University of Tehran
Engineering Facility
Electrical and Computer
Engineering Faculty

Intelligent Systems

Final Project

Shiva Shakeri- Atefeh Mollabagher-Ashkan Jafari
810197691 - 810197590 - 810197483

February 2022

Contents

1. Introduction

By definition, in reinforcement learning, an agent takes discrete or continuous actions in a given environment in order to maximize some notion of reward that is coded into it.

We will utilize two model-free and off-policy reinforcement learning methods in this project and observe the effects on the performance of the agent in two environments: our chosen environment from the OpenAI gym and our implemented environment.

Atari Pong is a game environment provided on the OpenAI “Gym” platform. Pong is a two-dimensional sport game that simulates table tennis which was released in 1972 by Atari. The player controls an in-game paddle by moving it vertically across the left or right side of the screen. They can compete against another player controlling a second paddle on the opposing side. Players use the paddles to hit a ball back and have three actions (“stay”, “down”, and “up”). The goal is for each player to reach 21 points before the opponent, points are earned when one fails to return the ball to the other.

OpenAI’s environment gives an RGB image for its state observation (100,800 possible states). This is far too much information for simple algorithms to handle. So, to acquire the results quickly, we use 'pygame' to build a simpler environment.

2. Related Work

There's a paper relating to our project that we'll discuss about its applications. In the following, we will summarize this article and we will see how the data is obtained and will be used. We will also discuss the preprocessing and the model used in it.

2.1. Abstract Of Introduction

Learning to control agents directly from high-dimensional sensory inputs like vision and speech is one of the long-standing challenges of reinforcement learning (RL). Recent advances in deep learning have made it possible to extract high-level features from raw sensory data, leading to breakthroughs in computer vision and speech recognition. However, reinforcement learning presents several challenges from a deep learning perspective.These issues and challenges can be categorized in two groups: First, most successful deep learning applications have required large amounts of hand labeled training data while RL algorithms must be able to learn from a scalar reward signal that is frequently sparse, noisy and delayed. The delay between actions and resulting rewards seems particularly daunting when compared to the direct association between inputs and targets found in supervised learning. Second is that most deep learning algorithms assume the data samples to be independent, while reinforcement learning encounters sequences of highly correlated states. Furthermore, in RL the data distribution changes as the algorithm learns new behaviors, which can be challengeable for deep learning methods that assume a fixed underlying distribution.

The paper we are discussing demonstrates a convolutional neural network which can overcome the above challenges to learn successful control policies from raw video data in complex RL environments.

This research applies its approach to a range of Atari 2600 games implemented in The Arcade Learning Environment (ALE). Atari 2600 is a challenging RL that presents agents with a high dimensional visual input (210 × 160 RGB video at 60Hz) and a diverse and interesting set of tasks. The paper's goal is to create a single neural network agent that is able to successfully learn to play as many of the games as possible and for this goal the network is trained with a variant of the Q-learning algorithm, with stochastic gradient descent to update the weights.

2.2. Data Usage

Actually, there is no data in RL. We have an environment and an agent which the agent should try and try to learn the environment. So, data in this research is the environment and the reward which agent gets in each action it does.

2.3. Collecting Data

In RL projects there is no Data collecting and instead of that we need an environment. In this paper the environments were raw Atari games as Pong, Breakout, Space Invader, Seaquest and Beam Rider.

2.4. Preprocessing

This paper applies a basic preprocessing step aimed at reducing the input dimensionality. The raw frames are preprocessed by first converting their RGB representation to gray-scale and down-sampling it to a 110×84 image. The final input representation is obtained by cropping an 84 × 84 region of the image that roughly captures the playing area. The final cropping stage is only required because we use the GPU implementation of 2D convolutions from, which expects square inputs.

For the experiments in this paper, the function φ from the algorithm below applies this preprocessing to the last 4 frames of a history and stacks them to produce the input to the Q-function.

There are several possible ways of parameterizing Q using a neural network. This paper uses an architecture in which there is a separate output unit for each possible action, and only the state representation is an input to the neural network.

2.5. Model

In RL projects and this paper there was no classifier. The RL algorithm which was used in this project was DQN and also in this paper there was another algorithm named HNeat Best which reflects the results obtained by using a hand-engineered object detector algorithm that outputs the locations and types of objects on the Atari screen.

2.6. Model Performance

For performance this paper used a table in which different RL methods were compared with the authors methods. The table shows the final score of each algorithm for each game.

