Assignment 2: DQN for Atari

CS6482 Deep Reinforcement Learning

Elsa Anza Martín (21272808)

Albert Martínez (21251266)

Kong Sheng Loi (21077363)

22nd April 2022

Lecturer: J.J Collins

UNIVERSITY OF LIMERICK

Table of Contents

DECLARATION

We declare that this assignment is our own work, except where specifically noted. We confirm that we have appropriately cited all information derived from the published and unpublished work of others.

In this assignment a regular Deep Q-Learning Network is used alongside a Dueling Network for the exploration of recent developments in DQN, both executed to the end without error.

Both implementations are taken and adapted from Dave (2019).

Section 1: Why is Reinforcement Learning the paradigm of choice for learning in the Atari Environment

1.1. Reinforcement Learning and Q Learning

In the early 2010's Reinforcement Learning yielded numerous successful approaches in the optimisation of decision-making tasks as the one in question in this project. They mainly centred around Convolutional Neural Networks, LSTM and Autoencoders (Wang et al. 2016) as well as combinations of these and other neural architectures adapted into Reinforcement Learning ideas. One of the most relevant ones were the Deep Q Networks, which are a neural approach to control tasks based on the off-policy Q-Learning control paradigm. The DQN architecture achieved significant performance in the Atari Learning Environment, which we'll explore further in this report, and became state-of-art in the field of Reinforcement Learning, thus becoming the subject of this project.

1.2. Deep Q Networks in the Atari Environment

Deep Q Networks are an approach to Reinforcement Learning applied for the first time in the domain of Atari 2600 games by Minh et al. (2015) who aimed to develop agents that could learn how to play arcade games achieving human performance. Their approach takes in high-dimensional inputs, this is, images of the game; as well as the score achieved by the agent. The DQN agent presented in the aforementioned article was a novel approach that outperformed all previous solutions in terms of score achieved when compared to human players, random play and other linear learning approaches, and it did so in a range of different games. Further details on this will be explored in section 3.

1.3. The relevance of Deep Q Networks in machine learning

The task at hand is a sequential, decision-making task. The agent perceives image frames of the game, performs an action from the set at a given time step and registers the observed reward caused by the performance of said action in said state.

In the big picture, an agent that is trained using a Deep Q Network learns the weights that allow maximising expected discount return, which underpins the pivotal concept of Exploration versus Exploitation: in the learning context, the Exploration versus Exploitation dilemma states the trade-off between long term benefit and short term benefit (greedy approaches versus epsilon-greedy)

In Reinforcement Learning this could be translated to how much the agent explores the action space versus how much the agent exploits its already learned knowledge. Being too conservative tends to be efficient but average-performing, on the other hand being innovative is risky but can bring variable improvement to the agent's performance.

Section 2: The Environment

2.1. Atari Learning environment in Reinforcement Learning

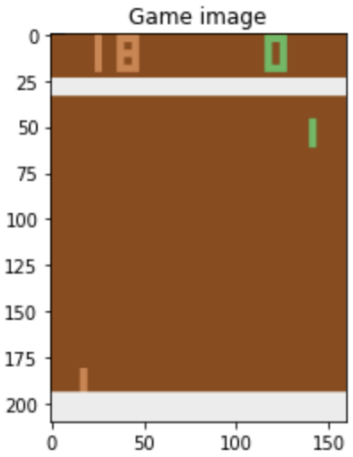
The environment used in this project, often referred to in the literature as the Arcade Learning Environment, is a joint platform that gathers a collection of hundreds of games that emulate the emblematic Atari 2600 games from the 1980s.

More specifically, our environment is the PONG game, which is deterministic, episodic and has a unique starting point. (Machado et al. 2018).

2.2. The Environment for Atari Pong

We decided to use "Pong" as the game for the implementation of our DQN algorithm. This game was released in 1972 and the main concept is an electronic ping-pong game in which the player needs to score 21 points to win against the opponent.

This was a difficult choice as the first few weeks of work were employed to implement and train a different DQN for the game "Breakout" which was finally discarded as the maximum score we were obtaining for "Pong" (21) is lower than "Breakout" (448 x 2), we saw that the training was faster in order to get good results. This first approach is briefly outlined in section 3.



2.1- SCREENSHOT OF THE GAME "PONG" EXTRACTED FROM ONE OF THE FRAMES OF OUR CODE.

2.3. OpenAI Gym environment's inputs & outputs

OpenAI Gym is the library used in this project. It is full of different Atari games (it also has other games). Thanks to this library we can test the game without having to build the environment ourselves. After building this environment (gym.make), our agent is capable of interacting with it using different functions such as env.reset, env.step or env.render.

To let the agent perform an action, we use the "step" function so that this action is taken as an input and applied to the environment. Then it will make the transition to a new state. When we call it, it returns:

- The observation of the state (observation)
- The reward that you can get from the environment after the action executed in the step function (reward)
- Whether the episode has been terminated and you may need to reset the environment or end the simulation (done)

- The additional information of the environment like the number of lives or other general information (info)

```
#create Pong env and test it a bit
env = gym.make('PongNoFrameskip-v4')
env.reset()

for i in range(3000):
    env.render("rgb_array")
    action = env.action_space.sample()
    next_state, reward, done, info = env.step(action)
    if done:
        env.reset()
env.close()

n_actions = env.action_space.n
state_dim = env.observation_space.shape

print('The original number of state features: {}'.format(state_dim))
print('Number of possible actions: {}'.format(n_actions))
```

```
Number of state features: (210, 160, 3)
Number of possible actions: 6
```

FIGURE 2.2- MAIN LOOP OF OUR CODE IN WHICH WE USE OPEN AI GYM TO PERFORM THE ACTIONS IN THE ENVIRONMENT OF "PONG"

2.4. The control settings for the joystick

In the Atari Environment we refer to actions as an action the controls specified by joystick direction and/or buttons press.

Because Atari Games don't allow learning individual values for a given pair of state-action (because possible states are too many) we will use theta to parameterise the action-value function q(s, a, theta). In the approaches to atari learning environment, a vector representing possible states when taking actions is outputted.

Action set in atari are a collection of 18 discrete actions available for the agent to choose from, which in our game pong lowers to 6 actions (Noop, fire, right, left, rightfire, leftfire) being the second and fourth the ones controlling the agent to go up, and 3 and 5 direct the agent to go lower.

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

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

FIGURE 2.3 IMAGE SHOWING THE ACTIONS AVAILABLE FOR THE AGENT WHEN PLAYING PONG.

Section 3: Implementation

The goal of our project is to implement an agent that is capable of learning how to play Atari's arcade game Pong. In this section, we'll go over our implementation of a Deep Q Network for our chosen environment, Pong, in python. Our main objective will be to build an agent that achieves human performance, which will be measured by whether the agent achieves a significant score while playing the game.

We will additionally produce an evaluation of what aspects we have found to play key roles in an implementation of this type.

3.1. Deep Q Networks

Deep Q Networks in the context of agent learning consists in a combination of Convolutional Neural Networks and Q-learning techniques put together to address two main issues: first, the instabilities caused when applying deep learning methods to a reinforcement learning problem, which include for instance how changes in Q affect target

values significantly; and also the underlying relations between observations which otherwise are ignored.

More thoroughly, a Deep Q-network optimises the funtion Q(s,a) by learning the weights theta of an approximate action-value function Q(s, a, theta), which under the hood is a maximum value of the sum of a function of the rewards r and the discount rate.

This ultimately enables an efficient calculation of the optimal q, this is, a better-performing action selection for the agent (Mnih et al. 2015).

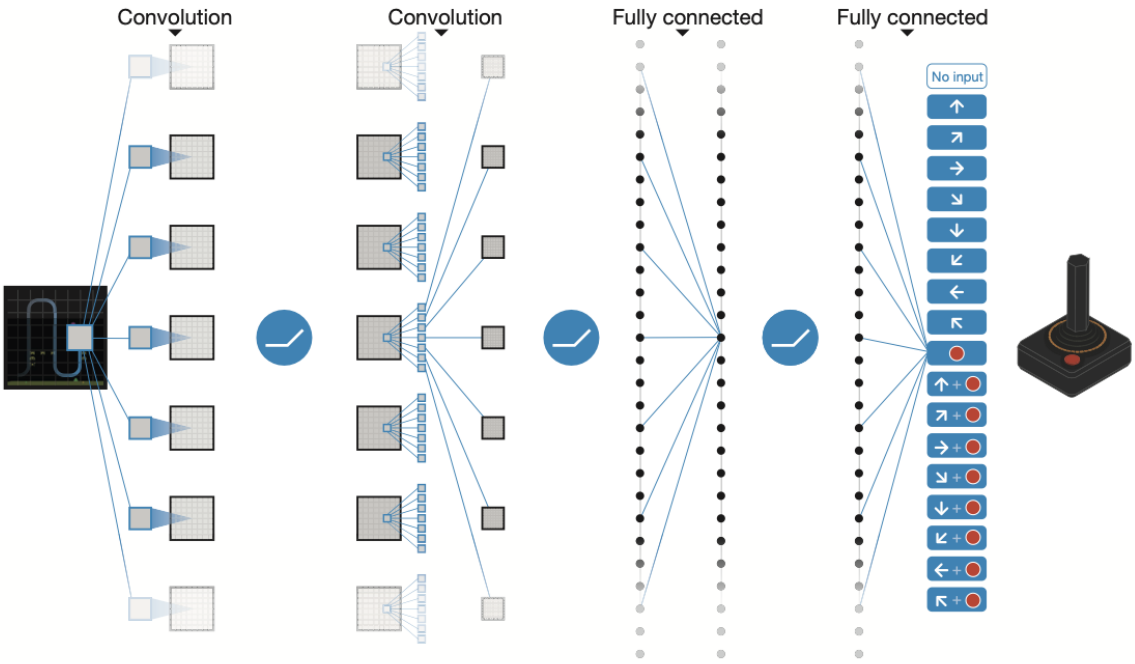


FIGURE 3.1- ARCHITECTURE AS PROPOSED BY MNIH ET AL. 2015.

In the previously mentioned paper by Mnig et al. (2015) DQNs are introduced as an algorithm that is capable of learning representations through deep neural architectures. Deep Q Networks as proposed by said author consist of hidden convolutional layers

followed by a fully connected layer. Backpropagation is used to update the weights, updating weights theta with the following formula:

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t + \alpha\Big[R_{t+1} + \gamma \max_{a\in\mathcal{A}} q(S_{t+1}, a, \boldsymbol{\theta}_t^-) - q(S_t, A_t, \boldsymbol{\theta}_t)\Big]\nabla_{\boldsymbol{\theta}_t}\ q(S_t, A_t, \boldsymbol{\theta}_t),$$

University of Limerick

FIGURE 3.2 - IMAGE SHOWING FORMULA FOR THE UPDATE OF WEIGHTS.

3.2. Research conducted for our implementation of DQNs

The main focus was looking for approaches for DQNs in the Atari Environment, at first with the aim of finding a game that was simple enough that would facilitate the task, which is substantially complex to begin with. Casual conversation with the Lecturer led us to look for approaches for the game Pong, which was recommended due to its simplicity: it requires less computation time and is ultimately quicker to deploy. Keras framework approaches were tried, but ultimately PyTorch frameworks were preferred. This is explained in detail at the end of this section.

Main code is based on an approach by Dave (2019) and minor addition of a function for plotting video was taken from Sanz Ausin (2020).

It's worth mentioning that an approach that was ultimately discarded was fully based on Sanz Ausin (2020), but wasn't working as expected due to long training times.

3.2.1. Alternatives to the final implementation

We tried first a different approach before the final code described in this third section. We implemented a DQN using Keras instead of PyTorch. Also, we tried Breakout instead of Pong as the game to work with.

