# Machine Learning Nanodegree
# Capstone Project
# A Game Bot trained by Deep Q-Learning

Peng Xu

March 6, 2018

## 1 Definition

### 1.1 Project Overview

Reinforcement Learning is a type of machine learning that allows you to create AI agents that learn from the environment by interacting with it. Just like how we learn to ride a bicycle, this kind of AI learns by trial and error. As seen in Figure 1, the brain represents the AI agent, which acts on the environment. After each action, the agent receives the feedback. The feedback consists of the reward and next state of the environment. The reward is usually defined by a human. If we use the analogy of the bicycle, we can define reward as the distance from the original starting point.

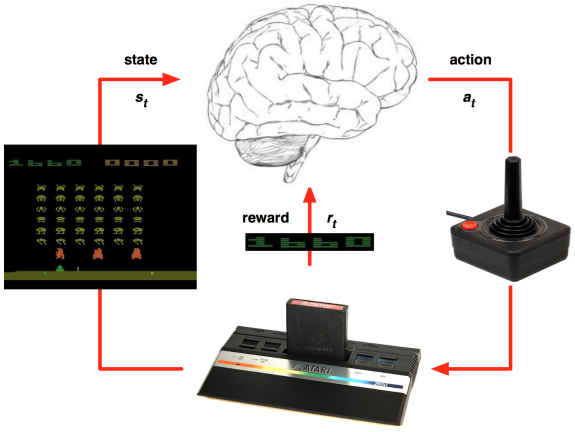



**Figure 1:** How an agent interacts with the environment.

Google's DeepMind published its famous paper Playing Atari with Deep Reinforcement Learning, in which they introduced a new algorithm called Deep Q Network (DQN for short) in 2013. It demonstrated how an AI agent can learn to play games by just observing the screen without any prior information about those games. The result turned out to be pretty impressive. This paper opened the era of what is called ?deep reinforcement learning?, a mix of deep learning and reinforcement learning.

### 1.2 Problem Statement

In this project, a deep reinforcement learning method, Deep Q Network, would be implemented and applied to play a Coast Racer game in OpenGym / Universe using TensorFlow.

Q-learning, a model-free online off-policy algorithm, whose main strength is that it is able to compare the expected utility of the available actions without requiring a model of the environment. In Q-Learning Algorithm, there is a function called Q Function, which is used to approximate the reward based on a state. We call it Q(s,a), where Q is a function which calculates the expected future value from state

s and action a. Similarly in Deep Q Network algorithm, we use a neural network to approximate the reward based on the state. We will discuss how this works in detail.

OpenAI Gym is a toolkit for reinforcement learning research. It includes a growing collection of benchmark problems that expose a common interface, and a website where people can share their results and compare the performance of algorithms. Universe is a software platform for measuring and training an AI's general intelligence across the world's supply of games, websites and other applications. Universe allows an AI agent to use a computer like a human does: by looking at screen pixels and operating a virtual keyboard and mouse. We must train AI systems on the full range of tasks we expect them to solve, and Universe lets us train a single agent on any task a human can complete with a computer. With Universe, any program can be turned into a Gym environment. Universe works by automatically launching the program behind a VNC remote desktop. Hundreds of games have been translated into Gym environments and are ready for reinforcement learning, which mostly can be freely run with the universe Python library as follows:

```
import gym
import universe # register Universe environments into Gym

env = gym.make('flashgames.DuskDrive-v0') # any Universe environment ID here
observation_n = env.reset()

while True:
  # agent which presses the Up arrow 60 times per second
  action_n = [[('KeyEvent', 'ArrowUp', True)] for _ in observation_n]
  observation_n, reward_n, done_n, info = env.step(action_n)
  env.render()
```

Among the several racing car games provided in Universe, the Coaster Racer flash game arose in front of me since it could be a typical simulation of Autonomous Driving, in which a vehicle is simply controlled by 3 inputs, left, right, forward. It is expected that the racing car can learn a smart driving behavior after a series of training steps leading to a maximal reward or namely score here. The trained bot for the Coaster Racer flash game will determines whether it should turn and which way it turns.

**Target function:**

$Q^{\pi} : \mathcal{S} \times \mathcal{A} \rightarrow \mathbb{R}$, where $\mathcal{S}$ is the set of *states* and $\mathcal{A}$ is the set of *actions* (turn left, forward or turn right), and $\mathbb{R}$ represents the value of being in a state $s \in \mathcal{S}$, applying a action $a \in \mathcal{A}$, and following policy $\pi$ thereafter.

