CS6482 Deep Reinforcement Learning

Assignment 3: Sem2 AY23/24 - Reinforcement Learning Agent for Atari Breakout Game

**Open AI Gym Atari Breakout: BreakoutNoFrameSkip-v4 (Dueling DQN Implementation)**

SUBMITTED BY:

1. SAJIN MOHAMED PALLIKKATHODI ERATHALI (23037601)

2. AKSHATA BHEEMASAMUDRA MALLIKARJUNAPPA (23164204)

**Table of Contents**

2.3 Control Settings for the **Joystick**……………………………………………4

1. Implementation

3.3.4 Speeding up the learning

1. Results

4.1 Plots

4.2 Evaluation of performance

[**1. Objectives 2**](#_mv95wkqv4b5v)

[1.1 Why Reinforcement Learning is the machine learning paradigm of choice for this task? 3](#_t8od8ce297og)

[**2. The Gym Environment 3**](#_7d14a1kwbkg7)

[2.1 Atari Game Selected 3](#_prij6m6380cb)

[2.2 Inputs received from OpenAI Gym Environment 3](#_q4cciquqw2)

[2.3 Control Settings for the Joystick 4](#_w04cee13uv7z)

[**3. Implementation 4**](#_7bl01xmi5yww)

[3.1 Capture and Preprocessing of data 4](#_6iftfagcl5jo)

[3.2 Network Structure 6](#_ayahhstg8atr)

[3.3 The Q learning update applied to the weights 7](#_oe90x0d3osj0)

[3.4 Prioritized Experience Replay to speed up learning 9](#_12s7gyyfbdjw)

[3.4.1 Proposed work plan 10](#_hdmbj1slhapj)

[**Exploration of recent developments in DQN 12**](#_ohnx5u1co46j)

[Double DQN 12](#_6lod4eb7s48)

[Why we need Double DQN 12](#_qewdolapsw93)

[Dueling DQN 12](#_hpfcasx94a3i)

[**Results 15**](#_349wnampilkw)

[**Bibliography 15**](#_i16fby2ihbb6)

# 1. Objectives

To implement a Reinforcement Learning (RL) agent using a Deep Q Network (DQN) applied to the Breakout game of Atari in the OpenAI Gym environment.

### **1.1 Why Reinforcement Learning is the machine learning paradigm of choice for this task?**

(1 marks, 0.5 pages)  
Reinforcement Learning (RL) is particularly effective for games like Atari Breakout due to its ability to handle decision-making under uncertainty, manage delayed rewards through temporal credit assignment, and adaptively balance between exploring new strategies and exploiting known ones. In Atari Breakout, the RL agent learns from direct interactions with the game, improving over time as it navigates varying game scenarios and conditions. This adaptability, combined with the direct feedback loop where the agent's actions are continuously adjusted based on outcomes, makes RL a natural fit for the dynamic and strategy-driven environment of video games, enabling it to effectively maximize long-term gains and enhance game performance.

As per ([Mnih, et al., 2013](https://arxiv.org/abs/1312.5602)), Reinforcement learning (RL) is particularly well-suited for Atari games due to its ability to learn control policies directly from high-dimensional sensory inputs, such as video frames from the games themselves, without relying on pre-engineered features or understanding of the game's internal state. The use of convolutional neural networks trained with a variant of the Q-learning algorithm allows the model to successfully interpret raw pixel data to estimate future rewards and make decisions accordingly. A key challenge that makes RL a fitting choice for Atari games is the requirement to operate effectively from complex, noisy, and high-dimensional sensory inputs where the rewards are delayed and the data samples are highly correlated. These characteristics make traditional machine learning techniques less effective since they often require hand-labeled training data or well-defined feature sets.

# 2. The Gym Environment

### **2.1 Atari Game Selected**

The game we selected is Breakout with as it was reported to have a high performance with respect to human game testers, as illustrated by [Mnih et al 2015](#edw89b1ih6fl). Note that the normalized performance of DQN, expressed as a percentage, is calculated as: 100 × (DQN score − random play score)/(human score − random play score). The percentage value for Breakout is 1327% which indicates that, for Breakout, DQN significantly outperforms human game testers.

The environment chosen for the assignment is ‘BreakoutNoFrameskip-v4’

### **2.2 Inputs received from OpenAI Gym Environment**

The target of the game is to breakthrough a wall and let the ball break the wall on brick at a time. As there are many layers for the wall, the reward obtained would be proportional to the color of the brick.The agent has 5 lives to play the game before being done, ie. the agent can let the ball pass the paddle after breaking the bricks only 5 times before the game finishes. The game allows for 4 actions, 0 - no operation, 1 - fire the ball, 2 - move the paddle right and 3 - move the paddle left. The environments returns RGB image to the player

Since we chose a version without frame skip, the agent can take one action per time step. However, in order to mitigate this we do a frame stacking every 4 time steps such that the agent only takes an action on the last frame of the game (i.e., on the 4th frame)

### **2.3 Control Settings for the Joystick**

a.

the Atari game selected,

b.

the inputs received from the OpenAI Gym environment, and

c.

the control settings for the joystick.

