# CM50270 Reinforcement Learning
# Deep Q-Learning for Atari Breakout

**Kafkalias A., Papayiannis K., Theodosiou M.**
Department of Computer Science
University of Bath
2021

## Abstract

We present a deep reinforcement learning model that learns to play the arcade game Breakout. Our agent is trained on the OpenAI Gym environment using a variant of Deep Q-Learning and a convolutional neural network that takes images of the game as input and outputs the expected reward for each action. We find that our model performs 500% better than the average human score and receives an almost perfect score for the best runs. We compare our model against a second agent trained using the console RAM as input, and suggest several improvements to further increase scores and decrease training times.

## 1 Introduction

In this project we train a reinforcement learning agent to play the game of Breakout using Deep Q-Learning (DQN), drawing inspiration from the success of Mnih et al. (2013). DQN is an algorithm that combines the fields of Deep Learning and Reinforcement Learning, training agents to interact with complex environments such as those present in robotics and video games. Over the past years, video games have gained popularity as domains in which to apply Reinforcement Learning given their well defined rewards and self contained environments.

Atari 2600 games are considered a challenging RL test-bed, providing a visual output of 210 x 160 RGB video at 60 Hz with millions of potential states. Learning to control agents from such complex environments is one of the open research areas within Reinforcement Learning, as traditional tabular methods cannot be used due to the large state space. We evaluate our agents performance against human benchmarks and results from literature, and compare performance against a second model trained using the Atari RAM the state representation.

Our state representation is a 260x160x3 RBG image of the game screen, with rewards given for breaking bricks (see appendix for details). Our action space has four possible actions: paddle left, paddle right, fire (resetting the ball after a lost life or environment reset) and no-op which keeps the paddle stationary. An episode is terminal during evaluation when five lives are lost or all bricks are cleared, while for training we terminate episodes after the loss of one life.

## 2 Method

We use the OpenAI Gym toolkit to provide the environment. This easy to use interface provides functionality to set up and interacting with the environment with minimal additional code, providing state representations at each time step.

## 2.1 Environment Wrappers

We employ a number of wrapper functions imported from the gym library in order to change the default state representations to facilitate learning. The first wrapper used (*AtariPreprocessing*) provides multiple pre-processing steps in conjunction with the above. The standard state representation is a 210x160x3 RGB array, while the wrapper converts this image to grey-scale, resizes it to 84x84 pixels and normalises the pixel values between 0 and 1 as shown in figure 1 below. To prevent any impact from screen flickering, the wrapper also utilises max pooling on any two most recent observations to ensure a meaningful state representation from all time steps.

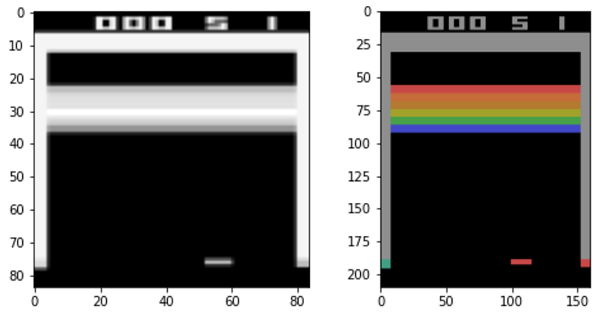

Figure 1: Left: Re-scaled and grey-scaled state representation after pre-processing. Right: Standard RBG state representation.

The second wrapper is the frame stacking wrapper which repeats each action taken for four consecutive time steps, outputting a 3d state representation matrix with dimensions HxWx4. The inclusion of consecutive frames stacked across a third axis provides temporal information about the motion of the ball and paddle, enabling the network to predict the direction and speed of both. This step is crucial as a two dimensional state representation would only provide information about the location of objects without any information regarding their motion, making it impossible for the network to accurately predict the trajectories of objects.

## 2.2 Model architecture and parameters

As mentioned above our wrappers re-scale and stack the frames which results in an input of 84x84x4 to the neural network. We use a convolutional neural network with fully connected layers to approximate the q values of state, action pairs. The first convolutional layer uses 32 8x8 filters with a stride of 4 and a Rectified Linear Unit (relu) activation function. The second convolutional layer uses 64 4x4 filters a stride of 2 similarly followed by relu activation. The final convolutional layer uses 64 3x3 filters with stride 1 followed once again by relu. The output of the final convolutional is flattened and passed to a fully connected layer of 512 neurons with relu activation. Lastly the output layer is fully connected with a linear activation and an output of 4, one for each action. Figure 2 demonstrates the architecture of the model.

