udacity Deep Learning Nano Degree Project 1 Navigation – Project Report

data.camp097@audi.de

April 8, 2020

Contents

1 Introduction

This report summarizes my implementation and results for the first project in the udacity Deep Reinforcement Learning Nanodegree. In the first section, the two algorithms used, Deep Q-Learning and Dueling Networks Q-Learning are briefly described, in the second section the results are presented and in the third section a brief outlook on possible future improvements is given.

2 Algorithms

For my solution I implemented two different algorithms, a vanilla Deep Q-Learning algorithm [1] and a dueling networks architecture [4].

2.1 Vanilla DQN

The update of the weights is done by gradient ascent with the following gradient function:

$$\nabla_{\theta_i} L_i\left(\theta_i\right) = \mathbb{E}_{s,a\sim\rho(\cdot);s'\sim\mathcal{E}}\left[\left(r + \gamma \max_{a'} Q\left(s', a'; \theta_{i-1}\right) - Q\left(s, a; \theta_i\right)\right) \nabla_{\theta_i} Q\left(s, a; \theta_i\right)\right] \tag{1}$$

Two key ideas to achieve good practical convergence properties are:

- **Experience Replay.** To alleviate the issue of highly correlated samples from sequential learning, DQN uses a replay buffer that stores the roll-outs. The actual learning is done by randomly samling from this buffer. The buffer will be refreshed periodically with new samples generated under the latest policy.
- **Fixed Q-targets.** The convergence of the learning can effectively be improved by fixing the action value function $Q(s, a; \theta))$ for the calculation of the TD-target for a few steps before updating to the latest set of weights. Instead of updating the Q-target every n-th step I implemented a "soft update" (an incremental step towards the current action value function) in every step.

The architecture for the neural net I used in the implementation is very simple:

```
class QNetwork(nn.Module):

    def __init__(self, state_size, action_size, seed):
        super(QNetwork, self).__init__()
        self.seed = torch.manual_seed(seed)
        self.input_size = state_size
        self.output_size = action_size
        self.hidden_size_1 = 64
        self.hidden_size_2 = 32

        self.fc1 = torch.nn.Linear(self.input_size, self.hidden_size_1)
        self.fc2 = torch.nn.Linear(self.hidden_size_1, self.hidden_size_2)
        self.fc3 = torch.nn.Linear(self.hidden_size_2, self.output_size)

        self.relu = torch.nn.ReLU()

    def forward(self, state):
        """Build a network that maps state -> action values."""
        x = self.fc1(state)
        x = self.relu(x)
        x = self.fc2(x)
        x = self.relu(x)
        x = self.fc3(x)
        return x
```

2.2 Dueling DQN

As one improvement, I additionally implemented a dueling network architecture. A dueling architecture decribes the action value function as the sum of the state value function $V(s)$ and the advantage function $A(s, a)$, where

$$Q(s, a) = V(s) + A(s, a)$$

The motivation behind this is the observation, that many action values are mainly determinded by the corresponding value function. This basic architecture is sketched in figure 1.

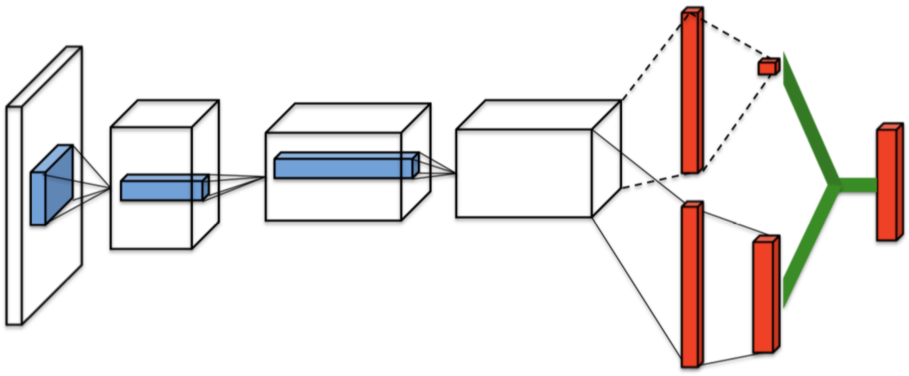

Figure 1: Basic Architecture of the Dueling Network

The neural net I used for the dueling architecture again is very simple:

```
class DDQN(nn.Module):

    def __init__(self, state_size, action_size, seed):
        super(DDQN, self).__init__()
        self.seed = torch.manual_seed(seed)
        self.state_size = state_size
        self.action_size = action_size
        self.common_size = 256
        self.value_size = 128
        self.advantage_size = 128
        self.fc_common = nn.Linear(state_size, self.common_size)
        self.fc_v_in = nn.Linear(self.common_size, self.value_size)
        self.fc_v_out = nn.Linear(self.value_size, 1)
        self.fc_a_in = nn.Linear(self.common_size, self.advantage_size)
        self.fc_a_out = nn.Linear(self.advantage_size, self.action_size)
        self.relu = torch.nn.ReLU()

    def forward(self, state):
        common = self.relu(self.fc_common(state))
        value = self.relu(self.fc_v_in(common))
        advantage = self.relu(self.fc_a_in(common))
        value = self.fc_v_out(value)
        advantage = self.fc_a_out(advantage)
        out = value + advantage - advantage.mean(dim=1).unsqueeze(1)
        return out
```

It consists again only of dense layers. One common hidden layer with size 256 is followed by one additional layer for the value function and one separate layer for the advantage function, both size 128. As non-linearity the ReLU-function was used.

