Deep Reinforcement Learning

Solving the Banana Unity Environment

Aneesh Chandran

Abstract

This project implements the Deep Q-network and Double Deep Q-network algorithms to solve the Banana Unity Environment. The algorithms were required to solve the environment within 1800 episodes with an average reward of 13 points or more. The Deep Q-network was able to solve with an average score of 13.33 for at least a 100 episodes in 1300 episodes. The Double Deep Q-network was able to solve the environment in 500 episodes and scored an average reward of 14.07 for at least a 100 episodes. Both algorithms had the same neural network parameters and are shown in detail in the report. Future improvements include the plan to implement Priority Experience Replay, Dueling DQN and Rainbow DQN.

Introduction

This report discusses the learning algorithms implemented to solve the Banana environment from Unity. The algorithms used are namely Deep Q-Network (DQN) and Double Deep Q-Network (DDQN), which are algorithms of model free reinforcement learning problems. The report contains a brief explanation of the environment, a summary of the learning algorithms and finally the implementation details as well as results of the algorithm in solving the environment.

The Banana Unity Environment

The goal of the environment is to collect yellow bananas as the agent is traversing. When the agent collects a yellow banana it receives a reward of +1 and it must avoid blue bananas as it will reward the agent a -1. The goal of the agent is to learn to consistently collect yellow bananas and avoid blue bananas. For the purpose of this project the agent needs to have an average reward of 13 or more for an average of 100 episodes. The environment has a state space of 37 dimensions, it contains information such as the agent's velocity and ray-based perception of objects around the agent’s forward direction. The actions available to the agent are move forward, move backward, turn left and turn right.

Deep Q-Network

In 2015 the team at DeepMind were able to develop the Deep Q Learning algorithm [1] and successfully train agents to play on classic Atari 2600 games. They were able to achieve human professional gamer levels for 49 games. They were able to train the algorithm with pixel level information of the game as well as the score of the games as input. Traditional Q-learning algorithms have the problem of scaling up as the state space and action space grows. The use of function approximators to predict the values in the Q-table have been proposed before, this algorithm uses the advancements in deep learning and implements neural networks for function approximation. An optimal action-value function can be represented as :

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

The work from DeepMind used convolutional neural networks to approximate equation 1. The way this was achieved was by parameterising the action-value function as $Q(s,a;\theta)$, with parameters θ, which are the weights of the neural network. This work uses two near identical neural networks, one for local training and the other for evaluation called the target network. The target network would have a fixed weights with parameter θ^- and a parameterised action-value function written as $Q(s,a;\theta^-)$. The reason why the target network exists is to prevent the local network from training on a moving “q-value”, the weights in the local network would change and thus change the prediction of the q-value. This would cause the local network to change and thus learning would happen in the changed environment, this is not conducive for learning and is analogous to chasing one's own tail. By evaluating the local network against the target network which is fixed for a short duration of time, would allow for learning to happen much smoother. Then periodically the target network is updated with the weights from the local network after a fixed number of steps. The update for this project is implemented using the Polyak averaging.

$$\theta^{-} = \theta \times \tau + (1-\tau)\,\theta^{-} \qquad (2)$$

The paper also also combines the two neural networks with a replay buffer, to sample past experiences to learn faster. The use of the replay buffer allows the neural network to not be victim to correlated experiences and thus overfitting to certain sequences of events. The loss for the neural network is worked out as:

$$L_i(\theta_i) = E_{(s,\,a,\,r,\,s`)\sim U(D)}[(r + \gamma\, max_{a`}\, Q(s`, a`;\, \theta^{-}{}_{i}) - Q(s,\, a;\, \theta_i))^2]\ , \qquad (3)$$

w ith s` and a` being the next state and next action.

Double Deep Q-Network

Later DeepMind introduced an optimised version of the DQN with Double Deep Q-Network (DDQN) which improved significantly on the original version [2]. The difference in the DDQN algorithm lies in the way the loss function is created. By taking the max q-value of the target network the algorithm will always overestimate the q-value. The fix for this is to select the action from the local network for the next state, this step is known as action selection. The action selection would happen in the local network. Based on the action which yields the max q-value for the local network, select the q-value from the target network for evaluation. The target network is used for the action evaluation. By modifying equation 3 we can construct the new loss function as:

$$L_i(\theta_i) = E_{(s,\,a,\,r,\,s`)\sim U(D)}[(r + \gamma\, Q(s`, argmax\ a\ Q(s`,\ a`;\ \theta_i);\ \theta^{-}{}_{i}) - Q(s,\ a;\ \theta_i))^2] \qquad (4)$$

Results

The results are shown in the table below, and it is evident that the DDQN has shown a significant improvement to the DQN. It is important to note that the neural network architecture and parameters were kept consistent to show the significance of the DDQN architecture.

