EE 569: Homework #5

Issue: 3/17/2019 Due: 11:59PM 04/07/2019

Name: Wenjun Li
USC ID: 8372-7611-20
Email: wenjunli@usc.edu
Submission Date: 04/07/2019

Problem: CNN Training and Its Application to the MNIST Dataset (100%)

(a) CNN Architecture and Training (40%)

Discussion

1. Discuss CNN components in your own words:
   1) The fully connected layer

      Fully connected layer means all the neurons in current layer is connected to every neuron in the previous layer and next layer, i.e. neurons have full connection between fully connected layers. Please see the below demonstration figure. Let's take the 3rd layer as an example, each neuron in 3rd layer connects to all the neurons in 2nd layer and 4th layer.

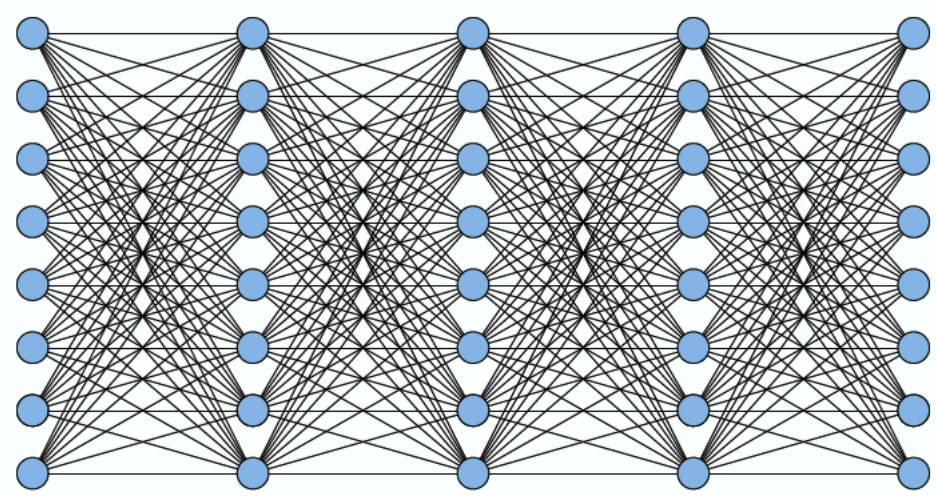


Figure 1 Fully Connected Layer

   2) The convolutional layer

      Convolutional layer is the layer that do convolution operation to the input image and pass the value to next layer. To understand convolutional layer, we first need to understand what is image convolution operation.

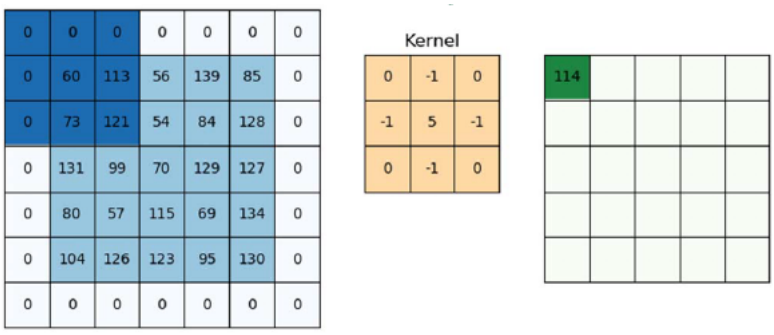


Figure 2 Convolutional Operation to Image

To help understand the operation, I will explain it based on the above Figure 2. As you may see, there is a kernel matrix, which usually has the size of 3*3, 5*5 or larger. The convolution operation is to apply this kernel matrix to each pixel in the input image by doing weight sum. Suppose we

have a matrix from input image, which is $\begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \end{bmatrix}$ and a kernel, which is $\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}$. Then, the convolution operation is to assign the center pixel 'e' with the following value

$$center\ pixel\ value = a + 2b + 3c + 4d + 5e + 6f + 7g + 8h + 9j$$

Normally, there will be multiple kernels to do convolution operation. For example, there are 6 kernels in the 1st convolutional layer in LeNet-5.

3) The max pooling layer

Pooling layer is a layer used between convolutional layers and it can reduce spatial size from the input. Figure 3 is an illustration of max pooling. When we input a 2N*2N matrix, if we use pooling, then the output matrix size is N*N. Actually, there are averaging pooling and other pooling methods, but, max pooling performs the best and the most commonly used one.

