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 14, 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

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. The hope is that the network learns abstractions of the games' mechanics, which gets translated and used to make learning other games with similar features more efficient.

B   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. 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 reinforcement learning 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 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 2D 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* ().

D   Achievements

II RELATED WORK

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.

There has also been similar successes in transfer learning for reinforcement learning ...

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 (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 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 that we wouldn't need to create our own API for Atari emulations, we used Gym, 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

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

7

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 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.

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:

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

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

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 advantage estimation equation:

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

---

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

9

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.

E   Implementation features

typically taking multiple steps of (mini-batch) SDG to maximise 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.

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.

- 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.

F   *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 . . .

G   *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 (4 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

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.

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**.
URL: *http://arxiv.org/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.
URL: *https://doi.org/10.1613/jair.301*

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**.
URL: *http://arxiv.org/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.
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., 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**.
URL: *http://arxiv.org/abs/1707.06347*

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

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

Watkins, C. J. C. H. & Dayan, P. (1992), 'Q-learning', *Machine Learning* **8**(3-4), 279–292.
URL: *https://doi.org/10.1007/bf00992698*

Williams, R. J. (1992), 'Simple statistical gradient-following algorithms for connectionist reinforcement learning', *Machine Learning* **8**(3-4), 229–256.
URL: *https://doi.org/10.1007/bf00992696*