# **3. Implementation**

### **3.1 Capture and Preprocessing of data**

The input data we get from the environment ‘BreakoutNoFrameskip-v4’ is an image of size 210 \* 160 of 3 different colors (RGB). This is converted to Greyscale image and the size is reduced to fit a square of 84 x 84 pixel in order to extract the part of the frame that plays the game and so that it can be passed to a convolutional neural network. The image pixels are also normalized to fall between 0 and 1 by division with 255(since pixel values range from 0 to 255). This aids computation to converge faster during training. Figure $#4 illustrates the preprocessing done in the code.

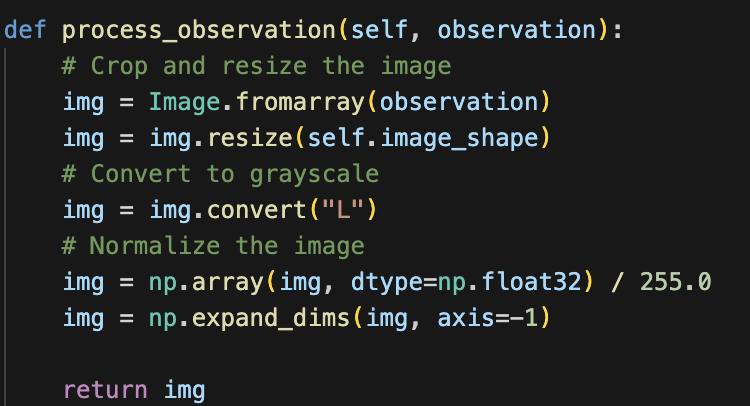


Figure $#4. Process observations

As part of the preprocessing, since environment.step for each action returns a single image and we need multiple images for training to be faster, we stack 4 images with the above mentioned dimensions ie (84 x 84) and pass this to the network as the next state resulting in the network accepting an input of shape 84, 84, 4.

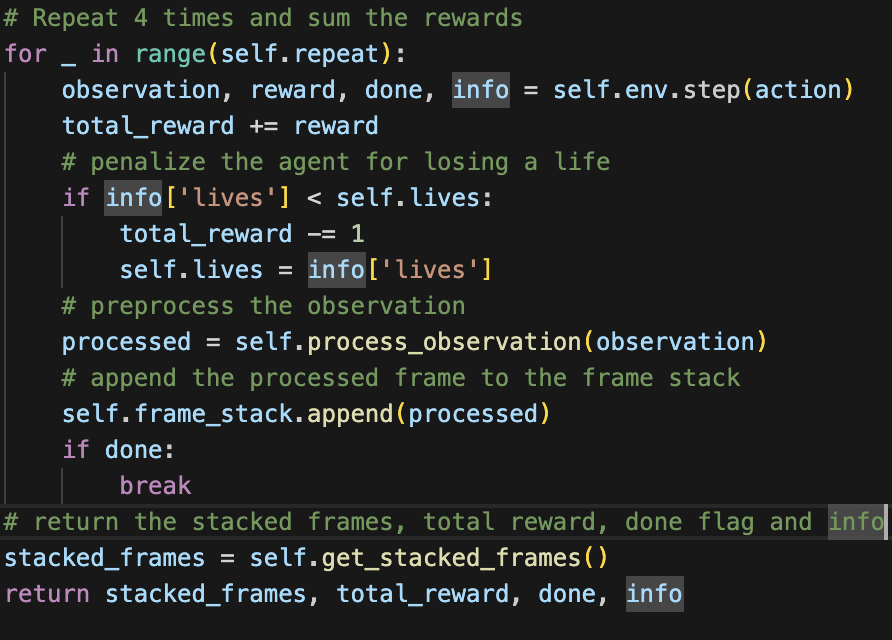


Figure $#3. The overridden step method used in play\_one\_step

The environment step function is wrapped with a custom defined step function where we add in more behaviour to the data obtained from the step function of the environment. The Figure $#3 shows how we override the step method of the environment to penalize the game for each of the 5 lives lost and to stack the 4 frames. When all of the 5 lives are lost the done state of the game would be true and exit the loop created. In each loop we iterate through each of the frames and stack them together and return that as the next state to meet the requirement of changing the input to 84 x 84 x 4. The preprocessing is done with the reset method call as well to get the states in the above mentioned shape.

NOTE: The reward obtained is decreased by -1 each time a life is lost, as a result in the ipynb notebook attached you can find that the maximum negative value associated with a reward would be -5 as there are only 5 lives associated with the agent. Once it loses all 5 lives the game reaches its done state and terminates the episode.