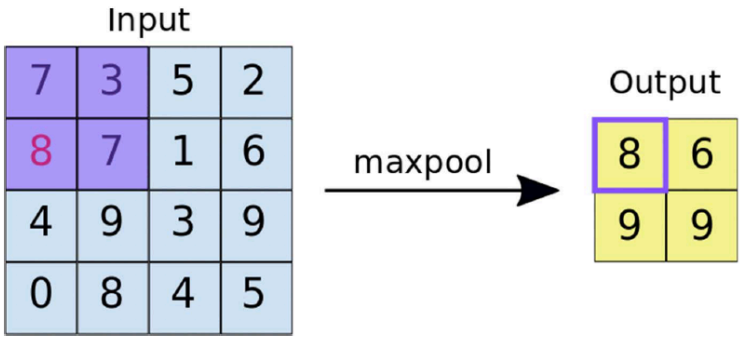


Figure 3 Maxpooling

4) The activation function

Activation layer fellows max pooling layer and it can do non-linear mapping. Take the most commonly used activation function ReLU as an example. When input is larger than 0, the gradient of ReLU is 1; and when input is <0, the gradient of ReLU is 0. So, why we need activation function? It is because activation function can add some non-linearity to the system, which help to build a complex network. If there is no activation, then the network is linear and it cannot be used to analyze complex problems.

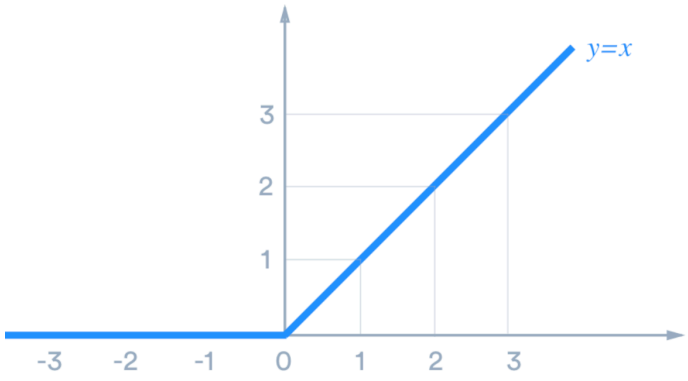


Figure 4 ReLU

There are other some common activation functions, such as sigmoid, tanh, eLU, leaky LU, etc. Activation function is a hot research area.

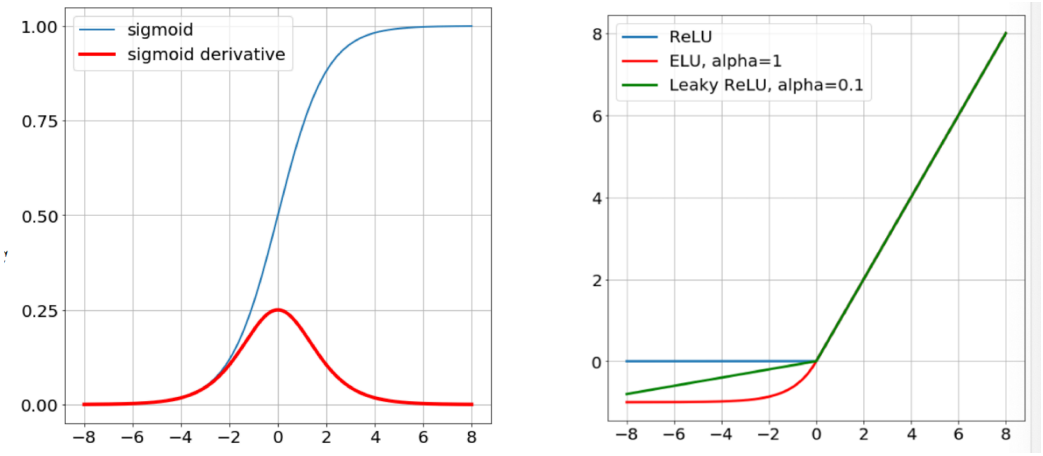


Figure 5 Some Other Activation Functions

5) The softmax function

Softmax function calculates the probability of each classification and predict which classification is the best. When the final output of fully connected layer is s = $\begin{bmatrix} s1 \\ s2 \\ s3 \\ \dots \\ sn \end{bmatrix}$, we need to find out the probability corresponding to each s. Instead of doing hard decision (only predict one '1' and all other to be '0'), we want to do soft decision, which can show the probability of each input dimension. Below is the equation of softmax, where a is the output probability vector.

$$\boldsymbol{a} = \begin{bmatrix} \frac{e^{s_1}}{\sum_{i=1}^{N} e^{s_i}} \\ \vdots \\ \frac{e^{s_N}}{\sum_{i=1}^{N} e^{s_i}} \end{bmatrix}$$

Figure 6 Softmax

2. What's the over-fitting issue in model learning? Explain any technique that has been used in CNN training to avoid the over-fitting.

In short, over-fitting is using a model that is too complex to fit a simple dataset. Overfitting means our trained model can achieve a good result on training data, but, it has bad performance on testing data and real-world data. See the below example. The data is a 2nd order distribution. We can use a higher order model to fit the training data. Let's say, we use 9th order, and this high-order model do perform good on training data. But, when we use it to testing data, that is new (new means the data the model has never seen before) 2nd order data, it will perform bad because this new data can be very different from the training data and the new data probably cannot be well-represented by a 9th order model.

