1 Overview

This project demonstrates the use of deep Q-Networks for reinforcement learning. Reinforcement learning is a branch of Machine learning where an agent is trained to correctly behave in a particular environment. The Agent interacts with the environment by choosing some action a. In response to the action the environment returns a new environment state s and a reward r. The goal of the agent is to maximize the cumulative reward.
In this particular project the agent uses Q-Networks to be able to easily deal with a big set of possible states S. Q-Network's are deep neural networks that serve as a nonlinear function approximator which are trying to find the best action value function. This means the Q-Network tries to estimate the expected sum of future rewards (discounted by γ) for a given state action pair.

$$Q_\pi(s,a) = \mathbb{E}[\gamma^0 R_1 + \gamma^1 R_2 + \gamma^2 R_3 + ...|S_0 = s, A_0 = a, \pi]$$

Due to the unstable nature of nonlinear functions there are multiple additions of the Deep Q-learning Algorithm to increase the stability. This project implements the following two:

Replay Buffer: Use a buffer to store N most recent Experiences (S, A, R, S' tuples). Sample n random elements from the buffer at xth every time step and learn from them.

Double DQN: Use two neural networks instead of one to select the best action in order to prevent the algorithm from propagating incidental high rewards received by chance. The target network is updated

2 Results

For this Project I tried many runs with different hyper parameters. I also tried to change the architecture of the neural networks. But as I added more layers/more nodes per layer I noticed that this dramatically decreased the performance of the agent. Probably because I would also have to change other hyperparameters such as the learning rate accordingly.
One trend I noticed was that each time I increased the number of nodes per layer, the resulting score plot fluctuated much more around the mean. This is probably because as the dimensionality of the action value function (characterized by the Q network) increases, the network also looses stability when trained with unchanged hyperparameters. After trying out multiple network architectures I settled on the following three fully connected layers:

Layer Nr.	Input size	Output size	Bias
0	37	70	yes
1	80	64	yes
2	64	4	yes

Additionally I tried to improve the decaying of epsilon for the epsilon greedy algorithm. The original implementation from the textbook is as follows:

$$\varepsilon = max(\varepsilon_0 * d^i, \varepsilon_{min})$$

where d is the decay rate for ε and i is the number of episodes. In order to get the last bit of performance out of my policy I introduced a new parameter called "eps cutoff". This parameter i_{cutt} is simply the number of episodes after which ε is set to ε_{min}.

$$\varepsilon = \begin{cases} max(\varepsilon_0 * d^i, \varepsilon_{min}) \text{ if } \varepsilon < i_{cutt} \\ \varepsilon_{min} \text{ if } \varepsilon \geq i_{cutt} \end{cases}$$

The sharp increase which is visible in the smoothed score chart in Figure 1 might be attributable to the epsilon cutoff, but I am not entirely sure.

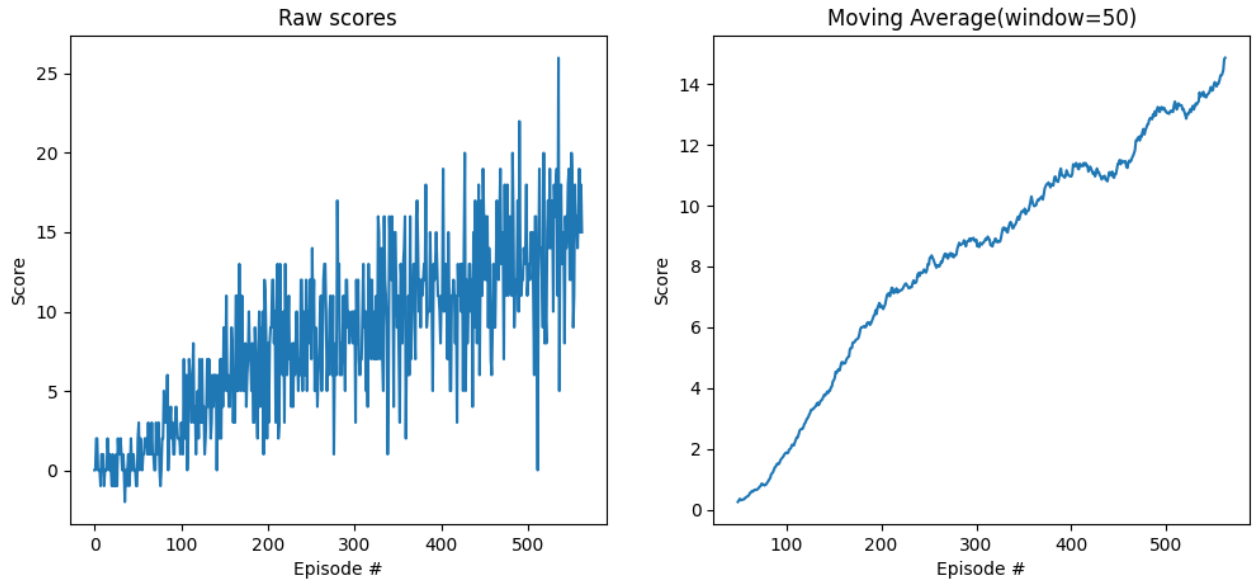

Figure 1: Scores that the agent achieved per episode

3 Ideas for future improvements

One way to improve the Model is always to fine tune the hyper parameters. Additionally, it is possible to implement additional algorithms that help to improve and stabilize the DQN algorithm. Such additions would be "Dueling DQN", "Distributional DQN" and "Noisy DQN".

Moreover I noticed that with the given environment states from the unity engine it seems that the agent is not able to detect bananas that are very far away. Thus, using actual pixel values as input might also improve performance in this particular case.