

---

# RL Project

---

**David S. Hippocampus**  
Department of Computer Science  
University of Bath  
Bath, BA2 7AY  
hippo@bath.ac.uk

**Christopher Burton**  
Department of Computer Science  
University of Bath  
BA2 7AY  
cb2516@bath.ac.uk

## 1 Problem Definition

## 2 Background

### 2.1 DQN (Mirco)

One category of reinforcement learning methods that can be used to solve the game of Breakout is value-based methods which use function approximation. These methods aim to learn a function  $f : (s, a, \theta) \rightarrow R, s \in S, a \in A, \theta \in \mathbb{R}$  to estimate the action-value functions for each state and therefore determine the optimal action  $a$  to take in the state  $s$ .

Deep Q-Network agent (DQN) introduced by Mnih et al., 2015, was the first value-based RL method to achieve and surpass human performances at Breakout making it the first candidate algorithm we decided to implement. Mnih et al., 2015 shows how DQN was able to reach an average score of 401 over 30 games, compared to an average score of 31 reached in the same testing conditions by a tester professional human player.

Earlier attempts, to use nonlinear function approximations to estimate the action value function, were made before the introduction of DQN agent (find reference) but presented limited and unstable learning. DQN breakthroughs addressed these limitations, more specially the utilization of a separate target network to estimate the next state maximum action value function combined with the introduction of an experience replay memory to avoid temporally correlated data during training allowed the agent to achieve stable learning and converge to a close to optimal policy.

DQN, of course, comes with its own limitations, more specifically, the generally slow training requiring the agent to play a number of frames in the order of millions to achieve human performances translate is a slow learning process. Hasselt, Guez and Silver, 2016 showed that, like Q-learning, DQN overestimates the action value functions which may translate in a sub optimal policy. They showed that this overestimation is due to the fact that, during the bootstrapping from the next state (max operation) the same values are used to select and evaluate an action. This limitation was addressed with the introduction of DDQN agents.

### 2.2 A2C (Chris)

A set of alternative methods to value-based reinforcement methods that can be applied to this problem are policy gradient methods. Whilst value-based methods such as DQN learn values of actions and use these estimates to select actions, policy gradient methods do not need to consult a value function and, instead, learn a parameterised policy to select actions (Sutton and Barto, 2018).

In selecting actions using a parameterised policy, policy gradient methods use *action preferences* rather than action values. This distinction is crucial as it confers several benefits to policy gradient methods such as the ability to learn a stochastic optimal policy (Sutton and Barto, 2018). One benefit particularly relevant to Breakout is that the policy can approach a deterministic policy which would

not be possible if action-values were used (Sutton and Barto, 2018). This is relevant to the problem of Breakout as in some situations such as where the ball is close to the paddle, the desired behaviour is for the agent to deterministically move toward the ball to return it and have no chance of moving away from it. Policy Gradient techniques have seen success in high-profile reinforcement learning breakthroughs in game-playing. Most notably, AlphaGo (Silver et al. 2016) used the REINFORCE algorithm to train the policy network used in the agent.

A Policy-Gradient method that has been shown to be effective on similar tasks to Breakout is the Actor-Critic method. Actor-Critic offers significant benefits over other policy-gradient methods such as REINFORCE. REINFORCE is a Monte Carlo algorithm and as such, suffers from problems such as slow learning and issues with online implementation. In contrast, Actor-Critic methods are analogous to TD methods and so avoid these issues (Sutton and Barto, 2018). Avoiding the issue of slow learning is particularly crucial for a complex environment such as Breakout. Training a DQN agent capable of performing above average performance on Atari games required a training time of 8 days on a GPU (Mnih et al. 2016). For a small team and the time window on this assignment, an agent that learned slower than this would not be feasible. Variants of Actor-Critic including Asynchronous Advantage Actor-Critic (A3C) and Advantage Actor-Critic (A2C) have shown strong performance on similar tasks to Breakout such as in the results of Minh et al. (2016).

