Comparative study between DQN & Dueling DQN with Prioritized Experience Replay on 2048 game environment

CS 5180 Fall 2021

Submitted To: Robert Platt **By : Sai Prasanth Duvvuri & Shireen Firdoz**

Project Motivation:

In previous years there have been many breakthrough achievements using deep representations in reinforcement learning. While we were learning about convolutional networks, upon research we found out that the dueling architecture can be easily combined with other algorithmic improvements. In another research paper, prioritization of the experience replay has shown significant improvement in the performance of Atari games (Schaul et al., 2016).However, as dueling architecture and prioritization addresses altogether different aspects of the learning process, their amalgamation is awesome. Moreover, as published that the dueling architecture enables Reinforcement Learning agent to outperform the state-of-the-art on the Atari 2600 domain.Therefore, in our final experiment, we investigated the integration of the dueling architecture with prioritized experience replay and compared it with DQN implementation on 2048 in order to find the most efficient way to solve the 2048 game.

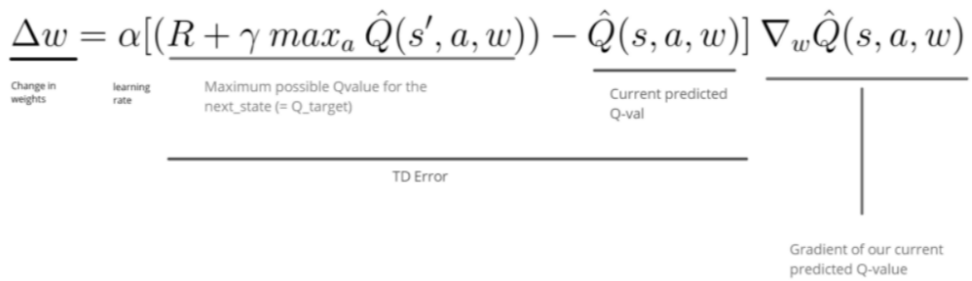
Goal:

Applying various deep Q techniques and finding the most efficient way to solve the 2048 game environment, which resulted in comparative study between DQN and Dueling DQN with prioritized experience replay on 2048.

Project Description & Algorithms:

We investigated the various deep Q techniques and related concepts like DQN, Double DQN, Dueling DQN,Experience Replay, Prioritized Experience Replay to finalize the comparative study between two techniques DQN and Dueling DQN (with PER).

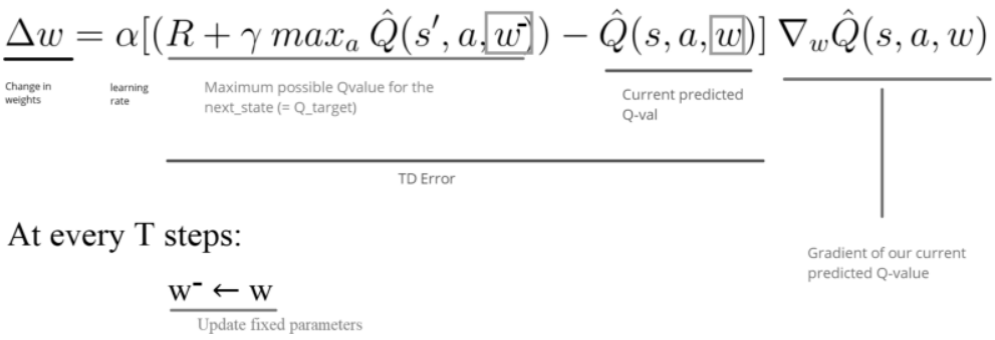
a) **Why on 2048 environment ?** 2048 game environment has 4 random components which are categorized as the initial configuration of the game, after every move addition of random tiles,unavailability of moves and exploration of the agent. However these four random components raise an issue of inherent randomness in the 2048 game. Therefore to solve this issue of inherent randomness, a prioritized experience replay buffer is used where the model trains to learn the best game-playing strategy from the experiences collected.
b) **DQN :** is an algorithm which calculates TD error using the difference between the TD target and the current Q values.