Parameters	DQN Values	DDQN Values
No of Layers	3 (Input, Hidden Layer, Output)	3 (Input, Hidden Layer, Output)
Input Layer Input size	37 (size of the state space)	37 (size of the state space)
Input Layer number of nodes	64	64
Input Layer output size	64	64
Input Layer Activation Function	Relu	Relu
First Hidden Layer Input Size	64	64
First Hidden Layer nodes	64	64
First Hidden layer Output	64	64
First Hidden Layer Activation Function	Relu	Relu
Output Layer Input	64	64
Output Layer nodes	4	4
Output Layer Output	4 (size of the action space)	4 (size of the action space)
Learning Rate	0.5	0.5
Epsilon Decay	0.995	0.995
No:of steps limited per episode	1000	1000
Discount Factor	0.99	0.99
Batch Size	64	64
Buffer Size	10000	10000
Steps to update Target Network	4	4
Soft Update Parameter tau	0.001	0.001
Average Score	13.33	14.07
No of Episodes to Solve Env	1300	500

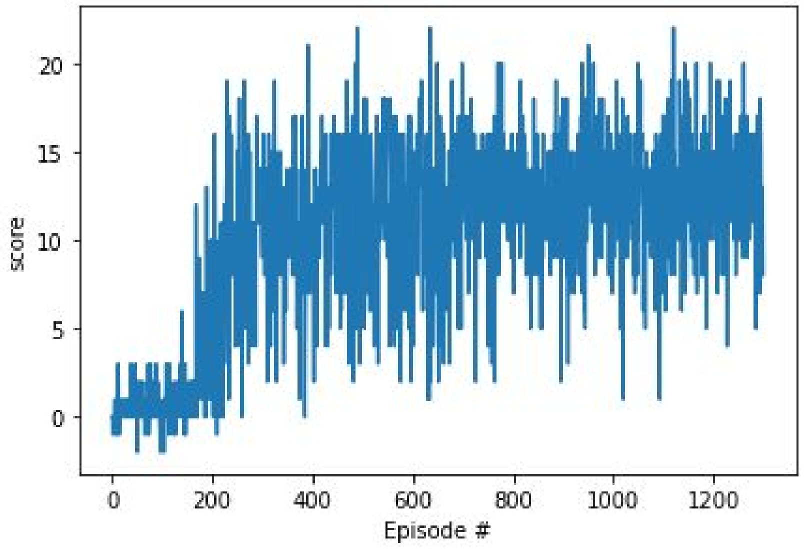

Figure 1: Average reward to number of episodes for DQN

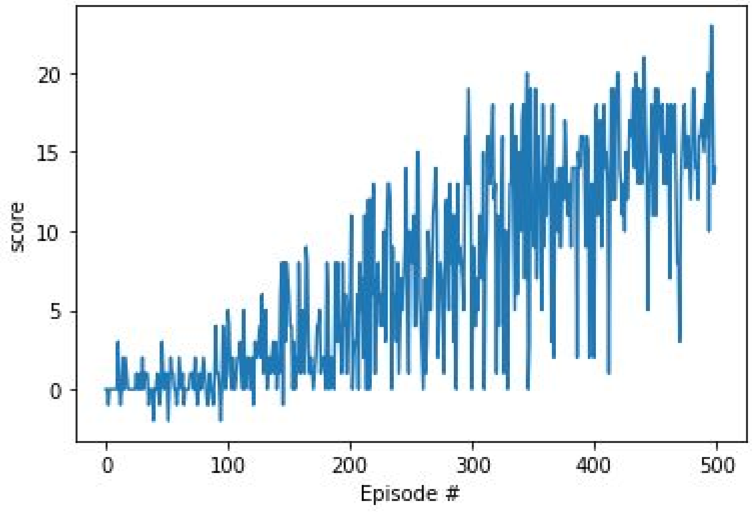

Figure 2: Average reward to number of episodes for DDQN

Future Work

In the future I would like to implement the Priority Experience Replay, Dueling DQN and Rainbow DQN. I would also like to work on the learning to solve the environment from pixels. This involves using a different neural network architecture involving Convolutional Neural Networks (CNN).

Conclusion

It was evident in the implementation of the DQN and DDQN how function approximators are useful to solve the Unity Banana environment. Both algorithms solved the environment in under 1800 steps with an average reward of 13 or more for 100 episodes.

References

1. Mnih, Volodymyr & Kavukcuoglu, Koray & Silver, David & Rusu, Andrei & Veness, Joel & Bellemare, Marc & Graves, Alex & Riedmiller, Martin & Fidjeland, Andreas & Ostrovski, Georg & Petersen, Stig & Beattie, Charles & Sadik, Amir & Antonoglou, Ioannis & King, Helen & Kumaran, Dharshan & Wierstra, Daan & Legg, Shane & Hassabis, Demis. (2015). Human-level control through deep reinforcement learning. Nature. 518. 529-33. 10.1038/nature14236.
2. Van Hasselt, Hado & Guez, Arthur & Silver, David. (2015). Deep Reinforcement Learning with Double Q-learning.