### **3.2 Network Structure**

The CNN structure was customized based on the work done by [Mnih, et al., 2013](https://arxiv.org/abs/1312.5602)$# where we have 210 X 160 size image which has undergone a preprocessing as mentioned in $# and forming the resultant image of size 84 x 84 which denotes the playing area. This along with the last 4 frames stacked together, provides the dynamics of the game, which the agent can take as input to learn faster. Hence the input to the CNN is the stack of last 4 frames from the game, of the shape 84 x 84 x 4. On top of the work done by [Mnih, et al., 2013](https://arxiv.org/abs/1312.5602) , we added another layer and the architecture is illustrated in figure $#5

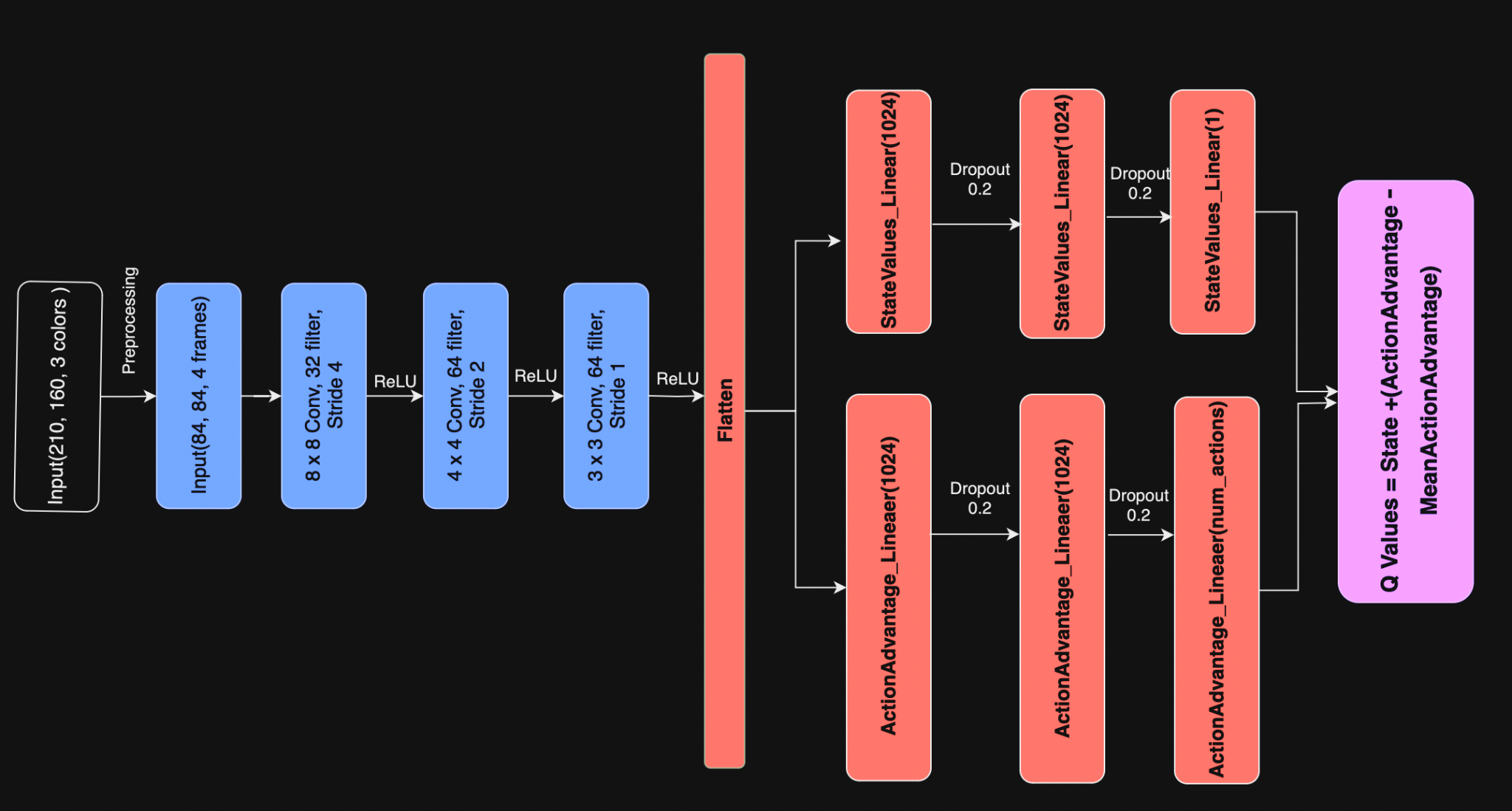


Figure $#5 CNN Architecture Diagram for Duelling DQN

The first hidden layer convolves 32 filters of size 8 × 8 with stride 4 with the input image and applies a Rectified Linear Unit (ReLU) activation function. The second hidden layer convolves 64 filters of size 4 × 4 with stride 2, followed by another ReLU activation. The third hidden layer convolves 64 filters of size 3 × 3 with stride 1, again followed by a ReLU activation.

