Deep Reinforcement Learning applied to solve a Banana Collecting Environment

Catarina Pires
Deep Reinforcement Learning Nanodegree, Udacity

***Abstract*—This work consists on the application of three deep reinforcement learning algorithms from literature, Deep Q Networks [1], Double Deep Q Networks [2], and Prioritised Experience Replay [3], to solve a banana collecting environment. Collecting yellow bananas rewards the agent with +1, while collecting blue ones gives a reward of -1. The task is episodic and the environment is considered solved when the agent achieves an average score of 13 over 100 consecutive episodes. The agent trained with Deep Q Networks solved the environment in 385 episodes, while the one trained with Double Deep Q Networks and Double Deep Q Networks with Prioritised Experience Replay solved the environment in 423 and 438 episodes respectively. Thus, all the implemented methods fulfilled the challenge of solving the environment in fewer than 1800 episodes.**

I. Introduction

Drawing inspiration from psychological and neuroscientific research in animal behaviour, reinforcement learning (RL) is a computational approach to automating goal-oriented learning and decision making [4]. Unlike most algorithms studied in the Machine Learning field, reinforcement learning techniques excel at discovering optimal action sequences in temporal decision tasks characterised by sparse external evaluations [5]. In these scenarios, the learner lacks prior knowledge about the effects of actions or the temporal delay between actions and their impact on performance. Over the last years, the combination of reinforcement learning with deep neural networks, giving rise to Deep Reinforcement Learning (DRL), has seen a significant growth in popularity, being applied in fields like robotics [6] and the gaming industry [7], to name a couple.

Inspired by arguably one of the biggest breakthroughs in DRL in the last decade, the Deep Q algorithm [1] presented by Google Deepmind, this work consists of its implementation, as well as two posterior Google Deepmind algorithms, to solve a banana collecting environment. This report is divided into six sections, where section II describes in more detail the algorithms used, section III introduces the environment used in this project, Section IV presents the hyperparameters and model architectures chosen as well as the performance for each algorithm, section V discusses and compares the results obtained, and finally section VI presents some ideas on how this work could be improved.

II. Methods

This section presents a short introduction to each of the algorithms implemented. Further detail can be found on the referenced original papers for each one.

A. Deep Q Networks

Deep Q-Networks (DQN) [1] is an algorithm that combines Q-learning [8] with deep learning to solve complex, high-dimensional reinforcement learning problems. This was achieved by using a deep neural network to approximate the optimal action-value function

$$Q^*(s,a) = \max_{\pi} \mathbb{E}[r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + ... | s_t = s, a_t = a, \pi]$$

where the maximum sum of rewards r_t is discounted by γ at each time step t, achievable by a behaviour policy $\pi = P(a|s)$, after making an observation (s) and taking an action (a).

Reinforcement learning is known to being prone to instabilities, and these may be especially prominent when a nonlinear function approximator, such as a neural network, is used to represent the action-value (also known as Q) function. The way the algorithm addresses these instabilities is twofold: it leverages a biologically inspired mechanism termed experience replay [9], [10], [11] that randomises over the data, thus removing correlations in the observation sequence and smoothing over changes in the data distribution; and, secondly, the action-values (Q) are iteratively updated towards target values that are only periodically updated, thereby reducing correlations with the target. The algorithm of the DQN is then described in algorithm 1.

B. Double Deep Q Networks

Q learning stands as one of the most popular reinforcement learning algorithms, however it is known to sometimes learn unrealistically high action values because it includes a maximisation step over estimated action values, which tends to prefer overestimated to underestimated values [2].

