# RL Project

**Peter Coates**
Dept of Computer Science
University of Bath
Bath, BA2 7AY
pmc52@bath.ac.uk

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

**Mirco Carciani**
Dept of Computer Science
University of Bath
Bath, BA2 7AY
mc2886@bath.ac.uk

## 1 Problem Definition

For this assignment, we chose to implement and train 3 different RL agents to play the game of Breakout, a 2D game part of the classic Atari 2600. In the game of Breakout, a player controls a paddle to bounce a ball against a wall composed by a stack of 6 layers of bricks without letting the ball fall behind the paddle. Each frame represents the environment state at step $t$. The reward is 0 for step $t$ unless a brick is destroyed. Each time the ball hits a brick, the brick is destroyed and the score is increased by 1,4 or 7 points, depending on the colour of the brick which was hit. If the ball falls behind the paddle one life is lost. The game ends when all 5 lives are lost. Four actions are available at each step $t$, namely, fire, no operation (noop), move left and move right. For Breakout v5, there is a 75% chance that the environment accepts the action chosen at time $t$ and 25% the action chosen at step $t-1$ is used. For v4, the environment always accepts the action.

The agent must learn to control the paddle to maximise the total reward, meaning hit and destroy as many bricks as possible within 5 lives. No points are removed when a life is lost. Breakout's vast state space proves very challenging for RL methods. The agent needs to figure out which of the available actions is best to take in each state to maximise the future reward. For the scope of this assignment, we will only consider the single player version. The highest score achievable for one player is 864; this is done by eliminating two screens of bricks worth 432 points per screen.

## 2 Background

### 2.1 DQN

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 $f : (s, a, \theta) \to \mathbb{R}$, with $s \in S$, $a \in A$ and $\theta \in \mathbb{R}^n$ to estimate the action-value functions for each state and therefore determine the optimal action $a$ to take in state $s$.

Deep Q-Network agent (DQN) introduced by Mnih et al. (2015), was the first value-based RL method to surpass human performances at Breakout making it the first candidate algorithm we decided to implement. Mnih et al. (2015) show how DQN was able to reach an average score of 401, compared to an average score of 31.8 reached in the same testing conditions by a professional human tester.

DQN introduced both a separate *target network* to estimate the next state maximum action value function and *experience replay memory* to avoid correlated data during training. These introductions allowed DQN to achieve stable learning and converge towards an optimal policy.

### 2.2 A2C

An alternative to value-based methods are policy gradient methods. Policy gradient methods do not need to consult a value function and, instead, learn a parameterised policy to select actions (Sutton and Barto, 2018). Policy gradient methods use *action preferences* rather than action values. This confers several benefits to policy gradient methods, such as the ability to learn a stochastic optimal policy and have the policy approach a deterministic policy (Sutton and Barto, 2018).

A Policy-Gradient method that has been shown to be effective on similar tasks to Breakout is Actor-Critic. Asynchronous Advantage Actor-Critic (A3C) was implemented by Minh et al (2016) to mitigate issues with instability due to temporally correlated data across time-steps (Li, Bing and Yang, 2018). 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). A synchronous version of A3C that implements the advantage function but does not execute multiple agents in parallel is known as Advantage Actor Critic (A2C). Whilst both methods provide strong results on Breakout and similar tasks, results from Wu et al. (2017) indicate that A2C is a more suitable algorithm for the problem of Breakout than A3C.

### 2.3 Async Q-Learning

Mnih et al (2016) presented asynchronous variants of Q-Learning. The paper claims the asynchronous variant can produce excellent results on single multi-core CPUs with training times of just a few hours. The results published by Mnih et al (2016) are impressive and show that after 10 hours training with Breakout, Async Q-Learning scored 240, whilst DQN only managed 25.

Unlike DQN, the asynchronous methods do not require a replay memory buffer as multiple agents working in parallel means that the data is fragmented and sufficiently decorellated.

