Deep Reinforcement Learning Nanodegree Project 1 - Navigation

Elias Martins Guerra Prado

Abstract—This project uses deep reinforcement learning (DRL) techniques to train an agent to collect bananas in a square world. The agent has four action options to take, walk forward, backward, left, or right. For every yellow banana collected, the agent gets a +1 reward, and for every purple banana collected -1. The goal of the project is to train an agent to get an average score of +13 in 100 consecutive episodes. To this end, in this work we implement four different agents based on DRL techniques: Deep Q-Networks (DQN), Prioritized Experience Replay (PER) DQN, PER Double DQN, and PER Dueling Double DQN. After training, the performance of the agents was compared in order to evaluate which agent was able to achieve the goal in the least number of episodes, and which obtained the maximum score after 2000 episodes.

Index Terms—Deep Reinforcement Learning, Dueling Q-Networks, Double Q-Networks, Prioritized Experience Replay

1 Introduction

Deep reinforcement learning (DRL) is an amazing subfield of artificial intelligence that combines reinforcement learning with deep learning. The goal in DRL is to train an agent to make decisions based on its past experiences. Deep Q-Learning [1] is one of the pioneers DRL algorithms, and attracted a lot of attention in recent times. The algorithm developed by Google DeepMind team can learn to play Atari 2600 games at expert human level. One of the main advances of the work was to use a Convolutional Neural Network (CNN; type of deep artificial neural network architecture) as a approximation of the optimal action-value function. In Reinforcement Learning (RL), the optimal action-value function is also called the optimal Q-Function. It specifies how good it is for an agent to perform a particular action in a state following the optimal policy. Train an agent in RL means to optimize the Q-Function in order to increase the agent reward. However, compute the Q-Function can be a hard step, especially when the environment has continuous states and/or actions that can not be discredited in a matrix. For this reason, the Q-learning approach deserves attention. By using a CNN to compute the Q-Function, many of the problems and limitations of traditional RL algorithms have been solved.

Several improvements to the original Deep Q-Learning algorithm have been suggested. One important aspect of the original Deep Q-Learning algorithm is the use of an experience replay memory which lets the RL agent remember and reuse experiences from the past by sampling uniformly from the memory. Tom Schaul et. al (2015) [2] proposed the use of a Prioritized Experience Replay (PER) memory intead of the original replay memory. In contrast with the original formulation, which simply replays transitions at the same frequency that they were originally experienced, the PER prioritize important transitions when sampling, repeating them more frequently, increasing the learning efficiency. The PER DQN implemented in the Tom Schaul's paper outperforms the original DQN algorithm in most of the Atari games. Van Hasselt et. al (2016) [3] showed that Deep Q-Learning tends to overestimate action values, and proposed a new algorithm called Double DQN (DDQN) to deal with this problem. The algorithm uses two action-value functions, one to determine the greedy policy and the other to determine its value. By doing this, the algorithm can find better policies then the original DQN algorithm, surpassing its performance. In addition to proposals for improvements in agent training, such as the ones just mentioned, some authors have also proposed improvements in the architecture of the deep neural network (DNN) used in the DQN algorithm. Ziyu Wang et. al (2016) [4] developed a DNN architecture called Dueling Network that has two separate estimators, one for the state value function, as in the original DQN paper, and one for the state-dependent action advantage function. The developed network can generalize learning across actions, allowing to assess the value of each state, without having to learn the effect of each action. The results of Wang's work showed that the Dueling network architecture outperform the original DQN, mainly because it leads to better policy evaluation when dealing with similar-valued actions.

This work is part of Udacity's Deep Reinforcement Learning Nanodegree course, and presents the submission of the first project of the course. In this project a RL agent must be trained to collect yellow bananas in a squared world. The agent has four action options to take, walk forward, backward, left, or right. For every yellow banana collected, the agent gets a +1 reward, and for every purple banana collected -1. The goal of the project is to train an agent to get an average score of +13 in 100 consecutive episodes. To this end, in this work we implement and compared the performance of four different agents based on DRL techniques: DQN, PER DQN, PER DDQN, and PER Dueling DDQN.

