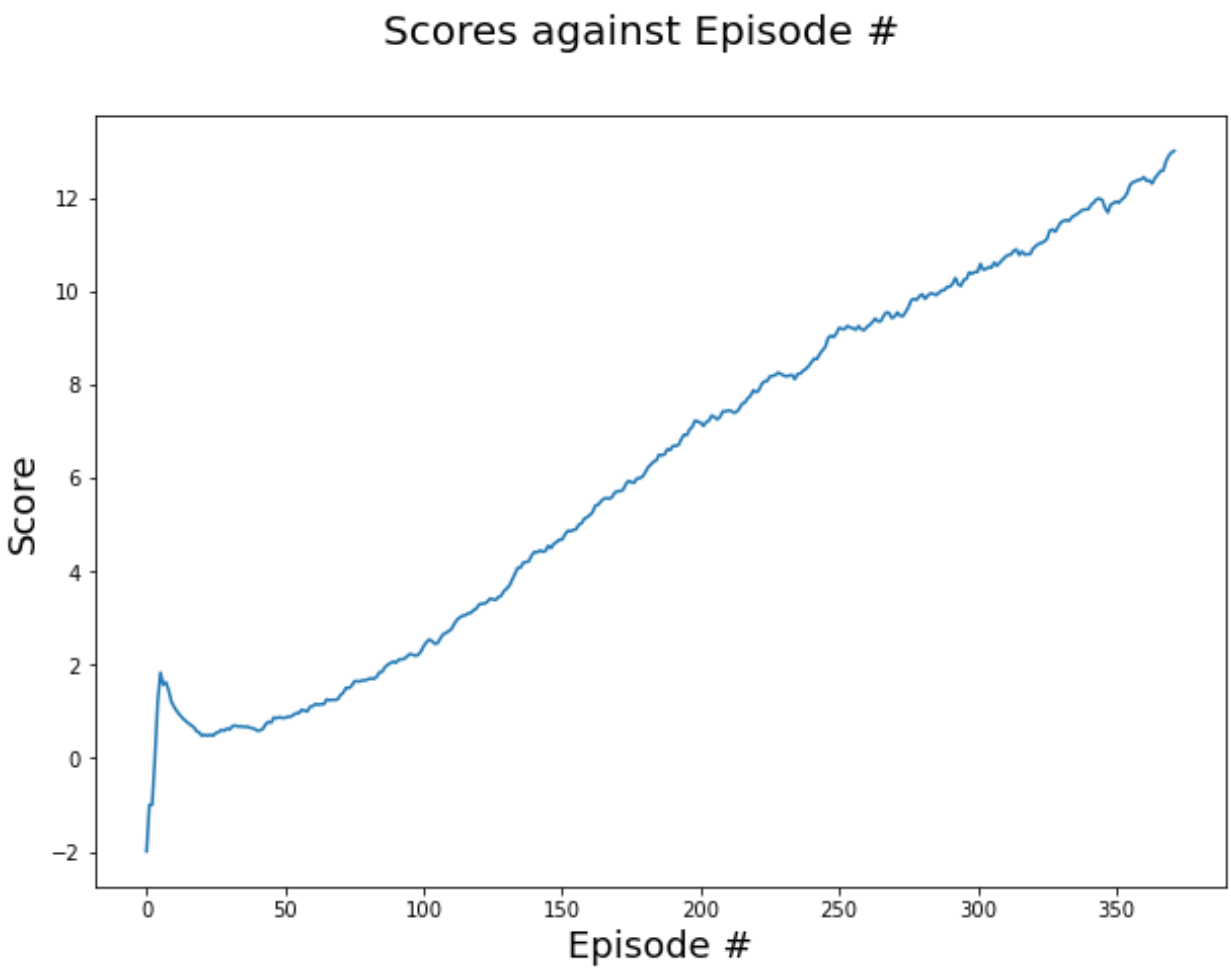
Deep Reinforcement Learning Nanodegree: Banana Collector

For training the agent, I use the vanilla DQN learning algorithms as they proved sufficient for solving the tasks. Solving the task was defined as getting an average reward of more than 13 for 100 episodes.

The learning agent solved the task in 372 episodes (see notebook). The scores plotted against the number of episodes is shown below:



Notes on implementation

General

The Navigation.pynb notebook is used to run the training. It creates an Agent (from dqn_agent.py), which in turn has a neural network as model (model.py) to map state → action-value.

The two most important functions the **agent** is called by the main function are:

1. act
2. step

The **act** function determines the next action the agent will take by using epsilon-greedy policy. Note that epsilon is decreased by every step the agent takes, starting with 1, down to minimum epsilon of 0.0001. Decaying epsilon had a positive effect on learning efficiency (learning in 542 episodes → learning in 372 episodes), especially in the beginning.

The **step** function is basically the agent's update function. It saves the currently seen (S,A,R,S') tuple in the replay buffer, samples from it, and updates the Q-values accordingly. Note that the updates to the state-action values are done by a greedy policy (unlike the epsilon-greedy used to select actions). Furthermore, I use the mean squared error as loss (loss averaged over the last BATCH_SIZE samples, here 64). The target network is used to predict the next_state's values.

Learning algorithm

The learning algorithm used is considered the vanilla DQN approach. Having a single neural network is not enough for the agent to learn, so there are basically two main tweaks implemented:

1. Separate target network
2. Replay buffer

A **separate target network** is used to increase the stability of the learning process. The target network is updated only after a fixed number of steps, by copying over the currently learned weights. We do this to prevent the Q values to diverge, i.e. giving the network time to converge.

The **replay buffer** is used in order to break correlations between subsequent states. We save (S,A,R,S') tuples of recently seen samples, and for learning, we randomly sample from these tuples. This way, we can learn from experiences multiple times, which allows for faster convergence. The second advantage is the breaking of correlations between states, hence increasing robustness.

Hyperparameters

I mainly use the same hyperparameter settings as used for solving the Taxi environment.

A great influence on the performance of the model was the structure of the neural network. Using two layers of 64 neurons, it took more than 3000 episodes to converge. Increasing to layers of 128 neurons decreases the episode # to 1823. Further increasing the amount of neurons actually decreased performance. Adding a separate, third layer, had a better effect, cutting down the number of episodes to 500-600 for solving the task.

Ultimately, I used a **fully-connected** feed-forward network with three hidden layers and sizes (128, 128, 64), respectively.

Improvements

I see some improvements one could add:

- Prioritized replay buffer
- Double DQN

A **prioritized replay buffer** samples the replay buffer non-uniformly. One way to prioritize is the TD error, i.e. the difference between target and expected value. This allows for more efficient learning.

Another improvement one could implement is the **Double DQN** algorithm. It is used to counter the bias of overestimating values. As in every step we use the maximum, greedy value for estimating the Q value, it introduces a maximization bias to the learning. Using two different estimators can avoid the bias by separating updates from estimated biases.