Nanodegree: Deep Reinforcement Learning

By: Kranthi Kumar Talluri

Section 1: Introduction

The project showcases the effectiveness of value-based methods, specifically Deep Q learning and its variants. In acquiring suitable policies for Reinforcement Learning tasks. These tasks involve navigating environments with continuous state spaces, where the objective is to collect yellow bananas for positive rewards and avoid blue bananas for negative rewards. The agents undergo training using Deep Reinforcement Learning techniques, which include the use of Deep Q Networks as nonlinear function approximators. By linking states to actions through action value functions the agents aim to optimize cumulative rewards and attain optimal policies. Throughout the projects an array of files such as Jupyter Notebooks and code files are utilized to set up dependencies, define agent characteristics and store trained models.

We implement a simple game based on DQN to demonstrate its model free learning. The agent must go around the arena and collect yellow bananas, it fetches a positive reward of +1. It must also avoid collecting blue bananas for it is a negative reward of -1. Agents can move in four directions, forward, back, left, and right. The unity space is 37 dimensions.

- The goal is to train an agent to achieve an average score of +13 over 100 consecutive episodes by collecting yellow bananas and avoiding blue ones.
- The state space consists of 37 features, including the agent's velocity and ray-based perception of objects, while the action space has a dimension of four.
- Deep Q-learning, an off-policy learning algorithm, is used as the core learning algorithm. It learns the action-value function Q(s, a) through temporal-difference learning and uses a neural network as a function approximator.
- Stabilization techniques are employed, including the use of fixed Q-targets, where separate online and target networks are used and periodically updated to avoid constantly moving targets.

Section 2: Implementation

Subsection 1: Algorithm

Deep Q-Network (DQN) is a very effective reinforcement learning technique that blends deep neural networks and Q-learning. It has achieved human-level competence in a variety of challenging tasks. The action-value function, written as Q(s, a), is at the heart of DQN and

expresses the predicted cumulative reward for doing action in state s.. The formula for updating the Q-value is given:

$$Q(s,a) \leftarrow (1-\alpha) \cdot Q(s,a) + \alpha \cdot (r + \gamma \cdot maxa'Q(s',a'))$$

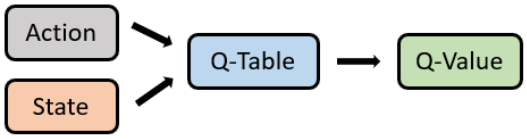
In this formula, α represents the learning rate, which determines the weight given to the new information compared to the existing Q-value. r is the immediate reward received after taking action in state s. γ is the discount factor that balances the importance of immediate and future rewards. s' is the new state obtained after taking action a, and a' is the optimal action to be taken in the new state according to the current Q-values. By iteratively updating the Q-values based on this formula, the agent learns to maximize its cumulative reward over time, leading to effective decision-making in complex environments.

One significant improvement is the adoption of fixed Q targets. In its original form the Q value utilized for training is reliant on the current parameters of the neural network. This causes the target to constantly fluctuate resulting in unstable training. However fixed Q targets address this issue by introducing a distinct target network with fixed parameters. The target Q values are then computed using this stable network. While the online network is updated incrementally. This modification enhances the training process by providing a more dependable target for learning.

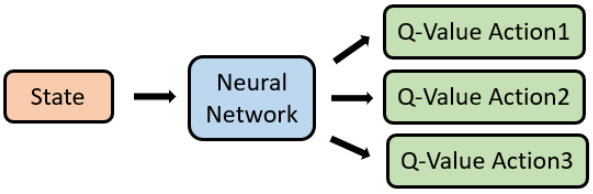
Subsection 2: Learning through Neural Nets

A neural network is utilized as a function approximator in the DQN algorithm to estimate the Q-values for various state-action pairings. The neural network's weight update algorithm uses gradient descent and backpropagation concepts to alter the network's parameters. Below is the sample diagram differentiating Q-learning and Deep Q-learning:

Q-Learning



Deep Q-Learning



Assume we have a neural network with weights w that receives a state s and outputs the Q-values for all feasible actions. The target Q-value for a particular action ***a*** in state ***s*** is calculated as:

The weight update rule used in DQN is based on the Mean Squared Error (MSE) loss between the predicted Q-value and the target Q-value. The weights ***w*** is updated by minimizing this loss using gradient descent. The weight update formula is as follows:

$$w \leftarrow w + \alpha * \nabla\_\mathrm{w}((\mathrm{Q\_target}(s,a)\ - Q(s,a))\text{^}2)$$

where α is the learning rate that controls the step size of the weight update, and ∇_w denotes the gradient with respect to the network weights.