The Network for the DQN agent was made up of:

- A convolutional layer of 32 filters 8x8 with 4 strides
- A convolutional layer of 64 filters 4x4 with 2 strides
- A convolutional layer of 64 filters 3x3 with 1 stride
- A convolutional layer of 1024 filters 7x7 with 1 stride
- A Flatten layer
- A Dense layer with a linear activation

```
from keras.layers import Conv2D, Dense, Flatten
class DQNAgent:
    def __init__(self, name, state_shape, n_actions, epsilon=0, reuse=False):
        """A simple DQN agent"""
        with tf.variable_scope(name, reuse=reuse):

            self.network = keras.models.Sequential()

            # Keras ignores the first dimension in the input_shape, which is the batch size.
            # So just use state_shape for the input shape
            self.network.add(Conv2D(32, (8, 8), strides=4, activation='relu',use_bias=False, input_shape=state_shape,kernel_initializer=tf.variance_scaling_initializer(scale=2)))
            self.network.add(Conv2D(64, (4, 4), strides=2, activation='relu',use_bias=False,kernel_initializer=tf.variance_scaling_initializer(scale=2)))
            self.network.add(Conv2D(64, (3, 3), strides=1, activation='relu',use_bias=False,kernel_initializer=tf.variance_scaling_initializer(scale=2)))
            self.network.add(Conv2D(1024, (7, 7), strides=1, activation='relu',use_bias=False,kernel_initializer=tf.variance_scaling_initializer(scale=2)))
            self.network.add(Flatten())
            self.network.add(Dense(n_actions, activation='linear',kernel_initializer=tf.variance_scaling_initializer(scale=2)))

            # prepare a graph for agent step
            self.state_t = tf.placeholder('float32', [None,] + list(state_shape))
            self.qvalues_t = self.get_symbolic_qvalues(self.state_t)

        self.weights = tf.get_collection(tf.GraphKeys.TRAINABLE_VARIABLES, scope=name)
        self.epsilon = epsilon
```

FIGURE 3.3– DQN AGENT DEFINITION IN THE CODE

On the main loop we were starting with an epsilon of 1 and every 500 steps/frames, we were adjusting the agent parameters. Since the algorithm was learning very slowly, we decided to save and load the weights into the agent so that we could train it at different times for several hours without losing it.

```
[22] agent.network.load_weights('/content/gdrive/My Drive/ULmsc/Atari/dqn_model_atari_weights.h5')
```

```
if i % 5000 == 0:
    agent.network.save_weights('/content/gdrive/My Drive/ULmsc/Atari/dqn_model_atari_weights.h5')
    mean_rw_history.append(evaluate(make_env(), agent, n_games=3))
```

FIGURE 3.4- EXAMPLE OF LOADING AND SAVING THE WEIGHTS IN THE CODE

The result of this code was the following:

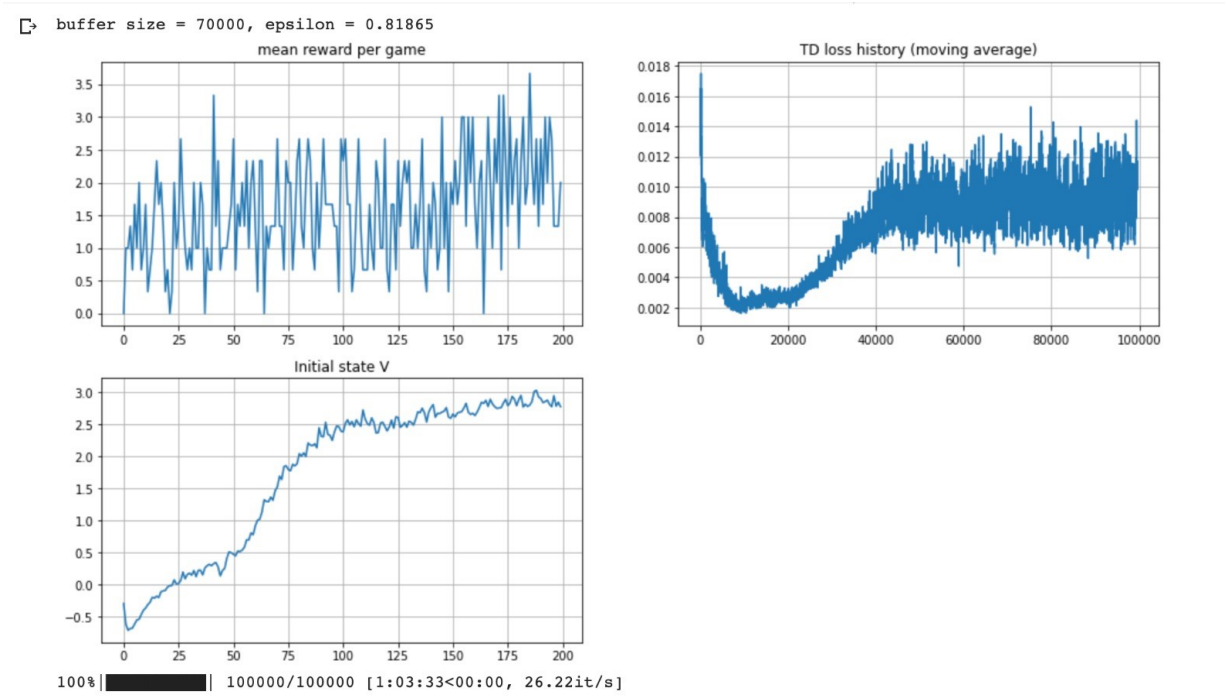


FIGURE 3.5- RESULTING GRAPHS AFTER TRAINING 100000 STEPS IN BREAKOUT DQN

As shown in the plots, the algorithm was learned after a certain number of iterations. The mean reward was increasing little by little, considering some exploitation and exploration. The temporal difference showed that, after the initial drawdown, it was learning also. And the initial state V also showed the same more smoothly . After 600000 steps, we got a score of 4 (out of 448 in one of the two screens of bricks).

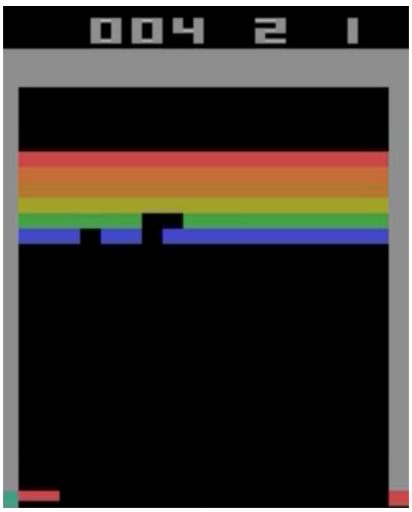


FIGURE 3.6- SCREENSHOT OF THE LAST FRAME OF A BREAKOUT GAME SHOWING THE MAXIMUM SCORE ACHIEVED ON THAT RUN

We decided to try a different game because Breakout needs a long time to get a considerably good score. Then, as explained in section 2, we decided to change to Pong.

Also, at the beginning we decided to use Keras instead of PyTorch because it offers more deployment options and the model is easier to export. But after running this first experiment, we thought it would be a better idea to use PyTorch because it is faster than Keras and has better debugging capabilities (Terra, 2020). As shown in this first approach, the amount of time that DQN needs for learning is very long, so the faster it learns, the better.

3.3. Structure of our Implementation

3.3.1. Capture and pre-processing of the data

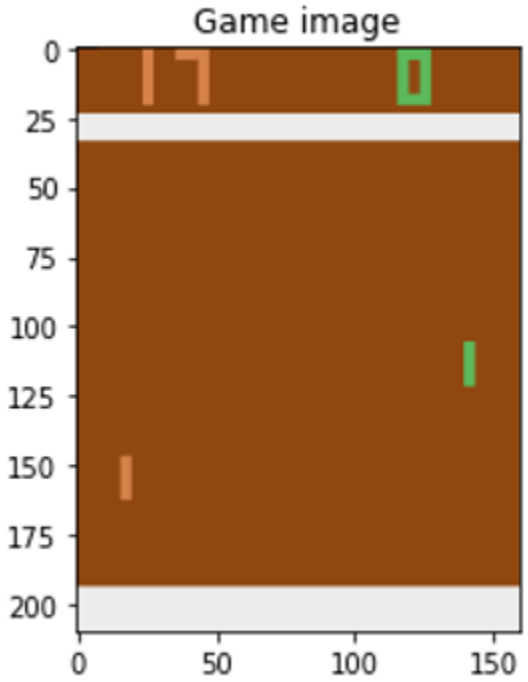


FIGURE 3.7- IMAGE SHOWING A FRAME OF ATARI PONG GAMEPLAY

The original Atari Pong game image is 210x160x3 by default, with 128 possible colours for each pixel, and there are parts of the image that we do not really need to use to train our DQN agent. Therefore, we have to pre-process the data to make the game environment more suitable for our DQN model to train the agent to play the game.

```
def make_env():
    env = gym.make("PongNoFrameskip-v4")
    env = wrap_deepmind(env, episode_life=True, clip_rewards=True, frame_stack=True)
    return env
```

FIGURE 3.8- IMAGE SHOWING MAKE FUNCTIONS FOR THE GYM ENVIRONMENT.

We define a make_env function to create an environment for our Deep Q-Learning.

The game that has been selected is PongNoFrameskip-v4. NoFrameskip is the environment consisting of one frame per step, and v4 means your issued action will determine each step (Tsou, 2021). In the function, the environment will be passed to wrap_deepmind function to do the pre-processing stage.

```
def wrap_deepmind(env, episode_life=True, clip_rewards=True, frame_stack=False):
    """Configure environment for DeepMind-style Atari.
    """
    if episode_life:
        env = EpisodicLifeEnv(env)
    if 'FIRE' in env.unwrapped.get_action_meanings():
        env = FireResetEnv(env)
    env = WarpFrame(env, width=84, height=84)
    env = MaxAndSkipEnv(env, skip=4)
    if clip_rewards:
        env = ClipRewardEnv(env)
    if frame_stack:
        env = FrameStack(env, 4)
    return env
```

FIGURE 3.9IMAGE SHOWING WRAP DEEPMIND FUNCTIONS.

EpisodicLifeEnv

The class of EpisodicLifeEnv makes our agent avoid dying as much as possible by setting done to True after each life dies and using a property self.was_real_done to mark true done after all lives are used up.

FireResetEnv

The function of this wrapper is to return an environment state that is not completed after the fire action has been selected. The FIRE button is required by some games, such as Pong, in order to launch the game. Pressing a fire button in a situation where it is needed to start the game will result in the following.

```
class WarpFrame(gym.ObservationWrapper):
    def __init__(self, env, width=84, height=84, grayscale=True):
        """Warp frames to 84x84 as done in the Nature paper and later work."""
        gym.ObservationWrapper.__init__(self, env)
        self.width = width
        self.height = height
        self.grayscale = grayscale
        shape = (1 if self.grayscale else 3, self.height, self.width)
        self.observation_space = spaces.Box(
            low=0, high=255, shape=shape, dtype=np.uint8
        )

    def observation(self, frame):
        if self.grayscale:
            frame = cv2.cvtColor(frame, cv2.COLOR_RGB2GRAY)
        size = (self.width, self.height)
        frame = frame[34: -16, :] # remove irrelevant parts
        frame = cv2.resize(frame, size, interpolation=cv2.INTER_AREA)
        if self.grayscale:
            frame = np.expand_dims(frame, -1)
        return frame.transpose((2, 0, 1))
```

FIGURE 3.10- IMAGE SHOWING WRAPFRAME FUNCTION.

WrapFrame

In this class, we wrap the original frame from 210x160 to 84x84, which is a technique to reduce the resolution of the image. At the same time, redundant areas of the images are cropped, and images will be turned from RGB colour to grayscale.

MaxAndSkipEnv

The wrapper provides a frame skipping operation that returns a tuple of environment states for each frame skipped, performs the same action in the skipped frame, overlaps its reward and takes the maximum of the pixel values of the last two frames. In Atari games, some frames only occur in odd-numbered frames, so the maximum value of the previous two frames is taken.