So, why overfitting will happen? Overfitting happens when the number parameters of the network is much larger than the number of training data.

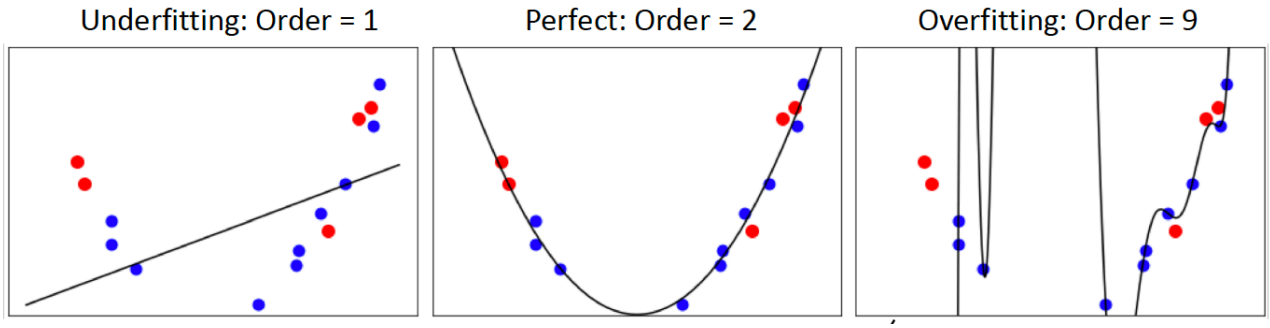


Figure 7 Over Fitting

How to prevent overfitting? There are some effective ways to avoid overfitting, such as early stopping, dropout, data augmentation and etc.

Early stopping: early stopping means we stop training when the model achieves a relatively good performance. For example, although the train the model by 50 epochs can reach a 99% training accuracy, we stop to train the model when we only train it 30 epochs and it reaches 95% accuracy. Since we train the model less, the model will have simple parameters and will be less likely to be overfitting.

Dropout: when we set a dropout rate for conv layer or fully connected layer, is means a certain percentage of parameters will not be updated in a backpropagation. We know that we parameters are updated (trained) by backpropagation and each parameter will be updated if we do not use drop out. So, in a long iteration of training, the parameters will be updated too many times, and model becomes more and more complex. By setting a dropout rate, the backpropagation will not update some of the parameter in one iteration. Since the iteration number is large enough, all of the parameters will be trained for enough time and the model will perform well. For example, when the dropout is 0.5, then it means we do update half of the parameters in one iteration (we cut the connection of these dropout neurons and only update other neurons).

Data augmentation: from the reason why overfitting will happen we know that the training data is too less. So, a straight forward way to solve this is to do data augmentation and generate more data. For example in computer vision area, there are several ways we can do data augmentation: first we can generate a new image by flipping the original image from left to right; second, we can crop the original image and then reshape it to other size; …

3. Why CNNs work much better than other traditional methods in many computer vision problems? You can use the image classification problem as an example to elaborate your points.
Because CNN can self-learn features and update its filters through massive computation. Before CNN shows up, researchers and people need to find filters by handcrafting. When the filter is simple, then people probably can do it. For example, people find laws filters and these filters are good at detecting some simple patterns like spot, wave and etc. However, when the task becomes more complex, for example, how to use filters to extract features of a cat, handcrafting detection filters are impossible for human. This kind of high-level filters are probably not one dimensional, i.e. we need to apply several filters in a sequence to extract features. Finding high-level filters and complex models need massive amount of trails and human cannot do that. Only computer can do this when it can self-learn features and find suitable filters automatically. Here comes CNN. By doing backpropagation, CNN can update its filters automatically and finally find good filters to extract features through massive computation.

4. Explain the loss function and classical backpropagation (BP) optimization procedure to train such a convolutional neutral network.

In classification problem, we use cross-entropy. The cross-entropy cost equation is as below:

$$C = -\sum_{I=1}^{N} y_i \, lna_i$$

where y_i are the ground truth labels and a_i are the network outputs. Minimizing cross-entropy is the same as minimizing $KL-$ divergence between the probability distributions y and a.

For backpropagation, we are doing SGD on all trainable parameters. BP is an algorithm for computing all these from network output and propagate backward to the input layer using the chain rule. Let's have an example in scalar case since it is easier to show and explain.

