Applications of Deep RL in arcade game environments

Abstract

We use recent advances in training deep neural networks to develop a novel artificial agent, termed a deep Q-network, that can learn successful policies directly from high-dimensional sensory inputs using end-to-end reinforcement learning. We tested this agent on the challenging domain of classic Atari 2600 games. We demonstrated that the deep Q-network agent, receiving only the pixels and the game score as inputs, was able to surpass the performance of all previously discovered algorithms according to the research paper "Human-Level Control Through Deep Reinforcement Learning." As a result, we were able to achieve an average score level comparable to that of a professional human games tester in PingPong using the Pong-v0 Open AI gm, with the same algorithm, network architecture, and hyperparameters.

Challenges Faced:

Learning Python, Learning how to use google collab and GitHub properly(Initially we used to code locally and share it with other members) Understanding the way to convert a paper to code, reviewing multiple complex research papers, watching some Udemy/ YouTube tutorials to help learn the fundamentals of Deep Learning and also the dive from RL to Deep RL which enabled us to make and better the algorithms, choosing a suitable atari game.

Introduction

"**Reinforcement learning** is an area of machine learning concerned with how intelligent agents ought to take actions in an environment in order to maximize the notion of cumulative reward." -(Wikipedia). The agents learn by interacting with the environment. Agent encounters a state, decides an action to perform and receives a reward by performing the action, and moves to another state in the environment.

The input for the agent is the current state, and the agent maintains a Q-function based on which the agent makes decisions accordingly. So, the agent should initially identify the current state. If we provide numerical data, the agent can understand the state and perform an action accordingly. But in a real-world scenario, the agents get the **input through high-dimensional** sensory inputs, images, signals, etc., and the agent has to derive representations from these inputs. The agent has to identify the state, and after performing appropriate action, it has to update the Q-function values. To accomplish all this, we can use the “Deep Learning” techniques to directly understand the high-dimensional inputs and compute the Q-function values. “**Deep neural networks**” can be used to derive data from high-dimensional data such as images. If we know Q-values, we can run a Reinforcement Learning (RL) algorithm, which decides the actions. This combination of Deep Learning and Reinforcement learning is known as “***Deep Reinforcement Learning***.”

The research paper “Human-level control through deep reinforcement learning” has proposed a Deep reinforcement learning algorithm called the “**Deep Q Network (DQN)**,” where a deep neural network is used to learn policies directly from high-dimensional input. We will be using a “**Deep Convolutional network**,” which uses hierarchical layers of filters that derive useful correlations and data from the images. This network is used to find or approximate the action-value function or the Q-function Q(s,a). From the Q-function values, we can choose a greedy action that is the Q-function’s optimal value. The general equation of the optimal Q-value is shown below.

$$Q^*(s,a) = \max_{\pi} \mathbb{E}\left[r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \ldots \middle| s_t = s,\, a_t = a,\, \pi\right]$$

As we calculate the Q-values using a deep neural network, a nonlinear function approximator, the RL algorithm may become unstable as the network is stochastic. If we use this in Q-function, then the stochastic values can lead to wrong decisions as the decisions are very sensitive to the action-values of the Q-function. The RL agent may also diverge or learn wrong as the observations are correlated. To overcome this, the paper used two tricks. One is “experience replay,” which shuffles the observations and removes the correlations. Another is to maintain two networks. One network updates iteratively, and the other is just the copy of the first network but updated or copied periodically, which acts as a target network.

DQN

Deep Q-Network (DQN) is a deep reinforcement learning algorithm that uses deep neural networks to estimate Q-function. The neural network used is a deep convolutional network consisting of three convolution layers and two linear layers. ReLU activation function was used. The neural network takes high-dimensional input or, in simple words, the state representation as input and estimates the Q-function. The neural network computes the Q-values for all the possible actions from the input, which is the current state.

The input image needs to be preprocessed to remove flickering and reduce input dimensionality and some preprocessing required for the atari environment. After understanding the state, the actions are selected based on an e-greedy policy based on Q(s,a). The action is performed on the emulator, and we receive the output image, which is again preprocessed and stored as a state representative. To perform experience replay, we have to store all these experiences: the state, action, reward, and observation pairs. Experience e can be represented as $e_t=(s_t,a_t,r_t,s_{t+1})$. While training the agent, we randomly pick a sample of experiences and update the Q-function. By doing this, we remove the correlation between the experiences, and we can send a batch of experiences as training data for the deep neural networks.

In short, the **deep Q network calculates the action values Q(s,a,θ)** where θ ate the parameters of the network. The DQN maps the high-dimensional input to a vector of action values. If we consider the input dimension as ‘n’ and the number of actions is ‘m’. Then the DQN maps the n-dimensional input to m-dimensional output, which is the action-values for ‘m’ actions.

