Machine Learning Engineer Nanodegree

Capstone Project

UDACITY

Capstone Project

Prepared by: Apoorv Malik

13 August 2019

1. Definition

1.1. Project Overview

In the field of Reinforcement Learning, numerous researches are ongoing. The prevalent trend in Artificial Intelligence is teaching AI agents to play video games. In 2014, Google acquired a company called Deep Mind. It created a neural network that learns how to play video games in a fashion similar to that of humans. The company made headlines in 2016 after its AlphaGo program beat a human professional Go player Lee Sedol, the world champion. It created an AI program that achieved outstanding performance on many different Atari 2600 games.

Research in this field is progressing at a swift pace. New techniques and algorithms are being discovered every year. It would be a great learning experience to implement the Deep Q-Learning algorithm and to design an AI agent that can beat a video game level on its own. Note that the DQN Agent is given only raw pixel data of what's happening on the game's screen.

One very famous game is Atari Breakout. The goal of this capstone project would be to develop a model using deep neural networks (using Deep Q-Learning algorithm) to train an AI agent to score as high as possible in the game.

My motivation for investigating this problem is that I am very fond of designing and testing AI agents to solve Reinforcement Learning tasks. Moreover, now that I understand how the Deep-Q Learning algorithm works, I ventured to implement it myself and to watch an agent get better and better through interaction.

1.2. Problem Statement

The main objective of this project is to design an AI agent that can score as high as possible in Atari Breakout. In Breakout, a layer of bricks lines the top third of the screen, and the goal is to destroy them all. A ball moves straight around the screen, bouncing off the top and two sides of the screen. When a brick is hit, the ball bounces back, and the brick is destroyed. The agent receives a reward of +1 for every brick destroyed. The agent loses a turn when the ball touches the bottom of the screen; to prevent this from happening, the agent has a horizontally movable paddle to bounce the ball upward, keeping it in play. There is a total of five turns for the agent. The episode terminates when all five turns are lost. The action space consists of four total actions: 'NOOP', 'FIRE', 'RIGHT', 'LEFT'. Given the state of the game in the form of raw pixel data, the agent needs to find the most appropriate action that will maximise the reward it will receive from the environment. The environment is completely solved when all the bricks are destroyed.

1.3. Metrics

The standard evaluation metric for Atari game playing is the score returned by the game itself. The score is incremented on every successful hit on the brick by the ball. There are six rows of bricks. The color of a brick determines the points scored when the ball hits it. These are as follows:

Red - 7 points   **Orange - 7 points**   **Yellow - 4 points**

Green - 4 points   **Aqua - 1 point**   **Blue - 1 point**

The player has five total turns, and the game gets over if all the five turns are over (A turn gets lost if the ball hits the bottom of the screen). The total score is calculated at the end of the game by adding all the points scored during the game. No score penalty is applicable in this environment.

The reward/score system for the DQN agent will follow different score criteria. Moreover, the real performance and evaluation of the agent will be measured on the basis of original score criteria mentioned above. The reward/score system for the agent is as follows:

The rewards received by the agent (after hitting each block) is scaled between +1 and -1 (there is no negative reward in this environment). Since the scale of scores varies significantly from game to game, the positive rewards are fixed to be +1 and all negative rewards to be −1, leaving 0 rewards unchanged. Clipping the rewards in this manner limits the scale of the error derivatives and makes it easier to use the same learning rate across multiple games. At the same time, it could affect the performance of our agent since it cannot differentiate between rewards of different magnitude. The episode ends after a single turn is lost. However, the game resets only on true game over. This is done by DeepMind in their original solution since it helps value estimation. The total reward obtained by the agent at the end of the episode will be the total number of bricks hit (irrespective of their points), while the total points scored by the agent follow the original Breakout game's score criteria.

2. Analysis

2.1. Data Exploration

For this project, we will use the OpenAI's Gym library. OpenAI is a company created by Elon Musk that has been researching deep reinforcement learning. OpenAI Gym is a toolkit for developing and comparing reinforcement learning algorithms. The open-source gym library gives us access to a standardised set of environments. It makes no assumptions about the structure of the agent and is compatible with any numerical computation library, such as TensorFlow or Theano. The Gym library is a collection of test problems — **environments** — that we can use to work out our reinforcement learning algorithms. These environments have a shared interface, allowing us to write general algorithms, that facilitate similar environments.

There are two primary components of reinforcement learning, the environment, and the agent. The agent interacts with the environment by choosing some action (from the set of all possible actions), and the environment responds to the agent by changing its state(observation) and giving out rewards(score) to the agent.

The dataset for this project comes from the OpenAI Gym's Atari game environment. For this project, we will use the Atari Breakout environment. We will use the modified version of the original Breakout environment, designed by Google DeepMind. This new version is called "BreakoutDeterministic-v4". In this version, the agent sees and selects actions on every 4th frame instead of every frame, and its last action repeats on the skipped frames. The action space of this environment consists of four possible actions, 'NOOP', 'RIGHT', 'LEFT', 'FIRE'. The definitions of these actions are as follows:

'NOOP': No Operation, no *real action* is executed when this is selected.
'RIGHT': This action moves the horizontal paddle to the right.
'LEFT': This action moves the horizontal paddle to the left.
'FIRE': This action bounces the ball upward, only executable once and at the start of a new episode.

2.2. Exploratory Visualization

The dataset for this project will be the raw pixel data representing the game’s screen. This will be the only input data that we will give to the model. The input data will vary as the agent progress in the simulation. The model is provided a new image as input at every time step.

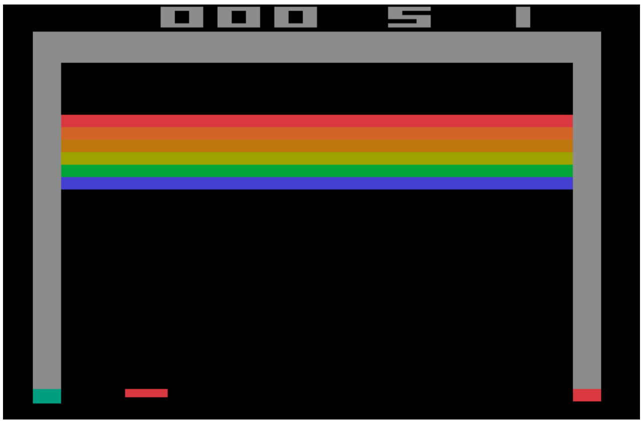
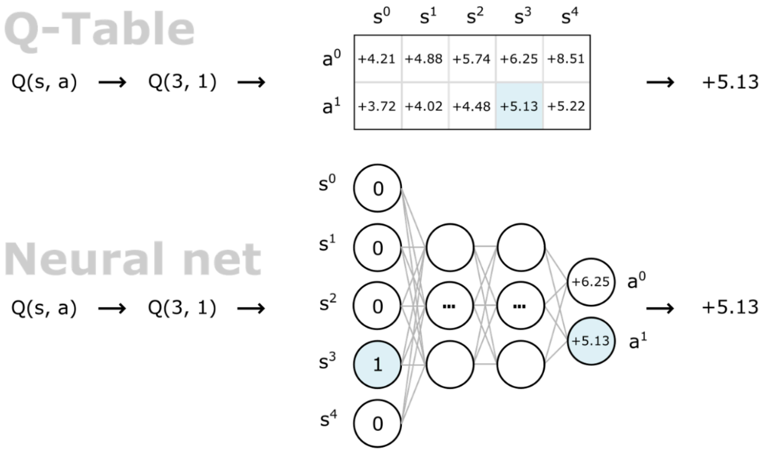


Figure 1: Starting image frame from Breakout.

