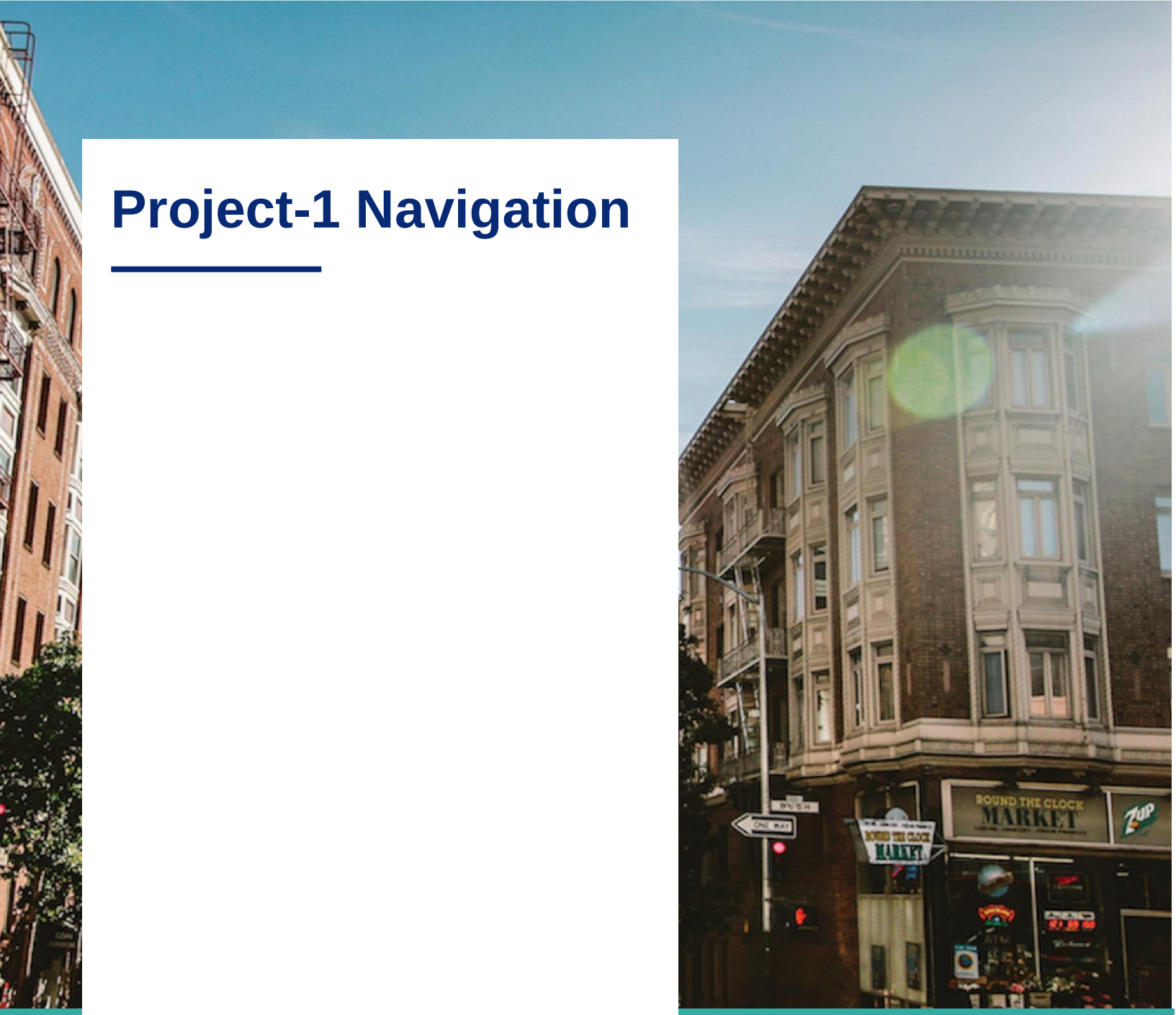
Project-1 Navigation



12TH MAR 2020

Amrit Paul

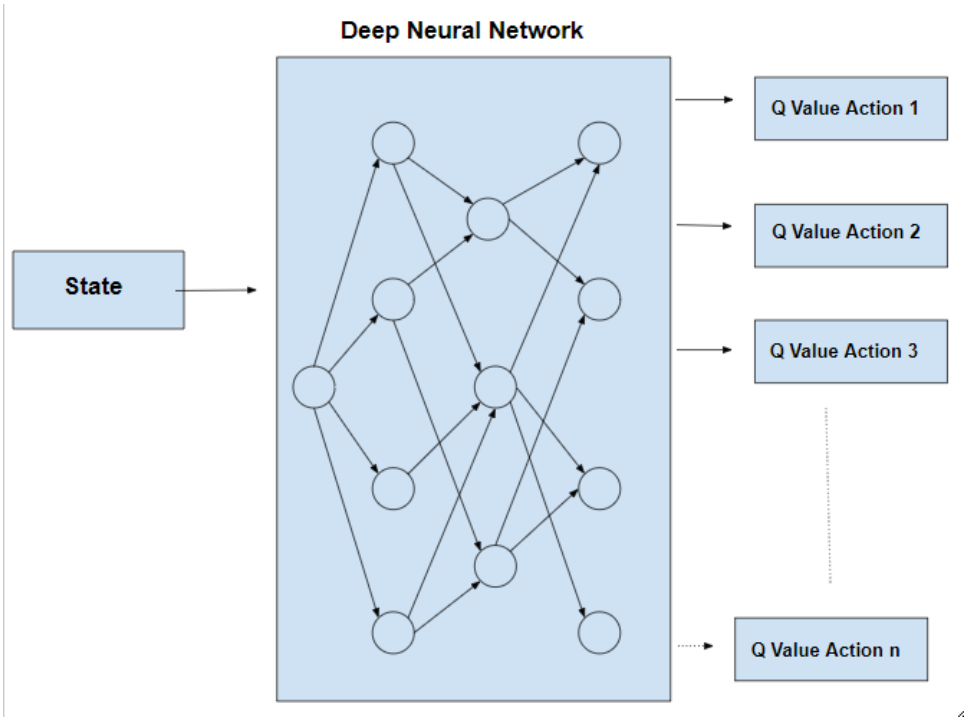
Problem Statement

The goal of the project is to demonstrate the abilities of a model-free reinforcement learning algorithm, particularly Deep Q-Learning. The project uses Unity environment, a game development framework and Pytorch, a deep learning framework, to train an agent to solve an environment consisting of 37 continuous states and 4 actions. The goal of the agent is to maximize expected cumulative reward by collecting only yellow banana's (+1 reward) and avoiding blue banana's (-1 reward).

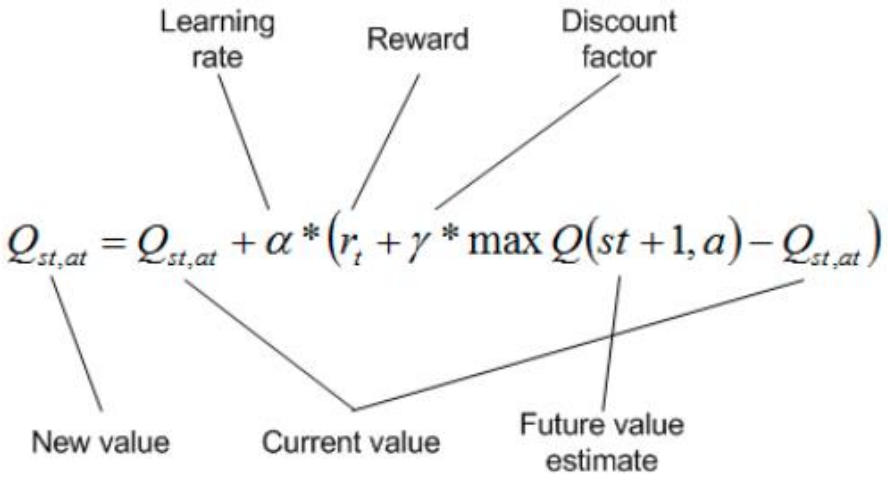
Description

The process of Q-Learning creates an exact matrix for the working agent which it can "refer to" to maximize its reward in the long run. Although this approach is not wrong, this is only practical for very small environments and quickly loses its feasibility when the number of states and actions in the environment increases. The solution for the above problem comes from the realization that the values in the matrix only have relative importance i.e. the values only have importance with respect to the other values. Thus, this thinking leads us to Deep Q-Learning which uses a deep neural network to approximate the values. This approximation of values does not hurt if the relative importance is preserved.

The idea of Q-learning is to learn the action-value function, often denoted as $Q(s, a)$, where s represents the current state and a represents the action being evaluated. Q-learning is a form of Temporal-Difference learning (TD-learning), where we can learn from each step rather than waiting for an episode to complete.



