DD2424 Deep Learning in Data Science
Assignment 2

Rui Qu
rqu@kth.se

April 15, 2019

1 Introduction

With prior knowledge of CIFAR-10 dataset and one layer network in Assignment1, Assignment2 needs to modify the code to two layer network which will have more parameters and different functions, e.g. 1) evaluate network, forward pass 2) compute gradients, backward pass. It's also required to implement hyper parameter optimization for regularization term and learning rate.

CIFAR-10 dataset
The CIFAR-10 dataset consists of 60000 32x32x3(RGB) color images in 10 classes, with 6000 images per class. There are 50000 training images and 10000 test images.

Mini-batch gradient descent with momentum
Details in instruction of assignment 2.

2 Gradient Check

With understanding of numerical &analytic derivative, it's easy to compute numerical &analytic gradient. First numerical derivative:

$$f'(x') = \frac{\partial f(x)}{\partial x}|_{x=x'} \tag{1}$$

First analytic derivative:

$$f'(x') \approx \frac{f(x'+\delta) - f(x'-\delta)}{2\delta} \tag{2}$$

Then compare the two gradient via relative error:

```
def rel_err(x, y):
 e=np.max(np.abs(x-y)/np.maximum(1e-8,np.abs(x)+np.abs(y)))
 return e
```

In this exercise, I set regularization lambda to 0 and epoch to 10, which improves efficiency of the computation. The relative error is very low, smaller than 1e-6. I could say that I successfully managed to write the functions to correctly compute the gradient. The code to compute slow numerical gradient is given in the attachment.

3 Algorithm

The code is written in Python on the basis of A1. Pseudo-code as following:

Algorithm 1 Mini-batch gradient descent

1: Data preprocess, dataset loading to data($size : N \times d$), one-hot($size : N \times K$), label($size : N \times 1$) implementing Z-scores Normalization
2: Generate random $w_1(size : m \times d)$ $w_2(size : K \times m)$, $b_1(size : m \times 1)$ $b_2(size : K \times 1)$ by He Initialization, where m is the number of hidden layer nodes.
3: Generate mini-batch Xbatch for data and Ybatch for label
4: **repeat**
5: Take j_{th} batch in mini-batches, using $argmax(SOFTMAX)$ to predict the label Pbatch of each image in Xbatch
6: Compute Cross-Entropy Loss and minimize cost function
7: Compute gradient with momentum, update weights and bias
8: **until** n-epochs
return Best epoch &cost; plotting of training &validation cost and representation of weight matrix

NB: Cross Entropy Loss with Softmax function and its derivative [1]

$$\frac{\partial L}{\partial w_i} = -\sum_k y_k \frac{\partial \log p_k}{\partial w_i} = -\sum_k y_k \frac{1}{p_k} \frac{\partial p_k}{\partial w_i} \tag{3}$$

$$= -y_i(1 - p_i) - \sum_{k \neq i} y_k \frac{1}{p_k}(-p_k p_i) \tag{4}$$

$$= -y_i(1 - p_i) + \sum_{k \neq i} y_k(p_i) \tag{5}$$

$$= -y_i + y_i p_i + \sum_{k \neq i} y_k(p_i) \tag{6}$$

$$= p_i \left(\sum_k y_k \right) - y_i = p_i - y_i \tag{7}$$

$\frac{\partial L}{\partial w_i} = p_i - y_i$ makes gradient computation easy to implement by code.

4 Exercise

4.1 Read in the data & initialize the parameters

The main purpose of Exercise1 is to preprocess the data. Before loading CIFAR-10 dataset, I take k-cross validation in a size of 1000 validation set. For all training, testing and validation set I take Z-scores normalization to reduce data redundancy and improve data integrity.

$$z = \frac{x - \mu}{\sigma} \tag{8}$$

$$where\ \mu : mean,\ \sigma : standard\ deviation \tag{9}$$

For initialization of weights and bias see pseudo-code in Section 3. Specifically, I use numpy function (np.random.normal) to randomly generate Gaussian distribution (He initialization with mean:0 and variance:$\sqrt{\frac{2}{n}}$).