### 2.3 Async Q-Learning (Peter)

Mnih et al (2016) present asynchronous variants of reinforcement learning algorithms including both Q-Learning and Actor-Critic (A3C). The paper claims the asynchronous variants can produce excellent results on single multi-core CPUs with training times of just a few hours.

Unlike DQN, the asynchronous methods do not require an experience replay memory because multiple agents working in parallel means that the data is fragmented and is sufficiently decorrelated.

The results published by Mnih et al (2016) are impressive and show that after 10 hours training with breakout, Async Q-Learning scored 240 and A3C scored 460, whilst DQN only managed 25.

TODO : Weaknesses? Check publications for A3C vs A2C, I've seen comments that say A3C does not improve on A2C, but can't find anything definitive. ??

## 3 Method

### 3.1 DQN (Mirco)

As already mentioned in section 2, DQN was derived from Q-Learning where the action-value function is estimated by a non linear function, more precisely, by a deep neural network, namely,

$$Q(S, A) = f(S, A, \theta) = \hat{Q}(S, A, \theta) \quad (1)$$

DQN as well as Q-Learning, uses bootstrapping to update the action value function as shown in the equation below.

$$Q(S, A) = \hat{Q}(S, A, \theta) + \alpha * \left( r + \gamma \max_a \hat{Q}(S', a, \theta) - \hat{Q}(S, A, \theta) \right) \quad (2)$$

where  $\alpha$  is the learning rate and  $\gamma$  is the discounting factor for future rewards.

With tabular RL methods, the learning process is achieved by directly updating the action value function correspondent to each state-action pair while the agent interact with the environment. With function approximation methods, the learning happens by updating the function parameters used to estimate the action value functions. More specifically, the function parameters are updated in the direction that minimize the loss between the current action value function estimation and its update via bootstrapping. The weight update is achieved by standard gradient descent.

$$\theta_{t+1} = \theta_t + \alpha \left( r + \gamma \max_a \hat{Q}(S', a, \theta_t) - \hat{Q}(S, A, \theta_t) \right) \nabla_{\theta_t} \hat{Q}(S, A, \theta_t) \quad (3)$$

DQN, as well as, Q learning belong to a class of algorithm called semi-gradient methods. These methods assume the next state action value function estimate does not depend on the value of the

function parameters. As it is shown in the equation above, the derivation of the term  $\max_a \hat{Q}(S', a, \theta_t)$  is simply neglected as it is considered to be a constant.

From the equation above it can be noted that by changing the network weights at each step  $t$ , the action value function  $\hat{Q}(S, A, \theta_t)$  will be updated towards a moving target as the next state maximum action value function will also change.

One of the major DQN innovations, as briefly mentioned in section 2, is the introduction of two separate networks. A q-network to estimate the action value function of state  $S$  and a separate target network to estimate the next state action value function. The weights of the target network are kept constant for a certain number of iterations before being synchronised with the q-network weights. By using a constant target network, the q-network can be updated towards a stable target removing the cause of instability.

DQN also uses a replay memory buffer to store the last  $N$  interactions with the environment. At each step, a subset  $n$  of experiences  $\in N$  is randomly selected and used to update the q-network weights. This allows the agent to train on time-uncorrelated data.

DQN, as well as Q-Learning, is an off-policy method. A behavioural  $\epsilon$ -greedy policy is used for exploration while the agent policy is updated by taking choosing the best action  $A^*$  (leading to the maximum future expected reward) from all available actions in the next states. The exploration factor  $\epsilon$  is reduced from an initial value of 1 to 0.1 over 1M steps and kept constant thereafter.

The loss function used to calculate the weight updates is the Huber loss which improves stability by clipping the gradient of the loss with respect to the estimated action value function. Mnih et al., 2015 shows how this improves stability during training.

