Machine Problem 1 Report

Yihui He*

I. Implementation

KEY points of implementing each part of neural network are illustrated as follow.
All matrix multiplications are done via np.dot(). Element-wise matrix operation is done via "*".

A. Forward pass and Loss

In Forward pass *np.maximum* between z and 0, is used to represent *ReLU* activation function.

In Softmax, first raise *scores* to the power of e element-wisely, then divide each element by row sum, finally we get $a^{(3)}$.

When computing Loss, pick up each true label element in each row is tricky: *a3[range(len(a3)),y]*. Part of Forward pass is shown below. Other part of codes can be found in source files.

```
a2=np.maximum(X.dot(W1)+b1,0)
scores=a2.dot(W2)+b2
a3=exp_scores\
  /(np.sum(exp_scores,1))[:,None]
```

B. Backward pass and Gradient check

Firstly for each input, we need to compute $\delta_j^{(3)}$ for each output unit j:

$$\delta_j^{(3)} = \begin{cases} \frac{1}{N}p(Y=j|X=x_i) - \frac{1}{N} & j = y_i \\ \frac{1}{N}p(Y=j|X=x_i) & j \neq y_i \end{cases}$$

```
delta_3=a3
delta_3[range(len(a3)),y]=\
  a3[range(len(a3)),y]-1
delta_3/=len(a3)
grads['W2']=a2.T.dot(delta_3)+reg*W2
grads['b2']=np.sum(delta_3,0)
```

Secondly, in the front hidden layer, for each input, we need to compute $\delta_j^{(3)}$ for each hidden node j:

$$\delta_j^{(2)} = (w_j^{(2)})^T \delta^{(3)} \circ f'(z_j^{(2)})$$

Note that, the second operator is Hadamard product.

```
dF=np.ones(np.shape(a2))
dF[a2==0.0]=0
delta_2=delta_3.dot(W2.T)*dF
grads['W1']=X.T.dot(delta_2)+reg*W1
grads['b1']=np.sum(delta_2,0)
```

To avoid divide by zero in gradient check, I made a small modification to the formula:

$$\frac{|A-B|}{max(10^{-8}, |A+B|)} \leq \delta$$

max errors among inputs are as follow:
w_1 3.56e-09 b_1 2.74e-09
w_2 3.44e-09 b_2 4.45e-11

C. Train and Predict

To perform a minibatching, *np.random.choice* can be used, and set *replace=True* to avoid same inputs being used. Then update each hyperparameter using SGD.

```
rand_idx=np.random.choice(
  num_train,batch_size,replace=False)
X_batch=X[rand_idx]
y_batch=y[rand_idx]
for var in self.params:
  self.params[var]-=\
    learning_rate*grads[var]
```

As for predicting, run forward propagation, and use *np.argmax(,1)*to find predicted y for each input.

```
y_pred=np.argmax(np.maximum(0,\
  (X.dot(self.params["W1"])\
  +self.params['b1']))\
  .dot(self.params['W2'])+\
  self.params['b2'],1)
```

II. Model Building

Basically there are two ways to tune a neural net: grid search and random search. For simplicity, I employ grid search to tune 3 hyperparameters: number of neurons in the hidden layer, regularization strength, and learning rate. My tuning procedure is twofold. First, run a coarse-grained search. Second, based on the result of first step, run a fine-grained search around top results.

*Exchange student, 2nd year CS undergrad from Xi'an Jiaotong University, eli.he@foxmail.com

For coarse-grained search, Number of neurons range from 50 to 550, step 50. Regularization strength and learning rate are selected from geometrical sequences. Regularization strength range from 0.5×10^{-3} to 0.5×10^{2}, with ratio 10. Learning rate range from 1×10^{-5} to 1×10^{-1}, with ratio 10. During this, top results have about **50%** validation accuracy. Some Top results are shown below:

Figures and tables should be labeled and numbered, such as in Table **??** and Fig. 2.

