Project Navigation

Introduction

This report describes my implementation and summarizes my results for the navigation project. It is a subproject of the Udacity reinforcement learning nanodegree program and is about training a reinforcement learning agent that is able to collect bananas of the right color by navigating in a squared environment. An example of the environment is shown in Figure 1. Further details about the environment are described in the readme file of this repository.

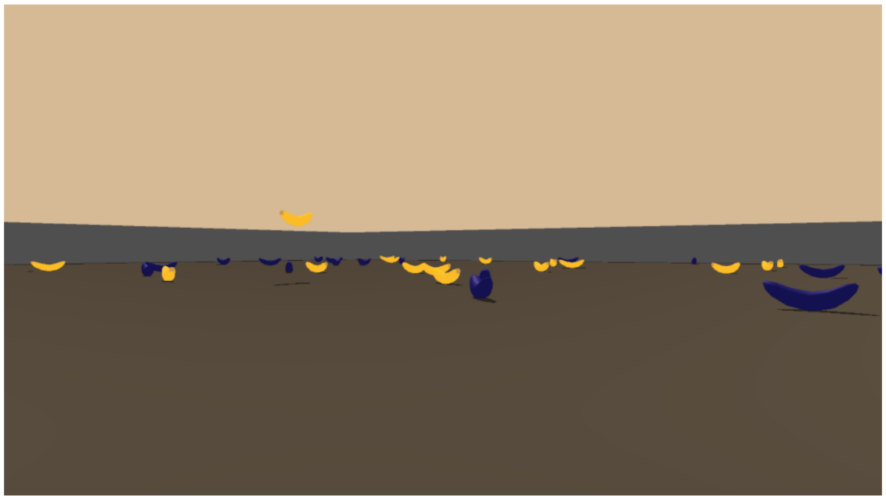

Figure 1: Environment screen

Reinforcement learning

Algorithm and Extensions

A deep reinforcement learning approach is applied to train a so-called *agent* that solves the navigation problem. Reinforcement learning has a sequence of repetitive steps (see also Figure 2):

1. The agent starts in a known state
2. Based on the agents knowledge, the best action is selected
3. The environment returns an observation containing information about the new state and a reward for taken action

This sequence of steps is repeated until the goal of the environment was reached or the number of actions exceeds a step limit. The reward returned in each step is stored and used for training the agent to select the best action.

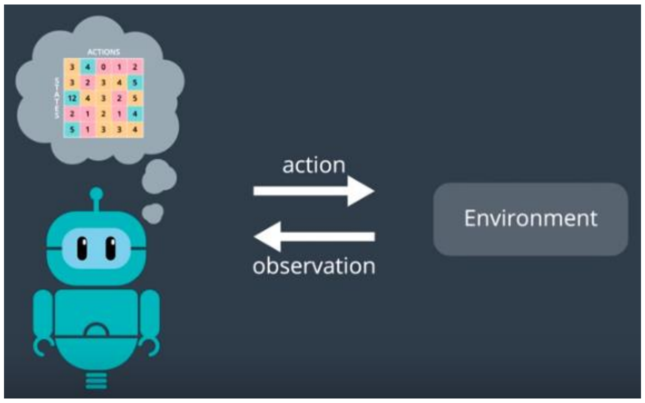

Figure 2: Refinforcement learning interactions

There are different reinforcement algorithms available. Approaches like the so-called *temporal difference* methods are training the agent during the episode with the latest rewards. These approaches do not apply neural networks and are only suited for simpler problems with discrete states. More complex problems like the banana navigation project require so-called *deep reinforcement learning* methods, which apply a neural network to store the environment knowledge.

Such a network receives the current state as input and provides scores for each action as output. Internally a neutral network consists of multiple layers with so-called *neurons.* Neurons from one layer are connected to the neurons from another layer. In each neuron, all inputs are considered and an output value is calculated with a non-linear function the so-called *activation function*. This output value is forwarded to all connected neurons in the upcoming layer. This calculation is repeated until the last layer is reached. This is the final output which describes in our cases the values of each action. Out of these, the algorithm selects the best action. During the training, the real rewards are compared to the rewards expected based on the neural network. The agent is trained and improves, because the network is optimized by minimizing the difference between the two values. There are a few extensions that are applied to improve and stabilize the learning performance.

Experience Replay

This extension adds a memory buffer of the past actions, observations and rewards to use these experience tuples multiple times for training. Usually the memory is limited in size and contains the latest events. A random sample is drawn out of the memory buffer and used for training. Additionally, drawing randomly removes the correlation that might appear when training with a sequence of events.

Fixed Q-Targets

There is a correlation when optimizing the same network that is used for selecting the best actions. To avoid this, a second network with the same architecture is introduced. One network is always updated and trained, while the second network is updated less often. This results reduced correlation effects and hence in a smoother learning process.

Double DQN
In the standard approach best next action is selected and evaluated with the same network. This might introduce correlations that reduce the learning performance. With the Fixed Q-targets we have already introduced a second network. The latter can be used to independently select and evaluate the best next action.