The Double Q-Learning algorithm aims to reduce overestimation by breaking up the max operation in the target into action selection and action evaluation. Leveraging the DQN algorithm, [2] proposes to evaluate the greedy policy according to the online network, but using the target network to estimate its value. The update is the same as for the DQN, but instead of the target Y_t^{DQN} we have

$$Y_t^{DoubleDQN} = r_{t+1} + \gamma Q(s_{t+1}, argmax_a(Q(s_{t+1}, a; \theta_t), \theta_t^-)$$

Algorithm 1 Deep Q Networks

```
Initialise replay memory $\mathcal{D}$ with a maximum size to store experience tuples
Initialise Q-network with random weights $\theta$
Initialise target Q-network with weights $\theta^- = \theta$
for $i \leftarrow 1$ to $num\_episodes$ do do
    $\epsilon \leftarrow \epsilon_0$
    Observe $s_0$
    for $t \leftarrow 0$ to $episode\_steps$ do do
        Choose action $a_t$ according to policy $\epsilon - greedy$
        Take action $a_t$ and observe $r_{t+1}$ and $S_{t+1}$
        Store $(s_t, a_t, r_{t+1}, s_{t+1}, d)$ in replay $\mathcal{D}$
        if Update network then
            Sample random mini-batch of experiences $(s_{t_j}, a_{t_j}, r_{t+1_j}, s_{t+1_j}, d_{t+1_j})$ from $\mathcal{D}$
            $y_j = r_{t+1_j} + \gamma max_{a_{t+1}} Q(s_{t+1_j}, a_{t+1}, \theta^-)$
            $loss = \frac{1}{N} \sum_j (y_j - Q(s_j, a_j, \theta))^2$
            Update weights $\theta$
            Update target network weights $\theta^-$
        end if
        $t \leftarrow t + 1$
    end for
end for
```

C. Prioritised Experience Replay

How an online reinforcement learning agent is defined to interact and learn from its experiences is crucial to its success. In its simplest form, where incoming data is discarded after a single update, the following issues arise:

- The i.i.d. assumption of many stochastic gradient descent algorithms is broken due to strongly correlated updates.
- Possibly rare experiences are rapidly forgotten.

Experience replay [12] addresses both these issues by breaking temporal correlations, since it selects randomly out of old and new experiences, allowing also for rare experiences to be used more than once. For these reasons, experience replay was also implemented with the previous two algorithms. Building on this idea, prioritised experience replay adds another layer by making the most effective use of the replay memory for learning, assuming that its contents are outside of our control.

Therefore, the criterion that by which the importance of each transition is measured constitutes the cern of prioritised experience replay. Inspired by the amount the RL agent can learn from a transition in its current state, a reasonable measure to use is the magnitude of a transition's TD error δ, which indicates how 'surprising' or unexpected the transition is. This can be particularly suitable for the Q-Learning and SARSA algorithms, since these are online RL algorithms that already compute the TD-error and update the parameters in proportion to δ.

However, this method also presents several issues such as being sensitive to noise spikes and focusing on high error transitions, which get replayed frequently, whereas transitions that have a low TD error on first visit may not be replayed for a long time. This lack of diversity makes the system prone to over-fitting.

Having these issues in mind, [3] propose a stochastic sampling method that interpolates between pure greedy prioritisation and uniform random sampling. The probability of sampling transition i is defined as

$$P_i = \frac{p_i^a}{\sum_k p_k^a}$$

where $p_i > 0$ is the priority of transition i, and the exponent α determines how much prioritisation is used, with $\alpha = 0$ corresponding to the uniform case.

The algorithm for the Double DQN with Prioritised Experience Replay [3] is described in algorithm 2.

Algorithm 2 Double DQN with proportional prioritisation

```
Input: minibatch $k$, step-size $\nu$, replay period $K$ and size $N$, exponents $\alpha$ and $\beta$
Initialise Q-network with random weights $\theta$
Initialise target Q-network with weights $\theta^- = \theta$
for $i \leftarrow 1$ to $num\_episodes$ do do
    $\epsilon \leftarrow \epsilon_0$
    Observe $s_0$
    for $t \leftarrow 0$ to $episode\_steps$ do do
        Choose action $a_t$ according to policy $\epsilon - greedy$
        Take action $a_t$ and observe $r_{t+1}$ and $S_{t+1}$
        Store $(s_t, a_t, r_{t+1}, s_{t+1}, d)$ in replay $\mathcal{D}$
        if Update network then
            Sample random mini-batch of experiences $(s_{t_j}, a_{t_j}, r_{t+1_j}, s_{t+1_j}, d_{t+1_j})$ from $\mathcal{D}$
            $y_j = r_{t+1_j} + \gamma max_{a_{t+1}} Q(s_{t+1_j}, a_{t+1}, \theta^-)$
            $loss = \frac{1}{N} \sum_j (y_j - Q(s_j, a_j, \theta))^2$
            Update weights $\theta$
            Update target network weights $\theta^-$
        end if
        $t \leftarrow t + 1$
    end for
end for
```