Neural network logic and estimation of Q-values:

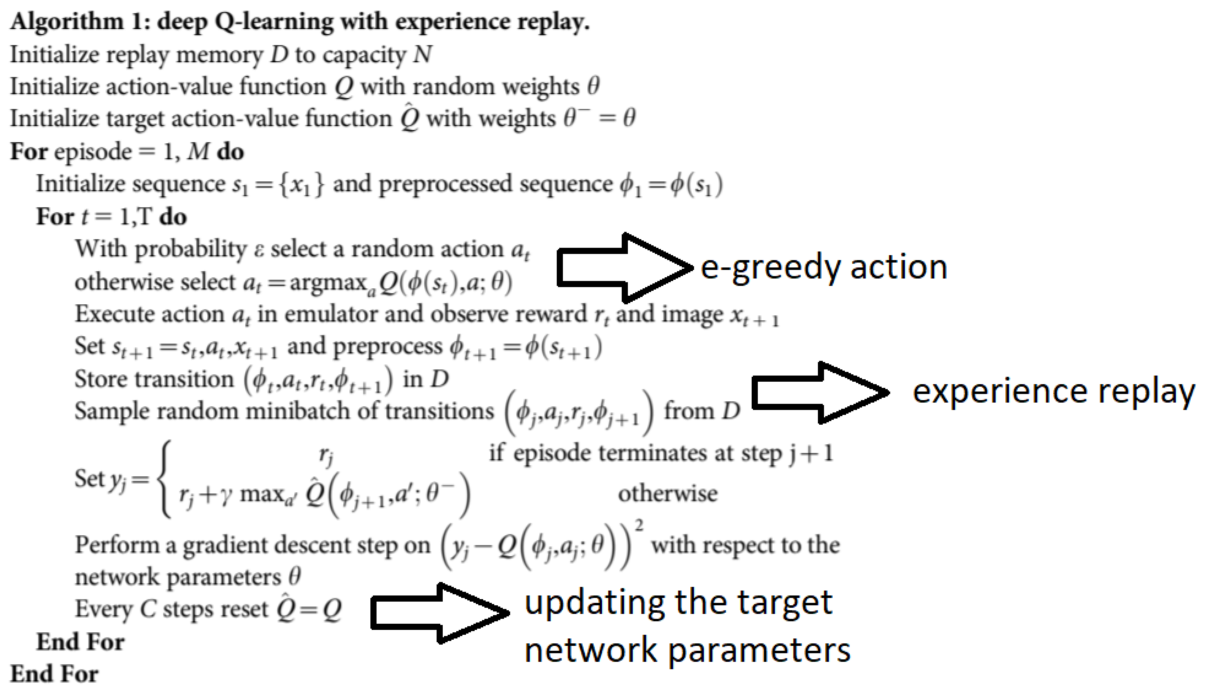
We take a sample of experiences and send them through the network and optimize the network by minimizing the loss. The loss function is the difference between the predicted Q-value and the target Q-value. The predicted Q-value is obtained by passing the state as input to the neural network, and the target Q-value is reward+(gamma*optimal Q-value). The loss function is shown in the figure below.

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

Where θ_i are the parameters of the Q-network at ith iteration and θ_i^- are the parameters of the second network or the target network.

We iteratively update θi values(weights) of the first network using the deep learning methods. And also, periodically updating the θi–, weights of the second neural network equal to the first one. This second neural network acts as a target network. Thus we optimize the target neural network to predict the actions based on the optimal Q-value. The network's final output can be considered the Q-values, the parameter for selecting the action.

Pseudocode of the algorithm:



```
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 $y_j = \begin{cases} r_j & \text{if episode terminates at step } j+1 \\ r_j + \gamma \max_{a'} \hat{Q}(\phi_{j+1}, a'; \theta^-) & \text{otherwise} \end{cases}$
        Perform a gradient descent step on $(y_j - Q(\phi_j, a_j; \theta))^2$ with respect to the
        network parameters $\theta$
        Every $C$ steps reset $\hat{Q} = Q$
    End For
End For
```

Implementation:

The implementation code - GitHub

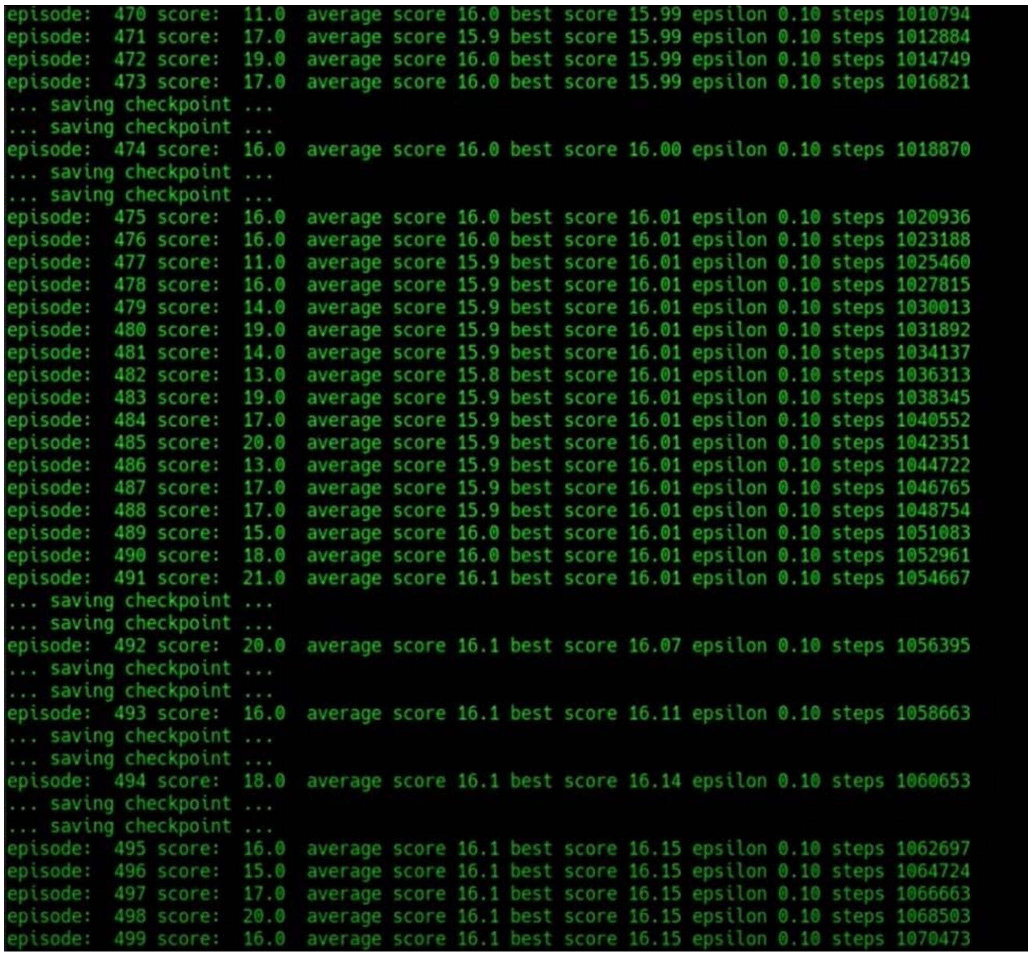
We have coded in python using PyTorch, *torch.nn* for implementing neural network, NumPy arrays to store the transitions/experiences to perform experience replay. We have done the preprocessing of the high-dimensional input. We have used the deep convolutional neural network for the neural network, and the policy used while training is an e-greedy policy. The transitions/experiences are stored in NumPy arrays to perform experience replay by taking a sample or a mini-batch of experiences for training the neural network.

Wrapper class and pseudocode for preprocessing

The screen images contain 3 channels and our agent only needs one channel to work with and we can deal with this by converting the images to grayscale. These images yet again are relatively large in size which makes the training slower given the hardware constraints so we deal with this by rescaling the images to 84x84 pixels. To deal with the constant flickering of the images we take the max of the previous 2 frames. This is all possible with the help of a wrapper class which allows us to make changes to the OpenAI gym and in turn, gives us a custom environment that is suitable for our needs. The wrapper class contains the *gym.ObservationWrapper* for preprocessing and stacking and the *gym.Wrapper* which is used for the step function.

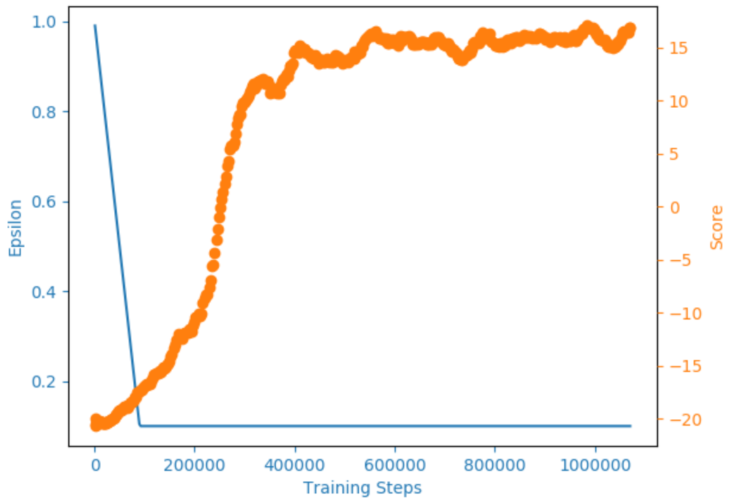
Results:

We have applied this algorithm to the pong game. We have seen that it produces a very good score after being trained for a good number of episodes/experiences. We have trained neural networks for a few days, and we have observed that the agent learns through playing and performs very well after training. It can outperform a professional player's score, which is remarkable for a trained agent to play just by seeing the images. We can see that at the end of 500 games, we have achieved an average score of 16.1 points with the best average score being around 16.15 points after running for about a million learning steps.



```
episode:  470 score:  11.0  average score 16.0 best score 15.99 epsilon 0.10 steps 1010794
episode:  471 score:  17.0  average score 15.9 best score 15.99 epsilon 0.10 steps 1012884
episode:  472 score:  19.0  average score 16.0 best score 15.99 epsilon 0.10 steps 1014749
episode:  473 score:  17.0  average score 16.0 best score 15.99 epsilon 0.10 steps 1016821
... saving checkpoint ...
... saving checkpoint ...
episode:  474 score:  16.0  average score 16.0 best score 16.00 epsilon 0.10 steps 1018870
... saving checkpoint ...
... saving checkpoint ...
episode:  475 score:  16.0  average score 16.0 best score 16.01 epsilon 0.10 steps 1020936
episode:  476 score:  16.0  average score 16.0 best score 16.01 epsilon 0.10 steps 1023188
episode:  477 score:  11.0  average score 15.9 best score 16.01 epsilon 0.10 steps 1025460
episode:  478 score:  16.0  average score 15.9 best score 16.01 epsilon 0.10 steps 1027815
episode:  479 score:  14.0  average score 15.9 best score 16.01 epsilon 0.10 steps 1030013
episode:  480 score:  19.0  average score 15.9 best score 16.01 epsilon 0.10 steps 1031892
episode:  481 score:  14.0  average score 15.9 best score 16.01 epsilon 0.10 steps 1034137
episode:  482 score:  13.0  average score 15.8 best score 16.01 epsilon 0.10 steps 1036313
episode:  483 score:  19.0  average score 15.9 best score 16.01 epsilon 0.10 steps 1038345
episode:  484 score:  17.0  average score 15.9 best score 16.01 epsilon 0.10 steps 1040552
episode:  485 score:  20.0  average score 15.9 best score 16.01 epsilon 0.10 steps 1042351
episode:  486 score:  13.0  average score 15.9 best score 16.01 epsilon 0.10 steps 1044722
episode:  487 score:  17.0  average score 15.9 best score 16.01 epsilon 0.10 steps 1046765
episode:  488 score:  17.0  average score 15.9 best score 16.01 epsilon 0.10 steps 1048754
episode:  489 score:  15.0  average score 16.0 best score 16.01 epsilon 0.10 steps 1051083
episode:  490 score:  18.0  average score 16.0 best score 16.01 epsilon 0.10 steps 1052961
episode:  491 score:  21.0  average score 16.1 best score 16.01 epsilon 0.10 steps 1054667
... saving checkpoint ...
... saving checkpoint ...
episode:  492 score:  20.0  average score 16.1 best score 16.07 epsilon 0.10 steps 1056395
... saving checkpoint ...
... saving checkpoint ...
episode:  493 score:  16.0  average score 16.1 best score 16.11 epsilon 0.10 steps 1058663
... saving checkpoint ...
... saving checkpoint ...
episode:  494 score:  18.0  average score 16.1 best score 16.14 epsilon 0.10 steps 1060653
... saving checkpoint ...
... saving checkpoint ...
episode:  495 score:  16.0  average score 16.1 best score 16.15 epsilon 0.10 steps 1062697
episode:  496 score:  15.0  average score 16.1 best score 16.15 epsilon 0.10 steps 1064724
episode:  497 score:  17.0  average score 16.1 best score 16.15 epsilon 0.10 steps 1066663
episode:  498 score:  20.0  average score 16.1 best score 16.15 epsilon 0.10 steps 1068503
episode:  499 score:  16.0  average score 16.1 best score 16.15 epsilon 0.10 steps 1070473
```

The plot of score vs. epoch is shown below. From the plot, we can see that the agent generally starts to learn as the epsilon decreases but the majority of the learning happens in the mostly greedy phase. By 500,000 steps the agent has achieved its top score with some oscillations around an average score of around 15.5-16.0. Hence, we have clear evidence of learning.



Improvement

Double Deep Q-Learning(DDQN)

The problem with DQN:

The above DQN algorithm, which combines Q-learning with a deep neural network, suffers from substantial overestimations in some games in the Atari 2600 domain. In basic Q-learning, the optimal policy of the Agent is always to choose the best action in any given state. The assumption behind the idea is that the best action has the maximum expected Q-value. However, the Agent knows nothing about the environment, in the beginning, it needs to estimate Q(s, a) at first and update them at each iteration. Such Q-values have lots of noise, and we are never sure whether the action with maximum expected/estimated Q-value is really the best one. Unfortunately, the best action often has smaller Q-values compared to the non-optimal actions in most cases. According to the optimal policy in basic Q-Learning, the Agent tends to take the non-optimal action in any given state only because it has the maximum Q-value. Such a problem is called the overestimation of action value (Q-value). The noisy estimated Q-values caused due to overestimation lead to large positive biases in the updating procedure of the Q-value. This is because the loss function discussed above depends on the current Q(s, a), which is very noisy.