My reinforcement learning implementation
The implemented deep reinforcement learning approach is based on PyTorch. All the extensions that were described in the previous chapter are implemented in my solution as well. All the code to load the environment and train the agent is stored in the 'Navigation.ipynb' notebook. This notebook calls an agent defined by the *agent()* class which is stored In the 'dqn_agent.py' file. The file does also contain a class to create the replay buffer. Additionally, there is the 'dqn_model.py' file that contains the class to create the neural network. My neural network architecture is a fully connected network that contains an input layer with 37 neurons, 1 hidden layer with 64 neurons and an output layer with 4 neurons. The *Relu* function is chosen as activation function for the whole network.

Parameters for tuning the agent are defined as standard parameters of the agent, but can also be set manually when creating the agent. The final set of parameters that is used is listed in the table below.

Parameter type	Parameter	Value
Exploration factor	Epsilon start value	1
Exploration factor	Epsilon minimum value	0.0001
Exploration factor	Epsilon decay factor	0.99
Memory buffer setup	Batch size for learning	32
Memory buffer setup	Number of steps between learning	5
Memory buffer setup	Memory buffer size	10000
Learning speed factor	Discount factor	0.99
Learning speed factor	Network learning rate	5e-4
Learning speed factor	Target network update factor	1e-3

Results
Evaluation methods
The score achieved by the agent in a single game episode is the key performance measure. In addition, the agent is considered to perform better if a good score is achieved with less training episodes. However, the score of successive episodes might vary due to the randomness in the game and due to randomness in the decisions of an exploring agent. Therefore, the average score of the last 100 episodes is used for evaluating the agent. In case this score is larger than 13, the environment is considered to be solved.

Agent scores

The training performance of my final agent is shown in Figure 3. The blue distribution shows the score for each episode. This distribution fluctuates strongly. The green line chart is more stable and shows the score values averaged over 100 episode. Based on this score, the agent has needed 448 episode to achieve the required score of at least 13. The final score achieved after 1000 episodes was 15.62.

This is only slightly higher than 500 episodes before, since the agent seems to have reached a nearly optimum strategy in earlier episodes. Hence, running further training with more than 500 episodes does only result in very small improvements.

Plot of rewards

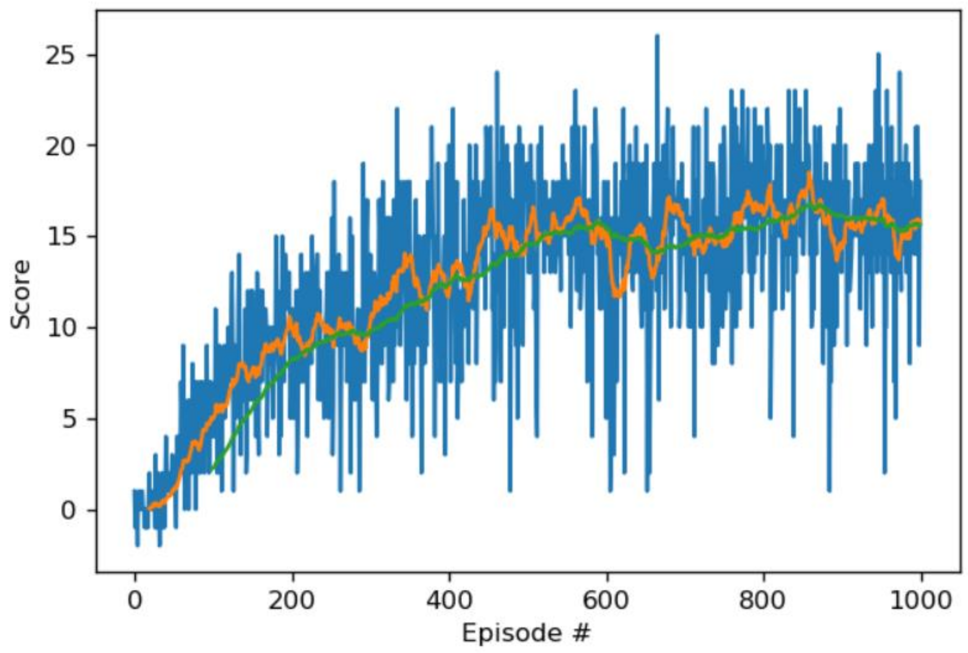

Figure 3: Training scores of the final agent (blue = 1 episode, orange = avg. 20 episodes ,green= avg. 100 episodes)

Extensions for performance improvements

Prioritized experience replay

This is an extension to the experience replay buffer. The main idea is to add a priority value to each to experience tuple that increases the likelihood that a tuple is selected from the sample. This enhances the possibility to learn from rare events or events where the agent takes bad decisions. To avoid learning from a subsample with high priority values only, based on two parameters, some actions are still selected randomly. An implementation of this add-on can speed up the learning process.

Dueling DQN

The concept of dueling networks is based on the idea to have one separate stream to estimate the state value and one to estimate the advantage of each action. This is valid since usually the state value is not strongly correlated to the action value. The final Q-values for state action pairs are retrieved by combining the two streams.

Extended grid search

My code provides already the environment for execution of a grid search, but so far only a small set of parameters was explored due to limited computation time. An exploration using more parameter variations is expected to improve the results even further. However, to avoid extreme computation times the full grid search should be replaced with a random grid search with only testing between 50 and 100 parameter sets.