5012 Final Project:
Reinforcement Learning with AI Agent

Group 1

Mukhit Ismailov
Jiasi Wu
Haomao Wang
Stefan Juang
Muhammad Talha SAMI

Abstract

Reinforcement Learning (RL) is an area of machine learning in which an agent is trying to find optimal policy or sequence of actions in the environment to maximize overall reward. These formulation of the problems of RL are relatively old, however with the advent of Neural Networks this area got a new wave of excitement and research. In this project we are trying to build most common algorithms for RL for the agents of the OpenAI Gym environment.

1. Introduction

Reinforcement Learning algorithms allow us to solve a wide array of unusual problems. In some situations even researches are surprised by the solutions that the algorithms are able to produce. In this project we are exploring 4 algorithms on Mountain Car environment of OpenAI Gym. They are epsilon-greedy Q-Learning, Deep Q Network, Policy Gradient, and Actor-Critic. This report has the following structure: section 2 talks about Environment, section 3 shows algorithms, section 4 describes results, section 5 is a conclusion.

2. Environment

For our project we use OpenAI Gym infrastructure. There are a number of different environments like some atari games or what we used Mountain Car. In the game Mountain Car there is a car on a one-dimensional track positioned between two “mountains”. The goal is to drive up the mountain on the right. However the car's engine is not strong enough to scale the mountain in a single pass. Therefore, the only way to succeed is to drive back and forth to build up momentum. Thus there are two actions to choose from and our task is to guide the car to the top of the mountain by learning from the reward based on the action as well as the current state.

Since the car has continuous states such as velocity and position we decided to treat the states also in discrete ways in addition to the continuous ones. We divided the value into different bins with the same interval and every bin was represented as one value. So the states value would be mapped into discrete ones.

3. Reinforcement Learning Environment:

OpenAI gym Mountain Car Problem

4. Programming Language and Libraries Used:

Python:
Tensorflow, numpy, gym, matplotlib, pyvirtualdisplay
(pip3 install if library does not exist)

5. Contribution:

1. **Cartpole with Random Exploration**
   Haomao Wang
   Cartpole is one of the most fundamental problem in reinforcement learning, I have used random exploration to explore the state space. The action is written into the replay memory for the machine learning algorithm to train and run.

2. **Video and Sound Editing**
   Haomao Wang
   (13 Hours)

3. **Q Learning with epsilon greedy with decay policy**
   Contributor: Mukhit Ismailov
   Q Learning algorithm is one the most popular approaches when we do not have information about environment state transition function and/or reward distribution function. I am using epsilon greedy with decay update policy. During training process we have make a tradeoff between exploration and exploitation. Epsilon is the probability we take random action, and we take an action based on our policy with probability of (1-epsilon). Decay parameter indicates that over time we want to make less random choices. That means that in the future we have more confidence in our policy that we rarely do a random exploration.
   Below is the psudo-code from Sutton and Barto book on Reinforcement Learning:

Sarsa: An on-policy TD control algorithm

Initialize $Q(s,a), \forall s \in \mathcal{S}, a \in \mathcal{A}(s)$, arbitrarily, and $Q(\textit{terminal-state}, \cdot) = 0$
Repeat (for each episode):
  Initialize S
  Choose A from S using policy derived from Q (e.g., ϵ-greedy)
  Repeat (for each step of episode):
    Take action A, observe R, S'
    Choose A' from S' using policy derived from Q (e.g., ϵ-greedy)
    $Q(S,A) \leftarrow Q(S,A) + \alpha\big[R + \gamma Q(S',A') - Q(S,A)\big]$
    $S \leftarrow S'$; $A \leftarrow A'$;
  until S is terminal

Findings: The algorithm is able to get to the top of the mountain relatively fast. However, there is no consistency. The reason for this is because we get a random initial position.

4. **DQN: (Baseline Model)**
   Stefan Juang
   DQN is implemented in Tensorflow.
   In particular, Bellman's Equation is implemented as the discount reward function.