TO FINISH: ++ network time CNN ++ input preprocessing ++ frame skipping ++ state history

### 3.2 A2C (Chris)

The second method chosen to solve our specific problem of Atari Breakout is Advantage Actor-Critic (A2C). Whilst several variations of Actor-Critic methods are used in practice, all Actor-Critic methods are comprised of two key components at their core: an Actor and a Critic. The role of the actor is to learn a parameterised policy which is used to select the actions. The critic evaluates state-action pairs through a learned value function which provides a reinforcing signal to the actor. This signal is used by the actor as part of its evaluation of actions selected and thus, update the parameters of the policy. (Graessner and Loon, 2020)

In this implementation, the actor policy is parameterised by a convolutional neural network (CNN) where the parameters are the weights of the neural network. The critic is also implemented as a convolutional neural network. The neural network architecture for both the actor and the critic is drawn from the architecture used in Minh et al. (2015) with some notable changes. For the Actor, the output layer uses a softmax activation function for the output layer with a single neuron for each possible action that the actor can take. This is to achieve a policy parameterisation known as softmax in action preferences (Sutton and Barto, 2018). The softmax output is then sampled in order for the agent to take an action in a given state. The critic network uses the same architecture but uses a single output neuron with linear activation. This output value represents the action-value the critic prescribes to a given state. Both networks also use a RandomUniform initialisation to the output layers with a relatively small range between min and max values (0 - 0.02). This was included as initial runs of the algorithm showed high sensitivity to initial weights on the output layers, particularly on the actor.

The input to both networks is a pre-processed 84x84x4 image that represents a state in the Breakout environment. The preprocessing applied is the same as that applied in Minh et al. (2015), including greyscale, image cropping and frame-skip. The 4 channels of the image are implemented using the StateHistory class. The instance of StateHistory passed to the agent as input includes the current frame and the 3 previous frames. This gives the agent context to the current frame such as the direction the ball is travelling in.

The issue with vanilla Actor-Critic is that it is usually implemented as an on-policy learning algorithm and on-policy learning algorithms are generally unstable due to temporally correlated data across time-steps (Li, Bing and Yang, 2018). This was addressed by Minh et al. (2016) through the introduction of Asynchronous Advantage Actor-Critic (A3C). In A3C, the asynchronous component

is that multiple agents are executed in parallel on multiple instances of an environment with the aim of removing the non-stationarity since agents will be experiencing a variety of different states at any one time step (Minh et al. 2016). The Advantage component of A3C refers to the use of an advantage function in the reinforcing signals sent to the actor. This advantage value quantifies how much better or worse an action is than the average available action by using the critic’s valuation of states (Graesser and Loon, 2020). An algorithm that implements the advantage function but does not execute multiple agents in parallel is known as Advantage Actor Critic (A2C).

Results from Wu et al. (2017) indicate that A2C may be a more suitable candidate for the problem of Breakout than A3C. Indeed on a set of Atari games, including Breakout, their results showed that A2C outperformed A3C and the noise introduced by the asynchronous implementation failed to deliver any performance benefit (Wu et al. 2017). As such, A2C is chosen as the Actor-Critic implementation.

The inclusion of baselines in policy gradient is to decrease variance and thus improve the stability of training (Sutton and Barto, 2018). In the case of A2C, the state-value produced by the critic is used as the baseline. The update rule for A2C is given in Yoon (2019) and shown below:

$$\nabla_{\theta} J(\theta) = \sum_{t=0}^{T-1} \nabla_{\theta} \log \pi_{\theta}(a_t | s_t) A(s_t, a_t) \quad (4)$$

Where:

$$A(s_t, a_t) = Q_w(s_t, a_t) - V_v(s_t) = r_{t+1} + \gamma V_v(s_{t+1}) - V_v(s_t) \quad (5)$$

The critic uses a TD update to adjust its estimate of the value of a state. The intuition behind which is as follows: Each step of the environment, the critic network calculates a value estimate for both the previous state and the next-state that the agent is in. In a similar fashion to TD(0), the critic uses the bootstrapped estimate of the next state, the reward at timestep t+1 and the discount factor to calculate a target value for the state. The mean squared difference between the critic’s initial value of the state and the target value is used to update the critic and force it to adjust its estimate of the state closer to the target value. The target TD value is also used in the calculation of the advantage function and, hence, the actor loss. As seen in the above equation, The advantage function is calculated by subtracting the critic’s valuation of the previous state from the target TD update. The intuition behind this is that the difference between these two values represents the value of taking the action that the actor took in that state against the average value of that state (Yoon, 2019).

