Navigation Project

FP Steiner

Abstract—Deep Reinforcement Learning is used to train an agent in a Unity environment to pick up certain objects while avoiding others.

Index Terms—Deep Reinforcement Learning, OpenAI, Unity.

1 Introduction

Let there be an agent sitting in the center of a large square world cluttered with yellow and blue bananas. The goal of the agent is to collect as many of the yellow bananas as possible while avoiding the blue ones. For this, the agent can move forward or backward and turn left or right.

This project repository is an implementation of a simplified version of the Banana Collector environment of the Unity ML Agents Toolkit [1].

Fig. 1: A single frame from a short video sequenc of the trained agent in action.

2 Model Configuration

2.1 Neural Network

The observation space is derived from a two-dimensional world. The agent's velocity is a tuple (v_x, v_y). The agent emits 7 beams with each reporting one of five possible observations: *yellow banana, blue banana, wall, other agent, distance* [1].

The input layer, thus, has 37 neurons corresponding to the 37 possible observation states.

Using two hidden layers with 64 neurons, each activated by a RELU function, the input layer is mapped to the four output logits corresponding to the four possible actions *forward, backward, left and right*.

1. Because in the studied setup only one agent is present, no other agent will ever be detected, and the observation state space could be reduced to $2 + 7 \cdot 4 = 30$.

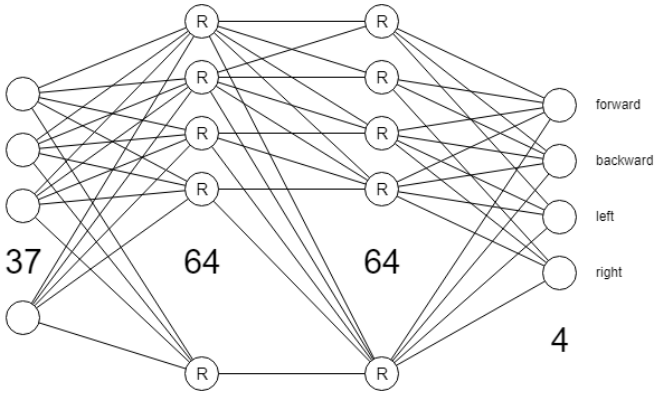

Fig. 2: Neural network mapping the 37 input states to the four possible actions via two activated hidden layers.

2.2 Learning Algorithm

2.2.1 Objective

The agent interacts with its environment by observing it, choosing an action, and subsequently getting a reward. The goal of the agent is to select actions that maximize cumulative future rewards which corresponds to the maximum sum of rewards at each time step discounted by a discount factor γ.

2.2.2 Experience Replay

Experience replay is used to randomize over the data to remove possible correlations in the observation sequence. With this, the agent is not learning from what it just did but from earlier situations. Each situation or experience is a tuple consisting of an observation, a chosen action, a reward, and the subsequent observation.

For experience replay, a circular buffer of such tuples is used, and a minibatch of experiences is drawn uniformly at random. The smaller the minibatch size is, the higher the variance of the gradient estimates and the slower the learning. The advantage of a smaller minibatch size, however, is a better screening of the potential outcomes and, thus, a better exploration of the goal function.

2.2.3 Soft Update

To add stability to the learning, the network is not frozen every couple of time steps but rather smoothly updated by only adding a fraction of the current value to the target

via Polyak averaging or a similar method. The smoothing parameter is τ.

2.2.4 Exploration vs. Exploitation

Clearly there is a trade-off between exploration and exploitation. One of the simplest algorithms to sample between a random experimentation and the instinct to maximize the currently achievable rewards is the ϵ-Greedy algorithm. ϵ represents the amount of exploration and, thus, the randomness in actions selections.

Initially, more exploration is necessary, but the more the agent has explored already, the more it has to exploit what it has learned already. This is realized by means of an ϵ-decreasing policy with ϵ exponentially decreasing from an initial value ϵ_0 to an end value ϵ_n.

2.3 Hyper-parameters

The hyper-parameters used are shown in Table 1. They worked out of the box.

Hyper-parameter		**Constant Name**	**Value**
Discount factor	γ	GAMMA	0.99
Replay buffer size		BUFFER_SIZE	10^5
Minibatch size		BATCH_SIZE	64
Soft update parameter	τ	TAU	10^{-3}
Learning rate		LR	$5 \cdot 10^{-4}$
Target update rate		UPDATE_EVERY	4
Initial ϵ	ϵ_0	eps_start	1.0
Final ϵ	ϵ_n	eps_end	0.01
ϵ decay	δ	eps_decay	0.995

TABLE 1: Hyper-parameters

2.4 Training

For training of the agent, the environment was first reset in train mode. Then, an agent was instantiated for the relevant state and action space size of 37 and 4, respectively.

