A Deep Q-Network Implementation

Nicolas Monti

January 2024

1 Introduction

This report describes a Deep Q-Network implementation which was carried out as part of the Deep Reinforcement Learning Nanodegree program from *Udacity*.

The goal is to train an agent to navigate a large square world to collect yellow bananas while avoiding blue bananas. This world, which we refer to as an environment, is provided by *Unity Machine Learning Agents (ML-Agents)* - an open-source plug-in that enables games and simulations to serve as environments for training intelligent agents.

A reward of +1 is provided for collecting a yellow banana, and a reward of -1 is provided for collecting a blue banana. Thus, the goal of the agent is to collect as many yellow bananas as possible while avoiding blue bananas.

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. Four discrete actions are available, corresponding to:

0. Move forward.
1. Move backward.
2. Turn left.
3. 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. Note that in this particular implementation, it was decided to stop training the agent if an average score of +15 was obtained over 100 consecutive episodes (see section 5).

2 Training Algorithm

The algorithm implemented to train the agent is based from the article titled *Human-level control through deep reinforcement learning* [1].

There are two main processes running on the Deep Q-Learning algorithm implemented: sampling and learning.

The first process takes care of sampling the environment at each time-step t by taking an action a_t and storing the observed experience tuple $e_t = (s_t, a_t, r_t, s_{t+1})$ into a replay memory $D_t = (e_1, ..., e_t)$ - where s_t and r_t represent the environment's state and the reward collected at time-step t, respectively, and s_{t+1} is the environment's next state after taking action a_t.

The second process randomly selects a batch of experiences e from the replay memory D and learns from that batch using a gradient descent step.

It is important to note that these two processes are not dependent on each other, and we can perform multiple sample steps and then one learning step – or even multiple learning steps from different batches. On this implementation, it was decided to perform a single learning step every C steps. A brief summary of the overall training procedure is depicted on Algorithm 1.

Algorithm 1 A Deep Q-Learn Training Algorithm

```
Initialize:
    * Replay memory $D$ to capacity $N$
    * Action-value function $Q$ with random weights $\theta$
    * Target action-value function $\hat{Q}$ with weights $\theta^- = \theta$
for $episode = 1$ to $M$ do
    Observe current state $S = s_0$

    for time-step $t = 1$ to $T$ do
        Choose action $a_t$ from current state $S$ using existing $\epsilon$-greedy policy $\pi$
        Take action $a_t$ and observe $r_t$, $s_{t+1}$
        Save experience tuple $e_t = (S, a_t, r_t, s_{t+1})$ into replay memory $D$

        Every $C$ time-steps do
            Randomly sample a batch of experiences $e$ from replay memory $D$:
                $e_j = (s_j, a_j, r_j, s_{j+1})$
            Compute the target action values using $\hat{Q}$:
                $Q_{target} = r_j + \gamma * \max_a \hat{Q}(s_{j+1}, \theta^-)$
            Compute the expected action values using $Q$:
                $Q_{expected} = Q(s_j, a_j, \theta)$
            Compute the loss using mean squared error:
                $L = \sum_{k=0}^{B} (Q_{expected,k} - Q_{target,k})^2 / B$ where $B$ is batch size
            Calculate gradients with backward pass
            Update $Q$ network weights $\theta$ with learning rate $LR$
            Update $\hat{Q}$ network weights $\theta^-$:
                $\theta^- = \tau * \theta + (1 - \tau) * \theta^-$

        Update current state: $S \leftarrow s_{t+1}$
    end for
end for
```

3 Q-Network Architecture

The chosen neural network architecture is quite simple and consists of three fully connected linear layers using a *ReLU* activation function after the first and second layers. The first and second linear layers output 64 features each.

4 Training Set-up

The hyper-parameters chosen for this implementation were selected based on experience and results from a similar environment solved within the Nanodegree program. The training set-up consist of the following:

- Total number of episodes: $M = 2000$
- Maximum time-steps per episode: 1000
- Learn every time steps: $C = 4$
- Batch size: $B = 32$
- Replay memory D with capacity $N = 100,000$
- $\gamma = 0.99$
- $\tau = 0.001$
- Learning rate: $LR = 0.0005$
- The ϵ parameter required to implement an ϵ-greedy policy π is initialized at $\epsilon = 1$ and then is decay at each time-step by a multiplicative factor of 0.995 with a limit lower bound of $\epsilon = 0.01$

Finally, it is worth to mention that the weights initialization on the linear layers of the Q-Network is automatically taken care of by *PyTorch*. The default initialization consist of randomly sampling from a uniform distribution $U(-\sqrt{k}, \sqrt{k})$, where $k = 1/(input\,features)$.

