Experiment Report

DQN Project Report

Author:
Liu Zhihan

June 27, 2020

Contents

1 Introduction

In this experiment, we will bulid projects based on Deep Learning Algorithms(2013, Mnih et al [1]) to solve two task: LunarLander and RiverRaid. Our task goal is to build agent from sratch and obtain satisfactory score in both tasks.

In the theory background part, basic theory and some implementary techiques will be briefly introduced. And all the details of our experiment will be listed in the Experiment part, including the preprocessing method, the network architecture, the definition of class and function and the basic program logic. As for experiment results, they will be listed in the Experiment part with further explanation and discussion.

2 Theory Background

In this section, some basic theory background notion will be introduced which our project is based on.

2.1 Convolution Nerual Network

For Deep learning, the basic method for updating the parameters of each layer is back-propagation. In this experiment, we are required to implement convolution neural network manully. To achieve this goal, we must analyse the basic structure of Convolution Nerual Network and the mechanism of forward and backward.

Traditional CNN consists of three parts: convolutional layers, maxpooling layers and fully connected layers. For forward process, the input, often images, will be convoluted with some filters with certain stride and new images, regularly called feature map is derived through the convolution. Then the feature map will be the input of next layer after activiation. As for the fully connected layer, it will multiply its weights martix with the input array and add a bias array.We can describe the process in formula and L means the L^{st} layer, Z means the output before activation, X means the input.

$Z^{[L]} = W^{[L]}X + b, \quad A^{[L]} = Activator(A^{[L]})$

However, the key problem is how to backward the CNN. According to backpropagation method, we should calculate the gradients of the activated output of each layer. For fully connected dA is an arrary with one dimension, that is, $dA = (\frac{\partial E}{\partial A_1}, ..., \frac{\partial E}{\partial A_n})$. And

we have the following equations:(here (.*)means element wise multiply)

$$\begin{aligned} dW^{[L]} &= (dZ^{[L]})^T \cdot A^{[L-1]} \\ db^{[L]} &= dZ^{[L]} \\ dA^{[L]} &= (W^{[L+1]})^T \cdot dZ^{[n+1]} \\ dZ^{[L]} &= dA^{[L]}(.*)(Activator^{[L]})'(Z^{[L]}) \end{aligned} \tag{1}$$

For convolutional layer, we have the conclusions in parallel:($\overline{X}$ means the zero padding of X; $W^{\#}$ means rotation of the W over 180 degree)

$$\begin{aligned} (dW^{[L]})_{i,j} &= \sum_{m,n} (dZ^{[L]})_{m,n} \cdot (dA^{[L-1]})_{i+m,j+n} \\ db^{[L]} &= \sum_{i,j} (dZ^{[L]})_{i,j} \\ dA^{[L]} &= \overline{dZ^{[n+1]}} \odot (W^{[L+1]})^{\#} \\ dZ^{[L]} &= dA^{[L]}(.*)(Activator^{[L]})'(Z^{[L]}) \end{aligned} \tag{2}$$

By the reccurent relation described in equations (1) and equations (2), we can implement the backward process.

2.2 Deep Q Network

DQN algorithms are the basic algorithms this project adopts and the presudecode is listed here Algorithm 1

In [1], they performed an experience replay and store agent's experiences $e_t = (s_t, a_t, r_t, s_{t+1})$ at each time-step t in a data set $\mathcal{D}_t = \{e_1, e_2, \ldots, e_t\}$. Then by randomly sampling training samples in $\mathcal{D}$: $(s, a, r, s') \sim U(\mathcal{D})$, a Q-Network can be trained by minimizing a sequence of loss functions

$$L_i(\theta_i) = \mathbb{E}_{(s,a,r,s')\sim U(\mathcal{D})}\left[\left(r + \gamma \max_{a'} Q(s', a'; \theta_{i-1}) - Q(s, a; \theta_i)\right)^2\right] \tag{3}$$

And then we can update the parameters by gradient decent method:

$$\nabla_{\theta_i} L_i(\theta_i) = \mathbb{E}_{(s,a,r,s')\sim U(\mathcal{D})}\left[\left(r + \gamma \max_{a'} Q(s', a'; \theta_{i-1}) - Q(s, a; \theta_i)\right) \nabla_{\theta_i} Q(s, a; \theta_i)\right] \tag{4}$$

For the stablility of the training, it is beneficial to introduce a target network outputing the target action valu$Q(s, a; \theta^-)$. The Q-learning update at iteration i uses the following loss function

$$L_i(\theta_i) = \mathbb{E}_{(s,a,r,s')\sim U(\mathcal{D})}\left[\left(r + \gamma \max_{a'} Q\left(s', a'; \theta_i^-\right) - Q(s, a; \theta_i)\right)^2\right] \tag{5}$$