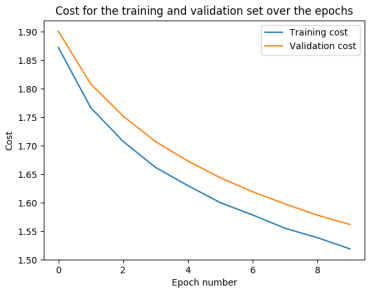
4.2 Gradient descent with momentum

When updating weights and bias, I added momentum term to them, using a new function:

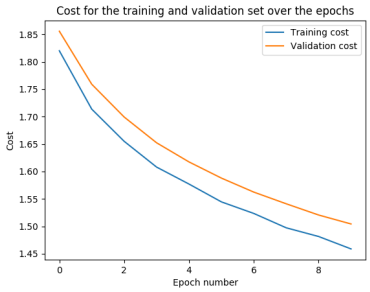
$$V_{dw} = \rho V_{dw} + (1 - \rho)dw \tag{10}$$

$$W = W - \eta V_{dw} \tag{11}$$

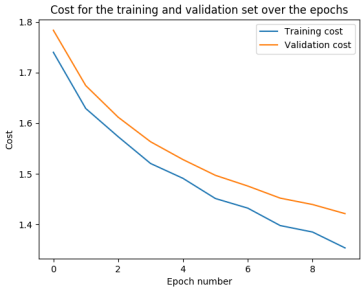
Then I tested how the momentum hyper parameter ρ affect the performance by setting $\rho\ to$ 0, .3, .6, .9 respectively with other parameters: cyclical learning rate(Section 4.3), lambda=0, n epochs=10, batch size=64, hidden nodes=200, ReLU



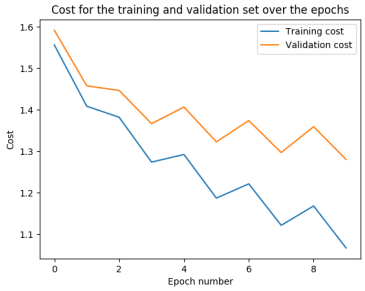
(a) $\rho = 0$ Testset accuracy: 0.4678



(b) $\rho = .3$ Testset accuracy: 0.4849



(c) $\rho = .6$ Testset accuracy: 0.5129



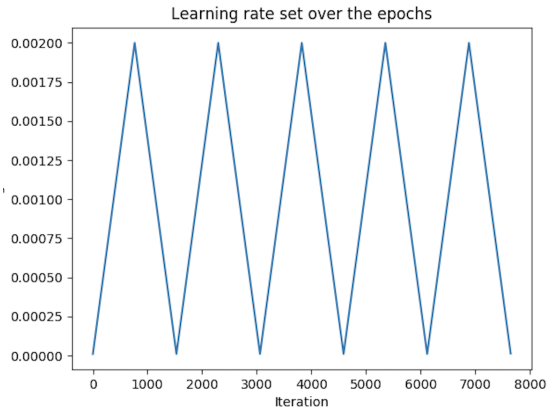
(d) $\rho = .9$ Testset accuracy: 0.5549

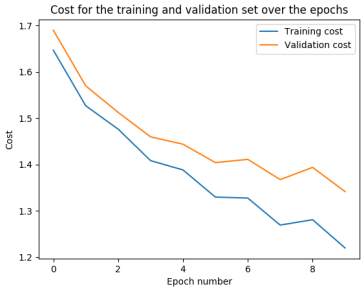
As shown in the figures above, increasing momentum term ρ significantly reduces converge time and improve final accuracy. Thus I'd set $\rho = .9$ in this exercise, which is also the default value from most researchers.