2 Background

This section presents a brief introduction to the methods.

2.1 DQN

To train the agent in original the DQN implementation [1], at each time step the algorithm uniformly sample transitions (state, action, reward, next state) from the replay memory, and use these transitions to update the weights of the Q-Network (approximation of the Q-Function) represented by a DNN. The loss function used to update the DNN is defined as:

$$\mathrm{L}_i(\theta_i) = \mathbb{E}_{(s,a,r,s')\sim U(D)}\left[\left(r + \gamma \mathrm{max}_{a'}\, Q(s', a'; \theta_i^-) - Q(s, a; \theta_i)\right)^2\right] \quad (1)$$

in which s is the actual state, a is the action taken given the state s, r is the reward obtained by taking action a, γ is the reward discount factor rate, θ_i are the parameters of the Q-network at iteration i and θ_i^- are the network parameters used to compute the target at iteration i. The target network parameters θ_i^- are only updated with the Q-network parameters (θ_i) every C steps and are held fixed between individual updates. According to the authors, using a distinct network for computing the target which is only periodically updated, reduces the correlation between the target and the actual action-value (Q) improving the learning performance.

2.2 PER

The PER algorithm [2] uses the same training procedure of the DQN algortihm. However, instead of sampling uniformly from the replay memory at each time step, the transitions with high expected learning progress are sampled more frequently. The expected learning progress is measured by the magnitude of the temporal-difference (TD) error, defined as:

$$\delta_i = r + \gamma \max_{a'} Q(s', a'; \theta_i^-) - Q(s, a; \theta_i)$$

where δ_i is the TD-error at interaction i.

In order to sample transitions from the replay memory a stochastic sampling method is used, where the probability of sampling transition i is defined as:

$$P(i) = \frac{p_i^\alpha}{\sum_k p_k^\alpha}$$

where $p_i > 0$ is the priority of transition i. The exponent α determines how much prioritization is used, with $\alpha = 0$ corresponding to the uniform case (without prioritization).

For the proportional prioritization variant of PER (implemented in this work), $p_i = |\delta_i| + \epsilon$, and ϵ is a small positive constant that prevents the probability p_i reach zero. In order to optimize the sampling process, the authors of the PER paper suggests using a 'sum-tree' data structure which can be a efficiently way of calculating the cumulative sum of priorities, allowing $O(\log N)$ updates. The 'sum-tree' structure consists of leaf nodes, which store the transition priorities, and internal nodes that are intermediate sums of the priorities, with the parent root node containing the sum over all priorities, p_{total}. When sampling, the range $[0, p_{total}]$ is divided into k ranges, where k is the minibatch size, and then a transition is uniformly sampled from each range. This is a form of stratified sampling that has the added advantage of balancing out the minibatch (there will always be exactly one transition with high magnitude, one with medium magnitude, etc).

The prioritized replay procedure introduce bias to the learning algorithm. To correct this bias the loss function of the original DQN algorithm is changed to:

$$L_i(\theta_i) = \mathbb{E}_{(s,a,r,s')\sim U(D)}\left[\left(w_i \delta_i^{DDQN}\right)^2\right]$$

where w_i is the importance-sampling weight at interaction i, defined as:

$$w_i = \frac{(N.P(i))^{-\beta}}{\max_i w_i}$$

where N is the total number of transitions stored in the replay memory, and the exponent β determines the correction magnitude, with $\beta = 1$ corresponding to the fully compensation for the non-uniform probabilities $P(i)$.

