Deep Reinforcement Learning on Car Racing Game

Jiahao Zhang, Minghe Ren, Weixuan Jiang, Zhonghao Guo
{jiahaozh, sawyermh, jwx0728, gzh1994}@bu.edu

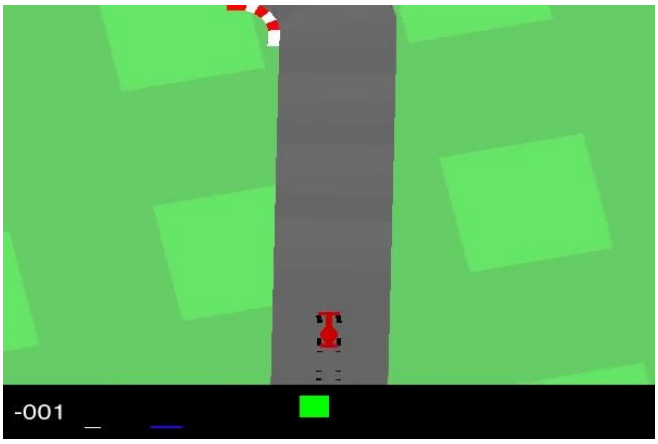

Figure 1.Car Racing Game powered by OpenAI Gym

1. Task

Our project will explore the algorithms of reinforcement learning (RL) used in game playing. We intend to study several RL algorithms and implement them in a car racing game. The RL methods include, but not limited to, DQN, Double DQN, and Dueling DQN. After the RL methods been implemented, we will make comparison between these results and analyze the characteristics of each method.

2. Related Work

A well-known success story of reinforcement learning on games is *Playing Atari with Deep Reinforcement Learning*[1]. It introduced the Deep Q-learning, a new reinforcement learning which was for the first time that researchers successfully learned control policies directly from high-dimensional sensory input using reinforcement learning with deep learning model[2]. However, there was a problem that using the same parameters (weights) for estimating the target and the Q value results in a big correlation between the TD target and the parameters (w) we are changing. The idea of fixed Q-targets solved the problem.

Later in 2016, two modified version DQN, Double DQN and Dueling DQN were proposed in [3] and [4]. DDQN was introduced by Hado van Hasselt. This method handles the problem of the overestimation of Q-values and, as a consequence, helps us train faster and have more stable learning. While for dueling-DQN(aka DQN), we decompose Q(s,a) as the sum of V(s)(the value of being at that state) and A(s,a) (the advantage of taking that action at that state). And then we combine these two streams through a special aggregation layer to get an estimate of Q(s,a). By decoupling the estimation, our DDQN can learn which states are (or are not) valuable without having to learn the effect of each action at each state.

3. Approach

Introduce DQN DDQN and Dueling DQN

We are going to implement three algorithms of RL in our application. They're DQN, Double DQN, and Dueling DQN. We will use tools like Tensorflow, OpenAI as our training and experiment platforms. We will write the source code of these algorithms and apply them on a existing game from OpenAI Gym.

Popular methods of RL can be roughly divided into to two categories: DQN and Policy Gradients. Although there are other algorithms that intend to improve the performance of both, we chose to focus on DQN and its subsidiaries as a startpoint of our project.

Deep Q Learning (DQN) adopts its idea from Q-learning. It can handle more complicate problems than Q-learning and also with less space (memory) complexity. DQN utilizes a Neural Network to estimate the Q-value function. Usually the input for the network is the current state, while the output is the corresponding Q-value for each of the action [5].

Double Deep Q Learning (DDQN) intends to solve the overestimating issue when approximating action values in DQN, which cause slow convergence. Here, two networks are used to decouple the action selection from the Q value function.

Dueling Deep Q Learning (Dueling DQN) decomposes the Q-value function into one for the state value function and one for the state-dependent action advantage function [4]. This architecture usually helps

to accelerate the training process and produce better performance.

4. Dataset and Metric

Reinforcement learning describes the set of learning problems where an agent must take actions in an environment in order to maximize some defined reward function.

Unlike supervised deep learning, large amounts of labeled data with the correct input output pairs are not explicitly presented.

In our car racing game, we don't have existing dataset. We imagine to set up an interesting game environment where a racing car can autodrive along the road. And it can not hit on the roadblocks. Here, agent is the racing car. We will track its actions and document corresponding rewards to construct its own dataset.

After searching into open source Open AI gym library https://github.com/openai/gym, we can learn from that to establish our own game environment.

Preliminarily, we can evaluate the results by documenting its moving coordinates and measuring the probability of driving out of road or hitting on roadblocks. Later on, we will have more detailed metrics to do evaluation.

5. Preliminary Results

5.1 Environment

We use Car-Racing-v0 from gym library as our basic game environment, which is easiest top-down racing environment to learn from data. When the environment is started up, indicators shown at the bottom of the window and the state RGB buffer. From left to right: true speed (white), four ABS sensors (blue), steering wheel position (green), gyroscope (red).

