# **DEEP REINFORCEMENT LEARNING MODEL - DDQN**

**Contents**

|  |
| --- |
| **Heading** |
| Introduction |
| Step 1 – Import an OpenAI Gym Game |
| Step 2 – Creating a Network |
| Step 3 – Connection of the Game to the Network |
| Step 4 – Deep Reinforcement Learning Model |
| Step 5 – Experimental Results |

**Introduction**

I chose to implement a deep reinforcement learning model for the Atari video game Space Invaders in the OpenAI Gym, implementing a form of Double Deep Q Networks (2 Deep Q Networks including convolutional neural networks). Refer step 3 to know more about the game (its actions, rewards etc.)

Initialization

On running main.py, a game object is instantiated which loads the network (refer step 1), initializes certain training and preprocessing hyperparameters, creates a DDQN object for the neural network and based on the user input, it either trains the agent or simulates the game on a pre-trained agent.

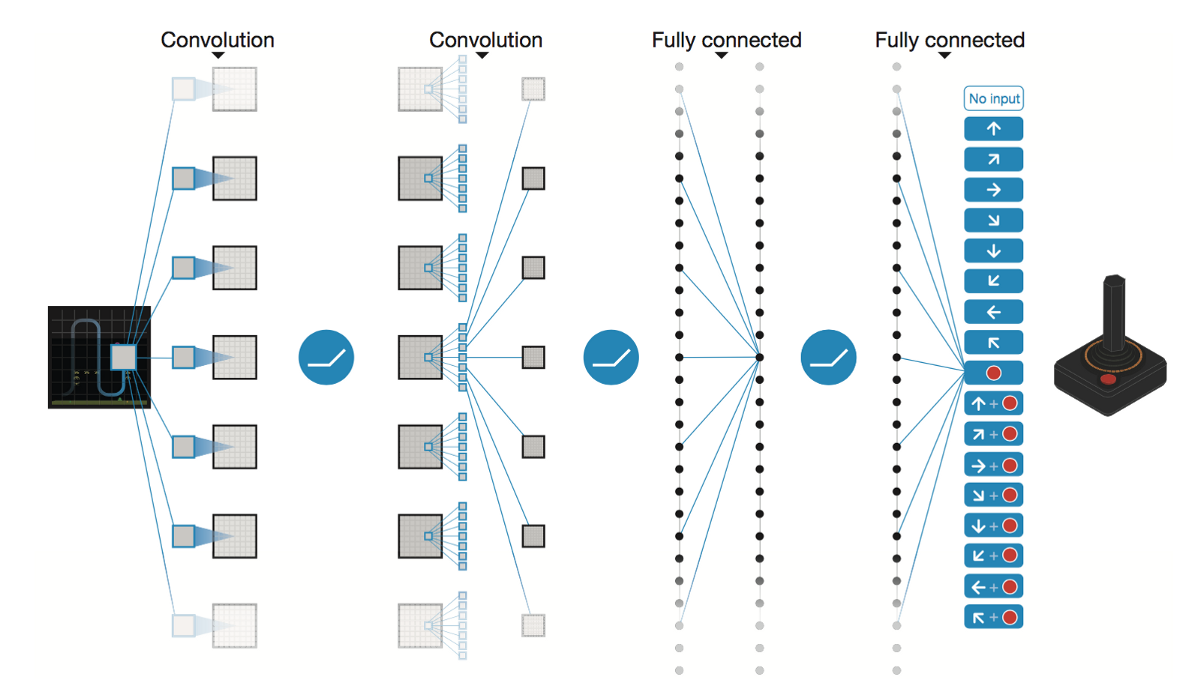
The DDQN class contains various hyperparameters for the model, q-learning architecture, memory/experience buffer and exploration. All of this is explained in \_\_init\_\_(). It then constructs the architecture (refer to step 2).

**Step 1 – Import an OpenAI Gym Game**

The code for this can be found in the game.py file’s createGame() method. The game environment is set up using the gym.make() command with ‘SpaceInvaders-v0’ as the parameter. And the environment is set to the initial state (when an episode begins).

