Udacity – Deep Reinforcement Learning Nanodegree

Navigation project

Deep reinforcement learning

Marwa Kandil
3-9-2021

Table of Content

1. Introduction

Artificial intelligence and machine learning algorithms are a major and essential part of our digital world. Using AI and ML machines try to mimic the human cognitive skills to solve problems. One of the most fascinating areas of AI is the concept of reinforcement learning (RL). In reinforcement learning machines learn from their own experiences and optimize their actions based on experience. Agent is rewarded or penalized based on their actions and the actions that get to the target outcome are rewarded. Agents of RL managed to learn actions in complex environments but are limited due to intensive computations.

Deep neural networks are another field of AI that exceled in learning advanced tasks such as object detection and sentiment analysis using layers of nodes that are connected similar to human neurons. Recently, RL is combined with deep neural network resulting deep Q-network (DQN) surpassed all previous algorithms and was able to learn concepts such as object categories directly from raw sensory data [1].

Deep reinforcement learning has achieved human-level scores in other games such as chess, poker, Atari games [1]. There are many open-source tools train DRL agents to encourage large-scale innovation for deep reinforcement learning. One of the main open-source tools that provide intelligent agents and rich environment for game development is Unity Machine Learning Agents (ML-Agents). The ML-agents Train and embed intelligent agents by leveraging state-of-the-art deep learning technology [2]. In this project I used Unity's rich environments to design, train, and evaluate deep reinforcement learning algorithms.

For this project, I trained an agent to navigate and collect yellow bananas in a large, square world that contains yellow and blue bananas as shown below. The environment has state space of 37 dimensions and contains the agent's velocity, and ray-based perception of objects around the agent's forward direction. The agent can choose from 4 actions: forward, backward, turn left and turn right. The agent gains a reward of +1 when collecting yellow banana and penalized as -1 when collecting blue bananas. The aim of the agent is to maximize reward, thus, collect as many yellow bananas as possible. The task is episodic and is solved when the average score is 13 for 100 consecutive episodes. I used two algorithms to solve this problem. The first algorithm used was vanilla Deep Q-network (DQN) and the second algorithm used is a Double DQN with soft-update.

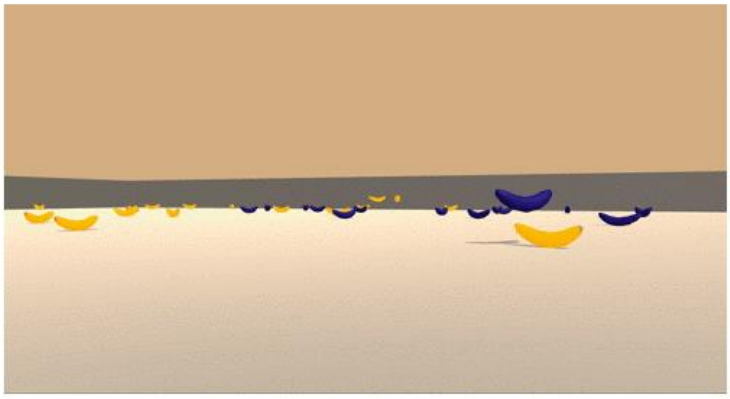

Figure 1: Unity ML-agent Banana collector

2. Background

Deep reinforcement learning combines function approximation and target optimization by combining deep artificial neural networks and reinforcement learning. Reinforcement learning framework consists of five main parts: states, actions, environment, agent, and rewards [3]. States are the different situations returned by the environment. Actions are set of possible moves that the agent can take to optimize the reward. The agent is a software that utilizes a policy to determine the next action based on the current state. The reward is the feedback by which the agent measures the success or failure of an action in each state. The agent's role is to evaluate different policies and identify the policy that maximizes reward. Thus, the agent evaluates the expected long-term return of following the agent's policy also known as the value function V(S). The value function uses discounted rewards as the effect of the reward is reduced the further away we move in the future. A better way to assess the policy is to use action-value function Q(S,A), it is the long-term return of an action taken under policy at the current state. The RL agent uses Bellman equation as an iterative update to estimate the action-value function such that:

$$Q^*(s,a) = E_{s'}[r + \gamma max_{a'} Q^*(s',a')|s,a]$$

Where the variables s and a stand for state and action, Q^*(s,a) stands for action-value function for s and a, γ stands for reward discount rate. Any function approximator can be used to estimate the action-value function, such as the non-linear function approximators of neural networks. Researchers at Deepmind designed a neural network that uses non-linear function approximators to estimate the action-value function called Deep Q-network (DQN) [1]. DQN has a Neural Network (NN) at its core that is used as function approximator producing a vector of action-values with max value indicating the best action that will generate the highest reward. The function approximator uses weight parameters Θ to reduce the mean squared error of the predicted Q^*(s,a) and train the NN. At each stage of the optimization, the previous iteration parameters Θ^- are fixed when optimizing the ith loss function L(Θ) resulting in optimization problem. The loss function is differentiated to minimize the loss and generating the gradient:

$$+_{\Theta} L(\Theta_i) = E[\big(r + \gamma max_{a'} Q^*(s',a',\Theta^-) - Q(s,a,\Theta_i)\big) + Q(s,a,\Theta_i)]$$

Stochastic gradient decent is used to maximize the Loss function. Due to the high correlation between actions and states some DQN will not be steady, and the weights can diverge. Thus, two techniques must be used: experience replay and fixed Q-Targets. Experience replay uses replay buffer that stores a set of states and actions tuples sequentially but sample the tuple randomly. This reduces the problem to a supervised learning problem and allowing for make better use of previous experience. Fixed Q-targets are used as previously we update a guess by another guess that can cause more harmful correlations. Thus, the use of a separate target network which uses weights that are not changed during the learning step reduces correlation.