**Target function representation:**

Deep neural network.

Therefore, I seek to build a Q-learning agent trained via a deep convolutional neural network to approximate the optimal action-value function:

$$Q^*(s,a) = \max_{\pi} Q^{\pi}(s,a), \forall s \in \mathcal{S}, a \in \mathcal{A} \tag{1}$$

which is the maximum sum of rewards achievable by a behavior policy $\pi$.

## 1.3 Metrics

Since Q learning is recursively defined, we did not consider fitting errors such as RMSE to be good metrics for evaluation. Instead, given our ultimate our goal to make a good game AI, we evaluated gameplay directly in order to gauge performance. We evaluated the performance of our algorithms using two metrics: game length and game score. Game length is the number of moves the player makes before they reach a game over, and game score is the score based on the number of lines they clear. While the game score is ultimately more important, it can only be used to judge the performance of sufficiently advanced networks because there is a large initial barrier to overcome before the algorithm can score with any degree of consistency. Thus, game length, which was typically correlated with game score, is a

good metric for showing intermediate performance since an algorithm can incrementally learn to survive longer.

# 2 Analysis

To tackle the problem described in Section 1.2, we will use Reinforcement learning with Deep Learning to automatically learn evaluation functions by playing games by itself. Unlike other approaches that need a very large dataset, this approach will try to learn to play games without any domain knowledge (no dataset will be used). This is a promising approach for creating game-playing algorithms for playing other two-player games of perfect information.

## 2.1 Data Exploration and Visualization

### 2.1.1 Coaster Racer Game environment

The browser screenshot as shown in Figure 2 is both for the user and the AI Game Bot. Namely, that is the raw vision of the Game Bot which apparently contains lots of pixels useless for deciding the steering. For more efficient and faster computing, some preprocessing procedures would be done before we throw the image data into a deep learning model, which will be described in later paragraph.

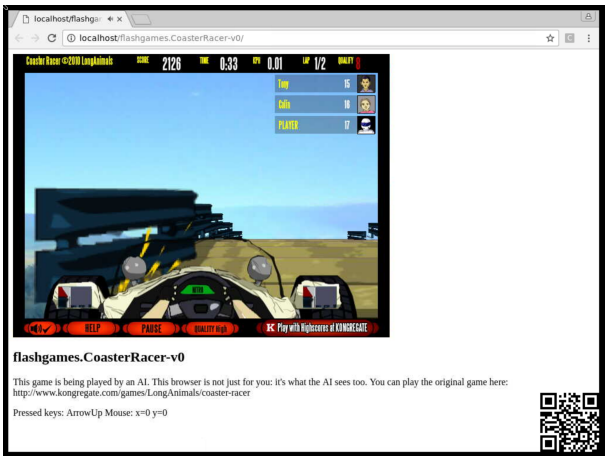



**Figure 2:** A browser screenshot of Coaster Racer Game.

### 2.1.2 States

For the purposes of winning the game, the state of the game at time t is simply the pixel array of the game screen at that particular time. I experimented with using only a cropped version of the game screen that contained the view of the driver. In order to avoid unnecessary computation and memory usage, we downsample this image. By reducing the size of each state, this allows us to fit more training examples in our replay history without running out of memory, which is a crucial component of the experience replay technique we used. In addition, with lower-dimensional input, our network requires fewer matrix multiplications and fewer parameters in the fully connected layers, greatly improving speed.

- State Space: Preprocessed screenshots of the browser loading the game

### 2.1.3 Actions

Coaster Racer Flash Game is a typical car racing game which is simply controlled by moving forward, left turn and right turn. With this simple setting, we only only an action space of three actions, namely, moving forward, left turn and right turn.

- Action Space: Move Forward action, Turn Left action, Turn Right action

### 2.1.4 Rewards

Rewards are primarily based off of the game score. From the perspective of human player, we judge a good player or not by how the driving behavior is close to real and safe case. While for a Game Bot, it only cares about the score. For this specific game, Coaster Racer Game, it unreasonably encourages a dangerous driving behavior. For example, when the car hits a traffic cone it gets 1000 reward, even if the price of hitting a cone is falling down the road. There is no punishment for falling down the road.