2.3 DDQN

The DDQN algorithm [3] modifies the TD-error of the original DQN algorithm, changing it to:

$$\delta_i^{DDQN} = r + \gamma Q(s', \underset{a}{\mathrm{argmax}}\, Q(s', a; \theta_i); \theta_i^-) - Q(s, a; \theta_i)$$

and the loss function to:

$$L_i(\theta_i) = \mathbb{E}_{(s,a,r,s')\sim U(D)}\left[\left(\delta_i^{DDQN}\right)^2\right] \quad (2)$$

2.4 Dueling DQN

The Dueling DQN algorithm [4] maintain the loss function of the original DQN implementation, as well as the training procedure. The difference relies on the Q-Network architecture, which explicitly separates the representation of the Q-Function into state-values and action-advantages. The dueling architecture maintains the initial hidden layers of the original Q-Network, while changing the latest fully connected layers by two stream that represent the state-value and action-advantage functions. The state-value stream output a scalar $V(s; \theta, \sigma)$, and the action-advantage stream output a vector $A(s, a; \theta, \rho)$ with dimension equal to the action space size. Here, θ denotes the parameters of the common hidden layers, while σ and ρ are the parameters of the two streams of fully-connected layers.

The two streams are then combined by an aggregating module so the output of the Dueling network is a Q-Function, as follows:

$$Q(s, a; \theta, \sigma, \rho) = V(s; \theta, \sigma) + \left(A(s, a; \theta, \rho) - \frac{1}{|A|}\sum_{a'} A(s, a'; \theta, \rho)\right) \quad (3)$$

where $|A|$ is the total number of actions in the environment action space.

3 IMPLEMENTATION

As mentioned, in this work four different agents were implemented in order to solve the project environment (get an average score of +13 in 100 consecutive episodes). The DQN agent was implemented based on the original DQN algorithm, changing only the hidden layers type of the Q-Network. The DDQN agent algorithm is basically the same of the DQN agent, differing only on the loss function, where the former uses equation (2) and the latter uses equation (1). The Q-Network for the DDQN agent was the same used for the DQN agent. The PER DDQN agent was implemented based on the original PER proportional prioritization variant. Thus a 'sumtree' algorithm was also implemented as in the original paper in order to optimize the sampling procedure. The Q-Network for the PER DDQN agent was the same used for the DQN and DDQN agents. The PER Dueling DDQN agent uses a different Q-Network architecture, based on the Dueling DQN paper, other than that the algorithm is the same of the PER DDQN agent.

3.1 Q-Network

The state of the environment provided in the project is represented by a two dimensional vector, therefore, the hidden layers of the Q-Networks implemented in this work are fully-connected layers (Linear layers) instead of convolutional layers as in the original DQN paper. The Q-Network implemented has a flexible architecture where the first and last hidden layers are fixed and the layers between them are increased ore decreased by a hiperparameter. The first hidden layer has an input size equal the size of the state space, and an output size equal F, and the last hidden layer has an input size of F and an output size equal the size of the action space. The network has $2L$ hidden layers between the first and last layers, with the input and output size for the first L layers before the first fixed layer given as:

$$input_l = F^{2^{l-1}}$$

$$output_l = F^{2^l}$$

and for the last L layers given as:

$$input_l = F^{2^{2L-l+1}}$$

$$output_l = F^{2^{2L-l}}$$

where $input_l$ and $output_l$ are input and output size for the hidden layer l, which ranges between $[1, 2L]$. F and L are network hiperparameters. All the hidden layers has ReLU activation except the last one. The Q-Networks were implemented using the PyTorch Python library.

The hiperparameters of the Q-Networks were selected by doing a grid-search. The grid-search uses the DQN agent for training, where the number of episodes to reach the goal of the environment is used as a measure of the Q-Network performance. The fewer the number of episodes better. For F the values 16, 32, 64, 128, and 256 were tested, and for L the values 1 and 2. All the possible combinations of these values were used to train the agent. The best performing hiperparameters were $F = 256$ and $L = 1$, achieving the goal in XXX episodes.