After these convolutional layers, the output is flattened into a one-dimensional array.

The network then splits into two separate branches, both fully connected to the flattened output.

The first branch, the state value branch, consists of a fully connected layer with 1024 ReLU units, followed by a dropout layer with dropout rate of 20%. This is again connected to a fully connected layer with 1024 ReLU units which is followed by another dropout layer with a dropout rate of 20%. The output is send to a fully connected layer with a single linear unit. This branch outputs the value of the state V(s).

The second branch, the action advantage branch, also consists of a fully connected layer with 1024 ReLU units, followed by a dropout layer with a dropout rate as 0.2, which is connected to a fully connected layer with 1024 ReLU units. The output of the dropout layer is fed to a fully connected layer with a number of linear units equal to the number of possible actions. This branch outputs the advantage of each action A(s, a).

The final output of the network is produced by combining state value and action advantage by uisng the below formula

***Q value = State\_value + (Action\_advantage - Mean\_Action\_advantage)***

The (Action\_advantage - Mean\_Action\_advantage) is calculating the advantage of each action relative to the average advantage of all actions. This helps to stabilize the learning process by centering the action advantages around zero. Hence, the Q value gives an estimate of the total expected reward for each action, taking into account both the value of the state and the relative advantage of the action.

This architecture implements a Dueling Network, where the state value and action advantage are computed separately and combined at the end. Here we have 4 actions as mentioned above

### **3.3 The Q learning update applied to the weights**

The Q-learning update applied here is that of Fixed Dueling DQN. While it is possible to derive the results using Deep Q-Network(DQN) without a target network. ie. using only neural network and do two passes over the network, one to output the predicted Q value and the second one to output the target Q value, with our implementation, we wanted to avoid updating the Q Network weights on each step to prevent the constant changes in the targeted Q-value, akin to chasing a moving target with each update.

Hence we introduce a second network called target\_model1 that does not update its weights on every step or episode, thereby obtaining a stable result. We update the target network every 10000 frames, which ensures stable learning.

