include those that combine the two methods, or are model-based.

A Value-based methods

One of the simplest and most popular value-based algorithms to define a policy is Q-learning. The basic version of Q-learning keeps a lookup table of values with one entry for every state-action pair (François-Lavet et al. 2018). In order to learn the optimal Q-value function, the Q-learning algorithm makes use of the Bellman equation for the Q-value function, which has a unique solution. This idea was expanded to involve approximations of the Q-values (Gordon 1995), and Q-values parameterised with a neural network where the parameters are updated (Riedmiller 2005). More recent approaches use deep neural networks as function approximators (Mnih et al. 2015), culminating in the deep Q-learning algorithm (DQN). Many improvements made to the DQN algorithm, incorporating dueling network architectures (), double Q-learning (), prioritised experience replays (), which were all combined to form Rainbow DQN (). Traditionally, the optimal policy was found by finding an exact solution; whereas, modern approaches approximate the optimal policy by using states as input to neural networks.

One limitation of 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 (François-Lavet et al. 2018). Basic approaches include using the expected finite-horizon undiscounted return as the performance objective (), and reward-to-go policy gradient, which reinforces actions only on rewards obtained after taking the action () — the latter method makes more sense than the former, which reinforces actions based on *all* rewards ever obtained. Approaches which build on this include those that use the infinite-horizon discounted return as the performance objective, and incorporate an advantage function to describe how much better or worse an action is than other actions on average (relative to the current policy) (). More advanced methods use special constraints, expressed in terms of KL-Divergence, on how close the new and old policies are allowed to be (Schulman et al. 2015), use Taylor expansions to make estimates (Schulman et al. 2015), and take mutiple steps of minibatch stochastic gradient ascent (Schulman et al. 2017). Most of these approaches have an actor-critic architecture, where the actor is the policy and the critic to estimates how good the actor's choices are.

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 (). 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 makes it dangerous to use large step sizes with vanilla policy gradients, thus hurting its sample efficiency. TRPO nicely avoids this kind of collapse, and tends to quickly and monotonically improve performance.

C Application of transfer learning

D Generative models

III SOLUTION (5 PAGES)

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 Design

The algorithm chosen to be implemented was proximal policy optimisation (PPO). PPO is an on-policy algorithm, and PPO 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.

PPO-clip updates policies via

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

typically taking multiple steps of (minibatch) SDG to maximise the objective. 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 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

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

To prevent ending up with a new policy which is too far from the old policy, we can stop taking the gradient steps if the mean KL-divergence of the new policy form the old grows beyond a threshold (early stopping). The method is susceptible to policies being trapped in local optima. To figure out what intuition to take away from this, let's look at a single state-action pair (s, a), and think of cases.

Advantage is positive: Suppose the advantage for that state-action pair is positive, in which case its contribution to 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.*

Advantage is negative: Suppose the advantage for that state-action pair is negative, in which case its contribution to 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---that is, if $\pi_\theta(a|s)$ decreases. But 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.*

What we have seen so far is that clipping serves as a regularizer 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. While this kind of clipping goes a long way towards ensuring reasonable policy updates, it is still possible to end up with a new policy which is too far from the old policy, and there are a bunch of tricks used by different PPO implementations to stave this off. In our implementation here, we use a particularly simple method: early stopping. If the mean KL-divergence of the new policy from the old grows beyond a threshold, we stop taking gradient steps.

```
1: Input: initial policy parameters $\theta_0$, initial value function parameters $\phi_0$
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 $V_{\phi_k}$.
```

```
6:    Update the policy by maximizing the PPO-Clip objective:
          $\theta_{k+1} = \arg\max_{\theta} \frac{1}{|\mathcal{D}_k|T} \sum_{\tau \in \mathcal{D}_k} \sum_{t=0}^{T} \min\left(\frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_k}(a_t|s_t)} A^{\pi_{\theta_k}}(s_t, a_t),\ \ g(\epsilon, A^{\pi_{\theta_k}}(s_t, a_t))\right),$
      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
```

C Implementation features

C.1 CartPole

For the simple version, we used a fully connected network.

DNN are an extension of the traditional artificial neural network, which is a technique for pattern recognition which attempts to replicate the function of neurons in the human brain. A network is composed of many elements called perceptrons, each of which simulates the activity of a neuron in the human brain. Each perceptron takes an input, and produces an output by applying its activation function to the input, which is then typically sent as input to another perceptron via a weighted edge. Typically, perceptrons are organized into layers, with a feed-forward architecture, meaning that the output of a perceptron in a given layer is only sent to the next layer in the network. Each perceptron usually has connections to every perceptron in the previous layer. Consequently, in a given network there

would typically be an input layer, an output layer, and a number of hidden layers between the input and output layers. These networks learn by computing the output for a given input, and then updating the connection weights depending on the difference between the actual output and the expected output, in order to better meet the expected output if a similar input is received (LeCun et al. 1990). We call this supervised learning - feeding data into the network along with labels which contain the expected output. With each training iteration, the network becomes better at classifying that specific input by adjusting connection weights – the network learns how it should classify each input. A deep neural network only differs from a standard neural network in that there are a greater number of hidden layers between the input and output layer of the network. Since each layer of perceptrons trains on the set of features which are based on the output of the previous layer, the deeper the network (i.e. the more layers it has), the more complex the features it can recognize. Therefore, when we process complex data such as images, DNN tend to perform better than standard neural networks.

