Q Learning vs Deep Q Learning

on Super Mario

Maria Anson, [1]Aditya Shanmugham [2]

Northeastern University [1,2]
anson.ma@northeastern.edu,[1] shanmugham.a@northeastern.edu[2]
https://github.com/aditya-29/Super-Marios

Abstract

To explore the advantages of Q learning vs Deep Q learning on the popular Super Mario game. To produce a rule of thumb to choose one methodology over the other.

Reinforcement Learning[1]

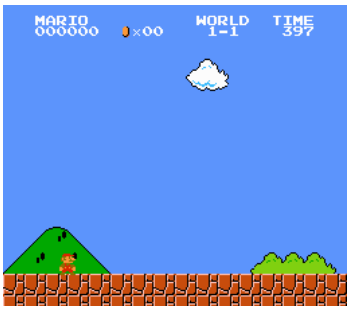
Similar to Supervised and Unsupervised learning, reinforcement learning uses *rewards* and *punishments* as signals for positive and negative behavior of the agents. The goal of the agent is to maximize the *total cumulative reward* by employing an action-reward feedback loop. Based on the problem in hand, a sophisticated estimator function is used to calculate the reward and current state of the agent to assist in the reinforcement step. Reinforcement learning is an advancement to the markov decision process. In this project we compared Q-Learning and Deep Q Learning by using the *Super Mario NES* game as a metric.

Every RL problem has essential elements to solve - *Environment, State, Reward, Policy*, and *Value*. The *Environment* corresponds to the level the agent competes for the Super Mario game. The *State* corresponds to the current situation the agent is facing. The *Reward* indicates the points earned by the agent in the game, *and Policy* indicates the best action the agent should take in a given state. Finally, the value indicates the future Reward the agent might receive if it takes action based on the optimal Policy.

Environment

We have used the "openai-gym" and "NES py" to set up the *Super Mario bros* game environment for our agent to play.

For the limited scope of our project we have only considered the level 1 of the original game. The game is setup in such a way that if the agent has only one life to reach the goal state.



An Illustrated image of the Super Mario Bros game running on NES python emulator.

Agent

The agent here refers to the *Super Mario* character in the emulator, and various Joypad inputs can control him. However, we have selected the right and *jump-right inputs* as the agent's only inputs.

Action

As indicated above, the action for our agent is limited to 2 discrete values: 0 - right and 1 - Jump-right. Since the game's goal is to reach the farthest right point without losing, these actions are the minimum for our agent to traverse the environment.

State

The State represents the current situation the agent is facing. Since we are comparing Q-Learning and Deep Q-Learning, the State has been modified as required. By default, the open-ai gym returns the simulated at the given timestep as the State of the agent. However, representing the whole image as a state is not feasible in Q-learning, so we used estimator functions to reduce the state space for the Q table.

Reward

The open-ai gym provides the reward, which calculates based on the agent's current state in points accumulated by the agent in the game. In addition, the rewards have

been modified to boost the training process in the Q learning model.

Bellman Optimal Q Table

The Bellman equation has determined the policy. In Q-Learning, the best actions are determined by taking the *argmax* of the current state in the Q-Table (state-action pair). The Q-Table has the size of *total_states x possible_actions*. Since it is impractical to list out all the states of the game in the Q-Table, we have devices specific methods to reduce the overall size of the Q-Table.

$$Q(s,a) = R(s,a) + \gamma \sum_{s'} \left(P\left(s,a,s'\right) \max_{a'} Q\left(s',a'\right) \right)$$

Bellman Equation used to update the current state-action pair of the Q-Table.

Temporal Difference

Temporal Difference is the component that helps the agent to calculate the Q-values with respect to changes to the environment over time. The TD consists of *a reward* for being in the current state added with the maximum possible Q-value for taking action from the current state scaled by the *discount factor (gamma)*

$$TD(a,s) = R(s,a) + \gamma \sum_{s'} \left(P(s,a,s') \max_{a'} Q(s',a') \right) - Q(s,a)$$

Temporal Difference equation consisting of reward, discount factor and current Q value.

> This is an example of an extract or quotation. Note the indent on both sides. Quotation marks are not necessary if you offset the text in a block like this, and properly identify and cite the quotation in the text.

Q-Learning

Game-State

Since the state returned by the open-ai gym consists of the simulation's snapshot at that given timestep, we have used estimator functions to reduce the state space of the Q-Table.

