DD2424 Assignment 2

Wei Wang

wewang@kth.se

Royal Institute of Technology

DD2424 Deep Learning in Data Science

Assignment Report

Introduction

The of object of this assignment is to train and test a two layer network with multiple outputs to classify images from the CIFAR-10 dataset. To learning how to evaluate the network (the forward pass) and compute the gradients (the backward pass). To learn how to set good parameters for network's regulation term and the learning rate.

Exercise

All these codes are written on my laptop but my laptop is too slow. So I use the KTH computer to run these script and these screenshots are mainly from both KTH computer and my laptop.

Exercise1

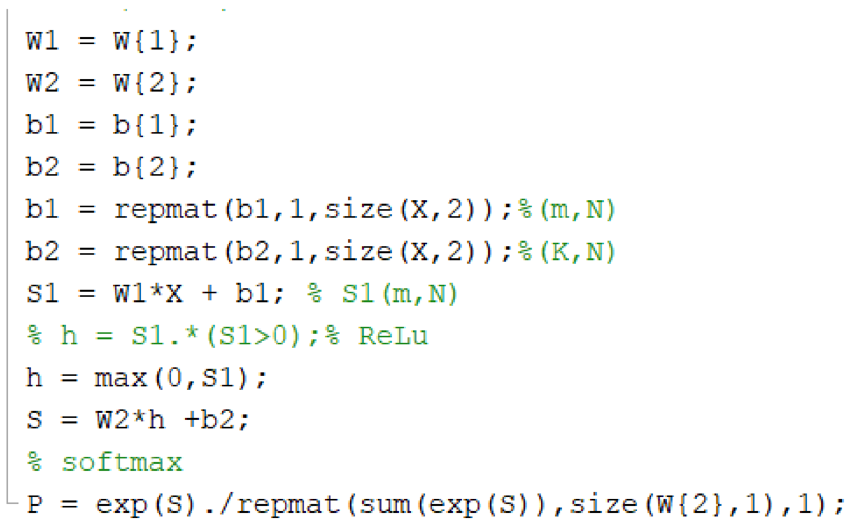
Read in the data & initialize the parameters of the network
I use the ***LoadBatch*** function mainly from assignment 1 and transform all data to zero mean. All these parameters are initialized by the function ***ParameterInit***.The weight matrices are initialized to 0 mean and 0.001 deviation. The hidden layers are set to be 50.

Exercise2

Compute the gradients for the network parameters
To compute the gradients for the network parameters, I have to change the code based on assignment 1.

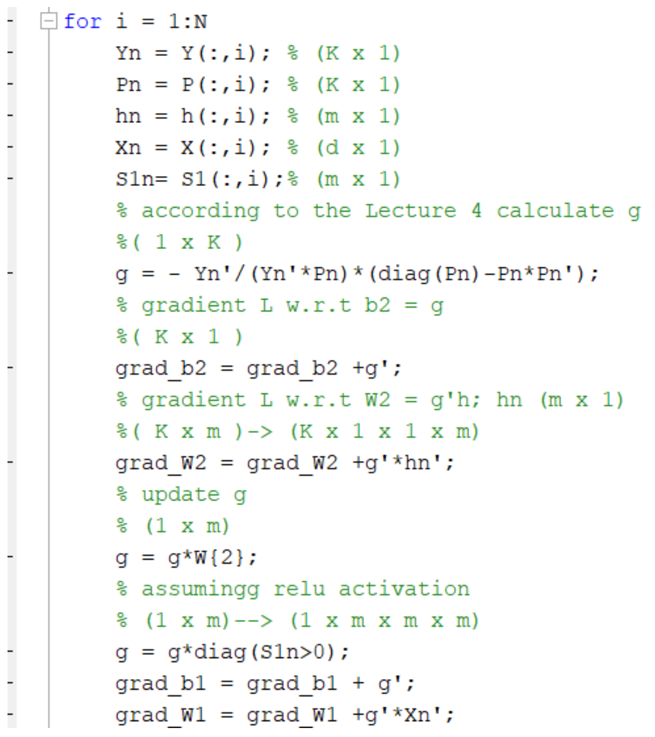
1. ***[P,S,h,S1] = EvaluateClassifier2(X,W,b)***
   EvaluateClassifier2 is calculated by equation 1-4 and the inter hidden layers are also returned.

2. ***ComputeGradients2(X,h,S1, Y, P, W, lambda)***
   This function is calculated by following the instruction in Lecture 4. When calculating gradients, the hidden layer h and S1 is also used. These variables are called from Evaluation( ) function. Since there are so many variables to be used in these function, the dimension of each variable matrices should be taken attention with.



```
W1 = W{1};
W2 = W{2};
b1 = b{1};
b2 = b{2};
b1 = repmat(b1,1,size(X,2));%(m,N)
b2 = repmat(b2,1,size(X,2));%(K,N)
S1 = W1*X + b1; % S1(m,N)
% h = S1.*(S1>0);% ReLu
h = max(0,S1);
S = W2*h +b2;
% softmax
P = exp(S)./repmat(sum(exp(S)),size(W{2},1),1);
```

Figure 1: function EvaluateClassifier2()