	B. Rider	Breakout	Enduro	Pong	Q*bert	Seaquest	S. Invaders
Random	354	1.2	0	−20.4	157	110	179
Sarsa [3]	996	5.2	129	−19	614	665	271
Contingency [4]	1743	6	159	−17	960	723	268
DQN	**4092**	**168**	**470**	**20**	**1952**	**1705**	**581**
Human	7456	31	368	−3	18900	28010	3690
HNeat Best [8]	3616	52	106	19	1800	920	**1720**
HNeat Pixel [8]	1332	4	91	−16	1325	800	1145
DQN Best	**5184**	**225**	**661**	**21**	**4500**	**1740**	1075

Figure 1

Note that as the figure below shows, in RL projects (as this paper) If we want to show our performance we should calculate the average reward in consequent episodes.

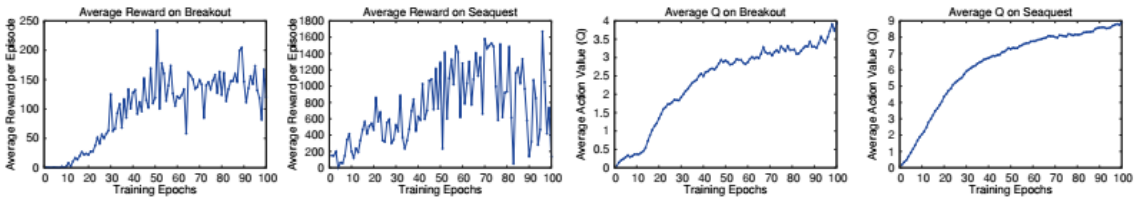

Figure 2

3. Methods

In this project, we'll use three methods: random action selection, Q-Learning, and Deep Q-Learning. In this section, we'll take a quick look at the general concepts of these methods, and then we'll try to apply them in our environment in the next section.

3.1. Random Action Selection

The main idea behind this method is that each action has the same probability, hence the agent chooses its actions at random. A random action is chosen in each game and step, and we can get observation, reward, and other essential information from the environment. After all games have been played, we can report the average rewards from all games to use this method to evaluate other methods.

3.2. Q-Learning

Q-learning is an off-policy reinforcement learning algorithm that seeks to find the best action to take given the current state. It's considered off-policy because the q-learning function learns from actions that are outside the current policy, like taking random actions, and therefore a policy isn't needed. More specifically, q-learning seeks to learn a policy that maximizes the total reward.

After Δt steps into the future the agent will decide some next step. The weight for this step is calculated as $\gamma^{\Delta t}$, where γ (the *discount factor*) is a number between 0 and 1 and has the effect of valuing rewards received earlier higher than those received later (reflecting the value of a "good start"). γ may also be interpreted as the probability to succeed (or survive) at every step Δt.

The algorithm, therefore, has a function that calculates the quality of a state–action combination: $Q: S \times A \rightarrow R$

Before learning begins, Q is initialized to a possibly arbitrary fixed value (chosen by the programmer). Then, at each time t the agent selects an action a_t, observes a reward r_t, enters a new state s_{t+1} (that may depend on both the previous state s_t and the selected action), and Q is updated. The core of the algorithm is a Bellman equation as a simple value iteration update, using the weighted average of the old value and the new information:

$$Q^{new}(s_t, a_t) \leftarrow \underbrace{Q(s_t, a_t)}_{\text{old value}} + \underbrace{\alpha}_{\text{learning rate}} \cdot \overbrace{\Big(\underbrace{r_t}_{\text{reward}} + \underbrace{\gamma}_{\text{discount factor}} \cdot \underbrace{\max_a Q(s_{t+1}, a)}_{\text{estimate of optimal future value}} - \underbrace{Q(s_t, a_t)}_{\text{old value}} \Big)}^{\text{temporal difference}}$$
$$\underbrace{\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad}_{\text{new value (temporal difference target)}}$$

where r_t is the reward received when moving from the state s_t to the state s_{t+1}, and α is the learning-rate.

Note that $Q^{new}(s_t, a_t)$ is the sum of three factors:

- the current value weighted by the learning rate. Values of the learning rate near to 1 make the changes more rapid.

- the reward to obtain if action is taken when in state (weighted by learning rate)

- the maximum reward that can be obtained from state (weighted by learning rate and discount factor)

An episode of the algorithm ends when state s_{t+1} is a final or *terminal state*. However, *Q*-learning can also be learned in non-episodic tasks (as a result of the property of convergent infinite series). If the discount factor is lower than 1, the action values are finite even if the problem can contain infinite loops.

3.3. Deep Q-Learning

We mentioned problems with q-learning in the last section, and we'll mention them again here. It is pretty clear that we can't infer the Q-value of new states from already explored states. This presents two problems: First, the amount of memory required to save and update that table would increase as the number of states increases and second, the amount of time required to explore each state to create the required Q-table would be unrealistic.