By consistently updating the weights of the neural network utilizing this formula and actively engaging with the environment to gather experiential data, gradual enhancements are achieved in terms of Q-value estimations by said network.

Consequently, an optimal approach for executing a specific task is learned over time. It is significant to acknowledge that practical implementations typically incorporate supplementary methods such as experience replay and target network updates within the DQN algorithm in order to bolster and optimize learning processes within said neural networks.

Subsection 3: Experience Replay to Stabilize Training Neural Nets

Experience replay is a crucial improvement for the DQN algorithm. By keeping track of past experiences and selecting random batches during training this method enables better utilization of experience data and breaks the connection between consecutive samples. As a result, learning is greatly improved. In conclusion. The addition of fixed Q targets and experience replay has significantly enhanced the stability and effectiveness of the DQN algorithm. As a result. Training deep neural networks for reinforcement learning tasks has become more reliable and successful.

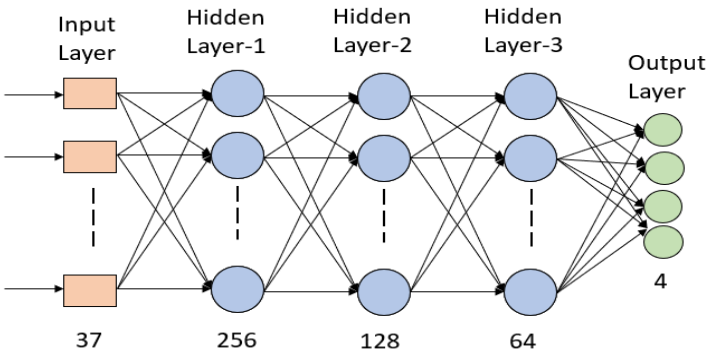


Fig: Model Architecture

Subsection 4: DQN Neural Network

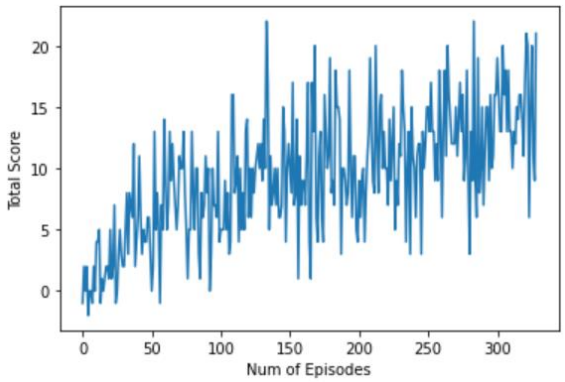
The neural network utilized in this implementation is multilayered MLP with nonlinear activation function to aid learning complex data. As shown in the above figure, it consists of 3 hidden layers,

with 256, 128, and 64 nodes respectively. Each hidden layer is followed by a Leaky Rectified Linear Unit (LeakyReLU) activation function. The network takes the state as input and outputs the action value function, providing the estimated value for each possible action in that particular state. Adam optimizer is used to optimize weights. Below are the hyperparameters used by the model:

Hyperparameters used	**Assigned values**
Learning rate	1e-3
TAU	2e-3
Relay buffer size	1e5
Batch size	64
Gamma (discount factor)	0.99
Update interval	4
Epsilon start	1.0
Epsilon end	0.025
Epsilon decay	0.85

Section 3: Results

From the below graph plotted between the number of episodes on X-axis and total score on Y-axis. After tweaking some parameters, the below results were obtained. The maximum number of episodes for solving was **229** episodes.




```
Episode 100     Average Score: 5.18
Episode 200     Average Score: 9.63
Episode 300     Average Score: 11.80
Episode 329     Average Score: 13.02
Environment solved in 229 episodes!     Average Score: 13.02
```

Conclusion and Ideas for Future Improvement

It's learnt from the above figure that DRL is unstable as the score keeps fluctuating during the whole training. Techniques like Double Q-Learning, Prioritized Relay and Fixed Q-Targets can be

used to solve the stability problem and improve agent's performance. Combining techniques like Prioritized experience relay and Dueling networks with Double Q-Learning can also be beneficial when compared to the traditional approach.

Improvements to original DQN algorithms like learning from multi-step bootstrap targets, Distributional DQN and Noisy DQN that were mentioned in the course can also be beneficial. Finally, fine-tuning the Neural network architecture with some better choice of number of neurons in hidden layers as well as selection of hyper parameters is also crucial without which it can lead to over-fitting or under-fitting phenomenon. Hence having an optimal parameter could lead to improved performance.