Navigation

Christoffer Edlund

Abstract—Here goes a short abstract about this project i guess

Index Terms—Reinforcement Learning, Udacity, deep learning.

1 Introduction

The navigation project aims to use Deep Reinforcement Learning to teach an agent to collect yellow bananas and avoid blue bananas when moving around in a rectangular world, see Figure 1. The environment is based on Unity Machine Learning Agents (ML-Agents) engine with modifications done by the Udacity staff for the Deep Reinforcement Learning Nanodegree.

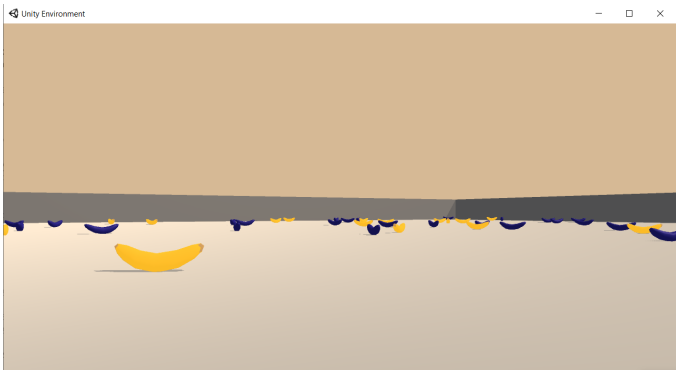

Fig. 1. Example view of the agent inside the Unity Environment

2 Deep Q-Learning

This projects revolves around the Deep Q-Learning Algorithm (DQN) that was introduced in 2015 by Mnih et al, at Google DeepMind [1]. The authors demonstrate a reinforcement learning algorithm that uses a deep neuronal network to approximate the Q-values used to guide an agent towards optimal actions given current state. What is novel in the DQN approach is the introduction of a *replay buffer* and *fixed Q-targets*, two implementations that solved the previous notorious problem of oscillation and difficulties to converge during the learning process when combining deep neuronal networks with reinforcement learning.

2.1 Replay Buffer

The replay buffer is used to store experiences, (state, action, reward, next-state), gathered by the agent while acting in the environment. After a certain amount of interaction with the environment the agent enters a learning phase. In the learning phase the agent samples experiences from the replay buffer at random with the aim to learn how to correctly predict the expected reward for a certain state-action pair. Drawing samples from multiple training runs gives the agent access to less correlated data to learn from compared to only being able to learn from one run and/or last seen experiences.

2.2 Fixed Q-Targets

The goal of the neuronal network in the DQN setting is to correctly estimate the current and future rewards given a state-action pair and discounting future rewards. To learn the current and future rewards for each state-action pair the network predicts what will be the expected future reward for current state-action pair. To produce a ”ground-truth” for that prediction we will once again rely on the network to infer the expected reward both for the next-state, assuming the agent taking the optimal action in current state. If the networks expected value prediction is correct, it should be equal to that of the next step times a discount factor plus the immediate reward, the last part is often refer to as the TD target. And the difference between the TD target and predicted future rewards value is refer to as the TD error.

Using a network to produce the prediction and TD target in this fashion while also constantly updating the weights of the network can make it hard for the network to converge and have been observed to produce oscillating behaviour [1]. Fixed Q-Targets solves this issue by creating a copy of the network that will have it’s weights frozen and then use this copy to produce the TD targets. To ensure that it also improves over time, the new copy will be updated with the weights of the original network from time to time. Thus producing more stable Q-targets that will still improve over time, reducing the oscillating behaviour and helping the network converge.

3 DQN Improvements

DQN have experienced a number of improvements over the years, here we will focus on two such improvements that will be used in this project.

3.1 Double DQN

In the original DQN algorithm there is a tendency for the network to overestimate the action values. This has empirically been shown to be mitigated with the Double Q-learning (Double DQN) algorithm [2]. When producing the

TD targets, instead of using the same network to select the best action and estimate its state-action value, the Double DQN algorithm uses one network to select the best action and another to evaluate the state-action value. In the context of a DQN with fixed Q-Targets there already exists two different networks, so the only modification that needs to be done is to swap out the Q-target network from both selecting and evaluating the action to using the main network to do the action selection and the Q-target network to evaluate it.

