



Assignment 3

Reinforcement Learning using DQN applied to Atari

December 2021

Submitted by

Ranya El-Hwigi 18227449

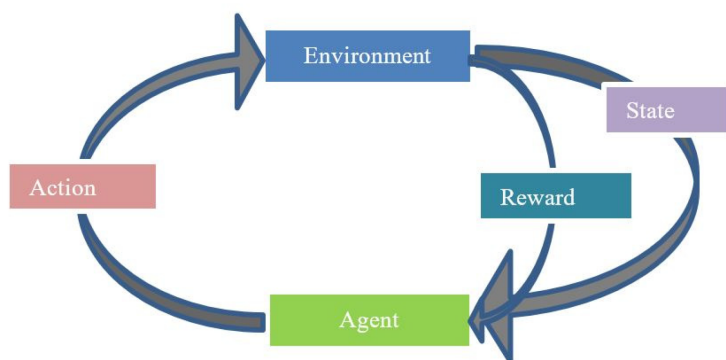
Nutsa Chichilidze 18131956

Table of Contents

Table of Contents	1
Why Reinforcement Learning	2
The Environment	2
Atari game selected	3
Inputs received from the OpenAI Gym environment	3
Control settings for the joystick	4
Implementation	4
Data Preprocessing	4
The Network Structure	6
Network Architecture	6
Hyper Parameters	7
The Q learning update applied to the weights	8
The Bellman Equation	8
Experience Replay	9
Other significant concepts	10
Results	11
Logging	11
Visualisation	11
Huber Loss	12
Rewards	13
Evaluation of Results	13
How to evaluate the performance of the RL agent	13
Is the agent learning?	13
References	14

Why Reinforcement Learning

Reinforcement Learning (RL) is learning through trial and error. It tries to learn an optimal mapping from states to actions, states can refer to a board configuration and the actions would be the permissible actions for a particular state. It's goal is to maximise a long term reward as it is rewarded for wins and penalised for losses.



RL can solve various complex decision making tasks that we previously couldn't have imagined a machine solving with human-like intelligence, which is why it's the most trending type of Machine Learning.^[1]

Fig 1: Reinforcement Learning ecosystem^[2]

Why RL is particularly suitable for this task is because when learning to play a game, like an Atari game, it's not sufficient to just know the rules to be competitive. The model has to reason and strategise (like a human) as it performs actions.

For example, AlphaGo which defeated the best professional human Go player, the AlphaStar Agent which beat professional StarCraft II players, and OpenAI's Dota-2-playing bot which became the first AI system to beat the world champions in an esports game, all used Deep Reinforcement Learning (DRL).^[1] So it's clear that RL is the way to go for the task at hand.

The Environment

Gym is a famous toolkit offered by OpenAI which is used for training RL agents to develop and compare RL algorithms.^[1] Gym offers a range of different environments for training an RL agent ranging from classic control tasks to Atari game environments (it provides 59 Atari environments). So we can train our RL agent to learn in these simulated environments using various RL algorithms.

Atari game selected

The Atari game we selected for this project is **Breakout**.



Fig 2: Original Atari Breakout Screen^[5]

Breakout begins with eight rows of bricks. Each two rows have a different colour. The colour order from the bottom to the top is yellow, green, orange and red. Using a single ball, the player must knock down as many bricks as possible by using the walls and/or the paddle below to ricochet the ball against the bricks and break them. If the player's paddle misses the ball's rebound, they will lose a turn.^[4] The player has three turns to try to clear two screens of bricks. Yellow bricks earn one point each, green bricks earn three points, orange bricks earn five points and the top-level red bricks score seven points each.^[4] The paddle shrinks to one-half its size after the ball has broken through the red row and hit the upper wall.^[4] Ball speed increases at specific intervals: after four hits, after twelve hits, and after making contact with the orange and red rows.^[4]