In deep Q-learning, we use a neural network to approximate the Q-value function. The state is given as the input and the Q-value of all possible actions is generated as the output. The comparison between Q-learning & deep Q-learning is illustrated below:

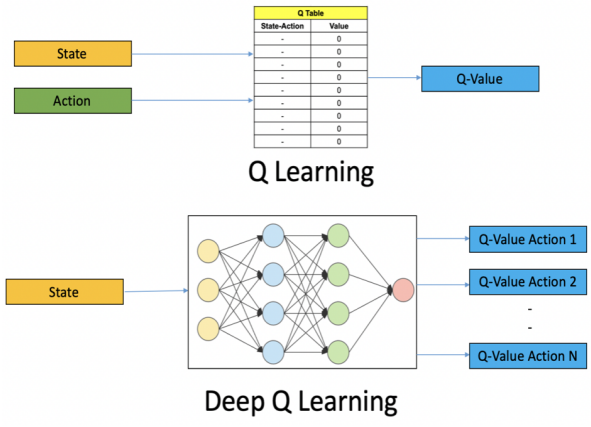

Figure 3. Comparison between Q-learning and deep Q-learning

There are two main phases that are interleaved in the Deep Q-Learning Algorithm. One is where we **sample** or remember the environment by performing actions and store away the observed experienced tuples in a replay memory. The other is where we select the small batch of tuples from this memory, randomly, and **learn** from that batch using an algorithm (gradient descent or 'adam') update step.

These two phases are not directly dependent on each other and we could perform multiple sampling steps then one learning step, or even multiple learning steps with different random batches. In practice, you won't be able to run the learning step immediately. You will need to wait till you have enough tuples of experiences in ***D***.

So the steps involved in reinforcement learning using deep Q-learning networks are as follows:

- All the past experience is stored by the user in memory
- The next action is determined by the maximum output of the Q-network
- The loss function here is mean squared error of the predicted Q-value and the target Q-value – Q*. This is basically a regression problem. However, we do not know the target or actual value here as we are dealing with a reinforcement learning problem. Going back to the Q-value update equation derived from the Bellman equation. we have:

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha[R_{t+1} + \gamma max_a Q(s_{t+1}, a) - Q(s_t, a_t)]$$

The section $R_{t+1} + \gamma max_a Q(s_{t+1}, a)$ represents the target. We can argue that it is predicting its own value, but since R is the unbiased true reward, the network is going to update its gradient using backpropagation to finally converge.

4. Implementation

In this section, we'll go through two environments and their features (OpenAI gym and our implemented environment), and also three methods (Random Action Selection, Q-Learning, and Deep Q-Learning) for training the agents.

4.1. Environments

4.1.1. OpenAI Gym: Pong

The OpenAI gym platform is a toolkit for developing and evaluating reinforcement learning algorithms. It supports teaching agents everything from walking to playing games. Atari Pong is a game environment provided on the OpenAI "Gym" platform. Pong is a two-dimensional sport game that simulates table tennis which was released in 1972 by Atari. It consists of two paddles: one for the agent and one for the computer. It has a ball that is in the center whenever the game begins. It moves to the right or left at random in the first game of each episode, and then to the player who loses the last point.

The openAI "gym" framework gives us the dataset we need in terms of observations at every time step. An observation consists of pixel values of the game screen taken for a

window of k consecutive frames. This is a numpy.array of shape (210, 160, 3), where the first value x is the height of the screen, the second value y stands for the width of the screen, and the third represents the RGB dimension for each pixel at coordinate (x, y).

After we import gym, there are only 4 functions we will be using from it. These functions are; gym.make(env), env.reset(), env.step(a) and … .

- **gym.make(env):** This simply gets our environment from the open AI gym. We will be calling env = gym.make('Pong-v0'), which is saying that our env is Pong.

- **env.reset():** This resets the environment back to its first state

- **env.step(a):** This takes a step in the environment by performing action a. This returns the next frame, reward, a done flag, and info. If the done flag == True, then the game is over.

- **frames:** the frames received from OpenAI are much larger than we need, with a much higher resolution than we need.(as it's shown in Figure 3.1)

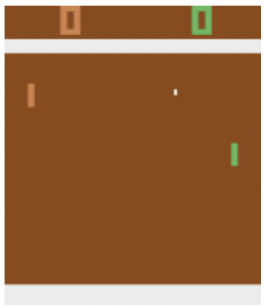

Figure 4. Frame from OpenAI, "Pong"

4.1.2. Our Implemented Environment

In this part, every step of building and designing our Pong Environment will be explained. First, like Gym, we design our elements which are necessary to the agent to develop a sense of its body and how to take different actions to hit the ball, minimize the difference of the scores and finally score more goals to win every game.