## 3 Method

### 3.1 DQN

DQN was derived from Q-Learning where the action-value function is estimated by a nonlinear function, more precisely, by a deep neural network.

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

Bootstrapping is used to update the action value function as shown in the equation below.

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

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

The function parameters are updated in the direction that minimizes the loss between the current action value function estimation and its update. The weight update is achieved by a single step of gradient descent as described in equation **??**.

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

As a result of the weights update at each step $t$, the action-value function estimation $\hat{Q}(S, A, \theta_t)$ is updated towards a moving target as the next state maximum action value function $\max_a \hat{Q}(S', a, \theta_t)$ will also change. One of the major breakthroughs introduced by Mnih et al. (2015) was the introduction of separate networks. A Q-value network is used to estimate the action value functions of the current state $S$, while a target network is used to estimate the action value functions of the next state $S'$. The weights of the target network are kept constant for a number of steps before being synched with the Q-value network weights. Using a target network means updates are made towards a stable target, limiting instability during training.

DQN uses a replay memory buffer to store the last $N$ interactions with the environment. At each step, the following details are recorded in the memory: state $S$, action $A$, reward $R$, observed next state $S'$ and flag indicating whether or not $S'$ is terminal.

A subset $n \in N$ of experiences is then selected and used to update the Q-value network weights using equation **??**. This allows the agent to train on time uncorrelated data.

Our DQN implementation tests three different methods to retrieve the experiences from the experience replay buffer.

1. Selects $n$ experiences at random from the buffer of capacity $N$.
2. Uses cosine similarity to select experiences that closely matching the next state $S'$. The idea behind the cosine similarity approach is to create a synthetic dejavu.
3. Selects half randomly and half using cosine similarity.

DQN uses an $\epsilon - greedy$ policy to provide exploration whilst choosing the best action $A^*$ 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.

Mnih et al. (2015) show how the Huber loss function used here improves stability during training by clipping the gradient of the loss with respect the estimated action value function.

Two identical convolutional neural networks (CNN) are used as function approximations for the Q values. The raw pixels are preprocessed before being used as input to the CNNs. The output of the CNNs are the Q values for each action available. The preprocessing crops the raw data, resizes to 84x84 pixels and converts to grey scale. To give the CNN the information about the ball trajectory, the CNN input is a stack of the current and 3 previous frames. **??**.

### 3.2 A2C

The second method is Advantage Actor-Critic (A2C). In this implementation, the actor policy is parameterised by a CNN where the parameters are the weights of the neural network. The critic is also implemented as a CNN, with both architectures following a modified version of that used in Minh et al. (2015).The actor output layer uses a softmax activation function with a neuron for each possible action that the actor can take. This achieves softmax in action preferences (Sutton and Barto, 2018). The critic network uses the same architecture but has a single output neuron with linear activation, representing the action-value that the critic prescribes to a given state. RandomUniform initialisation is used with a small range of min and max values (0 - 0.02) 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 similar to that used for DQN, and the preprocessing is implemented using an OpenAIGym provided Atari wrapper (OpenAI, 2022) that follows the guidelines in Machado et al. (2018). The input includes the current and the 3 previous frames.

The critic uses a TD update. At each step, the critic network calculates a value estimate for both the previous state and the next-state that the agent is in. The critic uses a bootstrapped estimate of the next state, the reward at $t + 1$ 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 adjust its estimate of the state closer to the target value. The target TD value is used also to calculate the advantage function for the actor loss by subtracting the critic's valuation of the previous state from the target TD update. The full A2C actor update rule from Yoon (2019):

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

Where:

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

The first component of the actor loss function is the gradient in parameter space (delta). This value represents the action taken and whether it followed or went against the current actor policy. The delta term implementation in this assignment follows an extended version of the method followed by Wang (2021). The advantage component, which represents the reinforcement signal from the critic, tells the actor whether the action taken was better than the average reward for that state (Positive advantage) or worse (Negative advantage). By multiplying the gradient of action selection with respect to the current policy and a positive or negative advantage value, the actor evaluates the action taken and current policy against the valuation states from the critic. The agent is encouraged to reinforce the current policy if the action taken follows the current policy and the action taken resulted in a value better than the average value for that state or the action taken went against policy which resulted in a lower than average value and vice versa. Log probabilities provide better numerical stability as they do not tend to 0 when multiplied.