The input that we will give to our Deep Learning Model will be the raw pixel data of the game's screen. The Atari frames are 210×160 pixel images with a 128 color palette. Providing these raw pixel data can be computationally challenging, and the algorithm may converge very slow due to high dimensional and complex input. So we need to apply an essential preprocessing step, the raw frames are preprocessed by first converting their RGB representation to a grey-scale representation, and then they are down-sampled to an 84×84 image. The last four frames (before the current frame) are preprocessed and then stacked together to produce a tensor of shape (1, 84, 84, 4) which will be the input to the DQN model. This stacking of previous frames allows the model to infer the velocity of the ball and make better decisions while choosing actions.

2.3. Algorithms and Techniques

Recall that in Q-Learning we use Q-table to store the Q-values. But if the number of states are very large, this algorithm will be inefficient, as storing a large amount of values is very difficult. Therefore, the original Q-Learning algorithm fails for high dimensional and very large state spaces. In Deep Q-Learning (DQN), instead of storing the Q-values in a table, we will use neural networks to approximate the policy function which maps each state to an appropriate action.



	s^0	s^1	s^2	s^3	s^4
a^0	+4.21	+4.88	+5.74	+6.25	+8.51
a^1	+3.72	+4.02	+4.48	+5.13	+5.22

Figure 2: Q-Table versus Neural Network

We are dealing with a problem consisting of high dimensional state space. Therefore we cannot use the original Q-learning algorithm for this problem, instead we will use Deep Neural Networks and use Deep Q-Learning algorithm to train them.

To train the Deep Neural Network model, we will use the DQN algorithm. The network consists of convolutional layers that can extract information from the image data, followed by fully-connected layers to form relations between states and actions. With the use of both convolutional and fully-connected neural network layers, the model can estimate the policy function.

The output layer is a fully-connected linear layer with a single output for each of the valid action. The output of the model will be a vector consisting of four values (for four valid actions) with each value corresponding to the probability of taking that action. The visual representation of the deep neural network is shown below.

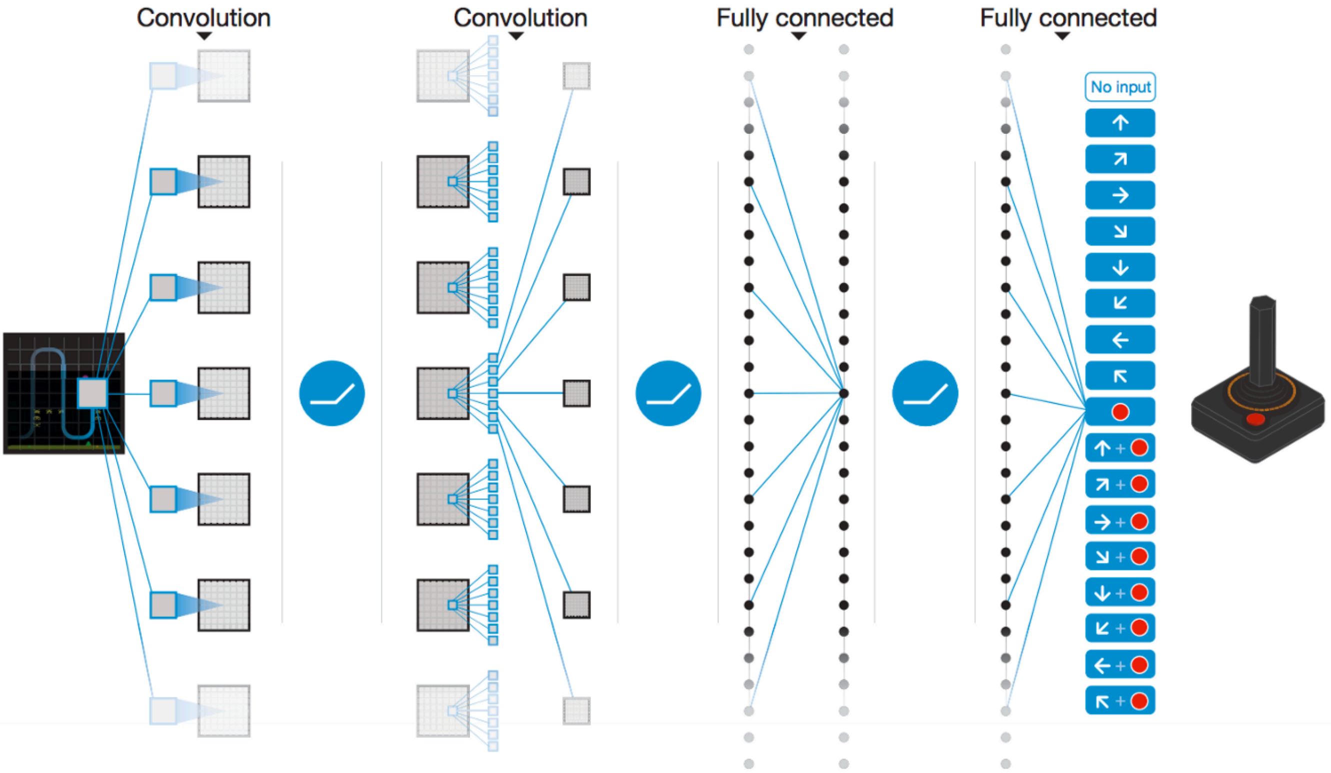
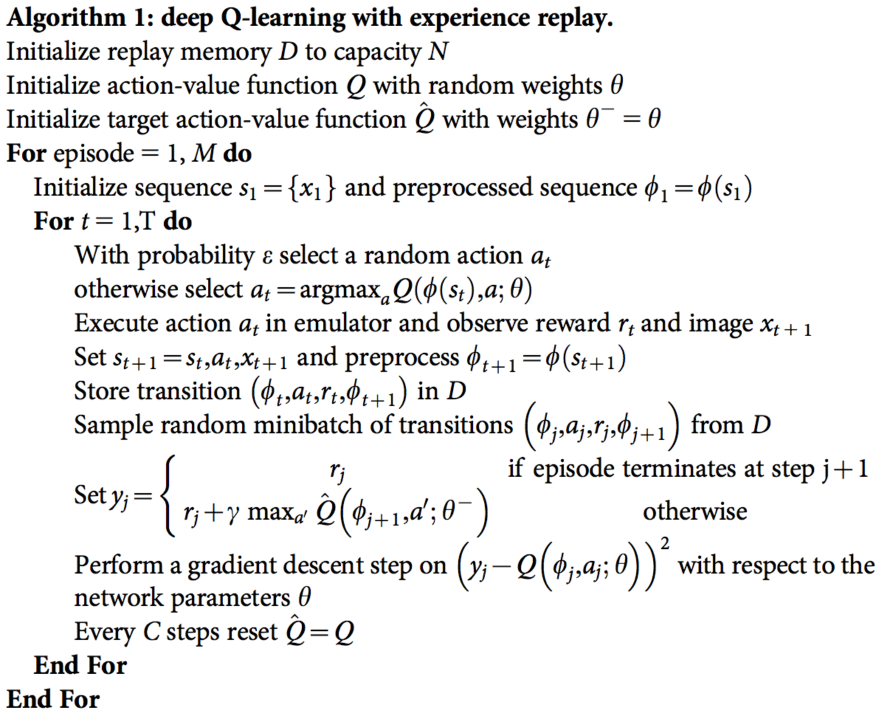


Figure 3: A visual representation of the network.