The update equation above is calculated for the actor. The first component of the loss function represents the gradient (delta), i.e whether the agent went with or against the current policy, which sets the direction of the update for each action. This tells the actor to move the parameters toward the gradient for more of the current policy behaviour or opposite to the gradient for less of the current policy behaviour (Steinbach, 2018). For a discrete action space, the log term represents the negative log probabilities for action selection. This has the affect of a stronger or weaker reinforcing signal depending on the confidence of the action selection since if the agent chooses an action with a higher certainty, then the negative log probability for that action will be larger. Log probabilities are used in place of actual probabilities since log probabilities do not tend to 0 when multiplied and thus, provide better numerical stability during training. The delta term implementation follows an extended version of the method followed by Wang (2021).

The advantage, which represents the reinforcement signal from the critic, tells the actor whether the action taken was better than the average reward for that state. By combining these values, the agent is encouraged to reinforce the current policy if it went with the current policy and the action taken resulted in a value better than the average for that state or the agent went against policy which resulted in a lower value. Conversely, the agent is encouraged to update away from the current policy if actions went with current policy and resulted in a worse value than average for the state or if the agent went against current policy and the resulting value was better than the average for the state.

In this implementation, the critic loss is implemented using mean squared loss of the TD update. However, the loss for the actor is calculated in a custom loss function, *actorcustomloss*. This function implements the update described above along with techniques useful for ensuring stability in training such as clipping values. However, the loss function implemented here also includes an entropy regularisation term. This yields a new update equation for the actor update.

$$\nabla_{\theta} J(\theta) = \sum_{t=0}^{T-1} -\nabla_{\theta} \log \pi_{\theta}(a_t|s_t) A(s_t, a_t) + \beta H_t \quad (6)$$

Here,  $\beta$  is the entropy weight, which is a hyperparameter that controls how much the actor loss comprises of the entropy value. The entropy term was included as initial runs showed that even with very small learning rates (1e-5 and below), the agent would find a local minima, selecting one action every time, making it incapable of sampling other actions. The entropy term, originally proposed by William and Peng (1991) for use with the REINFORCE algorithm, encourages the actor to output a distribution with higher entropy by penalising low entropy outputs through the loss function. A distribution of actions with higher entropy is a distribution with a high chance of sampling each of the different values. For example, in an environment with four actions such as Breakout, a maximum entropy output distribution would have a 0.25 probability of selecting each action.

### 3.3 Async Q-Learning (Peter)

Async Q-Learning was chosen as it was similar to DQN, but promised improved learning in a shorter time. Pseudo code is shown in Appendix C: Async Q-Learning pseudo-code

Like DQN, Async Q-Learning uses an e-greedy policy based on a value network to select actions to take and then uses a target network to get an action value used to calculate the loss. The weights in the target network are occasionally reset to match the value network.

Where Async Q-Learning differs from DQN is that it runs multiple worker agents in separate processes which share the value and target networks. Each worker agent keeps a local copy of the value and target networks to calculate the losses and gradients which it then applies to the shared networks every few steps. Once the worker agent has updated the shared network, it resets the weights in its own copies to match shared ones. As the worker agent takes a number of steps before updating the shared network, the updates are done like mini-batches and the chances of one worker agent overwriting changes being made by another are reduced.

Having multiple worker agents running at the same time, and the non-deterministic nature of breakout means that different areas of the game will be explored at the same time. The different worker agents were also given different epsilon values for their e-greedy policies to further increase the variety of exploration.

The e-greedy policies for the worker agents all started with an epsilon of 1.0, which decayed down to 0.05, 0.1 or 0.1 after 1,000 episodes.