ClipRewardEnv

In our Pong environment, an agent gets 1 point for beating an opponent and minus 1 point for losing a game. For different games, scores are measured differently, so to facilitate uniform measurement and learning, all rewards are uniformly defined as 1 (reward > 0), 0 (reward = 0) or -1 (reward < 0).

FrameStack

The role of this wrapper is to merge k grayscale frames into one frame, providing some sequential information to the CNN "Human-level control through deep reinforcement learning" (Minh et al.,2015). The wrapper maintains a deque of size k and then sequentially replaces the oldest ob with the newest one, achieving the effect of superimposing states at different times. We will also use a LazyFrame, which is much more memory efficient.

```
#Instatntiate gym Atari-Breakout environment
env = make_env()
env.reset()
action_size = env.action_space.n
state_dim = env.observation_space.shape

print('Number of state features: {}'.format(state_dim))
print('Number of possible actions: {}'.format(action_size))
```

```
Number of state features: (4, 84, 84)
Number of possible actions: 6
```

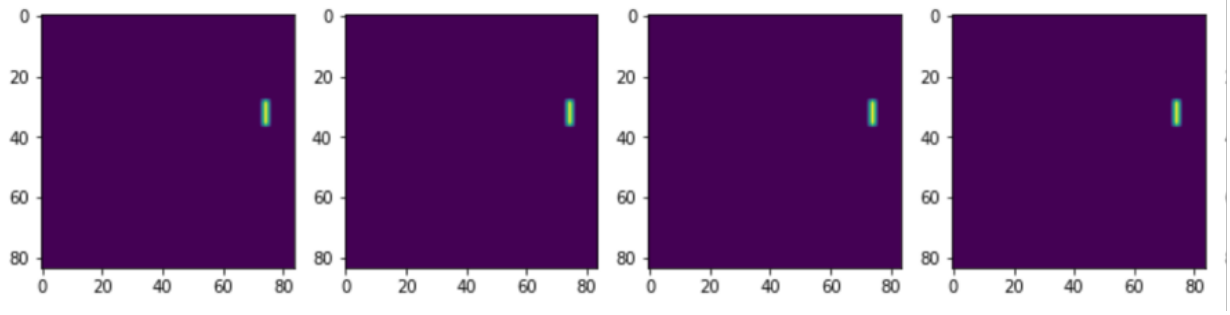


FIGURE 3.11- IMAGE SHOWING PRE-PROCESSING OF THE GAME ENVIRONMENT IMAGES.

Figure 3.11 represents the game images that have been processed, as we can see that those images have been resized to the size of (4, 84, 84) instead of (210, 160, 3), which can significantly save a lot of computational time and cost. The original scoreboard and the bottom part of the game have also been cropped as they are not necessary for our DQN agent to learn. Moreover, the number four at the beginning of the vector simply means that we have a game environment stack with the last four frames. Lastly, we also converted the image's colour from RGB format to grayscale as the colour of the pixels, in this case, is redundant, which may decrease our training speed, so we can just get rid of them (Vedpathak, 2019).

3.3.2. Network structure

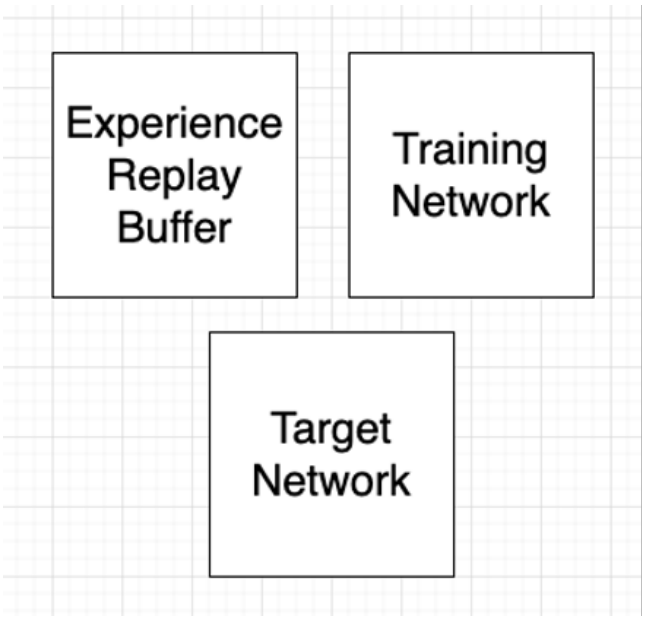


FIGURE 3.12- IMAGE SHOWING THE MAIN COMPONENTS OF THE DQN ARCHITECTURE.

Our DQN model is built on these three modules: the experience replay buffer, training network, and target network, to allow our Pong agent to play the game itself. The recordings of each game will be saved to the experience replay, and the training network will use this information to train the model. The target network can be thought of as an identical branch of the training network, and more information about it is discussed in the next section.

Experience Replay Buffer

Overfitting is a problem that often occurs in deep neural networks. They can easily be overfitting to the current episode. Once a DNN is overfitting, it is challenging to generate a variety of experiences. Lin (1992) first introduced a technique called experience replay that acts as a database-like function in a supervised learning setting to overcome this issue

(Lin, 1992). When an agent is playing a game at each episode, the experience replay buffer will simultaneously store the information such as current state, action, reward, next state, and game status. This data is vital for performing Q-learning, which helps to process the network in a mini-batch to update the neural network's weights for optimisation (Seno, 2017).

```
# create replay buffer of tuples of (state, next_state, action, reward, done)
class ReplayBuffer():
    def __init__(self, max_size=1e6):
        self.storage = []
        self.max_size = max_size
        self.ptr = 0

    def add(self, data):
        if len(self.storage) == self.max_size:
            self.storage[int(self.ptr)] = data
            self.ptr = (self.ptr + 1) % self.max_size
        else:
            self.storage.append(data)

    def sample(self, batch_size):
        ind = np.random.randint(0, len(self.storage), size=batch_size)
        x, y, u, r, d = [], [], [], [], []

        for i in ind:
            X, Y, U, R, D = self.storage[i]
            x.append(np.array(X, copy=False))
            y.append(np.array(Y, copy=False))
            u.append(np.array(U, copy=False))
            r.append(np.array(R, copy=False))
            d.append(np.array(D, copy=False))

        return np.array(x), np.array(y), np.array(u).reshape(-1,1), np.array(r).reshape(-1,1), np.array(d).reshape(-1,1)
```

FIGURE 3.13 - IMAGE SHOWING THE EXPERIENCE REPLAY BUFFER FUNCTIONS.

Figure 3.13 depicts the code of the experience replay buffer. Whenever we call the ReplayBuffer class, it will first initialise list storage to store the data and size of the storage. Usually, the size of the experience replay is considerably huge as every gameplay is significant to training our model. The size of our experience replay is 1 million, which means it can save 1000000 records of the game. The storage will keep saving and holding the records until the size is full and the oldest gameplay is removed once the storage runs out of space to save any new states.

3.3.3. Deep Q-Networks Structure overview: Training and Target Networks

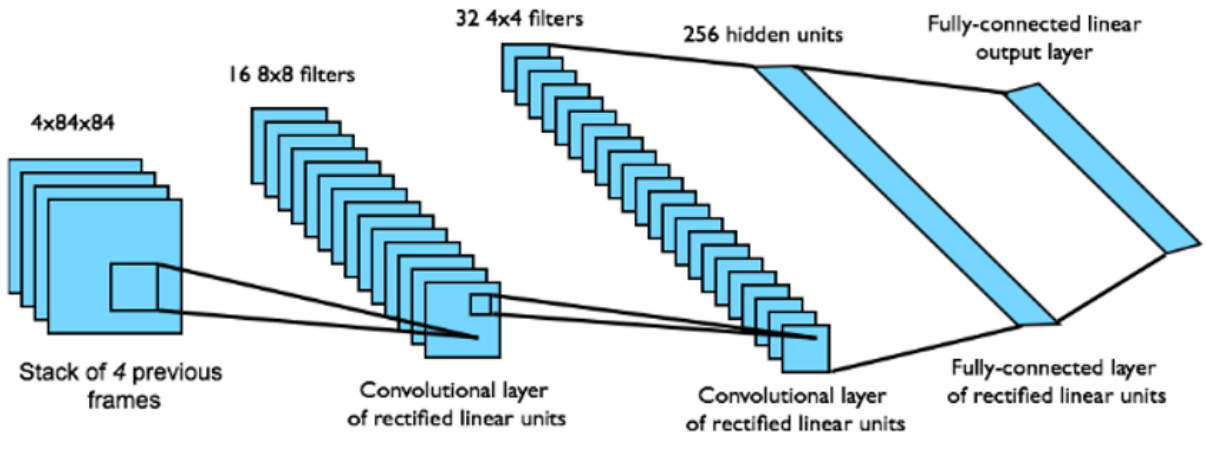


FIGURE 3.14 - IMAGE SHOWING THE STRUCTURE OF THE FIRST DQN ARCHITECTURE PROPOSED BY MNIH ET AL. (2013).

Mnih et al. (2013) proposed a deep neural network model shown in figure 3.14 for their seven Atari games in their paper "Playing Atari with Deep Reinforcement Learning" they called these convolutional networks Deep Q-Networks (DQN). Based on the paper's description, there are a total of four hidden layers in their Deep Q-Networks, namely two convolutional layers and two fully-connected layers.

Firstly, there is an input of the networks consisting of an 84 x 84 x 4 image, and the features of the image will then be passed to the deep learning network. There are two convolutional layers in the model, the first convolutional layer has 16 8 × eight filters with stride 4 with the input image, and the activation function is ReLu. The second convolutional layer includes 32 4 × 4 filters with stride 2, and the activation function is ReLu again. There is also one fully-connected layer consisting of 256 rectifier units and the final last output layer, which comprises each of the possible actions performed by the agent.

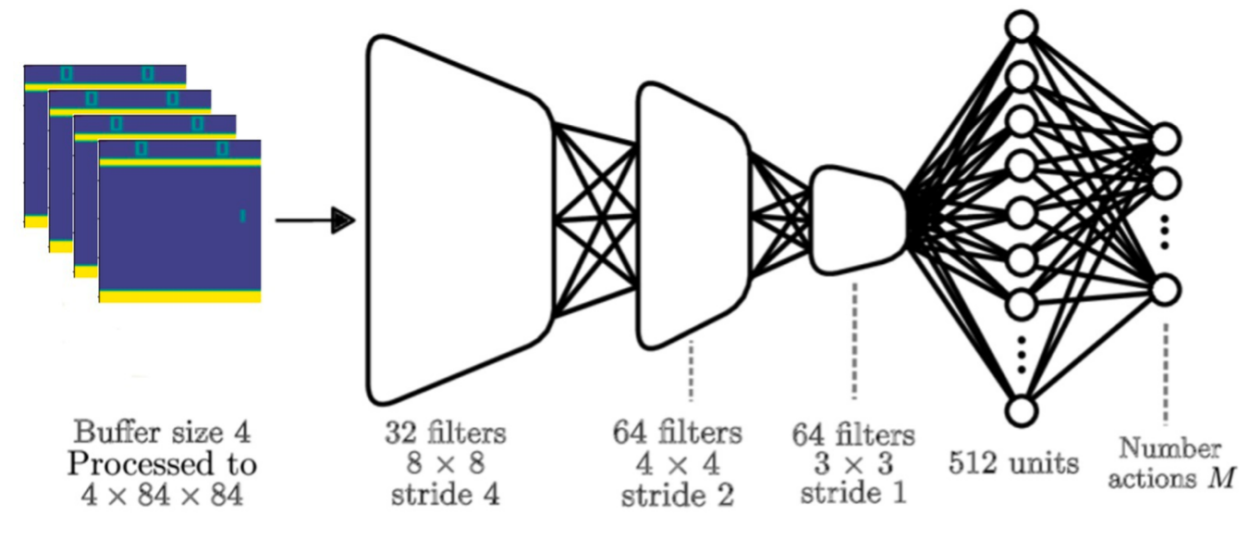


FIGURE 3.15- IMAGE SHOWING THE STRUCTURE OF A MORE RECENT DQN ARCHITECTURE PROPOSED BY MNIH ET AL. (2015).