The elements are:

- The main Field
- Paddle
- Ball

The main field contains all the elements which are needed and they are 2 paddles and a ball. The field has a certain width and height and they are 760 and 720. 2 paddles are placed on 2 opposite sides of the field and the ball position is in the middle of the field. The main method of the field is updating. This method is like the step method in the Gym environment and as a result of each action, based on the current state, we move the paddles and balls so we can generate the 'Next state', 'Reward', 'Running' and 'Hit'. 'Next state' is the state we go to based on the action and it generates after moving paddles and the ball. 'Reward' has the same meaning as the reward in the previous environment and if a paddle(player) scores a goal it can receive a 100-reward, otherwise the reward is -100. 'Running' is similar to

‘Done’ in the Gym and if the ball is in the field, it means the game is not finished yet so the game is running so this output is true. otherwise one of the players has scored a goal so the game is not running anymore so this output is false which means the game should restart again. ‘Hit’ will be used as a sign a paddle which is near to the ball has touched the ball and hit it so it has defended the other player’s atack.

The paddle is our player or agent and also the AI player (It means we have 2 paddles in the main Field). For each paddle we should define its current x and y position in the main field of the game, length, width and speed. The main method for this element is the move method. In this method we explain in which condition and how the paddle should move. The x position of each paddle is constant and due to the rules of the game, each paddle can just move vertically. If the action which has been taken is up direction, the paddle’s y coordinates will be decreased. If the action is down direction, the paddle’s y coordinates will be increased.

The other element is the ball. We have just a ball in the main Field. As said before for paddles, we have some attributes for the ball too. These attributes are: the ball x and y coordinates, its velocity in both directions (x and y) and the ball size. Move is the main method for this element either. Due to the field walls, first, we check that the ball will stay in the field and it can not go out of the field from the top and bottom walls and if the ball bounces on these walls it somehow should be reflected by a reversed velocity, As a consequence, the direction of the ball changes. Second, we check if the ball is hit by any player or does any player score a goal or not. If the ball is out of the field from both left or right walls, it means one of the players has scored a goal so the game should start again so the ball should be placed in the middle of the field. If the ball is near any paddles and it did hit, it should be reflected like top and bottom walls and return to the field. If none of these conditions happen, the ball can follow its way.

4.2. Random Action Selection Implementation

In this method, a Pong bot is built which moves randomly. There are two possible actions in this game, so the probability of moving down and moving up are the same and are equal to 0.5.

4.2.1. OpenAI Gym Environment

The environment which is used in this part is ‘Pong-v0’ from gym default environments. In this variant of openAI Gym, the agent only makes action every k-frame and the same action is performed for k frames. The k parameter is chosen randomly from values 2, 3 and 4 at every step and this is called frame skipping. Sticky actions are another technique which are used in openAI Gym different variants and it sets a probability p of action being repeated without agent’s control. This parameter in ‘Pong-v0’ is 0.25. Algorithm 1 shows our implementation algorithm, and the result of this part is shown in figure

Algorithm 1 Random action selection in Pong gym

```
env ← Pongv0                                              ▷ Initialize
upAction ← 2
downAction ← 3
maxEpisodeNum ← 5000
episodeReward = 0
rewards = []
while episodeNum < maxEpisodeNum do
    action ← random(upAction, downAction)
    observation, reward, done, info ← env.step(action)
    episodeReward+ = reward
    if done then
        observation = env.reset()
        rewards ← episodeReward
        episodeReward = 0
        episodeNum+ = 1
    end if
end while
```

4.2.2. Our implemented environment

In this part, we used our implemented environment to see how random action selection works. For 5000 episodes (each episode contains a finished game - a finished game means one player reaches 21 goals), in each step we generate a random action between up and down with probability of 0.5, that the agent must do and then based on the environment after the action taken, we can get new state, reward, running and hit as output which have same meaning as new state, reward, done and info in the Gym environment. Then we can save the result of each episode and use these results to evaluate other methods. So, we can report the average reward per episode.

4.3. Q-Learning Implementation

In Algorithm 2, there is a q-learning algorithm for reminder.

Algorithm 2 Q-Learning in Pong gym

Initialize Q(s,a) arbitrary
Repeat (for each episode):
    Initialize s
    Repeat (for each step of episode):
        Choose 'a' from 's' using policy derived from Q (epsilon-greedy)
        Take action 'a', observe reward 'r' and next state, s'
        Update Q
            $Q(s,a) \leftarrow Q(s,a) + \alpha[r + \gamma max_{a'} Q(s',a') - Q(s,a)]$
        $s \leftarrow s'$;
Until s is terminal

Learning rate(α): Determines how much the agent learns. From the formula we can see that if an α is close to 0 then it relies more on the old value, whereas if α is close to 1 then it relies more on the current information it has learned.