Algorithm 1: Deep Q-Learning Algorithm with Soft Update Strategy

Initialize replay memory $\mathcal{D}$ to capacity N;
Initialize action-value function Q with random weights θ;
Initialize target action-value function Q^- with random weights θ^-;
Initialize $\epsilon, \tau \in (0, 1)$;
for *episode=1,2,...,M* **do**
  for $t = 1, 2, \ldots, T$ **do**
    With probability ϵ select a random action a_t;
    Otherwise select $a_t = \max_a Q(s_t, a; \theta)$;
    Execute action a_t in emulator and observe reward r_t and state s_{t+1};
    $\mathcal{D} \leftarrow \mathcal{D} \cup (s_t, a_t, r_t, s_{t+1})$;
    Sample random minibatch of transitions (s_j, a_j, r_j, s_{j+1}) from $\mathcal{D}$;
    Set $y_j = \begin{cases} r_j & \text{for terminal } s_{j+1} \\ r_j + \gamma \max_a Q^-(s_{t+1}, a; \theta^-) & \text{for non-terminal } s_{j+1} \end{cases}$;
    Perform a gradient descent step on $L(\theta) = (y_j - Q(s, a; \theta))^2$;
    $\theta^- \leftarrow \tau\theta + (1 - \tau)\theta^-$;
  end
end

where θ_i^- are the parameters of the target network at i^{th} iteration. And we update θ_i^- equal to θ_i every C steps.

2.3 Soft Update Strategy

Instead of copying the parameters of main network to target network every C steps, Timothy et al proposed a soft update strategy which further alleviate the vibration of the model. The update strategy takes the form of

$$\begin{aligned} \theta_i &\leftarrow \theta_{i-1} - \alpha \nabla_{\theta_i} L_i(\theta_i) \\ \theta_i^- &\leftarrow \tau\theta_i + (1 - \tau)\theta_{i-1}^- \end{aligned} \tag{6}$$

And our experiment further shows that the model is stabler and easier to train by applying to soft update strategy.

3 Experiment

In this section, we will introduce the details of our experiment as well as the model implementation.

3.1 Hand-made Convolutional Neural Network

In this project, we bulid deep Q-learning network from scratch. Considering the difficulties of building autograd tool which requires dynamic calculating maps, we choose to write the backward propagation process by analyzing the the form of graidents forehead. Based on the theoritiacal analysis in the Theory background section, we can establish the recurrent relation with the gradients and finally implement the back propagation and updating of each layer. While the only tool we are dependent on is numpy so the calculating process is time-consuming. For the limitation of time, we use the cuda version of the algorithms requring torch package for emprical experiment whose result should be in accordance with our hand-made network version.

We save our hand-made deep Q-learning network on the file *model.py*, and model calls the class *FClayer* and *cnn*, which respectively define our hand-made fully-connected layer and convolution layer. The whole hand-made network nearly implements the major functions of network inherited the class *torch.nn.Module*, including:

- Initial the model and set the fully-connected layer and convolutional layer (only used in the second task) .
- Forward the network and get the final output through fully-connexting and convolution.
- Backward the network. Calculate the gradients of each parameters of each layer by the given loss based on the back propagation method.
- Update all the layer of the network by the gradients calculated before
- Record all the parameters of each layer so that the network can be saved, loaded and updated.

3.2 LunarLander

3.2.1 Introduction to the task and our ideas

For the LunerLander task, its goal is simple: to successfully land, and the action space and observation space are both one dimensional array, which means no complex convolution structure in nerual network is needed because the convolution part is used to extract high dimensional feature and has quantities of parameters to be trained. So we choose a simpler network to ensure efficiency. Noting another feature of this task is the lack of time restriction, to prevent a situation where the agent gets stuck in flying up and down, we introduce a time limitation T_{max} inside the episode loop so that the agent can avoid such delimma.

3.2.2 Project Description

Our project folder consists of three major parts, including *train, dqn_agent and model* and they will be introduced here in details:

1.*train* :Run the file *train* then the training will start. This file contains four functions which respectively are used for initializing environment, DQN training , plot the score and save the result and finally display the training results on the screen.