4.3 Cyclical learning rates

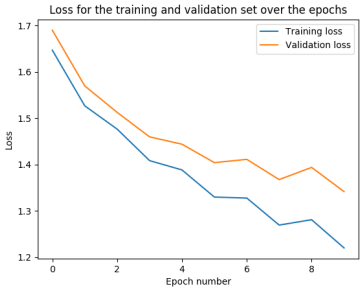
Details of cyclical learning rates is written in instruction of Assignment2. For each batch in a data set, I set a condition that eta increase from maximal to minimal when i is odd and vice versa. Maximal eta: 2e-3, minimal eta: 1e-5 The figure variation of cyclical eta plotted by my code as below, which self-evidently convinced me that it's a bug free implementation of cyclical learning rates. lambda=0, n epochs=10, hidden nodes=200, ReLU

1.batch size =128 Testset accuracy: 0.5397



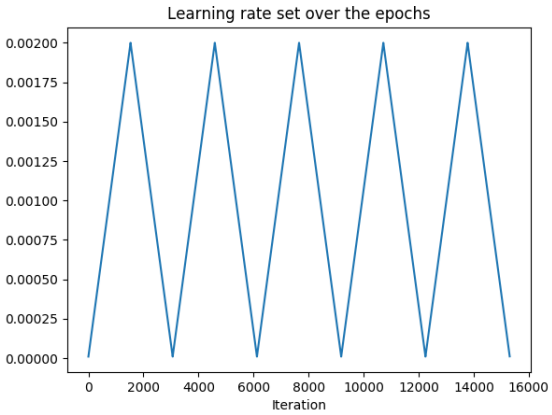


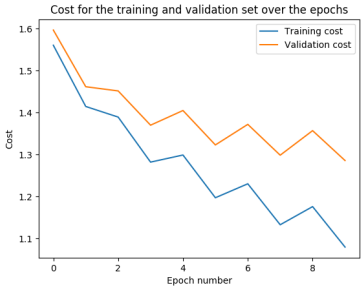
(e) Training &validation cost



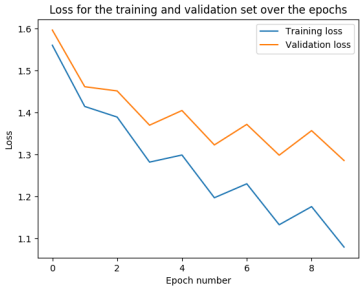
(f) Training &validation loss

2.batch size=64 Testset accuracy: 0.5524





(g) Training &validation cost



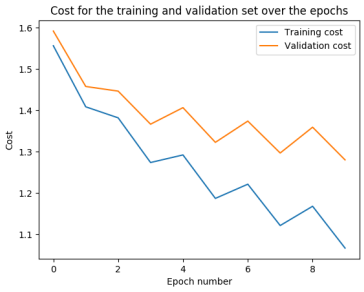
(h) Training &validation loss

4.4 Regularization term

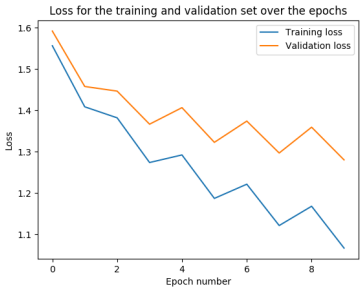
4.4.1 Coarse search

On top of Assignment1, it gets highest testset accuracy when $\eta = .01, \lambda = 0$. Thus, I took cyclical learning rate explained in Section 4.3 and manually select parameter lambda in a rangeλ =[0, .0001, .0025, .025,] in the condition of: cyclical eta, epochs=10, batch size=64, hidden layer nodes=200, ReLU to get test set accuracy and plotting of training &validation cost &loss.