An ideal reward function would encourage the car to stay in the middle of the road and to try not to hitting other cars and guard bars. Here I simply cancel the one time 1000 score reward. But there is still room to fine tune the function in the future.

## 2.2 Algorithms and Techniques

### 2.2.1 Overal Representation

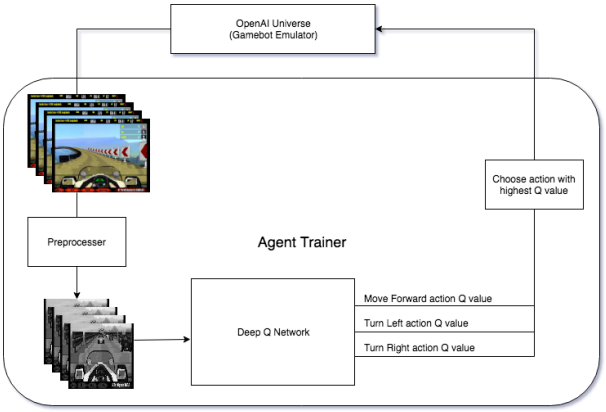



**Figure 3:** A flowchart of training the Coaster Racer Game Bot.

An overall representation of how the different components relate during a play evaluation, centered around the deep Q-network for playing, the main decision component is shown in Figure 3. Each game screen is resized to a desaturated pixels image, and if you might be wondering why each state is a sequence of four game screens instead of one, that is because the agent's history is used for better motion perception. Achieving this requires a sequence of preprocessed images to be stacked in channels (like you would stack RGB channels on a colored image) and fed to the network.

### 2.2.2 Deep Neural Network

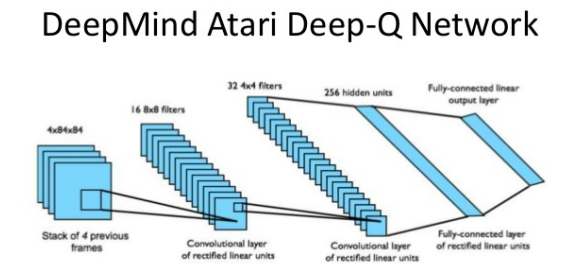
Convolutional Neural Networks, or CNNs, are a special type of neural network that has a known grid-like topology. Like most other neural networks they are trained with a variant of the back-propagation algorithm. CNN's strength is pattern recognition directly from pixels of images with minimal processing. We use a convolutional network as a function mapping the preprocessed images to Q values, since the actions are highly based on what would be seen as pixel matrix.

- **DeepMind's Deep Q Network**

  The network's architecture that I firstly used is essentially the same used by DeepMind, except for the first convolutional neural network's input (80x80x4 instead of 84x84x4, to account for the different input sizes) and the linear layer's output (3 instead of 18, to account for the different number of actions available).

- **LeNet-5.**

  For an image classification problem, there are several mature models other than the one above such as LeNet, VGG, Inception and Xception. I then tried a LeNet-5 model here and expected it to perform better than the DeepMind one for predicting actions. LeNet is deeper than the DeepMind




| Layer | Input | Filter size | Stride | Num filters | Activation | Output |
|---|---|---|---|---|---|---|
| conv1 | 84x84x4 | 8×8 | 4 | 32 | ReLU | 20x20x32 |
| conv2 | 20x20x32 | 4×4 | 2 | 64 | ReLU | 9x9x64 |
| conv3 | 9x9x64 | 3×3 | 1 | 64 | ReLU | 7x7x64 |
| fc4 | 7x7x64 | | | 512 | ReLU | 512 |
| fc5 | 512 | | | 18 | Linear | 18 |

**Figure 4:** Deep Neural Network model from DeepMind paper.

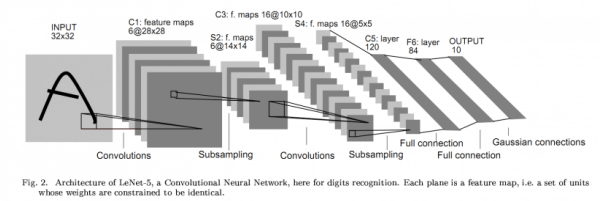



Fig. 2. Architecture of LeNet-5, a Convolutional Neural Network, here for digits recognition. Each plane is a feature map, i.e. a set of units whose weights are constrained to be identical.