Initial runs showed that even with very small learning rates (1e-5 and below), the agent would find a local minima, selecting the same action every time. Thus, an entropy term has been added to improve training stability. The entropy term (William and Peng, 1991) encourages the actor to output a distribution with higher entropy (more even probabilities) by penalising low entropy outputs. For Breakout, a maximum entropy output distribution would have a 0.25 probability of selecting each action. The inclusion of the entropy term yields a new update equation below where $\beta$ is the hyper-parameter, entropy weight.

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

### 3.3 Async Q-Learning

The third method chosen is Asynchronous Q-Learning. Like DQN, Async Q-Learning uses a value network to select actions to take and then uses a target network to estimate an action value that is 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 uses local copies of the value and target networks to calculate the gradients, then applies them to the shared networks every few steps as a batch. Once the worker agent has updated the shared network, it refreshes the weights in its local copies to match shared ones.

As the updates are done in batches, it is easy to also implement the n-step variant of Async Q-Learning, which according to Mnih et al (2016) offers further performance improvements. **??**.

Having multiple worker agents means that different areas of the game will be explored at the same time. The different worker agents were given different epsilon values for their $\epsilon$-greedy policies to increase the variety of exploration. Some experimentation was also done using both decaying epsilon values, and learning rates. However, the time taken to run tests prevented a thorough examination.

A controller process handles the creation of, and communication with, multiple workers and a stats collector. Each worker plays games of Breakout letting the controller know the results. The controller keeps track of the total number of episodes completed by all the workers, and uses the stats collector to provide a measure of how well the training is going.

## 4 Results

Both v4 and v5 of the Arcade Learning Environment (ALE) were used during testing. v5 introduces a *repeat_action_probability=0.25* meaning that 25% of the time the action sent by the agent to the Breakout game is ignored, and the previous action is used instead. This makes the environment stochastic and harder to train an agent against. The following results were obtained by training DQN against v5, A2C against v4, and Asynch n-step Q-Learning against both.

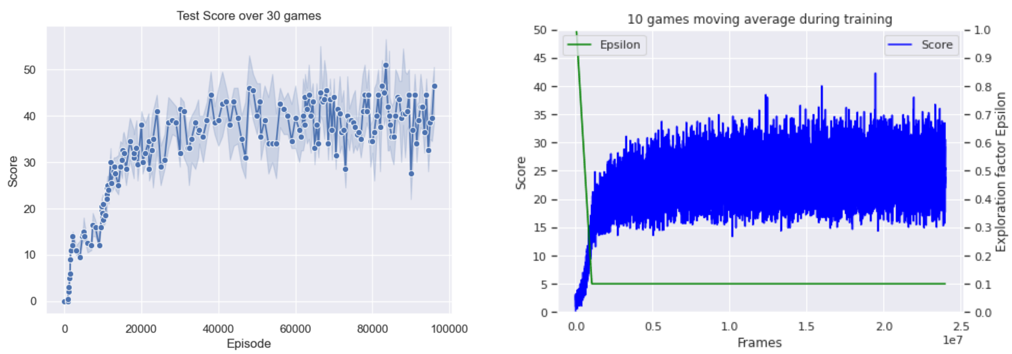
### 4.1 DQN

Different configurations of memory capacity and experience replay were tested for 7M frames.

1. DQN1: N=200k, random experience replay selection.
2. DQN2 :N=1M, random experience selection.
3. DQN3: N=200k, half prioritized experience selection.
4. DQN4: N=500k, full prioritized experience selection.

Picture in **??** shows the test score trajectories of 4 different DQN agent configurations.

