Weekly report (June 28, 2018 - July 6, 2018)

Wyk
18250763559@163.com

Abstract

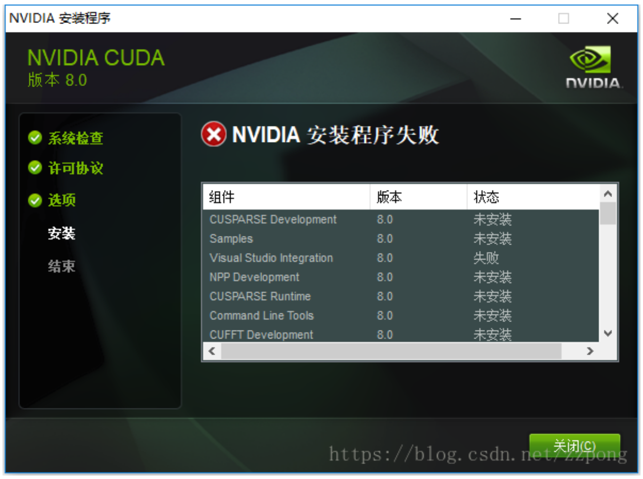
This week, I setup the basic environment for deep-learning at my new computer, implement neural style transfer with keras and roughly learn the concept of DQN.

1 Build deeplearning environment from scratch

Just recently I install a new PC at my dormitory, so I have to setup the environment from scratch. The process went relatively quick since I've done this before on my laptop.

1.1 Install tensorflow-gpu

The first thing I do is setting up windows10 operating system using the iso file I downloaded from msdn website.Then is the installation of anaconda to get the basic environment of python, after that I download visual studio 2015 and then install CUDA and cudnn which are necessary for tensorflow to train on a gpu. And this is when I encounter my first problem :



NVIDIA 安装程序失败

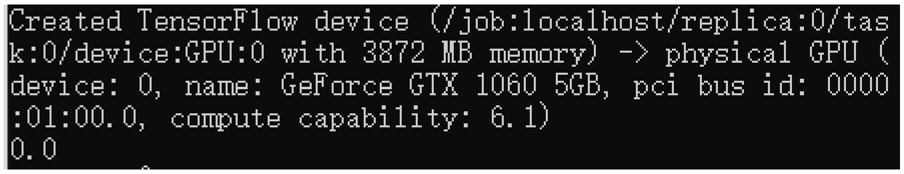
组件	版本	状态
CUSPARSE Development	8.0	未安装
Samples	8.0	未安装
Visual Studio Integration	8.0	失败
NPP Development	8.0	未安装
CUSPARSE Runtime	8.0	未安装
Command Line Tools	8.0	未安装
CUFFT Development	8.0	未安装

https://blog.csdn.net/zzpong

I search the problem online and find out others have encountered the same situation and provided a solution, so I follow the post and manual install the visual studio support (directly decompress installation files and copy them to the right path) to finally fix this. Then I use conda on anaconda prompt to install the tensorflow-gpu package and the required wheels is installed automatically at the same time. After the process completed I tested the following code:

```
>python
>>>import tensorflow as tf
>>> a= tf.constant(0.0)
>>> with tf.Session() as sess:
...     print(sess.run(a))
```

And the output is as follows:



```
Created TensorFlow device (/job:localhost/replica:0/tas
k:0/device:GPU:0 with 3872 MB memory) -> physical GPU (
device: 0, name: GeForce GTX 1060 5GB, pci bus id: 0000
:01:00.0, compute capability: 6.1)
0.0
```

which suggest the install is complete.

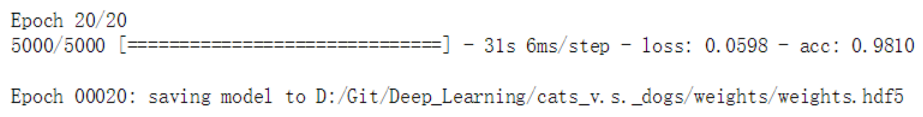
1.2 Install Keras and test a model

Keras is a high-level neural networks API, which runs on top of Tensorflow and makes it very easy to construct a deep network structure.
The installation is quite simple with one line in the powershell:

```
>>> pip install keras -U --pre
```

Then I open jupyter notebook to test the cats and dogs recognition model I wrote before at `https://github.com/W-yk/Deep_learning/blob/master/cats_v.s._dogs/cats_v.s._dogs_v2.py`