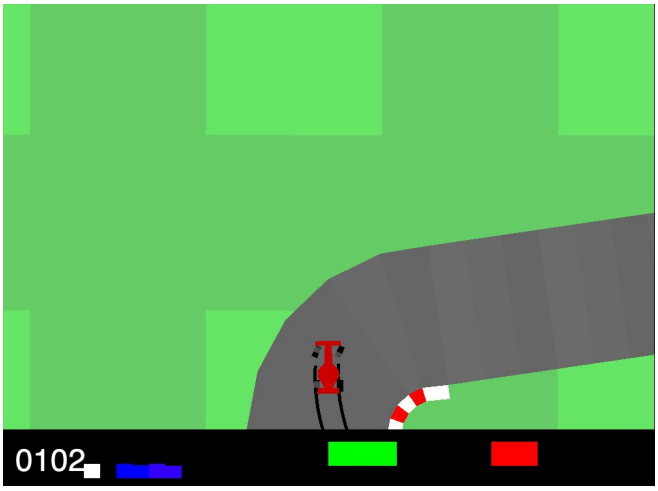

Figure 2. Reward and Indicator

The number shown in the graph is reward. Reward is -0.1 every frame. For example, if you have finished in 732 frames, your reward is 1000 - 0.1*732 = 926.8 points. Game is solved when agent consistently gets 900+ points. Track is random every episode. Episode finishes when all tiles are visited. Car also can go outside of playfield, that is far off the track, then it will get -100 and die.

At the same time, it also return the car's action every 200 steps or when it goes outside of playfield or one episode finishes. The action actually is monitoring which key is pressed by user. The screenshot below shows an example. The action map contains three parts. First value represents left or right. If left then -1 and if right then +1. Second value shows up or down. If up then +1. And third value is down which means braking. If down then +0.8.

```
action ['+0.00', '+0.00', '+0.80']
step 1200 total_reward -68.44

action ['+1.00', '+0.00', '+0.00']
step 1400 total_reward -88.44

action ['+0.00', '+1.00', '+0.00']
step 1600 total_reward -108.44
```

Figure 3. Action Map

Besides, accompanied with car's actions, picture will be created every 200 steps. We call it state in our model. A state consists of 96×96 pixels. It is the input of our reinforcement model.

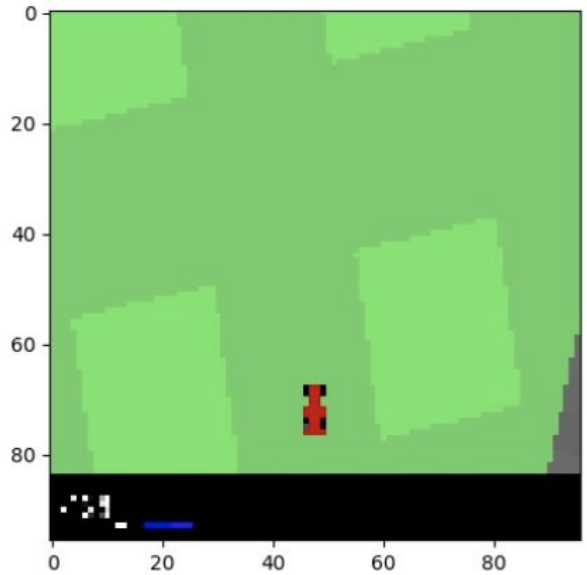

Figure 4. One State

5.2 Q-learning

Combining the Q-learning algorithm, our goal is to interact with the emulator by selecting the best actions in order to maximize future rewards.

Based on the Algorithm 1 *Deep Q-Learning with Experience Reply*[1], we design several algorithms to build our DQN.

Algorithm 1 Deep Q-learning with Experience Replay

Initialize replay memory $\mathcal{D}$ to capacity N
Initialize action-value function Q with random weights
for episode $= 1, M$ **do**
  Initialise sequence $s_1 = \{x_1\}$ and preprocessed sequenced $\phi_1 = \phi(s_1)$
  for $t = 1, T$ **do**
    With probability ϵ select a random action a_t
    otherwise select $a_t = \max_a Q^*(\phi(s_t), a; \theta)$
    Execute action a_t in emulator and observe reward r_t and image x_{t+1}
    Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$
    Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in $\mathcal{D}$
    Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from $\mathcal{D}$
    Set $y_j = \begin{cases} r_j & \text{for terminal } \phi_{j+1} \\ r_j + \gamma \max_{a'} Q(\phi_{j+1}, a'; \theta) & \text{for non-terminal } \phi_{j+1} \end{cases}$
    Perform a gradient descent step on $(y_j - Q(\phi_j, a_j; \theta))^2$ according to equation 3
  end for
end for

We haven't finish the reply memory part.