A controller process handles the creation of multiple workers as different processes. The controller gives each worker a queue which allows communication between controller and workers. Each worker starts the atari environment and plays games of breakout. An episode is a complete game, and at the end of each episode the worker sends some details to the controller.

The controller keeps track of the total number of episodes completed by all the workers, and uses another process to play games with a policy based on the shared value network with an epsilon of 0.01 to provide a measure of how well the policy performs. The graphs in the results section use

TODO : Maybe a diagram of the different processes would help here?

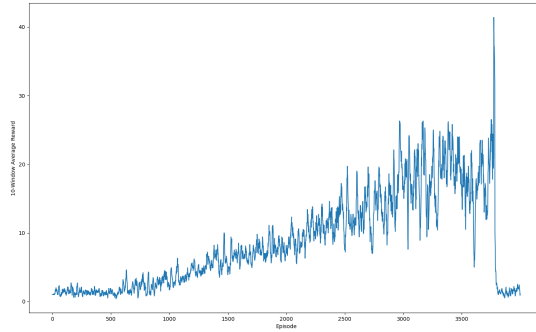
## 4 Results

### 4.1 DQN (Mirco)

### 4.2 A2C (Chris)

*Figure 4.1: 10-episode running average for Breakout A2C agent over 4000 episodes*

In the above figure, a running average of performance over a window of 10 episodes is used to provide stability in estimating agent performance. From episode 1000, the agent shows steady and notable improvement, maintaining a running average of over 10 points by episode 2500. The agent performance peaks around episode 3750 with a 10 game average of 41.4 and peak single game



performance of 233. Beyond this point the agent shows evidence of overtraining. The performance significantly deteriorates to the level seen in the earliest episodes. Observing the softmax output from the actor during this period shows the agent is outputting almost the same probabilities for every state with very little change in the distribution between states or between episodes. Thus, the trained agent in the assignment video uses a weight checkpoint from earlier in the training before overtraining.

### 4.3 Async Q-Learning (Peter)

TODO : Add images for the training using asynch Q-Learning. Show score against number of frames processed.

TODO : add a graph showing 10 hours training of asynch Q-Learning ?

## 5 Discussion

### 5.1 DQN (Mirco)

### 5.2 A2C (Chris)

Significant literature exists on the training of different reinforcement agents on Atari learning environments. This section compares the results obtained from this implementation with implementations in the literature along with a comparison against a baseline of performance from a professional human game tester.

Human (Baseline)	A2C Implementation	Wu et al. (2017) A2C (50mil timesteps)
31.8	41.4	581.6

Table 5.1: Col 1- Human performance on Breakout from a profession games tester (Mnih et al., 2015). Col2 - Peak performance of A2C implementation in this assignment averaged over 10 episodes. Col3 - Performance of A2C implementation in Wu et al. (2017) after 50 million timesteps averaged over 100 episodes

As shown in Table 5.1, using an average over 10 games, the A2C implementation in this assignment was able to significantly outperform a professional human game tester, reaching 130.19% of human performance. This was achieved in 3750 episodes of training which equated to approximately 8 hours training time on CPU. As the implementation outperforms the human baseline, this indicates the agent has solved the chosen problem relatively well and achieves superhuman performance.

However, the implementation does not come close to achieving the 100 game running average of 581.6 achieved by the A2C implementation in Wu et al. (2017). Thus, whilst the agent does solve the problem effectively enough to achieve superhuman performance, it does not match similar implementations in the literature. The agent in this implementation begins to sink into local minima significantly before the 50 million timeframe point. Interestingly, across several runs, the instability and subsequent local minima appear after a particularly high score (In this case 233), indicating that

a potential cause for the deterioration in performance is a very large update to the networks that the agent is unable to properly incorporate. This suggests that, given more time and resources in the assignment, a potential route to match the high scores reached in the literature is to reduce the learning rate and training the agent for significantly longer. Unfortunately, the 50 million time-step training will require the availability of more suitable hardware than is available to the group in this assignment.