Due to the time required to train multiple agents, the training process was resumed up to 24M frames with DQN2 agent only. There was no clear-cut winner from the initial test and DQN2 was chosen because its hyper-parameters mimic the values used by Mnih et al., 2015 which gives a clear baseline for performance comparison.




**(a) Test score over 30 games with epsilon = 0** **(b) 10 games score moving average during training**

**Figure 1. DQN2 performances**

### 4.2 A2C

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. Performance significantly deteriorates to the level seen in the earliest episodes.

Observing the actor's softmax output, we see that the agent is outputting almost the same probabilities for every state with very little change in the distribution between states or between episodes.

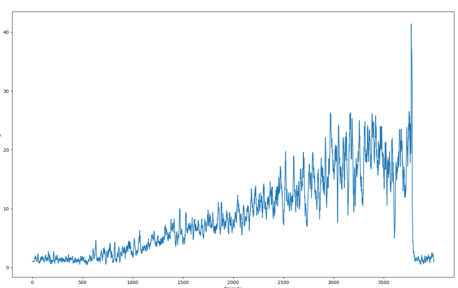
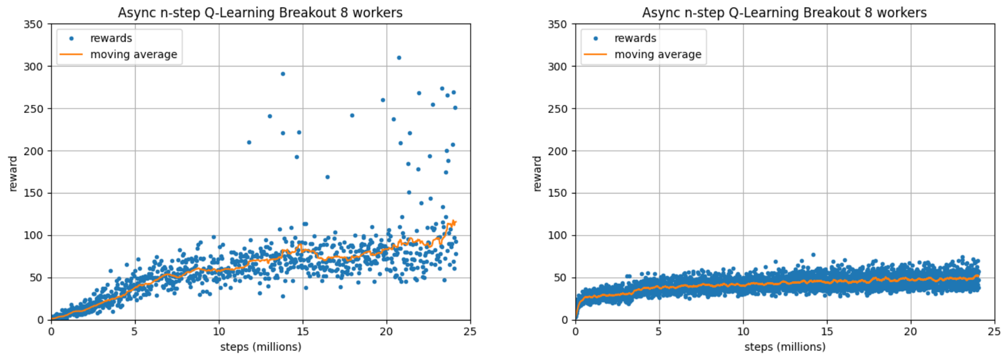


**Figure 2. 10-episode running average for Breakout A2C agent over 4000 episodes**

### 4.3 Async Q-Learning

The figures below show the results from a long run of approximately 40 hours (24 million frames) using Async n-step Q-Learning (n=8). A learning rate of 0.0001, epsilon values in the $\epsilon$-greedy policies varied in different processes and were reduced from a maximum of 0.4 down to 0.1 after approximately a million frames.

The figures clearly show that v5 of the ALE with *repeat_action_probability=0.25* is harder to train against than v4 of the ALE which does not have repeat action. Scores achieved with v5 tended to improve at about half of the rate of scores achieved with v4.




**(a) Run against ALE v4**

**(b) Run against ALE v5**

**Figure 3. Long runs of Async n-step Q-Learning against v4 and v5 of the ALE**

# 5 Discussion

## 5.1 DQN

Mnih et al., 2015 report a maximum average score of 401 after 50M frames. We were not able to replicate the same level of performance. The best average score recorded in our implementation of DQN is 51 at around 21M frames (83500-th episode) Although we could not match Mnih et al., 2015 results, our agent reaches 160% of human performance of 31.8.

It was interesting to notice the early difference in performances amongst DQN agents having different memory buffer sizes and experience replay selection methods shown in Figure**??**. DQN1 and DQN2 differ in the size of the memory buffer (200K vs 1M), but, no appreciable difference was seen between the two. Comparable results were obtained selecting half of the training samples using cosine similarity. An overall degradation in performances was recorded while only selecting experiences which resemble the agent observed state $S'$. This might suggest that by only selecting experiences which are similar to the current observation may introduce a form of data correlation.

