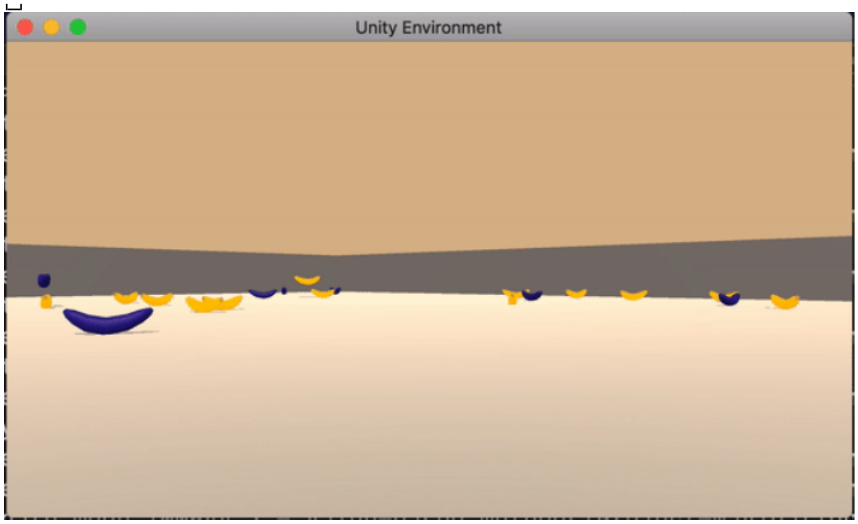
Project - Navigation

Jasdeep Sidhu

Introduction

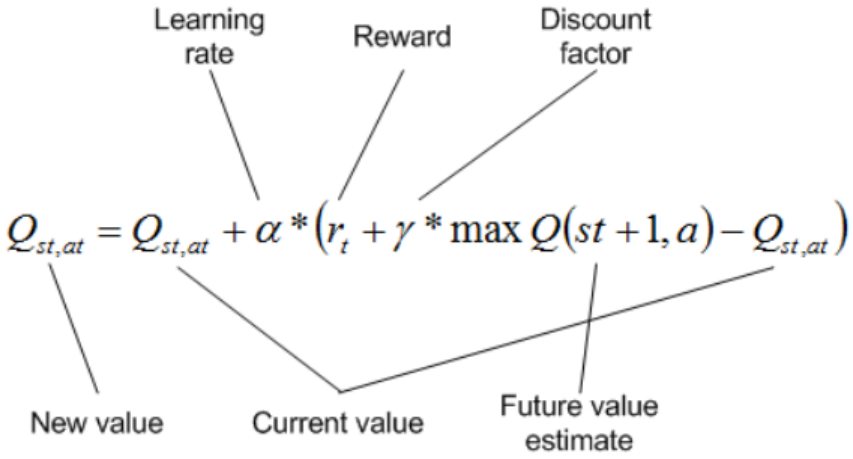
For this project, I trained an agent to navigate (and collect bananas!) in a large, square world. A reward of +1 is provided for collecting a yellow banana, and a reward of -1 is provided for collecting a blue banana. Thus, the goal of our agent is to collect as many yellow bananas as possible while avoiding blue bananas. The state space has 37 dimensions and contains the agent's velocity, along with ray-based perception of objects around the agent's forward direction. Given this information, the agent has to learn how to best select actions. Four discrete actions are available, corresponding to moving forward, backward, left and right. The task is episodic, and in order to solve the environment, the agent must get an average score of +13 over 100 consecutive episodes.



Learning Algorithm

Q-Learning

The learning algorithm we use is a subclass of value-based methods and is a model free reinforcement learning algorithm known as Deep Q-learning which is built using basic Q-learning algorithm. The goal of the Q-learning algorithm is to find an optimal policy that maximizes the total expected reward. It does so by learning the action-value function, represented as Q(s, a) , where s represents the current state and a represents the action being evaluated. In Q-learning the agent learns at each time-step (t) and does not have to wait till the end of the episode like in Monte-Carlo methods. The key is that upon taking an action, the agent enters a new state in which the current action value (Q-value) of that state is used as an estimate for the future reward.