C.2 Atari

This is how DNN work with a general input, however to process complex images we must make use of a more sophisticated structure. Due to images having three dimensions (width, height, and depth), they contain a huge number of parameters, and so processing an image with a standard network would be very inefficient. To enable neural networks to efficiently process images, we use something called convolutional neural networks (LeCun et al. 1998), in which the responses of each perceptron are produced by performing a convolution operation upon the image. This allows us to arrange perceptrons in three dimensions (width, height, and depth), and therefore massively reduce the number of parameters required to process an image. 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. Pooling layers reduce the dimensions of their input by separating the input into grids, and performing a down-sampling operation upon each grid. This operation is usually the max function, which returns the maximum of a set of numbers. These layers reduce the number of parameters in the network, and help prevent overfitting. Fully-connected layers typically occur after the convolutional and pooling 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, which in our case is 2. All networks presented in our solution are DCNN with varying structures.

- We used an architecture that shares parameters. We don't have an entropy term.
- We used clipped surrogate objective.
- We used our own convolutional neural network.
- We did not do frame skip. We tried frame skip.
- We did preprocessing of frames
- We tried resetting after end of each life.
- The original algorithm does minibatch updates where we do batch updates.
- The original algorithm anneals the adam step size and cliping parameter which we don't.
- The original algorithm has an entropy term.
- The original algorithm runs 8 agents in parallel. We only did one due to hardware issues. The architecture was also slightly different.

D Tools used

We used the Python programming language. This was because Python has numerous libraries for machine learning, which support deep learning and reinforcement learning. Examples of libraries are Keras, PyTorch, and TensorFlow.

We used the PyTorch library. This was because PyTorch 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 ot them, without the additional complexity of programming in low-level PyTorch.

We used OpenCV and Scikit-learn for image manipulation, so we did not have to implement pre-processing techniques from scratch.

We used Gym, a toolkit developed by OpenAI for developing and comparing reinforcement learning algorithms. We chose this so we could focus on the algorithm. Gym provides a collection of environments. Environments include Atari games, control theory problems, and physics simulations. The environments have a shared interface, which allows us to write general algorithms.

We used Google Colab, which Python code to be written and executed in the browser. This was because it doesn't require configuration and has free access to GPUs (including CUDA). There was the option to use Durham University's NVIDIA CUDA Centre (NCC). We chose not to use this because we had no remote server experience or job queuing system experience (such as SLURM).

- Rather than implement everything from scratch, we built on frameworks to allow focus on algorithm design.
- OpenAI Gym, arcade learning environment
- Adam optimiser
- PyTorch

E Verification and validation

- Verification was done by ...
  - Do implementations work there? ...
  - What do I do to judge the outcome/success?
  - Try to answer whether transfer learning is generalisable
- Validation was done by ...

F Testing

[Bridging paragraph]

- Testing was done by ...
  - reproduce on simple problems

IV RESULTS (3 PAGES)

[Bridging paragraph]

A Evaluation method

- The evaluation methods adopted were …

B Experimental settings

- These were the experimental settings for each experiment carried out: …

C Results

- The results generated by the software were …

V EVALUATION (3 PAGES)

[Bridging paragraph]

A Suitability of the approach (more SE, maybe exclude?)

PPO methods are significantly simpler to implement, and empirically seem to perform at least as well as TRPO.

- The approach was/was not suitable because …
- Was it a good idea to use PyTorch, etc.?

B Strengths and limitations of the algorithm

- The strengths of the algorithm were …
- The limitations of the algorithm were …
- The lessons learnt were …

Limitation: PPO trains a stochastic policy in an on-policy way. This means 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.

C Project organisation

- The project was organised as well as you would expect in a global pandemic …

VI CONCLUSIONS (1 PAGE)

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

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

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.

Kaelbling, L. P., Littman, M. L. & Moore, A. W. (1996), 'Reinforcement learning: A survey', *Journal of Artificial Intelligence Research* **4**, 237–285.
URL: *https://doi.org/10.1613/jair.301*

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.
URL: *https://doi.org/10.1038/nature14236*

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.
URL: *https://doi.org/10.1007/978-0-387-30164-8*

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

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