A2C Implementation	Wu et al. (2017) A2C (50mil timesteps)
3774	14464

*Table 5.2: Col 1- Episodes required for 10 episode average to surpass human performance Col2 - Episodes required for Wu et al. (2017) A2C implementation to surpass human performance over 100 episode average.*

Table 5.2 compounds the idea that the agent implemented in this paper learns quickly but that this may be the cause of the instability. Whilst the episode average windows are of different sizes, the difference number of episodes required for the A2C to surpass human performance over the chosen window is relatively large. This suggests that although the agent is unable to match the final performance of similar implementations in the literature, the agent in this implementation learns quickly. This, again, suggests that with more time and resources, a lower learning rate and a greater training time may allow this implementation to reach scores comparable to those in the literature.

### 5.3 Async Q-Learning (Peter)

Initial results were encouraging, but the performance tended to level off quite quickly and often started to decrease.

Asynch Q-Learning did not produce results in the time scales mentioned by Mnih et al (2016). However, matching the scores on the data efficiency and training time graphs from Mnih et al (2016) show that they completed approx 25 epochs in 10 hours. A single epoch corresponds to 4 million frames giving 100 million frames in 10 hours. In contrast our implementation for asynch Q-Learning only managed to process 5 million frames in 10 hours, so approximately 20 times slower.

TODO : put these numbers into a suitable table ??

## 6 Future Work

### 6.1 A2C (Chris)

In the A2C implementation used, the advantage function is calculated using  $n$ -step (Where  $n=1$ ) returns. This only includes one step of actual rewards. Since rewards are generated from a single trajectory they tend to have high variance but are unbiased. However, the state valuation reflects all trajectories seen so far and thus has lower variance but introduces bias. As a result, the value of  $n$  controls the bias-variance tradeoff (Graesser and Loon, 2020). Thus, using 1 step returns provides us with a tradeoff of low variance but introduces significant bias through the reliance on state value.

In contrast, Generalised Advantage Estimate (GAE) overcomes the need to set  $n$  as a hyperparameter by using an exponentially-weighted average of advantages calculated with different values of  $n$  (Schulman et al., 2015). GAE provides a method by which the low-variance, high-bias one step advantage is most heavily weighted but the lower-weighted contributions from higher values of  $n$  still contribute. By tweaking the decay rate of contribution from these higher-variance estimators with higher values of  $n$ , GAE allows a more finely controlled tradeoff between bias and variance than the hard cut-off tradeoff in  $n$ -step methods (Graesser and Loon, 2020). In comparison to the 1-step method implemented, this is likely to allow a significant reduction in bias whilst controlling the increase in variance. By testing several values for the GAE decay rate, it is possible that a lower-bias, higher-variance solution may exist that performs better than the  $n$ -step advantage implementation.

As indicated in section 5, the evidence from training time and stability in this implementation against the literature also suggests that worthy future work to extend this implementation would be to train the agent for significantly longer with a lower learning rate. However, this may require hardware above what is available to this group for the assignment.

## 7 Personal Experience

### 7.1 A2C (Chris)

In the training of the A2C agent, one factor that was both a difficulty and a surprise was the sensitivity of the agent to changes in the algorithm hyperparameters. For example, the entropy weight was found to be a very important and sensitive hyperparameter. If set slightly too low the agent would revert to selecting a single action with certainty every time. If set too low the agent would not move significantly from a random sample across the four actions. However, the difference between these outcomes was often a very small change. The value used in the implementation was 0.06. A value of 0.075 was found to get stuck very close to an even weighting of actions for all states (0.25 uniformly) and a value of 0.045 would get stuck in a local minima of selecting only one action. Thus, with only a difference of 0.015 either side of the value used, the performance did not just degrade slightly but became practically untrainable. It is, however, possible that this sensitivity may have been mitigated at the cost of increased training time by reducing the learning rate of the networks.