Overoptimistic value estimates are not necessarily a problem in and of themselves. Suppose all values would be uniformly higher i.e., equally overestimated. In that case, the relative action preferences are preserved, and we would not expect the resulting policy to be any worse since these noises don't impact the difference between the Q(s', a) and Q(s, a).

The solution to overestimation - Double Deep Q-Learning:

The max operator in DQN(mentioned below) uses the same values both to select and to evaluate an action :

$$Y_t^{\mathrm{DQN}} \equiv R_{t+1} + \gamma \max_a Q(S_{t+1}, a; \boldsymbol{\theta}_t^-)$$

This makes it more likely to select overestimated values, resulting in overoptimistic value estimates. To prevent this, we can decouple the selection from the evaluation, which is the idea behind Double Q-learning.

In the original Double Q-learning algorithm, two value functions are learned by assigning each experience randomly to update one of the two value functions, such that there are two sets of weights, θ, and θ'. For each update, one set of weights is used to determine the greedy policy and the other to determine its value.

For a clear comparison, we can first untangle the selection and evaluation in Q-learning and rewrite its target as :

$$Y_t^{\mathrm{Q}} = R_{t+1} + \gamma Q(S_{t+1}, \underset{a}{\operatorname{argmax}}\, Q(S_{t+1}, a; \boldsymbol{\theta}_t); \boldsymbol{\theta}_t)$$

The Double Q-learning error can then be written as :

$$Y_t^{\mathrm{DoubleQ}} \equiv R_{t+1} + \gamma Q(S_{t+1}, \underset{a}{\operatorname{argmax}}\, Q(S_{t+1}, a; \boldsymbol{\theta}_t); \boldsymbol{\theta}'_t)$$

Notice that the selection of the action, in the argmax, is still due to the online weights θ_t. This means that, as in Q-learning, we are still estimating the value of the greedy policy according to the current values, as defined by θ_t. However, we use the second set of weights θ_t**'** to fairly evaluate the value of this policy. This second set of weights can be updated symmetrically by switching the roles of θ and θ_t.

The idea of Double Q-learning is to reduce overestimations by decomposing the max operation in the target into action selection and action evaluation. Although not fully decoupled, the target network in the DQN architecture provides a natural candidate for the second value function without introducing additional networks. However, we can evaluate the greedy policy according to the online network, but using the target network to estimate its value leads to the Double DQN algorithm. Its update is the same as for DQN, but replacing the target Y_t^{DQN} with

$$Y_t^{\mathrm{DoubleDQN}} \equiv R_{t+1} + \gamma Q(S_{t+1}, \underset{a}{\operatorname{argmax}}\, Q(S_{t+1}, a; \boldsymbol{\theta}_t), \boldsymbol{\theta}_t^{-})$$

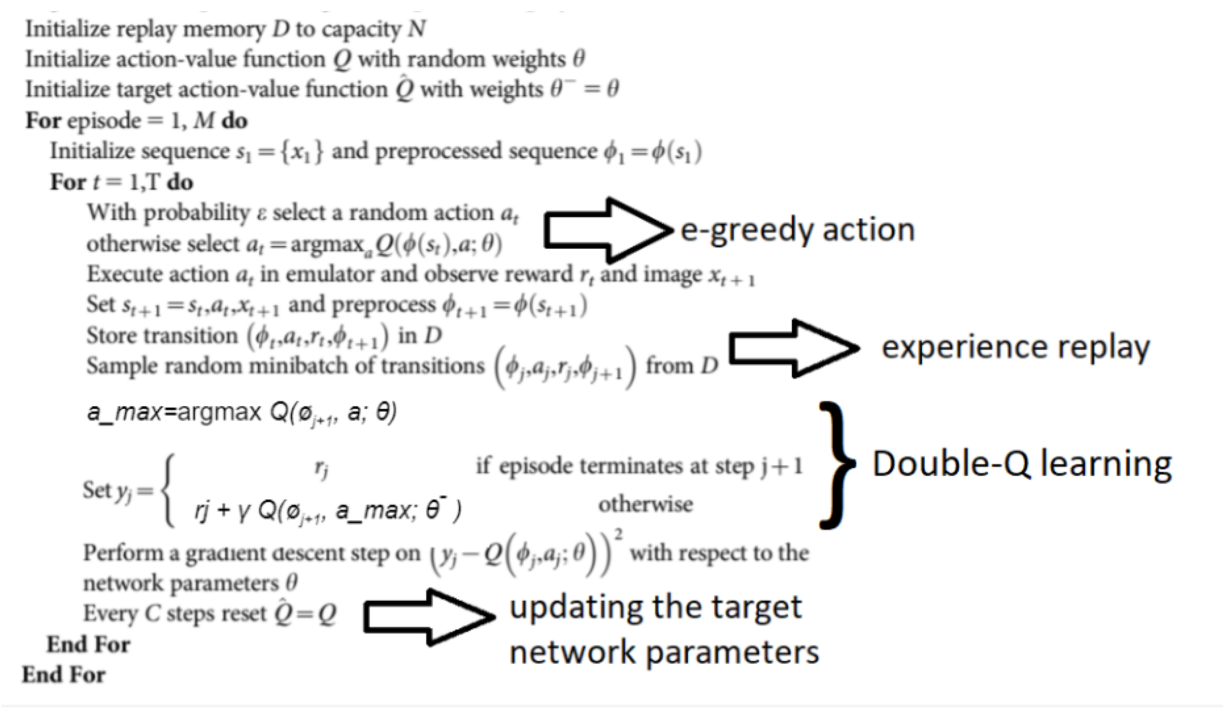
In comparison to Double Q-learning, the weights of the second network θ_t**'** are replaced with the weights of the target network θ_t^{-} for evaluating the current greedy policy. The update to the target network stays unchanged from DQN and remains a periodic copy of the online network. This version of Double DQN is perhaps the minimal possible change to DQN towards Double Q-learning. The goal is to get most of the benefit of Double Q-learning while keeping the rest of the DQN algorithm intact for a fair comparison, and with minimal computational overhead.