3.2 Training Algorithm

All the four agents were implemented in the same algorithm, where the agent to be trained is chosen by the function parameters. In this way, all the four agents have a common implementation based on the original DQN algorithm. This base implementation executes the following steps. The agent observes the environment state then selects and executes actions according to an ϵ-greedy policy based on the Q-Network output. The agent experiences (current state, action, reward, next state) are stored in the replay memory at each time-step. When the number of experiences on the replay memory is equal or larger then the batch size, the agent starts updating the Q-Network weights. For this, at each time step the agent uniformly sample from the replay memory N samples, where N is equal to the batch size. These samples are then used to compute the loss and the TD-error (for PER). If using PER, the algorithm uses the computed TD-errors for update the probabilities of the 'sumtree' nodes, and correct the loss. The weights of the Q-Network (local) are then updated by stochastic gradient descent using the Adam algorithm. The target Q-Network weights are updated using soft update at each time-step, as follows:

$$\theta_i^- = \tau * \theta_i + (1 - \tau) * \theta_i^-$$

where θ_i^- and θ_i are the target and local network weights at time step i, respectively. τ is a constant between $[0, 1]$ that determine the magnitude of the update, with $\tau = 1$ corresponding to the case where the weights of the local network are copied to the target network.

The hiperparameters used in all the experiments are showed in Table 1. The buffer size for both the replay memory without prioritization and for the PER are set to 100,000. In the PER implementation, the hiperparameter α used to compute the prioritization probabilities was set to 0.6, and β, used to compute the loss correction factor, was set to 0.4 and increased until it reaches 1 at the end of learning. The batch size was set to 64. The reward discount rate γ, was set to 0.99. The learning rate for the Adam algorithm was set to 0.0005 for the DQN and DDQN agents, and 0.0001 for the PER DDQN and PER Dueling DDQN agents. The soft update constant τ, was set to 0.001. Finally, in order to increase the learning efficiency the agents were set to learn (update the Q-Network weights) every k time steps, where k was set to 4. In this way, the agent take less time to finish an episode, because running the algorithm without learning requires less computation, allowing the agent to finish more episodes without significantly increasing the run-time. During training the agents were limited to make a maximum of 1000 action at each episode, and the total number of episodes was set to 2000.

4 RESULTS AND DISCUSSION

Figures 1 to 4 shows the cumulative score for the implemented agents after each episode, as well as the moving average over 100 episodes. The DQN agent (Figure 1) obtained the second best performance, reaching the cumulative score of +13 after 589 episodes and a maximum average score over 100 episodes of +16.52. The DDQN agent (Figure 2)

TABLE 1: Hyperparameters used for training the implemented agents

Hyperparameter	DQN	DDQN	PER DDQN	PER Dueling DDQN
Replay Memory Buffer Size	100,000	100,000	100,000	100,000
α (PER exponent)	-	-	0.6	0.6
β (PER correction)	-	-	$0.4 \rightarrow 1$	$0.4 \rightarrow 1$
Batch Size	64	64	64	64
γ (reward discount rate)	0.99	0.99	0.99	0.99
Exploration ϵ	$1 \rightarrow 0.01$	$1 \rightarrow 0.01$	$1 \rightarrow 0.01$	$1 \rightarrow 0.01$
Exploration ϵ decay	0.995	0.995	0.995	0.995
Optimizer	Adam	Adam	Adam	Adam
Optimizer: Learning Rate	0.0005	0.0005	0.0001	0.0001
Optimizer: β_1	0.9	0.9	0.9	0.9
Optimizer: β_2	0.999	0.999	0.999	0.999
Optimizer: ϵ	10^{-8}	10^{-8}	10^{-8}	10^{-8}
τ (soft update)	0.001	0.001	0.001	0.001
k (update weights every)	4	4	4	4
Max actions per episode	1000	1000	1000	1000

obtained the worst performance, reaching the environment goal after 697 episodes and obtaining a maximum average score over 100 episodes of +16.64. The PER DDQN agent (Figure 3) performed slightly better then the DDQN agent, solving the environment in 672 episodes and obtaining a maximum average score over 100 episodes of +15.63. Last but not least, the PER Dueling DQN agent obtained the best performance, as expected, needing only 501 episodes to achieve the environment goal, reaching a maximum average score over 100 episodes of +16.33. This ranking, as can be observed, was based only on the number of episodes needed for the agent to solve the environment. However, if comparing the maximum average score obtained over 100 episodes, the DDQN agent, which needed the largest number of episodes to achieve the goal, obtained the largest maximum average score of +16.64, followed by the DQN, PER Dueling DDQN and PER DDQN, respectively (all the agents were trained for a maximum of 2000 episodes).