Function approximator and Deep Q-Learning

The Q-learning algorithm is represented in a discrete spaced table comprising of state-action value pairs where the probablity of taking a certain action action is evaluated for each state. However this becomes increasingly cumbersome when dealing with large state and action spaces. A

continuous space is needed, and is achieved through a Function Approximator. A function approximator introduces a new parameter θ that helps us to obtain an approximation of the Q(s, a) such that

$Q(s, a, w) \backsimeq Q(s, a)$. In Deep Q-learning, neural networks are used as a function approximator and state spaces are represented by a large input vector. The problem is reduced to a supervised learning algorithm where the approximation represents the expected value and becomes the target. The mean-square error is used as the loss function and the weights are updated accordingly using gradient descent. A 2-hidden layer network is used as a function approximator with both the layers having 64 hidden units with relu activation applied after each fully-connected layer. A solution is found iteratively using Adam optimizer which yields the optimal weights.

$$\mathbf{w} \leftarrow \mathbf{w} + \alpha\big[R + \gamma\hat{q}(S', A', \mathbf{w}) - \hat{q}(S, A, \mathbf{w})\big]\nabla\hat{q}(S, A, \mathbf{w})$$

However, there is a problem with the algorithm as we are updating the guess with another guess and this can lead to harmful correlations. Two techniques are used to reduce these correlations:

Fixed Q-targets

As seen in the above equation, the target during training itself is dependent on w , the parameter being updated. This is problematic because the updated parameter has to deal with a constantly moving target which makes it hard to find convergence for an optimal solution. The problem can be eliminated by introducing a fixed target parameter w^{-} for the purposes of learning. The target parameter is trained separately, and is only updated periodically, while making surethat during each learning step it remains unchanged.

$$\Delta w = \alpha \cdot \overbrace{(\underbrace{R + \gamma \max_{a} \hat{q}(S', a, w^{-})}_{\text{TD target}} - \underbrace{\hat{q}(S, A, w)}_{\text{old value}})}^{\text{TD error}} \nabla_w \hat{q}(S, A, w)$$

Experience Replay

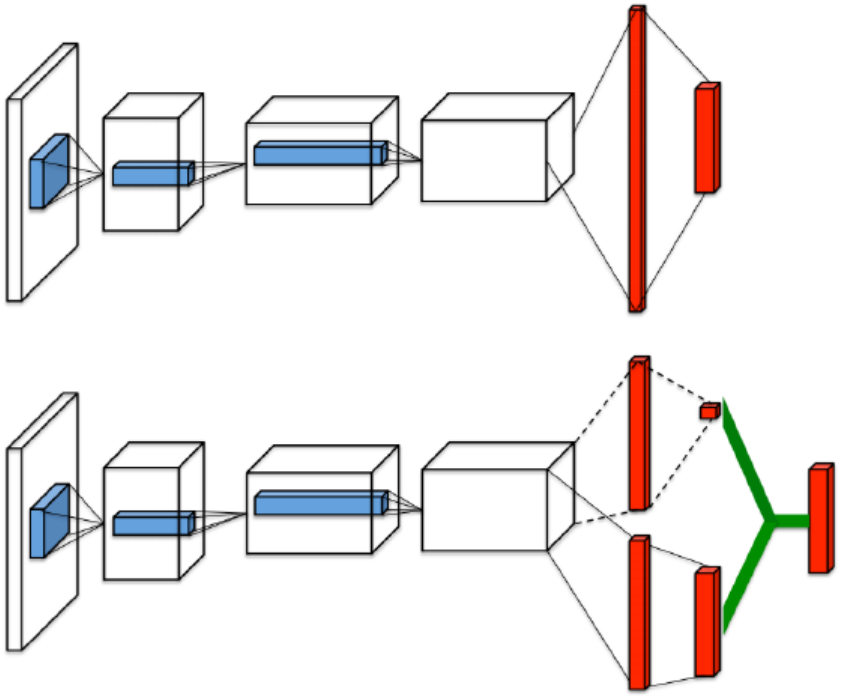
In reinforcement learning, the agent interacts with the environment and learns from sequence of state(S), actions(A) and rewards(R) and next state(S') which is stored in the form of an experienced tuple. These sequence of experience tuples can be highly correlated, and the naive Q-learning algorithm that learns from each of these experienced tuples in sequential order and be swayed by the effects of this correlation. This can lead to oscillation or divergence in the action values, which can be prevented by having a replay buffer, from which experience replay tuples are used to randomly sample from the buffer. The replay buffer contains a collection of experience tuples (SS, AA, RR, S'S ´). The tuples are gradually added to the buffer as we are interacting with the environment. The act of sampling a small batch of tuples from the replay buffer in order to learn is known as experience replay. In addition to breaking harmful correlations, experience replay allows us to learn more from individual tuples multiple times, recall rare occurrences, and in general make better use of our experience.