**Figure 5:** LeNet-5.

one to grasp more features and has some dropout layers to avoid overfitting. In practice, it shows an advantage especially on predicting turning actions.

### 2.2.3 Q-learning

One of the most basic and popular methods to estimate action-value functions is the *Q-learning* algorithm. It is model-free online off-policy algorithm, whose main strength is that it is able to compare the expected utility of the available actions without requiring a model of the environment. Q-learning works by learning an action-value function that gives the expected utility of taking a given action in a given state and following a fixed policy thereafter.

A value function estimates what is good for an agent over the long run. It estimates the expected outcome from any given state, by summarizing the total amount of reward that an agent can expect to accumulate into a single number. Value functions are defined for particular policies.

The *state value function* (or V-function), is the expected return when starting in state $s$ and following policy $\pi$ thereafter (1),

$$V^{\pi}(s) = \mathbb{E}_{\pi}\left[R_t | s_t = s\right] \tag{2}$$

The *action value function* (or Q-function), is the expected return after selecting action $a$ in state $s$ and then following policy $\pi$,

$$Q^{\pi}(s,a) = \mathbb{E}_{\pi}\left[R_t | s_t = s, a_t = a\right] \tag{3}$$

The *optimal value function* is the unique value function that maximises the value of every state, or state-action pair,

$$Q^{*}(s,a) \quad = \quad \max_{\pi} Q^{\pi}(s,a), \forall s \in \mathcal{S}, a \in \mathcal{A} \tag{4}$$

An *optimal policy* $\pi^{*}(s,a)$ is a policy that maximises the action value function from every state in the MDP,

$$\pi^{*}(s,a) = \operatorname*{argmax}_{\pi} Q^{\pi}(s,a) \tag{5}$$

The update rule uses action-values and a built-in max-operator over the action-values of the next state in order to update $Q(s_t, a_t)$ as follows,

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha \left[ r_{t+1} + \gamma \max_a Q(s_{t+1}, a) - Q(s_t, a_t) \right] \tag{6}$$

The agent makes a step in the environment from state $s_t$ to $s_{t+1}$ using action $a_t$ while receiving reward $r_t$. The update takes place on the action-value $a_t$ in the state $s_t$ from which this action was executed. This version of Q-learning works well for tasks with a small a state-space, since it uses arrays or tables with one entry for each state-action pair.

In this project the policy is using the **$\epsilon$-greedy** policy:

- **$\epsilon$-greedy.** Selects the best action for a proportion $1-\epsilon$ of the trials, and another action is randomly selected (with uniform probability) for a proportion,

$$\pi_\epsilon(s) = \begin{cases} \pi_{\text{rand}}(s, a) & \text{if } rand() < \epsilon \\ \pi_{\text{greedy}}(s, a) & \text{otherwise} \end{cases} \tag{7}$$

  where $\epsilon \in [0, 1]$ and $rand()$ returns a random number from a uniform distribution $\in [0, 1]$.

### 2.3 Benchmark

This benchmark consists in playing against an agent that takes uniformly random moves. This is the most basic benchmark, but first we have to be sure that our learned evaluation function can play better than a random agent before moving into a harder benchmark. Also, this will help us to detect bugs in the code and algorithms: if a learned value function does not play significantly better than a random agent, is not learning. The idea is to test against this benchmark using Alpha-beta pruning at 1, 2 and 4-ply search.

## 3 Methodology

Q-learning, a model-free online off-policy algorithm, whose main strength is that it is able to compare the expected utility of the available actions without requiring a model of the environment:

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha [r_{t+1} + \gamma \max_a Q(s_{t+1}, a) - Q(s_t, a_t)] \tag{8}$$

### 3.1 Data Preprocessing

- Crop.


**Figure 6:** Cropped input image.

The raw input image is captured by Universe in the function "step()" which is simply a screenshot of the browser as shown in Figure 2. Obviously the useful vision information are only from the game screen. If we go further, the top half of the game screen displaying the sky does not change very much and the bottom half of the is heavily effected by the turning actions. Thus a region of interest is carefully chosen by whether considering whether it relates to decide an expected steering behavior. Then the image is simply cropped by a new window with the "cropFrame()" function. An example cropped image is as shown in Figure 6.