## 5.2 A2C

The A2C implementation was able to significantly outperform a professional human game tester (41,4 vs 31,8), 130.19% of human performance. This was achieved in 3,750 episodes (8 hours) of training. The implementation outperforms the human baseline, so, the agent has solved the chosen problem relatively well. Wu et al. (2017) required 14,464 episodes to surpass human performance, whereas our A2C implementation only required 3,774. This shows that our implementation learns quickly.

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). The agent training becomes unstable significantly before the timesteps taken in Wu et al. (2017). 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 is a very large update to the networks that the agent is unable to properly incorporate. This suggests that 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 or implementing reward clipping.

## 5.3 Async Q-Learning

The performance of Async n-step Q-Learning and DQN was similar with both achieving an average score around 50 after processing 24 million frames with v5 of the ALE.

Comparing the results from Async n-step Q-Learning and A2C when trained against v4 of the ALE shows that the configuration used for A2C learns more quickly. Async n-step Q-Learning was slower, but kept on increasing, managing an average score of over 100 after processing 24 million frames.

The results obtained from running Async n-step Q-Learning did not match the results and time scales achieved by Mnih et al (2016). However, they completed approx 25 epochs in 10 hours to get scores in excess of 250. A single epoch being 4 million frames giving 100 million frames in 10 hours. In contrast our implementation is 16 times slower, processing about 6 million frames in 10 hours.

Our results compare favourably with those from Mnih et al (2016). We reached a score of 50 in half the number of frames. However, we have not trained over enough frames to compare final scores.

# 6 Future Work

## 6.1 DQN

Experience replay is a key ingredient in the success of DQN agent. In this assignment random experience replay was implemented, and, although effective, Schaul et al., 2016 work prove that rank-based and proportional based prioritized experience replay improves the agent performances. Narasimhan et al., 2015 also suggest an alternative method of resampling the experiences. The method separates experience into two buckets, one for positive and one for negative rewards. The experiences are then picked from the two buckets by a fixed ratio.

## 6.2 A2C

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, leading to low variance but high bias state valuation estimates (Graesser and Loon, 2020). In contrast, Generalised Advantage Estimate (GAE) uses 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.

## 6.3 Async Q-Learning

Having multiple processes gives scope for using different hyperparameters at the same time. So, it would be interesting to try and produce a version of the code that attempts to tune itself. It would also be interesting to investigate the asynchronous version of advantage actor critic (A3C) to see how it compares to A2C.

# 7 Personal Experience

DQN was quite fiddly to tune. Parameters that seems not to bear a great deal of importance over the training process actually affected results quite heavily. A key factor for the agent learning stability was the frequency at which the Q-value network are updated. I found that updating the agent policy at every step triggered instability after a few hundred thousand frames. By updaing the network weights every four actions, the learning was much more stable. The experience batch size also had a sizable impact on the training outcome. Batches of size 64 presented more unstable behaviour compared to batches of size 32.

With A2C, one factor that was both a difficulty and a surprise was the sensitivity to changes in the algorithm hyperparameters. For example, the entropy weight was very important and sensitive. If set slightly too low the agent would revert to selecting a single action with certainty every time. If set too high, 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 in entropy weight.

Creating a multiprocessing application in python using Keras was awkward, however, PyTorch offered much better support. It was nice to get a chance to try out PyTorch, and from first use, it seems to be a nicer API than keras.

## 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].

Machado, M.C., Bellemare, M.G., Talvitie, E., Veness, J., Hausknecht, M. and Bowling, M., 2018. Revisiting the Arcade Learning Environment: Evaluation Protocols and Open Problems for General Agents [Online]. *Journal of Artificial Intelligence Research* Available from: https://www.ijcai.org/proceedings/2018/0787.pdf [Accessed 31 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]

Narasimhan, K., Kulkarni, T. and Barzilay, R., 2015. Language Understanding for Text-based Games using Deep Reinforcement Learning. *Proceedings of the 2015 Conference on Empirical Methods in Natural Language Processing* [Online], EMNLP 2015, Lisbon, Portugal. Lisbon, Portugal: Association for Computational Linguistics, pp.1–11.
Available from: https://doi.org/10.18653/v1/D15-1001 [Accessed 3 January 2023].