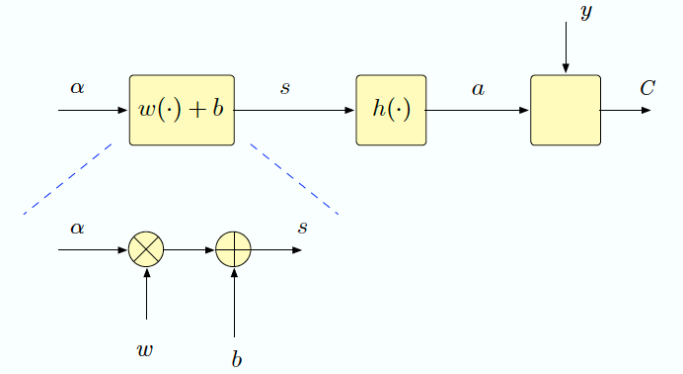


Figure 8 BP Structure

Figure 8 shows a simple layer structure of network, where w is weight and b is bias, w(~)+b computes the intermediate output 's' using input ' α ', h(~) is the activation function, a is the output after activation, y is the ground truth labels and C is the loss function. From SGD, we need to compute $\frac{\partial C}{\partial w}$ and $\frac{\partial C}{\partial b}$ so that we can do SGD as

$$\begin{cases} w = w - \eta \dfrac{\partial C}{\partial w} \\ b = b - \eta \dfrac{\partial C}{\partial b} \end{cases}$$

Note that $\frac{\partial s}{\partial w} = \alpha$ and $\frac{\partial s}{\partial b} = 1$, so we can find $\frac{\partial C}{\partial s}$ using chain rule and then convert it to the desired partials easily.

$$\begin{cases} \dfrac{\partial C}{\partial w} = \dfrac{\partial C}{\partial s}\dfrac{\partial s}{\partial w} = \alpha \dfrac{\partial C}{\partial s} \\ \dfrac{\partial C}{\partial b} = \dfrac{\partial C}{\partial s}\dfrac{\partial s}{\partial b} = \dfrac{\partial C}{\partial s} \end{cases}$$

Continue to simply our equation, we have $\frac{\partial C}{\partial s} = \frac{\partial C}{\partial a}\frac{\partial a}{\partial s} = \frac{\partial C}{\partial s}\frac{\partial h(s)}{\partial s} = C'(a)h'(s)$, $C'(a) = \frac{\partial C}{\partial a}$ and $h'(s) = \frac{dh(s)}{\partial s}$. Finally, we find $\frac{\partial C}{\partial w}$ and $\frac{\partial C}{\partial b}$.

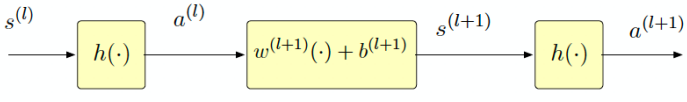


Figure 9 Scalar Example of BP

Figure 9 shows 2 layers, where 'l' is the layer.

Next, let denote $\frac{\partial C}{\partial s}$ of the layer 'l' as δ^l and we can express $\frac{\partial C}{\partial w^l} = \delta^l \, a^{(l-1)}$ and $\frac{\partial C}{\partial b^l} = \delta^l$. Now, we have the complete form as below:

$$\begin{cases} w^l = w^l - \eta \delta^l \, a^{(l-1)} \\ b^l = b^l - \eta \, \delta^l \end{cases}$$

Above example is the scalar explanation of BP, actually, in CNN, we are dealing with matrix BP and it is quite similar. Here, I skip the matrix BP notation and explanation since it's too long and similar to scalar BP.

Above BP equation are used in fully connected layers, for CNN, we also need BP for convolutional layer and pooling layer. Below is the forward convolution operation and gradient computation.