Implementation of DDQN:

The pre-processing step, deep neural network architecture, usage of experience replay, usage of two networks remain the same as the implementation of the DQN agent. The only change is in the calculation of the ‘y’ value or the expected reward. In DQN, the action which has maximum state-action value was selected, and ‘y’ was set as reward + gamma * Q(s,a,θ). So here, the action is chosen using the same network, and the return is calculated using the same. But in DDQN, the action is chosen based on the maximum action value of the first network (which we here considered as another network other than the target network). Still, the return is calculated using the target network. In DDQN, ‘y’ is equal to reward +(gamma * Q(s,a,θ^-)).

The network is trained similar to the DQN, where a sample is taken and performed training. But here, the expected return or the ‘y’ value is different, which is the ‘y’ based on the Double-Q learning algorithm. The loss function is the mean squared error of the target value ‘y’ and predicted value Q(s,a,θ).

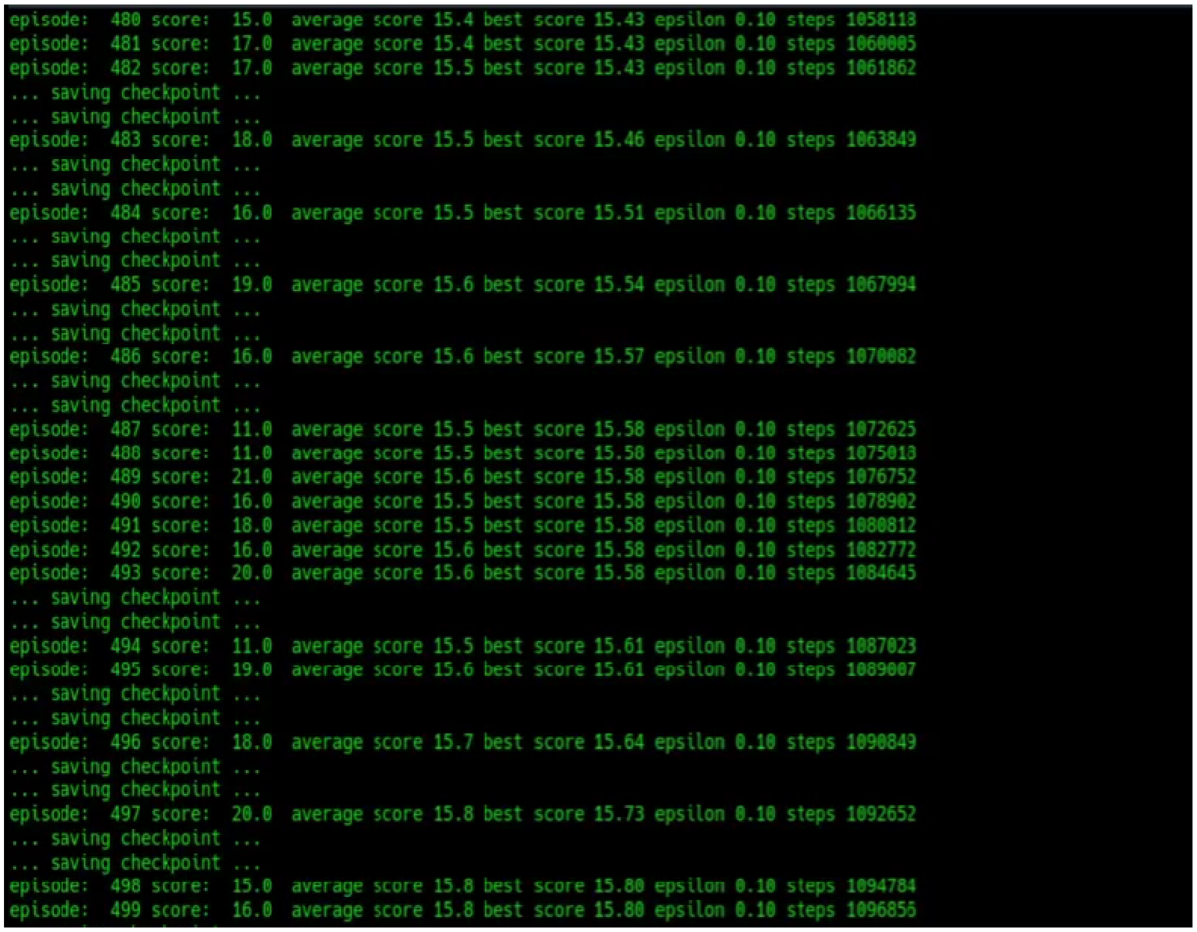
Pseudocode of the algorithm:



```
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 $a_t = \mathrm{argmax}_a Q(\phi(s_t), a; \theta)$
    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 $D$
    $a\_max = \mathrm{argmax}\ Q(\varnothing_{j+1}, a; \theta)$
    Set $y_j = \begin{cases} r_j & \text{if episode terminates at step } j+1 \\ rj + \gamma Q(\varnothing_{j+1}, a\_max; \theta^-) & \text{otherwise} \end{cases}$
    Perform a gradient descent step on $(y_j - Q(\phi_j, a_j; \theta))^2$ with respect to the
    network parameters $\theta$
    Every $C$ steps reset $\hat{Q} = Q$
  End For
End For
```