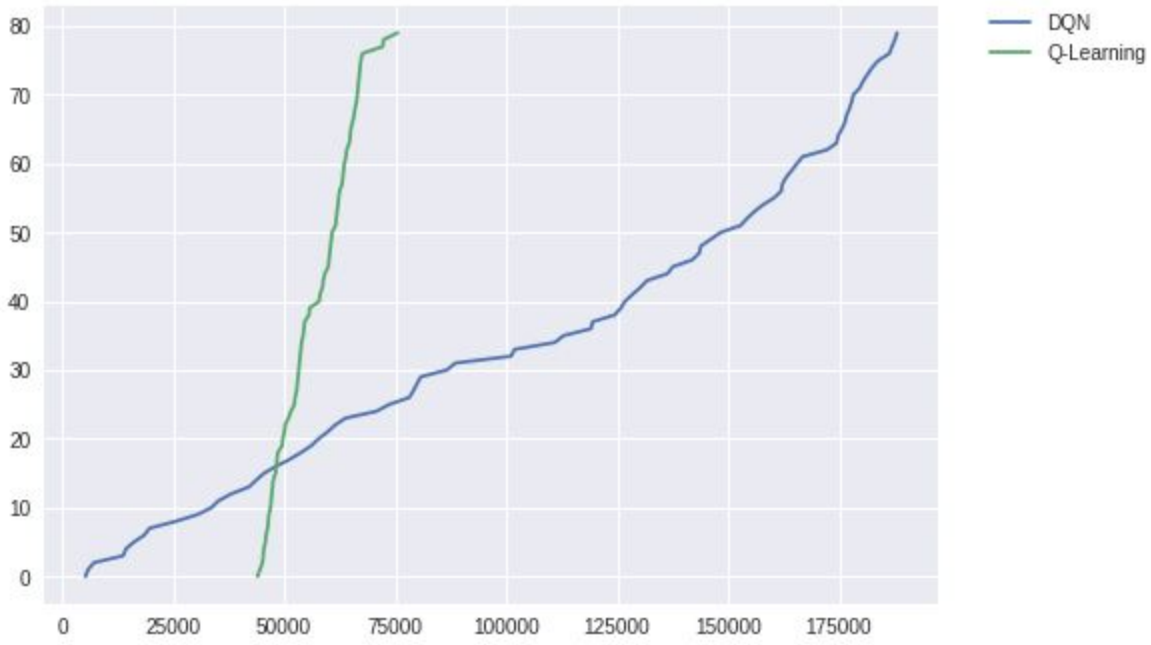
   Test and Discoveries:
   I have tested both deeper (7 layers) and wider (each 1000 nodes) networks to test convergence.
   Without fixed target being implemented, the network learning goal is constantly changing. The result is the network does not converge. Episode will run on more than 50000. After shrinking the network down to one or two layers, the training stabilize and can actively solve the game after 50 episodes of training in ~3000 frames. Gamma value is also tested. When gamma value falls below 0.9 the agent will start to experience difficulty training. Average run increases to ~5000 or more.When gamma is set to 1, average run could achieve sub 2000 frames.

   Acknowledgement:
   Training is done on Google Colab platform with GPU option enabled.
   Base framework utilized: https://bit.ly/2PcCddU

5. **Reward Modification with Domain Knowledge:**
   Stefan Juang
   While using the previous two layer network, I modified the reward function by multiplying the environment reward by inverse of position as well as inverse of velocity at each frame. The intuitive thinking is to give agent a better feedback function instead of getting negative 1 at every time step.

   Test and Discoveries:
   The interesting finding is that this does not help the agent as much as it has hurt it. My guess is with two multipliers the reward is drifting multiplicatively, despite having better represented reward feedback, the agent will have an unstable value state each time.
   The average frame to solve the problem after 50 episodes training is ~5000.
   By only multiplying reward function with inverse of velocity training stabilizes and agent is able to perform active game solving after 50 episodes of training with ~800 frames.

   Mukhit and Stefan's algorithm comparison:



Q-Learning is set with a learning rate of 0.5 which learns much faster than DQN (0.01 learning rate), but it is not to say that DQN is not a good algorithm for this problem. As good training runs picks up, DQN's learning curve towards the end is comparable to Q-Learning's.

6. Fixed DQN Research and Report

Jiasi Wu

By using the Bellman's Equation, TD target, namely discount reward function in our implementation, will be the reward of taking certain action in the current state plus the discounted highest Q value for the next state.

