Project: Navigation

The goal of this project is to train an Agent to navigate and collect bananas in a large, square world. A reward of +1 is given for collecting a yellow banana and -1 for blue banana. Therefore, the goal of an Agent is to collect more yellow bananas while avoiding the blue bananas.

State space has 37 dimensions and contains agent's velocity along with ray-based perception of objects around agent's forward direction. In this context this, the Agent's task is to pick in every step of executed environment one of following 4 actions:

- 0: move forward
- 1: move backwards
- 2: turn left
- 3: turn right

The task is episodic and is considered solved if the Agent has achieved an average score of +13.0 over 100 consecutive episodes.
The project environment is running inside the Unity simulation engine and trained Agent can interact with it using the unityagents API.

Learning Algorithm

To achieve the goal of this project I decided to use the Deep Q-learning algorithm, which combines the traditional Q-Learning method of Reinforcement Learning (aka Sarsa Max) with the learning of Q-Table approximation function using Convolutional Neural Network (aka Deep Q-Network) for training the Agent to solve the project environment. The Q-Learning algorithm uses the so-called **Epsilon-Greedy** policy to select actions while being trained. Epsilon specifies the probability of selecting a random action instead of following the "best action" in the given state (exploration-exploitation tradeoff)
Also, to minimize the known weak points of Q-Learning algorithm, my implementation includes the following two improvements, initially offered by Google's DeepMind team as part of their solution for Atari 2600 challenge:

- **Fixed Q-targets** The TD target is dependent on the network parameters that we're trying to learn. This can lead to harmful correlations. To remove this, we use a separate network with identical architecture. The target network gets updated slowly with hyperparameter TAU during so called soft update and local network updates regularly after each Agent's action.
- **Experience Replay** When the agent interacts with the environment, the sequence of experience tuples can be highly correlated. The naive Q-learning algorithm that learns from each of these experience tuples in sequential order runs the risk of getting biased by the effects of this correlation. By sampling from stored experience records at random, the action values can be prevented from oscillating or unwanted diverging.

Taking into account above mentioned improvements the Deep Q-Learning algorithm for training an Agent has following structure:

- Initialize replay memory capacity.
- Initialize the policy network with random weights.
- Clone the policy network, and call it the *target network*.
- *For each episode:*
    - Initialize the starting state.
    - *For each time step:*
        - Select an action *Via exploration or exploitation*
        - Execute selected action in an emulator.
        - Observe reward and next state.
        - Store experience in replay memory.
        - Sample random batch from replay memory.
        - Preprocess states from batch.
        - Pass a batch of preprocessed states to the policy network.
        - Calculate loss between output Q-values and target Q-values. This requires also pass to the target network for the next state
        - Gradient descent updates weights in the policy network to minimize loss.
    - After X time steps, weights in the target network are updated to the weights in the policy network.

The Neural Network used for learning of Q-Table approximation function has following architecture :

- Input (37 nodes)
- Fully Connected Hidden Layer (64 nodes, Relu activation)
- Fully Connected HIdden Layer (64 nodes, Relu activation)
- Output (4 nodes)

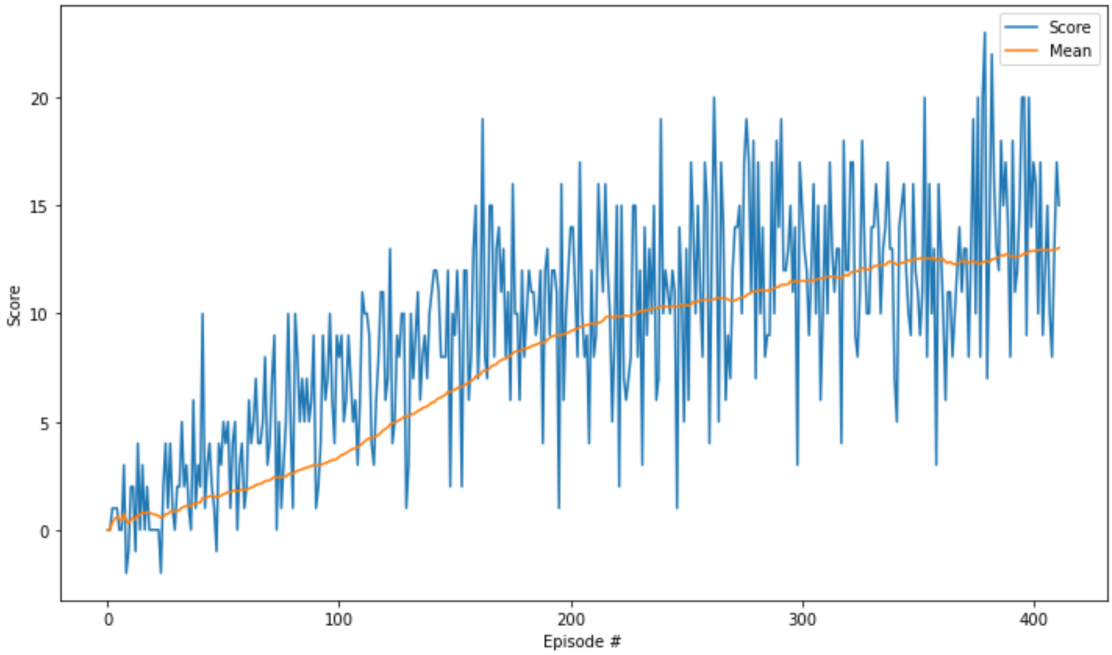
Used optimizer type: Adam optimizer with a learning rate LR=5e-4.

Agent training hyperparameters:

- Max episode length: 1000 time steps
- Epsilon param for epsilon-greedy policy evaluation: start=0.9, end=0.0, decay=0.993
- Experience replay memory buffer size: 100000
- Batch size: 64
- Network weights update frequency: every 4 steps
- discount factor for prioritizing future rewards (gamma): 0.99
- hyperparameter for soft update of weights of target network (Tau): 0.001

Plot of Rewards

Following graph demonstrates the progression of rewards score per episode during the Agent's training phase. Additionally, the progression of mean reward over the episode scores sliding window is displayed in another color. The graph shows that the Agent successfully learned to reach an average reward of at least +13 over 100 consecutive episodes. The Agent managed to achieve this goal after 412 episodes of learning.



Ideas for Future Work

Using the original Q-Learning algorithm with Experience Replay and Fixed Q-Targets allowed the Agent to achieve a target score of at least +13 over 100 consecutive episodes after 412 episodes of training. Additional playing with hyperparameters of neural network could probably allow to slightly decrease this value. But to get even better results, the valious improvements for the original DQN network architecture and algorithm should be tested. Most common among them to implement are:

- Double DQN
- Dueling DQN
- Prioritized experience replay

Probably even more impressive results in improving a speed of solving the project environment by the Agent could be achieved using these advanced improvements or DQN algorithm:

- A3C - Asynchronous advantage actor-critic
- Noisy DQN
- Distributional DQN

Also a combination of multiple of mentioned DQN improvements (a.k.a. Rainbow DQN algorithm) may show even better results than each of them separately.