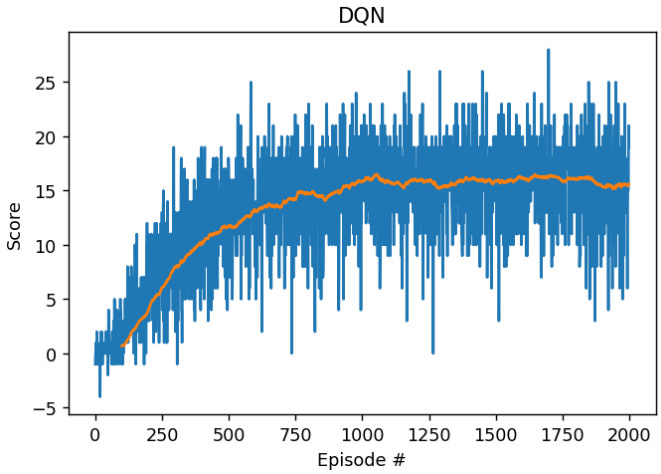

Fig. 1: DQN learning performance on the Unity Banana Collection environment.

Analysing the learning curves (Figures 1 to 4), it can be observed that the agents trained with PER (PER DDQN and PER Dueling DDQN; Figure 3 and 4) maintains a steeper increase of reward over the first 500 episodes, archiving a maximum reward quicker. The curve for the DDQN agent

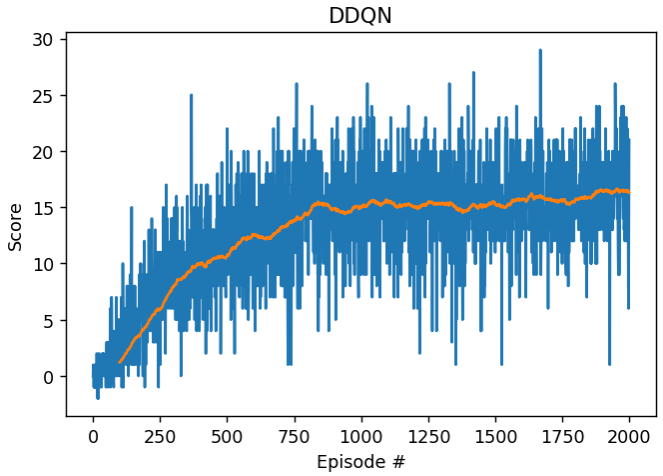

Fig. 2: DDQN learning performance on the Unity Banana Collection environment.

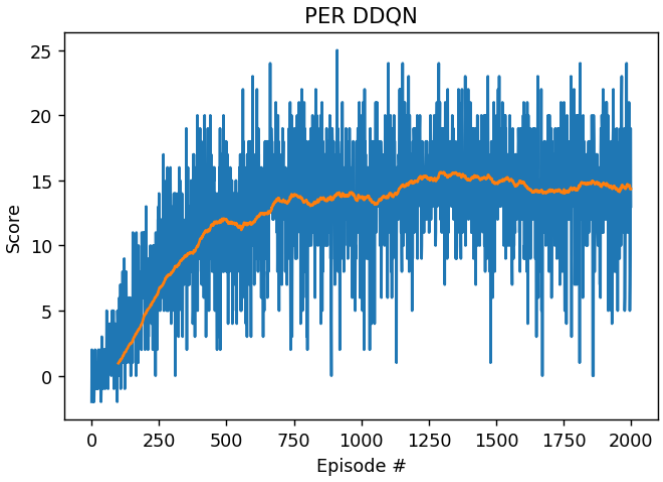

Fig. 3: PER DDQN learning performance on the Unity Banana Collection environment.