Discount factor(γ): Determines the importance of future rewards. A factor of 0 will make the agent (or short-sighted) by only considering current rewards, while a factor approaching 1 will make it strive for a long-term high reward.

ε-greedy: At the beginning of the learning process, we would like to explore more of the actions that the agent can perform, and as we proceed, we will want to Exploit the knowledge we have accumulated. In order to implement this idea, we define a variable(ε), which will start high and the agent will learn that the variable will be reduced to a certain value. At each step of the agent, it will choose a random number between 0 and 1, if the number obtained is greater than ε, the agent will select a random action to explore. else, he will select the optimal action according to the Q-table.

In this section, we will apply this method to two environments (OpenAI gym and our implemented one) and discuss and give solutions to each of their problems.

4.2.1. OpenAI Gym Environment

Now in order to use this method in the Pong game, we will use a dictionary (as a Q-table) that will include all the possible states in the game as a key, and for each state 2 possible actions as a value. Each state in the dictionary is a vector of 3 variables [(Ball-x,Ball-y),P1,(Ball direction)], Where (Ball-x,Ball-y) represents the coordinates of the ball, P1(Agent) is the y coordinate of the left-top part of our player, and Ball direction represented by 2 points Old and New as (New.x - Old.x, New.y - Old.y).So we'll need four variables for the q-table: the agent's paddle, the ball's position, the ball's direction (states), and the actions (up and down). It's interesting to note that the dimensions of this q-table are around 73*80*80*4*2=3,737,600!

According to Algorithm 2, we set the hyperparameters as follows:

Learning rate(α): we built a function that reduces the learning rate from 0.8 to 0.01 during the training.

Discount factor(γ): So this is valued 0.99 in our implementation.

Epsilon(*ε*): it will start high (0.9) and the agent will learn, the variable will be reduced to a certain value (0.02).

Now, to implement this in a gym Pong environment, we create a class called "Agent," as we discussed earlier. We create the environment and initialize some positions and parameters, including the previous ball's position, previous ball's direction, and the minimum value of the parameter epsilon, in the first function called “__init__.”

in the "preProcess" function, To begin, we crop the image so that just the most crucial parts are visible. The image is then converted to grayscale. Finally, we use cv2 with nearest-neighbor interpolation to resize the frame. Then, using a threshold (110), we decide whether each pixel should be white or black. (The paddles and ball will be white). Figure 5 is the result of this process.

Figure 5

The states are then down-sampled in the next step to make the algorithm run faster. Then we set up the agent's paddle column, as well as the game's left and right boards. We'll try to find our paddle and ball in the next step. It's worth noting that we could have used the computer paddle's position but to reduce the states, we didn't. We simply search 5 pixels surrounding (right, left, up, and down) the previous ball's position to discover the ball faster. We also give the agent an extra reward for hitting the ball with the paddle. Actually, we get a reward every time the ball's x-axis direction changes from positive to negative. Instead of using vectors for the ball's direction, we simply use four main directions: up, down, left, and right, to decrease the states even more. At last, The "preProcess" function returns the position of the ball, the paddle of the agent, the direction in which the ball is moving, and the reward for hitting the ball by the agent.

In the next steps, we implement the epsilon-greedy taking action method and get the best action (the action which maximizes the Q) that is used in the "Q-learning" function. First, we initialize the Q-table, hyperparameters, and actions in the "Q-learning" function. In each episode, we update the q-table by first getting an action, then taking a new state, reward, and done signal from that action, calculating the overall reward, and finally getting the next state's best action from the q-table; and at the end, after the ball's volley, we reset the previous ball's position and direction.

In the result, we have seen a small improvement in agent performance after using this method in the gym's environment. It's because this environment has too many states, and learning about each one takes too long for the agent. As stated previously, there are much too many states to visit in training and hold the q-values in memory. To solve this problem, we came up with two solutions. To begin, we can build a simpler environment with fewer states, as described in section 3.2.2. Second, we can use another method that uses experience to learn about a few states and then generalizes that knowledge to new and similar ones. This method is also referred to as "deep q-learning" and is described in section 3.3.

4.2.2. Our Implemented Environment

As we explained previously, applying Q-learning in the gym environment takes too much time for training, thus we created a simpler environment (described in section 3.1.2) with fewer states. In this section, we will implement the Q-learning method in our environment and evaluate the results.