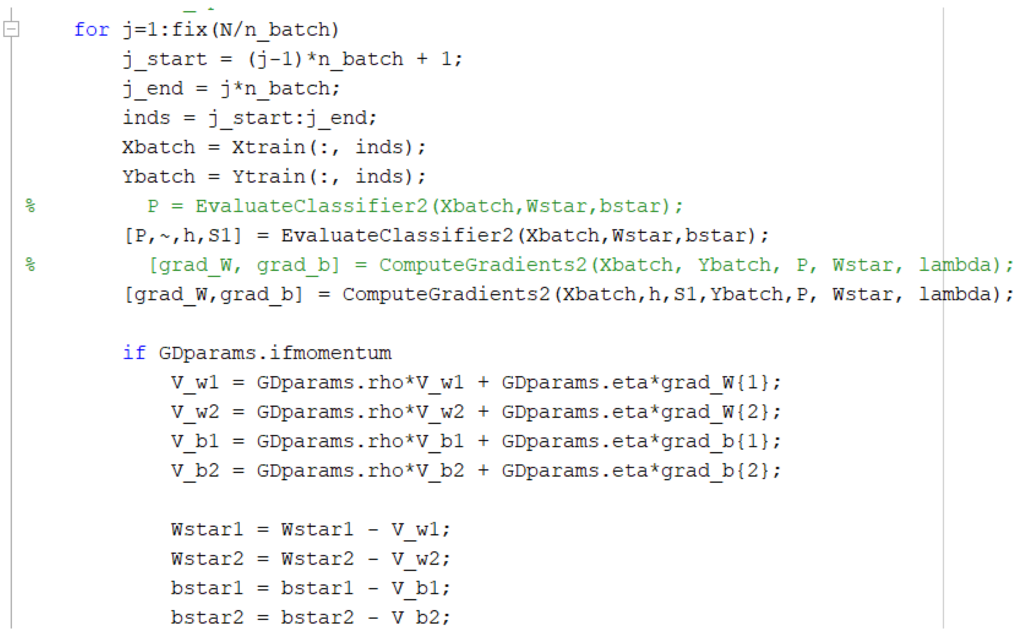


```
for i = 1:N
    Yn = Y(:,i); % (K x 1)
    Pn = P(:,i); % (K x 1)
    hn = h(:,i); % (m x 1)
    Xn = X(:,i); % (d x 1)
    S1n= S1(:,i);% (m x 1)
    % according to the Lecture 4 calculate g
    %( 1 x K )
    g = - Yn'/(Yn'*Pn)*(diag(Pn)-Pn*Pn');
    % gradient L w.r.t b2 = g
    %( K x 1 )
    grad_b2 = grad_b2 +g';
    % gradient L w.r.t W2 = g'h; hn (m x 1)
    %( K x m )-> (K x 1 x 1 x m)
    grad_W2 = grad_W2 +g'*hn';
    % update g
    % (1 x m)
    g = g*W{2};
    % assumingg relu activation
    % (1 x m)--> (1 x m x m x m)
    g = g*diag(S1n>0);
    grad_b1 = grad_b1 + g';
    grad_W1 = grad_W1 +g'*Xn';
```

Figure 2: function ComputeGradients2()

3. ***MiniBatchGD2(X, Y, GDparams, W, b, lambda)***
   In this function, I need extract the training data into batches at first, then it should conduct an iteration based on how many batches you choose. In each iteration, the gradient is updated based on the equation 10-11. The equation is updated by subtract the gradient obtained by each specific batch multiplied

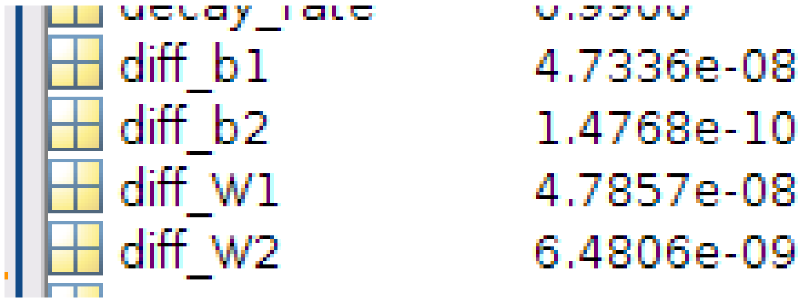
learning rate. Where should be cautious of is that do not forget update the W and b in ComputeGradients2( ). I forget to update W and b at beginning. After a long-time debugging, I noticed this small but crucial mistake.



```
    for j=1:fix(N/n_batch)
        j_start = (j-1)*n_batch + 1;
        j_end = j*n_batch;
        inds = j_start:j_end;
        Xbatch = Xtrain(:, inds);
        Ybatch = Ytrain(:, inds);
%          P = EvaluateClassifier2(Xbatch,Wstar,bstar);
        [P,~,h,S1] = EvaluateClassifier2(Xbatch,Wstar,bstar);
%          [grad_W, grad_b] = ComputeGradients2(Xbatch, Ybatch, P, Wstar, lambda);
        [grad_W,grad_b] = ComputeGradients2(Xbatch,h,S1,Ybatch,P, Wstar, lambda);

        if GDparams.ifmomentum
            V_w1 = GDparams.rho*V_w1 + GDparams.eta*grad_W{1};
            V_w2 = GDparams.rho*V_w2 + GDparams.eta*grad_W{2};
            V_b1 = GDparams.rho*V_b1 + GDparams.eta*grad_b{1};
            V_b2 = GDparams.rho*V_b2 + GDparams.eta*grad_b{2};

            Wstar1 = Wstar1 - V_w1;
            Wstar2 = Wstar2 - V_w2;
            bstar1 = bstar1 - V_b1;
            bstar2 = bstar2 - V_b2;
```

Figure 3: function MiniBatchGD2( )

To make sure the gradient is obtained correctly, I extract 1:10 images from training dataset and compared the outcome between ComputeGradient2( )and ComputeGradsNumSlow( ) using $h \approx 1e^{-}5$ Finally I got the difference as follows. These differences



```
diff_b1        4.7336e-08
diff_b2        1.4768e-10
diff_W1        4.7857e-08
diff_W2        6.4806e-09
```

Figure 4: gradient differences between these two methods

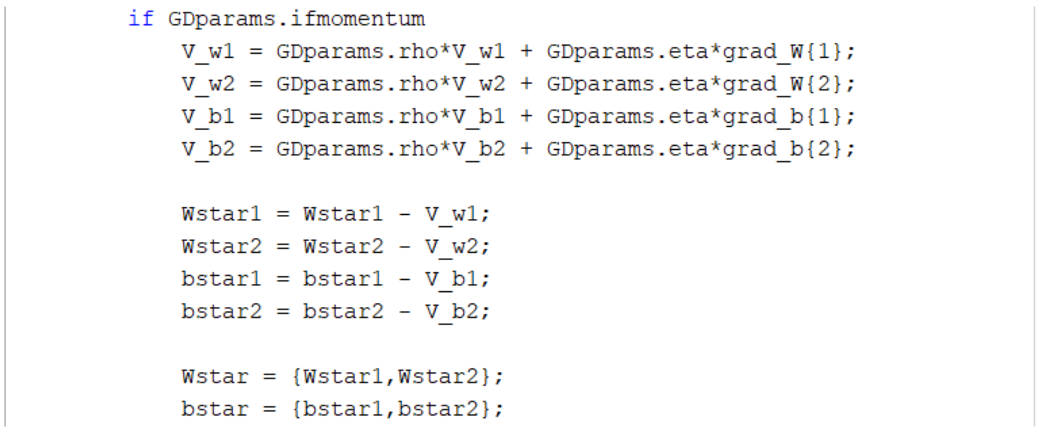
are small enough to prove the calculation of gradient is probably right.