DQN tends to overestimate action values as Q value update depends on choosing the max Q values based on the set of state and actions, however, initially the values are random. Thus, its needed to compare the selected Q value against a different set of parameters to ensure it is evaluated correctly. One of the improvements to the DQN is the Double DQN where two separate action value functions are used, one for update and a second for evaluation [4]. Both functions must agree on the best action value to optimize the Q-value function.

3. Implementation and Results

In this paper I implemented DQN and DDQN to solve the Banana collector problem. The neural network model That I used consists of one input layer that has same shape as the number of states, four linear hidden layers with number of nodes 32,64,128 and 256. Finally, I used dropout layer with dropout probability of 0.3 to reduce overfitting, and an output layer with the number of possible actions as the output. For training, Adam optimizer was used with learning rate of 0.0005.

For both implementations, DQN and DDQN, I used a replay buffer of size 100,000 unit and 64 mini batch size and discounted factor (ɣ) of value 0.99. To ensure balanced trade off between exploration and exploitation, epsilon-greedy action selection algorithm is used where random actions are selected at the rate epsilon, otherwise actions are selected greedily. For training, the epsilon rate starts at 1.0 to ensure maximum exploration, the rate decays as training proceeds reaching a minimum of 0.01 at the end of training to ensure exploitation. The scores through out the episodes were tracked and presented on the plot below every 100 episodes. As shown the DQN model solved the problem in 397 episodes and the DDQN solved the model in 396 episodes. Both models had similar performance, however, the training in Figure 3 of DDQN was much smother than DQN training in Figure 2. The strong correlation between state and actions in DQN could have caused overestimation at the beginning of training, thus, it had more fluctuations. The use of evaluation and update function approximators in DDQN made the model more stable especially during the initial training.

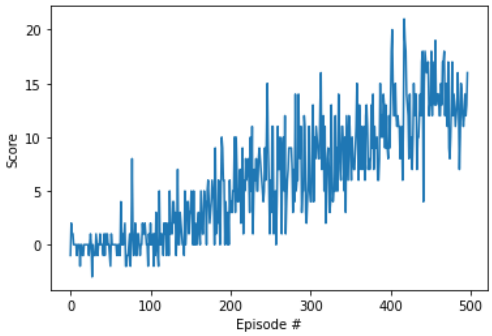

Figure 2: DQN average score vs training episodes

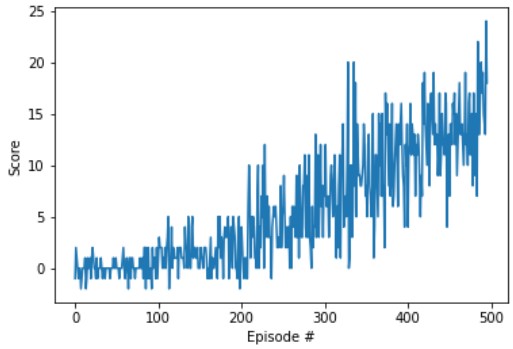

Figure 3: DDQN average score vs training episodes

Algorithm	Total number of episodes	Average Score	Test score
DQN	397	13.04	13.0
DDQN	396	13.06	16.0

4. Conclusion and Future Work

Deep reinforcement learning is showing great potential in multiple fields. In this project, I implemented DQN and Double DQN to train and evaluate an agent to collect as much yellow bananas as possible in the unity ML-agent environment Banana collector. The results show that using DQN the agent solved the problem in 397 episodes, on the other hand, Double DQN solved the problem in 396 episodes.

Future work to the current solution would be to experiment with other DQN improvements such as Dueling DQN and prioritized experience replay. Furthermore, combination of multiple improvements can be considered as represented in the Rainbow paper [5].

5. References

[1]: V. Mnih et al., "Human-level control through deep reinforcement learning", Nature, vol. 518, no. 7540, pp. 529-533, 2015. Available: 10.1038/nature14236 [Accessed 3 September 2021].

[2]: U. Technologies, "Machine Learning Agents | Unity", Unity, 2021. [Online]. Available: https://unity.com/products/machine-learning-agents. [Accessed: 03- Sep- 2021].

[3]: R. Sutton and A. Barto, Reinforcement Learning, 2nd ed. 2019.

[4]: H. van Hasselt, A. Guez and D. Silver, "Deep Reinforcement Learning with Double Q-learning", 2015. [Accessed 3 September 2021].

[5]: M. Hessel et al., "Rainbow: Combining Improvements in Deep Reinforcement Learning", 2021. [Accessed 3 September 2021].