## References

- Graesser, L.H. and Keng, W.L., 2019. *Foundations of deep reinforcement learning: theory and practice in Python*. First edition. Pearson addison-wesley data & analytics series. Boston: Addison-Wesley.
- Hasselt, H. van, Guez, A. and Silver, D., 2016. Deep Reinforcement Learning with Double Q-Learning. *Proceedings of the AAAI Conference on Artificial Intelligence* [Online], 30(1). Available from: <https://doi.org/10.1609/aaai.v30i1.10295> [Accessed 31 December 2022].
- Li, S., Bing, S. and Yang, S., 2018. *Distributional Advantage Actor-Critic*. [Online]. Available from: <https://doi.org/10.48550/ARXIV.1806.06914> [Accessed 30 December 2022].
- 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. and Hassabis, D., 2015. Human-level control through deep reinforcement learning. *Nature* [Online], 518(7540), pp.529–533. Available from: <https://doi.org/10.1038/nature14236>. [Accessed 27th December 2022]
- Schulman, J., Moritz, P., Levine, S., Jordan, M. and Abbeel, P., 2015. *High-Dimensional Continuous Control Using Generalized Advantage Estimation* [Online]. Available from: <https://doi.org/10.48550/ARXIV.1506.02438> [Accessed 30 December 2022].
- Silver, D., Huang, A., Maddison, C.J., Guez, A., Sifre, L., van den Driessche, G., Schrittwieser, J., Antonoglou, I., Panneershelvam, V., Lanctot, M., Dieleman, S., Grewe, D., Nham, J., Kalchbrenner, N., Sutskever, I., Lillicrap, T., Leach, M., Kavukcuoglu, K., Graepel, T. and Hassabis, D., 2016. Mastering the game of Go with deep neural networks and tree search. *Nature* [Online], 529(7587), pp.484–489. Available from: <https://doi.org/10.1038/nature16961>. [Accessed 23rd December 2022]
- Steinbach, A. 2018. *RL introduction: simple actor-critic for continuous actions* [Online]. Medium. Available from: <https://medium.com/@asteinbach/rl-introduction-simple-actor-critic-for-continuous-actions-4e22afb712> [Accessed 22nd December 2022]
- Sutton, R.S. and Barto, A.G., 2018. *Reinforcement learning: an introduction*. Second edition. Adaptive computation and machine learning series. Cambridge, Massachusetts: The MIT Press.
- Williams, R.J. and Peng, J., 1991. Function Optimization using Connectionist Reinforcement Learning Algorithms. *Connection Science* [Online], 3(3), pp.241–268. Available from: <https://doi.org/10.1080/09540099108946587>. [Accessed 27th December 2022]
- Wu, Y., Mansimov, E., Liao, S., Grosse, R. and Ba, J., 2017. *Scalable trust-region method for deep reinforcement learning using Kronecker-factored approximation* [Online]. Available from: <https://doi.org/10.48550/ARXIV.1708.05144> [Accessed 30 December 2022].
- Wang, M. 2021. *Advantage Actor Critic Tutorial: minA2C* [Online]. Medium. Available from: <https://towardsdatascience.com/advantage-actor-critic-tutorial-mina2c-7a3249962fc8> [Accessed 24th December 2022].



Wu, Y., Mansimov, E., Liao, S., Radford, A. and Schulman, J., 2017. *OpenAI Baselines: ACKTR & A2C* [Online]. OpenAI. Available from: <https://openai.com/blog/baselines-acktr-a2c/> [Accessed 30 December 2022].

Yoon, C. 2019. *Understanding Actor Critic Methods and A2C* [Online]. Towards Data Science. Available from: <https://towardsdatascience.com/understanding-actor-critic-methods-931b97b6df3f> [Accessed 24th December 2022]

Mnih, V., et al., 2016. Asynchronous Methods for Deep Reinforcement Learning. ArXiv, [Online] 1602.01783. Available from: <https://arxiv.org/pdf/1602.01783.pdf> [Accessed 28 Dec 2022]

## Appendices

### Appendix A: DQN pseudo-code

Taken from the course notes.