**Step 2 – Creating a Network**

The \_\_init\_\_() method of DQN class (in model.py) calls constructArchitecture() method, that implements the following CNN (Convolutional Neural Network) architecture in Keras, as described in DeepMind’s paper [http://files.davidqiu.com//research/nature14236.pdf]:



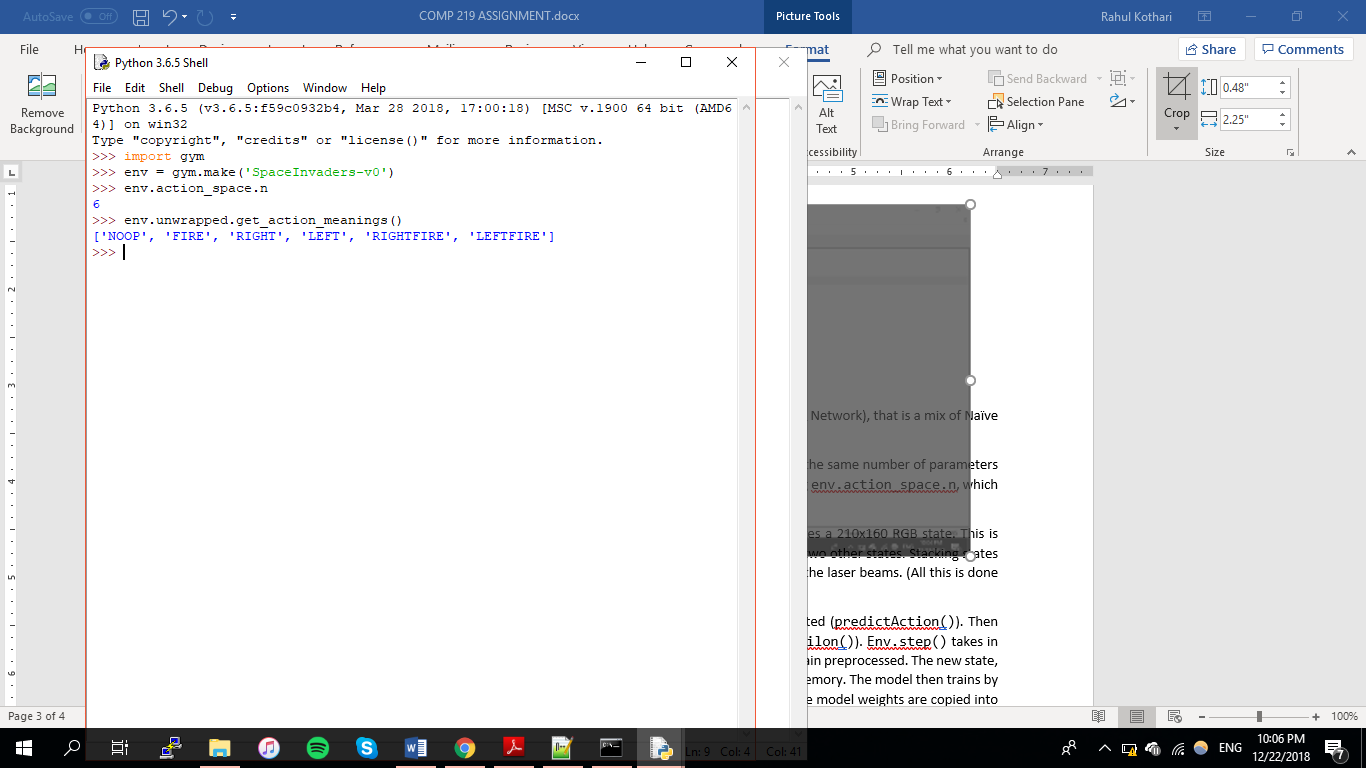
“The input to the neural network is an 84 x 84 x 3 image produced by the preprocessing methods. The first hidden layer convolves 32 filters of 8 x 8 with stride 4 with the input image and applies a rectifier nonlinearity. The second hidden layer convolves 64 filters of 4 x 4 with stride 2, again followed by a rectifier nonlinearity. This is followed by a third convolutional layer that convolves 64 filters of 3 x 3 with stride 1 followed by a rectifier. The final hidden layer is fully-connected and consists of 512 rectifier units. The output layer is a fully-connected linear layer with a single output for each valid action. The number of valid actions varied between 4 and 18 on the games we considered.”

There are two such networks created, the second one being the “target\_model”. This helps in stable learning and better convergence as the regular DQN tends to overestimate Q-values of potential actions in a given state. This makes it hard for the agent to explore the environment uniformly and find the right policy. Thus, I used two separate networks. The primary network is used to select an action and a target network is used to generate a Q-value for that action. In order to synchronize the networks, I copy weights from the primary network to the target one every 10,000 training steps. This is done in target\_train() method of model.py.