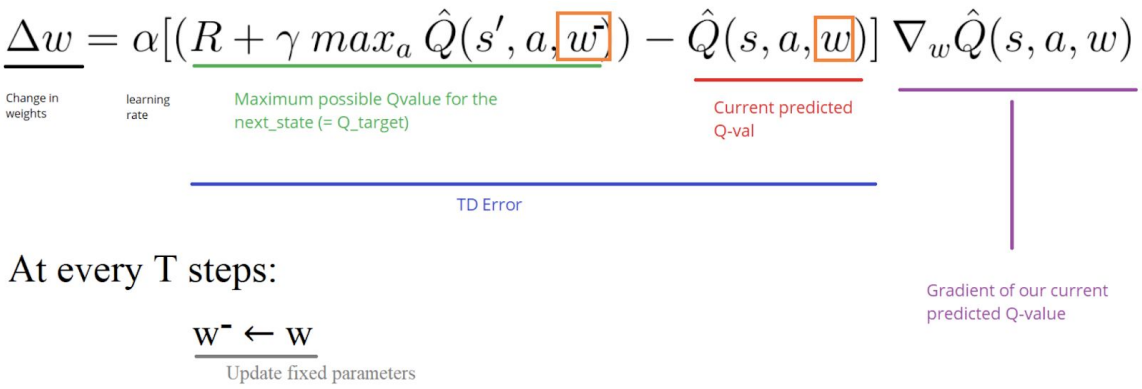
$$Q(s,a) = r(s,a) + \gamma max_a Q(s',a)$$

And TD target will be applied in the calculation of the loss during the backpropagation to get the difference between that and the Q value we estimated. Since the TD target is not determined the problem would occur if we use the same parameters for estimating both target and predicted Q value. There will be a big correlation between the target and the parameters we constantly change, which leads to a big oscillation in training.

We tried the idea of fixed Q-targets introduced by DeepMind:

- Using a separate but similar network with a fixed parameter (w- shown in the picture) for estimating the TD target.
- At every T step, we copy the parameters from our DQN network to update the target network. Here T is a hyperparameter we should set at advance.

So the loss gathered on the weights would be:



At every T steps:

$\bar{w} \leftarrow w$

Update fixed parameters

By using this method the training process would be more stable.
We has been working on the implementation of fixed Q target yet so far we haven't got a result. The realized framework and structure is shown below.

Different from the baseline model, we sealed the network structure into a class so that we could directly create two networks for Target and predicted Q value.

And the loss was based on the difference between them other than the softmax of the action category.

```
# The loss is the difference between our predicted Q_values and the Q_target
self.loss = tf.reduce_mean(tf.square(self.target_Q - self.Q))
```

Then a function to update the parameters from another network was created:

```
def update_target_graph():
    # Get the parameters of our PredictionNetwork
    from_vars = tf.get_collection(tf.GraphKeys.TRAINABLE_VARIABLES, "PredictionNetwork")

    # Get the parameters of our Target_network
    to_vars = tf.get_collection(tf.GraphKeys.TRAINABLE_VARIABLES, "TargetNetwork")

    op_holder = []

    # Update our target_network parameters with PredictionNetwork parameters
    for from_var, to_var in zip(from_vars, to_vars):
        op_holder.append(to_var.assign(from_var))
    return op_holder
```

Finally The parameters of target network were updated every T time during the iteration:

```
if Time > T:
    update_target = update_target_graph()
    sess.run(update_target)
    Time = 0
```

We treat this fixed Q-targets method as our future work and we may encounter some memory resource issues during the training and need to tune that later on.

7. **Policy Gradient: (Reinforce)**
   Talha Sami
   Policy Gradient (reinforce) method implemented using Tensorflow and Sklearn.

   Test and Discoveries:
   We decided to use Policy gradient (as opposed to value based methods) for the continuous state space problem as the state space would have been too large to store a table of Q values for each state.

   Learning made huge leaps initially but as we inched closer to the total rewards of 0, the gradient change became smaller. As a result, our updates to the policy become smaller and smaller.

   We tweaked the hyperparameters such as the learning rate to try to get our algorithm out of the local minima, but even after significant training, we still could not get our algorithm to converge and successfully climb to the hilltop.

8. **Actor Critic**
   Talha Sami
   Actor Critic method implemented using Tensorflow and Sklearn (for featurizing the state space)

   Test and Discoveries:
   Convergence was seen to be faster with actor critic methods as compared to policy gradient methods. However, undesirably large variance was seen in the results which we could not get rid of by hyperparameter tuning.

9. **Advantage Actor Critic (A2C)**
   Talha Sami
   Advantage Actor Critic method implemented using Tensorflow and Sklearn (for featurizing the state space)

   Test and Discoveries:
   Convergence was even faster and far less variant as we introduced a value based baseline function. Within a couple of hundred episodes, we started seeing consistent convergence.