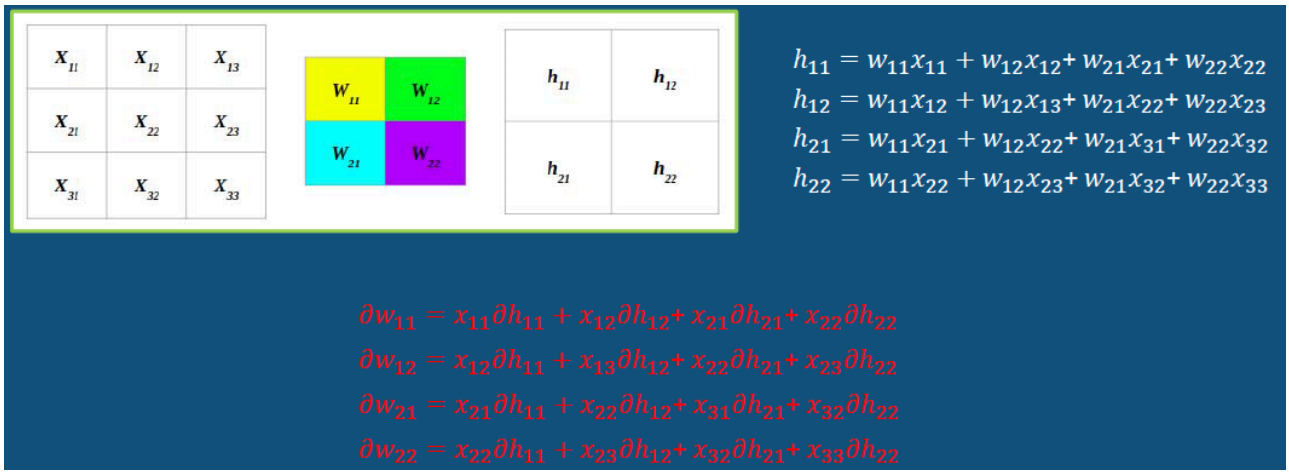


Figure 10 Forward Propagation and Gradient

After we have the gradient of the kernel weights, we can do conv layer BP by doing a flipped convolution. Figure 11 shows how to do this 'flipped' convolution. The original conv operation is from left to right, i.e. from input matrix to activation layer, now, it's flipped, i.e. from activation layer to input matrix. Below is the equation of how to do this flipped convolution operation.

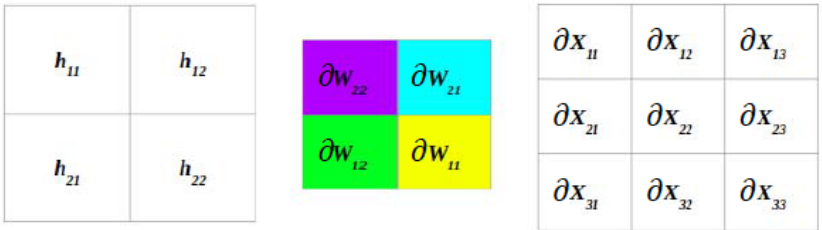


Figure 11 Conv Layer BP

$$\begin{cases} \partial W_{11} = X_{11}\partial h_{11} + X_{12}\partial h_{12} + X_{21}\partial h_{21} + X_{22}\partial h_{22} \\ \partial W_{12} = X_{12}\partial h_{11} + X_{13}\partial h_{12} + X_{22}\partial h_{21} + X_{23}\partial h_{22} \\ \partial W_{21} = X_{21}\partial h_{11} + X_{22}\partial h_{12} + X_{31}\partial h_{21} + X_{32}\partial h_{22} \\ \partial W_{22} = X_{22}\partial h_{11} + X_{23}\partial h_{12} + X_{32}\partial h_{21} + X_{33}\partial h_{22} \end{cases}$$

For pooling layer, the BP operation is shown in the Figure 12. Basically, for max pooling, we assign all δ to the winning input. And for averaging pooling, we spread δ over all inputs.

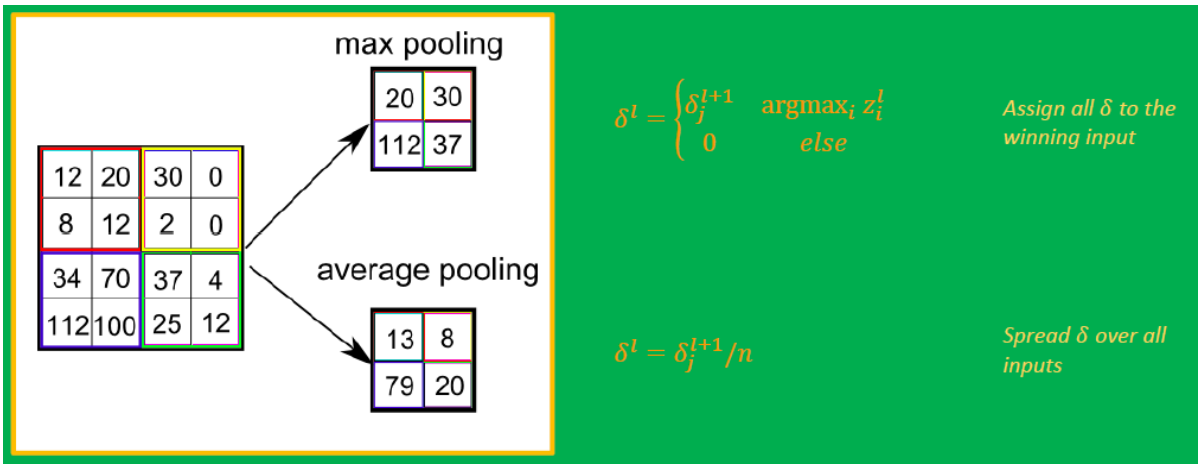


Figure 12 Pooling Layer BP

All in all, above explanation and procedure are BP in CNN.

(b) Train LeNet-5 on MNIST Dataset (30%)

1. Motivation

CNN can learn feature from training images through massive computation. Nowadays, there are many CNN structures, but, which one is the first CNN structure for processing images?

The first CNN to process images is LeNet-5, proposed by Le Cun in 1998[2]. Please see next part for detailed LeNet-5 structure.