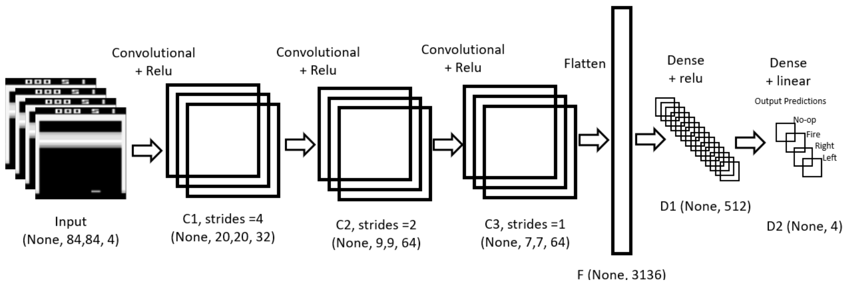

Figure 2: Model Architecture.

Large errors between the expected q value and the predicted q value can result in large shifts to the model weights, which in turn creates imbalances to the policy of the agent. Using the Huber loss can reduce the impact of such outliers due to the linear loss at higher errors. The Adam optimiser is used with a learning rate of 0.001 and gradient clipping to avoid large sudden changes to the model weights due to exploding gradients. An important dilemma for every RL agent is the exploration vs exploitation trade off. In order to allow for exploration, the agent chooses actions randomly for the first fifty thousands frames. We then decay epsilon linearly from 0.9 to 0.1 in the following one million frames.

### 2.3 Learning algorithm

Deep Q-Networks, otherwise known as DQN, are value based, model free, bootstrapping deep Reinforcement Learning methods. DQNs use two networks models; a main model used to calculate the q-values and choose actions, and the target network which is used to calculate the 'target' q-value in order to calculate the error. This implies that the target network parameters must be updated every steps to match the parameters of the main model. In our case, the target model is updated every ten thousands frames, which is low enough to mimic the main network relatively well but not high enough to drop the accuracy.

The agent stores its experience in a buffer, usually known as replay buffer or experience buffer. We train our model by sampling from this buffer with a batch size of 32 every four frames, assuming that the buffer has more than 32 samples to train from. Due to computational expenses, we were forced to limit the buffer size to just fifty thousands experiences as our personal computers could not handle the memory usage and the University of Bath GPU cloud seemed to not work with any buffer above that size. Other popular approaches, like Deepmind's DQN (Mnih et al., 2013) used a buffer size of one million. This may hinder our results, as the agent does not have a wide pool of samples to learn from. In order to help the agent, we force him to choose the Fire action when the game starts. In this way the agent doesn't have to teach itself how to fire which reduces training time.

### 2.4 Model training and evaluation

All training was performed on the University of Bath GPU cloud in order to improve training times thanks to the performance benefits of GPUs for deep learning. During training, the model weights and replay buffer values were saved every 100k frames until 1 million frames, and then every 500k frames until the end of training at 13.5 million frames. This ensured that model weights were available for investigating the performance at various points in the training process, and ensured that training could be continued from any of these checkpoints in the event of any issues with the GPU cloud, which happened several times.

We later used these stored model weights to reproduce the runs of the agent at each time step. Our evaluation process was run in Google Colab due to its interactive environment, and involved running 100 episodes for each model to determine the expected episode rewards. We employed an epsilon greedy policy with e = 0.01 during evaluation to calculate the average, minimum and maximum episode rewards. States of episodes producing the maximum reward at each checkpoint were stored in order to animate them and investigate learned behaviour over time.

### 2.5 Learning using the system RAM

As well as providing an environment for breakout with screen images as the state representation, Gym provides an alternative environment that uses the state of the console RAM as its state representation. Atari 2600 consoles have 128 bytes of RAM (1024 bits) for scratch space, the call stack, and the state of the game environment. Given the extremely limited memory of the console, game developers often had to encode information using individual bits within each byte of memory. For this reason, our model unpacks the 128 dimensional state representation returned by the environment to create a 1024 dimensional representation of the values of each individual bit of game memory. For this state configuration, we also simplify the action space by removing the fire action, enforcing this action at each life lost and environment reset in order to generate a new ball.