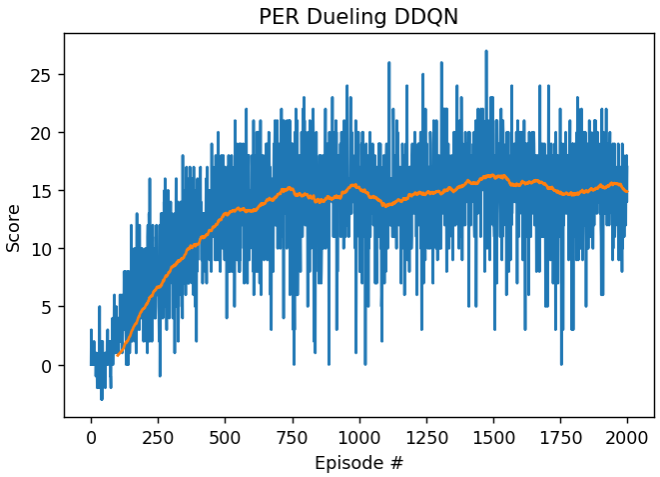

Fig. 4: PER Dueling DDQN learning performance on the Unity Banana Collection environment.

(Figure 2) is the least steep, but it is the only one that continues to rise until the end of training. In addition, it can be seen that the DDQN agent obtained larger maximum scores during training when compared to the others. The agent trained with the Dueling Q-Network (PER Dueling DDQN; Figure 4) has a similar curve to the PER DDQN agent, showing a slightly more stabilized learning, decreasing the spread of rewards over time.

5 Conclusion / Future work

The experiments developed in this works have shown that the implemented RL agents were able to successfully achieve the environment goal in less then 1800 episodes (Udacity benchmark impementation). Beyond that, the PER Dueling DDQN agent obtained the best performance, solving the environment in 501 episodes, and the DDQN agent obtained the maximum average reward over 100 consecutive episodes. The DQN agent obtained a relative good performance, reaching the environment goal quicker then the DDQN and the PER DDQN agents. Probably given the lack of time to tune the hiperparameters, the DDQN and PER DDQN agents takes long time to solve the environment, however, when seen the overall performance, that can be measured by the maximum average scores, its clear that the Double DQN agent implementation improved the learning performance of the agent.

Despite the relatively good performance of the agents implemented in this work, there is still much room for improvement. Max Schwarzer et. al (2020) [5] proposed a Self-Predictive Representation (SPR) algorithm to train the agent. The SPR algorithm trains an agent to predict its own latent state representations multiple steps into the future, substantially improving the learning performance. Another improvement that can be made is in the replay memory. Jiseong Han et. al (2021) [6] proposed the use of a double experience replay memory that exploits both important transitions and new transitions simultaneously. As showed by the authors, the proposed framework also significantly improves the agent performance. In addition to these improvements many other papers have been published in the last 4 years showing new methods and techniques to increase the training performance of Q-Learning agents. DRL is a newborn topic that is on the rise, and will probably remain hot for a long time due to its amazing ability and performance to solve RL problems.

References

[1] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski, *et al.*, "Human-level control through deep reinforcement learning," *nature*, vol. 518, no. 7540, pp. 529–533, 2015.

[2] T. Schaul, J. Quan, I. Antonoglou, and D. Silver, "Prioritized experience replay," *arXiv preprint arXiv:1511.05952*, 2015.

[3] H. Van Hasselt, A. Guez, and D. Silver, "Deep reinforcement learning with double q-learning," in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 30, 2016.

[4] Z. Wang, T. Schaul, M. Hessel, H. Hasselt, M. Lanctot, and N. Freitas, "Dueling network architectures for deep reinforcement learning," in *International conference on machine learning*, pp. 1995–2003, PMLR, 2016.

[5] M. Schwarzer, A. Anand, R. Goel, R. D. Hjelm, A. C. Courville, and P. Bachman, "Data-efficient reinforcement learning with momentum predictive representations," *CoRR*, vol. abs/2007.05929, 2020.

[6] J. Han, K. Jo, W. Lim, Y. Lee, K. Ko, E. Sim, J. Cho, and S. Kim, "Reinforcement learning guided by double replay memory," *Journal of Sensors*, vol. 2021, 2021.