3.2 Prioritized Experience Replay

Instead of using random sampling to draw experiences to learning from the replay buffer, the Prioritized Experience Replay (PER) [3] buffer introduces sampling with higher probability to draw experiences that the network are struggling with (high TD error). This is done by assigning a priority

$$P_i = |\delta_i| + e \tag{1}$$

were δ is the TD error, e is a small number epsilon that prevent any priority from becoming to small and never selected, and P is the priority assigned to the experience sample i. The odds of sampling a specific priority is given by

$$P_i = \frac{P_i^a}{\sum_k P_k^a} \tag{2}$$

were a is a hyperparameter that controls how much weight the priority should have. An a equal to zero takes us back to uniformed sampling, while a one is fully sampling based on the priority. To account for the new sampling when updating the weights for the network we need to add a term to the error calculation and multiply it with the TD error as defined by

$$w_{error} = \left(\frac{1}{N}\frac{1}{P(i)}\right)^b \delta_i \tag{3}$$

Were b decide how big attention we pay to the priority when updating the weights, a low b makes all experiences weighted equally while a b close to one makes the algorithm pay more attention to examples that gets sampled less often (based on it's priority).

4 Enviornment

The environment is provided via Unity and is based on an agent collecting yellow bananas and avoiding blue bananas when walking around in 3D space. The agent have 4 possible discrete actions to chose from:

```
0 - move forward.
1 - move backward.
2 - turn left.
3 - turn right.
```

The agent senses the environment via a 37 dimensional vector representing the velocity and a ray-based perception of objects centered around the front of the agent.

Collecting a yellow banana results in a reward of +1 and collecting a blue banana results in a reward of -1, the goal is to achieve a reward of 13+ over 100 consecutive episodes. An example view of the environment can be seen in Figure.1 The agent was trained over multiple episodes, where one episode is defined as the agent interacting with the environment over 1000 frames.

5 Solution

For this project we explore four different solutions, the standard DQN proposed by Mhin et al, a Double DQN, Prioritized Experience Replay (PER) DQN and finally a combination of the methods named mini-rainbow.

The network architecture is a fully connected ANN with 5 layers, each hidden layer has 64 neurons with a ReLu activation function, the input layer is the same size as the environment vector (37) and the output the same as the action space (4). The batch size used for training was 32, with a learning rate of $5e^{-4}$ and with the network being updated every fourth iteration of interacting with the environment. The update of the target network was done using soft updates defined as

$$\theta_{target} = \theta_{main} \times \tau + \theta_{target} \times (1 - \tau) \tag{4}$$

where τ was set to $1e^{-3}$, θ_{target} is the target network weights and θ_{main} is the weights of the main network. The discount factor used to calculate the TD-targets was set to $\gamma = 0.99$. The agent used a ϵ-greedy policy that was set to 1.0 at the beginning and slowly decayed down to 0.01 by a decaying factor of 0.95 per episode. Meaning that the agent will select actions at random for the first episodes and move over to a more greedy behavior over time.

5.1 Double DQN

For the Double DQN we used at the same parameters as described above, but with the addition of a updating frequency for the target network which was set to five. In practice this means that our primary network gets updated every 4th frame and the target network being updated (with a soft update) every 20th frame.

5.2 PER DQN

The PER DQN introduces new hyperparameters such as the priority alpha (a) and priority beta (b) as described in section 3.2. The priority alpha is set to 0.6 and priority beta is initialized to 0.4 and increased to 1 over 800 episodes and the learning rate is decreased by a factor of 4 as done in the PER article [3]. Meaning that the weight updates will increasingly account for the priority for of the sample it learns from.

To increase the odds that each experience gets sampled at least once, the first experience recorded gets a priority of one and each new experience after that gets the max priority based on recorded experiences. Once the experienced actually gets sampled, they get a priority based on the equations outlined in section.3.2. Each priority

gets at least a TD-ϵ of $1e^{-2}$ to make sure that the odds of sampling a certain experience is never to close to zero.

5.2.1 SumTree

The priority replay buffer used to store and sample experiences are implemented with a SumTree, which is a binary tree that stores elements and a value associated with them as leafs, in our case experiences and their priority. Each node going up the tree stores the sum of priorities for all leafs under it. Sampling can easily be done by taking a value between 0 and the total priority and travel down the tree by going left if the sampled value is under the nodes left branch-node priority and otherwise going right. One of the great upsides with using a sum tree for this role is that the both lookup and insertion complexity for a SumTree is only O(log N).

5.3 Mini-rainbow DQN

The final method created for these experiments are a combination of both of the previously mentioned DQN improvements, Double DQN and PER DQN. The hyperparameters are the same as described above and the name is the a homage to the rainbow algorithm that is a combination of these improvements as well as others [4].