Using Bellman equation, to get an estimate of the TD target which is the sum of reward of taking that action at that state(s) and the discounted highest Q value for the next state(s')

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

Now the issue is we are using the same parameters of weight for estimating the target and the Q value and we are changing the parameter w, which results that at every step of training our Q value shifts however the target value also shifts.So as we are getting closer to the target the targets also moves. Therefore to solve this issue we used a fixed Q-target and calls it w- for estimating the TD target and at every T steps we copy the parameters from our DQN network to update the target network because of which we have stable learning and the target function also stays fixed.



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

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

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

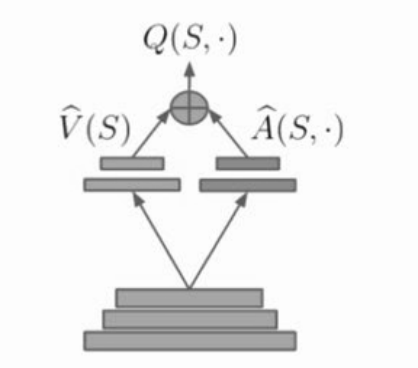
    Perform a gradient descent step on $\left(y_j - Q(\phi_j, a_j; \theta)\right)^2$ with respect to the network parameters θ

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

  End For

End For

c) How Does Dueling DQN Help ? Dueling DQN separately depicts the state values and action advantage. It consists of two streams one represents the value function and second represents the advantage function, however it shares the common convolutional feature learning module. So the two streams are combined by a separate aggregating layer to get an estimate of Q which is a state-action value function. Therefore the benefit of using such an architecture is it can learn which states are valuable and which are not without learning the effect of each action for each state, which is beneficial in a situation or states where its actions does not affect the environment in any significant way.

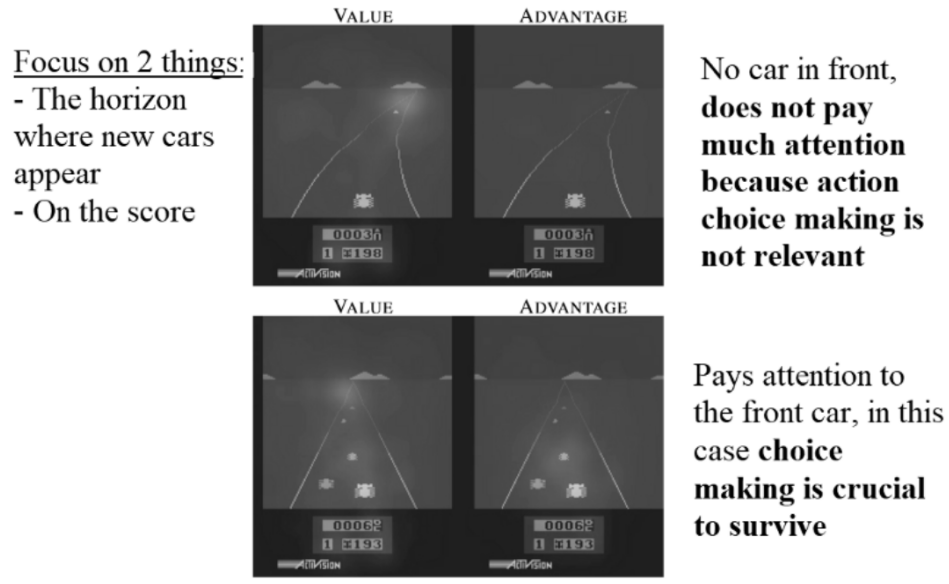


Q(s,a) : Q-values tells how good it is to be in the state s by taking action at that state.

Q(s,a) = V(s) value of being in that state + A(s,a) the advantage of taking action a in the state s.

A(s,a) : how much better is to take this action versus all other possible actions at that state.

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



We take an example mentioned in the Dueling Network research paper .

As depicted in the diagram that value network stream focus on the road and to the horizon. It also pays attention to the scores whereas the advantage stream in the first frame right section does not pay much attention to the road as there are no cars in front of it but in second frame right side advantage stream should focus on the car as well which makes the action choice more important.Therefore the below equation subtracts the average advantage of all actions possible at the state.