2. Approach

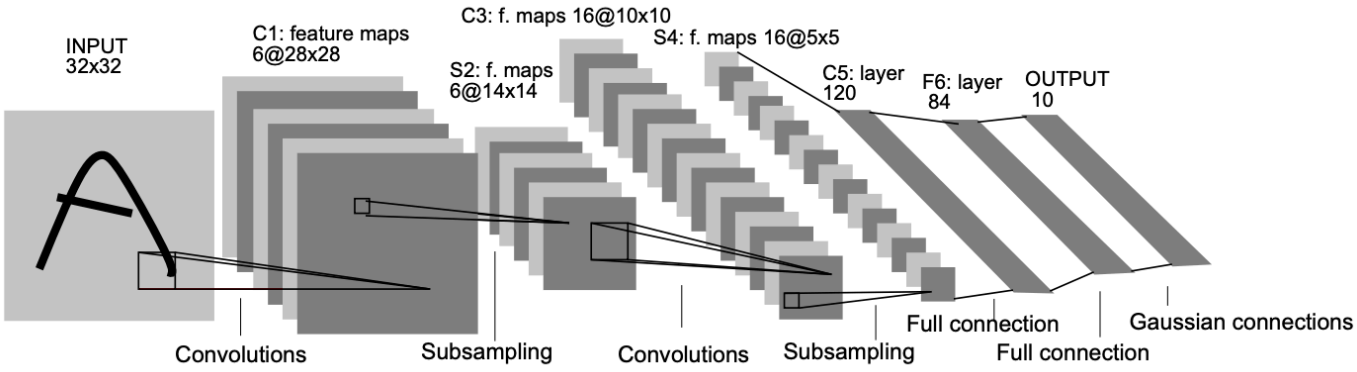


Figure 13 LeNet-5 Illustration

Figure 13 shows the detailed structure of LeNet-5. It consists of 2 convolution layers, 2 max pooling layers, and 3 fully connected layers. There are six 5*5 kernels used in the first convolution layer and 16 5*5 kernels used in the second convolution layer. There are 120, 84, 10 neurons in the three fully connected layers respectively. The input image size is 32*32, and the output is a vector with 10 digits. LeNet-5 can be used to classify MNIST dataset, and its performance if quite good. Please see *Result* part for performance detail.

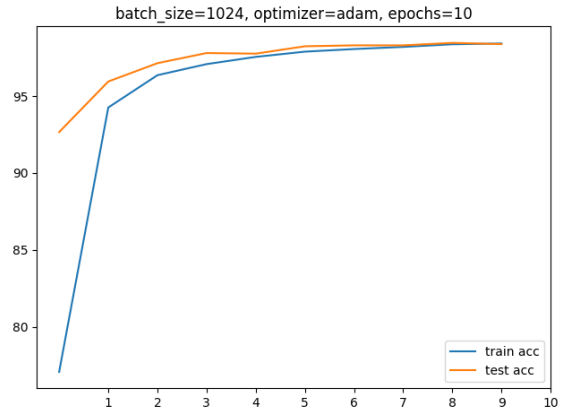
3. Result

Here I show 5 parameter settings. The left picture is training log and the right picture is performance cureve.

- 1st parameter settings: batch_size=1024, optimizer='adam', epochs=10
  The final training acc is 98.42% and testing acc is 98.38%.