The reason is that we have a very large ‘discrete state-space’ and Q-Learning works by assigning a ‘value’ to every possible state-action pair and our ball can be centered on any pixel on a 780x720 pixel surface. This is a lot of possible states (and that just for the ball). To overcome this, we will group many of the possible states together! By splitting the game surface into a 40 by 40 pixel grid the ball can now only be in 342 possible locations! (This is less than the 561,600 possible locations we had before)

To start off we need to discretize the environment as the figure below:

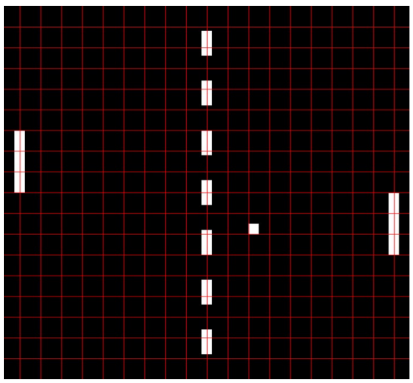

Figure 6

The functions we should add to the environment described before is as below:

1. Ball:

To bring this into practice in our environment, we'll need four variables for the q-table: the agent's paddle, the ball's position, the ball's velocity (states), and the actions (up and down), as well as a zero-filled initial q-table. According to Algorithm 2, we set the hyperparameters as follows:

Learning rate(α): we built a function that reduces the learning rate from 0.8 to 0.02 during the training.

Discount factor(γ): So this is valued 0.99 in our implementation.

Epsilon(ε): it will start high (0.5) and the agent will learn, the variable will be reduced to a certain value (0.02).

We begin by looping the episodes and building the environment using "DiscretePong". We start by setting the score to zero and then loop through each step in each episode. We use the epsilon-greedy policy to take action, then use "update" to get the next state and reward. Finally, the q-table and the next state are updated. The score is calculated after receiving the reward (r = +100 for a goal), and the episode ends when a player obtains a score of 21.

4.4. Deep Q-Learning Implementation

We only utilize deep q-learning in our implemented continuous environment, unlike the two previous ways. Algorithm 3 is a brief overview of what we'll be doing.

Initialize network Q
Initialize target network $\hat{Q}$
Initialize experience replay memory D
Initialize the *Agent* to interact with the Environment
while *not converged* **do**
  `/* Sample phase`
  $\epsilon \leftarrow$ setting new epsilon with ϵ-decay
  Choose an action a from state s using policy ϵ-greedy(Q)
  Agent takes action a, observe reward r, and next state s'
  Store transition $(s, a, r, s', done)$ in the experience replay memory D
  if *enough experiences in* D **then**
    `/* Learn phase`
    Sample a random *minibatch* of N transitions from D
    for *every transition* $(s_i, a_i, r_i, s'_i, done_i)$ ***in*** *minibatch* **do**
      if $done_i$ **then**
        $y_i = r_i$
      else
        $y_i = r_i + \gamma \max_{a' \in \mathcal{A}} \hat{Q}(s'_i, a')$
      end
    end
    Calculate the loss $\mathcal{L}$ =1/N $\sum_{i=0}^{N-1}(Q(s_i, a_i) - y_i)^2$
    Update Q using the SGD algorithm by minimizing the loss $\mathcal{L}$
    Every C steps, copy weights from Q to $\hat{Q}$
  end
end

Figure 7 shows our networks for a better understanding.