Therefore, this setup improved the learning where one can calculate the value function without calculating the Q(s,a) for each action at that state and the Q values for each action by decoupling the estimation between two streams are more relevant.

$$Q(s,a;\theta,\alpha,\beta) = V(s;\theta,\beta) + \left(A(s,a;\theta,\alpha) - \underbrace{\frac{1}{\mathcal{A}}\sum_{a'} A(s,a';\theta,\alpha)}_{\text{Average advantage}}\right)$$

Common network parameters

Value stream parameters

Advantage stream parameters

d) **Experience Buffer :** The parameters like policy, value function are incrementally updated by the Reinforcement learning agents as they observe a stream of experiences and disregard the incoming data immidiately, after a single update due to which two issues occur.

   i) Strongly correlated updates which fails the assumption of stochastic gradient-based algorithm.

   ii) The rapid forgetting of possibly rare experiences that would be useful later on.

   Experience replay solves both the above problems by storing the experience in the replay buffer, which breaks the temporal correlations by mixing more and less recent experience for the updates. It can also reduce the amount of experience required to learn and replace it with more computation and more memory which are cheaper resources than Reinforcement learning agents interacting with the environment.

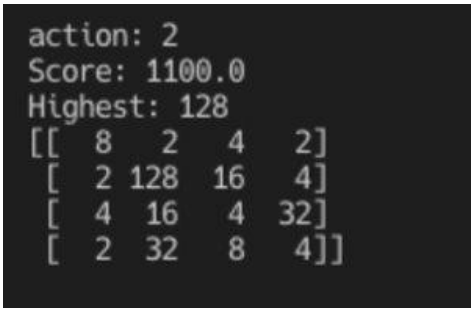
e) **Prioritized Experience Buffer :** Prioritization of the experiences can make a difference and at times has proved effective and efficient instead of all the transitions replayed uniformly.This can be established because Reinforcement learning agents learn more effectively from some transitions than from others. In uniform transition replay all transitions may not be effective and task relevant. Also few transitions may not be effective in the beginning but as the agent competence increases they become effective and useful. Prioritized replay liberates agents considering the transitions with the same frequency that they are experienced**.**

$$p_t = \underbrace{|\delta_t|}_{\text{Magnitude of our TD error}} + \underbrace{e}_{\text{Constant assures that no experience has 0 probability to be taken.}}$$

f) **Logic behind prioritized experience replay ?** We have used more frequent replay transitions with maximum expected learning which is measured by the temporal difference error. However this can lead to a loss of diversity which we have resolved using stochastic prioritization and introducing bias.

Implementation:

Environment: We have written the 2048 environment ourself with reference to the 2048 openai gym environment.



State Space: All observations are 4 x 4 arrays representing the game screen of 2048 env. If the location in the array is 0, it means that there doesn't exist a tile in that location. TIles are randomly generated from all the directions. A cell with a non-zero value signifies that there is a tile of the value at that location. Typically, these values are powers of 2. Merging 2 tiles would result in the tiles adding up if they both have the same value.

Action Space: We can alter the state space by performing actions on it. We can perform UP, DOWN, LEFT, RIGHT actions on the state space which are represented as 0,1,2 and 3 respectively.

Reward: Default reward implemented by the openAI environment was to generate the scores by the moves executed. This is nothing but the scores obtained by the merged tiles for the given move. We intended to experiment with various reward functions. One such reward function is to maximize the episode length. This is because 2048 is a very unstable environment and one wrong move would quickly result in losing the game. SInce we want the agent to maximize its ability to stay alive for as long as possible. Note that this will help the agent learn the strategies such as, not wanting to make a move which would generate maximum reward but to postpone the move to a few more steps to stay alive for longer duration (if there exists such a way). Hence this is a very simple stochastic, deterministic, fully observable environment with discrete action space. We are still in the experimentation phase with our newly proposed reward function.