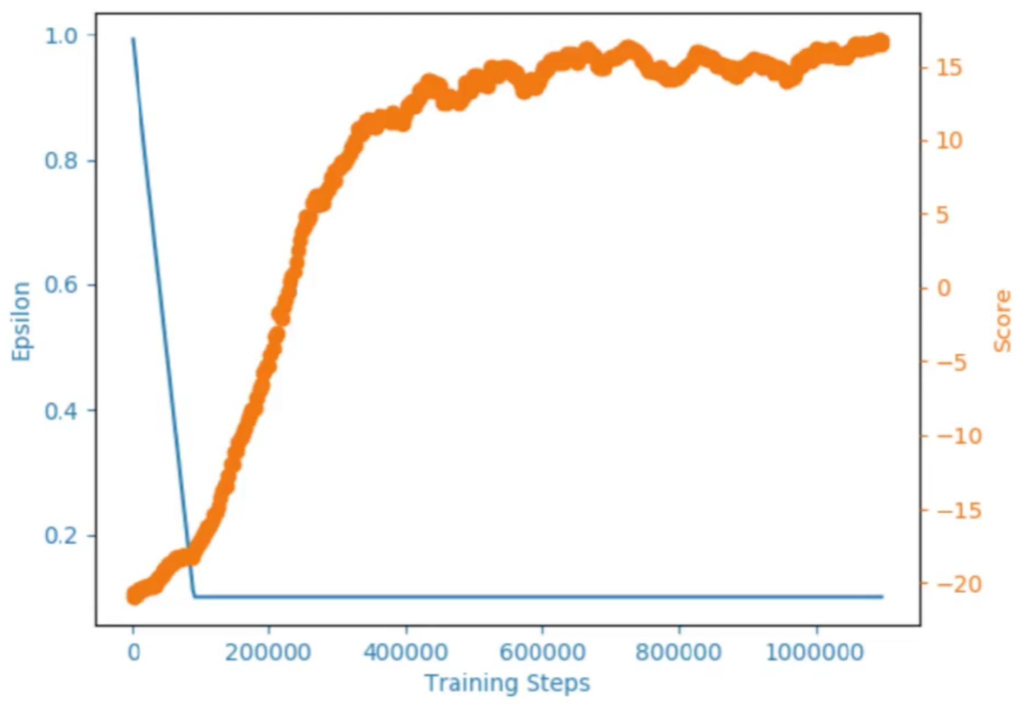
Results:

We have run the DDQN agent for 500 episodes, which run for around ten lakhs steps, and the time taken was around 24hrs. From the picture attached below, we see that we have achieved an **average score** of **15.8,** an **average best score** of **15.80** at the end of 500 episodes. We can observe that the average score of 15.8 is **slightly below** the average score of DQN Learning which was around **16.1**. This is normal since there is usually run-to-run variation -- we can get values such as 15,16,17. So, a delta of 0.3 between the two algorithms isn't very significant considering the run-to-run variation. However, we can notice that in episode 489 and in several episodes before, that the agent **does learn to beat the game** as it gets a **perfect score of 21**.



```
episode:  480 score:  15.0  average score 15.4 best score 15.43 epsilon 0.10 steps 1058118
episode:  481 score:  17.0  average score 15.4 best score 15.43 epsilon 0.10 steps 1060005
episode:  482 score:  17.0  average score 15.5 best score 15.43 epsilon 0.10 steps 1061862
... saving checkpoint ...
... saving checkpoint ...
episode:  483 score:  18.0  average score 15.5 best score 15.46 epsilon 0.10 steps 1063849
... saving checkpoint ...
... saving checkpoint ...
episode:  484 score:  16.0  average score 15.5 best score 15.51 epsilon 0.10 steps 1066135
... saving checkpoint ...
... saving checkpoint ...
episode:  485 score:  19.0  average score 15.6 best score 15.54 epsilon 0.10 steps 1067994
... saving checkpoint ...
... saving checkpoint ...
episode:  486 score:  16.0  average score 15.6 best score 15.57 epsilon 0.10 steps 1070082
... saving checkpoint ...
... saving checkpoint ...
episode:  487 score:  11.0  average score 15.5 best score 15.58 epsilon 0.10 steps 1072625
episode:  488 score:  11.0  average score 15.5 best score 15.58 epsilon 0.10 steps 1075018
episode:  489 score:  21.0  average score 15.6 best score 15.58 epsilon 0.10 steps 1076752
episode:  490 score:  16.0  average score 15.5 best score 15.58 epsilon 0.10 steps 1078902
episode:  491 score:  18.0  average score 15.5 best score 15.58 epsilon 0.10 steps 1080812
episode:  492 score:  16.0  average score 15.6 best score 15.58 epsilon 0.10 steps 1082772
episode:  493 score:  20.0  average score 15.6 best score 15.58 epsilon 0.10 steps 1084645
... saving checkpoint ...
... saving checkpoint ...
episode:  494 score:  11.0  average score 15.5 best score 15.61 epsilon 0.10 steps 1087023
episode:  495 score:  19.0  average score 15.6 best score 15.61 epsilon 0.10 steps 1089007
... saving checkpoint ...
... saving checkpoint ...
episode:  496 score:  18.0  average score 15.7 best score 15.64 epsilon 0.10 steps 1090849
... saving checkpoint ...
... saving checkpoint ...
episode:  497 score:  20.0  average score 15.8 best score 15.73 epsilon 0.10 steps 1092652
... saving checkpoint ...
... saving checkpoint ...
episode:  498 score:  15.0  average score 15.8 best score 15.80 epsilon 0.10 steps 1094784
episode:  499 score:  16.0  average score 15.8 best score 15.80 epsilon 0.10 steps 1096855
```

From the plot below, we observe that much like the case with DQN, the vast majority of learning occurs after the agent reduces its epsilon to the minimum value of 0.1. We can see that by 500,000 steps it has achieved **relatively maximum performance** where it is oscillating around its endpoint of around 15.2 and there are occasional spikes into the 15.8 regions as evidenced by the high score. Hence, we have clear signs of learning done by the agent.



Conclusion:

In arcade/atari game environments, the agent can only see the images which are high-dimensional vectors. The agent has to understand the state, choose appropriate action, perform an action, and understand the result state. So, the agents have to understand the high-dimensional image and calculate the Q-values for all possible actions. This is made possible by the use of deep neural networks, which take high-dimensional input like images and give the vector of Q-values for all possible actions. We can then choose appropriate actions based on the Q-values.

The agent learns by playing and training the neural network using these experiences. After rigorous training, the neural network can now accurately calculate the Q-values for new input, and the agent can perform the optimal action. This is a remarkable achievement as the agents themselves can learn through real-world inputs and process the Q-values for the selection of actions, and this has set the new state of the art in Deep Reinforcement Learning. The agent's performance is also high, with an average score of around 16.1, and sometimes even beat the game with 21 points.

The DQN agent had overcome a few challenges of using the non-linear neural network approximators using experience replay and usage of two networks. But, there were a few problems in DQN, which were overestimating the Q-values and initial noise. We came up with the idea of using Double Q-learning instead of Q-learning to overcome these problems. Learning is much more stable with Double DQN, suggesting that the cause for these instabilities is in fact, Q-learning's overoptimism. The Double DQN agent performed better than the DQN agent. Double DQN does not just produce more accurate value estimates but also better policies.

Contribution of group members:

Group number**: “**RL15”

Name	***ID No***	***Contribution***
Abdul Azeem Shaik	2019AAPS1234H	Implemented the improvement idea and ran both codes. Coded preprocessing and DDQN agent.
K Venkat Kedarnath	2019A7PS0155H	Implementation of the paper and searched for an improvement. Wrote report.
Sai Tushar Bandaru	2019A7PS0046H	Implementation of the paper and searched for an improvement. Wrote report.
Aryaman Krishna Velampalli	2019A7PS0140H	Wrote code for experience replay, and DDQN agent. Explained the implementation (DQN) and improvement (DDQN) in the report
Ravi Teja Venigalla	2019AAPS0232H	Coded DQN agent and main function. Collected research papers