The architecture of this model is different to the above as convolutional networks work best with image data. Instead, a 4 layer Multi-Layer Perceptron (MLP) is used. Rectified Linear Unit activation functions are used for the hidden layers, and a linear activation used for the output layer. The input layer expects an input of 1024 (number of bits) and the three hidden layers contain 256, 256 and 128 neurons respectively. The output layer is fully connected with an output dimension of 3; one for each action except fire.

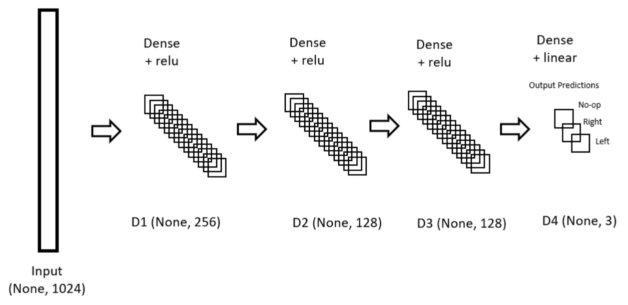

Figure 3: RAM Model Architecture.

## 3 Results and Discussion

### 3.1 Evaluation of the RAM agent

To compare the performance between the RAM and image state representations, we take the mean episode rewards of 100 evaluation runs at various stages of training, shown in table 1 below. To ensure consistency between comparisons, we use the number of lives trained for rather than frames.

Table 1: Episode rewards for RAM vs. Image state representations by training time in lives averaged over 100 runs.

| | Episode Rewards | |
|---|---|---|
| Lives | RAM | Image |
| 5 000 | 12 | 12 |
| 10 000 | 17 | 25 |
| 15 000 | 13 | 63 |
| 50 000 | 31 | 160 |

As can be seen from the table above, the model using screen images significantly outperformed the RAM agent, achieving a mean reward of over five times greater at 50000 training lives. Given these results, it's clear that the screen images provided a much better state representation and hence remained the focus for further training. The assumption that a simple mapping existed between the system memory and relevant state of the game did not hold as the agent was able to learn far more quickly using the images as input.

### 3.2 Evaluation of the screen image agent

Figure 4 demonstrates how the trained DQN adapts its q values between the different states. When the ball comes down at the left of the paddle (left half of figure 5), the q-values drive the agent to move the paddle towards the ball. However, once the ball is bounced away from the paddle (right half of figure 5), it is clear that all actions have similar q-values. This is because the agent learns that all actions return similar rewards in these states.

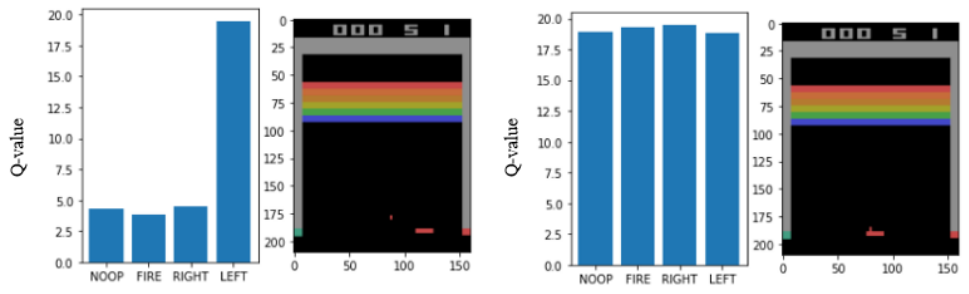

Figure 4: Demonstration of how the Q values change depending on the current state

Figure 5 below shows how the agent performance improves over a training period of 13.5 million frames across approximately 30 training hours. During the first 1 million frames, our agent is mostly exploring and therefore there is not that much increase in score but still surpasses average human performance. Around 4 million frames, there is a big spike in performance indicating that the agent discovered a strategy that yielded a high score. After the 6 million frames mark, it does not appear that the agent made any significant progress, averaging at a score around 150 with a maximum at 358 on the 11 million frame mark.