5 Results

On figure 5, we show the agent's cumulative scores per episode. At episode 666 we consider the environment as solved which is when the agent achieved the goal of an average score of +15 over 100 consecutive episodes. This result seems reasonable considering the benchmark implementation provided by Udacity as part of the Nanodegree program where it could take in some instances approximately 1800 episodes to solve this environment.

6 Future Work

Although this implementation managed to successfully solve the environment in a reasonable amount of episodes, there are still some changes that could be explored to improve the agent's performance. Some of these include:

- **Error clipping**. Clipping the error term on the loss function to be between -1 and 1 could further improve the stability of the algorithm as mentioned on [1].

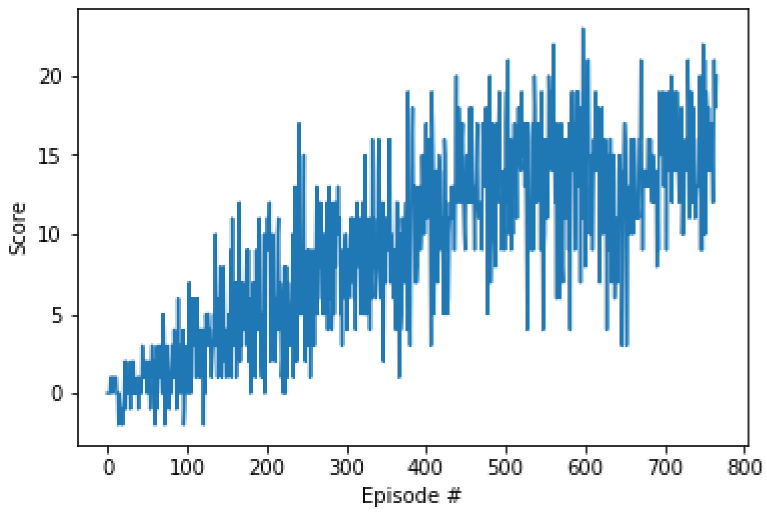

Figure 1: Agent's cumulative rewards per episode

- **Prioritized experience replay**. Based on the algorithm used on this implementation, we sample experiences uniformly at random when going through a learning step. This approach is to some extent limited because we might be missing the chance to learn more effectively from valuable experiences. This is where using a sophisticated sampling strategy where more important transitions are sampled with a higher probability could improve the agent's performance. More details can be found on [2].
- **Dueling DQN**. As described on [3], the authors present a new neural network architecture for model-free reinforcement learning. This architecture referred as a dueling network represents two separate estimators: one for the state value function and one for the state-dependent action advantage function. Intuitively, the dueling architecture can learn which states are valuable, without having to learn the effect of each action for each state. This is particularly useful in states where its actions do not affect the environment in any relevant way.

References

[1] Volodymyr Mnih1, Koray Kavukcuoglu1, David Silver1, Andrei A. Rusu1, Joel Veness1, Marc G. Bellemare1, Alex Graves1, Martin Riedmiller1, Andreas K. Fidjeland1, Georg Ostrovski1, Stig Petersen1, Charles Beattie1, Amir Sadik1, Ioannis Antonoglou1, Helen King1, Dharshan Kumaran1, Daan Wierstra1, Shane Legg1, and Demis Hassabis1. Human-level control through deep reinforcement learning. *Nature*, 518:529–523, 2015. doi:https://doi.org/10.1038/nature14236.

[2] Tom Schaul, John Quan, Ioannis Antonoglou, and David Silver. Prioritized experience replay. *Arxiv*, Feb 2016. doi:https://doi.org/10.48550/arXiv.1511.05952.

[3] Ziyu Wang, Tom Schaul, Matteo Hessel, Hado van Hasselt, Marc Lancto, and Nando de Freitas. Dueling network architectures for deep reinforcement learning. *Arxiv*, Apr 2016. doi:https://doi.org/10.48550/arXiv.1511.06581.