III. Environment

The Unity Machine Learning Agents (ML-Agents) toolkit [13] is an open-source Unity plugin that enables games and simulations to serve as environments for training intelligent agents. For the purpose of this project, one of the Unity's rich environments was used to design, train, and evaluate the chosen deep reinforcement learning algorithms. Note that this environment is similar but not identical to the Banana Collector present in [13], The environment consists of a large square world where an agent must accumulate reward by collecting bananas, Fig. 1. However, not all bananas are created the same as yellow bananas reward the agent with +1, while blue ones reward the agent with -1. The state space has 37 dimensions and contains the agent's velocity, along with ray-based perception of objects around the agent's forward direction. Given this information, the agent has to learn how to best select actions, given that there are four possible: move forward, move backward, turn left and turn right. The task is episodic, and in order to solve the environment, the agent must get an average score of +13 over 100 consecutive episodes.

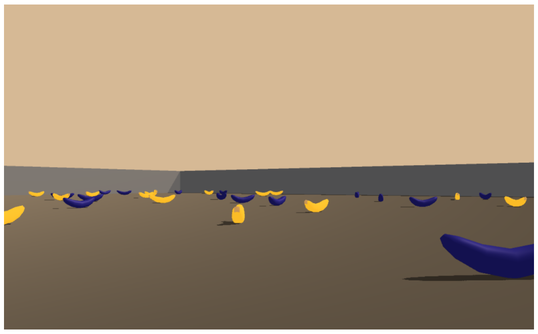
Figure 1. Environment

IV. Implementation and Results

The neural network architecture was designed for reinforcement learning, specifically for approximating the action-value function (Q-function). It consists of the following components:

- **Input Layer**: Takes the state as input, with a size equal to the number of possible states, $state\_size$.
- **Two Fully Connected Hidden Layers**:
  - The first hidden layer has 64 units and applies a ReLU activation function.
  - The second hidden layer also has 64 units, followed by another ReLU activation.
- **Output Layer**: Maps the transformed state into $action\_size$ dimensions, representing the action-value (Q-values) for each possible action.

The network uses ReLU activation to introduce non-linearity and is designed to predict Q-values given a state in a reinforcement learning environment.

The same neural network was used for all implementations and Adam was used as the optimiser.

A. Deep Q Networks

The parameters used in the implementation of the Deep Q Networks algorithm were:

- Discount factor: $\gamma = 0.99$;
- Learning rate: $LR = 5e - 4$;
- Epsilon for epsilon-greedy policy: starting value = 1.0, minimum value = 0.01, decay = 0.995;
- Mini batch size: 64;
- Replay buffer size: $1e5$
- Soft update of the target: $\tau = 1e - 3$;
- Network update rate: update every 4 steps;

Figure 2 shows the training results for the Deep Q Network implementation:

- "Score" denotes the unfiltered final reward at each episode, which is noisy.
- "Average over the last 100 episodes" denotes the same scores, but smoothed out with a 100-episode moving average to better capture trends.
- "Goal" marks the value $score = 13$.

The goal was reached in 385 episodes.