Training Log	Acc Performance Curve

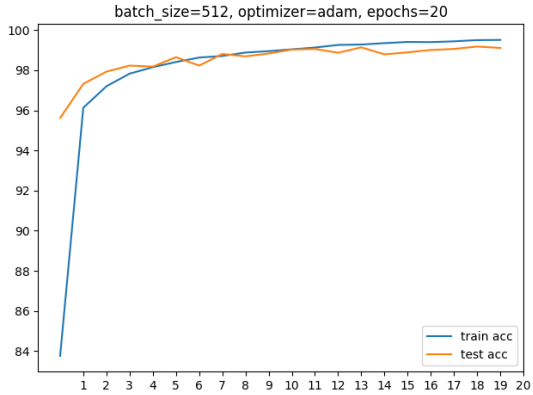
```
(tensorflow_p36) ubuntu@ip-172-31-44-168:~/ee569/hw5$ python p1.py
Using TensorFlow backend.
Downloading data from https://s3.amazonaws.com/img-datasets/mnist.npz
11493376/11490434 [==============================] - 0s 0us/step
Train on 60000 samples, validate on 10000 samples
Epoch 1/10
2019-03-27 23:04:25.566719: I tensorflow/core/platform/cpu_feature_guard.cc:141] Your CPU supports instructions that this TensorFl
2019-03-27 23:04:25.567561: I tensorflow/core/common_runtime/process_util.cc:69] Creating new thread pool with default inter op se
60000/60000 [==============================] - 8s 128us/step - loss: 0.9095 - acc: 0.7707 - val_loss: 0.2572 - val_acc: 0.9266
Epoch 2/10
60000/60000 [==============================] - 7s 111us/step - loss: 0.1976 - acc: 0.9426 - val_loss: 0.1367 - val_acc: 0.9595
Epoch 3/10
60000/60000 [==============================] - 7s 111us/step - loss: 0.1249 - acc: 0.9636 - val_loss: 0.0924 - val_acc: 0.9714
Epoch 4/10
60000/60000 [==============================] - 7s 111us/step - loss: 0.0959 - acc: 0.9708 - val_loss: 0.0723 - val_acc: 0.9780
Epoch 5/10
60000/60000 [==============================] - 7s 111us/step - loss: 0.0817 - acc: 0.9755 - val_loss: 0.0672 - val_acc: 0.9776
Epoch 6/10
60000/60000 [==============================] - 7s 111us/step - loss: 0.0694 - acc: 0.9789 - val_loss: 0.0536 - val_acc: 0.9824
Epoch 7/10
60000/60000 [==============================] - 7s 111us/step - loss: 0.0627 - acc: 0.9806 - val_loss: 0.0514 - val_acc: 0.9830
Epoch 8/10
60000/60000 [==============================] - 7s 111us/step - loss: 0.0591 - acc: 0.9820 - val_loss: 0.0557 - val_acc: 0.9830
Epoch 9/10
60000/60000 [==============================] - 7s 111us/step - loss: 0.0545 - acc: 0.9837 - val_loss: 0.0511 - val_acc: 0.9846
Epoch 10/10
60000/60000 [==============================] - 7s 111us/step - loss: 0.0520 - acc: 0.9842 - val_loss: 0.0492 - val_acc: 0.9838
finish
```



- 2nd parameter settings: batch_size=512, optimizer='adam', epochs=20
  The final training acc is 99.51% and testing acc is 99.11%.

Training Log	Acc Performance Curve

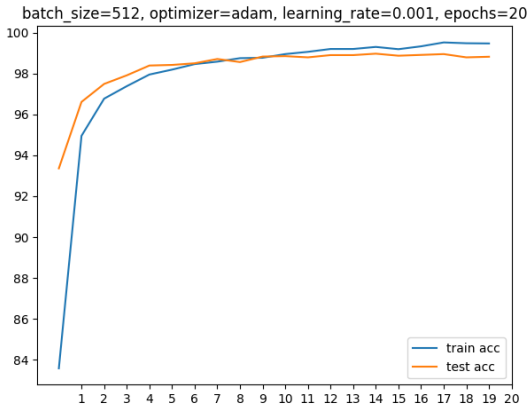
```
(tensorflow_p36) ubuntu@ip-172-31-44-168:~/ee569/hw5$ python p1.py
Using TensorFlow backend.
Train on 60000 samples, validate on 10000 samples
Epoch 1/20
2019-03-28 04:52:20.908654: I tensorflow/core/platform/cpu_feature_guard.cc:141] Your CPU supports instructions that this Tensor
2019-03-28 04:52:20.918160: I tensorflow/core/common_runtime/process_util.cc:69] Creating new thread pool with default inter op
 performance.
60000/60000 [==============================] - 8s 133us/step - loss: 0.5997 - acc: 0.8376 - val_loss: 0.1511 - val_acc: 0.9562
Epoch 2/20
60000/60000 [==============================] - 7s 114us/step - loss: 0.1314 - acc: 0.9613 - val_loss: 0.0855 - val_acc: 0.9732
Epoch 3/20
60000/60000 [==============================] - 7s 114us/step - loss: 0.0914 - acc: 0.9720 - val_loss: 0.0638 - val_acc: 0.9793
Epoch 4/20
60000/60000 [==============================] - 7s 115us/step - loss: 0.0722 - acc: 0.9783 - val_loss: 0.0536 - val_acc: 0.9823
Epoch 5/20
60000/60000 [==============================] - 7s 115us/step - loss: 0.0614 - acc: 0.9815 - val_loss: 0.0537 - val_acc: 0.9818
Epoch 6/20
60000/60000 [==============================] - 7s 115us/step - loss: 0.0533 - acc: 0.9841 - val_loss: 0.0410 - val_acc: 0.9865
Epoch 7/20
60000/60000 [==============================] - 7s 114us/step - loss: 0.0463 - acc: 0.9863 - val_loss: 0.0521 - val_acc: 0.9823
Epoch 8/20
60000/60000 [==============================] - 7s 115us/step - loss: 0.0415 - acc: 0.9871 - val_loss: 0.0357 - val_acc: 0.9881
Epoch 9/20
60000/60000 [==============================] - 7s 115us/step - loss: 0.0380 - acc: 0.9888 - val_loss: 0.0385 - val_acc: 0.9869
Epoch 10/20
60000/60000 [==============================] - 7s 114us/step - loss: 0.0335 - acc: 0.9895 - val_loss: 0.0338 - val_acc: 0.9883
Epoch 11/20
60000/60000 [==============================] - 7s 115us/step - loss: 0.0320 - acc: 0.9904 - val_loss: 0.0318 - val_acc: 0.9903
Epoch 12/20
60000/60000 [==============================] - 7s 114us/step - loss: 0.0283 - acc: 0.9913 - val_loss: 0.0304 - val_acc: 0.9906
Epoch 13/20
60000/60000 [==============================] - 7s 115us/step - loss: 0.0246 - acc: 0.9926 - val_loss: 0.0325 - val_acc: 0.9887
Epoch 14/20
60000/60000 [==============================] - 7s 114us/step - loss: 0.0233 - acc: 0.9928 - val_loss: 0.0278 - val_acc: 0.9914
Epoch 15/20
60000/60000 [==============================] - 7s 114us/step - loss: 0.0224 - acc: 0.9935 - val_loss: 0.0350 - val_acc: 0.9879
Epoch 16/20
60000/60000 [==============================] - 7s 115us/step - loss: 0.0194 - acc: 0.9941 - val_loss: 0.0355 - val_acc: 0.9889
Epoch 17/20
60000/60000 [==============================] - 7s 114us/step - loss: 0.0189 - acc: 0.9940 - val_loss: 0.0301 - val_acc: 0.9901
Epoch 18/20
60000/60000 [==============================] - 7s 114us/step - loss: 0.0166 - acc: 0.9944 - val_loss: 0.0294 - val_acc: 0.9906
Epoch 19/20
60000/60000 [==============================] - 7s 114us/step - loss: 0.0164 - acc: 0.9950 - val_loss: 0.0265 - val_acc: 0.9918
Epoch 20/20
60000/60000 [==============================] - 7s 114us/step - loss: 0.0147 - acc: 0.9951 - val_loss: 0.0314 - val_acc: 0.9911
finish
```