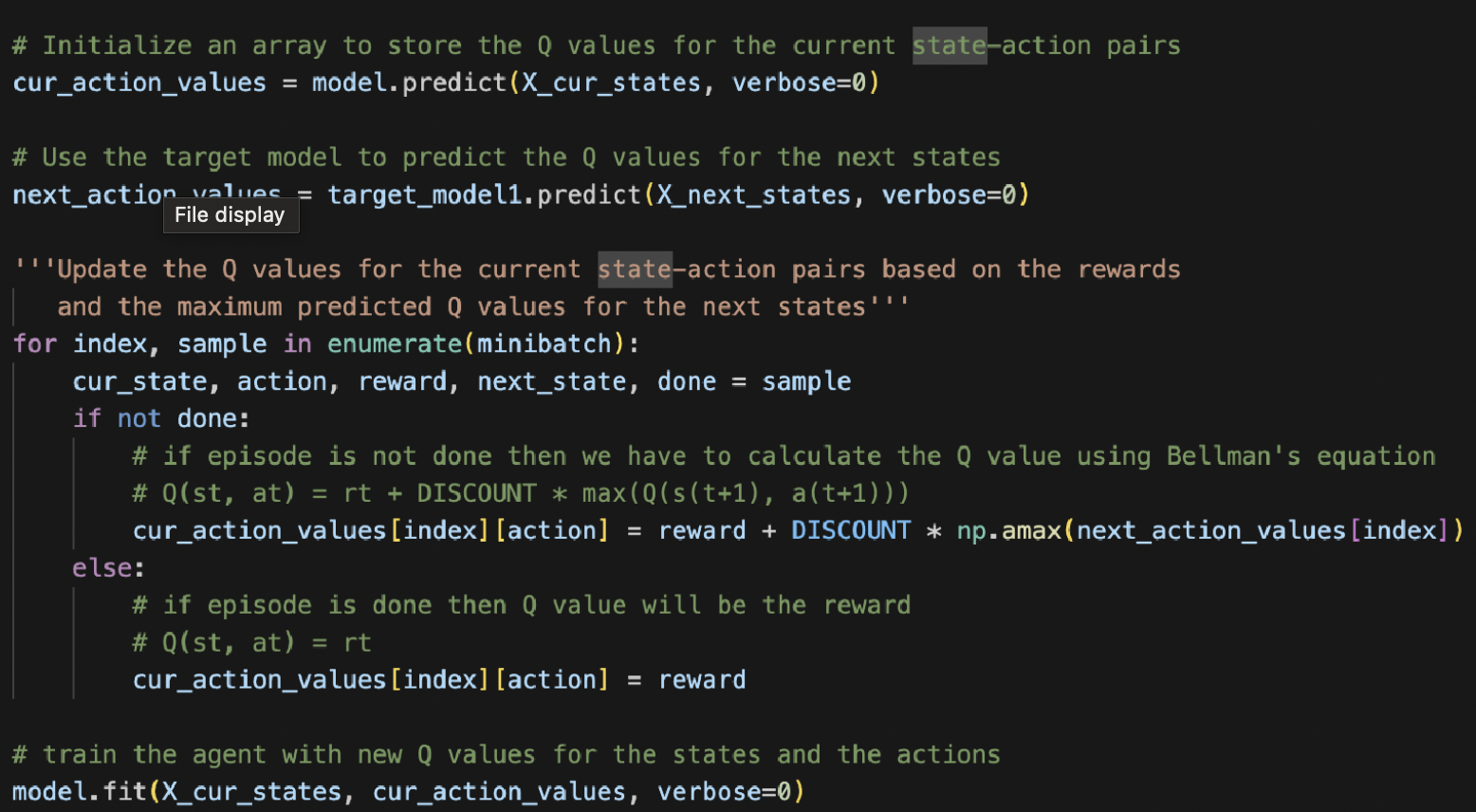


Figure $#2. Code for updation of Q values in Fixed DQN

The training of both model(main network) and target\_model1(target network) starts with random weights. For each episode, we execute a maximum of 10,000 steps. During each step, we select an action using the ε-greedy approach, where the choice of action is determined by a predefined epsilon value, influencing whether an action is selected randomly or based on the current policy.

Since we have adopted an ε-greedy policy, we started with a high value of epsilon as we need to explore the environment and try out various states and actions. Once the agent learns more about the environment, we need it to exploit the values learned to make better decisions on each step. To achieve this, we used Epsilon Decay with a rate of 0.99 times its previous value. We kept a threshold at 0.001 as the lowest value that epsilon can assume. We allowed maximum exploration by setting the value of epsilon as 1 initially.

The Replay memory size is 100000, this means we can keep 100000 records in memory in a deque out of which we can sample experiences for training.

The samples obtained from the replay buffer are used as input for both the training network and the target network and we make the train\_model to predict the current action values i.e., all the actions that can be taken from the current state. This is the initial predicted Q values on top of which the target network (target\_model) is used to predict the next action values i.e., all the actions that can be taken from the next state and selects the maximum value and we use the Bellman equation to assign the Q value for all actions.

The update formula for Q values of a fixed DQN is

Q(s,a) = reward + γ\*maxa′Q(s’,a′),

Where s’ is the next state after s, γ is the discount factor.

The target Q value is the output of the target network(maxa′Q(s’,a′)) + reward from sample. NOTE: the output of the target network is multiplied by the Discount Factor for reducing the impact of the later actions on the current actions and making the results focus on the immediate actions more than the later actions.

The update equation for Fixed DQN is,

Qtrain(s,a) = Qtrain(s,a) + ɑ[R(s,a) + 𝛾 maxQtarget(s’, a’) - Qtrain(s,a)]

With the Bellman equation we are approximating a good Q-function using which we update the weights of the neural network (train and target network). The Training network is updated at every step; however, we update the Target network only every 10000 frames$# to maintain stability in the training process. Finally we call the fit() method on the training network where we have defined 15

the model using the loss function as Mean Square Error and Optimizer as Adam with a learning rate of (0.005)

This in combination with the [Dueling DQN](#5w5oged73gs7) architecture provides us a network that separates action advantage and the value function which enables better learning.

### **3.4 Prioritized Experience Replay to speed up learning**

We wanted to implement the prioritized replay should we have any more time.

The other issue that DQNs face is [Catastrophic Forgetting](#v6pm1s1p7ou6) caused by sampling the data randomly out of the replay buffer(replay memory). Here the batch size and the size of the replay memory plays a critical role. However, when drawing ‘batch\_size’ number of random samples out of the replay memory, there exists a possibility of sampling experiences that does not provide a high reward, as the selection is following a random pattern.

Prioritized Experience Replay provides a way to prioritize the observations that yield a better reward over the ones that do not. The core concepts lies around prioritizing observations that deviate significantly from the current estimate of the Q function. These observations tell us that there is more for the agent to learn.

Here we can define the error of a sample S = (s, a ,r, s’) as a Temporal Difference error

as a distance between the Q(s, a) and its target T(S):

***error=|Q(s,a)−T(S)| (eqn 2)***

For the Dueling DQN described above, we have T(S)= r + γQ̃(s′, argmaxQ(s′,a))

#### **3.4.1 Proposed work plan**

Priorities can be assigned to each experience in the buffer. In order to do this we take the priority value to be directly proportional to the absolute value of error from eqn 2

***Priority = |error| (eqn 3)***

However, when the error = 0, we cannot have the priority value as 0 as that would mean best example would not be picked. Hence, an offset value should be added to fix this issue.