Skip-Frame

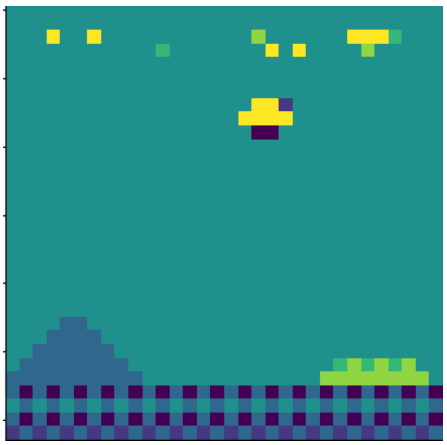
The NES py renders the environment at 30fps; most frames are rendered multiple times due to the input delay of the agent. Utilizing this factor, we process every four frames of the game by adding the rewards collected by the agent in the set of four frames. By skipping the frames, we can increase the training time by overlooking duplicate frames.

Grayscale-Observation

The state returned by the open-ai gym consists of three color channels - RGB. We have transformed the color image to a grayscale version, thereby reducing the Q-Table size by a factor of three, resulting in increased training speed.

Resize

The state has been resized to *a 32x32* image, which further reduces the size of the Q-Table. Along with resizing the given state image, we have also reduced the *bit-depth* of the image to 3, i.e., the values are segregated into six possible values, acting as an estimator function that reduces the image's complexity and displays the most prominent feature of the input image. Resizing and reducing the color depth has also increased the performance by 10-fold.



Output image after the preprocessing steps

Training the Model

The Q Agent is trained by employing the *exploration-exploitation* factor where an action is chosen based on previous experiences if the epsilon value is greater than the *exploration-exploitation* factor or a random action is chosen otherwise.

The agent is trained over 2000 *episodes*, an *episode* represents the lifetime of the agent. Throughout the episode the rewards are collected and the Q-Table is updated based on the Bellman Equation. The *exploration-exploitation* is determined by a random variable ranging from (0,1), if the random variable is greater than the *exploration-exploitation* factor then the agent is instructed to explore the environment, on the other hand if the value

is lesser then the agent is instructed to exploit the values from the Q-Table.

Epsilon Decay

The random variable is represented by the term *epsilon*, its equipped with a decay factor. As the number of epoch increases, the epsilon value reduces - forcing the agent to rely upon its previous experiences to select the best course of action.

$$epsilon = min_{eps} + (max_{eps} - min_{eps}) * exp(- decay * ep$$

Equation to determine the epsilon value after every epoch

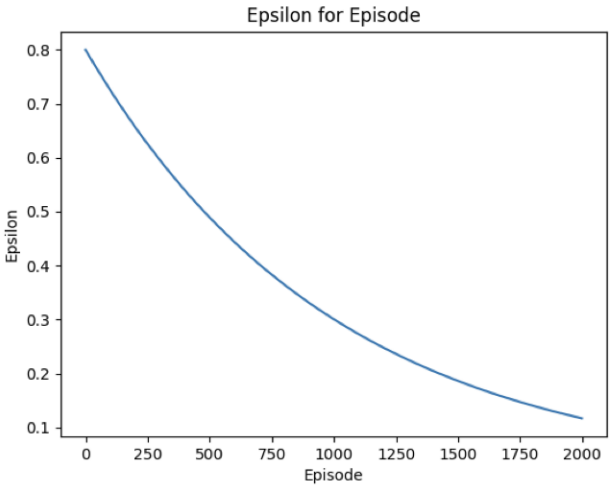
Experience Factor

We introduced an experience factor where we store the number of states the agent has visited over all the episodes. This number is then scaled down and added with the epsilon, thereby advocating the agent to rely upon the exploitation rather than the exploration value.

Distance Metric

We calculate the distance travelled by the agent from the beginning state. Add the scaled down value to the step reward function for calculating Q-Value. This punishes the agent when its stuck in the same state, eg: Jumping infront of a wall.

Training Graph : Total rewards over each episode



Training Graph : Epsilon decay for each episode

Deep Q-Learning

Deep Q-Learning is a modified version of Q learning, the main difference is how we construct the Q table. The agent uses the Q table at each step to select the best next action given a game state.

The Q table is constructed using two (Online and Target) Convolution neural networks. Two networks are used to avoid overestimation of values and to avoid initial bias when exploring new states and actions.

Features of the Game

Each state in the game emulator is an RGB image of dimension 240x256x3. At each state, the agent can take 256 different possible actions.

As a result creating two neural networks that process the RGB image and explore different action possibilities to learn the State, Action to Q value mapping is a heavy computational task. So we preprocess the states and limit the action to lower the computational complexity.

Data Preprocessing

Gray Scale Conversion - The RGB image is converted to the greyscale image as the image's color doesn't matter in figuring out the state. Now the image dimension is dropped from 240x256x3 to 240x256x1.