DQN Architecture :

We implemented a simple 2 layered Convolutional Neural Nets for the DQN. You can see the implementation below.

```
class CNN_Net(nn.Module):
    def __init__(self, input_len, output_num, conv_size=(32, 64), fc_size=(1024, 128), out_softmax=False):
        super(CNN_Net, self).__init__()
        self.input_len = input_len
        self.output_num = output_num
        self.out_softmax = out_softmax

        self.conv1 = nn.Sequential(
            nn.Conv2d(1, conv_size[0], kernel_size=3, stride=1, padding=1),
            nn.ReLU(inplace=True)
        )
        self.conv2 = nn.Sequential(
            nn.Conv2d(conv_size[0], conv_size[1], kernel_size=3, stride=1, padding=1),
            nn.ReLU(inplace=True),
        )

        self.fc1 = nn.Linear(conv_size[1] * self.input_len * self.input_len, fc_size[0])
        self.fc2 = nn.Linear(fc_size[0], fc_size[1])
        self.head = nn.Linear(fc_size[1], self.output_num)

    def forward(self, x):
        x = x.reshape(-1,1,self.input_len, self.input_len)
        x = self.conv1(x)
        x = self.conv2(x)
        x = x.view(x.size(0), -1)
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))

        output = self.head(x)
```

Dueling DQN Architecture:

We implemented a 2 layered Convolutional Neural Nets with a branching (same as the architecture shown in “How Does Dueling DQN Help?” Section) for the Dueling DQN. You can see the implementation below.

```
def __init__(self, c_in_1, c_in_2, c_out_1, c_out_2):
    super(CNN_2048_MODEL, self).__init__()
    self.__c_in_1 = c_in_1
    self.__c_in_2 = c_in_2
    self.__c_out_1 = c_out_1
    self.__c_out_2 = c_out_2

    self.__expanded_size = (
        2 * 4 * c_out_2 * 2 + 3 * 3 * c_out_2 * 2 + 4 * 3 * c_out_1 * 2
    )

    self.__dense_value_1 = nn.Linear(self.__expanded_size, 256)
    self.__dense_value_2 = nn.Linear(256, 1)
    self.__dense_advantage_1 = nn.Linear(self.__expanded_size, 256)
    self.__dense_advantage_2 = nn.Linear(256, 4)

    self.__cnn_1 = nn.Conv2d(
        c_in_1,
        c_out_1,
        kernel_size=(1, 2),
        stride=(1, 1),
        padding=(0, 0),
        dilation=(1, 1),
    )

    self.__cnn_1_2 = nn.Conv2d(
        c_out_1,
        c_out_2,
        kernel_size=(1, 2),
        stride=(1, 1),
        padding=(0, 0),
        dilation=(1, 1),
    )

    self.__cnn_2 = nn.Conv2d(
        c_in_2,
        c_out_2,
        kernel_size=(2, 1),
        stride=(1, 1),
        padding=(0, 0),
        dilation=(1, 1),
    )

    self.__cnn_2_2 = nn.Conv2d(
        c_out_1,
        c_out_2,
        kernel_size=(2, 1),
        stride=(1, 1),
        padding=(0, 0),
        dilation=(1, 1),
    )
```

SumTree:

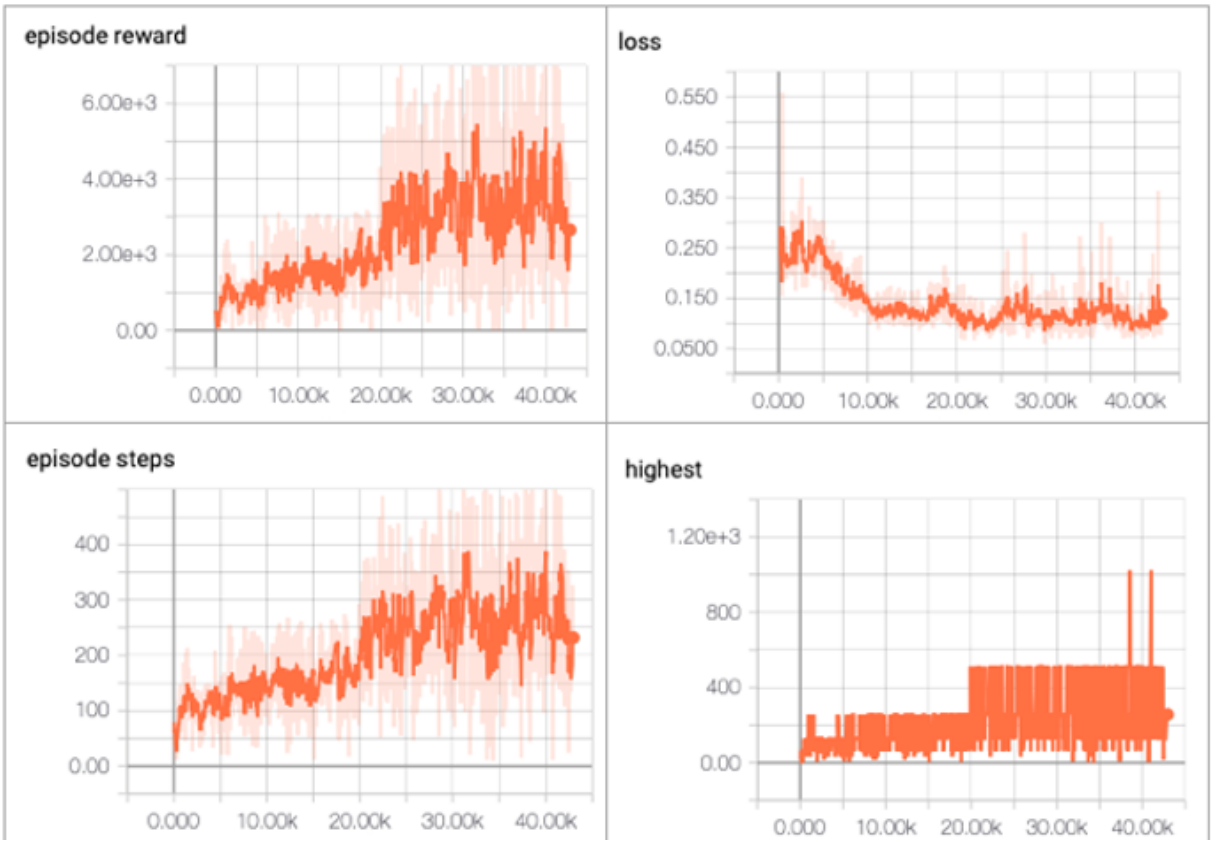
This is used to implement the prioritized experience replay buffer. This is a data-structure which stores a pair of transition and TD errors. The prioritization on the TD error shows how important a particular transition is to the training of a neural network. Higher the TD error, the better for the network to tune itself. During the experience process, we select a mini-batch of transitions and update the TD-errors for this mini-batch. 40% of the transitions are randomly selected. Below is a snippet of the code or SumTree data structure used for Experience Replay.

```
class SumTree:
    def __init__(self, capacity, size_board=4):
        self.__data_pointer = 0
        self.__capacity = capacity
        self.__tree = np.zeros(2 * capacity - 1)

        self.__state = np.zeros((capacity, 1, size_board, size_board, 16))
        self.__action = np.zeros(capacity, dtype=np.int64)
        self.__reward = np.zeros(capacity)
        self.__next_state = np.zeros((capacity, 1, size_board, size_board, 16))
        self.__done = np.zeros(capacity, dtype=np.bool_)

    def update(self, tree_index, priority):
        change = priority - self.__tree[tree_index]
        self.__tree[tree_index] = priority
        while tree_index != 0:
            tree_index = (tree_index - 1) // 2
            self.__tree[tree_index] += change

    def add(self, priority, state, action, reward, next_state, done):
        self.__state[self.__data_pointer] = state
        self.__action[self.__data_pointer] = action
        self.__reward[self.__data_pointer] = reward
        self.__next_state[self.__data_pointer] = next_state
        self.__done[self.__data_pointer] = done

        tree_index = self.__data_pointer + self.__capacity - 1
        self.update(tree_index, priority)

        self.__data_pointer += 1

        if self.__data_pointer >= self.__capacity:
            self.__data_pointer = 0

    def get_leaf(self, value):
        parent_index = 0

        while True:
            left_child_index = 2 * parent_index + 1
            right_child_index = left_child_index + 1
            if left_child_index >= len(self.__tree):
                leaf_index = parent_index
                break

            else:
                if value <= self.__tree[left_child_index]:
                    parent_index = left_child_index
                else:
                    value -= self.__tree[left_child_index]
                    parent_index = right_child_index

        data_index = leaf_index - self.__capacity + 1
```