- DQN function is the core function that calls the class *Agent* defined in the file *dqn_agent*, there is a loop where the agent does actions and steppings by calling the Agent class and construct the environment interations by importing gym functions like *reset* and *step*. And the loop is ended when the episode the agent has experienced exceeds the max episode we defined. In this loop, the agent interactes with environment continuos and the scores for each episode are recorded. For the all process is abstract , not involving the details of DQN algorithms thus making it easy for migration. For convenience to analyse , we import logger here to record the specific training process.

2. *dqn_agent*: In the file *dqn_agent* ,the class *Agent* and the class *Buffer_Replay* are defined which are the main body of the algorithm. The properties and methods of class Agent are listed in the following

- the class has the properties include the input channels (for LunarLander the input it is the state size), action size, seed, two DQN network calling the DQN network class defined in *model.py*, optimizer , the frames and the buffer replay calling the class defined in this file
- method *step*: receieve the experience and add it in the buffer replay. If the frames satisfy the learning condition (here is 4) than call the method *learn* of this class
- method *act* :receive observation and apply ϵ-greedy algorithm to output an action. If the random number is less than ϵ, then the action values which the local DQN-network outputs will be calculated the maximum to choose the best action (See Q learning)
- method *learn*: In this method, the value parameter θ will be updated using given batch of experience tuples. The main body of the process is derived by the SGD formula in the Theory Background section. And it is worth noting that the loss here calls the method *huber_loss* and the target network parameters updating calls the method *soft_update*.
- method *huber_loss*: a method of defining loss, which is beneficial when dealing with discrete data.

- method *soft_update*: Derived from Theory Background section, updating parameters in this way help smooth the update, increasing the stablility.

And the *Buffer_Replay* class is defined here

- Properities: action size, buffer size (maximum of the buffer) ,batch size , seed and memory. Here memory is a deque whose max length is the buffer size.
- method *add*: add experience into buffer replay
- method *sample* :sample experience (a batch) from buffer replay
- method *len* :return length of buffer replay

3. *model* : This part includes *model.py*, *FClayer.py* and *cnn.py*. The network model used is on the *model.py*, where stores the architecture of the nerual network. The detailed description is on the former section.

3.2.3 Q-Network for LunarLander

For convenience and sample efficiency, our network disposes convolution and maxpooling layer, applying only three fully connected layer only, which is a simple edition of the network applied in the RiverRaid task.

3.2.4 Other Details

1. As for the reward, we clipped all positive rewards at 1 and all negative rewards at -1, leaving 0 rewards unchanged as Mnih et al suggested.

2. Set the maximum frames per episode as stated in the desription.

3.ϵ decay we apply is simple form of index-decay for the training process is short (compared to the second game)

3.3 Riverraid

3.3.1 Introduction to the task and our ideas

Compared with the former task, Riverraid is a more complicated and hard problem, not only for its obseravtion space is two -dimension colorful picture but the action agent take will significantly effect the environment especially under the circumstances that the obseravation varies in one episode, not like the first game.

To solve the task with 2D image input, we must do first is to preprocess the input, through gray tuning and resizeing, the orginal input will be transformed into 84x84 tensor which enables the cuda to compute. After preprocessing the input, we still need to initial the buffer, that is, to store some experience ahead, which is an important implemention facing with complicated environment. As for the network, we must apply convolution layer here to help the agent extract information in the input. Another point is dealing with tensor, for the observation space is 2D, and we must do some transformation so that the computation by cuda can be as smoothy as in the former task.

3.3.2 Project Description

The project of Riverraid is mostly copied from the projects of former task while some adaptations are made:

1. Preprocess: The preprocess part is in the file *atari_wrappers.py*.
2. Other details will be listed in the Other Detail section.

3.3.3 Preprocessing for RiverRaid

Unlike the the environment in LunarLander, the raw observation in RiverRaid is a way more complex 210×160 RGB image with a 128-colour palette. Taking such image as the input of CNN is computational complicated. Thus we resized it to 84×84 and gray-scaled it. To extract time information, we further stacked 4 most recent frames as the $84 \times 84 \times 4$ input of Q-Network.

3.3.4 Q-Network for RiverRaid

Because of the complication of RiverRaid's observation, we choose Convolution Neural Network(CNN) with 3 convolution layers and a fully-connected hidden layer to better deal with image instances, the main architecture is described in Figure 1.

The input to the network is a $84 \times 84 \times 4$ tensor containing a rescaled and gray-scale version of the last four frames. The first convolution layer convolves the input with 32 filters of size 8 (stride 4), the second layer has 64 layers of size 4 (stride 2), the final convolution layer has 64 filters of size 3 (stride 1). This is followed by a Fully-Connected hidden layer of 512 units. All these layers are separated by ReLU. Finally, a fully-connected linear layer projects to the Q-values.