After training for 20 epochs on the whole dataset the accuracy reach 0.9810, it is not very high actually, likely due to the dropout layer I added to reduce overfitting. And the model performance



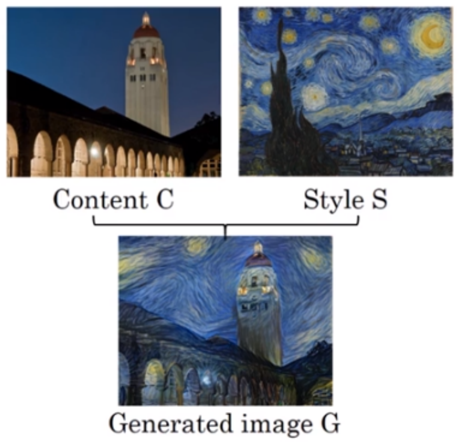
```
Epoch 20/20
5000/5000 [==============================] - 31s 6ms/step - loss: 0.0598 - acc: 0.9810

Epoch 00020: saving model to D:/Git/Deep_Learning/cats_v.s._dogs/weights/weights.hdf5
```

on the testset is 94% accurate, which still have a lot of room to improve but it shows that keras is properly installed.

2 Implement neural style transfer

I first learned about neural style transfer is from Andrew Ng's CNN course. Now after final exams I finally have the time to implement this myself.

The idea of Neural style transfer is to take the style of a image and the content of another then combine them into one new image. which is quite amazing for you can turn a simple photo into a painting with various styles.



So the first thing I do is download the original paper *A Neural Algorithm of Artistic Style*, and below is my understanding and implementation according to the paper.

The basic idea to make this happen is to train the pixels of the picture we generate. And to build the model we first need a pre-trained convolutional neural network to get the features of both image. I use keras to load the pre-trained VGG19 model without the top layers. And with some build in function in keras, the image pre-process function and de-process function is simple.

And then is the most important part of neural style transfer — define the cost function. According to the paper, the cost function has two parts – the content cost and the style cost.
For the content loss, it's defined as the squared-error loss between the two feature representation which in this case is the last convolutional layer's activation.

$$J_{content}(C,G) = \sum_{all\_entries} (a^{(G)} - a^{(C)})^2$$

To compute the style loss, first we need to understand what the gram matrix is. In linear algebra, the Gram matrix G of a set of vectors (v1,,vn) is the matrix of dot products, whose entries are $G_{ij} = v_i^T v_j = np.dot(v_i, v_j)$. In other words, Gij compares how similar the v_i is to v_j. And here we can use it to demonstrate the degree of features appearing together.

In NST we take the activations of some convolution layers to get both the primary features learned by early layers and the detailed features learned by deeper layers, and use them to compute the gram matrix. then we compute the square loss of the gram matrix between the style image and the generated image to get the cost function of style.

$$J_{style}(S,G) = \sum_{layers} \sum_{i=1}^{n_C} \sum_{j=1}^{n_C} (G_{ij}^{(S)} - G_{ij}^{(G)})^2$$

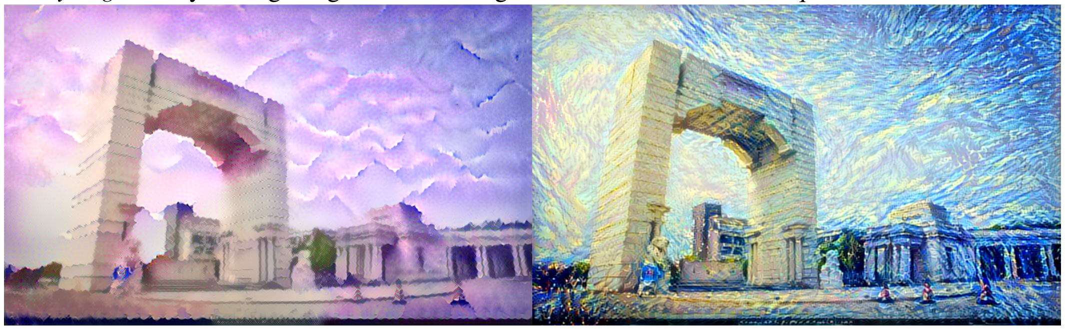
And finally we can combine these two cost after times the weights α and β

$$J(G) = \alpha \cdot J_{content}(C,G) + \beta \cdot J_{style}(S,G)$$

From the original paper, the cost function of content and style function is actually divided by the scale of the matrix. But since we have α and β, in my own implementation I ignore this step.