Exercise3

Add momentum to your update step

Initialize a momentum vector (matrix) v0 for each parameter of the network at each time step t:

$$V_t = \rho V_{t-1} + \eta \frac{\partial J}{\partial \theta}$$
$$\theta_t = \theta_{t-1} - V_t$$



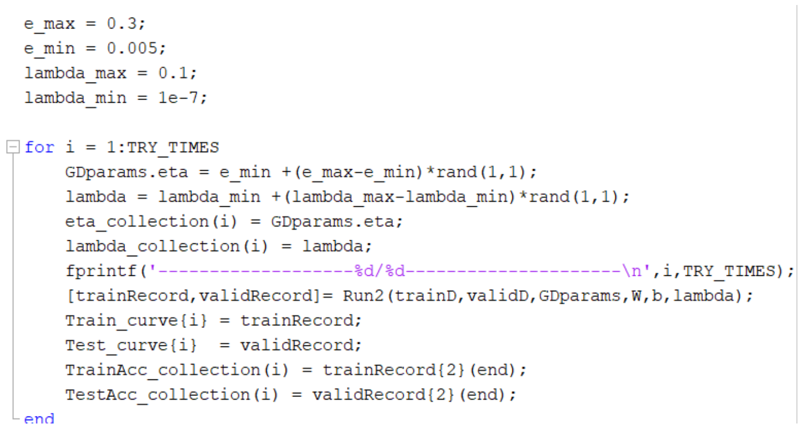
```
if GDparams.ifmomentum
    V_w1 = GDparams.rho*V_w1 + GDparams.eta*grad_W{1};
    V_w2 = GDparams.rho*V_w2 + GDparams.eta*grad_W{2};
    V_b1 = GDparams.rho*V_b1 + GDparams.eta*grad_b{1};
    V_b2 = GDparams.rho*V_b2 + GDparams.eta*grad_b{2};

    Wstar1 = Wstar1 - V_w1;
    Wstar2 = Wstar2 - V_w2;
    bstar1 = bstar1 - V_b1;
    bstar2 = bstar2 - V_b2;

    Wstar = {Wstar1,Wstar2};
    bstar = {bstar1,bstar2};
```

Figure 5: momentum

Exercise4

Training your network

To conduct coarse-to-fine random search, I initialize lambda and eta by setting the range (min, max), and generate them randomly using rand(1,1)



```
e_max = 0.3;
e_min = 0.005;
lambda_max = 0.1;
lambda_min = 1e-7;

for i = 1:TRY_TIMES
    GDparams.eta = e_min +(e_max-e_min)*rand(1,1);
    lambda = lambda_min +(lambda_max-lambda_min)*rand(1,1);
    eta_collection(i) = GDparams.eta;
    lambda_collection(i) = lambda;
    fprintf('------------------%d/%d--------------------\n',i,TRY_TIMES);
    [trainRecord,validRecord]= Run2(trainD,validD,GDparams,W,b,lambda);
    Train_curve{i} = trainRecord;
    Test_curve{i}  = validRecord;
    TrainAcc_collection(i) = trainRecord{2}(end);
    TestAcc_collection(i) = validRecord{2}(end);
end
```

Figure 6: Coarse to fine

Task

1. State how you checked your analytic gradient computations and whether you think that your gradient computations were bug free.

1. Keep the all the codes' steps follow the instruction from Lecture 4

2. Extract 1:10 images from training data and compare the gradient difference between function *ComputeGradient2()* and *ComputGradNumsSlow()*. The difference of W1 is $4.7857e^{-08}$, W2 is$6.4806e^{-09}$,b1 is$4.7336e^{-08}$and b2 is $1.4768e^{-10}$. See Figure 4.

3. In this part, I set " ifmomentum = false " so the momentum is turned off. eta = 0.1, lambda = 0, I did a 10 batch gradient descent and ran 200 epochs. The loss curve is shown below. As we can see, the training cost curve went down to close 0 after 50 epochs and the valid cost curve went up gradually. It shows that the model is over-fitted.

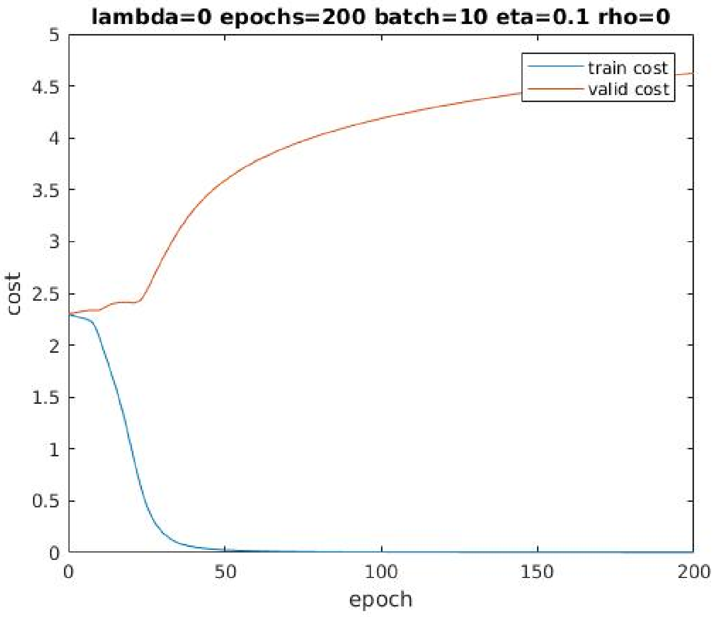


Figure 7: Test gradient

All these 3 points shows that the gradient computations were bug free.