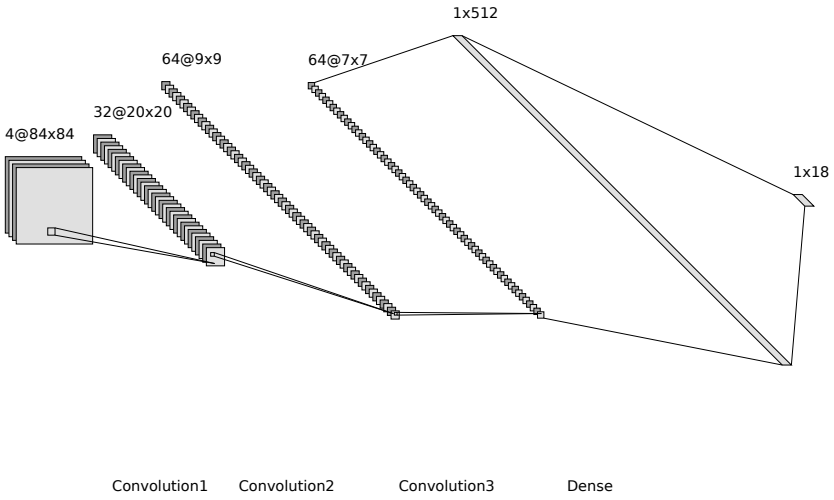


Figure 1: The CNN architecture in the Q-Network of RiverRaid

3.3.5 Other Details

1. As for the scores, due to the rule of the game, one shooter has 4 lives per episode, the score calculations are different from the previous task.

2. For the convenience of training and tuning hyper parameter, we choose to load parameter dictionary before training.

3.Due to the comlexity of the game, we choose a different ϵ decay form ,that is $\epsilon = \epsilon_f + (\epsilon_s - \epsilon_f) \times e^{-\frac{frams}{epsilon\_decay}}$ The hyperparameters can be refered in the Appendix A

3.4 Hyperparameter Selection

We mainly follow the setting as Mnih et al did. The complete hyperparameter settings are shown in Appendix A.

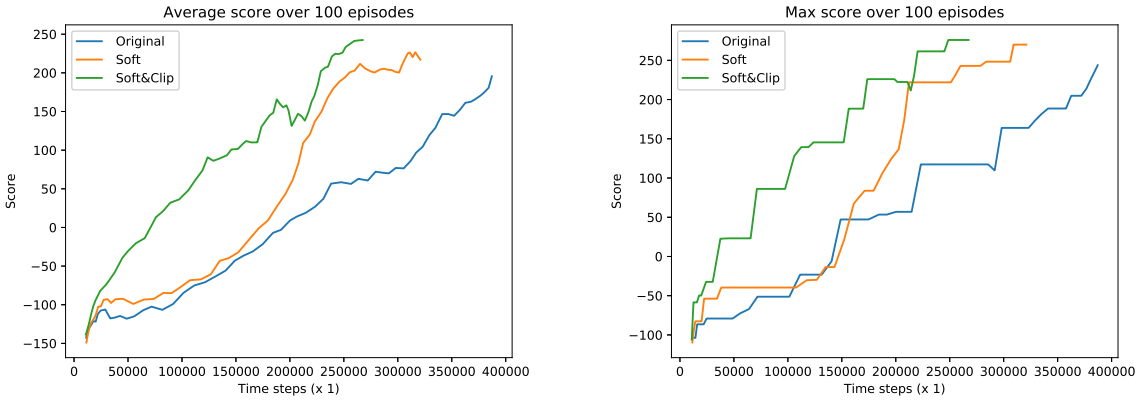
4 Results

4.1 Lunarlander

Following the hyperparameters listed in the Appendix, we make three individual experiment to test the effect of our implemention techinques. The first agent is based on original DQN algorithm, acting as the baseline. The second and the third agents

are both applied with soft update techiques while the reward of the third agent is even clipped(the signal function of reward). We train all three agents for the same episodes and the Figure 2 describes the difference of their trainig effect. In these two figures, the blue, orange and green line respectively represents the performance of three agent. Though all the agents finished the task, the difference of sample effiecncy is obvious from the graph. Equipped with soft update and clipped reward, the agent run all the episodes required the in the leats time steps and gain the best scores, while the agent with only soft update also beats the original agent not in time steps and scores. It means both two methods help the agent find a better policy and boost sample-effiency.

