Introduction to Reinforcement Learning Algorithm

Meili Liu

1. Introduction

Reinforcement learning (RL) is one of the hottest research topics in the area of Artificial Intelligence, which has achieved great performance in various problems, including robotic manipulation, Go(AlphaGo), playing games, and autonomous driving.

In this report, we are going to introduce two popular deep reinforcement learning algorithms: Deep Q-Network (DQN) and Policy Gradient (PG). DQN is a value-based approach that first tries to compute optimal state-action value, then extract optimal policy from it, While the latter is a policy-based approach, which can directly optimize the policy. All these two algorithms can learn successful policies from high-dimensional inputs, such as pixels of images, by using end to end reinforcement learning.

2. Reinforcement learning Algorithm

Reinforcement learning (RL) is a general framework where agents learn to take action in an environment so as to achieve a maximal reward. The agent continuously interacts with the environment: At each time step, the agent observes the environment, then takes actions a_t based on the observation s_t and its policy $\pi(a_t|s_t)$. After that, the agent receives a reward $R(s_t, a_t)$ from the environment and enters into a next observation s_{t+1}. The goal for the agent is to improve the policy so as to maximize the expected total rewards.

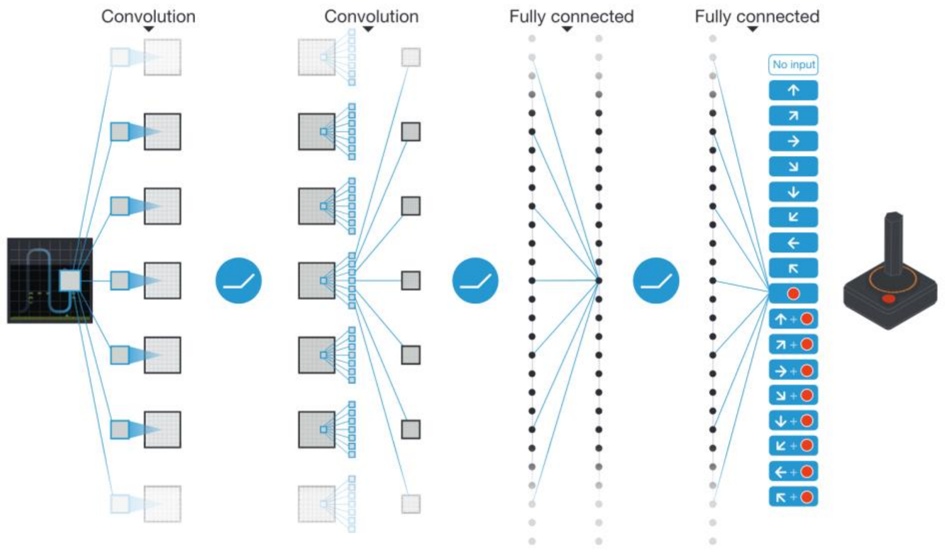


Figure1: Illustration of the Deep Q-network: the input to the neural network consists of an 84*84*4 preprocessed image, followed by three convolutional layers and two fully connected layers.

2.1 Deep Q-Network (DQN)

DQN was introduced by DeepMind team in 2015[1], which has been demonstrated to be able to solve Atari games (some to superhuman level). An Illustration of the DQN architecture is shown in Figure 1. The input of the network is a pixel image from Atari games, which can be seen as the state representation, and the output is a vector of Q-values for each action in terms of the input state. The best action is the one that resulted in the biggest Q-value.