The Deep Q-learning algorithm is implemented in this project with several improvements. These are Experience Replay, Target Networks, and Huber Loss. These improvements allows the algorithm to converge faster and efficiently. These improvements are explained in the later sections.

The pseudocode for the DQN algorithm is shown below.



```
Algorithm 1: deep Q-learning with experience replay.
Initialize replay memory $D$ to capacity $N$
Initialize action-value function $Q$ with random weights $\theta$
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 $\varepsilon$ 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 $y_j = \begin{cases} r_j & \text{if episode terminates at step j+1} \\ r_j + \gamma \max_{a'} \hat{Q}\left(\phi_{j+1}, a'; \theta^{-}\right) & \text{otherwise} \end{cases}$
     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 $\theta$
     Every $C$ steps reset $\hat{Q} = Q$
  End For
End For
```

Figure 4: Pseudocode for Deep Q-Learning algorithm

2.4. Benchmark

For the benchmark, we will use the following scores achieved by various algorithms and human-level performance. The DQN model is expected to beat all these scores by a significant margin. The highest score in these benchmarks is that of a human. This score would be our threshold for measuring the agent's performance. The table below shows the result of various systems that played the game.

PLANNER	MODEL	SCORE
Random Policy	**N/A**	**1.2**
Human	**N/A**	**31**
Policy Gradients	**N/A**	**10.5**
MCTS Depth 10, 50 Samples	**Videoless**	**2.7**
MCTS Depth 10, 150 Samples	**Videoless**	**3.0**
Wt. MCTS Depth 10, 150 Samp.	**Videoless**	**6.1**

Table 1: Scores achieved by various methods for the game Breakout.

3. Methodology

3.1. Data Preprocessing

The DQN agent is given the pixel data of the game's screen as input.

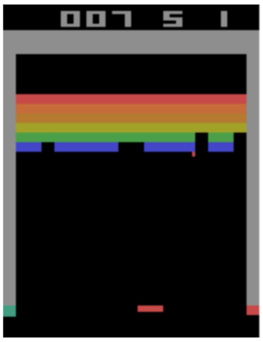


Figure 5: Single frame from Breakout

As explained earlier, the raw pixel data of the game's screen needs to be preprocessed first before supplied as input to the DQN model. The raw frames are preprocessed by first converting their RGB representation to a grey-scale representation, and then they are down-sampled to an 84×84 image.

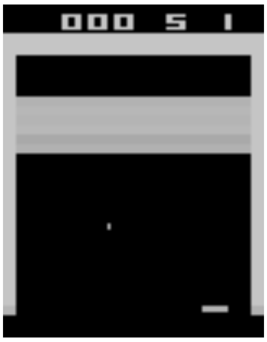


Figure 6: Single preprocessed frame from Breakout

The last four frames (before the current frame) are preprocessed and then stacked together to produce a tensor of shape (1, 84, 84, 4) which will be the input to the DQN model. This stacking of previous frames allows the model to infer the velocity of the ball and make better decisions while choosing actions.

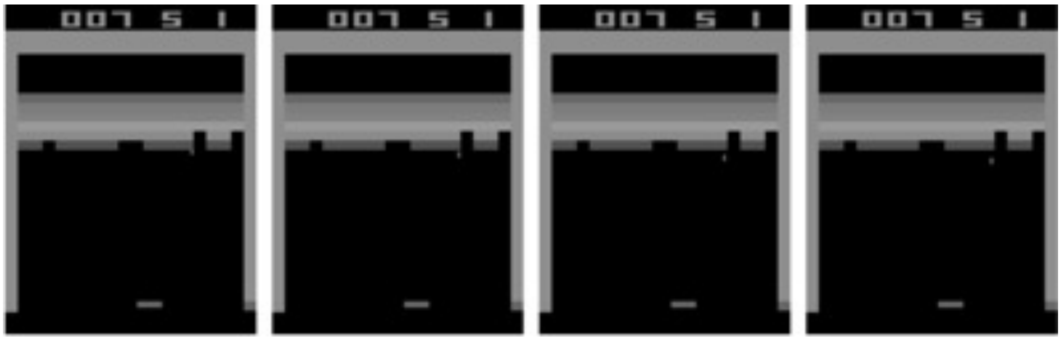


Figure 7: Four consecutive preprocessed frames from Breakout

This pre-processing step is done with the help of gym wrappers. Very frequently, we will want to extend the environment's functionality in some generic way. For example, if an environment gives us some observations, but we want to accumulate them in a buffer and provide to the agent the '*N*' last observations. This is a common scenario for dynamic computer games when one single frame is just not enough to get full information about the game state. To do so, we will make use of the gym's Wrapper class.

Another example is when we want to be able to crop or preprocess an image's pixels to make it more convenient for the model to recognize, or if we want to normalize the reward scores. There are many such situations which have the same structure: we would like to "wrap" the existing environment and add some extra logic doing something. The gym provides us with a useful framework for these situations, called a Wrapper class.

The Wrapper class inherits the Env class. Its constructor accepts the only argument: the instance of the Env class to be "wrapped". To add extra functionality, we need to redefine the methods we want to extend like step() or reset(). The only requirement is to call the original method of the superclass.