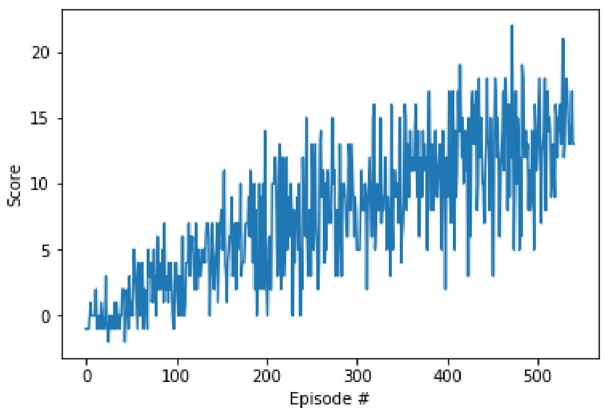

Figure 2: Learning Curve of the vanilla DQN

2.3 Hyperparameters

There was no urgent need for hyperparameter tuning, the default values did ok.

- Size of the replay buffer: 1e5
- Batch size: 64
- Discount factor γ: 0.99
- Soft update τ: 0.99
- Update interval: 4
- Epsilon start: 1
- Epsilon decay: 0.005
- Epsilon minimum: 0.01
- Maximum number of time steps per episode: 1000
- Learning rate: 5e-4

3 Results

This section briefly shows the learning curves of both the implemented algorithms. The learning curve (score as a function of episode number) for the vanilla DQN is shown in figure 2. The training with the vanilla DQN architecture yields:

```
Episode 100 Average Score: 1.10
Episode 200 Average Score: 4.88
Episode 300 Average Score: 7.73
Episode 400 Average Score: 9.76
Episode 500 Average Score: 12.44
```

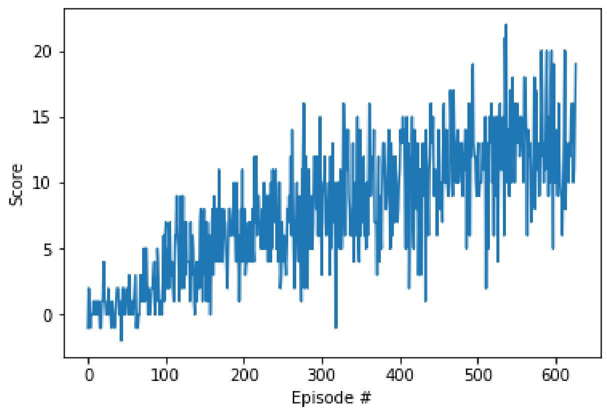

Figure 3: Learning Curve of the dueling DQN

´

```
Episode 541 Average Score: 13.00
Environment solved in 441 episodes! Average Score: 13.00
```

The learning curve (score as a function of episode number) for the dueling DQN is shown in figure 2. The training with the dueling architecture yields:

```
Episode 100 Average Score: 1.14
Episode 200 Average Score: 4.94
Episode 300 Average Score: 7.23
Episode 400 Average Score: 8.90
Episode 500 Average Score: 11.08
Episode 600 Average Score: 12.71
Episode 627 Average Score: 13.03
Environment solved in 527 episodes! Average Score: 13.03
```

A formal comparison of the results is not valid, since there has been almost no hyperparameter optimization in the algorithms and we do not have done enough training runs to get so statistically valid result. The samle of size $n = 1$ indicates that both algorithms solve the environment in a similar number of episodes.

4 Outlook

There have been many ideas put forward to improve the Deep Q-Learning approach. The following list mentions a few popular ones, that each address waeknesses of the Deep Q-Learning approach.

- Double Q-Learning [2]
- Prioritized Experience Replay [3]
- Distributional DQN [5]

- Rainbow [6]
- Multi-Step Bootstrap Targets [7]
- Noisy DQN [8]

It would be interesting, to implement these and try them on the navigation task of this project and compare the results. This would make for an interesting future project. In my personal opinion, conceptionally the most interesting approach is the idea to use the full probability distribution for each action value by Bellmare et al. [5].

References

[1] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller, “Playing atari with deep reinforcement learning,” 2013.

[2] H. van Hasselt, A. Guez, and D. Silver, “Deep reinforcement learning with double q-learning,” 2015.

[3] T. Schaul, J. Quan, I. Antonoglou, and D. Silver, “Prioritized experience replay,” 2015.

[4] Z. Wang, T. Schaul, M. Hessel, H. van Hasselt, M. Lanctot, and N. de Freitas, “Dueling network architectures for deep reinforcement learning,” 2015.

[5] M. G. Bellemare, W. Dabney, and R. Munos, “A distributional perspective on reinforcement learning,” 2017.

[6] M. Hessel, J. Modayil, H. van Hasselt, T. Schaul, G. Ostrovski, W. Dabney, D. Horgan, B. Piot, M. Azar, and D. Silver, “Rainbow: Combining improvements in deep reinforcement learning,” 2017.

[7] V. Mnih, A. P. Badia, M. Mirza, A. Graves, T. P. Lillicrap, T. Harley, D. Silver, and K. Kavukcuoglu, “Asynchronous methods for deep reinforcement learning,” 2016.

[8] M. Fortunato, M. G. Azar, B. Piot, J. Menick, I. Osband, A. Graves, V. Mnih, R. Munos, D. Hassabis, O. Pietquin, C. Blundell, and S. Legg, “Noisy networks for exploration,” 2017.