1. $\lambda = 0$ Test accuracy: 0.5549

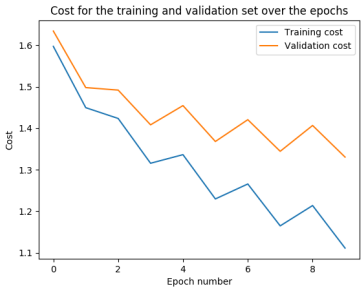


(i) Training &validation cost

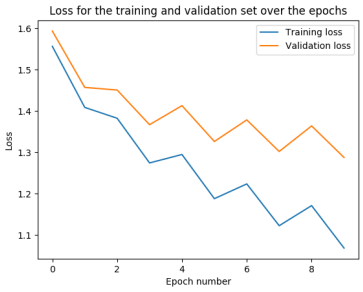


(j) Training &validation loss

2. $\lambda = .0001$ Testset accuracy: 0.5567

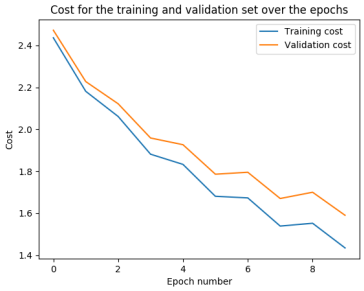


(k) Training &validation cost

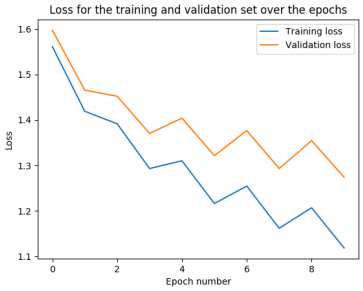


(l) Training &validation loss

3. $\lambda = .0025$ Testset accuracy: 0.5606



(m) Training &validation cost



(n) Training &validation loss

5 Exercise Bonus

5.1 Fine search

Since it's getting lower accuracy when λ is over .025 in prior exercise of Coarse search, I randomly set regularization term $\lambda \in [0, .025]$ to compute validation cost and iterate for 100 times to search the best λ in the condition of: cyclical eta, epochs=10, batch size=64, hidden layer nodes=200, ReLU. See the function 'random search' in the code.
After 100 iterations, I got the best $\lambda : 0.0021$.

5.2 Data shuffling

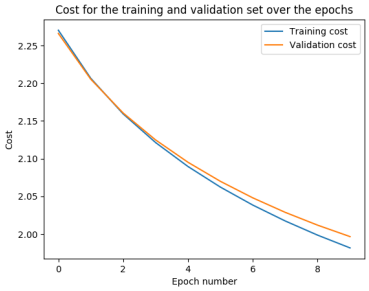
As what's done in Assignment1,I set shuffling step in the code to get better generalization.See the function 'data shuffling' in the code.

5.3 Batch size

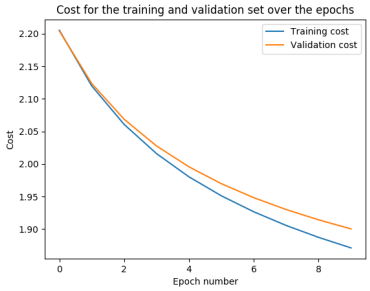
The CPU and GPU memory in a computer are stored in binary mode. In order to improve computation efficiency, I take the batch size in the power of 2 e.g. 32, 64, 128, etc. in this exercise.

5.4 Leaky ReLU activation function

For different activation function, I used Leaky ReLU which has a small slope for negative values, instead of ReLU's altogether zero . For example, leaky ReLU $y = 0.01x$ when $x < 0$. I took a simple test on ReLU and Leaky ReLU(slope:0.01) activation function with parameters: decay learning rate, lambda=0, batch size=64, n epoch=10.



(o) ReLU



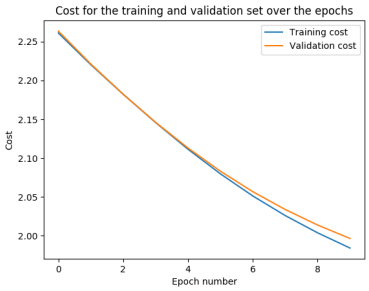
(p) Leaky ReLU

Activation function	Testset accuracy
ReLU	0.3064
Leaky ReLU	0.3495

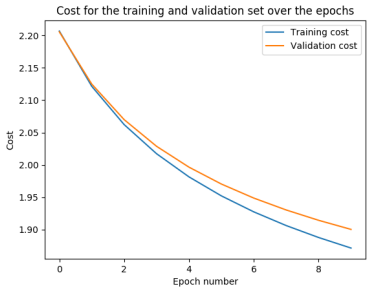
According to the testset accuracy,I would take Leaky ReLU in the coming experiment.