The exact architecture is as follows:

1) Input: consists of an 84*84*4 image.
2) first convolutional layer: 32 filters of size 8*8 with stride 4, followed by a rectifier nonlinearity (RELU)
3) second convolutional layer: 64 filters of size 4*4 with stride 2, followed by a RELU
4) third convolutional layer: 64 filters of size 3*3 with stride 1, followed by a RELU
5) hidden fully connected layer: 512 rectifier units
6) output layer: fully connected linear layer with a single output for each valid action, between 4 and 18 on different games

2.1.1 training algorithm

DQN is the value-based approach, where the state value state is the cumulative future reward. The goal of the DQN is to use a deep convolutional neural network to approximate the optimal action-value function $Q(s, a)$. The algorithm is presented in Algorithm 1. Essentially, the Q-network is learned by minimizing the following mean squared error:

$$J(w) = \mathrm{E}[y_t - \hat{q}(s_t, a_t, w)^2]$$

Where y_t is the target value, generated from target network with parameters w^-

$$y_t = r_t + \gamma max_{a'} \hat{q}(s_{t+1}, a', w^-)$$

and $\hat{q}$ is the predicted value, generated from the value network with parameters w

2.1.2 separated target network and value network

The separated target network and value network can improve the stability of learning and deal with the non-stationary learning task. To be more specific, during training, the parameter of the target network would keep fixed, except for every C step, updating by copying the parameters' value $w^{-} = w$ from the value network. This trick can make the algorithm more stable and easier to converge bu reducing correlations between the target network and the value network.

2.1.3 Experience replay

To further improve the stability of DQN on the Atari game, a replay buffer is implemented to store previous transitions. Then during training, instead of using just the latest transition to compute the loss and its gradient, we compute them using a minibatch of transitions sampled from the replay buffer. This can increase data efficiency by reusing them and results in better stability using uncorrelated transitions in a batch.

2.1.4 ϵ −greedy policy

In order to achieve a balance between exploration and exploitation, we use a naïve but effective strategy, called ϵ −greedy policy. The strategy is to take a random action with a small probability and take the greedy action the rest of the time. At the beginning of training, ϵ is set as 1, which means only choosing random action. During training, the value of ϵ keeps decreasing with the episode, until ending up with 0.1 at the final exploration steps.

Mathematically, a ϵ −greedy policies with respect to the state-action value takes the following form:

$$a = \begin{cases} random\ action & with\ probability\ \epsilon \\ argmax_a Q(s,a) & with\ probability\ 1 - \epsilon \end{cases}$$

2.1.5 Algorithm for Deep Q-learning

```
Algorithm 1 deep Q-learning
1. Initialize replay memory $D$ with a fixed capacity
2. Initialize action-value function function $\hat{q}$ with random weights $w$
3. Initialize target action-value function $\hat{q}$ with weights $w^{-} = w$
4. for episode m=1,…,M do
5.      Observe initial frame $x_1$ and preprocess frame to get state $s_1$
6.      for time step t=1,….T do
7.           Select action $a_t = \begin{cases} random\ action & with\ probability\ \epsilon \\ argmax_a \hat{q}(s_t, a, w) & otherwise \end{cases}$
8.           Execute action $a_t$ in emulator and observe reward $r_t$ and image $x_{t+1}$
9.           Preprocess $s_t$, $x_{t+1}$, to get $s_{t+1}$ and store transition $(s_t, a_t, r_t, s_{t+1})$ from $D$
10.          Sample uniformly a random minibatch of N transition$(s_j, a_j, r_j, s_{j+1})$
11.          Set $y_j = r_j$ if episode ends at step j+1;
12.          otherwise set $y_j = r_j + \gamma max_{a'} \hat{q}(s_{j+1}, a', w)$
```

```
13.    Perform a stochastic gradient descent step on $J(w) = \frac{1}{N}\sum_{j=1}^{N} y_j - \hat{q}(s_j, a_j, w)^2$ $w.r.t.$ parameter $w$
14.    Every C step reset $w^- = w$
```

2.2 Policy gradient

Not like the value-based approach, where we find an optimal state value function $Q(s, a)$, and then extract a policy, in a policy-based approach, we can directly parameterize the policy. Furthermore, a policy-based RL has a few advantages over value-based RL:

1) handle high-dimensional or continuous action spaces, e.g., robotics.
2) Learn stochastic policies

