Report

July 3, 2018

1 Project 1: Navigation

The goal of this project is to train an agent to navigate (and collect bananas!) in a large, square world.

This report provides description of the implementation algorithms.

More detailed description of the project and the corresponding files and procedures is provided in the README

1.0.1 1. Learning Algorithm

Description of the learning algorithm, along with the chosen hyperparameters.

Model architectures for the neural networks The model architecture is simple fully connected neural network with two hidden layers, each with 64 units. The output of all layers except the last one pass through `ReLU` non-linearity.

The model is saved in the file **model.py**. It is and used by the agent. In **agent.py** we have `from model import QNetwork`. Each agent has two such networks, as will be explained below.

Description of the learning algorithm The training algorithm is implemented in the function `dqn()`.

For each step, the agent performs an action

```
action = agent.act(state, eps)
```

then gets information from the environment about the next state, the reward and the game status:

```
env_info = env.step(action)[brain_name]        # send the action to the environment
next_state = env_info.vector_observations[0]   # get the next state
reward = env_info.rewards[0]                   # get the reward
done = env_info.local_done[0]                  # see if episode has finished
```

Next the agent and environment states are updated:

```
agent.step(state, action, reward, next_state, done)
state = next_state
```

The `agent.step` step includes:

- saving into memory the experienced state, action, reward, next state

```
self.memory.add(state, action, reward, next_state, done)
```

- sampling from the memory for training purposes

```
experiences = self.memory.sample()
```

- and learning

```
self.learn(experiences, GAMMA)
```

The Agent learning is implemented in the `learn()` module of the `class Agent()`. The algorithm is Deep Q-Networks (DQN) as described in the DQN paper (https://storage.googleapis.com/deepmind-media/dqn/DQNNaturePaper.pdf)

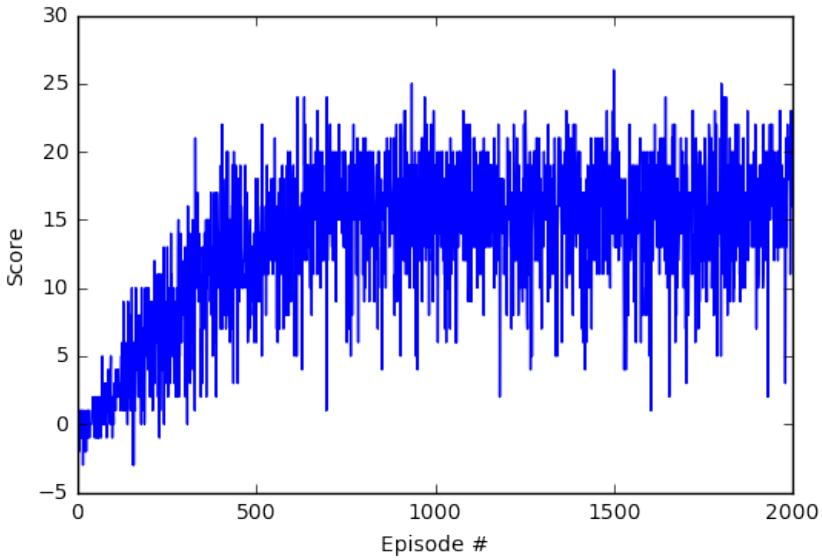
The loss function is defined as the Mean Squared Error (MSE) between **the expected Q-value**, based on the trained local network `self.qnetwork_local` and **the target Q-value**, based on the Bellman equation and a separete network `self.qnetwork_target`. The target network has the same architecture as the local network but its' parameters is updated slower by low-pass filtering the local network parameters ($\theta_{target} = \tau \cdot \theta_{local} + (1 - \tau) \cdot \theta_{target}$) as implemented in the `soft_update()` method.

```
# Get max predicted Q values (for next states) from target model
Q_targets_next = self.qnetwork_target(next_states).detach().max(1)[0].unsqueeze(1)
# Compute Q targets for current states
Q_targets = rewards + (gamma * Q_targets_next * (1 - dones))

# Get expected Q values from local model
Q_expected = self.qnetwork_local(states).gather(1, actions)

# Compute loss
loss = F.mse_loss(Q_expected, Q_targets)

# Minimize the loss
self.optimizer.zero_grad()
loss.backward()
self.optimizer.step()
```

Hyperparameters We have not played with the dafault training hyperparameters:

```
BUFFER_SIZE = int(1e5)  # replay buffer size
BATCH_SIZE = 64         # minibatch size
GAMMA = 0.99            # discount factor
TAU = 1e-3              # for soft update of target parameters
LR = 5e-4               # learning rate
UPDATE_EVERY = 4        # how often to update the network
```

The neural network parameters were not optimized also.



Training Scores

1.0.2 2. Plot of Rewards

A plot of rewards per episode. We can see thatafter 500 episodes there is no further improvement and the average reward (over 100 episodes) is about +15.

Number of episodes needed to solve the environment

We get a target average score of 14 for 460 episodes:

```
Environment solved in 460 episodes! Average Score: 14.00
```

1.0.3 3. Ideas for Future Work

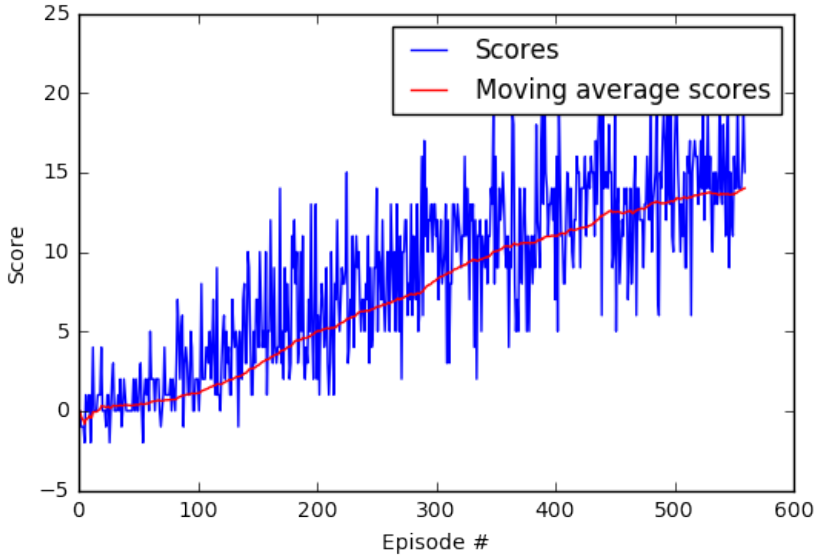
- explore networks with more layers and neurons per layer for better performance;
- explore different training parameters, e.g. `BATCH_SIZE` for faster training;
- implement prioritized experience replay, Double DQN, or Dueling DQN (as recommended in the Deep Q-Networks lesson;
- complete the optional challenge - learn directly from pixels as opposed from the current ray-based perception.

1.0.4 References

Mnih V., Kavukcuoglu K., Silver D., et.all., Human-level control through deep reinforcement learning, Nature, 2015 https://storage.googleapis.com/deepmind-media/dqn/DQNNaturePaper.pdf

Finding moving average from data points in Python, https://stackoverflow.com/a/34387987

Udacity lesson on Deep Q-Networks



Training Scores