MASTERING ATARI USING DQN

Tanul Gupta	Abir Mehta	Gagan Jain	Parth Sastry
180020110	180100005	180100043	180260026

1) Abstract

Through this project, we aim to use a combination of Reinforcement Learning and Hierarchical Sensory Processing for mastering game play for a simple Atari game called Breakout. We aim to develop an agent that has access to the raw real-time pixels (using Digital Image Processing) and a valid set of moves of the game along with the score that it has to maximize. Deep Convolutional Networks will be used for the same(to approximate the optimal action-value function) and will be integrated with Reinforcement Learning. We will also be using Replay Algorithm which involves storage and representation of recently experienced transitions. We will use the Double Deep Q-learning algorithm along with Experience Replay to train the agent.

2) Methods and Approaches

One approach to this project can be to approximate the Q-value of each state-action pair using a deep convolutional NN. The network can either take a state s, and an action a, as input and output the value Q(s,a) or take the state s, as input and return the Q-value of all possible actions (if the action space is discrete).In Deep Q-learning, the MSE loss function to be minimised is -

$$\mathcal{L}(\theta) = \mathbb{E}_{\pi}[(r_t + \gamma \max_{a'} Q_\theta(s', a') - Q_\theta(s, a))^2]$$

However this approach has some problems, namely overfitting and non stationarity of the target. The definition of the loss function is an expectation operator, which implies that the samples should be i.i.d to give the full picture. However, when gathering transitions in real-time, the samples are correlated, (s_t, a_t, r_{t+1}, s_{t+1}) is followed by (s_{t+1}, a_{t+1}, r_{t+2}, s_{t+2}).With correlated inputs/outputs, the Deep NN will overfit, and fall into a local minimum.
The second problem is the non-stationarity of the target. We're approximating both, the value of a state, and the return of the state via the same network, and performing regression on those two things. With non-stationarity of the target, deep NN performs poorly, as they try to iteratively converge towards the target value, but which is also moving.

Utilising a Deep Q Network:
Instead of successively storing transitions into the minibatch and then training the NN on it, transitions (or frames) are stored in a huge buffer, called the Experience Replay Memory, and transitions are randomly sampled from it to train the NN via SGD. To solve the non-stationarity of the target, instead of computing the expected return via the current parameters of the NN, they are computed via some old parameters of the NN, which are only updated after every so-and-so epochs. This way, the target doesn't change often, and the problem is more

stationary. This however, adds to the complexity, since millions of transitions are needed to obtain a good policy.

The idea with a ***double DQN*** is that the target y = r(s,a,s') + $\gamma \max_{a'} Q_{\theta'}(s',a')$ is frequently over-estimating the true return because of the max operator. At the beginning, if the target is overestimated of a state-action pair, the learned Q-value $Q_\theta(s,a)$ will also overestimate, and this will propagate to all successive actions. Van Hasselt (2015) showed that this over-estimation is inevitable in regular Q-learning and proposed the Double DQN. The idea is to train two networks, one to find the greedy action (with maximal Q-value) and the other to estimate the Q-value of the current state-action pair. Even if the first network overestimates the value of a state-action pair, the second might provide a better value.

$$Q^*(s_t, a_t) \approx r_t + \gamma\ Q(s_{t+1}, \text{argmax}_{a'} Q'(s_t, a_t))$$

$$\theta' \leftarrow \tau\theta + (1-\tau)\theta'$$

<u>The following is the pseudocode for the Double DQN Network:</u>

Initialize replay memory D to capacity N
Initialize action-value function Q with random weights θ — Main Network
Initialize target action-value function $\hat{Q}$ with weights $\theta^- = \theta$ — Target Network
For episode = 1, M **do**
  Initialize sequence $s_1 = \{x_1\}$ and preprocessed sequence $\phi_1 = \phi(s_1)$ — Initial observation for the main network
  For t = 1,T **do**
    With probability ε select a random action a_t
    otherwise select $a_t = \text{argmax}_a Q(\phi(s_t), a; \theta)$ — Choose an action. With probability ε, this action is random. Otherwise, it is the action with the highest Q-value.
    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 $y_j = \begin{cases} r_j & \text{if episode terminates at step } j+1 \\ r_j + \gamma \max_{a'} \hat{Q}(\phi_{j+1}, a'; \theta^-) & \text{otherwise} \end{cases}$ — Estimate target Q (y_j) with the Bellman equation
    Perform a gradient descent step on $(y_j - Q(\phi_j, a_j; \theta))^2$ with respect to the network parameters θ — Loss
    Every C steps reset $\hat{Q} = Q$ — Update the target network
  End For
End For

3) The Implementation:

<u>Breakout Environment:</u>

The game Breakout has a deterministic environment wrapped in the gym package, which enables users to conveniently interact with the environment. The state is to be extracted by using the (210, 210, 3) dimensional image provided by the gym environment. Further, the action space consists of 4 actions: Do nothing, Fire the ball to start the game, Move Left, Move right.

The breakout environment provides frames of the gameplay to the agent that are high dimensional and thus directly using them as inputs in the deep Q-learning network will require heavy computation. Instead, we utilize some simple image processing techniques to reduce this computational cost, while also ensuring that the representation still contains all the important features crucial for learning.