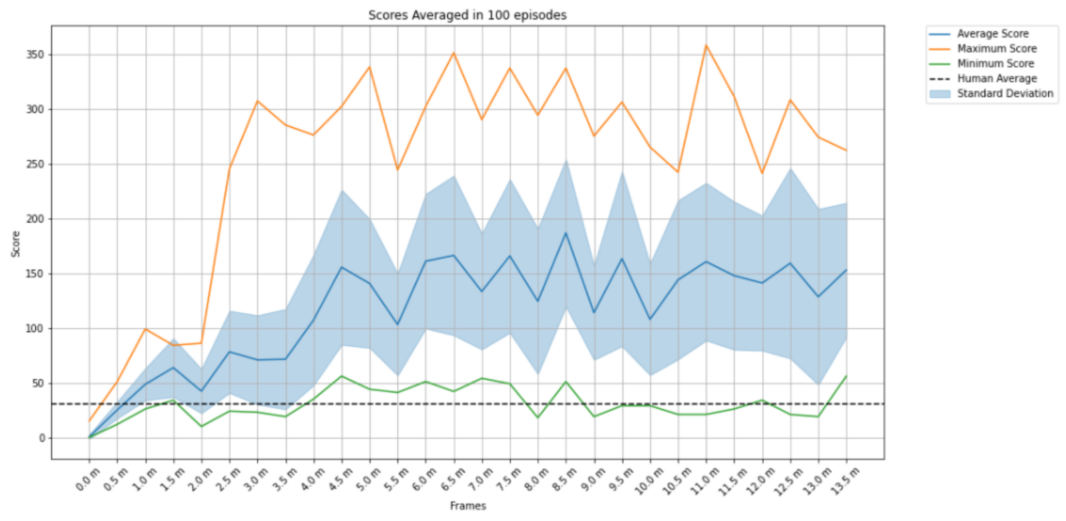

Figure 5: Evaluation of the Agent performance

## 3.3 General Discussion

The maximum reward that can be achieved in any Breakout game is 896, which as achieved by cleaning two screens of bricks. However, in practice any score above 488 is considered a perfect run since the agent knows how to clear a full board of bricks. The average human score (Wang et al. 2016) is around 30. Our agent showcased a professional performance of 500% better than the human average.

At the exploration stage of the agent, it was able to learn the basic aim of the game - to bounce the ball off the paddle to break the bricks. After further training, the agent discovers that by strategically clearing a pathway to the back of the board, it can earn a lot of rewards very fast. Figure 5 shows this increase in performance at around 4 million frames.

Despite significant progress, the agent was not able to achieve perfection as it appears to fail at the final stages of the game. One potential reason for this failure is the small buffer size. The agent does not reach high scoring states very often, so having a small buffer size implies that the final states of the environment are not sampled frequently enough for the agent to learn how to deal with them. As a result, the agent does not sufficiently explore these states, as the high exploration rate has decayed by the time these states have been reached.

The original DQN agent by Deepmind Technologies (Mnih et al. 2013), achieved an average score of 385 with a simpler network architecture (less rectified units in each hidden layer). Their agent was able to outperform our agent. However, their agent was given training time of a week with additional computational power. Given additional training time, our agent could potentially reach the same level of performance as it showcased promising results.

## 3.4 Improvements

*Achieveving a higher score*
Demonstrated by our results, our DQN agent didn't converge to a perfect score. Instead its mean reward seemed to plateau at 150 for approximately 8 million frames. It is possible that if the training continued, the agent could find a better strategy which would result in better performance and more stable runs. Using a buffer with a bigger size would likely improve the performance of the agent since it would have a larger variety of samples to train on.
Furthermore, a disadvantage of the DQN implemented its that it uses both the main network model for calculating the q values and choosing an action. This introduces the following issue; Consider a network that produces an output for two actions, alpha and beta. If action alpha has a higher value, it will be chosen every time, resulting in higher alpha value. If for any reason in the future, action beta becomes the better choice under some conditions, the model will not adapt easily due to the high alpha value. As demonstrated by our results, our network eventually converges but does not know how to completely finish the game after it reaches a very high score. The Double DQN (Van Hasselt et al. 2016) algorithm solves this issue by evaluating the greedy policy according to the main network, but estimating the q value using the target network.
In the case of breakout, the agent must wait for the ball to hit the paddle and then ricochet on the bricks in order to get a reward. This causes a delay between the chosen action and the respective reward. This limits the exploration using our e-greedy policy. However, adding parametric noise to the weight parameters of the neural network (Fortunato et al., 2017) introduces greater variability to the decision making which has potential for even more exploratory actions. In addition, the noise added can be tuned by the algorithm automatically which eases any hyper parameter tuning. This method is known as Noisy Nets.