- Downscale the resolution.

**Figure 7:** Downsized input image.

A high resolution is usually redundant for a computer vision task. By resizing with a smaller size, the space information are almost remained and the computing time are greatly saved. The cropped image is then downsized to a smaller size, [80, 80] as in Figure 7. Downscaling the resolution doesn't hurt the information for turning left or right but highly accelerating the computing since much smaller data are being processed.

- Grayscale.

**Figure 8:** Grayscaled input image.

Grayscale processing is another useful technique in computer vision tasks since it is a great help for accelerating the computing and is at least three times faster than that of color image processing. This is because grayscale image has only one color channel as opposed to three in a color image. The color information are usually dumped when unnecessary for the computer vision tasks. As here, the downsized image is gray scaled as in Figure 8, because the color information does not help much for the steering control.

## 3.2 Implementation

### 3.2.1 Environment Setting

OpenAI Universe is designed with using Docker so it can reset the Game continuously. Docker is a tool that lets you run virtual machines on the local computer. I created an image wrapping up everything necessary, like TensorFlow, OpenCV, Gym and Universe.

To start up a very first Game Environment, only gym and universe modules are needed. Since we are to generate some random behaviors, "random" is also imported.

```
import gym
import universe # register the universe environments
import random
```

```
env = gym.make('flashgames.CoasterRacer-v0')  # You can run many environment in parallel
env.configure(remotes=1)  # automatically creates a local docker container
# env.configure(remotes='vnc://localhost:5900+15901')

# define our turns or keyboard actions
left = [('KeyEvent', 'ArrowUp', True),
          ('KeyEvent', 'ArrowLeft', True),
          ('KeyEvent', 'ArrowRight', False)]
right = [('KeyEvent', 'ArrowUp', True),
            ('KeyEvent', 'ArrowLeft', False),
            ('KeyEvent', 'ArrowRight', True)]
forward = [('KeyEvent', 'ArrowUp', True),
                ('KeyEvent', 'ArrowLeft', False),
                ('KeyEvent', 'ArrowRight', False)]

observation_n = env.reset()  # Initiate the environment
while True:
    action = random.choice([left, right, forward])
    action_n = [action for ob in observation_n]  # your agent here
    observation_n, reward_n, done_n, info = env.step(action_n)  # rl action by agent
    print("ACTION", action, "\t/ REWARD", reward_n)
    env.render()  # Run the agent on the environment
```

### 3.2.2 Build the Deep Neural Network

The Deep Neural Network is based on the DeepMind paper except for the changes of input and output sizes.

```
def createGraph():

    W_conv1 = tf.Variable(tf.zeros([8, 8, 4, 32]), name='W_conv1')
    b_conv1 = tf.Variable(tf.zeros([32]), name='b_conv1')

    W_conv2 = tf.Variable(tf.zeros([4, 4, 32, 64]), name='W_conv2')
    b_conv2 = tf.Variable(tf.zeros([64]), name='b_conv2')

    W_conv3 = tf.Variable(tf.zeros([3, 3, 64, 64]), name='W_conv3')
    b_conv3 = tf.Variable(tf.zeros([64]), name='b_conv3')

    W_fc4 = tf.Variable(tf.zeros([2304, 512]), name='W_fc4')
    b_fc4 = tf.Variable(tf.zeros([512]), name='b_fc4')

    W_fc5 = tf.Variable(tf.zeros([512, ACTIONS]), name='W_fc5')
    b_fc5 = tf.Variable(tf.zeros([ACTIONS]), name='b_fc5')

    # input for pixel data
    inp = tf.placeholder("float", [None, 80, 80, 4], name='input')

    # Computes rectified linear unit activation fucntion on  a 2-D convolution
    # given 4-D input and filter tensors. and
    conv1 = tf.nn.relu(tf.nn.conv2d(inp, W_conv1, strides=[1, 4, 4, 1], padding="VALID")
                                + b_conv1)
    conv2 = tf.nn.relu(tf.nn.conv2d(conv1, W_conv2, strides=[1, 2, 2, 1], padding="VALID")
```

```
                                + b_conv2)
    conv3 = tf.nn.relu(tf.nn.conv2d(conv2, W_conv3, strides=[1, 1, 1, 1], padding="VALID")
                                + b_conv3)

    # flatten conv3:
    conv3_flat = tf.reshape(conv3, [-1, 2304])

    fc4 = tf.nn.relu(tf.matmul(conv3_flat, W_fc4) + b_fc4)

    out = tf.matmul(fc4, W_fc5) + b_fc5

    return inp, out
```