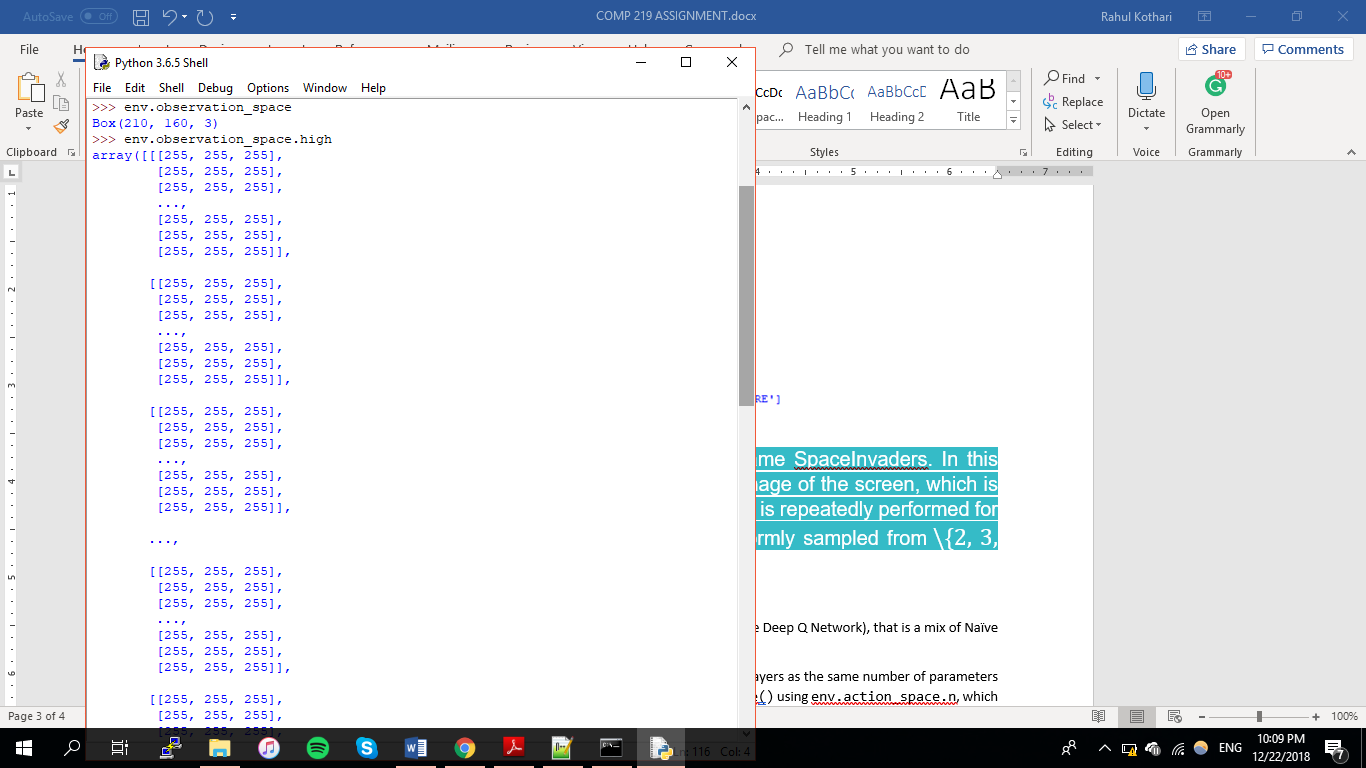
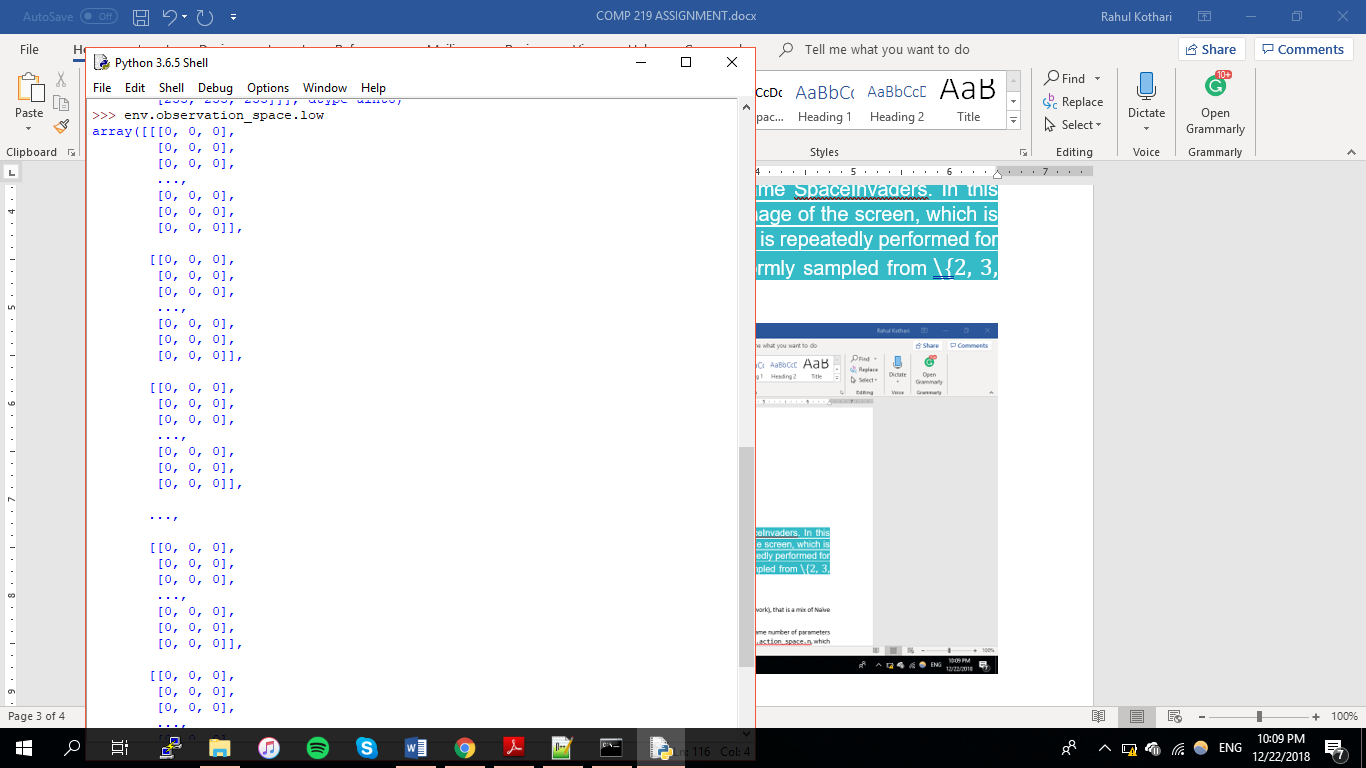
**Step 3 - Connection of the game to the network**