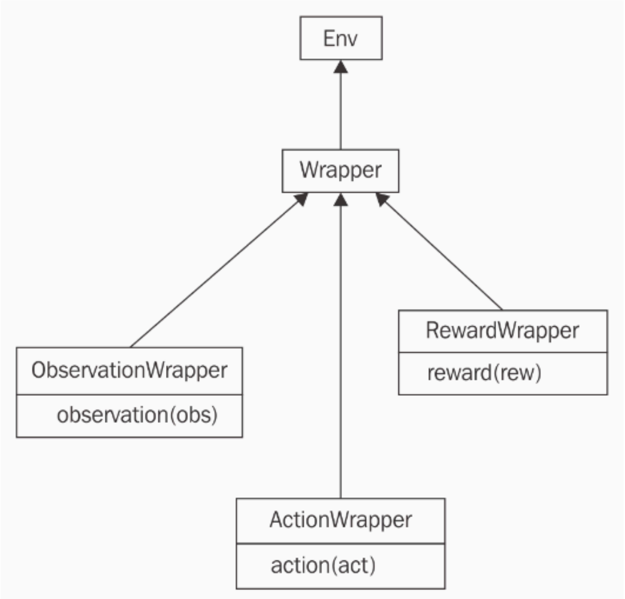


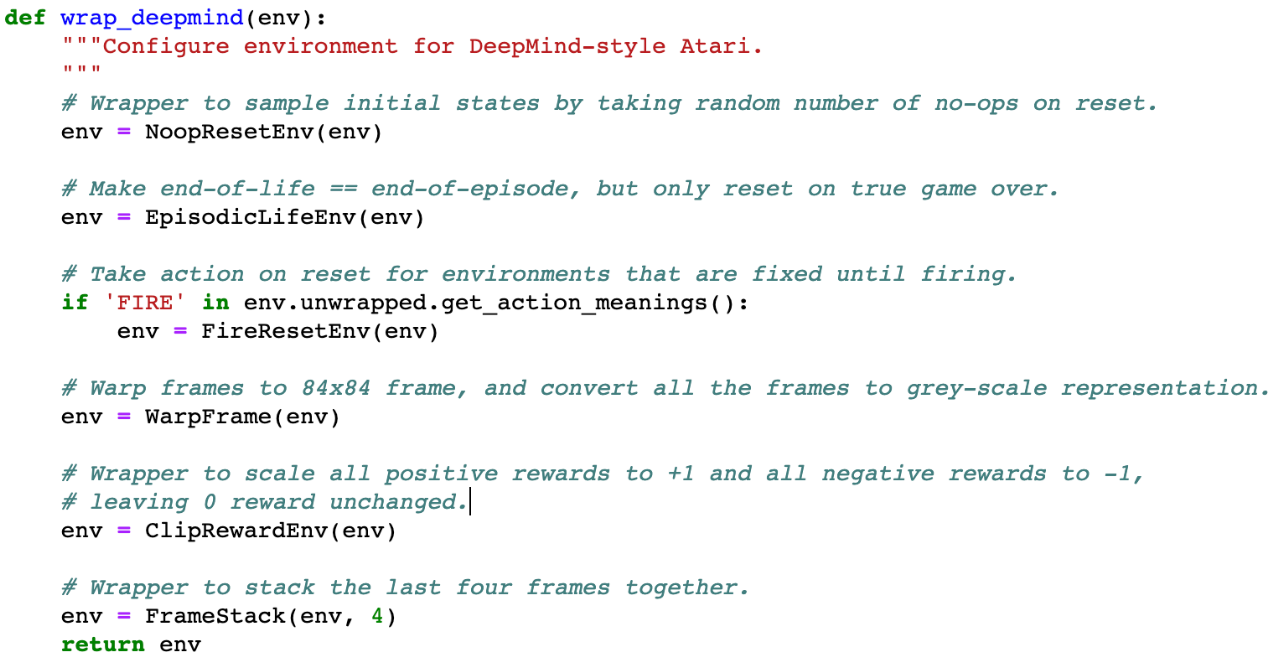
Figure 8: The hierarchy of Wrapper classes in Gym.

To handle more specific requirements, like a Wrapper which wants to process only observations from the environment, or only actions, there are subclasses of Wrapper which allow filtering of only a specific portion of information.

These are:

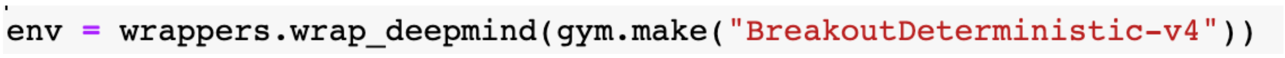
- ObservationWrapper: It allows to redefine the observation(obs) method. Argument obs is an observation from the wrapped environment, and this method should return the observation which will be given to the agent.
- RewardWrapper: It exposes the method reward(rew), which could modify the reward value given to the agent.
- ActionWrapper: It allows to override the method action(act) which could tweak the action passed to the wrapped environment to the agent.

The following wrappers are used to wrap the environment. Note that the original environment can be wrapped multiple times by different wrappers.



```
def wrap_deepmind(env):
    """Configure environment for DeepMind-style Atari.
    """
    # Wrapper to sample initial states by taking random number of no-ops on reset.
    env = NoopResetEnv(env)

    # Make end-of-life == end-of-episode, but only reset on true game over.
    env = EpisodicLifeEnv(env)

    # Take action on reset for environments that are fixed until firing.
    if 'FIRE' in env.unwrapped.get_action_meanings():
        env = FireResetEnv(env)

    # Warp frames to 84x84 frame, and convert all the frames to grey-scale representation.
    env = WarpFrame(env)

    # Wrapper to scale all positive rewards to +1 and all negative rewards to -1,
    # leaving 0 reward unchanged.
    env = ClipRewardEnv(env)

    # Wrapper to stack the last four frames together.
    env = FrameStack(env, 4)
    return env
```

Now to create the instance of the original environment wrapped with following wrappers, we just need to create **"wrap_deepmind"** object and pass the original environment as an argument to its constructor. This is shown below.



```
env = wrappers.wrap_deepmind(gym.make("BreakoutDeterministic-v4"))
```

All the preprocessing steps are done now, when the model calls the **env.reset()** or **env.step()** method, the environment will return the preprocessed frames to the model.

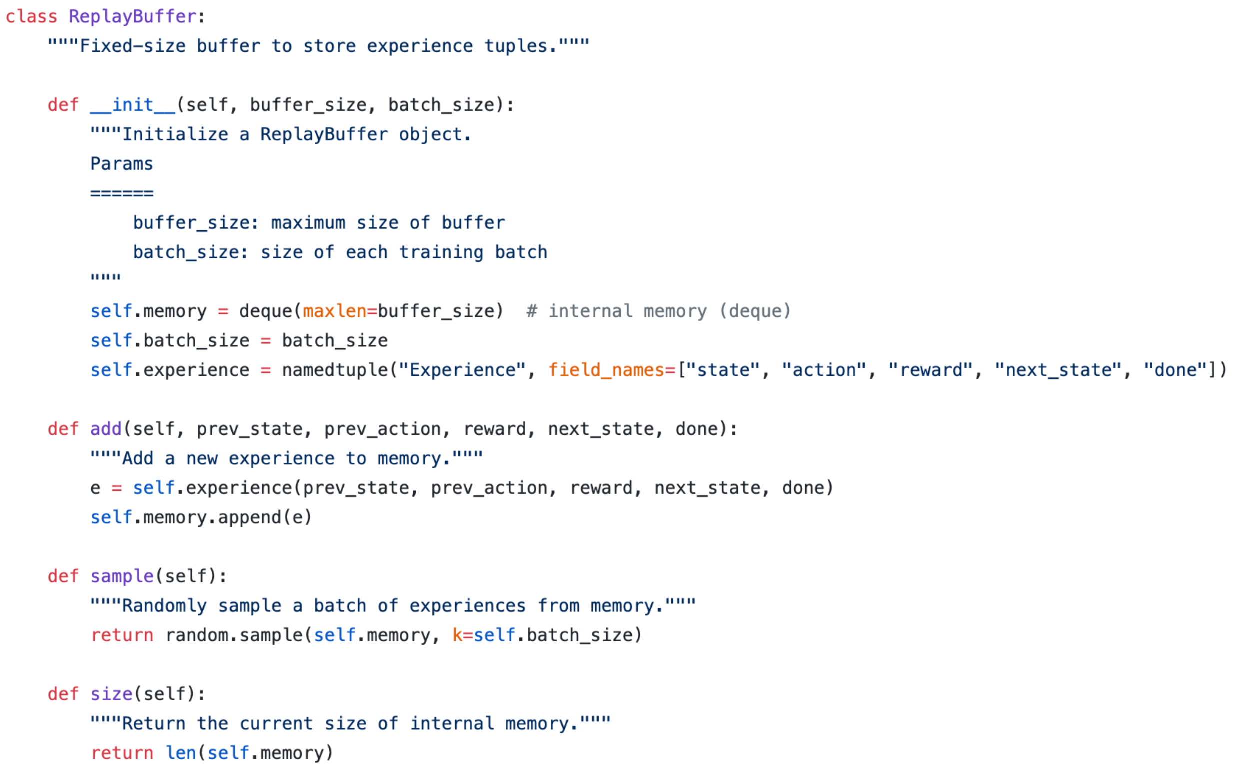
3.2. Implementation

To train the Deep Neural Network model, we will use the DQN algorithm. The Deep Learning model is expected to successfully learn control policies directly from high-dimensional sensory input using reinforcement learning. For this, we require convolutional layers that can extract information from the image data, and then we need fully connected layers to form relations between states and actions. The output layer is a fully-connected linear layer with a single output for each of the valid action. The output of the model will be a vector consisting of four values (for four valid actions) with each value corresponding to the probability of taking that action With the use of both convolutional and fully connected neural network layers, the model can estimate the policy function. We will use Deep Q-learning algorithm to train our neural network. Some essential features of the DQN algorithm are explained below.