### 3.2.3 Preprocessing Functions

As described in the previous chapters, the crop, downsample and grayscale operations are done with the following functions:

```
# crop video frame so NN is smaller and set range between 1 and 0; and
# stack-a-bitch!
def processFrame(observation_n):
    if observation_n is not None:
        obs = observation_n[0]['vision']
        # crop
        obs = cropFrame(obs)
        # downscale resolution
        obs = cv2.resize(obs, (80, 80))
        # greyscale
        obs = cv2.cvtColor(obs, cv2.COLOR_BGR2GRAY)
        # Convert to float
        obs = obs.astype(np.float32)
        # scale from 1 to 255
        obs *= (1.0 / 255.0)
        # re-shape a bitch
        obs = np.reshape(obs, [80, 80])
    return obs

# crop frame to only flash portion:


def cropFrame(obs):
    # adds top = 84 and left = 18 to height and width:
    return obs[284:564, 18:658, :]
```

### 3.2.4 Finetuning Hyperparameters

I played with some hyperparameters:

- The number of time steps
- The capacity of the replay memory,

- The layers of the network and the region of interest.

Currently this process is nothing more than trial and error. I could be better designed by grid search or random search. Well, it could be even easier to design the pipeline of implement a Deep Reinforcement Learning problem but tougher in hyperparameter tuning. Without a deeper understanding of the features of model, it has a high probability to fall into a less efficient experiment process.

## 3.3 Refinement

### 3.3.1 Relay Memory

Along the first several trials, I experimented with different memory capacities. The memory is a pool where the Game Bot would pick actions by Q values from. A small pool means it would be quick to converge during the training stage while it also limits the capacity it can learn. So there is a tradeoff here.

### 3.3.2 The Complexity of the Nerwork

The original works well for some simple game such as Atari and CartPole. The Coster Racer Game here is more complex to play because it scores by driving along the way with a high speed. The delay between an action and a reward is bigger. Through a long time training, it has a bad classification on Left Turn and Right Turn cases.

I replace the DeepMind model with LeNet-5 which then shows a better ability in classifying and quicker convergence.

# 4 Results

## 4.1 Model Evaluation and Validation


**Figure 9:** High score by the Game Bot.

Figure 9 is a screenshot when the car reach the finishing line and it got a score of 10269. The network was trained over 10,000 time steps and the behavior has been learned gradually. Compared to random bot, it has shown obvious better performance. For better or human level behavior, more training time steps are needed.

## 4.2 Justification

The reward histories shown in Figure 10 indicates how it performs better than a random Game Bot when the DQN one is trained 100000 time steps. It proves a much better ability to earn scores or rewards though there is still a big room to improve. The Game Bot can earn as high as 10269 scores though it still usually ranks the last.

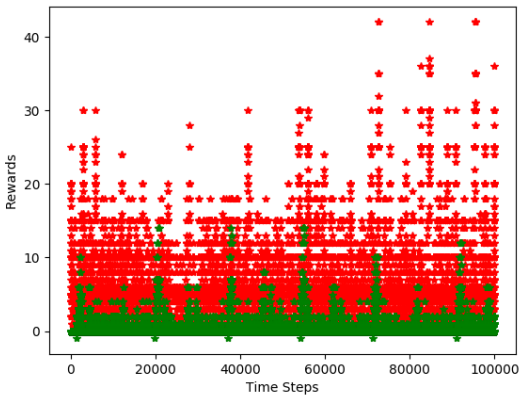



**Figure 10:** A comparison between DQN trained Game Bot and a random Game Bot.

# 5 Conclusion

## 5.1 Free-Form Visualization

A Youtube video has also been uploaded with the link https://youtu.be/VdVA3od4tVs here and it shows how it performs after 17000 time steps' training. It shows a capacity to stay in the road though it has some difficulty choosing right actions when blocked by the guard bars. The guard covered part of its view and the always acceleration actions made it even hard to get out of the stuck.