Figure 3.15 illustrates our Pong agent's deep Q-network model structure to obtain the optimal Q value to play the game. The model's foundation is derived from the model created by Mnih et al. 2015. The improved version of DQN has one more layer and adjusts the parameters. The input layer remains the same, and the input image size is 84 x 84 x 4. We now have three convolutional layers instead of just the two proposed by Mnih et al. in 2013. The first hidden layer consists of 32 filters 8 x 8 with stride 4, the second one convolves 64 filters 4 x 4 with stride 2, and the last hidden layer convolves 64 filters 3 x 3 with stride 1. Another difference is the parameter of the fully-connected network, and our layer consists of 512 rectifier units instead of 256. The output layer remains unchanged, providing our Pong agent with the maximum probability of outputting an action. Eventually, this improved version of Deep Q-Network has grown to be more than twice the size of the previous one.

```
class DeepQNetwork(nn.Module):
    def __init__(self, action_size, hidden_size):
        super(DeepQNetwork, self).__init__()
         # The first hidden layer convolves 32 8×8 filters with stride 4
        self.conv_layer_1 = nn.Conv2d(4, 32, kernel_size=8, stride=4)

        # The second hidden layer convolves 64 4×4 filters with stride 2
        self.conv_layer_2 = nn.Conv2d(32, 64, kernel_size=4, stride=2)

        # The third hidden layer convolves 64 3x3 filters with stride 1
        self.conv_layer_3 = nn.Conv2d(64, 64, kernel_size=3, stride=1)

        # Fully connected layer made up of 512 rectifier units.
        self.dense_layer = nn.Linear(7 * 7 * 64, hidden_size)

        # The output layer is a fully-connected linear layer with a single output for each valid action."
        self.out_layer = nn.Linear(hidden_size, action_size)

    def forward(self, x):
        x = x / 255. # image data is stored as ints in 0 to 255 range. Divide to scale to 0 to 1 range
        x = F.relu(self.conv_layer_1(x))
        x = F.relu(self.conv_layer_2(x))
        x = F.relu(self.conv_layer_3(x))
        x = F.relu(self.dense_layer(x.view(x.size(0), -1)))
        return self.out_layer(x)
```

FIGURE 3.16- IMAGE SHOWING CLASSES FOR DQN.

Figure 3.16 shows the complete Deep Q-network structure, and we are using the PyTorch framework to implement our DQN model. There are three hidden convolutional layers, one dense fully-connected layer and the final output layer for the output of action probabilities. The PyTorch framework does not support flatten layers, unlike TensorFlow. Therefore, we need to implement a forward() function to turn the 3D tensors into a 1D vector. By doing that, we need to use the view() function to resize the tensors (Torres 2020).

```
class DQNAgent():
    def __init__(self, action_size, hidden_size, learning_rate ):
        self.action_size = action_size

        # Main Training Network
        self.train_net = DeepQNetwork(action_size, hidden_size).to(device)

        # Target Network
        self.target_net = DeepQNetwork(action_size, hidden_size).to(device)
        self.target_net.load_state_dict(self.train_net.state_dict())

        # Optimizer
        self.optimizer = optim.Adam(self.train_net.parameters(), lr=learning_rate)
```

FIGURE 3.17 - IMAGE SHOWING DQN AGENT CLASS.

As mentioned above, our model structure consists of a target network and a training network. In our DQNAgent class, we have created two separate networks, namely, train_net and target_net. Both have the same neural network structure, but differ in their parameters.

3.3.4. Training Network

Training network Q(s, a; θ) is utilised to control the Pong agent to play the game. It trains the neural network based on the data that has been stored in the experience replay buffer to get the predicted Q value for that particular action. After a few steps of the training, it will copy and paste its current weight to the target network.

3.3.5. Target Network

The target network was first proposed by Minh et al. (2015), a control solution addressed through deep reinforcement learning to overcome the overestimation and unstable network problem caused by their older DQN structure, which used only one network to compute predicted Q value and target Q value also known as temporal difference target. This will

cause the DQN to update the DQN itself with its own parameters, also known as bootstrapping.

$$(r_t + \max_a Q(s_{t+1}, a; \theta) - Q(s_t, a_t; \theta))^2$$

FIGURE 3.18 - IMAGE SHOWING FORMULA FOR LOSS FUNCTION.

As shown in figure 3.18, the formula is used to calculate the loss between predicted Q value $Q(st,at; \theta)$ and target Q value $rt+\max a\ Q(st+1,a; \theta)$ to make stochastic gradient descent to update the weight of the model. The learning will become very uneven as both the predicted Q value and target Q value rely on the weight of model θ, and this phenomenon will modify the direction of our temporal difference target values. This situation can be described as "chasing your own tail", as we are just training a network to calculate its output (Zai and Brown, 2020). In other words, we are moving closer to our goal, but we're also moving toward it (Ecoffet, 2017).

Therefore, the second neural network is essential to freeze the model's weight. The only use of the target network is to calculate the temporal difference target, which helps us prevent the DQN model from updating the weights itself. The parameter of the target network can be seen as Q^(st+1, a; θ-). The w- value is the weight of the target network, which is different from the training network.

The Q learning update applied to the weights

Algorithm 1: deep Q-learning with experience replay.

Initialize replay memory D to capacity N

Initialize action-value function Q with random weights θ

Initialize target action-value function $\hat{Q}$ with weights $\theta^{-} = \theta$

For episode = 1, M **do**

  Initialize sequence $s_1 = \{x_1\}$ and preprocessed sequence $\phi_1 = \phi(s_1)$

  For t = 1,T **do**

    With probability ε select a random action a_t

    otherwise select $a_t = \text{argmax}_a Q(\phi(s_t), a; \theta)$

    Execute action a_t in emulator and observe reward r_t and image x_{t+1}

    Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$

    Store transition $\left(\phi_t, a_t, r_t, \phi_{t+1}\right)$ in D

    Sample random minibatch of transitions $\left(\phi_j, a_j, r_j, \phi_{j+1}\right)$ from D

    Set $y_j = \begin{cases} r_j & \text{if episode terminates at step j+1} \\ r_j + \gamma \max_{a'} \hat{Q}\left(\phi_{j+1}, a'; \theta^{-}\right) & \text{otherwise} \end{cases}$

    Perform a gradient descent step on $\left(y_j - Q\left(\phi_j, a_j; \theta\right)\right)^2$ with respect to the network parameters θ

    Every C steps reset $\hat{Q} = Q$

  End For

End For

FIGURE 3.19 IMAGE SHOWING PSEUDOCODE FOR DQN WITH EXPERIENCE REPLAY (MNIH ET AL. 2015).

Figure 3.19 depicts the pseudocode of Mnih et al. (2015) for deep Q-learning with experience replay, which is the entire process of how DQN agents learn how to play the game. We can also refer to this pseudocode to see how Q-learning updates are applied to weights, which we will discuss line by line to make it easier and better explain the concept behind deep Q-networks. Our Pong's reinforcement learning training loop might be slightly different from the pseudocode above, but it is just a difference in order.

```
# set seed
seed = 31
env.seed(seed)
np.random.seed(seed)
torch.manual_seed(seed)
if torch.cuda.is_available():
    torch.cuda.manual_seed(seed)
    torch.cuda.manual_seed_all(seed)

# hyperparameters
timesteps = 2000000 # run env for this many time steps
hidden_size = 512   # side of hidden layer of FFNN that connects CNN to outputs
learning_rate = 0.00001 # learning rate of optimizer
batch_size = 64    # size of batch trained on
start_training_after = 10001 # start training NN after this many timesteps
discount = 0.99 # discount future states by

epsilon_start = 1.0 # epsilon greedy start value
epsilon_min = 0.01  # epsilon greedy end value
epsilon_decay_steps = timesteps * .2 # decay epsilon over this many timesteps
epsilon_step = (epsilon_start - epsilon_min)/(epsilon_decay_steps) # decrement epsilon by this amount every timestep

update_target_every = 1 # update target network every this steps
tau = 0.001

# create replay buffer
replay_size = 100000 # size of replay buffer
replay_buffer = ReplayBuffer(max_size=replay_size)

# create DQN Agent
dqn_agent = DQNAgent(action_size, hidden_size, learning_rate)
```

FIGURE 3.20 - IMAGE SHOWING HYPERPARAMETERS FOR DQN.

Before getting started to train the agent, we need to initialise some hyperparameters to aid in the development of the finest performing agents. The selection of hyperparameters can be tricky since improper hyperparameter selection can prohibit an agent from learning anything. The optimal option is determined by the algorithm and the problem. In the last line of code in Figure 3.20, we created a DQN agent, assigned action_size, which is the three actions of Pong in this example, and for a convolutional network with dense layers, the hidden size is set to 512, and the learning rate of the optimiser is 0.00001.

Step 1

Initialize replay memory D to capacity N

For this line of pseudocode, we have implemented an experience replay buffer class at the beginning of the code, with the capacity N set to one million.

Step 2

Initialize action-value function Q with random weights θ
Initialize target action-value function $\hat{Q}$ with weights $\theta^{-} = \theta$

As shown in above, in the DQNAgent class, we initialised the Q values of the training network with random weights θ and initialised the Q values of the target network with the same weights as the training network.

Step 3

For episode = 1, M **do**
Initialize sequence $s_1 = \{x_1\}$ and preprocessed sequence $\phi_1 = \phi(s_1)$

```
state = env.reset()

for ts in range(timesteps):
```

FIGURE 3.21 - IMAGE SHOWING PSEUDOCODE AND PYTHON IMPLEMENTATION FOR THE ENVIRONMENT'S INITIALISATION.

This line of pseudocode can simply be interpreted as the code below. For each episode of the game, we reset the state, also known as the screen of the game. We use state = env.reset() to reset the game once every episode ends. The reason the authors added the pre-processing sequence is that at that point, they could only use a 1:1 screen as input. However, we have already pre-processed our Pong game environment (4, 84, 84), so this is a step we can just skip.

Step 4

For $t = 1,\mathrm{T}$ **do**

With probability ε select a random action a_t

otherwise select $a_t = \mathrm{argmax}_a Q(\phi(s_t), a; \theta)$

```
# select an action from the agent's policy
action = dqn_agent.select_action(state, epsilon)
# decay epsilon
epsilon -= epsilon_step
if epsilon < epsilon_min:
    epsilon = epsilon_min
```

```
def select_action(self, s, eps):
    # select action according to epsilon-greedy method
    if np.random.rand() <= eps:
        a = env.action_space.sample()
    else:
        # greedy action is the largest Q value from the train network based on the input
        with torch.no_grad():
            input_state = torch.FloatTensor(np.array(s)).unsqueeze(0).to(device)
            a = self.train_net(input_state).max(1)[1]#.view(1, 1)#.detach().cpu().numpy()[0]
            a = int(a)
    return a
```

FIGURE 3.22 - IMAGE SHOWING PSEUDOCODE AND PYTHON IMPLEMENTATION FOR ACTION SELECTION.

In every episode, we select the actions for the agent based on the epsilon greedy strategy to balance the exploration and exploitation, unlike supervised learning in which we have already been given a set of ground truth labels for the model to understand what is right and wrong. In reinforcement learning, we do not have these labels or guides to let the agent know the result. We do not tell our agent what to do, and we simply give them rewards based on what they have done for their actions. The only mission of the agent is to learn what to do and how in order to maximise the rewards as large as possible. This, therefore, raises the question of how we decide what actions our agents should choose to take? The

solution is to allow our agent to explore and exploit and make use of the actions itself. Exploration enables the agent to examine the space randomly, it may be a space it has been to or not, which will increase its understanding of each activity now, and should lead to long-term advantage. On the contrary, exploitation allows our Pong agent to choose the greedy action to get the most valuable rewards via making use of the agent's existing action-value estimations (Shawl, 2020). Epsilon-Greedy is a simple strategy for balancing exploration and exploitation that involves periodically selecting between exploration and exploitation.