***Priority = |error| + offset***

When we sample these experience we can convert the priority to a probability of choosing that for the current batch. This can be done by

Priorityi =pi/∑kpk

But since the priorities are calculated from the |error| as mentioned in [eqn 3](#ktnc5opu9kjv), which is based on the difference between the Q value and the Q target value which is estimated, the priority could have approximation errors, which could cause some priorities to be too high or too low. In order to address this we need to restrict the priority towards 1 by raising it to a power of a scaling constant a, where a is between 0 and 1. When a = 1 we take full priority sampling and a = 0 would lead to pure random sampling from the experience replay

Probi =(pi/∑kpk)a

However, this creates some bias in the network towards those experiences that have a higher priority. This would lead to overfitting. The loss function taken here is Mean Squared Error(MSE), which we minimize by updating the weights and biases associated with each neurons towards the negative gradient of the loss function(weighted by α). In order to remove the overfitting we need to scale down the update step by a weighting factor according to the probability they were sampled with. This weight can be defined as

Wi = 1/(total replay buffer size) \* 1/probi

This reduces the step size for the higher probability experiences and therefore avoid excessive update steps from the increased frequency of training on those experiences.

Since the bias correction is more important later down the line we can introduce an exponent ‘b’ that starts with a smaller value and increases to 1 over time, similar to the epsilon decay. This weights are referred to as **importance sampling weights**, which needs to be added to the network update step to complete the prioritized replay.

Wi = [1/(total replay buffer size) \* 1/probi]b

—----------------------------------------------------------------------------------------------------------------------------

3.

Implementation (8 marks, approximately 4-6 pages approximately):

a.

Capture and pre-processing of the data (2 marks),

b.

The network structure (2 marks),

c.

The Q learning update applied to the weights (3 marks),

d.

Other concepts that you deem to be of significance, particularly independently researched techniques to speed up learning should you happen to use any (1 marks).

Page 2

Coding fragments and/or diagrams should be included to illustrate the concepts under discussion.

4.

Results (4 marks)

a.

Plots with short accompanying explanations of the information conveyed.

b.

How does one evaluate the performance of the RL agent?

c.

Is the agent learning?

5.

Exploration of recent developments in DQN i.e Dueling DQNs (6 marks)

6.

References (1 marks)

The suggested page count for sections above are for guidance only and are not mandatory.

Submit a Jupyter notebook with the code where:

•

The book is named CS6482-Prj2-ID1-ID2

o

Where ID1 and ID2 are the student id numbers of the team members

•

The first line in the book is a comment with names and ID numbers of the team members

•

The second line in the book should be a comment stating if the code executes to the end without an error.

•

The third line in the book should be a comment with a link to the original source where you opted to reuse an existing implementation.

# **Exploration of recent developments in DQN**

### **Double DQN**

#### Why we need Double DQN

There are two problems that are faced by vanilla DQN and Fixed DQN which are,

A. Maximization bias,

B. Catastrophic forgetting

Catastrophic forgetting is a situation within the environment when the agent does not have access to previously recorded data. Random sampling, that is, the process of randomly picking the state action values from the experience buffer (replay buffer) is one of the causes for this, it can be addressed by prioritized experience replay and Double DQN does not contribute much for the solution?

While with Fixed DQN, we are using maxa′Q(s’,a′) to get the maximum state-action value from the Q table. Taking the maximum values of the state-action values leads to the agent overestimating the action’s value which will lead to choosing that action as the best action and hence the overestimated value will be used as the target. This might lead to the agent learning the Q values incorrectly and is referred to as the Maximization bias.

This problem is addressed by using a Double DQN, as the name suggests, it makes use of two Q networks, that is it uses two different neural network models to select an action and perform an evaluation on that action independently during training. Introduced in the paper, [Hasselt et al. 2015](#r958d8w18gu), addressed the maximization bias problem by the following method. This method uses two separate Q functions to estimate the values and each of these Q functions are updated using one another. While updating Q1, the selection of the best action is done by Q1, however the estimation of its value is carried out by Q2 to tackle overestimating the state-action value to an extent.

The updated equation for Double DQN is,

Qtrain(s,a) = Qtrain(s,a) + ɑ[R(s,a) + 𝛾 Qtarget(s’, maxQtrain(s’,a’)) - Qtrain(s,a)]

### **Dueling DQN**

[Wang et al. 2015](https://docs.google.com/document/d/1rty7EdPXnZ-QGW69LJ3qodtvT1g4EVY9xln6SdU-1s0/edit?pli=1#bookmark=id.1b4fxmuxubjt) presented the Duelling architecture for DQN, which divides the representation of state values and state dependent action advantage into 2 separate branches. Here only the structure of the model changes. Certain games assess the value of each action continuously throughout every time step. In the paper, they talk about the Atari game Enduro(A racing video game), where determining appropriate action is only crucial when a collision with the walls or the opponent cars are imminent.