Down Size observation - The image is downscaled so that the Convolutional neural networks will look fewer pixels up. Now the image dimension is dropped from 240x256x1 to 84x84x1

Skip Frame - The game is a series of images (video), and there are 24 images in a second. There will not be many variations between consecutive frames, so we skip four frames in the video input and perform the same action for the four frames while adding rewards.

Frame Stack - To have continuity in understanding the movement we can take the historic frames to know the trajectory of Super Mario better. We stack four consecutive frames here to know the actions and state four steps back.

Neural Networks

The Online neural network architecture consists of two convolutional neural networks and a fully connected neural network with Relu activation.

The Target neural network has the same architecture but the weights are updated only after every 10,000 steps taken by the agent.

Parameter Update

The online model gives us an estimate of the Q value when given a state and action. The target model gives us the Q value given the next state with the experiences learned before.

The parameter update will essentially try to reduce the loss between the two TD values. The target network weight is copied from online network every 10000 steps.

$$a' = argmax_a Q_{online}(s', a)$$

$$TD_e = Q^*_{online}(s, a) \qquad TD_t = r + \gamma Q^*_{target}(s', a')$$

$$\theta_{online} \leftarrow \theta_{online} + \alpha \nabla (TD_e - TD_t)$$

Equation of parameter updates and loss

Exploration Vs Exploitation

Exploration - Trying different actions when you don't know the outcome of those action.
Exploitation - Taking the action you know which gives best reward from the previous experiences.

An important decision in reinforcement learning is to decide when to explore and when to exploit. The exploration is done whenever we get a random value less than our exploration_factor.

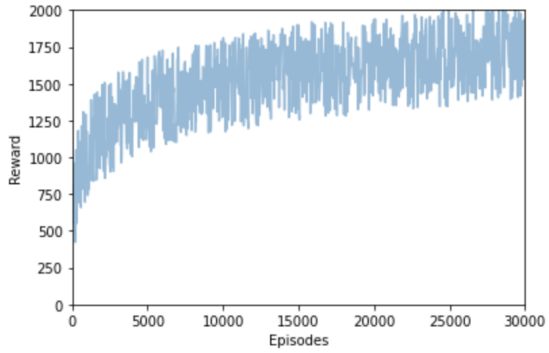
We use an exploration rate decay of 0.99 and our exploration rate is initially 1. At each exploration step, we decrease the value of exploration rate by a factor of exploration decay until it reaches 0.1.

exploration_factor = exploration_factor*decay
exploration_factor = min(0.1, exploration_factor)

After reaching an exploration factor of 0.1, our agent uses exploitation 90% of the time and 10% of the time on exploration.

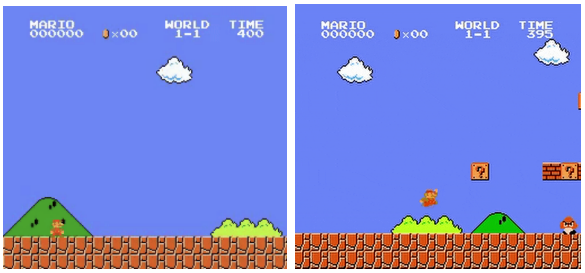
Training

The neural network is trained for 30000 episodes with 1060 Ti GPU for 20 hours. The checkpoint of the model is stored with the progress of environment, model and updated exploration rates.



Reward plot of the agent

Simulation



Simulation of Q Learned vs Deep Q Learned Super Mario

Summary

Q-Learning

The Q-Learning agent was trained over 2000 episodes which reached an overall reward of 1510, due to lack of training time the agent couldn't finish the game. The size of the Q-Table was 1,000,000 and the space occupied was 3GB.

Deep Q-Learning

The Deep Q-Learning agent was trained over 30,000 episodes, and the average reward was around 2000. The agent was able to complete the game without fail.

Based on the above results, we conclude that employing Deep Q Learning is the efficient way to train an agent to survive in a complex environment

References

Roderick, M., MacGlashan, J., and Tellex, S., “Implementing the Deep Q-Network”, arXiv e-prints, 2017.

Van Hasselt, H., Guez, A., and Silver, D., “Deep Reinforcement Learning with Double Q-learning”, arXiv e-prints, 2015.

Hausknecht, M. and Stone, P., “Deep Recurrent Q-Learning for Partially Observable MDPs”, arXiv e-prints, 2015.

B. Jang, M. Kim, G. Harerimana and J. W. Kim, "Q-Learning Algorithms: A Comprehensive Classification and Applications," in *IEEE Access*, vol. 7, pp. 133653-133667, 2019, doi: 10.1109/ACCESS.2019.2941229.

[1]Mnih, V., “Playing Atari with Deep Reinforcement Learning”, arXiv e-prints, 2013.