Preprocess part already introduced in 5.1. And, for this part:

With probability ϵ select a random action a_t
otherwise select $a_t = \max_a Q^*(\phi(s_t), a; \theta)$

We implement as below:

```
if np.random.rand() > self.epsilon:
    action_idx = self.session.run(
                    self.best_action)
else:
    action_idx = self.get_random_action()
```

The best action is the output of our Deep Q-Networks which will be introduced in 5.3. At here, we use np.random.rand() compared with a constant variable $epsilon$.

After get the action_idx, we can get the best action by indexing the action map.

```
action = self.action_map[action_idx]
```

Then, execute action a_t in emulator.

```
observation, r, done, info = self.env.step(action)
if self.render:
    self.env.render()
reward += r
```

It will be three main outcome after render the action. *observation* is the screenshot, *r* is this step reward which should be added to the total reward, *done* is a boolean variable which is True when the car is jump out the border.

The gradient descent part will be introduced in 5.3.

5.3 Deep Q- Networks

In traditional Q-learning algorithm, it's typically a linear function approximator used to estimate the action-value function (Q function). We will （maintain a Q table, of which rows and columns indicating states and actions. Unfortunately, it doesn't work when input states are raw images, for the Q table could be tremendous. A nonlinear function estimator, such as neural network is adapted instead. The Q-network is such a neural network with weight parameter θ.

(1) Loss function

A sequence of loss functions that changes at each iter *i* are defined as,

$$L_i(\theta_i) = \mathbb{E}_{s,a\sim\rho(\cdot)}[(y_i - Q(s,a;\theta_i))^2]$$

where $y_i = \mathbb{E}_{s'\sim}[r + \gamma max_{a'} Q(s', a', \theta_{i-1})|s, a]$ is the target for iter i and $\rho(s,a)$ is the behaviour distribution. Differencing the loss function w.r.t θ and we arrive at the gradient,

$\nabla_{\theta_i} L_i(\theta_i) =$

$\mathbb{E}_{s,a\sim\rho(\cdot);s'\sim}[(r + \gamma max_{a'} Q(s', a'; \theta_{i-1}) - Q(s,a;\theta_i))\nabla_{\theta_i} Q(s,a;\theta_i$

Then we will update the weight by using SGD.

(2) Model Architecture

Some previous papers estimated Q-values by feeding in history-action pairs. The drawback is that a separate forward pass is required for each action. Here, we use an trick to build a separate output unit for each possible action, and the input is the state representation. Hence, with a single forward pass through network, we get Q-values for all possible inputs. The exact architecture used here are slightly different from the one in paper[1]. The input raw image to the neural network is an 96x96x4 image produced by preprocessing function ϕ on last 4 frames of the history and stack them together. The first conv layer convolves 8 7x7 filters with stride 3. The second layer is a 2x2 maxpooling layer with stride 2. The third conv layer convolves 16 3x3 filters with stride 1. Then a maxpooling layer again. The final hidden layer is a FC layer with 256 units. The output layer is a FC layer with a single output for each valid action. The code here is an implementation of NN above,

```
def create_network(input):
    wr = slim.l2_regularizer(1e-6)
    input_t = tf.transpose(input, [0, 2,
3, 1])
    net = slim.conv2d(input_t, 8, (7, 7),
```

```
data_format="NHWC",
activation_fn=tf.nn.relu, stride=3,
weights_regularizer=wr)
    net = slim.max_pool2d(net, 2, 2)
    net = slim.conv2d(net, 16, (3, 3),
data_format="NHWC",
            activation_fn=tf.nn.relu,
weights_regularizer=wr)
    net = slim.max_pool2d(net, 2, 2)
    net = slim.flatten(net)
    net = slim.fully_connected(net, 256,
activation_fn=tf.nn.relu,
            weights_regularizer=wr)
    q_state_action_values =
slim.fully_connected(net,
self.dim_actions, activation_fn=None,
weights_regularizer=wr)

    return q_state_action_values
```

Todo list:

- Implement experience replay
- Implement whole structure
- Fine Tuning parameters

6. Detailed Timeline and Roles

Task	Deadline	Lead
Run test on DQN	11/25/2018	all
Implement DDQN, Duel DQN	12/2/2018	all
Run tests on DDQN, Duel DQN	12/9/2018	all
Prepare report and presentation	12/16/2018	all

References

1) Mnih V, Kavukcuoglu K, Silver D, et al. Playing atari with deep reinforcement learning[J]. arXiv preprint arXiv:1312.5602, 2013.
2) Mnih V, Kavukcuoglu K, Silver D, et al. Human-level control through deep reinforcement learning[J]. Nature, 2015, 518(7540): 529.
3) Van Hasselt H, Guez A, Silver D. Deep Reinforcement Learning with Double Q-Learning[C]//AAAI. 2016, 2: 5.
4) Wang Z, Schaul T, Hessel M, et al. Dueling network architectures for deep reinforcement learning[J]. arXiv preprint arXiv:1511.06581, 2015.
5) Steeve Huang, Introduction to Various Reinforcement Learning Algorithms. Part I (Q-Learning, SARSA, DQN, DDPG), https://towardsdatascience.com/@huangkh19951228