6 Results

When comparing the different methods the Double DQN method is the fastest to achieves the goal of a score of +13 over 100 consecutive episodes, which the agent completed with 421 episodes of training. The original DQN the next best model followed by the PER DQN model, and the worse model by a large margin was the mini-rainbow DQN as can be seen in Figure.2.

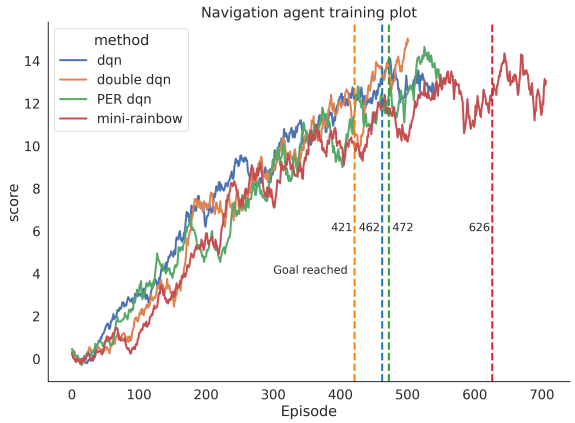

Fig. 2. Smoothed training plots for agents implementing different DQN methods, the plots display the score obtained after the number of training episodes the agent have went trough. The vertical lines marks when each agent have obtained the goal of a consecutive of score of over +13 plus over 100 episodes.

7 Discussion

The best model was the double DQN method, highlighting the positive effects of not over-estimating the Q-values. The second best is the original DQN method, with PER and mini-rainbow DQN performing worse than the original implementation, the possible reason why these more advanced models did not fair as well are many and hard to pin-down. It could be that the hyperparameters used was not fine-tuned enough to find parameters that was optimal for each model, there is also always the possibility of bugs in the code. But setting such parameters aside, one conclusion might be that the environment is easy enough for the original and double DQN model to solve. And introducing an bias for the model on hard examples as in the case of PER DQN does not particularly help the model in general to gain a high score.

As a follow up it would be interesting to see how the results would differ with a thorough hyper-parameter search and see if the results would vary. Further, it would be of interest to see how the models compares in a harder environment such as the Navigaton environment used here but with images/pixels as agent input instead of current input vector. It may very well be that the PER DQN would shine more in a harder environment.

8 Conclusion / Future work

Double DQN was showed to be the best performing method, reaching the goal after 421 episodes, beating the benchmark of the original DQN method. The mini-rainbow performed the worse by a large margin. To make more reliable results it is recommended to calculate the scores across multiple training's instead of individual runs as done here.

To further improve the agent performance one can look into further extensions to DQN such as Dueling DQN(cite) which predicts two values, a state value and a advantage value and adds them together into a Q-value instead of predicting the Q-value directly. Dividing up the prediction into two explicit parts has been showed to increase the performance further [5]. Other options can include multisteps-bootstrap targets [6], Distributional DQN [7] and Noisy DQN [8].

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, S. Petersen, C. Beattie, A. Sadik, I. Antonoglou, H. King, D. Kumaran, D. Wierstra, S. Legg, and D. Hassabis, "Human-level control through deep reinforcement learning," *Nature*, vol. 518, pp. 529–533, Feb. 2015.

[2] H. v. Hasselt, A. Guez, and D. Silver, "Deep Reinforcement Learning with Double Q-Learning," *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 30, Mar. 2016. Number: 1.

[3] T. Schaul, J. Quan, I. Antonoglou, and D. Silver, "Prioritized Experience Replay," *arXiv:1511.05952 [cs]*, Feb. 2016. arXiv: 1511.05952.

[4] 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," *arXiv:1710.02298 [cs]*, Oct. 2017. arXiv: 1710.02298.

[5] Z. Wang, T. Schaul, M. Hessel, H. van Hasselt, M. Lanctot, and N. de Freitas, "Dueling Network Architectures for Deep Reinforcement Learning," *arXiv:1511.06581 [cs]*, Apr. 2016. arXiv: 1511.06581.

[6] 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," *arXiv:1602.01783 [cs]*, June 2016. arXiv: 1602.01783.

[7] M. G. Bellemare, W. Dabney, and R. Munos, "A Distributional Perspective on Reinforcement Learning," *arXiv:1707.06887 [cs, stat]*, July 2017. arXiv: 1707.06887.

[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," *arXiv:1706.10295 [cs, stat]*, July 2019. arXiv: 1706.10295.