For a maximum of 2000 episodes of 1000 time-steps each, the agent was to choose an action based on the current step, get a reward and the next state until the average reward over the past 100 episodes exceeded a set threshold. The project rubric required the threshold to be 13, but a slightly more challenging level of 15 was used.

3 Results

Using the hyper-parameters as described in Table 1 and the training setup descibed above, the environment was solved. The following is a typical example of a training run completing the task in 645 episodes with an average score over the last 100 episodes of 15.03.

```
   Episode 100 Average Score: 1.00
   Episode 200 Average Score: 4.14
   Episode 300 Average Score: 7.55
   Episode 400 Average Score: 10.66
   Episode 500 Average Score: 13.34
   Episode 600 Average Score: 13.40
   Episode 700 Average Score: 14.55
   Episode 745 Average Score: 15.01
   Environment solved in 645 episodes!
Average Score: 15.01
```

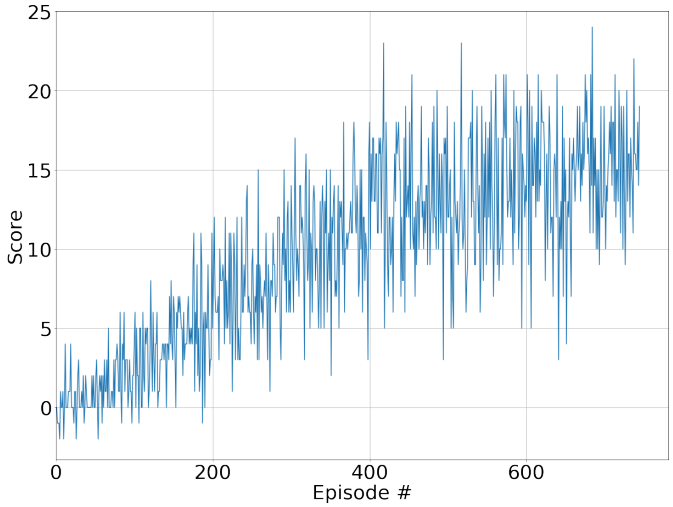

Fig. 3: Scores by episode. Training was completed once the score — averaged over the past 100 episodes — reached 15.

4 Discussion

The project requirements were successfully met. The project provided a good way to revisit the DeepQ Learning algorithm, ϵ-decreasing policy and experience replay. Setting up the environment on a virtual machine running Ubuntu was not without problems. Initially, the Unity environment kept crashing, but after updating some of the packages everything worked.

5 Future Work

Several fields could be investigated for further improvements of the results:

- Current network architecture optimization: deeper and/or wider architectures could be studied, together with optimization techniques such as batch normalization, dropout and different weight initialization specifically targeting the insignificant inputs due to the fact that there is only one agent and, thus, 7 out of the 37 input dimensions are always zero.
- Alternative network architectures: Double DQN [2], dueling DQN [3] or even recurrent neural networks (RNNs) such as LSTM (Long Short Term Memory) [4] could be applied.
- Prioritized Experience Replay (PER): because the batch is sampled uniformly and, thus, the experiences selected randomly, rich but infrequent experiences have little chance of being selected. PER assigns priority to experiences where there is a big difference between the prediction and the target.
- As an additional challenge, the ray-based perception leading to the 37-dimensional input state space could be replaced by learning directly from the raw pixels of the agent's FPV (first person view) image.

REFERENCES

[1] "Unity GitHub page, ml-agents repository." http://bit.ly/2yXg3Xw. [Online, accessed 2-November-2018].

[2] H. v. Hasselt, A. Guez, and D. Silver, "Deep reinforcement learning with double q-learning," in *Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence*, AAAI'16, pp. 2094–2100, AAAI Press, 2016.

[3] Z. Wang, T. Schaul, M. Hessel, H. Van Hasselt, M. Lanctot, and N. De Freitas, "Dueling network architectures for deep reinforcement learning," in *Proceedings of the 33rd International Conference on International Conference on Machine Learning - Volume 48*, ICML'16, pp. 1995–2003, JMLR.org, 2016.

[4] B. Bakker, "Reinforcement learning with long short-term memory," in *Proceedings of the 14th International Conference on Neural Information Processing Systems: Natural and Synthetic*, NIPS'01, (Cambridge, MA, USA), pp. 1475–1482, MIT Press, 2001.