3.2.1. Experience Replay

The neural network's weights will not be updated after every episode. We will use *Experience Replay* to store the various experiences encountered in the game.

Experience replay will store the observation tuple (state, action, reward, next state, done) that are seen during the gameplay, in a memory. After a fixed number of episodes, the model will sample a batch (of fixed size) of random experiences from the memory and use them to perform gradient descent and update step on the network. The reason why experience replay is helpful has to do with the fact that in reinforcement learning, successive states are highly similar. This means that there is a significant risk that the network will completely forget about what it's like to be in a state that it hasn't seen in a while. Replaying experience prevents this by still showing old frames to the network. It will also fix the issue of correlation between consecutive experiences, that can make the model incline towards choosing one kind of action. Hence, experience replay will ensure that the model has a stable learning rate and will prevent it from diverging. The implementation for replay buffer is shown below.



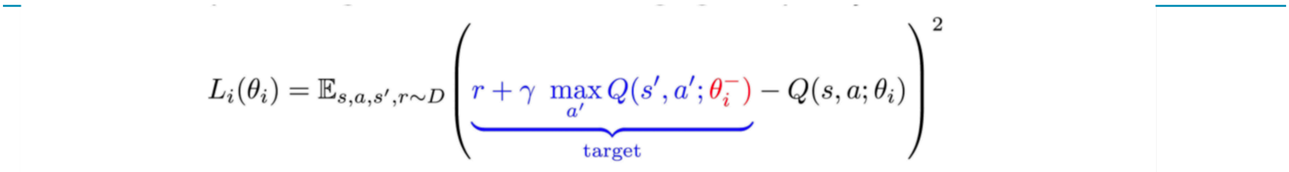
```python
class ReplayBuffer:
    """Fixed-size buffer to store experience tuples."""

    def __init__(self, buffer_size, batch_size):
        """Initialize a ReplayBuffer object.
        Params
        ======
            buffer_size: maximum size of buffer
            batch_size: size of each training batch
        """
        self.memory = deque(maxlen=buffer_size)  # internal memory (deque)
        self.batch_size = batch_size
        self.experience = namedtuple("Experience", field_names=["state", "action", "reward", "next_state", "done"])

    def add(self, prev_state, prev_action, reward, next_state, done):
        """Add a new experience to memory."""
        e = self.experience(prev_state, prev_action, reward, next_state, done)
        self.memory.append(e)

    def sample(self):
        """Randomly sample a batch of experiences from memory."""
        return random.sample(self.memory, k=self.batch_size)

    def size(self):
        """Return the current size of internal memory."""
        return len(self.memory)
```

3.2.2. Target Network

The Target network is another neural network with the same architecture as the main one. We update this network, after every n episodes. Recall, that the Q-values are estimated by the following formula:

$$Q(s, a) = r + \gamma \max_{a}'(Q(s', a'))$$

We use the target network to predict the targets for improving our model, more specifically, we use it to predict the future Q-values, i.e Q(s', a'). And use the main network for predicting the current Q-values. Therefore, the loss function is computed by taking the difference of the predictions of the two networks. This is shown below.



$$L_i(\theta_i) = \mathbb{E}_{s,a,s',r\sim D}\left(\underbrace{r + \gamma \max_{a'} Q(s', a'; \theta_i^-)}_{\text{target}} - Q(s, a; \theta_i)\right)^2$$

Here, θ_i^- is the target network and θ_i is the main network, we synchronise both the networks after every n episodes.

The use of target network makes the algorithm more stable compared to the standard Q-learning, where an update that increases $Q(s_t, a_t)$ often also increase $Q(s_{t+1}, a)$ for all a and hence also increases the target y_j, possibly leading to oscillations or divergence of the policy. Generating the targets using an older set of parameters adds a delay between two consecutive Q function updates. This assures that the target y_j is stable, making divergence or oscillations much more unlikely. Eventually, we will converge the two networks so that they are the same, but we want the network that we use to predict future Q-values to be more stable than the network that we actively update every single step.

The target network is defined along with the main model in the __**init**__ method of the agent.

```
# Main model
self.model = self.build_model()

# Target model
self.target_model = self.build_model()
# Set the weights of the target model to that of the main model.
self.target_model.set_weights(self.model.get_weights())
```

We use the target network to predict the future Q-values, as shown below.

```
next_states = np.array([transition[3][0] for transition in minibatch])
next_qs = self.target_model.predict(next_states)
```

Finally, the target network weights are updated and synchronized with the weights of the main network after every n-episodes.

```
# If terminal state is encountered, increase the update counter.
if terminal_state:
    self.target_update_counter += 1

# If counter reaches set value, update target network with weights of main network
if self.target_update_counter > self.target_update_frequency:
    self.target_model.set_weights(self.model.get_weights())
    self.target_update_counter = 0
```

The figure below shows the functionality of the target network.

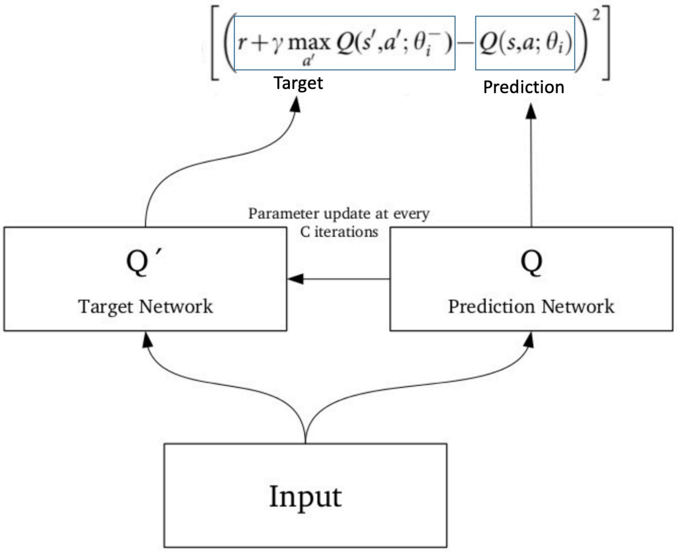

$\left[\left(r+\gamma \max_{a'} Q(s',a';\theta_i^-) - Q(s,a;\theta_i)\right)^2\right]$
Target
Prediction
Parameter update at every C iterations
Q´
Target Network
Q
Prediction Network
Input

Figure 9: Functionality of target network.