Our initial epsilon is set to 1, and it will keep decreasing over the training time. When the DQN agent wants to select an action, it will go through the select action function. A random number will be generated each time to compare the value between the random number and the epsilon number. If the epsilon is more significant than the random number, the agent will choose an arbitrary action and return it to the main loop. Otherwise, if epsilon is less than a random number, we will need to get the state of the game (4 frames) as input to the training network to get the action with the maximum reward and return it to the agent.

Step 5

Execute action a_t in emulator and observe reward r_t and image x_{t+1}

```
# enter action into the env
next_state, reward, done, info = env.step(action)
total_reward += reward
episode_length += 1
```

FIGURE 3.23 - IMAGE SHOWING PSEUDOCODE AND PYTHON IMPLEMENTATION FOR ACTION EXECUTION REWARD OBSERVATION.

In the next step, we will use the action obtained from the epsilon greedy strategy and put it into the game environment. We will get four different pieces of data as feedback from previous actions on the game environment, namely, next state, reward, done, and info. Basically, we only need the first three data to run our Deep Q-learning agent. As the name suggests, the next state is the next screen of the game. Rewards are an essential part of training here; whether an agent wins, loses, hits or misses, rewards are assigned based on the score the agent gets for the action it performed. The done attribute simply means whether the episode has been completed, either by our agent beating the opponent or by the opponent beating our agent. We then add the total reward and the length of the episode.

Step 6

Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$
Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in D

```
# add experience to replay buffer
replay_buffer.add((state, next_state, action, reward, float(done)))
```

FIGURE 3.24- IMAGE SHOWING PSEUDOCODE AND PYTHON IMPLEMENTATION FOR GAME EXPERIENCE STORAGE IN THE EXPERIENCE REPLAY BUFFER.

We use the replay buffer to store the experience of the game, including current state, next state, action, reward, and done, just in time to use the store transition (experience) in D mentioned by Mnih et al. (2015) in their paper.

Step 7

Sample random minibatch of transitions $\left(\phi_j, a_j, r_j, \phi_{j+1}\right)$ from D

Set $y_j = \begin{cases} r_j & \text{if episode terminates at step } j+1 \\ r_j + \gamma \max_{a'} \hat{Q}\left(\phi_{j+1}, a'; \theta^-\right) & \text{otherwise} \end{cases}$

Perform a gradient descent step on $\left(y_j - Q\left(\phi_j, a_j; \theta\right)\right)^2$ with respect to the network parameters θ

```
if ts > start_training_after:
    # train the agent
    stats_loss += dqn_agent.train(replay_buffer, batch_size, discount)
    td_loss_history.append(stats_loss)
```

```
def train(self, replay_buffer, batch_size, discount):
    # train the training network
    # sample a batch from the replay buffer
    x0, x1, a, r, d = replay_buffer.sample(batch_size)
    # turn batches into tensors and attack to GPU if available
    state_batch = torch.FloatTensor(x0).to(device)
    next_state_batch = torch.FloatTensor(x1).to(device)
    action_batch = torch.LongTensor(a).to(device)
    reward_batch = torch.FloatTensor(r).to(device)
    done_batch = torch.FloatTensor(1. - d).to(device)

    # get train net Q values
    train_q = self.train_net(state_batch).gather(1, action_batch)

    # get target net Q values
    with torch.no_grad():
        # Vanilla DQN: get target values from target net
        target_net_q = reward_batch + done_batch * discount * \
                torch.max( self.target_net(next_state_batch).detach(), dim=1)[0].view(batch_size, -1)

    # get loss between train q values and target q values
        # DQN implementations typically use MSE loss or Huber loss (smooth_l1_loss is similar to Huber)
    #loss_fn = nn.MSELoss()
    #loss = loss_fn(train_q, target_net_q)
    loss = F.smooth_l1_loss(train_q, target_net_q)

    # optimize the parameters with the loss
    self.optimizer.zero_grad()
    loss.backward()
    for param in self.train_net.parameters():
        param.grad.data.clamp_(-1, 1)
    self.optimizer.step()
    # we return the loss so we can monitor loss and debug the network if necessary
    return loss.detach().cpu().numpy()
```

FIGURE 3.25 - IMAGE SHOWING PSEUDOCODE AND PYTHON IMPLEMENTATION OF THE TRAINING FUNCTIONS AND WEIGHT UPDATES.

We will now go to the main training part to train our DQN agent. We need to sample a random minibatch of older transitions from the replay buffer based on the pseudocode above. Therefore, when the current episode is more remarkable than start_training_after, which is 10001, the DQN agent starts training after this many timesteps. Three data points will be passed to the training function such as replay_buffer, batch_size , and a discount rate. The replay buffer is those experiences that we obtained earlier and have been stored in a memory or list with the size of 1 million. We use all those experiences and a batch_size to create a mini-batch in order to train the network. We use a mini-batch of randomly sampled data points to feed the neural network because we do not want the data to be correlated, as each decision the agent takes affects the next state it sees. Mini-batch replay buffer will also reduce the computational cost and enhance the learning speed. Moreover, the agent can also learn the memory from the past in order to avoid catastrophic forgetting (Seno, 2017).

Train_q is the predicted Q value calculated by the training network, and it uses a state batch and an action batch from the mini-batch to get Q(s, a; θ), while target_net_q is used to calculate the temporal difference target Q^(st+1, a; θ-).

$$Q(s,a) = r(s,a) + \gamma max_a Q(s',a)$$

FIGURE 3.26 - IMAGE SHOWING FORMULA FOR TEMPORAL DIFFERENCE TARGET VALUE.

The temporal difference target value is shown in figure 3.26, where the green line is our temporal difference target value, the red line is the reward of taking action at that state from

the mini-batch replay buffer, plus the blue line is a discount rate multiply by the discounted Max Q^ value among all possible actions from next state, we calculate the Max Q^ value through the target network.

$$l_n = \begin{cases} 0.5(x_n - y_n)^2/beta, & \text{if } |x_n - y_n| < beta \\ |x_n - y_n| - 0.5 * beta, & \text{otherwise} \end{cases}$$

FIGURE 3.27- IMAGE SHOWING FORMULA FOR SMOOTH L1 LOSS FUNCTION.

To calculate the loss between the predicted Q-value and the temporal difference target value, we use a smooth L1 loss in our deep Q-learning network, which is relatively close to the Huber loss (a variant of Huber loss) mentioned in the DQN Nature paper, who use Huber loss as an improvement to DQN. The difference between Huber loss and smooth L1 loss is delta value of the smooth L1 loss is set to 1. We use smooth L1 loss (Huber loss) because Mean Square Error tends to care more about significant errors than small ones because of the squaring operation. This might lead to an unintended consequence for DQN. If a colossal error is discovered, the network will change dramatically to minimise the error. However, since the network is trying to forecast its output, a dramatic change will imply that the target value shifts dramatically. The mean absolute error may be a good solution for dealing with minor errors as well as large ones. Still, it solely disregards significant errors that should somehow be considered more important and not fine tuneable at zero (Ecoffet, 2017). Therefore, smooth L1 loss (Huber loss) would be ideal for dealing with both large and low values. It uses the L2-loss function (Least Square Errors) or Mean

Square Error to cope with low values, while the L1-loss function (Least Absolute Deviations) or Mean Absolute Error for large values (Rowe, 2018).

$$\mathbf{w} \leftarrow \mathbf{w} - \alpha \cdot \delta_t \cdot \frac{\partial\, Q(s_t, a_t; \mathbf{w})}{\partial\, \mathbf{w}}.$$

FIGURE 3.28 - IMAGE SHOWING FORMULA FOR WEIGHTS UPDATE.

Like most neural networks do backward propagation of errors to update the weights through the process of gradient descent (K, 2019). In our Deep Q-learning network, we use Adam optimisation, which is a stochastic gradient descent method to do backpropagation in order to update network weights. Adam optimiser is effective in only needing to use a small amount of memory. It is a great way to solve significant problems with a lot of data and parameters (Harry, 2020). Adam optimisation algorithm combines the benefits of both Adaptive Gradient Algorithm (AdaGrad) and Root Mean Square Propagation (RMSProp), which are also a variant of stochastic gradient descent (Brownlee, 2021). Since we have already got the loss from the predicted Q value and the temporal difference target value, the next thing we are going to do is backpropagate the loss to our target network and update the weights of it using Adam Optimisation Algorithm. To begin the backpropagation phase in PyTorch, we normally need to set the gradient to zero before updating weights and biases, as PyTorch will collect the gradient in subsequent backpropagations. Therefore, if we do not set it to zero, the old gradient and the new gradient will be mixed together, which in turn will affect our DQN learning. The backpropagation process is shown in figure 3.25,

we multiply the learning rate by the TD error and then by the parameters of the training model in order to update the weights of the training model.

Step 8

Every C steps reset $\hat{Q} = Q$

```
if ts > start_training_after:
    # train the agent
    stats_loss += dqn_agent.train(replay_buffer, batch_size, discount)
    td_loss_history.append(stats_loss)

    # update the target network every (if conditions are met in update_target_network)
    dqn_agent.update_target_network_soft(ts, update_target_every, tau) # soft target update
```

```
def update_target_network_soft(self, num_iter, update_every, update_tau=0.001):
    # soft target network update: update target network with mixture of train and target
    if num_iter % update_every == 0:
        for target_var, var in zip(self.target_net.parameters(), self.train_net.parameters()):
            target_var.data.copy_((1.-update_tau) * target_var.data + (update_tau) * var.data)
```

FIGURE 3.29 - IMAGE SHOWING PSEUDOCODE AND PYTHON IMPLEMENTATION FOR UPDATING THE TARGET NETWORK.

The last step of the entire Deep Q-Learning is to update the weights of the target network after certain criteria are met. When the current step is greater than 10001, which we have set before, our DQN agent will start copying training network weights to the target network. Consequently, the target network is able to predict better Q values with the enhanced weights, and now the weights of both the training network and target network are identical again (Doshi, 2020).