Inputs received from the OpenAI Gym environment

The inputs we receive from the OpenAI Gym are screenshots of the environment in the form of an RGB image which is an array with the shape 210x160x3.

```
print(env.observation_space.shape)
(210, 160, 3)
```

Fig 3: Screenshot from Jupyter Notebook checking observation space

Control settings for the joystick

There are four control settings:

1. Noop: which is no operation i.e. the paddle doesn't move.
2. Fire: which starts off and initiates the game.
3. Left: which moves the paddle to the left of the screen.
4. Right: which moves the paddle to the right of the screen.

```
print(env.unwrapped.get_action_meanings())

['NOOP', 'FIRE', 'RIGHT', 'LEFT']
```

Fig 4: Screenshot from Jupyter Notebook checking actions

Implementation

Data Preprocessing

As mentioned previously raw atari images are 210x160x3 by default which makes them quite large. However, we don't need that level of detail in order for our model to learn so we preprocessed the images to save us time.

We implemented an Atari wrapper preprocessing class to perform different preprocessing techniques. The preprocessing we performed was:

- Cropping irrelevant parts of the image like the top which only contains the score, the bottom which is just black, and the sides which are just grey.

```
class Atari_Wrapper(gym.Wrapper):

    def __init__(self, env, env_name, k, dsize=(84,84), use_add_done=False):
        # cropping the game frames
        super(Atari_Wrapper, self).__init__(env)
        self.dsize = dsize
        self.k = k
        self.use_add_done = use_add_done

        self.frame_cutout_h = (31,-16)
        self.frame_cutout_w = (7,-7)
```

Fig 5: Screenshot from Jupyter Notebook showing cropping functionality

- Resizing the images to a smaller shape of 64x64.
- Converting the images from coloured to grayscale as the colour doesn't affect the goal of the game i.e. breaking blocks.

```
def preprocess_observation(self, ob):
    ob = cv2.cvtColor(ob[self.frame_cutout_h[0]:self.frame_cutout_h[1],
                        self.frame_cutout_w[0]:self.frame_cutout_w[1]], cv2.COLOR_BGR2GRAY)
    ob = cv2.resize(ob, dsize=self.dsize)
    return ob
```

Fig 6: Screenshot from Jupyter Notebook showing resizing and converting to grayscale functionality

- Frame stacking by concatenating four images as one input to the CNN so the agent can determine if the ball is moving up or down. We also skip 4 every four frames.

```
def step_frame_stack(self, frames):
    num_frames = len(frames)

    if num_frames == self.k:
        self.frame_stack = np.stack(frames)
    elif num_frames > self.k:
        self.frame_stack = np.array(frames[-k:])
        # in case that the episode is finished
    else:
        self.frame_stack[0: self.k - num_frames] = self.frame_stack[num_frames:]
        self.frame_stack[self.k - num_frames:] = np.array(frames)
        # adding a new frame into the frame stack
```

Fig 7: Screenshot from Jupyter Notebook showing frame stacking functionality

With preprocessing we're trying to move from something like this:

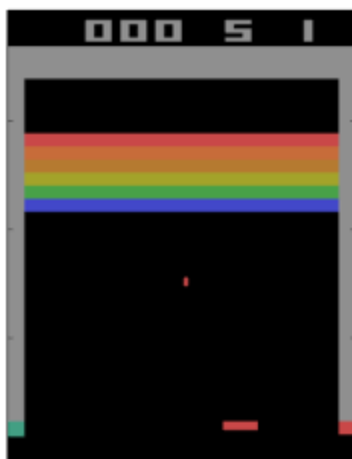


Fig 8: Breakout window before preprocessing

To something roughly like this:



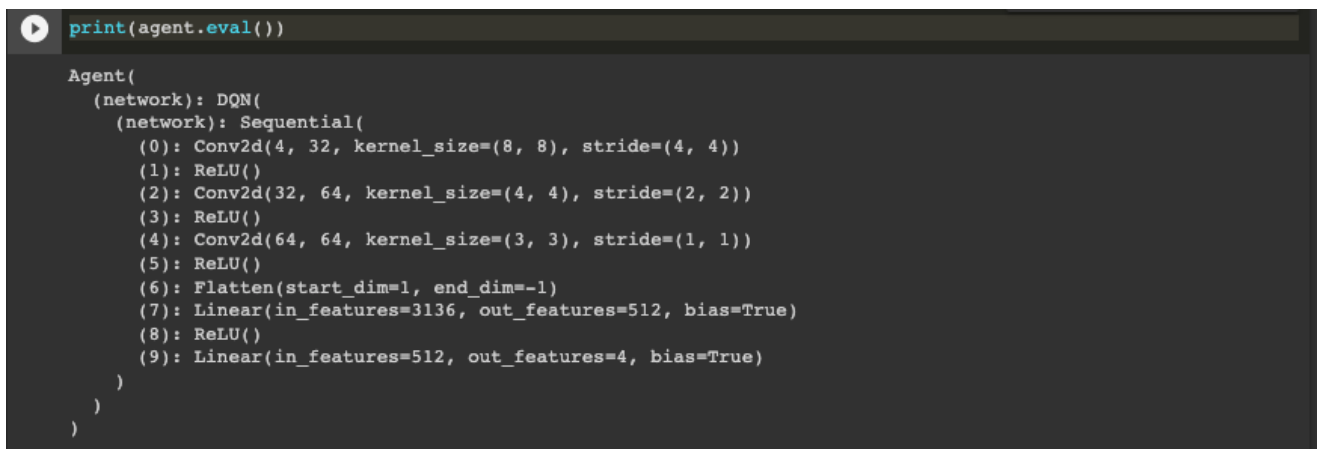
Fig 9: Breakout window after preprocessing

The Network Structure

In our implementation, we used the PyTorch library, which has predefined classes and functions that integrate with the OpenAI Gym framework. PyTorch provides functionalities for everything, starting from building to testing the model, therefore it helps to simplify the process of reinforcement learning.

The Network is constructed in a custom class named DQN. This class is a subclass of a pytorch module, called torch.nn.Module. PyTorch suggests that this module should be used for implementing any neural networks with this framework, as it provides all the necessary functions you need to reimplement for your custom model. [6]

Network Architecture



```
print(agent.eval())

Agent(
  (network): DQN(
    (network): Sequential(
      (0): Conv2d(4, 32, kernel_size=(8, 8), stride=(4, 4))
      (1): ReLU()
      (2): Conv2d(32, 64, kernel_size=(4, 4), stride=(2, 2))
      (3): ReLU()
      (4): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1))
      (5): ReLU()
      (6): Flatten(start_dim=1, end_dim=-1)
      (7): Linear(in_features=3136, out_features=512, bias=True)
      (8): ReLU()
      (9): Linear(in_features=512, out_features=4, bias=True)
    )
  )
)
```

Fig 10: model architecture

You can observe the structure of the model in the figure above, and the building blocks of the network are as follows:

- Convolution2d Layers: performs element-wise multiplication with the input data. This data then gets summed and presented as the output of the layer.
- ReLu layers: inputs a tensor and produces an output which converts all values in the tensor that are negative to zero.
- Flatten Layers: serve the purpose of flattening the output of the network into a single vector and connecting the final model.

- Linear Layers: creates a single layer feed forward network, placed at the end of the network to produce a linear output.

It is important to note that we had used these layers in the neural networks we had built in the past, however, they were previously implemented with the Keras framework, and in this case, they are built in Pytorch.nn layers.

This network allows us to map every state of the agent in the game to q-values. At every step, the agent calls this network to predict the following move.