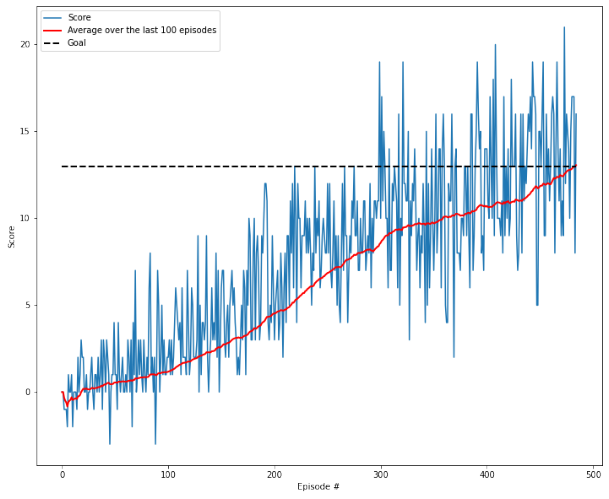

Figure 2. Training results for the Deep Q Network implementation

B. Double Deep Q Networks

The parameters used in the implementation of the Double Deep Q Networks algorithm were:

- Discount factor: $\gamma = 0.99$;
- Learning rate: $LR = 11e - 5$;
- Epsilon for epsilon-greedy policy: starting value = 1.0, minimum value = 0.01, decay = 0.995;
- Mini batch size: 64;
- Replay buffer size: $1e5$;
- Primary network update rate: update every 4 steps;
- Target network update rate: update every 1000 steps;

Figure 3 shows the training results for the Double Deep Q Network implementation, where the goal was reached in 423 episodes.

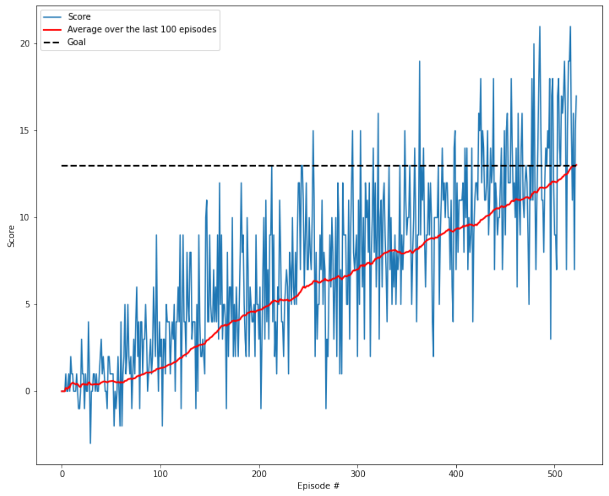

Figure 3. Training results for the Double Deep Q Network implementation

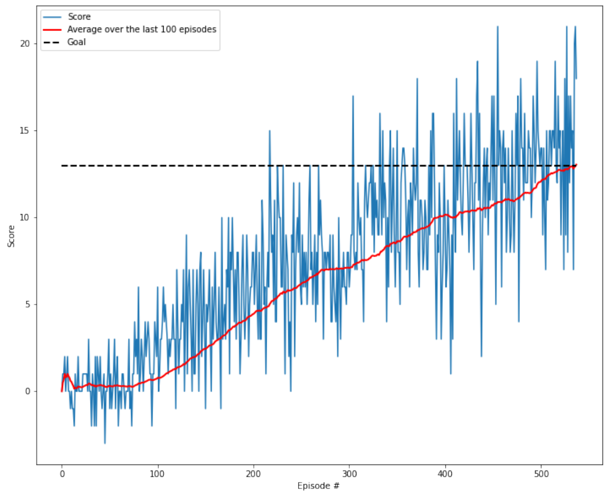

Figure 4. Training results for the Double Deep Q Network with Prioritised Experience Replay implementation

C. Prioritised Experience Replay

The parameters used in the implementation of the Double Deep Q Networks algorithm with Prioritised Experience Replay were:

- Parameters for the Double Deep Q Networks algorithm:
  - Discount factor: $\gamma = 0.99$;
  - Learning rate: $19.4e - 5$;
  - Epsilon for epsilon-greedy policy: starting value = 1.0, minimum value = 0.01, decay = 0.995;
  - Mini batch size: 64;
  - Replay buffer size: 12800
  - Primary network update rate: update every 4 steps;
  - Target network update rate: update every 1000 steps;
- Parameters for the Prioritised Experience Replay:
  - $c = 1e - 4$;
  - $\alpha = 0.7$;
  - $\beta_i = 0.5$;
  - $\beta_f = 1$;
  - $\beta_{updatesteps} = 1000$;

Figure 4 shows the training results for the Double Deep Q Network with Prioritised Experience Replay implementation, where the goal was reached in 438 episodes.