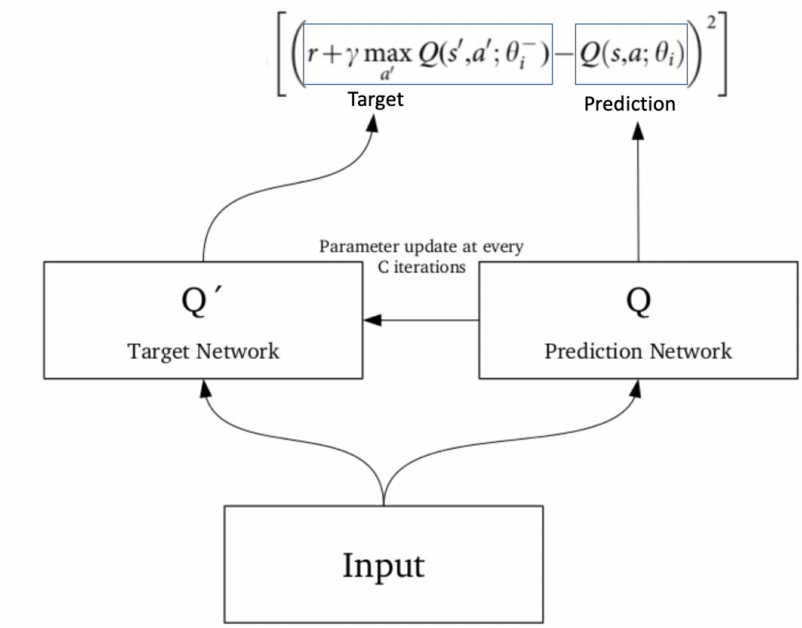

Figure 7

In order to explain the algorithm, we have a Q-network model with the state as input and q-values of each action in the output. To select an action we give our state to the model and we choose the action which has the maximum value in our main model's output. We have a memory that saves the s_t, a_t, s_t+1, r_t and done flag in each step. For each k-steps (Hyperparameter) we try to train our main model. In order to do this, we select a random batch (with batch size specified as a Hyperparameter) from the data stored in the agent's memory. Selecting random data ensures that we are training our model on states that are not sequential to each other, which prevents overfitting. After selecting a batch we will compute 'y' values for each datum. These 'y' values are equal to reward if the done flag is set for that datum or are equal to $\gamma(r_{t+1} + Q_{target})$. We fit our model on this batch of data in a single epoch. After C iterations('gameStep') we copy our main model's weights to the target model.

Now in order to use this method in the Pong game in our environment, we will define some methods in a class called "DQNAgent". To begin, we'll need a method called 'createDqnModel' to make a deep q-network model and a target model. These models have four fully connected layers, and we built them using "dense" from "keras". There are a few Hyperparameters that need to be specified as well: gamma = 0.99, batchsize = 32, syncModelEpisodes = 10.As the same as Q-learning we need an epsilon-greedy function called 'act' to choose the action based on epsilon's value. Another defined method in this class is called 'calcTarget' which gets the next state and reward. It calculates the 'y' value based on the target model's q-value for the next state and reward for that data. Next method is called 'remember' that gets s_t, a_t, r_t+1, s_t+1 and 'done' flag and appends this datum to the agent's memory; If memory's length is more than 10,000 we drop the oldest datum. In the 'prepState' method first we normalize values in each state and then add an empty dimension in the beginning of the stat array. We do this so that our model can get this datum as an input to predict its q-values. In the next method called 'replay' first we sample a random batch from the agent's memory and we prepare x and y train arrays as we previously mentioned. We then train our main model on the prepared data. In the last method called 'syncModels', we copy our main model's weights to the target value.

It is worthy to mention that we added ‘tf.compat.v1.disable_eager_execution()’ to make our models predictions faster. It was found in the github discussion section of tensorflow’s repository.

We start to loop from 0 to the specified number of episodes (5000) and perform the game steps for each episode. In each game step we create a new instance of the environment’s object and while the ball is in the frame, we act and update the environment. We then remember the required data for that step. If the length of the agent’s memory is more than 100, we perform replay every 50 steps.

5. Results and Conclusion

In this section, we'll show the results of each method and compare them over 5000 episodes of training.

5.1. Random Method As Evaluation Metric

As said before, we use the random method to compare and analyze other methods. Figure 8 shows the result of the average reward in the random method in a gym environment in 5000 episodes. It's obvious that the majority of the agent's rewards are about -21. It means that choosing random actions increases the chances of losing the game.

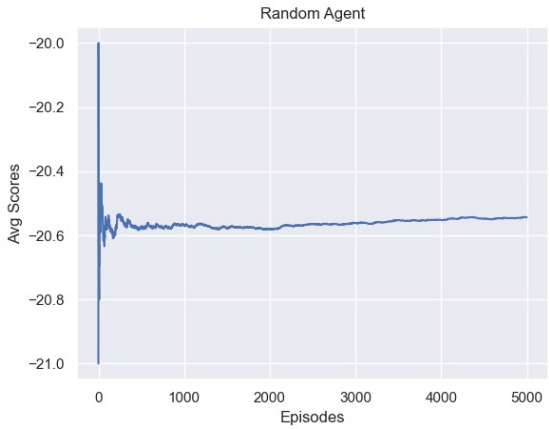

Figure 8

Figure 9 shows the result of the moving average reward in the random method in our environment.

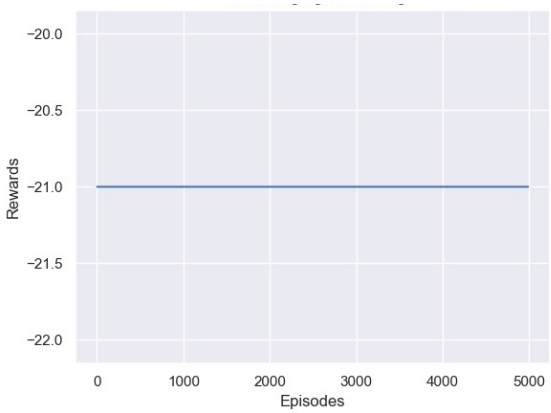

Figure 9

So the result of our environment, just like the gym, isn't good for this method and the agent mostly loses all the games. Now let’s see other methods' results and see their agent’s performance.