2.2.1 The training algorithm

1) Initialize the agent
2) Run a policy until termination
3) Decrease the probability of actions that resulted in low reward
4) Increase the probability of actions that resulted in high reward

2.2.2 loss function and gradient descent update

$$loss = -\log[p(a_t|s_t)]\, R_t$$

$$w' = w + \nabla \log[p(a_t|s_t)]\, R_t$$

2.3 Comparison between DQN and Policy Gradient method

Advantages of policy gradient compared with DQN:

1) Better converge properties: DQN tend to be very unstable, thus hard to converge, while policy gradient method can converge more quickly. In my experiment, training DQN cost almost 24 hours, while for policy gradient, only 4 hours.
2) Ability to learn stochastic policies in high-dimensional or continuous action space, e.g., robot, self-driving cars

3. Experiment 1: Playing Atari game – Pong

Pong is a classic Atari game, including two paddles competing. The right one is controlled by our agent, which can move up and down to hit the ball so that the opponent, controlled by a computer, is not able to reach it. The game is over once one player achieves a score of 21 points.

3.1 Preprocessing

In this report, we firstly implement a deep Q-network based on Atari game Pong. The Atari environment from OpenAI gym returns observations of size (210*160*3), which is height, weight and RGB channels, respectively, shown in figure 2. To reduce the computation and memory requirement, we will apply some preprocessing to the observations:

1) Encode a single frame: take the maximum value for each pixel color value over the frame being encoded and the previous frame.
2) Reduce dimensionality: convert the encoded image to greyscale, and rescale it to (84*84*1)

These preprocessing methods are applied to the 4 most recent observations and then stack them as the input (of shape (80*80*4)) to the DQN. Also, Follow the DeepMind paper [1], for each time we decide on an action, we perform that action for 4 steps. This can allow the agent to play more games without significantly increasing the runtime.

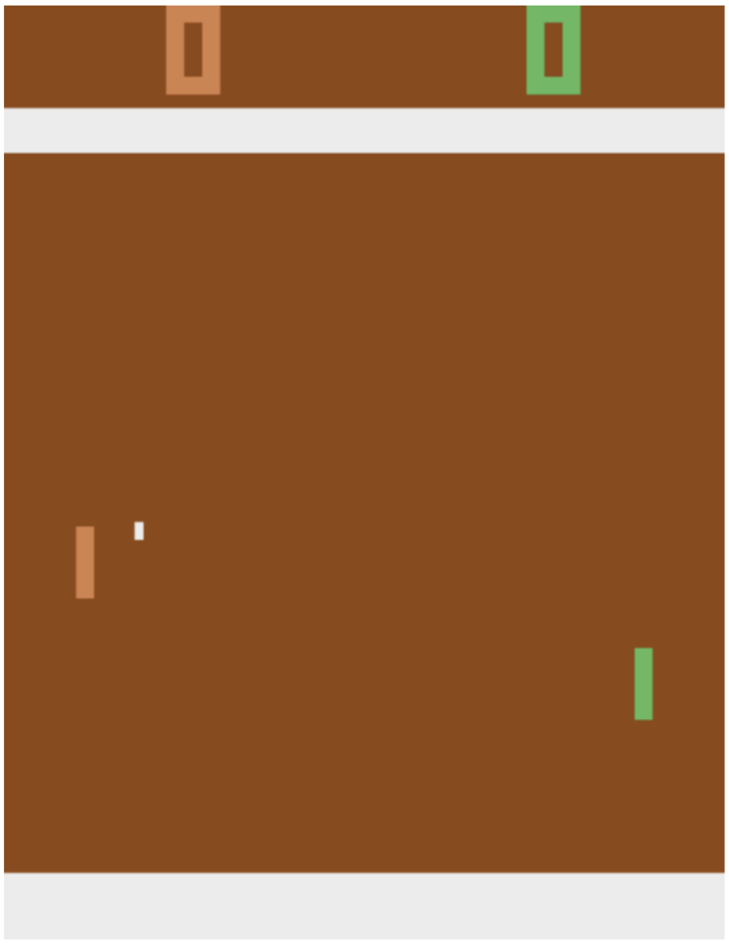


Figure2 Environment of Atari game Pong

3.2 Model Architecture and Hyperparameters

we experiment with the following hyperparameters:

Minibatch size	32
Replay memory size	1000000
Agent history length	4
Target network update frequency	10000
Discount factor	0.99
Action repeat	4
Update frequency	4
Optimizer	Adam
Learning rate	0.0001
Initial exploration	1
Final exploration	0.1
Final exploration frame	1000000
Replay start size	50000

Table 2. hyperparameters for the experiment on DQN

3.3 Results

At the beginning of training, the DQN agent performs only random actions, and thus get a reward of around -20, which means that it loses hopelessly. After a few hours, it starts to learn how to hit the ball and gain a few points. Figure 3.1 below shows the changes of loss during training. Although there is a considerable increase at the beginning, the value ends up declining to the minimum value. This trend is also shown in the evaluation score in Figure 3.3, after some oscillation, the score finally reaches the maximum value of +21

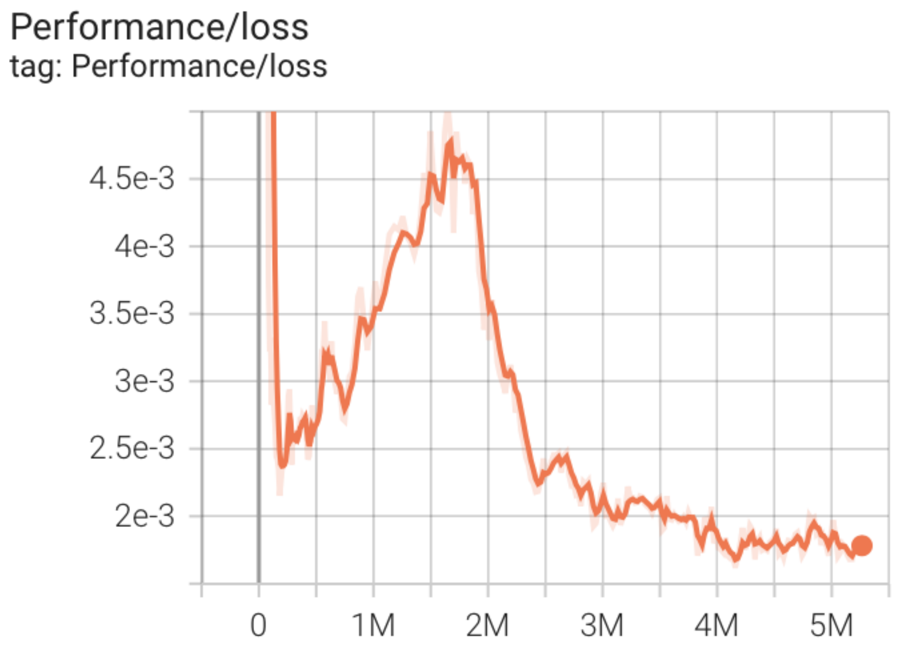