The dueling architecture allows the model to learn which of the states are valuable without having to learn what each action produces in every state. This can be applied to scenarios where every action taken may not always affect the environment adversely. For instance moving left or right only matter if there is a risk of collision when considering Enduro, and in most states the choice of action plays no role in the outcome.

With Duelling DQN we have Q(s,a) as

***Q(s,a) = V(s,a) + A(s,a) (eqn. 1)***

the sum of V(s,a), the value of being in state s and A(s,a) which is the advantage of taking that action at that state (how much better is to take this action versus all other possible actions at that state).

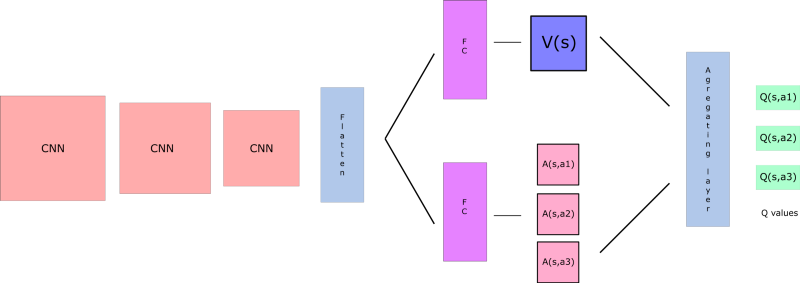


Figure $#6. Dueling DQN Architecture

source:<https://www.freecodecamp.org/news/improvements-in-deep-q-learning-dueling-double-dqn-prioritized-experience-replay-and-fixed-58b130cc5682/>

Here we try to separate the estimator of V(s,a) and A(s,a), then we combine the two using aggregation layer to get an estimate of Q(s,a) as illustrated in Figure $#6

The example provided by [Wang et al. 2015](https://docs.google.com/document/d/1rty7EdPXnZ-QGW69LJ3qodtvT1g4EVY9xln6SdU-1s0/edit?pli=1#bookmark=id.1b4fxmuxubjt) is using the Enduro makes this clearer.