After building the cost function, the model is almost complete. We can then train the model by updating the pixels in the generated image with gradient. to actual implement this part of code I reference the keras example code and finally finish the model.

I feed the picture of the triumphal arch in SJTU as content image with a wash painting and *The Starry Night* as style image to get the following results. Which I think are quite beautiful.



The complete code of my implementation is at `https://github.com/W-yk/Deep_learning/blob/master/neural_style_transfer/v1.py`

3 learning DQN

I sign up for the DQN implementing project, so I try to learn as much as I can on the topic first, follow are my learning process and notes.

3.1 Undderstanding basic concepts

To get the full picture of DQN first thing is to learn some concepts in reinforcement learning. Below are my own understandings, forgive me if I stumble at some point.

3.1.1 Markov decision process

In most of the deep learning problems the framework can be ascribed to a MDP problem. In a mrakov process, each new state of the environment is only depend on the one previous state, and MDP is basiclly this process adding the actions a agent can take.
A markov process can be described as a set $\{S, A, P, R, \gamma\}$ in which S denotes the state set, A denotes the action set, P denotes the state transfer prosibility, R is the reward gain after an action , and γ is the discount factor.

3.1.2 Value function and Q-function

The goal of reinforcement learning is to find the best policy for actions in the given states of a MDP. And the policy (π) we defined is the distribution of the actions considering states. It's often stochastic because this way AI agent can run trials in the environment to get various samples we can actually work on.
And to measure how good a policy is, we need a reward function. In MDP space, we already have R which is the feedback after the state shifts. We can introduce a 'Return function' to measure how good a state is :

$$G_t = R_{t+1} + \gamma R_{t+2} + ... = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1}$$

in which R is the reward feedback, γ is the discount factor. This equation means a state is whether good or bad depends on how much reward it will get in the future, and with γ the current feedback matter the most. But as you can see in this equation, unless a whole process is completed we can't get any return of a selected state. To address this we can define the 'Value function' which is the expectation of the future rewards for a current state :

$$v_\pi(s) = E[G_t|S_t = s]$$

Similarly we have 'Action-Value function' which is also called 'Q-function'. It's basically the same as the 'Value function' except the actions a agent take is considered which makes it dependent on both state and action. And its equation is :

$$Q_\pi(s, a) = E[r_{t+1} + \gamma r_{t+2} + ...|s, a]$$

. With this two functin we can then improve our policy to get better rewards.

3.1.3 Bellman equation

The way we can compute or estimate the 'Value function' ang 'Q-function' is though the Bellman equation :

$$v_\pi(s) = E[R_{t+1} + \gamma v(s_{t+1})|s]$$

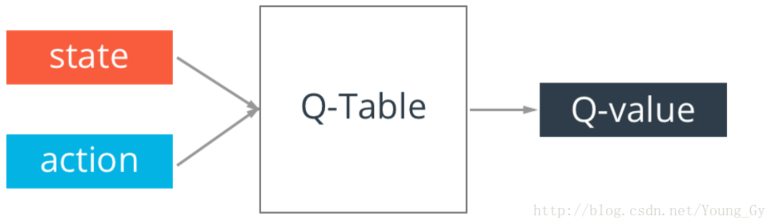
and

$$Q_\pi(s, a) = E[r_{t+1} + \gamma Q(s_{t+1}, a_{t+1})|s, a]$$

It's just the former functions irrational form. Note that there is still a future return needed, but In practice, we can use the data we sampled from previous trials to update the current function and it can be mathematically proved that the function will converged to something we want.

3.2 Q-learning

In a RL problem we want to find the best policy that get the most rewards in a MDP, and the idea of Q-learning is try to fit a Q-function, and by choosing the actions that maximize it in each state we get the best policy. So we need something called Q-table to store the value across different actions and states.



normally its a 2D matrix with its column as states and row as actions, by using Bellman equation to update this Q-table we can then select the best actions in every state.

The detailed Q-Learning algorithm goes as follows:

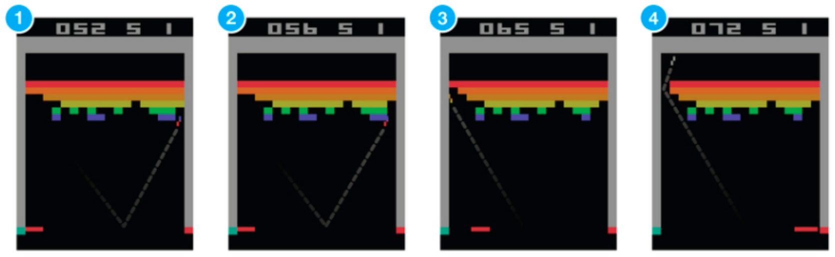
1. Set the gamma parameter, and environment rewards in matrix R.
2. Initialize matrix Q to zero.
3. For each episode:

   Select a random initial state.
   Do While the goal state hasn't been reached.