*Reducing training time*
Another improvement could be the use of Prioritized Experience Replay instead of a replay buffer. Instead of sampling a random state from the replay buffer we select the states which had the most error loss and use them to update the weights of our network. This method helps the agent to learn the optimal policy more efficiently and usually speeds the learning process by a factor of 2.(Schaul et al.,2016)
Furthermore we can use asynchronous methods for Deep RL which will speed up the training process. DQN uses a single agent, represented by a single network and interacts with a single environment. Asynchronous methods use a global network and multiple worker agents which each have their own set of parameters. Every agent interacts with a copy of its environment, and all the agents work together at the same time. In this way, the experience available is more diverse since each agent is independent of the others, speeding up the process as more work gets done. An example of such an agent is the Asynchronous Advantage Actor-Critic (Mnih et al. 2016) which was able to speed up the of DQN by a factor of 2.

Table 2: Raw scores across breakout. Published by Hessel et al. 2018

| Random | Human | DQN | DDQN | Noisy DQN | Prior. | A3C |
|---|---|---|---|---|---|---|
| 1.7 | 30.5 | 385.5 | 418.5 | 459.1 | 381.5 | 681.9 |

## 4 References

Fortunato, M., Azar, M.G., Piot, B., Menick, J., Osband, I., Graves, A., Mnih, V., Munos, R., Hassabis, D., Pietquin, O. and Blundell, C., 2017. Noisy networks for exploration. arXiv preprint arXiv:1706.10295.

Hessel, M., Modayil, J., Van Hasselt, H., Schaul, T., Ostrovski, G., Dabney, W., Horgan, D., Piot, B., Azar, M. and Silver, D., 2018, April. Rainbow: Combining improvements in deep reinforcement learning. In Proceedings of the AAAI Conference on Artificial Intelligence (Vol. 32, No. 1).

Mathias, L., Jacob, C., 2020, May, Deep Q-learning for Atari Breakout, URL : https://keras.io/examples/rl/deep_q_network_breakout/

Mnih, V., Badia, A.P., Mirza, M., Graves, A., Lillicrap, T., Harley, T., Silver, D. and Kavukcuoglu, K., 2016, June. Asynchronous methods for deep reinforcement learning. In International conference on machine learning (pp. 1928-1937). PMLR.

Mnih, V., Kavukcuoglu, K., Silver, D., Graves, A., Antonoglou, I., Wierstra, D. and Riedmiller, M., 2013. Playing atari with deep reinforcement learning. arXiv preprint arXiv: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. and Petersen, S., 2015. Human-level control through deep reinforcement learning. nature, 518(7540), pp.529-533.

Schaul, T., Quan, J., Antonoglou, I. and Silver, D., 2015. Prioritized experience replay. arXiv preprint arXiv:1511.05952.

Van Hasselt, H., Guez, A. and Silver, D., 2016, March. Deep reinforcement learning with double q-learning. In Proceedings of the AAAI Conference on Artificial Intelligence (Vol. 30, No. 1).

Wang, Z., Schaul, T., Hessel, M., Hasselt, H., Lanctot, M. and Freitas, N., 2016, June. Dueling network architectures for deep reinforcement learning. In International conference on machine learning (pp. 1995-2003). PMLR.

## 5 Appendix

### 5.1 Hyper-parameters

Table 3: Environment Hyper-parameters

| Hyper-parameter | value |
|---|---|
| Grey-scaling | True |
| Observation down-sampling | (84,84) |
| Frames stacked | 4 |
| Frames skipped | 4 |
| Reward clipping | False |
| Terminal on loss of life | True |
| Max frames per episode | Unlimited |

Table 4: Agent Hyper-parameters

| Hyper-parameter | value |
|---|---|
| gamma | 0.99 |
| max epsilon | 0.9 |
| min epsilon | 0.1 |
| epsilon random frames | 50000 |
| epsilon greedy frames | 100000 |
| buffer size | 50000 |
| batch size | 32 |
| update main model parameters | 4 frames |
| update target model parameters | 10000 frames |

## 5.2 Reward Shaping

Table 5: Rewards

| Event | reward |
|---|---|
| Clearing 1$^{st}$ & 2$^{nd}$ layer bricks | 1 |
| Clearing 3$^{rd}$ & 4$^{th}$ layer bricks | 4 |
| Clearing 5$^{th}$ & 6$^{th}$ layer bricks | 7 |
| Any other event | 0 |

## 5.3 Code

Our implementation was inspired form the implementation of Mathias  Jacob (2020) . We modified their code to suit the purpose of our problem the best by creating a class of the agent which holds all of the information of the model , hyper-parameters and replay memory , one function for training the agent and one function for evaluating the performance of the agent. Our code is build in a way that can run in a server and return logs of the printing statements and checkpoints of training steps.