Experiment:

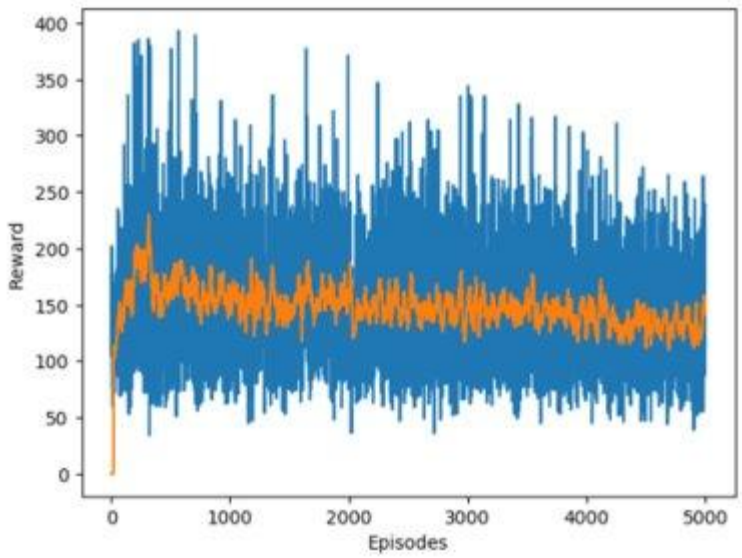
Plots for DQN on 2048 environment:

Below are the graphs for the DQN experiment

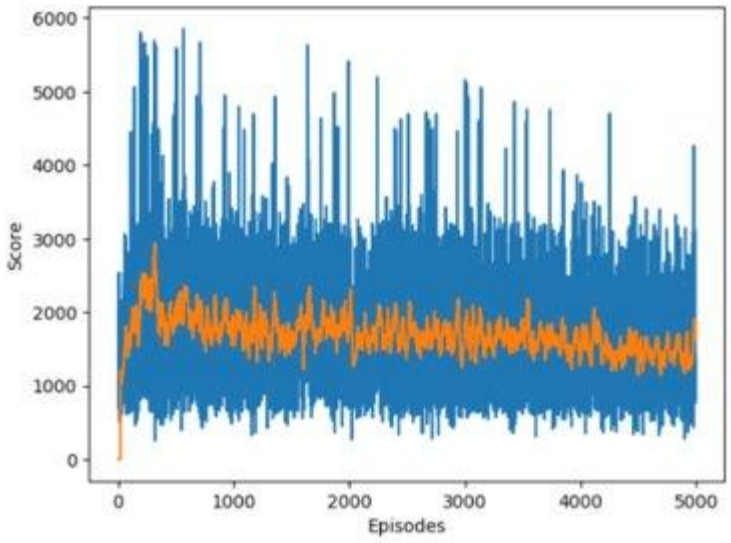


a) episode reward plot signifies the episode vs reward. b) loss plot signifies the episode vs loss. c) step plot signifies the episode vs episode length. d) highest plot signifies the episode vs highest tile generated for the given episode.

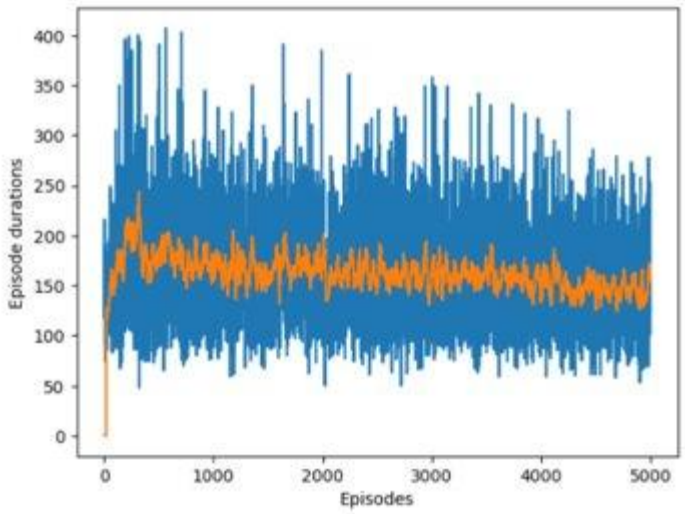
Plots for Dueling DQN on 2048 environment:



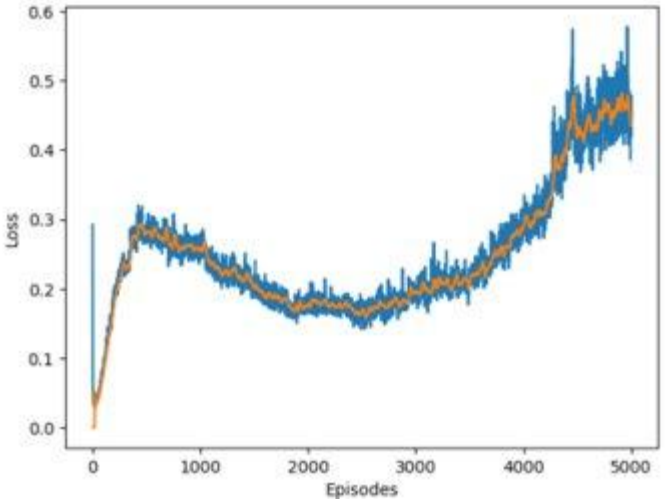
The above graph signifies the episode vs reward graph. The orange line indicates the moving average of reward for 20 episodes. Reward over here is the maximum tile generated for the given episode.



The above graph signifies the episode vs score graph. The orange line indicates the moving average of score for 20 episodes. Score over here is the sum of the scores generated for each of the moves for a given episode.



The above graph signifies the episode vs episode length graph. The orange line indicates the moving average of the episode length for 20 episodes. For the 2048 environment, the longer the episode length the better the agent.



The above graph signifies the episode vs loss of a graph. The orange line indicates the moving average of the loss for 20 episodes.

We can observe that dueling DQN agent reach approximately the same performance in 6000 episodes that DQN takes 45000 episodes to reach. This signifies the sample efficiency of the Dueling DQN algorithm over the vanilla DQN algorithm. Besides, the PER ensures that the agent prioritizes its learning hence making it more sample efficient.

Interpretability:

As we did the project and the comparative study, we also took the liberty to interpret the agent's learning. We really enjoyed this phase of the project because this helped us visualize the strategies that the AI learned to play the game.

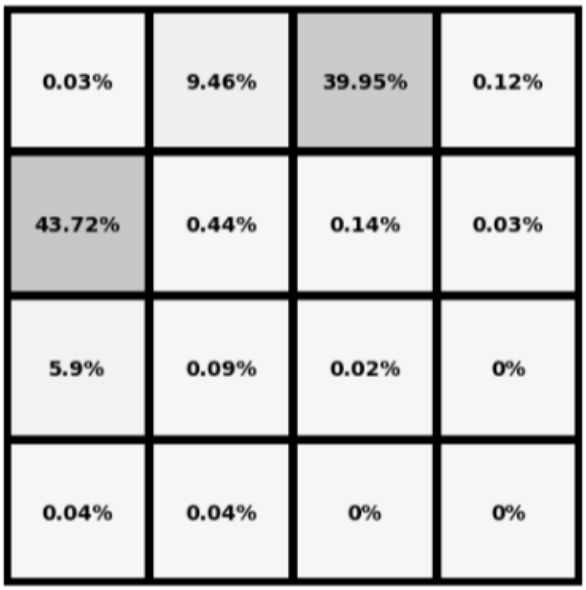
1) One of the strategies that the agent learnt very early in the game was to ensure that the maximum tail of the board so far was at the top left of the board and then build the rest of the game. This is a pretty simple and straightforward strategy that we use while playing this game. Me wrote a simple routine to keep track of the position of the highest tile while running our test runs and we found the below heatmap. 99.8% of the times we observed that the highest tile is always on the top left of the cell.