Double DQN

The Deep Q-Networks(DQN) are prone to overestimating the action value functions. The authors in this paper, proposed a new algorithm called Double DQN which can estimate the action value functions better. The idea of Double Q-learning is to reduce overestimations by decomposing the max operation in the target into action selection and action evaluation. The key is to use one network to choose the best action and the other to evaluate that action. The intuition here is that if one network chose an action as the best one by mistake, chances are that the other network wouldn't have a large Q-value for the suboptimal action. The network used for choosing the action is the online network whose parameters we want to learn and the network to evaluate that action is the target network described earlier:

$$Y_t^{\text{DoubleDQN}} \equiv R_{t+1} + \gamma Q(S_{t+1}, \underset{a}{\operatorname{argmax}}\, Q(S_{t+1}, a; \boldsymbol{\theta}_t), \boldsymbol{\theta}_t^-)$$

In the equation above, $Y_t^{DoubleDQN}$ is the TD target.



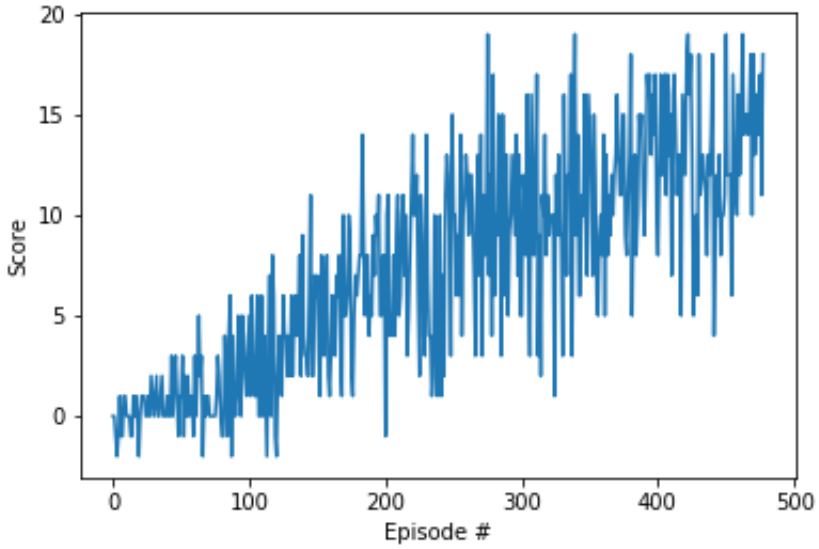
Hyperparameters

Hyperparameters	Values
Batch size	64
Replay buffer size	1e5
Discount factor	0.99
update interval	4
tau	1e-3
learning rate	5e-4
Number of episodes	1000
Number of time-steps per episode	1000
Epsilon decay	0.993
Epsilon start	1
Epsilon maximum	0.1

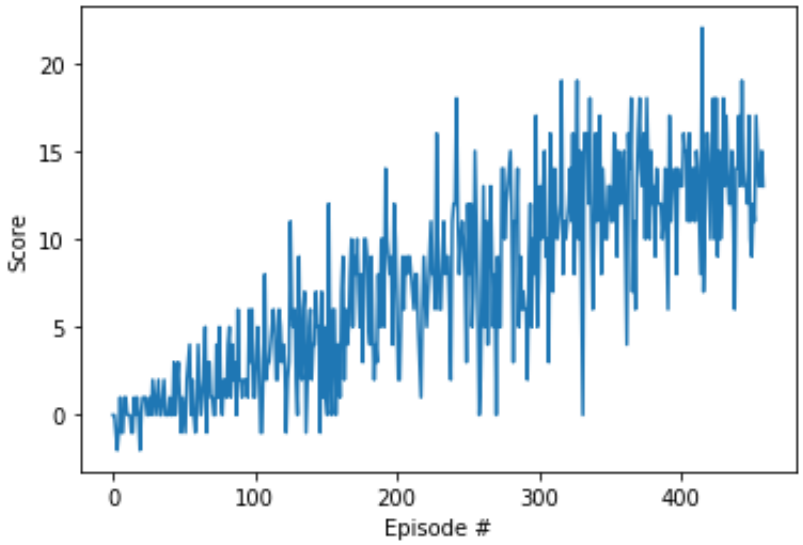
Results

The Deep Q learning algorithm took 378 episodes to solve the problem. The Double DQN gave the best performance solving the problem in 359 episodes. Below, the two figures plot the total score at each episode for DQN and Double DQN respectively.

DQN:



Double DQN:



Future Improvements

Some of the ideas for improvement that can be implemented are:

Using Dueling Double Q-Networks

- Adding prioritized experience replay
- Implementing distributional DQNs.