2. Comment on how much faster the momentum term made your training.

To conduct this part. I initialize the parameters: rng_number = 40, hidden_notes =50, n_epochs = 10, n_batch = 100, eta = 0.1, check = false, lambda = 0, ifplot = true, rho = 0, ifmomentum = false, decay_rate = 1. After 10 epochs, I got the curve below: After this, I change the "ifmomentum" parameter to be true and set rho =

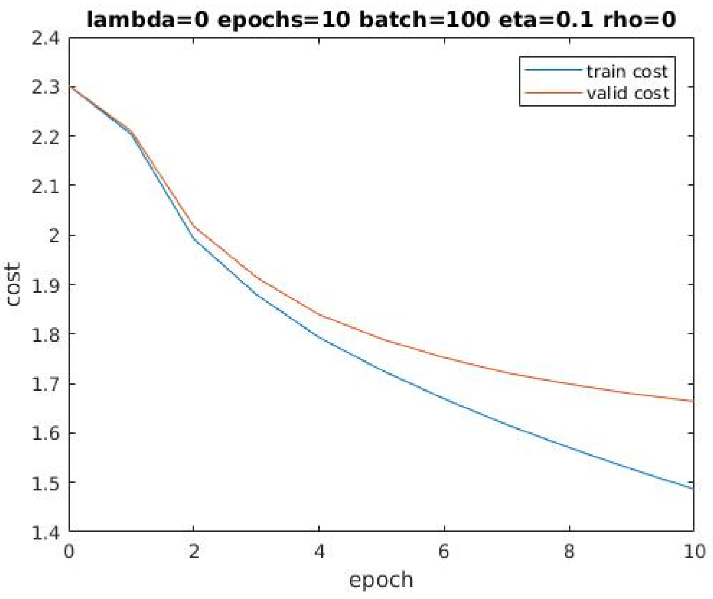


Figure 8: without momentum, without decay

{0.5 0.7 0.8 0.9 0.99} five group. In figure 9, we can see that after 10 epochs, training

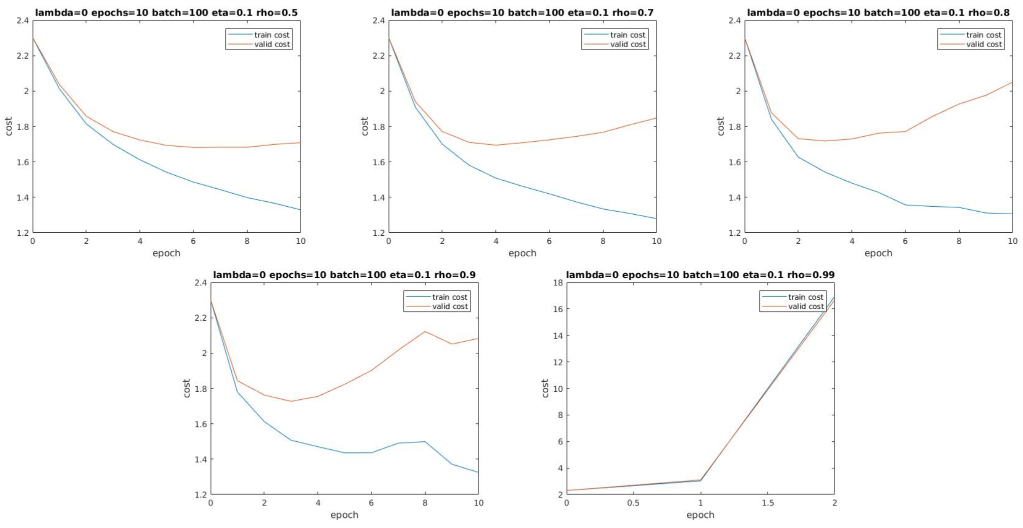


Figure 9: with momentum, without decay

curve's loss without momentum is around 1.5 while using momentum, after 10 epochs the loss is around 1.3, which shows that momentum can speed up training. When momentum is too big, the curve is not so stable. when choose rho = 0.99 without decay, the training curve reached infinity after 2 epochs. Figure 10 is the learning

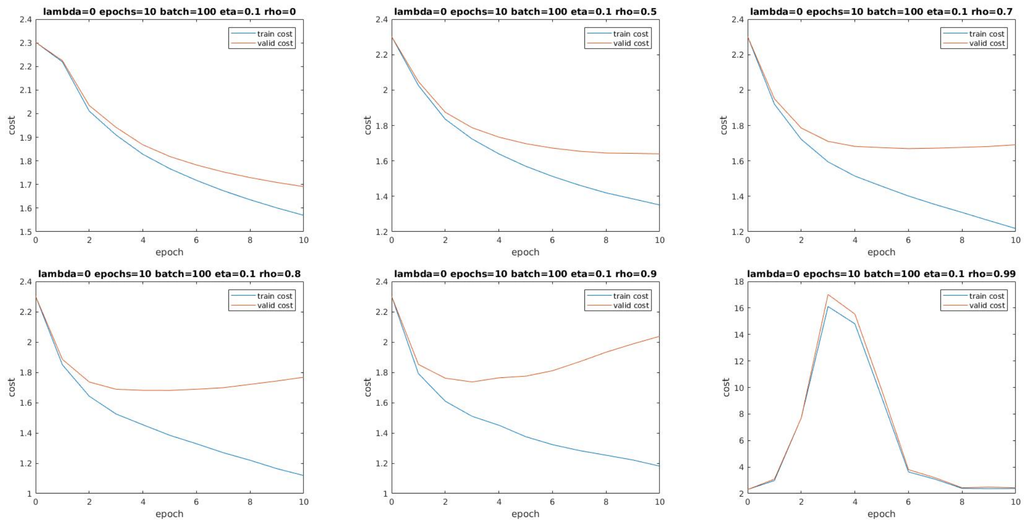


Figure 10: with decay 0.95

curve with decay rate 0.95. The training without momentum reached 1.5 loss after 10 epoch and the training with momentum reached around 1.2 loss. the training curves become more smooth after using decay and have better outcome. When I change the