5.5 Hidden layer nodes

For setting to hidden layer nodes, there are some empirically-derived rules-of-thumb, of these, the most commonly relied on is the optimal size of the hidden layer is usually between the size of the input and size of the output layers.[2] Thus I took a simple test on hidden layer nodes of 50(as required) and 200 with decay learning rate, lambda=0, batch size=64, n epoch=10.



(q) 50 hidden layer nodes



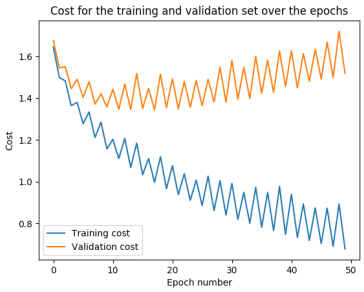
(r) 200 hidden layer nodes

Hidden layer nodes	Testset accuracy	Computation time of last epoch
50	0.3115	16.91s/it
200	0.3487	52.20s/it

As shown in the table, the outcome behaves better with more hidden layer nodes but it takes longer time to converge. Via tqdm library in Python, it's convenient to collect a list of time cost of each epoch. I recorded the last epoch of 50 and 200 nodes comparetively in this test.

5.6 Best performance

According to the exercise before, I could adjust parameters and run the code to get the best performance in the condition of taking cyclical learning rate, $\lambda = .00021$, $\rho = .9$,hidden layer nodes=200, batch size=64, epochs=50, Leaky ReLU, shuffling data.

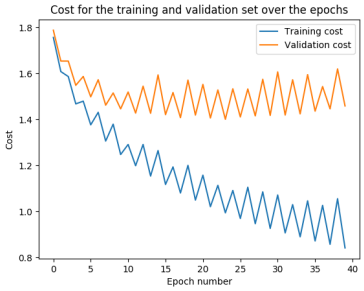


(s) Cost

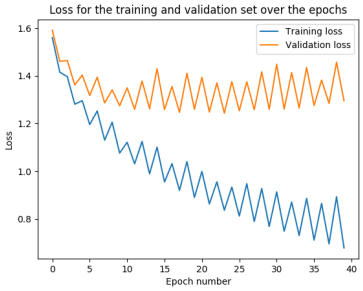
(t) Loss

Testset accuracy: 0.5644

We can see the cost of training set keep reducing but it caused overfitting problem. Thus I slightly increase λ to .0005.



(u) Cost



(v) Loss

Testset accuracy: 0.5669

6 Conclusion

In Assignment2, I build a 2 layer neural network by implementing mini-batch gradient with monument and did research in hyper-parameters optimization like learning rate, regularization term, hidden layer nodes, also other method to improve prediction accuracy like data preprocess, shuffling. It's really interesting to play with those parameters and my final best test accuracy on CIFAR-10 dataset is 0.5644, which is much better than that in Assignment1 but still needs improvement. We can see the cost of training set keep reducing but it caused overfitting problem.

I have a problem in the instruction of Exercise 2. n_s linearly related to n/n_{batch}, it's confused when programming in Python. Maybe it's better to demonstrate the relationship between n_s and epoch in the future instruction. And in my code I didn't use n_s to plot the cost but epoch which is easy to observe and understand. Perhaps I need to improve my programming ability to fix it.

A Screen cut of Best performance

The last 5 epochs:

References

[1] Eli Bendersky. The softmax function and its derivative, October 2016.

[2] Jeff Heaton. *Programming Neural Networks with Encog3*. Heaton Research, Incorporated, 2011.