Figure 3.1 Loss curve during training

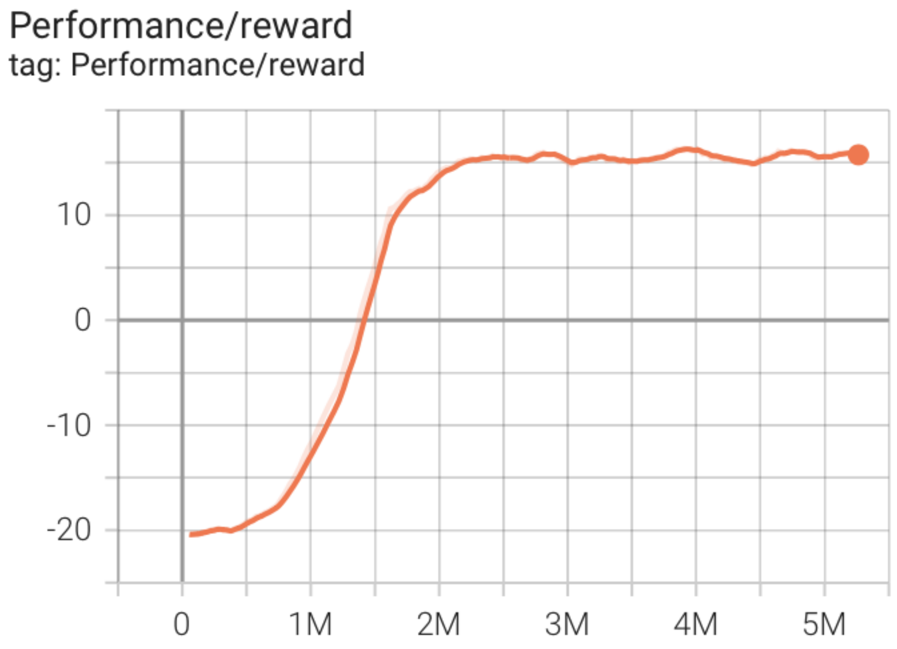


Figure 3.2 reward curve during training

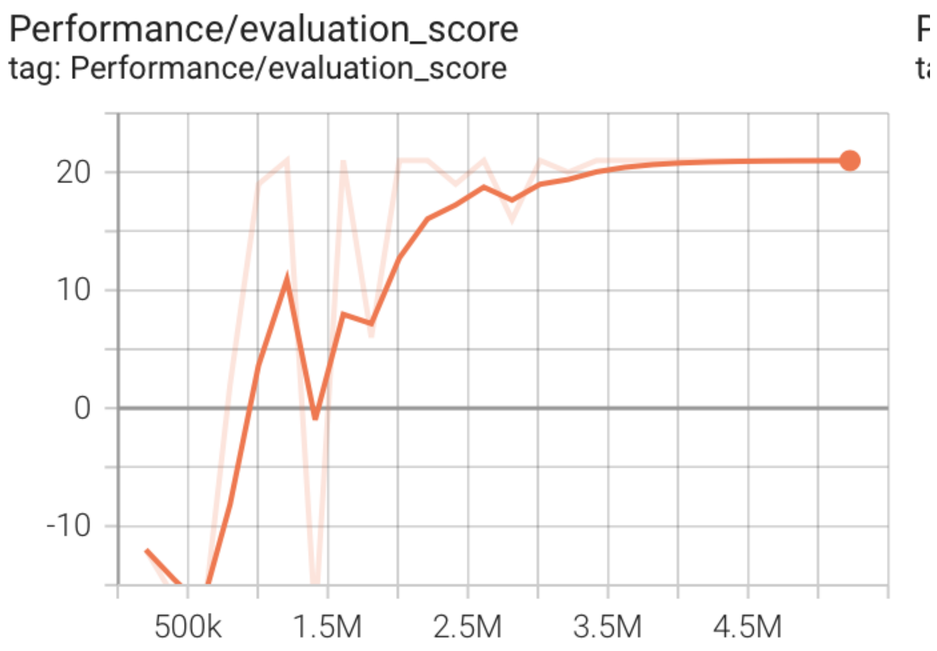


Figure 3.3 evaluation score curve during evaluation

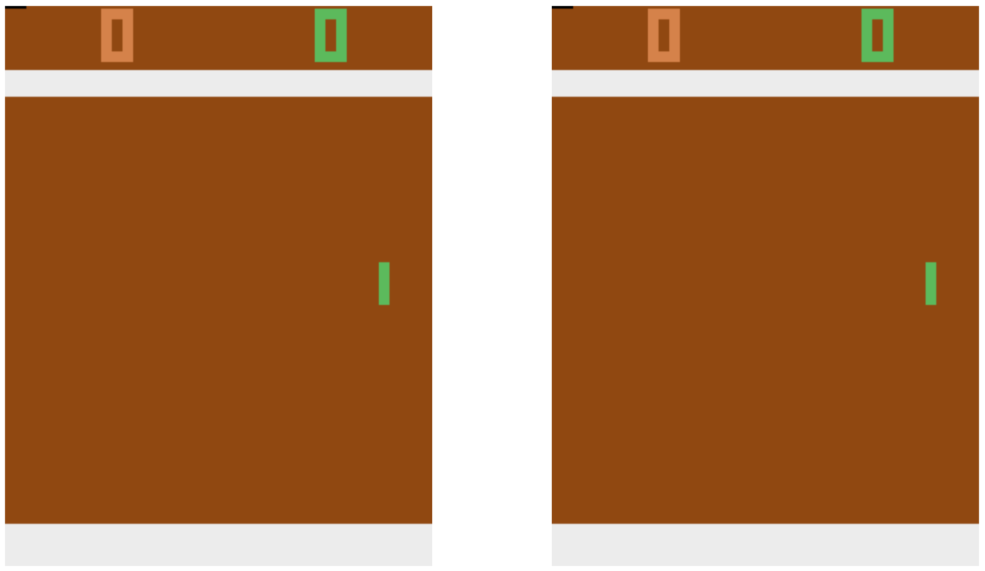
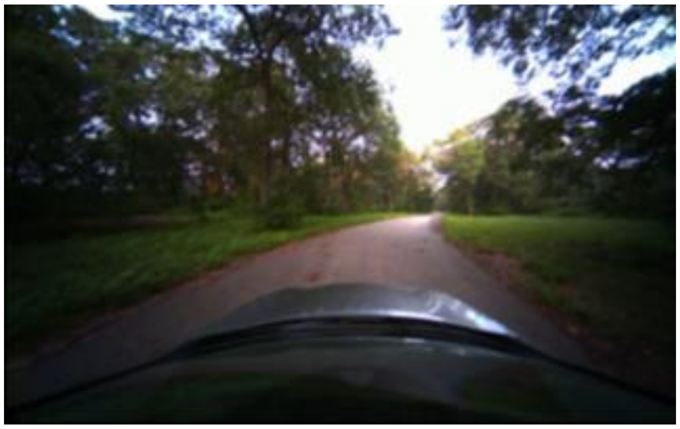


Figure 4: play video. Left: at the beginning of training. Right: after training

4. Experiment 2: Training Autonomous Driving in VISTA

In experiment 2, we train an autonomous driving agent on the simulation engine VISTA, which uses techniques in computer vision to synthesize new photorealistic trajectories and driving viewpoints. We can define an agent car moving around in the environment, and directly collect data for training and testing self-driving control policies. The simulated environment is shown on Figure 3.1.



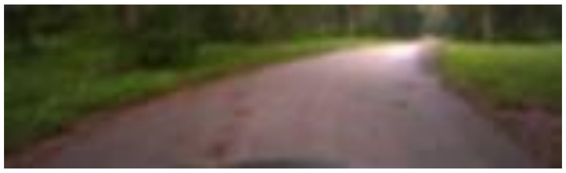


Figure 3.1, the full observation of VISTA simulator.   Figure 3.2 observation after preprocessing

4.1 preprocessing

Like training on the Atari game, we need to do some preprocessing on the full observation image like Figure 3.1, to make it easier for the network to learn and reduce the computation burden.

1) ROI: crop a smaller region of interest to discard the distorted edge area and make it easier for the model to learn from.

Figure 3.2 shows the image after preprocessing

4.2 action space

In the case of driving, the car agent can take actions, consisting of speed and curvature, to move around in the VISTA environment. To make the training easier, we keep the speed fixed, only optimizing policy control on curvature, which reflects the car's turn radius. Curvature has units 1/meter. So, if a car is traversing a circle of radius r meters, then it is turning with curvature $1/r$. Therefore, the curvature is correlated with the car's steering wheel angle.