About the game:

The objective of Space Invaders is to maximize the score (reward) by killing as many opponent spaceships. There are 6 actions (shown below). As said in the openAI documentation, “In this environment, the observation is an RGB image of the screen, which is an array of shape (210,160,3). Each action is repeatedly performed for a duration of k frames, where k is uniformly sampled from {2,3,4}.”



Thus the action of 0 means do nothing, 1 implies fire, 2 implies move right in the frame and so on…

 A 210X160X3 matrix.

Preprocessing

As required by the DDQN model, our input image (of 210x160 RGB state) must be grey-scaled (colour adds unnecessary complexity) and downscaled to 84x84. This frame must then be stacked with two other states. Stacking states allow the neural network to understand the direction of movement of the laser beams. (All this is done in stack\_frames() and downscale() of preprocess.py).

**Step 4 – Deep reinforcement learning model**

As reiterated before, I am implementing a kind of DDQN (Double Deep Q Network), that is a mix of Naïve Q Networks and Convolutional Neural Networks.

The model expects an input shape of 84x84x3, and the output layer has the same number of features as the number of actions of the game (calculated using env.action\_space.n, which gives the length of the action space).

During training [trainAgent() in game.py], after preprocessing the state, the next action is predicted [predictAction()]. Then Epsilon is decayed to eventually reduce exploration [decayEpsilon()]. Env.step() takes in the predicted action as a parameter to give the new state. This state is again preprocessed. The new state, along with the action, its reward and other information are saved in the memory. The model is then trained by calling the replay() method in model.py. Also, every 10,000 steps the model weights are copied into the target model [in target\_train()]

Replay():

From the memory, this method randomly samples 32 experiences (containing state, action taken at this state, reward got, if the episode was completed or not, next state thus obtained) and trains the model on them. Experience replay avoids forgetting previous experiences and it reduces the correlation between experiences. If there is a high correlation, it may be very harmful for decision-making. The randomness breaks this correlation.

For each experience, it maps the reward to the action associated with it. The model is used to predict the confidence for the various actions. The target network is used to calculate the target Q value of taking that action at the next state. The maximum target Q is calculated (using the formula below) and discounted so that the future reward is worth less than immediate reward. Lastly, we add the current reward to the discounted future reward to get the target value. Subtracting our current prediction from the target gives the loss. Keras’ model.fit() calculates loss and optimizes the model based on it.

S

Target Reward Discounted predicted Q value

s

Reward Target Predicted Value

Gamma (discount rate)

predictAction()

In the beginning, the model must “explore” as much as possible to gather unique experiences. If it would always select the action with max Q-value, then there is a higher chance of overfitting. But after many episodes, the model should predict actions based on the Q-value (“exploitation”).

**Step 5 – Experimental Results**

Please find the video – “playingSpaceInvaders.mp4” to watch the trained agent play a round of Space Invaders!

Also, on running the simulation, the program runs the game for 3 episodes, printing the total reward each episode.

You may notice that the agent has been optimized to target the horizontally moving purple spaceship as it gives a reward of 300 points.