Navigation Using Deep Reinforcement Learning

Uirá Caiado

According to [4], Reinforcement Learning (RL) is an area of machine learning that studies how a learner, called agent, interacts with an environment in order to maximize some notion of cumulative reward. As pointed out by [2], to use RL successfully in complex environments, with high-dimensional state space, the agent must derive efficient representations of these inputs and use them to generalize past experience to new situations.

In this project, I trained an agent to navigate in a vast, square world, using Unity Machine Learning Agents Toolkit[1] to design, train, and evaluate Deep Reinforcement Learning algorithms. These algorithms combine RL with a class of artificial neural network, known as deep neural network, to generalize its past experiences to new ones. I considered to the tests the algorithms Deep Q-Learning (DQN) [2], Double Deep Q-learning (DDQN) [5] and DDQN with prioritized experience replay (PER) [3].

The environment used for this project is the Udacity version of the Banana Collector environment, from Unity[2]. The goal of the agent is to collect as many yellow bananas as possible while avoiding blue bananas. The task is episodic, and to solve the environment, the agent must get an average score of +13 over 100 consecutive episodes.

A reward of +1 is provided for collecting a yellow banana, and a reward of -1 is provided for collecting a blue banana. The state space has 37 dimensions and contains the agent's velocity, along with a ray-based perception of objects around agent's forward direction. Given this information, the agent has to learn how to best select actions. Four discrete actions are available, corresponding to moving forward, backward, turn left and right.

All the algorithms implemented in this project used the following architecture to the deep Q-networks used. The input to the neural networks consists of an vector 37×1 produced by the environment. The input layer is a fully-connected linear layer with 128 neurons and applies a rectifier nonlinearity. It is followed by a hidden layer also consisting of a fully-connected linear layer with 128 neurons followed by another rectifier. The output layer is also a fully-connected linear layer with a single output for each action.

The first algorithm tested was the vanilla DQN that was first introduced by [2]. As explained in the paper, the basic idea of many RL algorithm is to estimate the action-value function by using the Bellman equation as an iterative update, such that

$$Q^*(s,\, a) = \mathbf{E}_{s'}\left[r + \gamma \max_{a'} Q^*(s', a') |\, s,\, a\right] \,,$$

where the variables s and a stands for state and action. In practice, it is common to use a function approximator to estimate the action-value function, $Q(s, a;\, \theta) \approx Q^*(s,\, a)$, as a linear or a nonlinear function approximator, such as tile-coding and neural networks. In the paper, the neural network function approximator, called Q-network, with weights θ, is trained by adjusting its parameters to reduce the mean-squared error in the Bellman equation presented above. The Q-values $Q^*(s', a')$ in the equation are substituted with approximate Q-values $Q(s', a';\, \theta^-)$, using parameters θ^- from some previous iteration.

The paper modified the standard online Q-learning in two ways to make it suitable for training large neural networks without diverging. First they used a experience replay buffer to store the agent's experiences at each time-step, $e_t = (s_t, a_t, r_t, s_{t+1})$, in a dataset $D_t = \{e_1, \cdots, e_t\}$. These experiences are drawn at random from the buffer to apply the Q-learning updates. Among the advantages cited in the paper, the authors suggested that learning from consecutive samples is inefficient, owing to the strong correlations between the samples, and randomizing the samples breaks

[1] Source: `https://github.com/Unity-Technologies/ml-agents`
[2] Source: `https://youtu.be/heVMs3t9qSk`

the correlations and reduce the variance of the updates. The second modification introduced was to use a separate network with parameters θ^- for generating the targets in the Q-learning update, or mini-batch updates, and cloning every C updates the parameters of the network $Q(s, a;\ \theta)$ to obtain the target network $Q(s, a;\ \theta^-)$.

In this project, I introduced more two modifications. Fist, as suggested by Udacity, instead of cloning the current network to the target network, I used a "soft-update" to make the target network slightly closer to the $Q(s, a;\ \theta)$ using $\theta^- = \tau\theta + (1 - \tau)\theta^-$. It guarantees that the target network is always different enough from the current "local" network. I used $\tau = 0.001$. The other modification I introduced that was particularly helpful was clipping the reward to be between -1 and 1.

I also used a buffer size of $100,000$ experiences, a mini-batch size of 64, $\gamma = 0.99$, the learning rate of 0.0005 and updated the Q-networks every four steps. Both local and target Q-networks presented the same architecture, described before. Using this configuration, I was able to solve the environment in 529 episodes. The figure 1 presents the results of the simulation. The left panel exposes the rewards obtained in each episode and, in the right panel, I plotted a moving average of the last 100 episodes of this rewards. The shaded area corresponds to one standard deviation σ of the rewards of those 100 episodes.

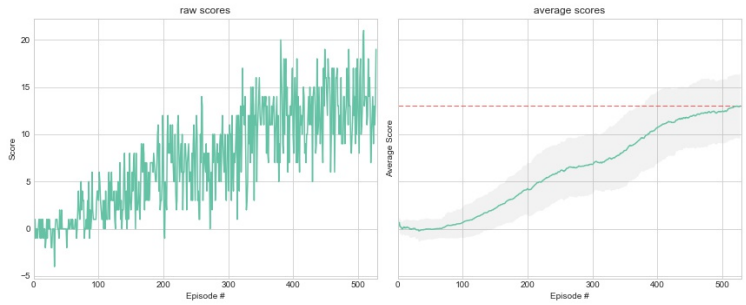

Figure 1: Learning Curve(left panel) and Moving-average(right panel)

The second model implemented and tested was the Double DQN (DDQN), that was first introduced by [5]. As explained by the authors, the max operator in DQN uses the same values both to select and to evaluate an action, what makes it more likely to select overestimated values. Thus, the paper decomposes the action selection and the action evaluation by using the target network to evaluate the action chosen by the local network, such that

$$Y_t = r + \gamma Q(s', \arg\max_{a'} Q(s', a'; \theta); \theta^-)\ ,$$

where Y_t corresponds to the target value. Both networks must agree over the best action to the Q-value returned to be the maximum one.