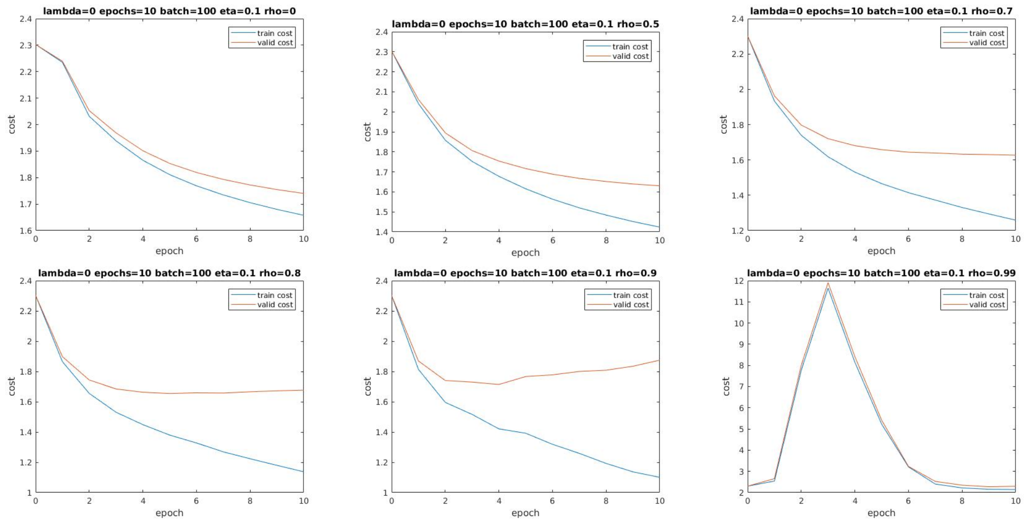


Figure 11: with decay 0.9

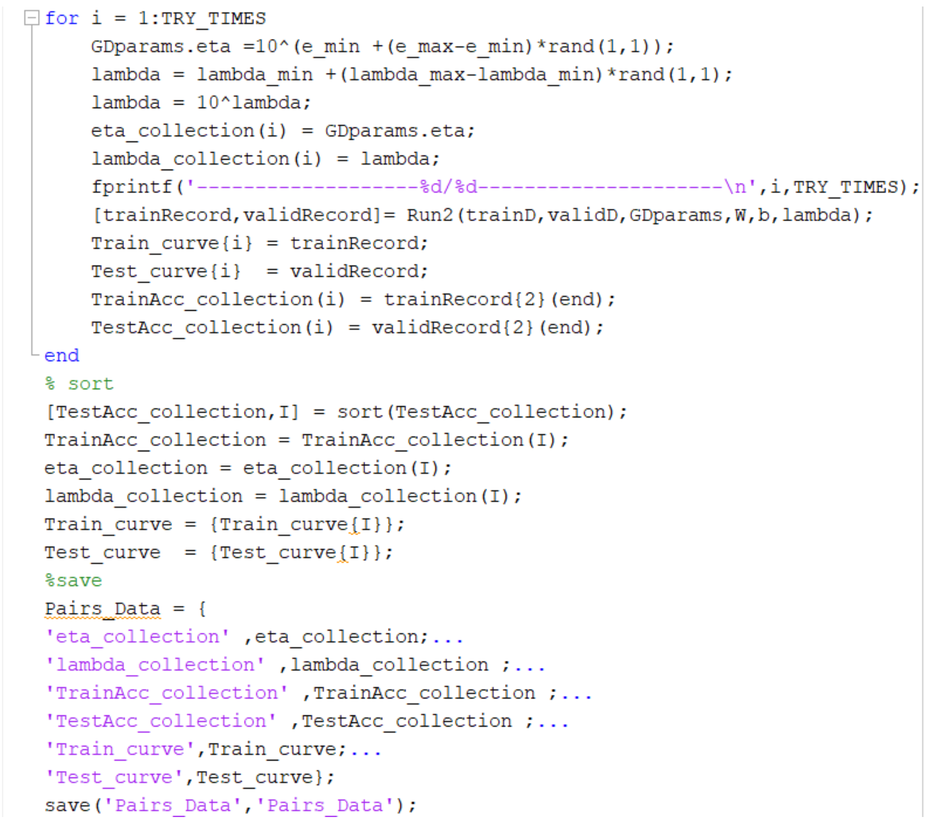
decay rate to be 0.9. I found the original training without momentum learned more slowly, it only reached a loss around 1.67 and the momentum 0.9 and 0.8 reached better outcome, getting around 1.1 loss. But it seems that momentum = 0.8 has more smooth curve than momentum = 0.9.

3. State the range of the values you searched for lambda and eta, the number of epochs used for training during the coarse search and the hyper-parameter settings for the 3 best performing networks you trained.

Consider the experience in task 2, I chose momentum =0.88 and decay_rate = 0.92 to be the default parameter. For coarse tuning, I choose the range of eta to be $e^1 - e^{-3}$ and the lambda to be $e^{-1} - e^{-7}$. That were large range for both eta and lambda, so I planed to check 50 times for coarse checking. I wrote a code for searching and recording the perform of training validation curve with different eta and lambda. To prevent some big parameters, I add the code in run2.m:

```
if train_cost(i+1)>3*train_cost(1)
    fprintf('bad parameter, too large\n');
    break
end
```

I also wrote an iteration for searching coarse parameters of eta and lambda: save the



```
for i = 1:TRY_TIMES
    GDparams.eta =10^(e_min +(e_max-e_min)*rand(1,1));
    lambda = lambda_min +(lambda_max-lambda_min)*rand(1,1);
    lambda = 10^lambda;
    eta_collection(i) = GDparams.eta;
    lambda_collection(i) = lambda;
    fprintf('-------------------%d/%d---------------------\n',i,TRY_TIMES);
    [trainRecord,validRecord]= Run2(trainD,validD,GDparams,W,b,lambda);
    Train_curve{i} = trainRecord;
    Test_curve{i}  = validRecord;
    TrainAcc_collection(i) = trainRecord{2}(end);
    TestAcc_collection(i) = validRecord{2}(end);
end
% sort
[TestAcc_collection,I] = sort(TestAcc_collection);
TrainAcc_collection = TrainAcc_collection(I);
eta_collection = eta_collection(I);
lambda_collection = lambda_collection(I);
Train_curve = {Train_curve{I}};
Test_curve  = {Test_curve{I}};
%save
Pairs_Data = {
'eta_collection' ,eta_collection;...
'lambda_collection' ,lambda_collection ;...
'TrainAcc_collection' ,TrainAcc_collection ;...
'TestAcc_collection' ,TestAcc_collection ;...
'Train_curve',Train_curve;...
'Test_curve',Test_curve};
save('Pairs_Data','Pairs_Data');
```

Figure 12: coarse searching

these training pairs data into ***Pairs_Data.mat***. The initial parameters are: epoch = 10, TRY_TIMES = 50 momentum = 0.88, decay_rate = 0.92, eta $=\{e^{1} - e^{-3}\}$, lambda $=\{e^{-1} - e^{-7}\}$. After running, I record all pairs' test accuracy and sort them ascending. Here is the table for top 10: There are some pairs whose training accuracy is as high

