Solving the Banana Environment Task

Marcel Kühn

Overview

The Banana Environment task is implemented in the Unity 3D framework and challenges an agent to collect as many yellow bananas while avoiding any blue ones. Every yellow banana collected yields a reward of +1 while any blue banana collected results in a reward of -1. The goal of the agent is to maximize the overall reward in an episode. Once the agent manages to collect an average of +13 per episode over the past 100 episodes, the task is considered to be solved.

The navigation within the environment allows for any movements in a two-dimensional space and are encoded as follows: (0) forward, (1) backward, (2) left, (3) right. These four possibilities define the action space of the agent and the output space of the neural network.

The agent's decisions are based on its current space. The space is a 37-dimensional vector that contains information about its current velocity and its field of vision (ray-based). This 37-dimensional vector serves as the input to the neural network.

Learning Algorithm

The learning algorithm is based on functional Q-learning in which the parameters of a neural network are tweaked to minimize the loss of the output. The architecture of the neural network consists of a 37-dimensional input layer (same dimension as the state space), three fully connected hidden layers of size 128, 64 and 32 and a four-dimensional output layer (same dimension as the action space). To stabilize the learning, two identical networks are initialized, one on which the actual training happens (local network) and one which serves as a target network that is only updated once in a while by copying over the current parameter values from the local network. Without this, the update rule for the parameters (weights) of the neural network are derived from the output of the network and thus its current weights. Having another neural network (target network) to calculate the output and then updating the weights of the local network according to this removes this correlation and improves the learning. This is called fixed Q-targets.

The learning follows a sequence of observing a state, picking and executing an action for that given space and subsequently observing how the environment responses (reward, next state) until an episode ends. The learning runs until one of the break conditions is reached (average of +13 reward per episode over the last 100 episodes or the maximum number of episodes reached, 1800).

The action selection is done according to an epsilon-greedy policy. The epsilon parameter decays after each episode and was tuned over several iterations of the algorithm. Setting the decay too high results in too much random behaviour that only allows the agent to solve the task after many iterations.

Once an action is selected and executed, the agent transitions into a new state and receives a reward. Each of these tuples, (state, action, next_state, done) is then stored into an experience buffer. After every fourth episode, the replay buffer is checked if it holds enough (more than the needed batch size for training) samples (tuples). If that is the case, a batch is sampled from a uniformly random distribution.

The sampled experiences are then used to update the weights of the (local) network. For each experience tuple, the total value is calculated by taking the received reward for the given action in the given state and adding the estimated future reward (discounted by gamma) by picking the next greedy action and its associated Q-value. In order to avoid the overestimation of the action values of the next states, this solution uses a Double DQN which lets the local network pick the greedy action but taking the values from the target network to determine the Q-value.

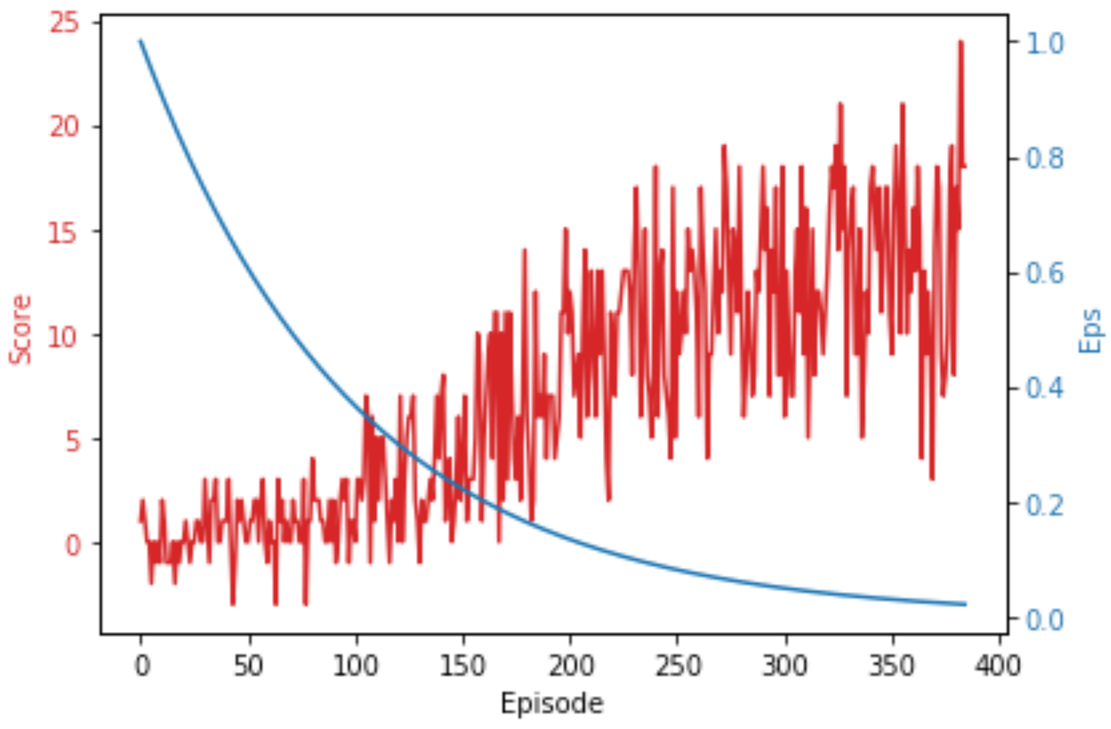
In a final step, the output of this calculation (Q_targets) is compared to the current output of the network (Q_expected). This defines the loss of the network for the given batch sample, calculated as the mean squared error. Once this is known, it can be backpropagated into the network to update the weight parameters of the local network. In this implementation a soft update is used to copy the parameters of the local network to the target network so that the update happens with weight tau.

Overview of the Hyperparameters

BUFFER_SIZE	10000
BATCH_SIZE	64
GAMMA	0.99
TAU	0.001
LR	0.0005
UPDATE_EVERY	4
WEIGHT_DECAY	0.00
EPS_START	1.0
EPS_MIN	0.01
EPS_DECAY	0.99
DIMENSION HIDDEN LAYERS	(128, 64, 32)

Results

As it can be seen in the following plot, the algorithm proved to be able to solve the task in less than 400 episodes. Tuning the epsilon decay heavily influenced the outcome of the training and was tweaked to be close to the minimum after 400 episodes.



Outlook

There are several approaches to further improve the performance of the algorithm, namely Prioritized Experience Replay (instead of sampling from a uniform random distribution, sample from a distribution that is proportional to the magnitude of error of the experience) and Dueling DQNs (state value and advantage value for each action is calculated in parallel layers of the network). Furthermore, the hyperparameter tweaking was mainly focused on epsilon. A different architecture of the neural network could yield very different (and superior) results.