Hyper Parameters

- Number of stacked frames: 4 is a number that's commonly used and performs better in most networks
- Learning rate: controlling how at what rate the model will change, when taking into consideration the estimated error. We have set it to $2.5e - 5$, which seems to be the commonly used rate for Breakout.
- Gamma: the discount factor for our DQN, quantifying how much importance we give for future rewards to the agent. We have set it to 0.99, common for all Atari games.
 - A larger gamma parameter means a smaller discount, therefore the agent cares more about long term reward. We pick a higher gamma parameter because we want the agent to win the game, rather than focus on performing well at one point in time.
 - A smaller gamma parameter means a bigger discount, which urges the agent to prioritise short term rewards.
- Optimizer: we are using Adam, from research we found that Adam performs well on most DQNs, as well as that we have used Adam in all of our previous projects, and it has performed well with a variety of differently structured networks.
- Loss function: Huber loss, which is a loss function that is often used for robust regression. The reason it works well with DQNs is that it is not particularly sensitive to outliers, like other loss functions are.

The Q learning update applied to the weights

The agent is able to play Breakout with the help of Deep Q-Learning, which gives it the ability to compute the most rewarding action at every state of the game. We use several different components to achieve a satisfactory result, such as the environment that the agent is able to explore and learn from, the network that consists of different layers that optimise the learning capabilities of the agent, as well as a buffer which can store the memory of the agent, for it to be able to "learn from memory" and draw information from it during the learning stage.

There are different parameters that the Q-Learning process is tuned with, such as gamma (the discount factor), epsilon (the parameter for the greedy policy), replay_size (the size of the replay buffer, i.e. the maximum number of experiences to store in memory) and minibatch_size.

The Bellman Equation

We can achieve optimal policies for our Atari games by estimating the $Q(s, a)$ function, which gives us an estimate of the discounted sum of rewards of taking action in state s , and playing optimally thereafter. Playing the action with the maximum Q -value in any given state is the same as playing optimally. Deep Q-Networks provide a way to estimate this function. ^[8]

The target network is a technique to stabilise the training of DQNs, which is essentially a copy of our original network, and is solely used for the $Q(s', a')$ values in the bellman equation. The target network is not trained and remains the same throughout the progression of deep learning, however, they do get synchronised with the parameters of the original network.

Below you can see how similarly we initialise the original and target agents, however, only the original agent gets the ability to train and learn.

```

# initializing the original agent
agent = Agent(in_channels, num_actions, start_eps).to(device)

# defining the target agent
# the target network is a copy of our original network, and we use it for the s and a value updates
target_agent = Agent(in_channels, num_actions, start_eps).to(device)
target_agent.load_state_dict(agent.state_dict())

replay = Experience_Replay(replay_memory_size)
runner = Env_Runner(env, agent)

```

Fig 11: original and target agent

Experience Replay

One of the main techniques that allows the agent to perform well with deep reinforcement learning is experience learning, which is a replay memory technique. In general, it is a way for the agent to learn from a past performance, and take into consideration the observations it has made before, when deciding on the following move. RL defines and stores an experience as follows:

- [S] - state
- [A] - action
- [R] - reward
- [S'] - new state

Experience replay is the first DL technique to store this information as a constant state buffer, instead of discarding it after its initial use. At each step, the network collects a sample size of "experiences" from the buffer, and makes a decision based on this collection. This increases the accuracy of the network, and allows for better results [7]. We implement this technique in a class and utilise it during Q-learning.

```

class Experience_Replay():
    # implementing the experience replay technique, which stores the necessary information about
    # the network's performance at every step of the game process.
    # the information stored are [STATE, ACTION, REWARD, NEXT STATE] a.k.a. sars
    # this allows for a more efficient learning and better convergence behavior of the network
    def __init__(self, capacity):
        self.capacity = capacity
        self.memory = []
        self.position = 0

    def insert(self, transitions):
        # store current transition in memory
        for i in range(len(transitions)):
            if len(self.memory) < self.capacity:
                self.memory.append(None)
            self.memory[self.position] = transitions[i]
            self.position = (self.position + 1) % self.capacity

    def get(self, batch_size):
        #return random.sample(self.memory, batch_size)
        indexes = (np.random.rand(batch_size) * (len(self.memory)-1)).astype(int)
        return [self.memory[i] for i in indexes]

    def __len__(self):
        return len(self.memory)

```

Fig 12: replay implementation

The advantages of experience learning are: breaking harmful or illogical correlations and biases and learning from individual experiences more than once.