99.81%	0%	0%	0%
0.01%	0%	0%	0%
0.02%	0%	0%	0%
0.15%	0%	0%	0%

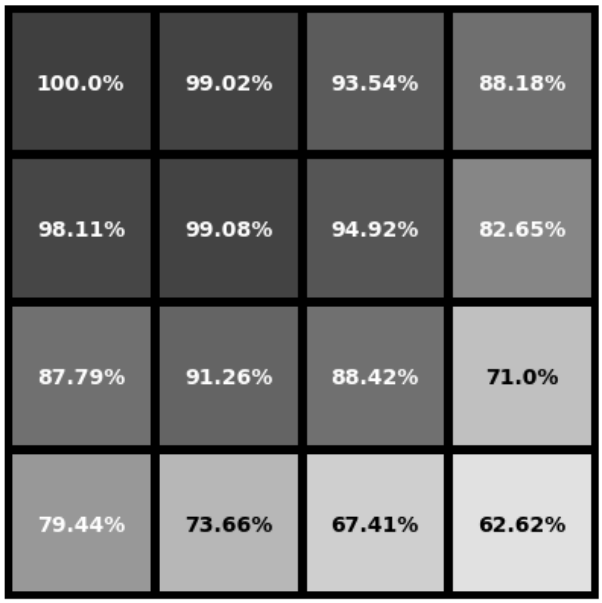
2) We extended our curiosity to see the heatmaps of where the second and third highest tiles of the game were kept, as a human, while playing, we keep the second and third highest tile at either [0,1] or [1,0] and [0,2] or [2,0] respectively. We were surprised to see that the agent had captured this strategy and below were the heatmaps. Below is the frequency of the second highest tile's location.

0.04%	84.1%	0.01%	0%
15.64%	0.03%	0%	0%
0.15%	0.01%	0%	0%
0.02%	0%	0%	0%

Below is the frequency of the third highest tile's location.



3) We have also observed that the agent keeps as many tiles as possible on the board. By doing this the agent ensures to control the location where the random tiles spawn hence making the state more predictable. We have also plotted a heatmap to see the location of the cells which the agent left unoccupied. 100% means that the agent kept the respective cell occupied 100% of the time. We also observed that this was the sorted ordering that was always maintained while making the moves.



As we discussed previously all these are the same strategies that we humans use playing this game.

Conclusion:

The key insight behind Dueling DQN is that it is not necessary to estimate the value of each action choice for many states which means that the dueling architecture can learn which states are valuable for each state without learning the effect of each action. In some states, it is of paramount importance to know which action to take, but in many other states the choice of action has no repercussions on what happens. This is particularly useful in states where its actions in no relevant way affect the environment. (which is the case with the 2048 environment). In addition to this, for a more stable optimization, we use an average baseline for Q evaluation. The dueling architecture and prioritization addresses different aspects of the learning process, however we intend to experiment their combination and the results were promising. The episodes were prioritized by the absolute value of each transition's TD error. During the implementation we faced the issue where some of the transitions subsets were more frequently occuring, to resolve this issue we used a priority queue implemented by SumTree to do a stochastic sampling that interpolates between 60% of the prioritization on the TD error and 40% on the uniform random sampling. The results that we got were Dueling DQN with PER converges in 5000 episode length whereas DQN on 2048 env game tools around 40,000 episode length to converge.

Future Scope:

Here we compared the Dueling DQN along with PER with the DQN algorithm on 2048 gym environments. In the future we will be developing code to compare Dueling DQN with other Deep Q technique algorithms to prove that it is one of the most efficient algorithmic techniques to solve the 2048 game environment.

References:

a) 1511.05952v4.pdf (arxiv.org)

b) How to implement Prioritized Experience Replay for a Deep Q-Network | by Guillaume Crabé | Towards Data Science

c) Understanding Prioritized Experience Replay (danieltakeshi.github.io)

d) Improvements in Deep Q Learning: Dueling Double DQN, Prioritized Experience Replay, and fixed… (freecodecamp.org)

e) 1511.06581.pdf (arxiv.org)

f) Improving the Double DQN algorithm using prioritized experience replay | Stochastic Expatriate Descent (davidrpugh.github.io)

g) *Game playing agent for 2048 using Deep Reinforcement Learning*