OpenAI. 2022. *ataripreprocessing gym wrapper*. [computer program]. Available from: https://github.com/openai/gym/blob/master/gym/wrappers/ataripreprocessing.py

Schaul, T., Quan, J., Antonoglou, I. and Silver, D., 2016. *Prioritized Experience Replay*.
Available from: http://arxiv.org/abs/1511.05952.

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.

Initialise replay memory $D$ to capacity $N$

Initialise action-value network $\hat{q}_1$ with parameters $\boldsymbol{\theta}_1 \in \mathbb{R}^d$ arbitrarily

Initialise target action-value network $\hat{q}_2$ with parameters $\boldsymbol{\theta}_2 = \boldsymbol{\theta}_1$

**for** each episode **do**

  Initialise $S$

  **for** for each step of episode **do**

    Choose action $A$ in state $S$ using policy derived from $\hat{q}_1 (S, \cdot, \theta_1)$

    Take action $A$ observe reward $R$ and next-state $S'$

    Store transition $(S, A, R, S')$ in $D$

    **for** each transition $\left(S_j, A_j, R_j, S'_j\right)$ in minibatch sampled from $D$ **do**

      $y = \begin{cases} R_j & \text{if } S'_j \text{ is terminal} \\ R_j + \gamma \max_{a'} \hat{q}_2 \left(S'_j, a', \boldsymbol{\theta}_2\right) & \text{otherwise} \end{cases}$

      $\hat{y} = \hat{q}_1 \left(S_j, A_j, \boldsymbol{\theta}_1\right)$

      Perform gradient descent step $\nabla_{\theta_1} L_\delta(y, \hat{y})$

    Every $C$ time-steps, update $\boldsymbol{\theta}_2 = \boldsymbol{\theta}_1$

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

From Steinbach (2018)

Randomly initialise critic network $V^U_\pi(s)$ and actor network $\pi^\theta(s)$ with weights U and $\theta$

Initialise environment $E$

**for** episode = 1, M **do**

  Receive initial observation state $s_0$ from $E$

  **for** t=0, T **do**

    Sample action $a_t \sim \pi(a|\mu, \sigma) = \mathcal{N}(a|\mu, \sigma)$ according to current policy

    Execute action $a_t$ and observe reward $r$ and next state $s_{t+1}$ from $E$

    Set TD target $y_t = r + \gamma \cdot V^U_\pi(s_{t+1})$

    Update critic by minimising loss: $\delta_t = (y_t - V^U_\pi(s))^2$

    Update actor by minimising loss: $Loss = -\log(\mathcal{N}(a|\mu(s_t), \sigma(s_t))) \cdot \delta_t$

    Update $s_t \leftarrow s_{t+1}$

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

From Mnih et al (2016)

*// Assume global shared $\theta$, $\theta^-$, and counter $T = 0$*

Initialise thread step counter $t \leftarrow 0$

Initialise target network weights $\theta^- \leftarrow \theta$

Initialise network gradients $d\theta \leftarrow 0$

Get initial stats $s$

**while** $T <= T_{\max}$ **do**

  Take action $a$ with $\epsilon$-greedy policy based on $Q(s, a; \theta)$

  Receive new state $s'$ and reward $r$

  $y = \begin{cases} r & \text{for terminal } s' \\ r + \gamma \max_{a'} Q(s', a', \theta^-) & \text{for non terminal } s' \end{cases}$

  Accumulate gradients wrt $\theta$: $d\theta \leftarrow d\theta + \frac{\delta(y - Q(s,a,\theta))^2}{\delta\theta}$