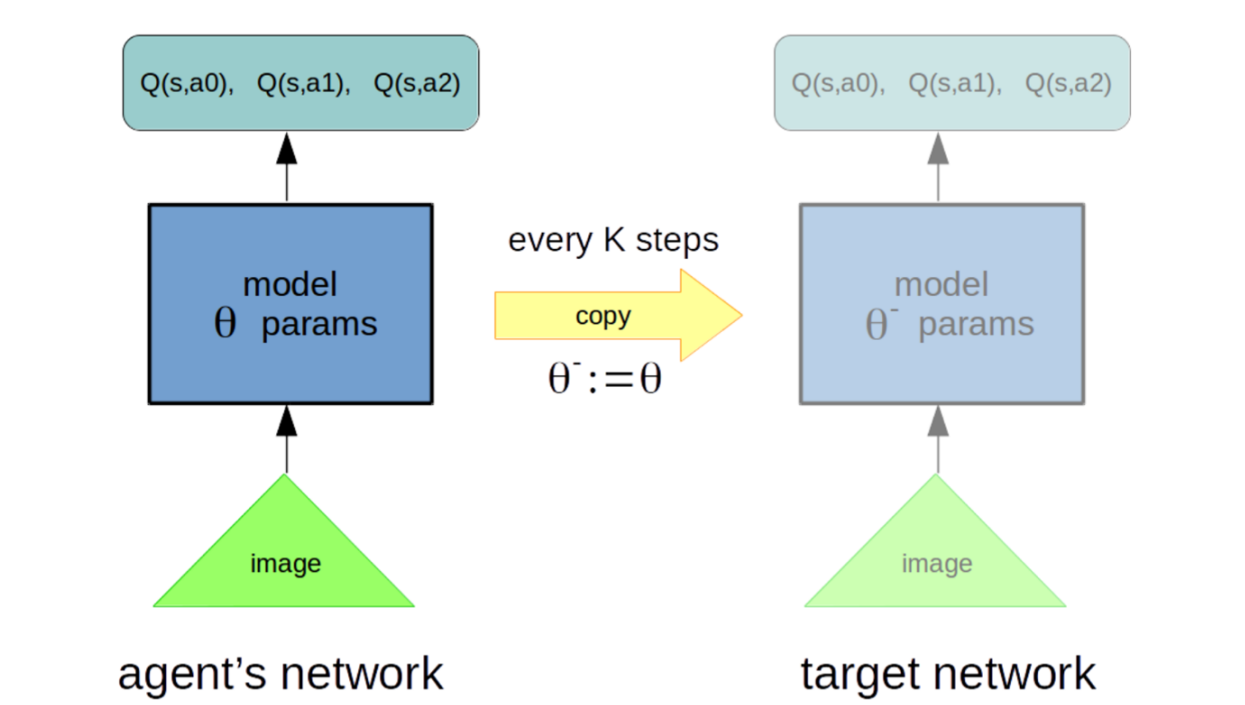


FIGURE 3.30 - DIAGRAM SHOWING THE TRANSFER OF WEIGHTS FROM THE TRAINING NETWORK TO THE TARGET NETWORK EVERY K STEPS.

```
if done:
    screen = env.render(mode='rgb_array')
    state = env.reset()
    stats_rewards_list.append((episode, total_reward, episode_length))
    episode += 1
    total_reward = 0
    episode_length = 0

    if ts > start_training_after and episode % stats_every == 0:
        print('Episode: {}'.format(episode),
              '/{}'.format(timesteps),
            'Timestep: {}'.format(ts),
            'Total reward: {:.1f}'.format(np.mean(stats_rewards_list[-stats_every:],axis=0)[1]),
            'Episode length: {:.1f}'.format(np.mean(stats_rewards_list[-stats_every:],axis=0)[2]),
            'Epsilon: {:.2f}'.format(epsilon),
            'Loss: {:.4f}'.format(stats_loss))
        stats_loss = 0.

        # Plotting Screenshot
        plt.title("Gameplay image")
        plt.imshow(screen)
        plt.show()

    # stopping condition for training if agent reaches the amount of reward
    if len(stats_rewards_list) > stats_every and np.mean(stats_rewards_list[-stats_every:],axis=0)[1] > 20:
        print("Stopping at episode {} with average rewards of {} in last {} episodes".
            format(episode, np.mean(stats_rewards_list[-stats_every:],axis=0)[1], stats_every))

        # Plotting Screenshot
        plt.title("Gameplay image")
        plt.imshow(screen)
        plt.show()
        break
else:
    state = next_state
```

FIGURE 3.31- IMAGE SHOWING CODE FOR THE MAIN LOOP.

The training process will be iterative, from how the agent gets its actions based on the ε-greedy strategy at the beginning to updating the weights of the target network to be precisely the same as the training network. If the game is over, whether our agent has beaten the opponent or the opponent has beaten our agent, we print all the essential data and plot the final game screen figure 3.31. Once our Pong agent has won more than 20 rewards, the reinforcement learning training process is terminated, meaning our agent can play the game and beat its opponents entirely on its own.

Fast Deep Q-learning can be achieved by using a smaller initial epsilon greedy value or decreasing the epsilon rate at a more rapid pace. However, a small initial value of epsilon

may cause learning to stay sub-optimal due to a lack of exploration (Causevic, 2019). Our current initial epsilon greedy value is 1, the end value is 0.01, and the epsilon decay step is 400000; the value will start decreasing once the training begins. Our strategy here is to use the initial epsilon value minus the end value and divide it by the epsilon decay step, which means our epsilon value here decreases slowly over time. In other words, our DQN agent will have more time to explore the space before exploiting it.

Section 4: Results

Our goal in the project, which was implementing an agent which achieved human-like performance in playing Atari Pong, was met with using the Deep Q Networks.

In this section we will explore the reasons behind why our DQN successfully manages to learn an optimal long-term strategy that will lead it to win the game, by learning policies from almost scratch.

4.1. Results obtained by our implementation of DQNs approach

The first aspect being measured in our graphs are rewards, which measure the score achieved by the agent while playing, which is directly related to its performance.

Secondly, we measure episode length, which accounts for the amount of time the agents spend playing until the end of the game, by being either defeated or by winning the game. We'll see how this aspect can show how the agent gathers experience from playing.

Lastly, the third graph shows the evolution of the Temporal Difference Loss function against the episodes played by the agent, which depicts the difference between the DQN network's predicted Q value Q(s, a; θ) and the target Q value Q^(st+1, a; θ-).

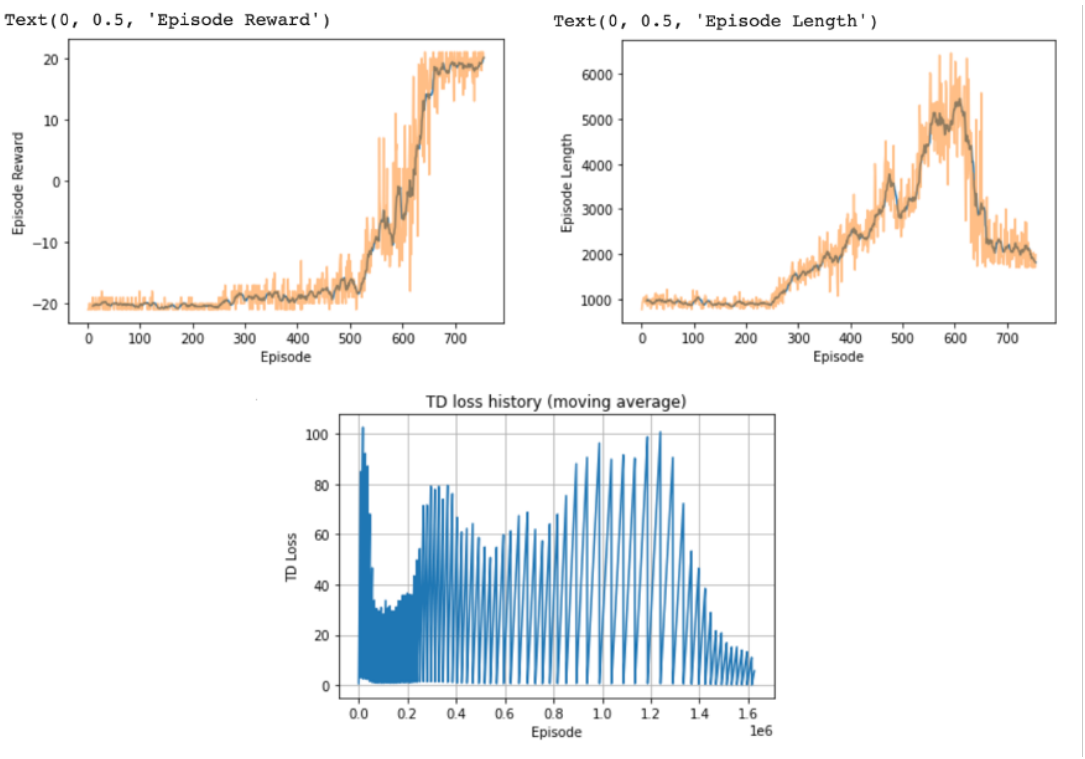


FIGURE 4.1- FROM LEFT TO RIGHT, AND TOP TO BOTTOM, THREE GRAPHS A B AND C DISPLAYING VARIATION OF REWARDS, EPISODE LENGTH AND TEMPORAL DIFFERENT LOSS, RESPECTIVELY, AGAINST EPISODE VARIATION.

4.2. Discussion of results

The key behind the improved convergence, is, in short, use of past experience as training data, which improves generalisation because it increases variability in the data used by the networks.

This can be seen through the lenses of the balance between exploration, which enables the agent to examine the search space more broadly, and exploitation, which optimises learning by making use of relevant past knowledge, is achieved using the epsilon greedy strategy, ultimately leading to our agent's long-term advantage.

With no prior knowledge exploration is key, shown in graph a.

Balance between exploration and exploitation is most clearly seen in graph b. Episode length measures how long the game was, which could loosely be interpreted as how much the agent is learning, this is, very little at first, when the tendency is that episodes are extremely short, meaning the agent doesn't last much before losing to its opponent. When there is so little prior knowledge the agent has to make the most of exploring the search area. The elbow in the data around episode 250 is a turning point for the agent, which incrementally gathers knowledge, marked by games being longer around episodes 400-600, when the agent matches its opponent in performance. This means the agent is relying in exploration to gather knowledge, which is reflected in graph a in how little rewards the agent obtains at first, because it's incapable of exploiting non-existing previous knowledge, and the slope is very little, and then when the game episode lengths reach a maximum around episode 550, a very steep ascent is observed simultaneously in graph a, meaning its optimising rewards by exploiting the knowledge now gained.

Episode length in graph b finally drops to a minimum around episode 650. The reason why length goes down is the agent has learnt a good enough policy that makes it take the minimum possible time to win at every game, and simultaneously, in graph b, rewards are maximised.

The evidence that our agent is learning can be better appreciated in graph c, showing the TD Loss function, which is unevenly varying for the first two thirds of the episodes, just to start to evenly decrease, showing convergence, around the last quarter of the training, which corresponds to stages in which the agent is starting to gather sufficient significant experience to win and to achieve its firsts maximum rewards (shown in graph a). On the other hand, exploration is appreciated in the first part of the TD loss graph, where the

variation between target and predicted values is very uneven due to the fact that the agent has no prior knowledge and is randomly exploring the search space, meaning random actions are being selected.

4.2.1. Conclusions

We conclude that CNNs and Reinforcement Learning implemented in a combined approach are highly linked to improved learning capabilities. The reason behind this achievement is likely the ability to store past knowledge (Experience Replay component, which is a way of storing experience and using said experience as training data), which could be linked to the neuroscientific evidence that in the human brain, recalling past experiences is key for complex learning processes such as becoming a good player at a certain game.

Section 5: Exploration of recent developments in Deep Q Networks

Dueling Architectures in DQNs are an alternative to existing Deep Reinforcement Learning approaches like LSTM or AEs. It is based on the optimisation of two estimates for states and actions separately, in contrast to plain DQNs which optimises the function Q(s, a) solely.

Other approaches are deep visuomotor policies (Levine et al., 2015), massively parallel frameworks (Nair et al. 2015) and expert move prediction Maddison et al., 2015).