TABLE I
TOP ACCURACY

hidden neurons	learning rate	regularization strength	validation accuracy
350	0.001	0.05	0.516
400	0.001	0.005	0.509
250	0.001	0.0005	0.505
250	0.001	0.05	0.501
150	0.001	0.005	0.5
500	0.001	0.05	0.5

For fine-grained search, I picked up one of the above top results. And search around it's hyperparameters. It can reach a validation accuracy of **52%**. After found a suitable set of hyperparameters, I start tuning numbers of iterations, and batch size. Finally, our original neural network reach a validation accuracy of **56%**.

III. Extra Credits

I put results in the next section, in order to compare our original two-layers neural network, with other enhanced neural networks.

A. momentum and other update methods

Implementation of momentum needs to change hyperparameters update code in *train*.

```
self.cache[param]=np.zeros(
grads[param].shape)
self.cache[param]=arg*self.cache[param
-learning_rate*grads[param]
self.params[param]+=self.cache[param]
```

In order to tune momentum parameter and compare result with SGD, all other hyperparameters are fixed. Intuitively, momentum should speed up training procedure. I test momentum and SGD with 1000 iterations to see their converge rate. It turns out that enjoys better converge rates.

I also tried other update methods. Nesterov momentum, which is a look-ahead version of momentum.

```
v_prev = cache[param]
cache[param]=arg*cache[param]\
  -learning_rate*grads[param]
self.params[param]+=-arg*v_prev\
  +(1+arg)*cache[param]
```

RMSprop, which is a per-parameter adaptive learning rate method.

```
cache[param]=arg*cache[param]\
+(1-arg)*np.power(grads[param],2)
self.params[param]-=learning_rate\
*grads[param]\
/np.sqrt(cache[param]+1e-8)
```

It turns out that, Momentum, Nesterov momentum and RMSprop all have a better converge rate than SGD. However, difference between these three update methods is ambiguous. Performances of these methods are compared in table III

TABLE II
DIFFERENCES BETWEEN UPDATE METHODS

accuracy	Train	Validation	Test
SGD	.27	.28	.28
Momentum	.49	.472	.458
Nesterov	.471	.452	.461
RMSprop	.477	.458	.475

B. Dropout

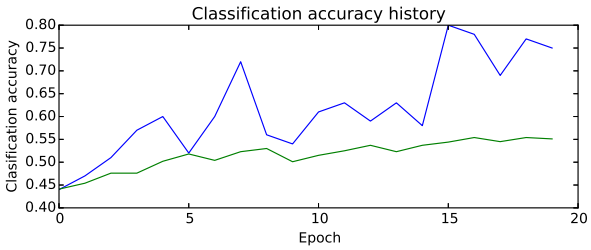
Because we only have one layer of hidden nodes, dropout only needs to be performed once per batch. In code, it only needs minor change to Forward pass.

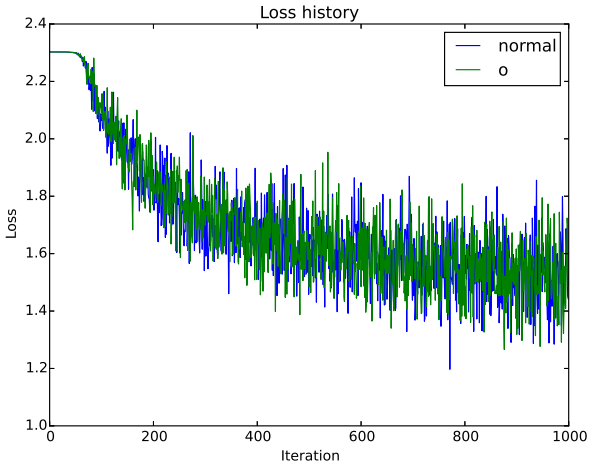
```
a2*=(np.random.randn(*a2.shape)<p)/p
```

Dropout is said to be a easy way to prevent overfitting, and it is also an ensemble of multi models which should enhance performance. So I test it with more hidden neurons(500 hidden neurons). Dropout rate is set to be 30% , 50% and 70% empirically. It turns out that the results from these three dropout rates do not differ a lot, with test accuracy ranging from 54% to 56%. And it also shows that, with out L2 regularization, dropout is able to constrain train accuracy not too higher than validation accuracy, which is 73%.