Other significant concepts

A main issue in Reinforcement Learning is balancing exploration vs exploitation.

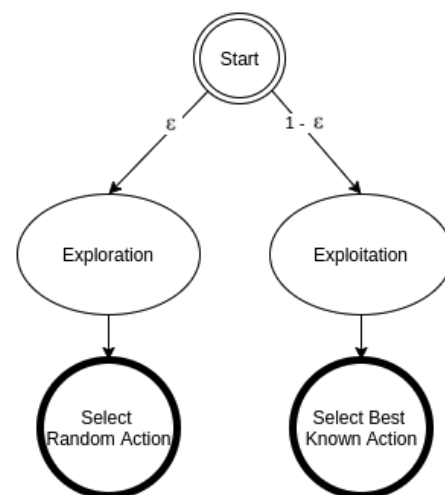
Exploration → trying new random moves (actions).

Exploitation → always performing moves (actions) you know are good.

There are two different methods of action selection:

Greedy action: ϵ (degree of exploration) = 0 → always select the action that maximises reward observed to date.

ϵ -Greedy: select the best action with probability $1-\epsilon$ (exploit), otherwise select a random action (explore)

Fig 13: methods of action selection^[9]

In our implementation we opted for ϵ -Greedy to help our model explore new states and actions.

We give ϵ both a start and a final value.

```
start_eps = 1
final_eps = 0.1
```

Fig 14: epsilon values

We then calculate a new ϵ each step to alter our degree of exploration accordingly.

```
# calculating the epsilon variable, setting the agent exploration to the new epsilon
new_epsilon = np.maximum(final_eps, start_eps - (eps_interval * num_steps / final_eps_frame))
agent.set_epsilon(new_epsilon)
```

Fig 15: recalculating epsilon



Results

Logging

We used a custom logging class to log some information during the training of our agent. It logs and documents the training steps the agent goes through and the cumulative reward it has received up until the point of training. The goal of the agent is to maximise the cumulative reward, in order to showcase that it has a good general performance.

The logging data gets stored in a csv file in the project files, therefore allows us to compare different techniques and parameters of dqn and see which one performs best.

Visualisation

```
plt.imshow(screen)
ipythondisplay.clear_output(wait=True)
ipythondisplay.display(plt.gcf())
```

Fig 16: plotting the game frame on the screen

We utilise the IPython library's Display class to visualise the game play process in our notebook. This library allows us to plot each step of the game as a frame on the screen, which gives us the ability to observe the game as it progresses. You are able to view the

animation of the game by re-running the gameplay cell. The display re-renders the grayscale frame of the game into rgb so that the game is presented realistically. The downside of this approach is that the rendering of the display is time consuming, therefore it is difficult to correctly analyse the game play by this metric alone.

Huber Loss

Huber Loss is a loss function which is a combination of the mean squared error function and the absolute value function. It has the goal of combining and balancing both of them, aiming to get "the best of both worlds". It is a good function to use when you have varied data and performs more robustly when it comes to outliers, but does not completely ignore them.

$$L_i = \begin{cases} \frac{1}{2}z^2 & \text{for } |z^{(i)}| \leq \delta \\ \delta |z| - \frac{1}{2}\delta^2 & \text{for } |z^{(i)}| > \delta \end{cases}$$