3.2.3. Huber Loss

We will use Huber Loss as a loss function for our Neural Network.

$$L_\delta(a) = \begin{cases} \frac{1}{2}a^2 & \text{for } |a| \le \delta, \\ \delta(|a| - \frac{1}{2}\delta), & \text{otherwise.} \end{cases}$$

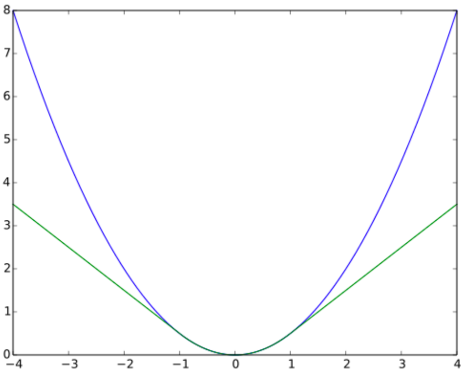


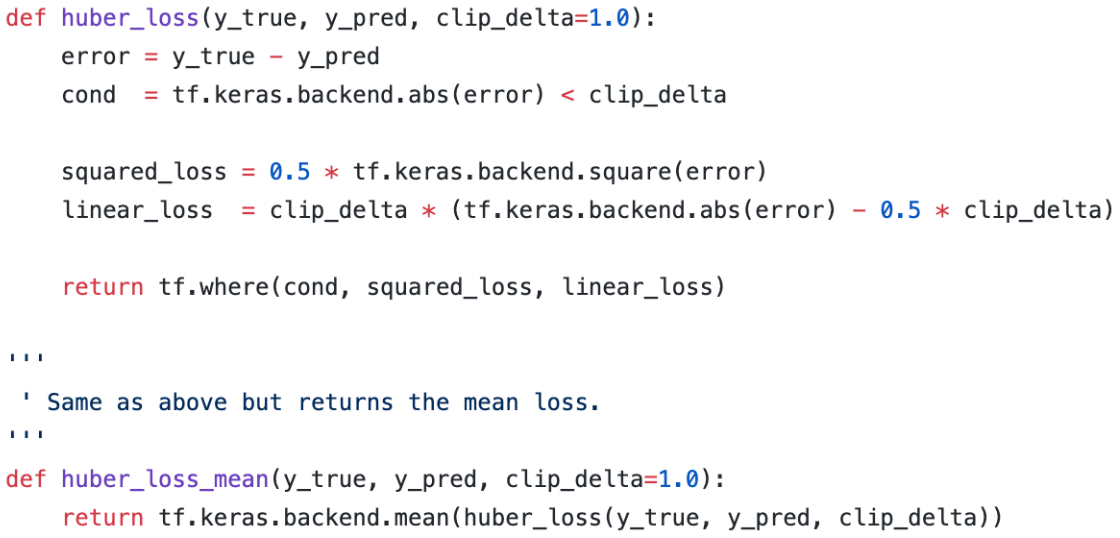
Figure 10: Green is the Huber loss and blue is the

This function is quadratic for small values of ‘a’, and linear for large values, with equal values and slopes of the different sections at the two points where $|a| = \delta$. The variable **a** often refers to the residuals, that is to the difference between the observed and predicted values, so the former expression can be expanded to:

$$L_\delta(y, f(x)) = \begin{cases} \frac{1}{2}(y - f(x))^2 & \text{for} |y - f(x)| \le \delta, \\ \delta\, |y - f(x)| - \frac{1}{2}\delta^2 & \text{otherwise.} \end{cases}$$

Here, **y - f (x)** is the difference between the true value and the predicted value. The introduction of Huber loss allows less dramatic changes which often hurt RL.

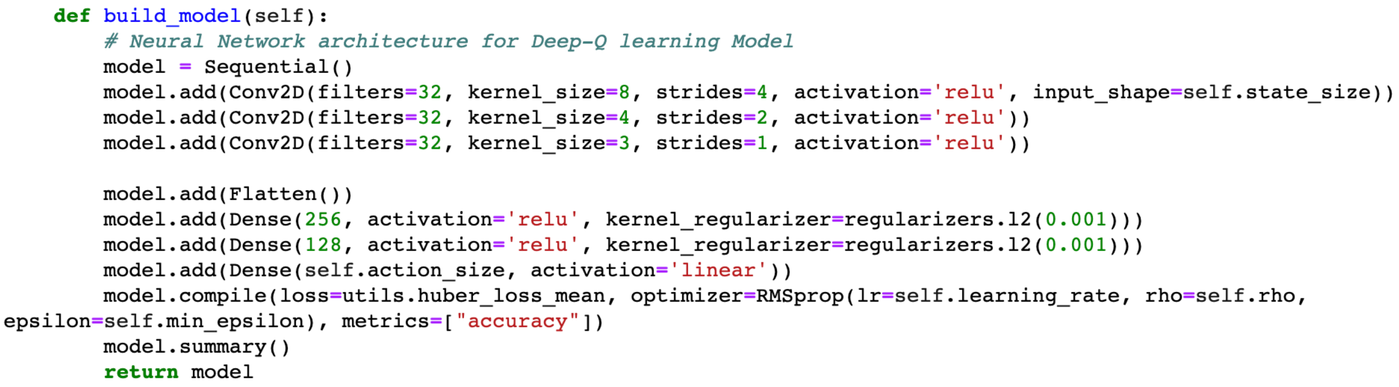
Its implementation is shown below.



```
def huber_loss(y_true, y_pred, clip_delta=1.0):
    error = y_true - y_pred
    cond  = tf.keras.backend.abs(error) < clip_delta

    squared_loss = 0.5 * tf.keras.backend.square(error)
    linear_loss  = clip_delta * (tf.keras.backend.abs(error) - 0.5 * clip_delta)

    return tf.where(cond, squared_loss, linear_loss)

'''
 ' Same as above but returns the mean loss.
'''
def huber_loss_mean(y_true, y_pred, clip_delta=1.0):
    return tf.keras.backend.mean(huber_loss(y_true, y_pred, clip_delta))
```

3.2.4 Network

The implementation of the neural network is shown below:



```
    def build_model(self):
        # Neural Network architecture for Deep-Q learning Model
        model = Sequential()
        model.add(Conv2D(filters=32, kernel_size=8, strides=4, activation='relu', input_shape=self.state_size))
        model.add(Conv2D(filters=32, kernel_size=4, strides=2, activation='relu'))
        model.add(Conv2D(filters=32, kernel_size=3, strides=1, activation='relu'))

        model.add(Flatten())
        model.add(Dense(256, activation='relu', kernel_regularizer=regularizers.l2(0.001)))
        model.add(Dense(128, activation='relu', kernel_regularizer=regularizers.l2(0.001)))
        model.add(Dense(self.action_size, activation='linear'))
        model.compile(loss=utils.huber_loss_mean, optimizer=RMSprop(lr=self.learning_rate, rho=self.rho,
epsilon=self.min_epsilon), metrics=["accuracy"])
        model.summary()
        return model
```

Here we have used 6 neural network layers:

1. ***Layer 1: Convolutional layer -> Filters = 32, Kernel size = 8, Strides = 4, Activation = Rectified Linear(relu)***
2. ***Layer 2: Convolutional layer -> Filters = 32, Kernel size = 4, Strides = 2, Activation = Rectified Linear(relu)***
3. ***Layer 3: Convolutional layer -> Filters = 32, Kernel size = 3, Strides = 1, Activation = Rectified Linear(relu)***
4. ***Layer 4: Dense -> Units = 256***
5. ***Layer 5: Dense -> Units = 128***
6. ***Layer 6: Dense -> Units = 4***

We have used L2 regularizer in the Dense layers. The final (last) layer has four outputs corresponding to the probability of taking each action.