5.2. Q-Learning and DQN

Figure 10 shows the moving average reward and figure 11 shows the scores average of Q-Learning in our environment. As we said before it wasn't really possible to get the results from this method in the gym environment because of too many states.

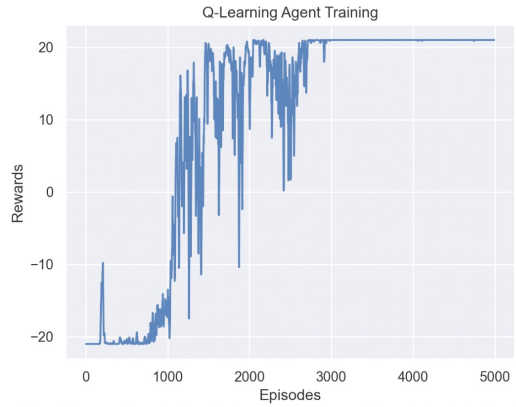

Figure 10

Figure 11

Now we compare Figure 5.3 to the result of the random method (figure 5.2), we can see that the agent has made significant progress and will be stable after around 3000 episodes, with an average reward of around 21.

Figure 12 and 13 also shows the agent's hits numbers in 400 first episodes and 500 last episodes which shows that agent has learned to hit so that it can win.

```
Average for previous 100 episodes = Experience:1.27  -  hits:0.01  -  episode:0
Average for previous 100 episodes = Experience:179.69  -  hits:1.7  -  episode:100
Average for previous 100 episodes = Experience:268.03  -  hits:2.88  -  episode:200
Average for previous 100 episodes = Experience:277.0  -  hits:3.0  -  episode:300
Average for previous 100 episodes = Experience:290.64  -  hits:3.2  -  episode:400
```

Figure 12

```
Average for previous 100 episodes = Experience:891.92  -  hits:13.8   -  episode:4500
Average for previous 100 episodes = Experience:899.49  -  hits:13.95  -  episode:4600
Average for previous 100 episodes = Experience:891.92  -  hits:13.8   -  episode:4700
Average for previous 100 episodes = Experience:902.0   -  hits:14.0   -  episode:4800
Average for previous 100 episodes = Experience:884.37  -  hits:13.65  -  episode:4900
Average for previous 100 episodes = Experience:891.92  -  hits:13.8   -  episode:5000
```

Figure 13

Figure 14 shows the Agent's performance during training time in DQN.

```
Episode 0 result: 21 - 3 , epsilon: 1.0
Average Reward   -1.9758507135016465
Episode 1 result: 21 - 1 , epsilon: 0.9995
Average Reward   -2.7
Episode 2 result: 21 - 0 , epsilon: 0.9990002500000001
Average Reward   -3.2
Episode 3 result: 21 - 0 , epsilon: 0.9985007498750001
Average Reward   -3.3
Episode 4 result: 21 - 1 , epsilon: 0.9980014995000627
Average Reward   -2.5
Episode 5 result: 21 - 0 , epsilon: 0.9975024987503127
Average Reward   -2.4
Episode 6 result: 21 - 0 , epsilon: 0.9970037475009377
Average Reward   -2.9
Episode 7 result: 21 - 0 , epsilon: 0.9965052456271872
Average Reward   -3.0
Episode 8 result: 21 - 0 , epsilon: 0.9960069930043737
Average Reward   -3.3
Episode 9 result: 21 - 1 , epsilon: 0.9955089895078716
Average Reward   -3.0
Episode 10 result: 21 - 0 , epsilon: 0.9950112350131177
Average Reward   -3.0
Episode 11 result: 21 - 0 , epsilon: 0.9945137293956112
Average Reward   -2.8
Episode 12 result: 21 - 1 , epsilon: 0.9940164725309134
...
Average Reward   -0.5
Episode 3731 result: 19 - 21 , epsilon: 0.15474657029843375
Average Reward   -0.2
Episode 3732 result: 20 - 21 , epsilon: 0.15466919701328455
Average Reward   0.1
```

Figure 14

5.3. Conclusion

We saw that for random agent results indicated that the agent cannot reach good results and loses all games. We use q-learning to give our agent the decision making module. We saw that our agent reached good results but to do this we simplified our environment and state space so that our memory usage would be able to handle seen states and q-table. If we used our complete and continuous environment, q-learning would have needed a large amount of memory and lots of game space to reach good scores. In order to solve these problems we use deep q-learning and neural networks so that we would be able to generalize agent's experience. We saw that this method would need less memory and seen states to reach equal or better results. The problem with deep q-learning was it took more time to learn in each step.