## 5.2 Reflection

### 5.2.1 Diffculties

The most difficult aspect of this project was that is extremely hard to stabilize reinforcement learning with non-linear function approximators. There are plenty of tricks that can be used and hyper-parameters that need to be tuned to get it to work, such as exploration policy, discount factor, learning rate, number of episodes, batch size, experience pool size and initial value.

All these techniques and parameters were selected by trial and error, and no systematic grid search was done due to the high computational cost. More than once it seemed that the implementation of the algorithms and techniques was incorrect, and it turned out that the wrong parameters were being used. A "simple" change such as decreasing $\epsilon$, or changing the neural network optimizer made big changes in the performance of the value function.

Also a huge difficulty of a reinforcement learning problem could be the time lag between the action and the reward. When training with grouped actions off of the heuristic reward function, the reward for a given action was immediate, and this network showed the best performance. The next best performance came from a network based off of grouped actions, where actions were only a few steps removed from the next reward. Our worst performance came from the networks trained to estimate actions, where actions were several dozen steps removed from the next rewards.

### 5.2.2 General Pipeline

In a Deep Q Network setting, there are several elements which we have to be careful to define.

- **Environment:** An environment defines what the agent interacts with. It receives states and actions and generate new states and plays a role as an online data generator.
- **State Space:** A state is the input of the Deep Q Network. In this project, the stats is an image or a pixel array. It will be trained by a Deep Neural Network and predict the next actions.

- **Action Space:** An action space could be discrete and continuous. It defines the all classes that would be generated from the Deep Q Network. A bigger action space indicates a bigger room for an agent to learn and improve but also means a much complexity to train.
- **Reward Functions:** The reward function determines in which way we would like the agent to grow. For example, we would define a bigger reward for the car to stay in the middle of the road than in the side of the road. It can be a discrete or a continuous function.
- **Deep Neural Network:** The Deep Neural Network is responsible to map the states to the Q values, which are corresponding to different actions by Q functions.
- **Fine tune the Hyperparameters:** By fine tuning the hyperparameters, we try to maximize the ability of the defined Deep Q Network. It can be subtle to modify the hyperparameter values which might change the output in different ways, like effecting the time of the convergence, the prediction accuracy, overfitting or underfitting and robustness.

### 5.3 Improvement

There are at least two aspects we can improve in the next stage,

1. **More Detailed Action Space**: Here only three actions are defined and the car is always in acceleration mode which is not the best choice in some cases. When the car is got stuck, acceleration wastes the effort of getting out. And it is easier to pass the sharp turn if it slows down the speed a little bit. Thus we can define more actions as follows.

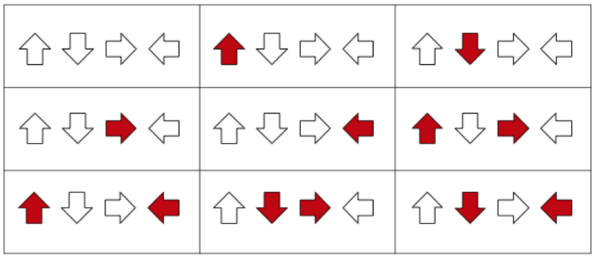


**Figure 11:** A bigger action space.

2. **Tuned Reward Functions:** In this project, I almost relied on the game score as the reward resource. In fact, there are some rules to get maximal rewards which I don't think help learning a reasonable driving behavior. For example, there is no punishment on being blocked and falling into the sky. The Game Bot may think it is no big deal to fall into the sky and sometimes it can be rewarded 1000 score if it happen to hit a traffic cone before it falls down. I put my focus on building a working Deep Q Learning structure and had a hard time hacking into the scoring system, so I have to put this task next time.
3. **Better benchmarks:** In most of this project we used simple benchmarks, such as playing against random players. While testing against random is probably the first thing to test against (if you can't beat a random player your learning algorithm is not working), it would be better to find a few heuristics and better players that can be used for testing.
4. **Incorporate other RL techniques:** The field of RL has been advancing fast in recent years. There are a few new and old techniques that I would like to try, such as asynchronous RL, double Q-learning, prioritized experience replay and Asynchronous Actor-Critic Agents (A3C).

## References


[1] R. S. Sutton and A. G. Barto. *Reinforcement Learning: An Introduction*. MIT Press, 1998.

[2] V. Mni et al. Human-level control through deep reinforcement learning. *Nature*, pages 529–533, 2015.