5.1. Dueling Deep Q Networks

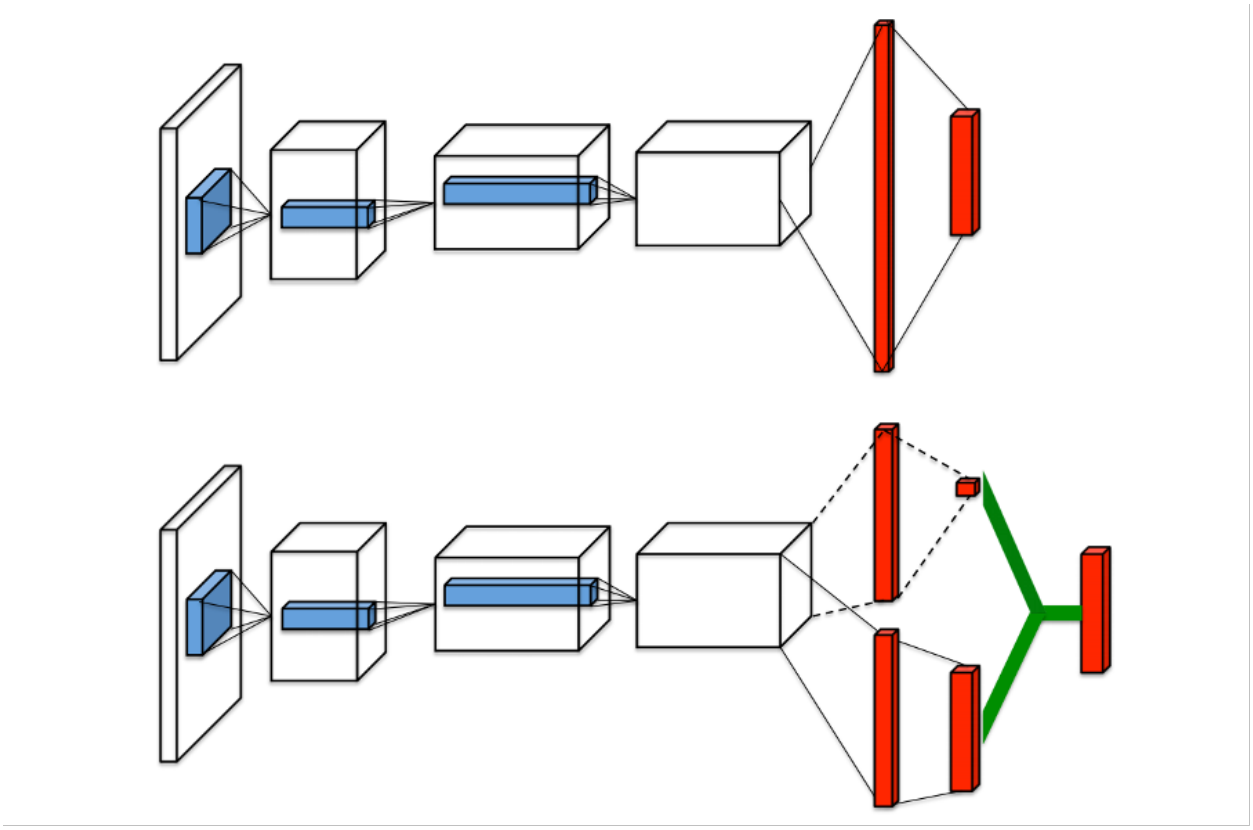


Figure 5.1- Top image shows a diagram of regular DQN model structure, bottom image shows the equivalent diagram for Dueling DQN model structure.

The concept of the Dueling Network first appeared in 2016 and was proposed by Wang et al. (2016) in their paper "Dueling Network Architectures for Deep Reinforcement Learning". In a sense, this structure is motivated by the fact that there are games, such as Atari Enduro, where it is not necessary to know the impact of every action at every moment (Yoon, 2019).

Its relevance lies in the novelty of merging two separate branches of the Q-network on top of a single Convolutional feature learning Framework. These branches separately optimise values for state and actions representations, which means they update advantage functions A(s,a) and state values functions V(s) independently and then merge them to produce a

single Q function value Q(s,a). The advantage function A measures whether an action will have a positive or negative impact in a given state and has been proven to converge faster than regular simple Q-learning approaches (Wang et al. 2016).

Dueling DQNs are therefore a novel extension to Deep Q Networks, and some of their contributions are the ability to learn how to tell valuable states separately from actions and improved generalisation.

5.2. Research conducted for our implementation of Duelling DQNs

Figure 5.1 depicts the standard Deep Q-Learning model structure and Dueling Network model structure. Basically, both models use the same way to calculate the optimal action-value function Q(s, a) for the agent to play the game in the area of reinforcement learning. However, The Dueling Network optimises the final output of the dense layers in Deep Q-Learning by splitting them into two distinct fully-connected layers, the optimal state value function and the optimal advantages function. The optimal state value function is used to evaluate whether the current state is good or bad and whether the agent is close to winning or losing the game, the formula is shown in figure 5.2. On the other hand, the optimal advantages function can be seen as an advantage that is a measure of the relative advantage of an action compared to other actions in a given state. The formula is shown in figure 5.3, we can see that the optimal advantages function is actually the value of the optimal value-action function Q(s, a) minus the optimal state value function; the better the action a value is, the more advantageous it is.

$$V^{\star}(s) = \max_{\pi} V_{\pi}(s).$$

FIGURE 5.2 - OPTIMAL STATE VALUE FUNCTION FORMULA

$$A^{\star}(s,a) = Q^{\star}(s,a) - V^{\star}(s).$$

FIGURE 5.3- OPTIMAL ADVANTAGES FUNCTION FORMULA

In fact, we can change the above formula to obtain the optimal action-value function Q(s, a), which is the final output we get from a normal Deep Q-Network, in order to enable our Pong agent to pass its own games.

$$Q(s,a) = V(s) + A(s,a)$$

FIGURE 5.4- OPTIMAL ACTION-VALUE FUNCTION FORMULA

By converting the mathematical formula, we obtain the latest formula in order to get the optimal action-value function, just like a regular Deep Q-Learning network. As you can see, we simply just sum up the optimal state value function and the optimal advantages function to get Q(s, a). However, Wang et al. (2016) point out in their paper "Dueling Network Architectures for Deep Reinforcement Learning" that this equation has a problem of non-identifiability.

$$Q(s,a;\theta,\alpha,\beta) = V(s;\theta,\beta) + (A(s,a;\theta,\alpha) - \max_{a' \in |\mathcal{A}|} A(s,a;\theta,\alpha))$$

FIGURE 5.5 - FORMULA FOR DUELING NETWORK WITH MAXIMUM VALUE OF A

Therefore, Wang et al. (2016) suggest that we still need to add a minus max value of the optimal advantages function at the end of the formula in order to overcome this problem. The method allows us to deal with the non-identifiability problem by making the largest value of Q(s, a) equal to the V(s). In other words, creating the maximum value of A(s, a) in the optimal advantages function to be zero, and the rest values become negative (Sanz Ausin, 2020). From that, we can calculate all the advantages straight away, thus solving the problem figure 5.5.

$$Q(s,a;\theta,\alpha,\beta) = V(s;\theta,\beta) + (A(s,a;\theta,\alpha) - \frac{1}{|\mathcal{A}|}\sum_{a'} A(s,a;\theta,\alpha))$$

FIGURE 5.6 - FORMULA FOR DUELING NETWORK WITH MEAN A

Lastly, Wang et al. (2016) mention that we can also replace the max function with the mean function. Hence, this is our final formula for Dueling Network to calculate the optimal action-value function Q(s, a) figure 5.6.

5.3. Structure of our implementation of Duelling DQNs

```
class DuelingDeepQNetwork(nn.Module):
    def __init__(self, action_size, hidden_size):
        super(DeepQNetwork, self).__init__()
        # The first hidden layer convolves 32 8×8 filters with stride 4
        self.conv_layer_1 = nn.Conv2d(4, 32, kernel_size=8, stride=4)

        # The second hidden layer convolves 64 4×4 filters with stride 2
        self.conv_layer_2 = nn.Conv2d(32, 64, kernel_size=4, stride=2)

        # The third hidden layer convolves 64 3x3 filters with stride 1
        self.conv_layer_3 = nn.Conv2d(64, 64, kernel_size=3, stride=1)

        # Fully connected layer made up of 512 rectifier units.
        self.fc_layer = nn.Linear(7 * 7 * 64, hidden_size)

        # V(s) value of the state
        self.dueling_value = nn.Linear(hidden_size, 1)
        # Q(s,a) Q values of the state-action combination
        self.dueling_action = nn.Linear(hidden_size, action_size)

    def forward(self, x):
        x = F.relu(self.conv_layer_1(x))
        x = F.relu(self.conv_layer_2(x))
        x = F.relu(self.conv_layer_3(x))
        x = F.relu(self.fc_layer(x.view(x.size(0), -1)))
        # get advantage by subtracting dueling action mean from dueling action
            # then add estimated state value
        x = self.dueling_action(x) - self.dueling_action(x).mean(dim=1, keepdim=True) + self.dueling_value(x)
        return x
```

FIGURE 5.7- CODE FOR THE CLASS OF DUELING DQNS

When comparing figure 3.16 and figure 5.7, you can find that the structure of Deep Q-Network and Dueling Network are identical. The Dueling Network also has three convolutional layers and one fully-connected layer. The only difference is removing the last output layer with 512 hidden sizes and action_size as output. There are two more layers in their place, namely optimal advantages function and optimal state-value function, to be added to our Dueling

the equation that we have already discussed above figure 5.6. As you can see, the last line in the forward() function indicates that we are using the advantage function to minus the mean value of the advantage function and then adding the optimal state value function to calculate the optimal action-value Q(s, a).

Fundamentally, Dueling Network is exactly the same as a standard Deep Q-Learning network. They have the same goal, which is to derive the optimal action-value function Q(s, a) for the agent. Consequently, we do not need to make any other significant changes besides modifying the neural network model structure to make it more advanced. We are also using the same way to calculate the temporal difference loss and backpropagate the loss to the network in order to update the weights and biases.

5.4. Results obtained in our implementation of Duelling DQNs

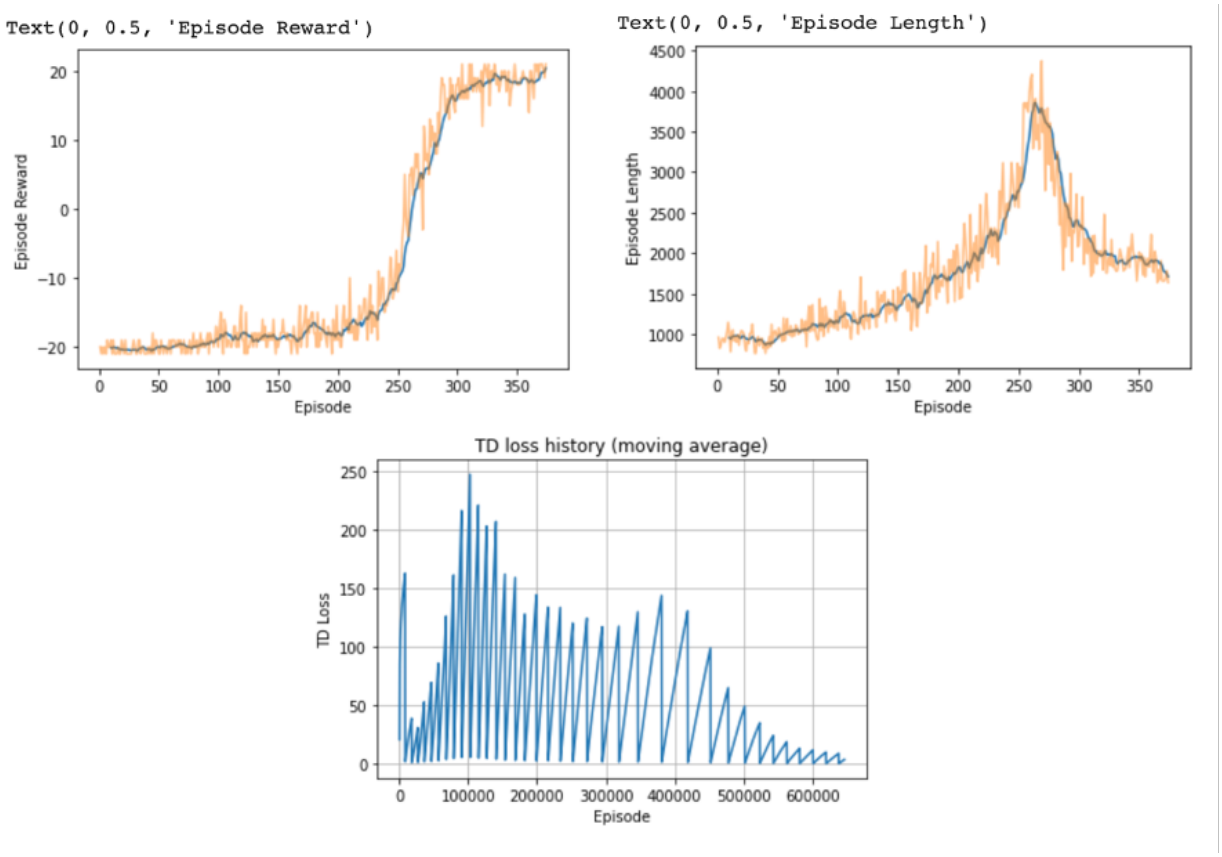


FIGURE 5.8- FROM LEFT TO RIGHT, AND TOP TO BOTTOM, THREE GRAPHS A B AND C DISPLAYING VARIATION OF REWARDS, EPISODE LENGTH AND TEMPORAL DIFFERENT LOSS, RESPECTIVELY, AGAINST EPISODE VARIATION IN THE CASE OF DUELING DQNS.