The primary processing which reduces the computational cost is the reduction of image size from (210,210,3) to (84,84,3). Moreover, this is converted to a grayscale image from an RGB image, thus reducing the number of channels to 1. So, the final input image to the network has a size of (84,84).

Agent and Environment:

Our first step was to use opencv to create a function to process the frames, which we described above. Following that, we created a wrapper for the environment and the game, to enable us to first process the frame and then feed it into the replay buffer.

The wrapper around the game, makes sure that the agent doesn't sit around doing nothing. If the agent loses a life, we force it to start a new game. The agent has functions to decide which action to take, and functions to 'learn', via sampling the batch (replay buffer) and using it to improve the target and train DQNs.

Sampling From the Buffer:

We implemented the idea of prioritized experience replay (Schaul et. al 2015) to order the transitions in the ERM in decreasing order of their 'error'

$$\delta = r(s, a, s') + \gamma Q_{\theta'}\left(s', \text{argmax}_{a'} Q_{\theta}(s', a')\right) - Q_{\theta}(s, a)$$

Those samples with a higher error are more likely (with a PRIORITY_SCALE variable value of 0.7) to be sampled, to train the model. We thus prioritize reducing the error for all samples, thereby ensuring that we don't get stuck because of bad sampling of the ERM.

Even though this improves the performance of the model, for a simple game like Breakout, the increased computational cost isn't worth the performance added, and thus, we didn't use the prioritised experience replay, even though it's part of the code.

Training the model:

The value of an action depends on two factors - the value of the state s in which we take that action, and how much that action is better than the other possible actions.

This leads to the definition of an advantage A(s,a), which is Q(s,a) - V(s). Instead of training for the expected return Q(s,a), we use a duelling DQN to train for the value of a state V(s) and the advantage A(s,a). The advantage has a lower variance than Q(s,a) which is better for training the NN.

The only 'forced' action we make via the agent is that the agent must 'fire' to start the game, if it loses a life. This is to ensure that the game continues. Apart from that, everything the agent does is to maximise the expected return.

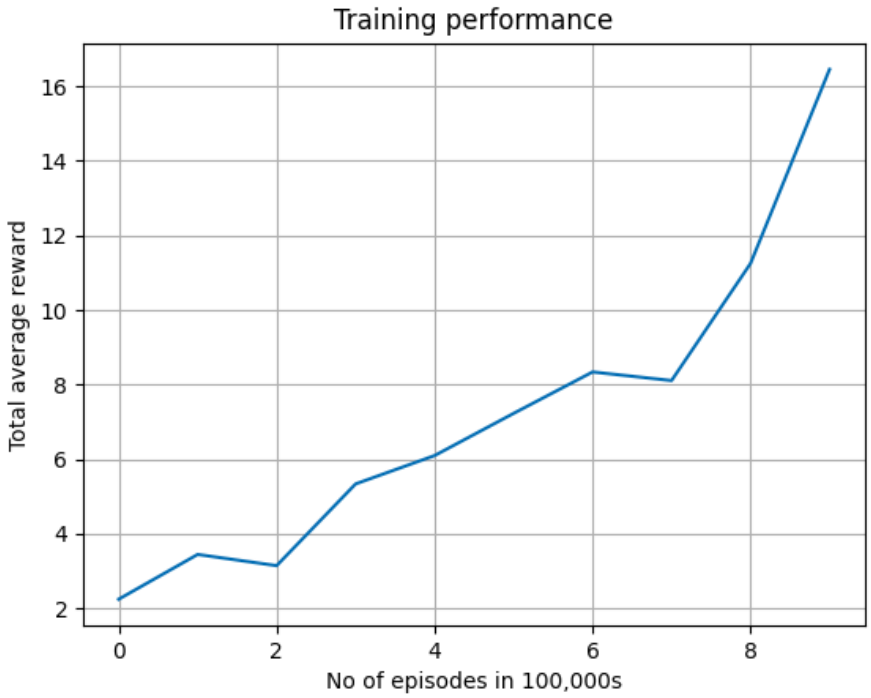
Visualization and Evaluation:

The visualisation of the Network performance, that we mentioned above, was taken from a GitHub repo - https://github.com/EvolvedSquid/tutorials/tree/master/dqn

We used the evaluation metric and the visualisation features given in the visualize.py file, to look at the 'attention heatmap' of the model, view the game screen and visualize the expected value over time as perceived by the model.

4) Training Results

The training reward vs number of episodes have been plotted below.



A setup for detailed visualisation has been coded in the repository. By running the file visualize.py, the heatmap along with the agent performance visualization and the Value function real time updates can be seen. It can be seen over here that the agent clearly learns to perform better and better and the learning hasn't quite peaked yet. A bit of online reading got us to know that actually for getting to an impeccable performance which corresponds to getting an average reward of around 300, the model needs to be trained for approximately 30,000,000 iterations while we did not have the computational power needed to do this very fast. We have been able to train it to 1,000,000 iterations and still the model can be seen to perform to a decent level (This can be seen by running the file evaluation.py)