Fig 16: the formula of the huber loss function

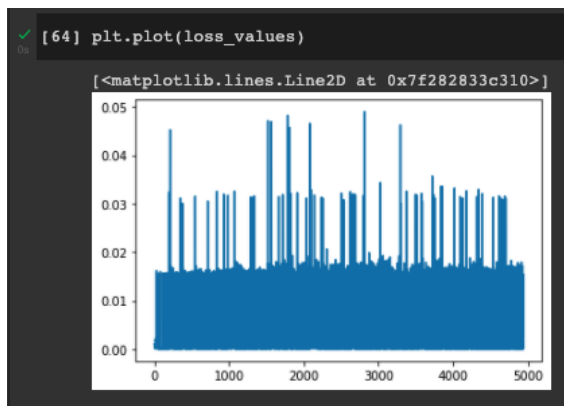


Fig 17: loss values plot

We keep track of the loss score throughout our training process and plot it at the end to observe whether the agent is able to minimise this parameter in the final stages of training.

Rewards

We track the cumulative rewards received by our agent throughout the game playing process. Our aim is that the agent is able to reach a high score of rewards, avoiding negative rewards as much as possible.

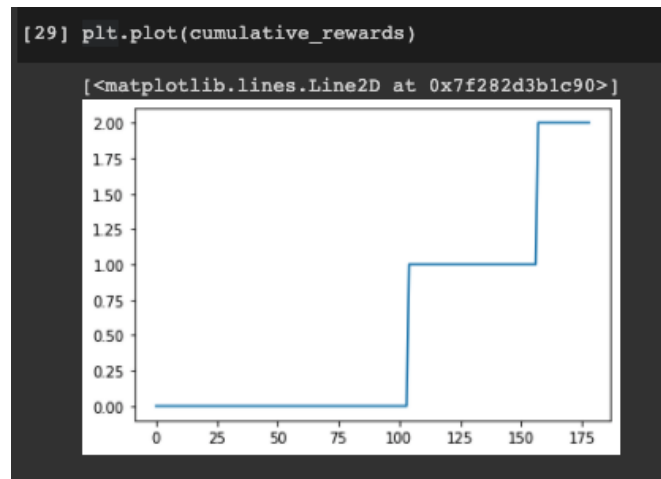


Fig 18: cumulative rewards plot

Evaluation of Results

How to evaluate the performance of the RL agent

We can judge an RL algorithm either by how good a policy it finds or by how much reward it receives while acting and learning.^[10] Which is more important depends on how the agent will be deployed.^[10] If there is sufficient time for the agent to learn safely before it is deployed, the final policy may be the most important.^[10] If the agent has to learn while being deployed, it may never get to the stage where it no longer needs to explore, and the agent needs to maximise the reward it receives while learning.^[10]

We opted for the accumulated reward approach as we believed it would be easier for us to understand and properly evaluate the performance of our agent.

Is the agent learning?

Yes, we believe it's fair to say our agent is learning because after plotting our cumulative rewards we can see that it's increasing hence it's getting better at the game and learning.

We would've liked to achieve better results however with our hardware limitations it was difficult to get the RL agent to run for very large steps that would be needed to perform better.

References

- [1] <https://towardsdatascience.com/drl-01-a-gentle-introduction-to-deep-reinforcement-learning-405b79866bf4>
- [2] <https://www.mdpi.com/2079-9292/9/9/1363/htm>
- [3] <https://www.kaggle.com/just4jorge/4-guide-to-openai-gym>
- [4] <https://www.endtoend.ai/envs/gym/atari/breakout/>
- [5] [https://en.wikipedia.org/wiki/Breakout_\(video_game\)](https://en.wikipedia.org/wiki/Breakout_(video_game))
- [6] <https://pytorch.org/docs/stable/generated/torch.nn.Module.html>
- [7] <https://proceedings.allerton.csl.illinois.edu/2018/media/files/0091.pdf>
- [8] <https://becominghuman.ai/lets-build-an-atari-ai-part-1-dqn-df57e8ff3b26>
- [9] <https://www.baeldung.com/cs/epsilon-greedy-q-learning>
- [10] <https://artint.info/2e/html/ArtInt2e.Ch12.S6.html>



Thank you. (this is a selfie of us working on this project in starbucks)