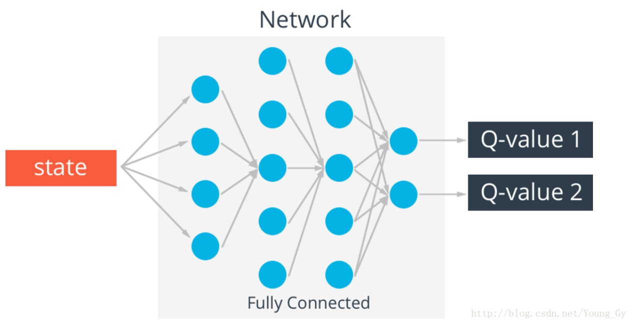
   - Select one among all possible actions for the current state.
   - Using this possible action, consider going to the next state.
   - Get maximum Q value for this next state based on all possible actions.
   - Compute: Q(state, action) = R(state, action) + Gamma * Max[Q(next state, all actions)]
   - Set the next state as the current state.

3.3 DQN

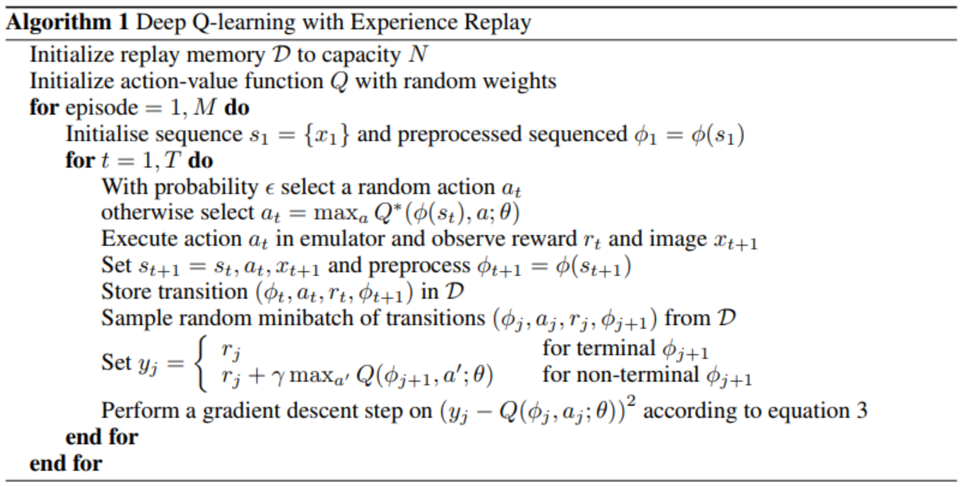
Q-learning focus on the update of Q-table, but what if the states can't fit inside it?
Like when try to play a Atari game:



The state is a whole image. The total number of state is $256^{210\times160}$ which is more than the number of particles in the whole universe.
So instead of using a list to store the value, we adopt a method called Value Function Approximation. The idea is to fit a function with a state as input and the Q-function value of each actions as output. And this is how we applied deeplearning into a RL problem.



By using the Q-function value computed with Bellman equation as labels, fitting the new function become a supervised learning problem. And then we can apply all those models in deeplearning to solve this. The code in the original paper *Playing Atari with Deep Reinforcement Learning* is as follows:



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 $\epsilon$ 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
```

I hope I can dig in this code and try to implement it nextweek, maybe.

4 Summary

It has been my first week since join in AISIG, and I think its a very enriching week. Aside from the work I mentioned above I learnt to use LaTeX by search online whenever I have a problem writing this paper, like including a image, adding a math formula ...

Questions

1. In the cat and dog recognition CNN model I wrote, I divide the training set into four parts due to the limited memory, I wonder how much will training some epochs on one part and switch to another affect my training process? I mean I have heard of the Covariate shift problem when the dataset changed. Will Batch normalization help in this case?

Plans for next week

1. Try to implement a DQN algorithm to solve Cart Pole game in gym library.
2. Learning the sequence model class on coursera and finish the first week's assignment.
3. Continue to read the book *Programming Collective Intelligence* by Toby Segaran