V. Discussion and conclusion

In the previous section the results for the implementation of each algorithm show that the ”vanilla” DQN outperfomed both the Double DQN and the Double DQN with PER. The progression of the results was the opposite of what was expected, considering that the last two algorithms were designed to address specific limitations of the DQN, such as overestimation bias and inefficient learning from experience. The poor performance of these algorithms may be explained by:

- **Hyperparameter Tuning:** Tuning was performed manually based on some reference values from previous exercises in the course, which particularly helped for the DQN implementation, and from literature. However, Double DQN and PER typically require more careful tuning to get the most out of them, as they introduce additional complexities (like the prioritisation scheme in PER).
- **Overfitting or Instability in PER**: While PER accelerates learning by prioritising certain experiences, it can also introduce biases or instability in training, especially if the prioritisation hyperparameters are not correctly set (e.g., too much focus on specific experiences). In addition, PER can lead to high variance in the gradients due to the prioritization of rare or highly surprising experiences. This variance can destabilize the training process and lead to suboptimal performance.

VI. Future Ideas

In order to address the poor hyperparameter tuning, an ablation study could be performed in which several combinations of hyperparameters are chosen and compared against each other to pick which optimise the problem. For best perfomrance, the agent should be trained several times for each hyperparameter combination and the mean/standard deviation over all trials should be collected in order to reduce noise.

Furthermore, it would also be interesting to apply and compare algorithms such as the Dueling DQN [14] and Rainbow [15], both also improvements on the ”vanilla” DQN.

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] 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, no. 1, 2016.

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

[4] R. S. Sutton and A. G. Barto, *Reinforcement learning: An introduction*. MIT press, 2018.

[5] S. Thrun and A. Schwartz, "Issues in using function approximation for reinforcement learning," in *Proceedings of the 1993 connectionist models summer school*. Psychology Press, 2014, pp. 255–263.

[6] M. Hutter, C. Gehring, D. Jud, A. Lauber, C. D. Bellicoso, V. Tsounis, J. Hwangbo, K. Bodie, P. Fankhauser, M. Bloesch *et al.*, "Anymal-a highly mobile and dynamic quadrupedal robot," in *2016 IEEE/RSJ international conference on intelligent robots and systems (IROS)*. IEEE, 2016, pp. 38–44.

[7] K. Souchleris, G. K. Sidiropoulos, and G. A. Papakostas, "Reinforcement learning in game industry—review, prospects and challenges," *Applied Sciences*, vol. 13, no. 4, p. 2443, 2023.

[8] C. J. C. H. Watkins, "Learning from delayed rewards," 1989.

[9] J. L. McClelland, B. L. McNaughton, and R. C. O'Reilly, "Why there are complementary learning systems in the hippocampus and neocortex: insights from the successes and failures of connectionist models of learning and memory." *Psychological review*, vol. 102, no. 3, p. 419, 1995.

[10] J. O'Neill, B. Pleydell-Bouverie, D. Dupret, and J. Csicsvari, "Play it again: reactivation of waking experience and memory," *Trends in neurosciences*, vol. 33, no. 5, pp. 220–229, 2010.

[11] L.-J. Lin, *Reinforcement learning for robots using neural networks*. Carnegie Mellon University, 1992.

[12] ——, "Self-improving reactive agents based on reinforcement learning, planning and teaching," *Machine learning*, vol. 8, pp. 293–321, 1992.

[13] "GitHub - Unity-Technologies/ml-agents: The Unity Machine Learning Agents Toolkit (ML-Agents) is an open-source project that enables games and simulations to serve as environments for training intelligent agents using deep reinforcement learning and imitation learning. — github.com," https://github.com/Unity-Technologies/ml-agents, [Accessed 30-05-2024].

[14] 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*. PMLR, 2016, pp. 1995–2003.

[15] 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," in *Proceedings of the AAAI conference on artificial intelligence*, vol. 32, no. 1, 2018.