Project Report

Teaching a Reinforcement Learning Agent to Collect Bananas

May 14, 2020

1 Learning Algorithm

To solve the environment as described in `README.md` we use the Double Deep Q-Learning (DDQN) algorithm [1]. We briefly summarize the algorithm in the following.

We train the agent for a maximum of 2000 episodes or until the environment is solved, where an episode ends after 2000 time steps. In each time step the agent interacts with the environment according to an ϵ-greedy policy and receives an experience tuple $E_i = (S_t^i, A_t^i, R_{t+1}^i, S_{t+1}^i)$, i.e. the current state and chosen action, as well as the resulting reward and next state of the environment. Each experience tuple is stored in the agents replay buffer, which can hold up to `BUFFER_SIZE` samples. After `UPDATE_EVERY` time steps, we sample a batch E of `BATCH_SIZE` experience tuples from the replay buffer and train the agent using this experience.

For DDQN learning happens as follows. We maintain two Q-Networks, the local and the target network, which we use to approximate the state-action value function. In fact, we simply maintain two sets of weights (θ^{local} and θ^{target}) for the same network architecture. We then train the local network by minimizing the mean-squared error loss function

$$L(E) = \sum_{i=1}^{\texttt{BATCH\_SIZE}} (Y_i - \hat{Q}(S_t^i, A_t^i, \theta^{local}))^2 \tag{1}$$

where the targets Y_i are given as

$$Y_i = R_{t+1}^i + \texttt{GAMMA} \cdot \hat{Q}(S_{t+1}^i, \arg\max_a \hat{Q}(S_{t+1}^i, a, \theta^{local}), \theta^{target}) \tag{2}$$

using the Adam [2] optimizer with learning rate `LR`.

After updating the local network, we perform a soft update of the target network towards the local network, i.e.

$$\theta^{target} = (1 - \texttt{TAU})\theta^{target} + \texttt{TAU} \cdot \theta^{local}. \tag{3}$$

In our solution we set $\tau = \texttt{TAU}$.

Hyperparameter	Value
BATCH_SIZE	32
BUFFER_SIZE	10^5
EPSILON_DECAY	1000
EPSILON_INIT	1.0
EPSILON_MIN	0.1
GAMMA	0.95
LR	$5 \cdot 10^{-4}$
TAU	$5 \cdot 10^{-2}$
UPDATE_EVERY	4

Table 1:

2 Implementation

In the following we present key aspects of our implementation. The learning algorithm is implemented in `ddqn_agent.py`, whereas training and evaluation happen in `Navigation_DDQN.ipynb`. The Q-Network architecture is defined in `model.py`.

Model Architecture

We use an architecture with 37 input neurons (i.e. the dimension of the state space), three fully-connected hidden layers with $(64, 32, 16)$ neurons and ReLU activation functions, and a linear output layer of four neurons, one representing the state-action value of each possible action.

ϵ-greedy Policy

During training we select action according to an ϵ-greedy policy where we approximate the state-action values utilizing the local network. We initialize ϵ to `EPSILON_INIT` at the beginning of training. Then we linearly anneal ϵ to a value of `EPSILON_MIN` within `EPSILON_DECAY` episodes. In our experiments, we observed that linear annealing enables a significantly faster performance increase when compared to an exponential decay. We would argue, that especially in early training steps, exploration decays too quickly in the latter case.

Hyper-parameters

The hyper-parameters of our implementation are declared in `ddqn_agent.py`. The hyper-parameter setting we used in our solution are given in Table 1. We note that by reducing the discount factor `GAMMA` from 0.99 to 0.95 sped up training remarkably. We argue, that a more short-sighted, greedy behavior fits the given task very well, since no sophisticated strategy is required to pick up the bananas (e.g. move towards the closest yellow banana, avoid any blue bananas at any time step...).

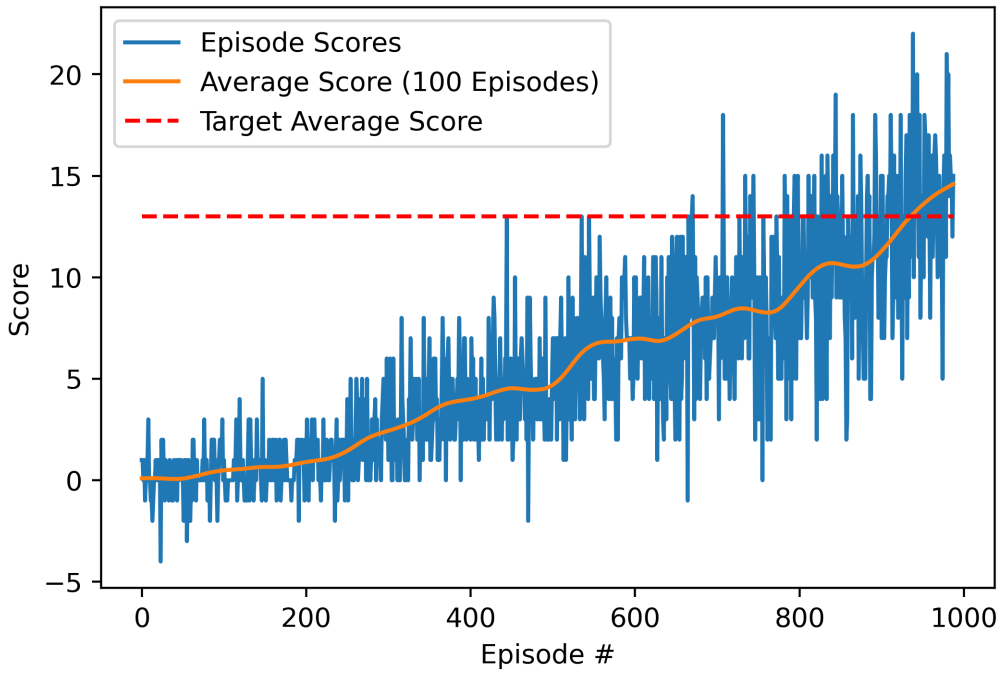

Figure 1: Episode scores of the DDQN agent during training.

3 Results

Our DDQN agent was able to solve the environment after **888 episodes** (the average score of episode 888 to 988 is greater of equal than +13). The corresponding model weights are stored in `./checkpoints/checkpoint_888.pth`. Figure 1 shows the evolution of the agents scores during training.

4 Ideas for Future Improvements

To further improve the learning time of the agent (we assume performance requirements in terms of return as satisfied), we would suggest the following extensions.

- **Prioritized Experience Replay**: In order to learn more efficiently it could help to employ Prioritized Experience Replay [3]. The key idea here is that experience tuples that yield a high training loss are more valuable for learning and hence should be sampled more frequently from the replay buffer.
- **Dueling Deep Q-Networks**: An easy-to-implement extension for our agent would be the implementation of a dueling architecture for the Q-networks as proposed in [4].

References

[1] Van Hasselt, Hado, Arthur Guez, and David Silver. "Deep reinforcement learning with double q-learning." Thirtieth AAAI conference on artificial intelligence. 2016.
[2] Kingma, Diederik P., and Jimmy Ba. "Adam: A method for stochastic optimization." arXiv preprint arXiv:1412.6980 (2014).
[3] Schaul, Tom, et al. "Prioritized experience replay." arXiv preprint arXiv:1511.05952 (2015).
[4] Wang, Ziyu, et al. "Dueling network architectures for deep reinforcement learning." arXiv preprint arXiv:1511.06581 (2015).