The model is fit using the following code:

```
for index, (prev_state, prev_action, reward, next_state, done) in enumerate(minibatch):
    # Setting the target for the model to improve upon.
    if not done:
        target = reward + (self.gamma * np.max(next_qs[index]))
    else:
        target = reward

    new_q_value = prev_qs[index]
    new_q_value[prev_action] = target

    X.append(prev_state)
    y.append(new_q_value)

# Fit on all samples as one batch.
self.model.fit(np.vstack(X), np.vstack(y), batch_size=self.replay_buffer.batch_size,
               verbose=0, shuffle=False)
```

3.3. Refinement

Its been reported that the maximum score achieved in breakout peaks at 35 when using original Deep Q-Learning algorithm to train the Deep Neural Network. But with some improvements to the original algorithm, the score can be increased significantly, these improvements are implemented in the solution and defined below:

- ***Target Network:*** This ensures that the target for Q-learning update step is stationary and stable.
- ***Experience Replay:*** This fixes the issue of correlation between consecutive experiences.
- ***Huber Loss:*** Using Huber Loss as a loss function, allows less dramatic changes, which often hurt RL.

4.1. Results

4.1. Model Evaluation and Validation

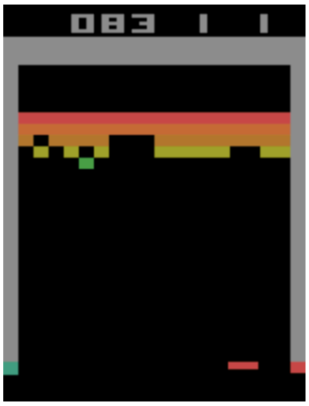
The following final hyper parameters are used to train the model.

Hyperparameter	Value	Description
Minibatch Size	32	Number of training cases over which each stochastic gradient descent (SGD) update is computed
Replay Memory Size	1000000	SGD updates are sampled from this number of most recent frames.
Agent History Length	4	The number of most recent frames experienced by the agent that are given as input to the Q network
Discount Factor	0.95	Discount factor gamma used in Q-learning update
Action Repeat	4	Repeat each action selected by the agent this many times. Using a value of 4 results in the agent seeing only every 4th input frame.
Update Frequency	4	The number of actions selected by the agent between successive SGD updates. Using a value of 4 results in agent selecting 4 actions between each pair of successive updates.
Learning Rate	0.00025	The learning rate used by RMS Prop.

Hyperparameter	Value	Description
Decay factor (rho)	0.95	"rho" is the decay factor or the exponentially weighted average over the square of the gradients.
Initial Epsilon	1	Initial value of epsilon in epsilon-greedy exploration.
Final Epsilon	0.01	Final value of epsilon in epsilon-greedy exploration.
Epsilon Decay	0.99995	Factor by which epsilon is decayed after each episode.
Replay Start Size	32	A uniform random policy is run for this number of frames before learning starts and the resulting experience is used to populate the replay memory.
No-op Max	30	Maximum number of "do nothing" actions to be performed by the agent at start of a new episode.
Target Update Frequency	15	Number of episodes after which the target network's weights are updated and synchronized with the weights of the main network.

The standard evaluation metric for Atari game playing is the score returned by the game itself. The performance of the agent is calculated by this score.

After 100+ hours of training, the agent is able to achieve a score of 83 in the Atari Breakout video game. This is certainly a good score. The screenshot of the final state of the game (before the game terminates) is shown below.



```
Score achieved by Intelligent agent:  83.0
```

Figure 11: Agent's performance

4.2. Justification

The agent achieved a score of 83 in the Breakout game. This is clearly much higher than the benchmarks previously specified. The human level performance is set to be the threshold value to be achieved by the agent, which is 31. The model clearly beats this score by a large margin. In fact, the agent achieved approximately 2.7 times higher score than the specified threshold.

Therefore, the final score is significant enough to have adequately solved the problem.

5. Conclusion

5.1. Free-Form Visualization

There were a total of 3 training sessions of the model. Rolling mean of the scores achieved during the multiple training session are visualized below. It is important to note here that the scores achieved by the agent are scaled. The score is calculated at the end of an episode and the episode is terminated when a single life/chance is lost. For example, a score of 5 means that the agent has successfully hit 5 bricks (regardless of their points) in a single life/chance.

- In the first training session, the model was trained for about 32k episodes. The model attained max average score of 3.6 (over 100 episodes) and max score of 14. The plot *scores vs episodes* is shown below. It can be seen that scores are increasing at a nice rate, as number of episodes increases.

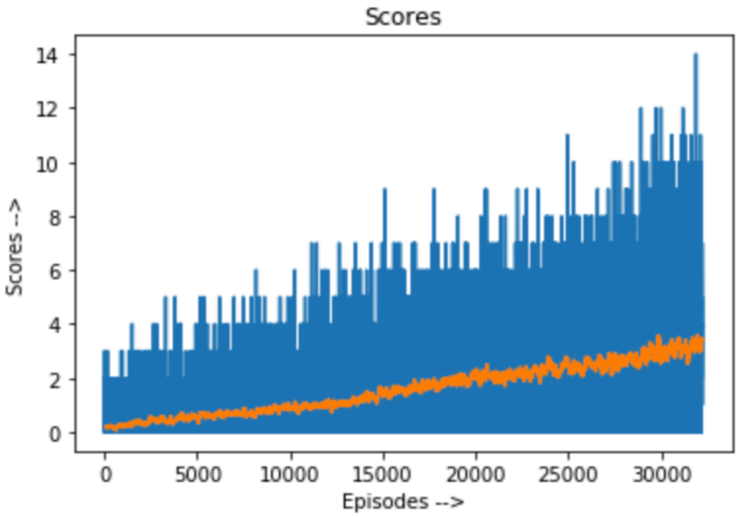


Figure 12: Scores (rolling mean) vs Episodes (1st training session)

- In the second training session, the model was trained for about 25k episodes. The agent attained a max average score of 6.89 (over 100 episodes) and max score of 30. This is almost double the score that was achieved in the first training session. The model has made a significant progress in this training session. The plot *scores vs episodes* is shown below. It can be seen that, at first the scores are increasing at a nice rate (as number of episodes increases) but at the end, the progress rate of the model seems to have slowed down.

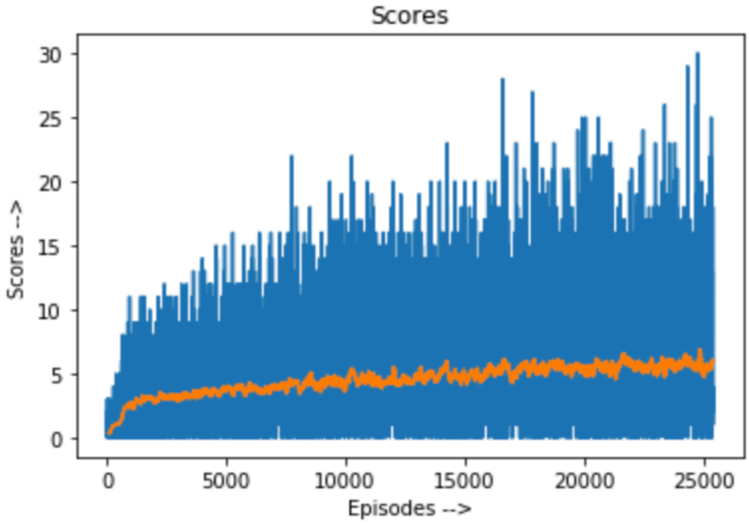


Figure 13: Scores (rolling mean) vs Episodes (2nd training session)

- In the third and the last training session, the model was trained for about 10k episodes. The model attained a max average score of 7.56 (over 100 episodes) and max score of 35. Here, only a slight improvement can be seen in contrast to the previous training sessions. The plot *scores vs episodes* is shown below. It seems that the algorithm has converged at this point. It is clear from the plot that the scores are not increasing further. Moreover, an increase in the agent's performance may be possible by resetting the epsilon (exploration rate) to a higher value and then training the model for more episodes. This may allow the agent to discover a special strategy (more about it in the *Improvement* section) to play this game, which may make it score 200+ points in the game!