The above figure represents a deep neural network used to represent the Q-values (state and action pair).



The above equation represents the update policy of the Q-values, which uses current Q-value, immediate reward, learning rate and future estimated value of the maximum action value for that state.

Deep-Q-Learning Algorithm: -

- Initialize replay memory D with capacity N
- Initialize action-value function q' with random weights w
- Initialize target action-value weights w' <-- w
- For the episode e <-- 1 to M:
    - Initial input frame x_1
    - Prepare initial state: S <-- Z(<x1>)
    - For time step t <--1 to T:
        - Choose action A from state S using policy π <-- €-greedy (q'(S,A,w))
        - Take action A, observe reward R and next input frame x_2
        - Prepare next state S' <-- Z(<x_{-1},x_0,x_1,x_2>)
        - Store experience tuple (S,A,R,S') in replay memory D
        - S <-- S'
        - Obtain random minibatch of tuples (s_j ,a_j ,r_j ,s_{J+1}) from D
        - Set target y_j = r_j + gamma*(max$_a$(q'(s_{j+1},a_j,w')))
        - Update: Δw = alpha*(y_j – q'(s_j,a_j,w)) Δw q'(s_j,a_j,w)
        - Every C steps, reset: w' <-- w

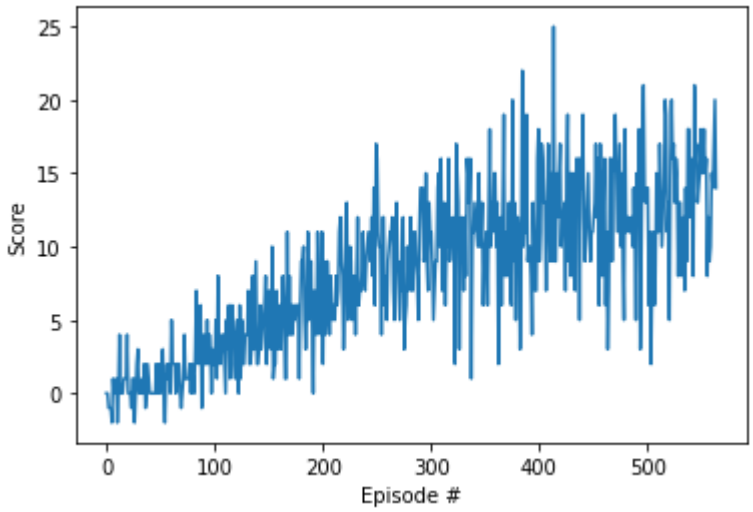
However, the algorithm described above in its raw form is highly unstable. Two techniques contributed significantly towards stabilizing the training

1. Experience Replay- Instead of running Q-learning on state/action pairs as they occur during simulation or actual experience, the system stores the data discovered for [state, action, reward, next_state] - typically in a large table. After a fixed number of iterations, sample a few experiences from this replay buffer and use that to calculate the loss and update the parameters. Sampling randomly this way breaks the sequential nature of experiences and stabilizes learning.
2. Fixed Q-targets- If we closely observed the Update: Δw step, we notice that the incremental weight step update is dependent on the weight w used in the parameter of q'(s_j,a,w). To solve this issue, two separate networks are used, one is the online network being learned and the other

being the target network. The weights of the target network are taken from the online network itself by freezing the model parameters for a few iterations and updating it periodically after a few steps. By freezing the parameters this way, it ensures that the target network parameters are significantly different from the online network parameters.

Results

The following diagram shows the results obtained after the neural network was trained.



Episode 100 Average score: 1.15

Episode 200 Average score: 4.70

Episode 300 Average score: 8.71

Episode 400 Average score: 10.80

Episode 500 Average score: 12.53

Episode 564 Average score: 13.01

Environment solved in 464 episodes!Average Score: 13.01

The hyperparameters used in this model are given below-

Hyperparameter	Value
Number of actions	4
Number of states	37
Number of episodes	2000
Max time steps per episode	1000
Epsilon start value	1.0
Epsilon end value	0.01
Epsilon decay rate	0.995
Replay Buffer Size	1e5
Batch Size	64
Gamma	0.99
Soft update of target parameters	1e-3
Learning rate (alpha)	5e-4
Update interval	4
Fully connected layer (neurons)	64

Improvements

We used DQN algorithm to train the agent, trained on Pytorch framework. To further reduce the time taken to reach the target score, better algorithms such as Double DQN, Dueling Network along with Prioritized Experience Replay can be used.