C. Initialization method

I tried 3 different ways of initialization:$\mathrm{N}(0,1)\sqrt{1/n}$, $\mathrm{N}(0,1)\sqrt{2/(n_{in}+n_{out})}$,$\mathrm{N}(0,1)\sqrt{2/n}$. Comparing with the our original initialization: $10^{-4}\mathrm{N}(0,1)$,





some outperform it. The neural network I'm testing on have 500 hidden neurons. I trained it 10000 iterations, 100 batch size, with dropout and momentum.

In our case, we have 3072 input neurons, so $\mathrm{N}(0,1)\sqrt{1/n}$ actually is 3.3×10^{-4}, whose weights are 3 times bigger than our original initialization. It is terrible at first 100 iterations. It reaches loss of 33.161757.

We only 10 output neurons. So $\mathrm{N}(0,1)\sqrt{2/(n_{in}+n_{out})}$ and $\mathrm{N}(0,1)\sqrt{2/n}$ do not differ so much between each other. Although, these two initialization methods' weights are 1.5 times bigger than our original initialization.Loss changing is almost the same as our original initialization. But maybe these methods are better, they are related to the input and output rather than a hand setting value.

D. Activation functions

I've tried 2 activation functions except ReLU: *leaky ReLU* and *tanh*.

Implementation of leaky RelU needs to change a few lines of forward pass and backpropagation. α is set emperically.

```
#forward pass
a2=np.maximum(inp,.01*inp)
#backpropagation
dF=np.ones(np.shape(a2))
dF[a2<0.0]=0.01
```

As is mentioned in research, leaky ReLU may lead to overfitting sometimes. In my test, with the same other hyperparameters, training accuracy of leaky ReLU improved by 4%, however, validation accuracy seems not improved.

As for tanh, with the same other hyperparameters, training time becomes longer, and validation accuracy dropped 2%.

E. Preprocessing

I employ PCAwhitening and K-means to do data preprocessing. It turns out that these two processes hugely improve test accuracy. Without special tuning, test accuracy could easily reach 65%. By intuition, PCAwhithening reduce effects of bright and contract, and K-means put images into different classes before we truely begin training.

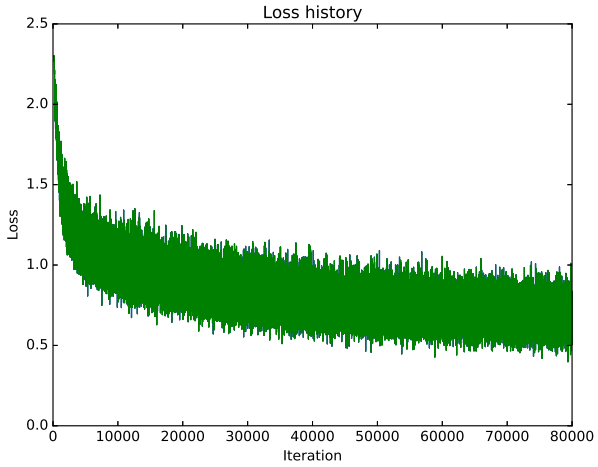
I use np.lingalg to implement PCAwhitening as follow:

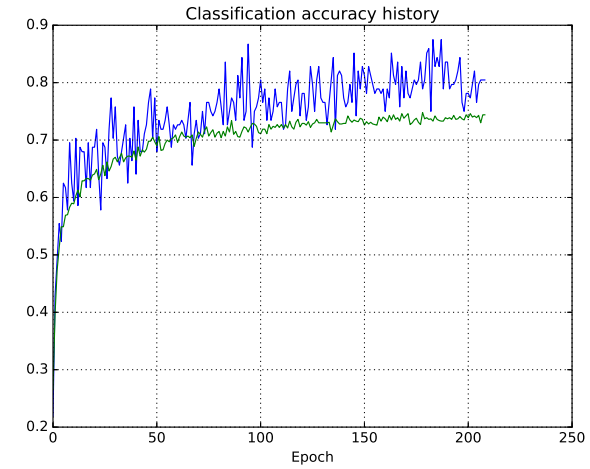
```
[D,V]=np.linalg.eig(
  np.cov(img,rowvar=0))
P = V.dot(
  np.diag(np.sqrt(1/(D + 0.1))))
    .dot(V.T)
img = img.dot(P)
```

Intuitively, Kmeans clusters different images into different clusters, which is much easier for our neural net to handle. I select 1600 centroids emperically from previous work. Also I encountered *MemoryError* when implementing Kmeans, because our data can not be clustered at once. So I changed the processing way a little bit:

```
for each iterations:
  for each batch in data:
    cluster batch
    cumulate result
  renew centroids
```

Finally, with these PCAwhitening and K-means, my best test accuracy successfully reaches **74%**. The following two graphs shows loss history and training, validation accuracy of the selected best hyperparameters. More details of hyperparameters is shown in the next section.





Due to our time limitation, I choose to use 200 hidden nodes. I guess it will work better with more hidden nodes, and longer training time.

IV. Results

As mentioned in II Model Building, I employed a grid search to find best hyperparameters. In order to get better result, I tuned number of iterations, finally fix it to 12000, with a batch size of 100. It increases the Validation accuracy 5%.

The following table shows my three methods accuracy:

TABLE III
DIFFERENCES BETWEEN UPDATE METHODS

	Naive	Dropout	Preprocessed
hidden nodes	350	500	300
learning rate	1×10^{-3}	1×10^{-4}	5×10^{-4}
learning rate Decay	.95	.95	.99
regularization	L2,0.05	Dropout,.5	Dropout,.3
Activation	ReLU	Leaky ReLU	ReLU
Update method	SGD	Momentum,0.9	Momentum,0.95
Iterations	1×10^4	1×10^4	4×10^4
Batch size	100	100	128
Time(min)	15	80	110
Train accuracy	60%	.458	.80
Validation	55%	.458	.75
Test	51.6%	.55	.74

V. Challenges

I encountered many challenges through this machine problem

When I'm doing gradient check. I spent great amount of time finding bugs in my code, but still not able to figure out what's wrong with my code. Then, instead of directly comparing gradient errors, I cut gradient check procedure into pieces. Check derivatives sequentially: $\frac{\partial H}{\partial a^{(3)}}$,$\delta^{(3)}$, $\frac{\partial H}{\partial w^{(2)}}$,$\frac{\partial H}{\partial b^{(2)}}$,$\delta^{(2)}$...

Tuning hyperparameters is really time consuming, I run my program on both on my laptop and CSIL lab. And create multi threads to speed up. I save training infomations like hyperparameters and accuracy to *CSV* files. View them when the program finish. It saves me a lot of time and avoids hand tuning.

Some parts of my code encountered *MemoryError*, because operation like matrix multiplications. I tried iteration way and batch way of matrix multiplications. Both solve *MemoryError*, but batch way turns out to be much faster.

VI. Possible Improvements

There are some other update methods(Adam, Adagrad, etc) I haven't tried. Maybe they are better than Momentum and RMSprop.

More complexed activation function like PReLU, maybe work better than ReLU and Leaky ReLU.

L1 regularization in some situation, maybe better than L2.

As for preprocessing, applying kernel trick may improve results.

With longer training time, it is possible to reach higher accuracy.