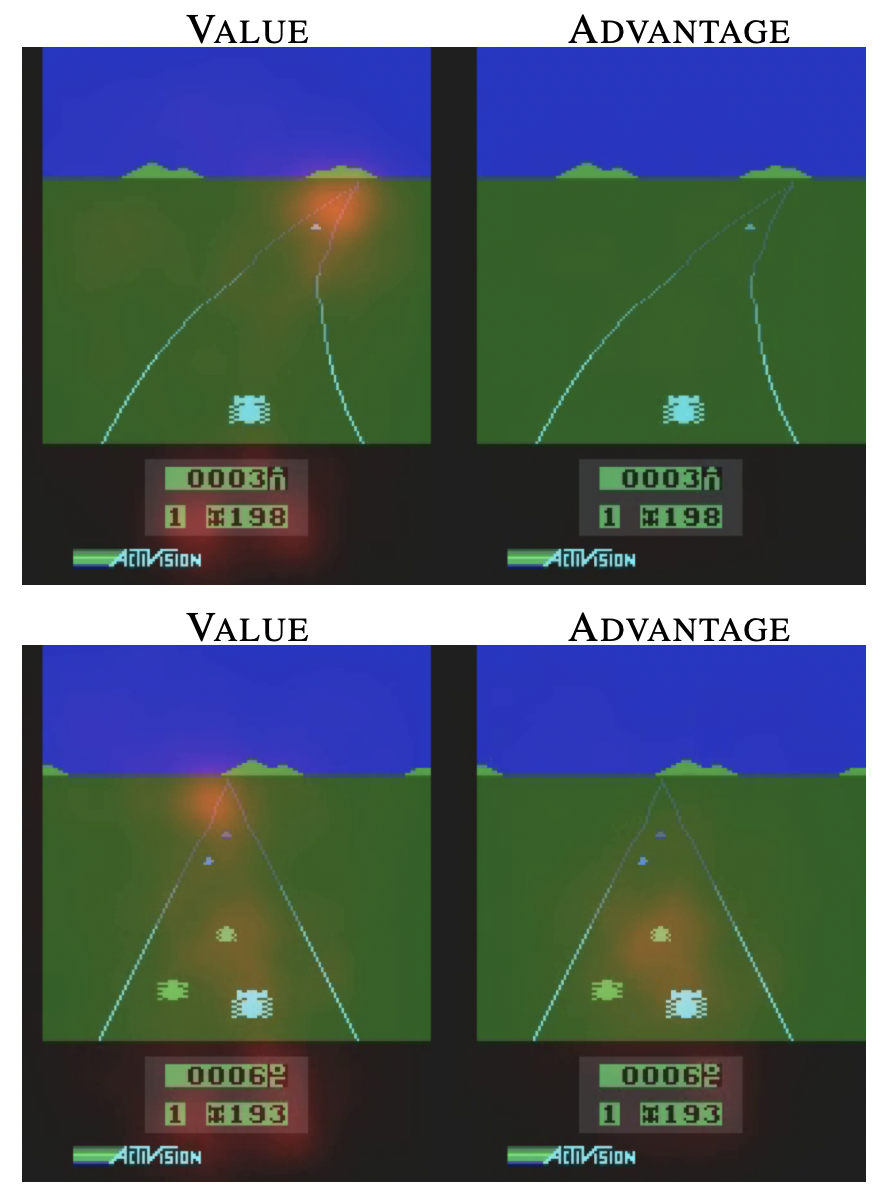


Figure $#7. Interpretation of splitting the value and action advantage to 2 branches

As illustrated in the Figure $#7, we see the agent focusing on the horizon and the score obtained(indicated using the orange spot) in the image corresponding to the Value(top left) as the target here would be for the agent to get the car to that state. However, the image for the advantage(top right) does not pay any attention as there are no cars in front of the car, the agent does not find it critical to take any alternate actions.

The images on the bottom show almost the same behviour with the value, however, notice that the image corresponding to the action advantage(bottom right) has a focus on the car in front, and the agent deems it worth to take the action to swerve to avoid collision.

This illustrates why the dueling DQN separates out the action advantage and the value function.

When combining the action advantage and the value function we need to subtract the mean of the action advantage from each of the action advantage values, rather than applying [eqn. 1](https://docs.google.com/document/d/1rty7EdPXnZ-QGW69LJ3qodtvT1g4EVY9xln6SdU-1s0/edit?pli=1#bookmark=id.ab4x9a78jrls). This helps with the identifiability issue, where in if provided with a Q value we need to identify the V and the Action Advantage. This is also helpful when working with scenarios where each of the actions would give the same Q value. This would mean that there is no good action in this state. If every action has the same result, then the Advantage of each action will have the same value. Now, if we subtract the mean of all the Advantages from each advantage, we get zero (or close to zero) and Q-value would actually be the Value that the state has. So overtime the Q-value would not overshoot thereby learning without ambiguity is enabled. The states that are independent of action would not have a high Q-value to train on. Hence we prefer

***Q(s,a) = V(s,a) + [A(s,a) - 1/|A|(A(s,a)],*** where |A| is the count of Action advantage

$# insert something here

# **Results**

|  |  |
| --- | --- |

Figure $#8. Fixed DQN showing maximization bias

# **Bibliography**

<https://web.stanford.edu/class/archive/cs/cs221/cs221.1192/2018/restricted/posters/yli03/poster.pdf>

Mnih, Volodymyr ., Kavukcuoglu, Koray., Silver, David., Graves, Alex., Antonoglou, Ioannis., Wierstra, Daan., Riedmiller, Martin., 2013, ‘Playing Atari with Deep Reinforcement Learning’, DeepMind Technologies, <https://arxiv.org/pdf/1312.5602.pdf>

Cowher, Robert., 2023, ‘Dueling DQN Breakout Pytorch’, Github, <https://github.com/bobcowher/duelling-dqn-breakout-pytorch/tree/main>.

Chapman, Jacob., Lechner, Mathias., 2020, ‘Deep-Q Learning for Atari Breakout’, Keras, <https://keras.io/examples/rl/deep_q_network_breakout/>

Wang, Ziyu., Schaul, Tom., Hessel, Matteo., van Hasselt, Hado., Lanctot , Marc., de Freitas, Nando, 2016, ‘Dueling Network Architectures for Deep Reinforcement Learning’, Google DeepMind, <https://arxiv.org/pdf/1511.06581>.

Gymnasium Documentation, ‘Breakout’, <https://gymnasium.farama.org/environments/atari/breakout/>

[Mnih et al. “Human-level control through deep reinforcement learning”, Nature 26 Feb. 2015](http://www.nature.com/nature/journal/v518/n7540/full/nature14236.html)

Hasselt, Hado van, et al.(2015) “Deep Reinforcement Learning with Double Q-learning.” *Association for the Advancement of Artificial Intelligence*. Available at: <https://arxiv.org/pdf/1509.06461>

Yoon, C. (2019, October 20). Dueling Network Architectures for Deep Reinforcement Learning. Towards Data Science. Available at <https://towardsdatascience.com/dueling-deep-q-networks-81ffab672751>