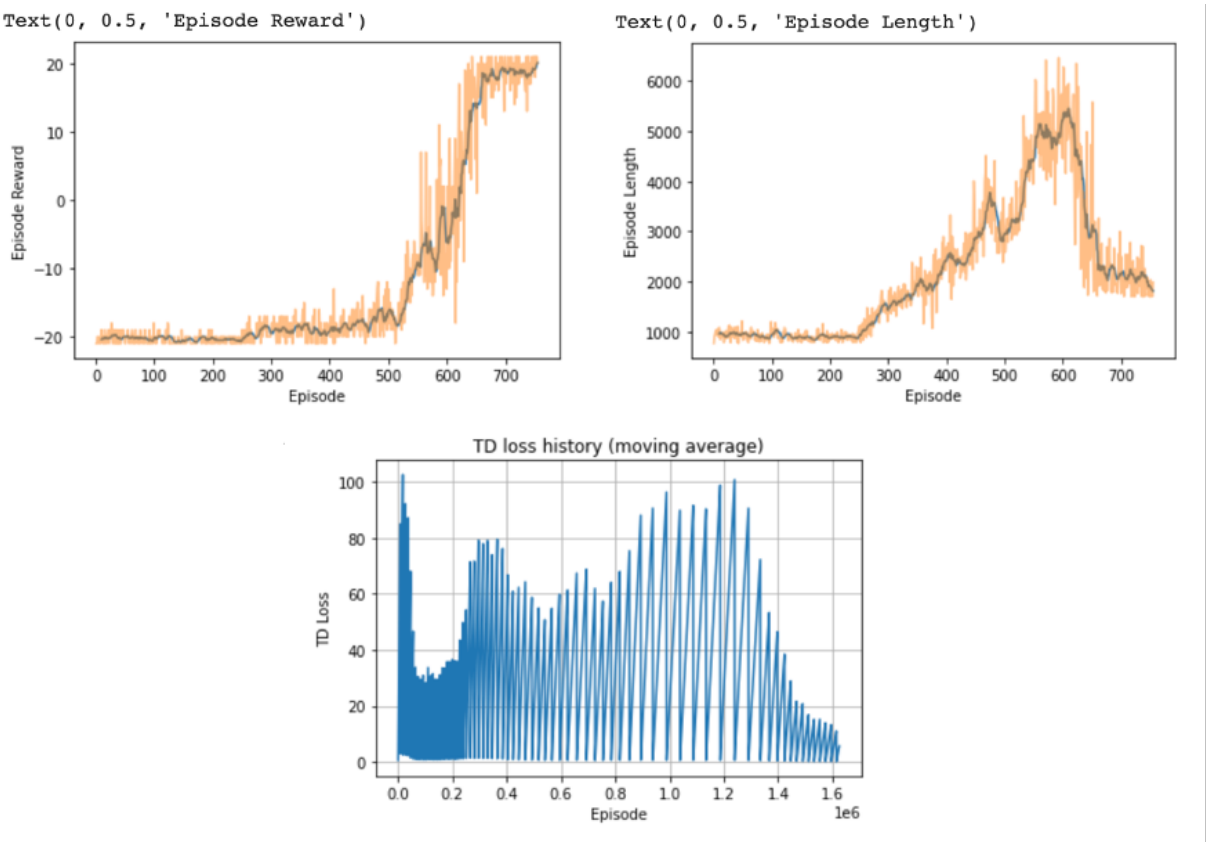


FIGURE 5.9 - FROM LEFT TO RIGHT, AND TOP TO BOTTOM, THREE GRAPHS A B AND C DISPLAYING VARIATION OF REWARDS, EPISODE LENGTH AND TEMPORAL DIFFERENT LOSS, RESPECTIVELY, AGAINST EPISODE VARIATION IN THE CASE OF REGULAR DQNS.

5.5. Discussion of results

The parameters used in both of our experiments are the same, so there will be no uneven events. We want to compare which agent in the two models can learn to play the game faster on its own and beat its opponent. Figure 5.8 represents the experimental results for the Dueling Network in the Atari Pong, while Figure 5.9 shows the experimental results for the Normal Deep Q-learning Network in Atari Pong.

Firstly, in figure 5.9 graph A, we can see from this episode reward graph that Dueling Network had a clear upward trend from the beginning, and at almost 240 episodes, its

rewards started to rise at a very fast rate. In the Pong environment, the rewards increase when we or the agent scores and decrease by one point when our opponent scores. Our agent starts the game with a negative reward; that is because it does not know how to play the game; plus, we set our epsilon greedy at 1, which means our agent was still in its exploration stage most of the time. It looked for possible space randomly and performed random actions to explore the game. The game was terminated at around 350 episodes, much faster than Normal Deep Q-Learning Network, which took about 700 episodes to complete the game. As we mentioned earlier, once the agent hits a specific reward, for example, the total rewards are greater than 20, then the game will stop itself, and the training is complete. Standard Deep Q-Learning Network took nearly twice as long to converge compared to Dueling Network. In other words, Dueling Network has a better performance as it uses less time to terminate the training.

In figure 5.9 graph B, the episode length of Dueling Network is also much less than the Deep Q-learning network. The peak of the episode was approximately 4000, while DQN plateaued out around 5500 and then started plummeting. Both episode lengths of Dueling Network and DQN stopped at nearly 2000.

Interestingly, when we look at figure 5.9 graph A, the temporal difference loss history graph, we can see that the Dueling Network had a higher starting point of around 160, while the DQN started at only slightly above 100. Despite this, Dueling Network has taken less time to catch up with DQN, which is still stuck in the high TD loss phase for a long time. From the graphs, we can see that there is actually a clear synchronised pattern of rising and falling TD losses for both. Both start at a high point, then lower to a certain point before rising again, then fall again, then stay at a steady level before rising again. The

difference is that this time the TD loss of Dueling Network does not grow as much as the DQN, and we can clearly see from the graphs of the DQN that its TD loss has increased much more than the Dueling Network. The Dueling Network is more effective and faster in reducing its TD loss than the DQN. This means that Dueling Network allows our agents to learn how to play the game and beat their opponents in less time, and we can observe that Dueling Network has significantly less TD loss than DQN at the end of training. The lower TD loss proves that our predicted Q values are getting closer to the temporal difference target values. From this result, we can demonstrate and infer that our agent is learning and our network parameters and weights are getting better.

Section 6: References

Brownlee, J., (2021) Gentle Introduction to the Adam Optimization Algorithm for Deep Learning. [online] Machine Learning Mastery. Available at: <https://machinelearningmastery.com/adam-optimization-algorithm-for-deep-learning/>.

Causevic, S., (2019) Deep Reinforcement Learning: Build a Deep Q-network(DQN) with TensorFlow 2 and Gym to Play CartPole. [online] Medium. Available at: <https://towardsdatascience.com/deep-reinforcement-learning-build-a-deep-q-network-dqn-to-play-cartpole-with-tensorflow-2-and-gym-8e105744b998>.

Dave, M., (2019) Hands-on-Reinforcement-Learning-with-PyTorch/3.7 Dueling DQN with Pong.ipynb at master · PacktPublishing/Hands-on-Reinforcement-Learning-with-PyTorch. [online] GitHub. Available at: <https://github.com/PacktPublishing/Hands-on-Reinforcement-Learning-with-PyTorch/blob/master/Section%203/3.7%20Dueling%20DQN%20with%20Pong.ipynb>.

Doshi, K., (2020) Reinforcement Learning Explained Visually (Part 5): Deep Q Networks, step-by-step. [online] Medium. Available at: <https://towardsdatascience.com/reinforcement-learning-explained-visually-part-5-deep-q-networks-step-by-step-5a5317197f4b>.

Ecoffet, A., (2017) Beat Atari with Deep Reinforcement Learning! (Part 2: DQN improvements). [online] Medium. Available at: <https://becominghuman.ai/beat-atari-with-deep-reinforcement-learning-part-2-dqn-improvements-d3563f665a2c>.

Harry, P., (2020) Intuition of Adam Optimizer - GeeksforGeeks. [online] GeeksforGeeks. Available at: <https://www.geeksforgeeks.org/intuition-of-adam-optimizer/>.

K, H., 2019. Backpropagation Step by Step. [online] Hmkcode.com. Available at: <https://hmkcode.com/ai/backpropagation-step-by-step/#:~:text=Backpropagation%2C%20short%20for%20%E2%80%9Cbackward%20propagation,proceeds%20backwards%20through%20the%20network.

Lin, L.J., 1992. Reinforcement learning for robots using neural networks. Carnegie Mellon University.

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. Journal of Artificial Intelligence Research, 61, pp.523-562.

Mnih, V., Kavukcuoglu, K., Silver, D., Graves, A., Antonoglou, I., Wierstra, D. and Riedmiller, M., 2013. Playing atari with deep reinforcement learning. arXiv preprint

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.

Rowe, W., 2018. Mean Square Error & R2 Score Clearly Explained. [online] BMC Blogs. Available at: <https://www.bmc.com/blogs/mean-squared-error-r2-and-variance-in-regression-analysis/#:~:text=There%20is%20no%20correct%20value,one%20prediction%20model%20over%20another.>.

Sanz Ausin, M., 2020. Introduction to Reinforcement Learning. Part 4. Double DQN and Dueling DQN. [online] Medium. Available at: <https://markelsanz14.medium.com/introduction-to-reinforcement-learning-part-4-double-dqn-and-dueling-dqn-b349c9a61ea1>.

Seno, T., 2017. Welcome to Deep Reinforcement Learning Part 1 : DQN. [online] Medium. Available at: <https://towardsdatascience.com/welcome-to-deep-reinforcement-learning-part-1-dqn-c3cab4d41b6b>.

Shawl, S., 2020. Epsilon-Greedy Algorithm in Reinforcement Learning - GeeksforGeeks. [online] GeeksforGeeks. Available at: <https://www.geeksforgeeks.org/epsilon-greedy-algorithm-in-reinforcement-learning/>.

Terra, J. (2020). Keras vs Tensorflow vs Pytorch [Updated] | Deep Learning Frameworks | Simplilearn. [online] Simplilearn.com. Available at: https://www.simplilearn.com/keras-vs-tensorflow-vs-pytorch-article#:~:text=PyTorch%20vs%20Keras

Torres, J. (2020) Deep Q-Network (DQN)-I. [online] Medium. Available at: https://towardsdatascience.com/deep-q-network-dqn-i-bce08bdf2af

Tsou, C., 2021. Reinforcement Learning: Deep Q-Learning with Atari games. [online] Medium. Available at: <https://medium.com/nerd-for-tech/reinforcement-learning-deep-q-learning-with-atari-games-63f5242440b1>.

Vedpathak, O., 2019. Playing Pong from pixels using Reinforcement Learning. [online] Medium. Available at: <https://towardsdatascience.com/intro-to-reinforcement-learning-pong-92a94aa0f84d>.

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.

Yoon, C., 2019. Dueling Deep Q Networks. [online] Medium. Available at: <https://towardsdatascience.com/dueling-deep-q-networks-81ffab672751>.

Zai, A. and Brown, B., 2020. Deep Reinforcement Learning in Action. Manning Publications Company.