- 1: Initialise replay memory  $D$  to capacity  $N$
- 2: Initialise action-value network  $\hat{q}_1$  with parameters  $\theta_1 \in \mathbb{R}^d$  arbitrarily
- 3: Initialise target action-value network  $\hat{q}_2$  with parameters  $\theta_2 = \theta_1$
- 4: **for** each episode **do**
- 5:   Initialise  $S$
- 6:   **for** each step of episode **do**
- 7:     Choose action  $A$  in state  $S$  using policy derived from  $\hat{q}_1(S, \cdot, \theta_1)$
- 8:     Take action  $A$  observe reward  $R$  and next-state  $S'$
- 9:     Store transition  $(S, A, R, S')$  in  $D$
- 10:    **for** each transition  $(S_j, A_j, R_j, S'_j)$  in minibatch sampled from  $D$  **do**
- 11:     
$$y = \begin{cases} R_j & \text{if } S'_j \text{ is terminal} \\ R_j + \gamma \max_{a'} \hat{q}_2(S'_j, a', \theta_2) & \text{otherwise} \end{cases}$$
- 12:      $\hat{y} = \hat{q}_1(S_j, A_j, \theta_1)$
- 13:     Perform gradient descent step  $\nabla_{\theta_1} L_\delta(y, \hat{y})$
- 14:    Every  $C$  time-steps, update  $\theta_2 = \theta_1$

### Appendix B: TD Advantage Actor Critic (A2C) pseudo-code

From Steinbach (2018)

- 1: Randomly initialise critic network  $V_\pi^U(s)$  and actor network  $\pi^\theta(s)$  with weights  $U$  and  $\theta$
- 2: Initialise environment  $E$
- 3: **for** episode = 1,  $M$  **do**
- 4:   Receive initial observation state  $s_0$  from  $E$
- 5:   **for**  $t=0, T$  **do**
- 6:     Sample action  $a_t \sim \pi(a|\mu, \sigma) = \mathcal{N}(a|\mu, \sigma)$  according to current policy
- 7:     Execute action  $a_t$  and observe reward  $r$  and next state  $s_{t+1}$  from  $E$
- 8:     Set TD target  $y_t = r + \gamma \cdot V_\pi^U(s_{t+1})$
- 9:     Update critic by minimising loss:  $\delta_t = (y_t - V_\pi^U(s_t))^2$
- 10:    Update actor by minimising loss:  $Loss = -\log(\mathcal{N}(a|\mu(s_t), \sigma(s_t))) \cdot \delta_t$
- 11:    Update  $s_t \leftarrow s_{t+1}$

### Appendix C: Async Q-Learning pseudo-code

From Mnih et al (2016)

- 1: // Assume global shared  $\theta, \theta^-$ , and counter  $T = 0$
- 2: Initialise thread step counter  $t \leftarrow 0$
- 3: Initialise target network weights  $\theta^- \leftarrow \theta$
- 4: Initialise network gradients  $d\theta \leftarrow 0$
- 5: Get initial stats  $s$
- 6: **while**  $T \leq T_{\max}$  **do**
- 7:   Take action  $a$  with  $\epsilon$ -greedy policy based on  $Q(s, a; \theta)$
- 8:   Receive new state  $s'$  and reward  $r$
- 9:   
$$y = \begin{cases} r & \text{for terminal } s' \\ r + \gamma \max_{a'} Q(s', a', \theta^-) & \text{for non terminal } s' \end{cases}$$
- 10:   Accumulate gradients wrt  $\theta$ :  $d\theta \leftarrow d\theta + \frac{\delta(y - Q(s, a, \theta))^2}{\delta\theta}$
- 11:    $s = s'$
- 12:    $T \leftarrow T + 1$  and  $t \leftarrow t + 1$
- 13:   **if**  $T \bmod I_{\text{target}} == 0$  **then**
- 14:     Update the target network  $\theta^- \leftarrow \theta$
- 15:   **if**  $t \bmod I_{\text{AsyncUpdate}} == 0$  or  $s$  is terminal **then**
- 16:     Perform asynchronous update of  $\theta$  using  $d\theta$

17:        Clear gradients  $d\theta \leftarrow 0$

**Appendix D:**