Table 1: Top10 coarse searching testing accuracy

	eta	lambda	train accuracy	test accuracy
top1	0.0325	4.1697e-5	0.5391	0.4372
top2	0.0605	0.0014	0.5814	0.4336
top3	0.0480	0.0061	0.5062	0.4331
top4	0.0362	5.4488e-4	0.5494	0.4327
top5	0.0595	1.5789e-7	0.5959	0.4290
top6	0.0193	1.0803e-5	0.4761	0.4210
top7	0.0189	2.2938e-6	0.4701	0.4144
top8	0.1251	0.002	0.5521	0.4076
top9	0.1030	2.2638e-5	0.6085	0.4073
top10	0.1369	1.0230e-5	0.6122	0.4068

as above 0.6 but the test accuracy is not good, that is because the overfitting. In the top5 in the table, we can see that the mode of eta is around 0.03-0.06 and the mode of lambda is around $e^{-2} - e^{-5}$. It is hopeful that the eta and lambda in this range will reach a better outcome. Here is the top 3 training curve:

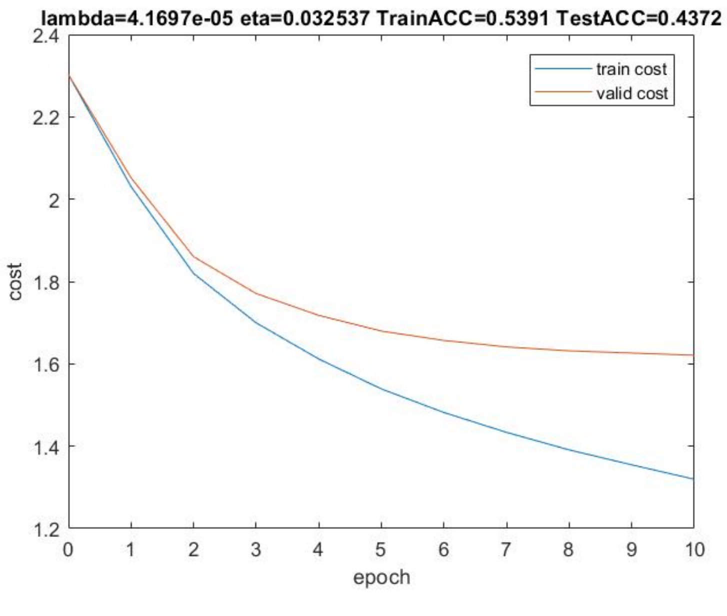


Figure 13: top 3 coarse training curve- top1

From the figures we can see, when lambda gets larger, the validation curve and training curve has small difference.

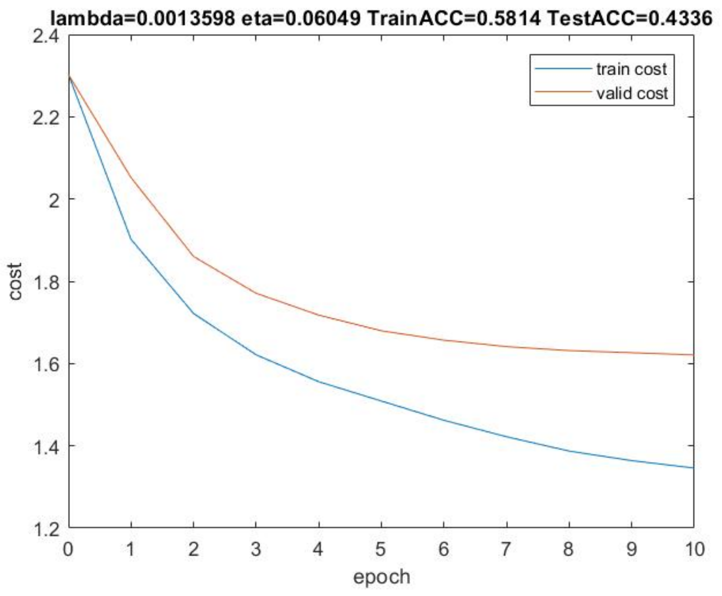


Figure 14: top 3 coarse training curve- top2

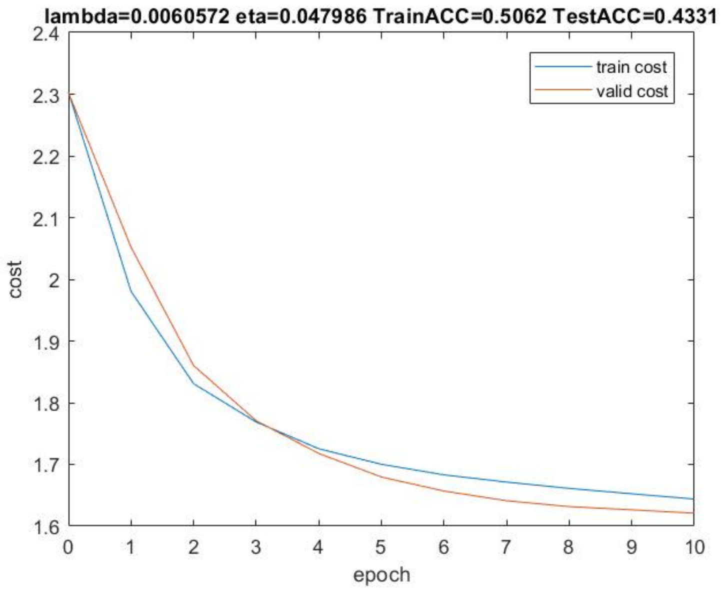


Figure 15: top 3 coarse training curve- top3

4. State the range of the values you searched for lambda and eta, the number of epochs used for training during the ne search, and the hyper-parameter settings for the 3 best performing networks you trained.