  $s = s'$

  $T \leftarrow T + 1$ and $t \leftarrow t + 1$

  **if** $T$ mod $I_{target} == 0$ **then**

    Update the target network $\theta^- \leftarrow \theta$

  **if** $t$ mod $I_{AsyncUpdate} == 0$ or $s$ is terminal **then**

    Perform asynchronous update ot $\theta$ using $d\theta$

Clear gradients $d\theta \leftarrow 0$

**Appendix D: Async n-step Q-Learning pseudo-code**

From Mnih et al (2016)

*// Assume global shared $\theta$, $\theta^-$, and counter $T = 0$*
Initialise thread step counter $t \leftarrow 0$
Initialise target network weights $\theta^- \leftarrow \theta$
Initialise network gradients $d\theta \leftarrow 0$
Initialise thread-specific parameters $\theta' \leftarrow \theta$
Get initial stats $s$
**while** $T <= T_{\max}$ **do**
    Clear gradients $d\theta \leftarrow 0$
    Synchronize thread-specific parameters $\theta' \leftarrow \theta$
    $t_{start} = t$
    Get state $s_t$
    **while** not terminal $s_t$ and $t - t_{start} < t_{max}$ **do**
        Take action $a_t$ with $\epsilon$-greedy policy based on $Q(s_t, a; \theta')$
        Receive new state $s_{t+1}$ and reward $r_t$
        $t \leftarrow t + 1$
        $T \leftarrow T + 1$
    $R = \begin{cases} 0 & \text{for terminal } s_t \\ \max_a Q(s_t, a, \theta^-) & \text{for non terminal } s_t \end{cases}$
    **for** $i \in \{t-1, ..., t_{start}\}$ **do**
        $R \leftarrow r_i + \gamma R$
        Accumulate gradients wrt $\theta'$: $d\theta \leftarrow d\theta + \frac{\delta(y - Q(s,a,\theta\prime))^2}{\delta\theta\prime}$
    Perform asynchronous update of $\theta$ using $d\theta$
    **if** $T$ mod $I_{target} == 0$ **then**
        Update the target network $\theta^- \leftarrow \theta$

**Appendix E: DQN Agent performance comparison**

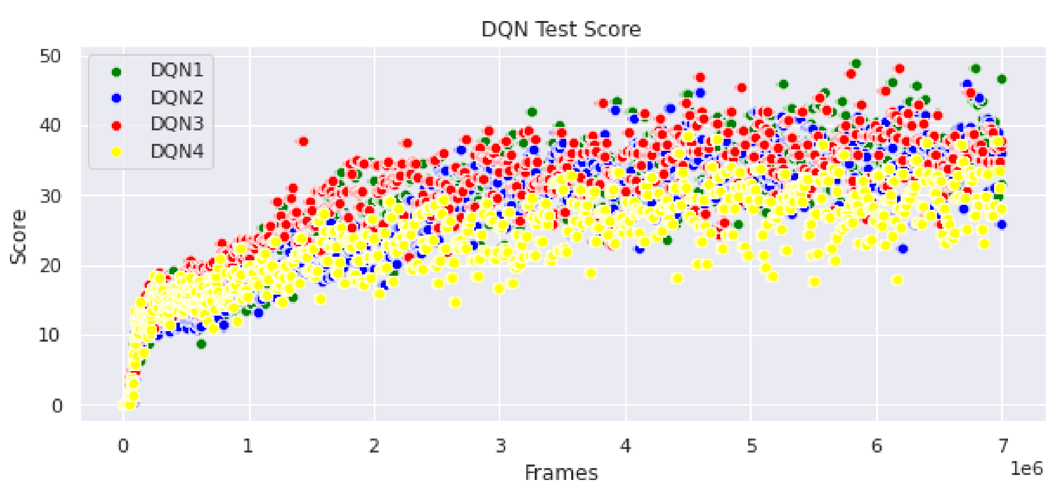



**Figure 4. DQN agent performances comparison. DQN1: N=200k, random experience replay selection. DQN2: N=1M, random experience selection. DQN3: N=200k, half prioritized experience selection. DQN4: N=500k, full prioritized experience selection.**