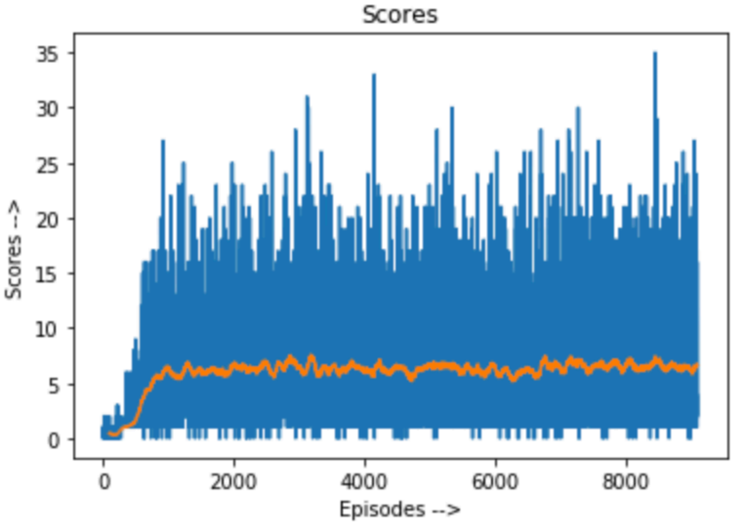


Figure 14: Scores (rolling mean) vs Episodes (3rd training session)

5.2. Reflection

In summary, with the use of Deep Q-leaning algorithm with experience replay and target network, the model is able to achieve a very good score in Atari Breakout video game. The basic preprocessing step is required for this problem, which mainly includes stacking and scaling of image frames. Also, to train the agent efficiently, *scaling of rewards* is necessary along with *frame skipping* and *episodic lives*. The model has clearly beaten all the thresholds specified, and performs better than an average human at this game.

The most important aspect of this project was the adoption of improvements to the DQN algorithm which includes: Experience replay, Huber loss, and Target Network. This allowed the agent's score to increase significantly.

The model was trained on my personal computer with tensorflow-gpu library. The training process was done in multiple sessions over a course of 5 days, using Nvidia GTX 1080 GPU and 16gb of ram. The model was trained for about 70k episodes in total. Following were the most challenging parts of the project:

- Implementation of the algorithm: The most difficult task that I faced was the implementation of the algorithm with target network and experience replay, there were a lot of trial and errors. But finally, I managed to make the program to work correctly.

- Neural network architecture: To come up with a good network architecture is very challenging, I tried many different architectures for this problem. At last, finally found a workable one.

- Identification of hyper-parameters: This was in fact very experiential problem, to come up with good values for hyper-parameters. This task required a lot of time and focus.

5.3. Improvement

Clearly, there is room for improvement for this solution. Google DeepMind's solution for this problem, achieved a much higher score. Wang et al. 2016 report a score of 418.5 (Double DQN) and 345.3 (Dueling DQN) for Breakout. This is much higher than my model's score. These new architectures: Double DQN and Dueling DQN are clearly better than my current implementation. Also, it is important to note that the DeepMind's solution network was trained for 10 days, while my network was trained for 3-4 days.

A higher score may be achieved by improving the hyper-parameters and amending the current deep neural network architecture. Increasing the training time will definitely help to achieve a higher score.

The best strategy to score high in this game is to dig a tunnel at one side by aiming the ball towards it, as shown below.

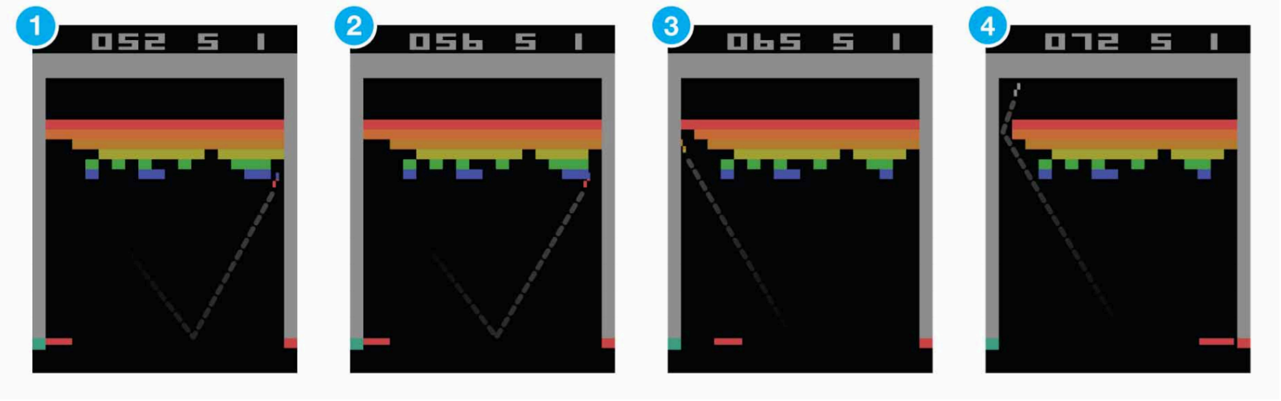


Figure 15: Efficient strategy to play Breakout.

In the DeepMind's solution it can be seen, that at one point the agent starts to aim directly at the sides to dig a tunnel and hit the blocks from above, this strategy makes the score to increase very rapidly.

In my solution, the agent did not discover this strategy yet. But with some tweaks and improvements which are discussed above. The agent may be able to exploit this strategy and perform even better!

6. References

- Justin Fu and Irving Hsu, Model-Based Reinforcement Learning for Playing Atari Games, http://cs231n.stanford.edu/reports/2016/pdfs/116_Report.pdf.
- Atari Breakout game manual, https://atariage.com/manual_html_page.php?SoftwareID=889.
- DeepMind, Human-level control through deep reinforcement learning, https://web.stanford.edu/class/psych209/Readings/MnihEtAlHassibis15NatureControlDeepRL.pdf
- Adrein Lucas Ecoffet, Beat Atari with Deep Reinforcement Learning, https://becominghuman.ai/lets-build-an-atari-ai-part-1-dqn-df57e8ff3b26
- https://en.wikipedia.org/wiki/Huber_loss
- https://hub.packtpub.com/openai-gym-environments-wrappers-and-monitors-tutorial/
- https://towardsdatascience.com/tutorial-double-deep-q-learning-with-dueling-network-architectures-4c1b3fb7f756
- https://towardsdatascience.com/reinforcement-learning-tutorial-part-3-basic-deep-q-learning-186164c3bf4