As mentioned in task 3, when eta is around 0.02-0.06 and the lambda is around $e^{-2} - e^{-6}$, the model has better performance. For the fine searching, I changed the decay rate from 0.92 - 0.93, hoping to have a faster learning curve. The initial parameters are: epoch = 10, TRY_TIMES = 100 momentum = 0.88, decay_rate = 0.93, eta =0.06 − 0.02, lambda =$\{e^{-2} - e^{-6}\}$. The table is shown below: The training

Table 2: Top5 fine searching testing accuracy

	eta	lambda	train accuracy	test accuracy
top1	0.0316	0.00144	0.5281	0.4412
top2	0.0284	5.295e-5	0.5298	0.4401
top3	0.0298	0.0013	0.5258	0.4397
top4	0.0288	0.00049	0.5294	0.4392
top5	0.0293	0.00112	0.5225	0.4385

curves of fine searching are shown below:

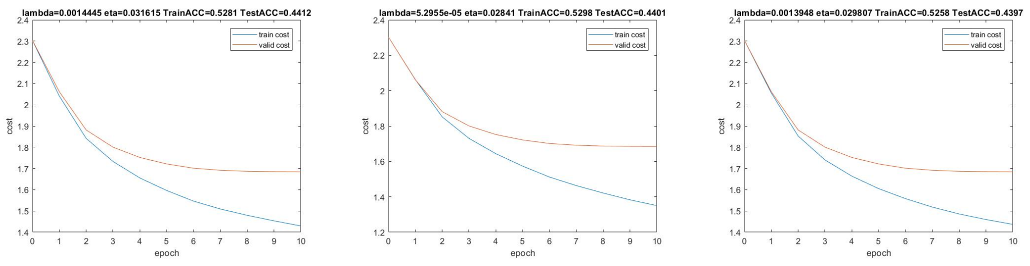


Figure 16: top 3 fine training curve

Then I began to try 30 epochs using ***Test & training*** Data with the best parameter pair: But I found that after 30 epochs, the model's performance: train

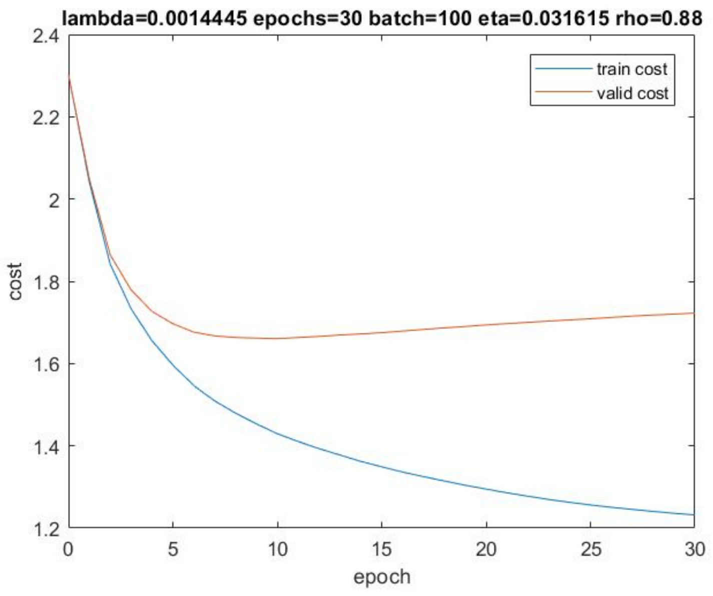


Figure 17: Test

acc=0.633800 **Test acc = 0.452300**. Apparently, the model is kind of over-fitted. so I tried the third time by shifting the lambda range to $\{e^{-1} - e^{-3}\}$ hoping to prevent over-fitting and changed the decay rate from 0.93 to 0.95 and change the momentum from 0.88 to 0.89 to accelerate speed. Here is the third cycle:

Table 3: Top5 the second time fine testing accuracy

	eta	lambda	train accuracy	test accuracy
top1	0.0336	0.0049	0.5162	0.4401
top2	0.0334	0.0054	0.5095	0.4392
top3	0.0325	0.0023	0.5392	0.4385
top4	0.0260	0.0025	0.5145	0.4381
top5	0.0268	0.0032	0.5101	0.4373

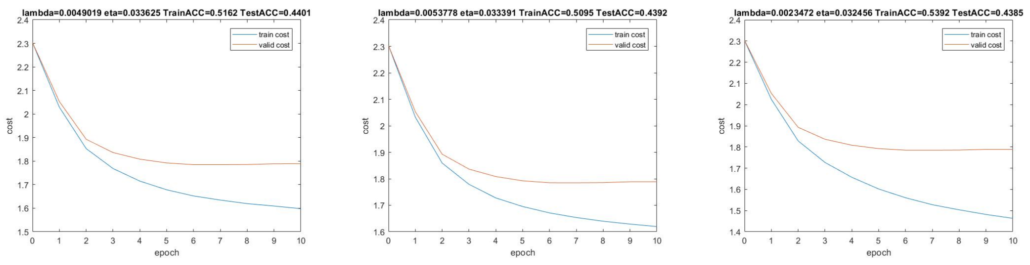


Figure 18: top 3 the second time fine training curve
Training with batch 1 and test using test data for 30 epochs:

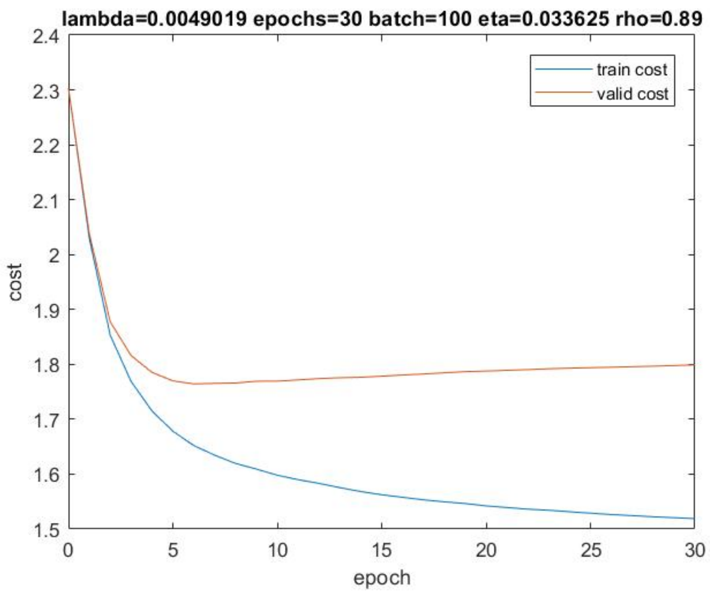


Figure 19: Second Test

We can see, the curve seems better and without so much difference between training curve and test curve.

5. For your best found hyper-parameter setting (according to performance on the validation set), train the network on all the training data (all the batch data), except for 1000 examples in a valida- tion set, for 30 epochs. Plot the training and validation cost after each epoch of training and then report the learnt network'sperformance on the test data.