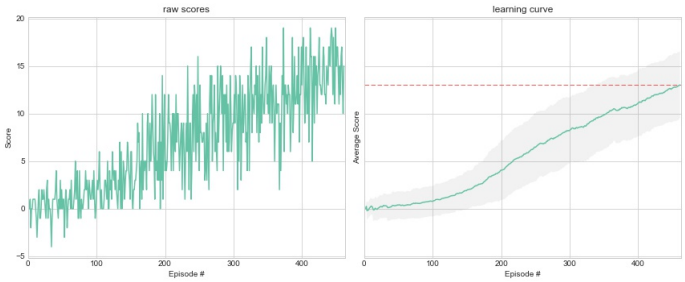

Figure 2: Learning Curve(left panel) and Moving-average(right panel)

I used the same configurations of the tests with DQN and the model implemented was able to solve the task in 462 episodes. The figure figure 2 above presents the results.

The last model tested was the prioritized experience replay presented by [3] and built on top of DDQN. Their key idea, well summarized by [6], was to increase the replay probability of experience tuples that have a high expected learning progress, measure by the absolute TD-error of the step that collected that experience.

The figure 3 presents the performance of the DDQN with PER. I used the same hiper-parameters of the previous model and includes the PER parameters importance sampling exponent $\alpha = 0.6$ and prioritization exponent $\beta = 0.4$. The values were chosen based on the original paper. The algorithms were able to solve the environment in 562 episodes.

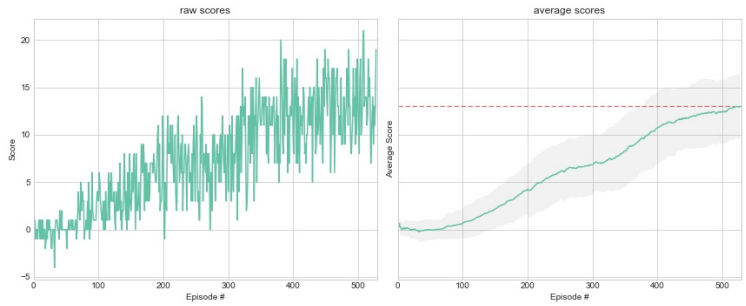

Figure 3: Moving Average of 100 episodes of the score

The table 1 below presents the final performance of each algorithm tested. The error margins were computed using the standard deviation of the distribution of the final score over the last 100 episodes. The figure 4 presents a comparison of the moving average of 100 episodes of the scores achieved by each model.

model	episodes	Score
DDQN with PER	562	13.06 ± 3.47
DDQN	462	13.01 ± 3.48
DQN	529	13.01 ± 3.42

Table 1: Summary of the tests

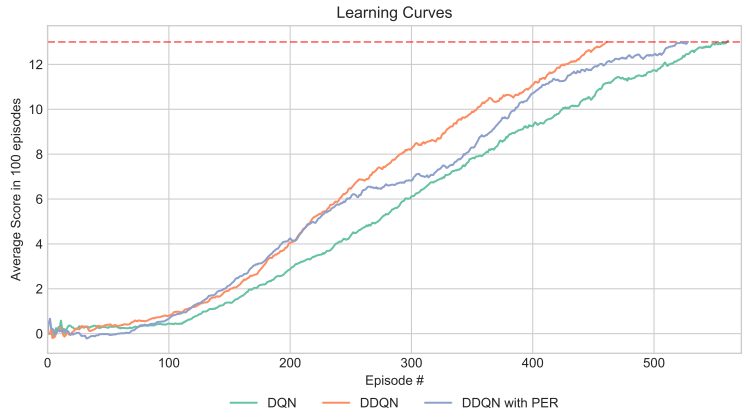

Figure 4: Moving Average of 100 episodes of the score

Finally, as some possibles extensions, the parameters of the models could be better tuned and another models could be added to the analysis, as the Dueling network, presented by [6], or the Rainbow, presented by [1], that combines different improvements in Deep Reinforcement Learning, including the models presented in this project.

References

[1] Matteo Hessel, Joseph Modayil, Hado van Hasselt, Tom Schaul, Georg Ostrovski, Will Dabney, Dan Horgan, Bilal Piot, Mohammad Azar, and David Silver. Rainbow: Combining Improvements in Deep Reinforcement Learning. *arXiv.org*, October 2017.

[2] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A. Rusu, Joel Veness, Marc G. Bellemare, Alex Graves, Martin Riedmiller, Andreas K. Fidjeland, Georg Ostrovski, Stig Petersen, Charles Beattie, Amir Sadik, Ioannis Antonoglou, Helen King, Dharshan Kumaran, Daan Wierstra, Shane Legg, and Demis Hassabis. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529–533, February 2015.

[3] Tom Schaul, John Quan, Ioannis Antonoglou, and David Silver. Prioritized experience replay. *CoRR*, abs/1511.05952, 2015.

[4] Thomas Lawrence Spooner. Reinforcement learning for high-frequency market making in limit order book markets. Master's thesis, the University of Liverpool, 2016.

[5] Hado van Hasselt, Arthur Guez, and David Silver. Deep reinforcement learning with double q-learning. *CoRR*, abs/1509.06461, 2015.

[6] Ziyu Wang, Tom Schaul, Matteo Hessel, Hado van Hasselt, Marc Lanctot, and Nando de Freitas. Dueling Network Architectures for Deep Reinforcement Learning. *arXiv.org*, November 2015.