And we observe an interesting phenomenon that on the early training process, the agent with clipped reward exceeded the other agent clearly, indicating that the clipped reward help control the magnitude of the absolute value of Q resulting the quicker updating speed of Q network in the early time so better policy can be found earlier. As for the explanation for the effect of soft update, perhaps soft update is relevatively more stable and continous, thus more robust facing noise and more sample efficient for continous updating target in a mild way. .



(a) Average Score

(b) Maximum Score

Figure 2: The average score(a) and maximum scores(b) over 100 episodes curves among original DQN and two implementations.

4.2 RiverRaid

Befor training our agent with this difficult task, we have pretrained our agent for a few episodes, which range from 1000 to 2000, depending on the results, in order to choose appropriate hyperparameters. But due to limitation of our own PC, the buffer size can be set no more than 250000, which is a pity for a larger buffer replay may lead to a promising result. And we adopt soft update method here which is proved to be effective in the first task.

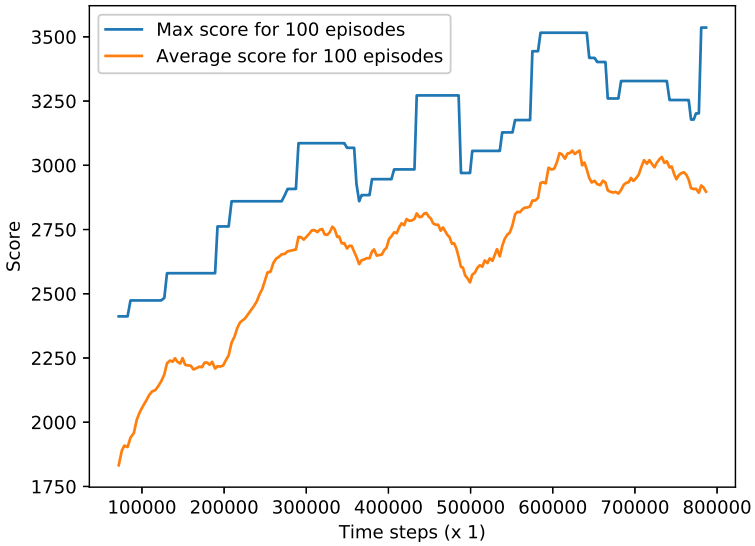


Figure 3: The training curve of our agent in the second task Riverraid. As depicted in the graph, the blue line and orange line represents respectively the max and the average score over 100 episodes during training.

Choosing the hyperparameters in Appendix A, we trained our agent for almost one day and half, obtaining the training result pictured in Figure 3. It shows that after nearly 0.8 millions frames training, Illustrated by the graph, in the end of the training the agent gains more than double scores gained by the agent at the beginning, which shows the learning progress of our agent. Also demonstrated by the graph, though the average score is always fluctuating, the lower bound of the performance and the uncertainty of our agent decreases obviously after 3000 episodes, which means our agent has the ability to handle some familiar problems with ease and accuracy.

However, it's still clear from the graph our training is not so stable even the ϵ is nearly 0.1 after training for more than 0.6 million frames. The problem may be derived from the DQN algorithm itself for it has no convergence guarantee in proof and the setting of hypermeters. It's worth noting that our buffer size is very limited which means the agent can only store limited memory And with no use of priortized buffer replay, our agent may more likey to forget the experience on its birthplace thus increasing the unstability. These problems may be solved by the implemention of DQN algorithms in reference with other papers, which requires further experiments.

Appendices

A Hyperparameter Settings

In Lunarlander, we set the hyper-parameters as follows

Table 1: Hyperparameters of LunarLander

Hyperparameter	Value
minibatch size	64
replay buffer size	1×10^5
main-target exchange factor τ	1
discount factor γ	0.99
learning rate lr	5×10^{-4}
RMS Alpha α	0.95
initial exploration	1.0
final exploration	0.01
ϵ decay index	0.995
max frame $T_{\max}$	1000

In RiverRaid, we set the hyper-parameters as follows

Table 2: Hyperparameters of RiverRaid

Hyperparameter	Value
minibatch size	32
replay buffer size	450000
buffer initial size	40000
main-target exchange factor τ	2×10^{-3}
discount factor γ	0.99
learning rate lr	0.003
RMS Alpha α	0.95
initial explorationϵ_s	1.0
final explorationϵ_f	0.05
ϵ decay parameter	40000

References

[1] Mnih Volodymyr, Kavukcuoglu Koray, Silver David, Graves Alex, Antonoglou Ioannis, W Daan, and R Martin. Playing atari with deep reinforcement learning. In *NIPS Deep Learning Workshop*, 2013.