Using batch 1,3,4,5 for training, the curve of this two fine searching pairs are shown below:

momentum:0.88, decay rate:0.93, eta:0.0316, lambda:$1.44e^{-3}$:

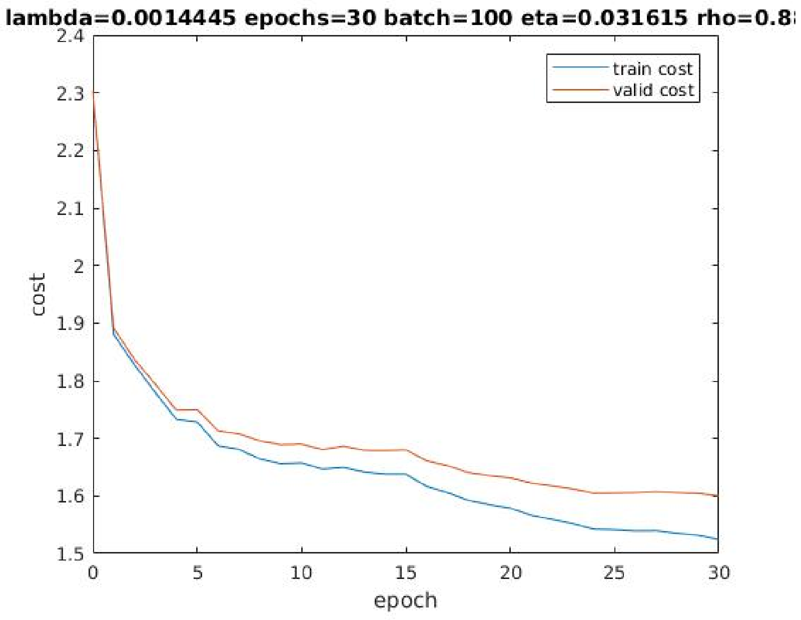
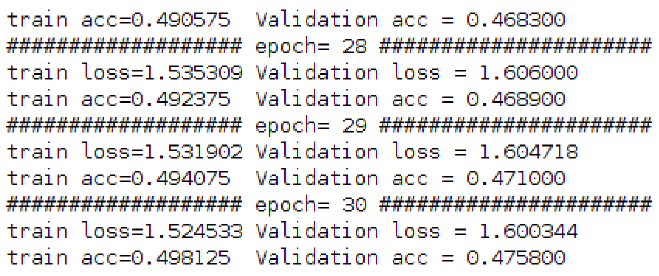


Figure 20: first Test



```
train acc=0.490575  Validation acc = 0.468300
################### epoch= 28 ######################
train loss=1.535309 Validation loss = 1.606000
train acc=0.492375  Validation acc = 0.468900
################### epoch= 29 ######################
train loss=1.531902 Validation loss = 1.604718
train acc=0.494075  Validation acc = 0.471000
################### epoch= 30 ######################
train loss=1.524533 Validation loss = 1.600344
train acc=0.498125  Validation acc = 0.475800
```

Figure 21: first test

momentum:0.89, decay rate:0.94, eta:0.0336, lambda:4.9^{-3} :

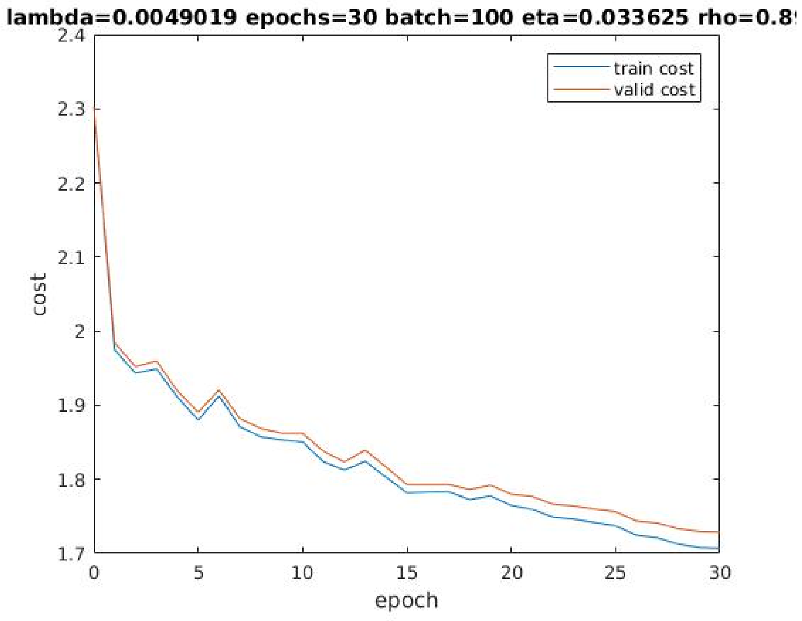
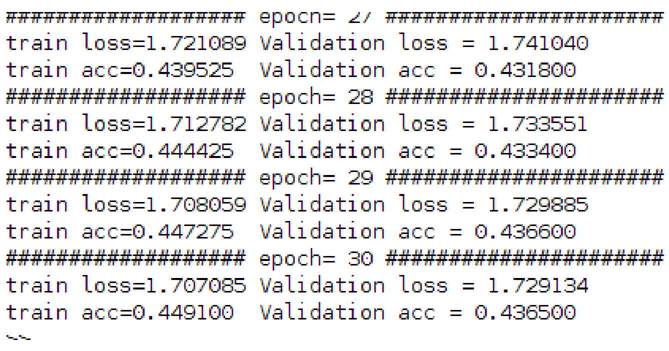


Figure 22: Second Test



```
#################### epoch= 27 ######################
train loss=1.721089 Validation loss = 1.741040
train acc=0.439525  Validation acc = 0.431800
#################### epoch= 28 ######################
train loss=1.712782 Validation loss = 1.733551
train acc=0.444425  Validation acc = 0.433400
#################### epoch= 29 ######################
train loss=1.708059 Validation loss = 1.729885
train acc=0.447275  Validation acc = 0.436600
#################### epoch= 30 ######################
train loss=1.707085 Validation loss = 1.729134
train acc=0.449100  Validation acc = 0.436500
```

Figure 23: first test

It's obvious that the fist parameter pair is better, getting around 47.5% test accuracy. What I learned from them is:

1. When you tried hard to find suitable parameter in specific training batch, the parameters may not be suitable for another training batch.

2. Enlarge dataset can also prevent over-fitting

3. When you find a suitable lambda parameter for small data, it might be harmful when you enlarge the dataset because both of these two method can prevent over-fitting. Too large regularization term can be harmful for model.