4.3 reward function

In our example, the reward function is quite simple: give a reward of 1 if the car did not crash, and a reward of 0 if it did crash

4.4 model architecture and optimizer

Since a convolutional network would be well suited to the task of end-to-end control learning from RGB images, we adopt a model with four convolutional layers and two fully-connected layer with details below:

1) Input: consist of a 45*155*3 image.

2) first convolutional layer: 32 filters of size 5*5 with stride 2, followed by a rectifier nonlinearity (RELU)
3) second convolutional layer: 48 filters of size 5*5 with stride 2, followed by a RELU
4) third convolutional layer: 64 filters of size 3*3 with stride 2, followed by a RELU
5) forth convolutional layer: 64 filters of size 3*3 with stride 2, followed by a RELU
6) hidden fully-connected layer: 128 rectifier units
7) output layer: fully connected linear layer with 2 units

we use the Adam optimizer with a learning rate of 1e-3

4.5 Result

In Figure 4 we report the rewards of our agent car accepted on each iteration. After 200 iterations, the traversing distance of the agent car can reach around 1000 steps.

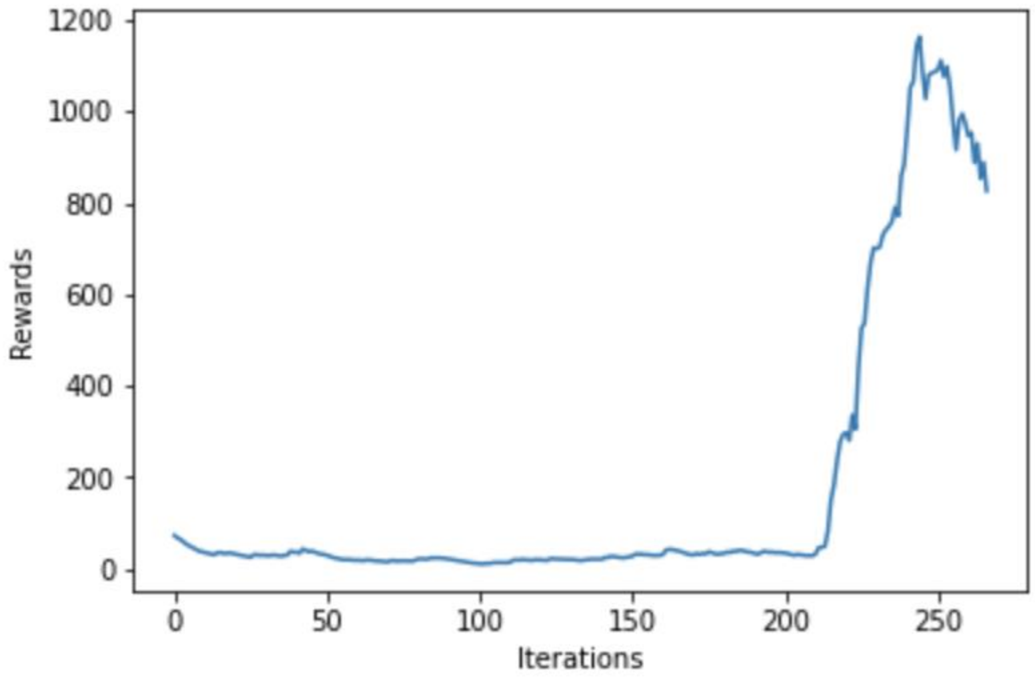


Figure 4. Rewards of each iteration

5. conclusion

In this report, we have implemented two different deep reinforcement learning algorithms, deep Q-network and policy gradient. Based on these, we have experimented a DQN to play an Atari game and a policy gradient to achieve self-driving.

6. reference

[1] Mnih et al., “Human-Level Control through Deep Reinforcement Learning.”
[2] http://www.scholarpedia.org/article/Policy_gradient_methods