- 3rd parameter settings: batch_size=512, optimizer='adam', learn_rate=0.001, epochs=20
  The final training acc is 99.47% and testing acc is 98.82%.

Training Log	Acc Performance Curve

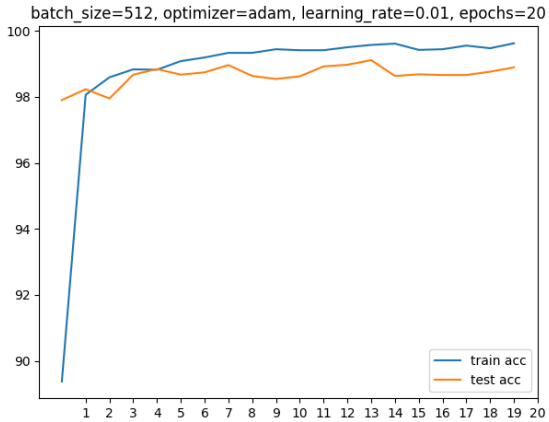
```
(tensorflow_p36) ubuntu@ip-172-31-44-168:~/ee569/hw5$ python p1.py
Using TensorFlow backend.
Train on 60000 samples, validate on 10000 samples
Epoch 1/20
2019-03-28 05:39:40.020620: I tensorflow/core/platform/cpu_feature_guard.cc:141] Your CPU supports instructions that this Tens
2019-03-28 05:39:40.021468: I tensorflow/core/common_runtime/process_util.cc:69] Creating new thread pool with default inter o
 performance.
60000/60000 [==============================] - 8s 129us/step - loss: 0.5923 - acc: 0.8358 - val_loss: 0.2207 - val_acc: 0.9336
Epoch 2/20
60000/60000 [==============================] - 7s 113us/step - loss: 0.1682 - acc: 0.9495 - val_loss: 0.1062 - val_acc: 0.9661
Epoch 3/20
60000/60000 [==============================] - 7s 113us/step - loss: 0.1059 - acc: 0.9677 - val_loss: 0.0826 - val_acc: 0.9749
Epoch 4/20
60000/60000 [==============================] - 7s 114us/step - loss: 0.0847 - acc: 0.9738 - val_loss: 0.0639 - val_acc: 0.9791
Epoch 5/20
60000/60000 [==============================] - 7s 114us/step - loss: 0.0664 - acc: 0.9795 - val_loss: 0.0503 - val_acc: 0.9839
Epoch 6/20
60000/60000 [==============================] - 7s 113us/step - loss: 0.0583 - acc: 0.9819 - val_loss: 0.0470 - val_acc: 0.9842
Epoch 7/20
60000/60000 [==============================] - 7s 114us/step - loss: 0.0497 - acc: 0.9846 - val_loss: 0.0441 - val_acc: 0.9850
Epoch 8/20
60000/60000 [==============================] - 7s 114us/step - loss: 0.0448 - acc: 0.9858 - val_loss: 0.0385 - val_acc: 0.9871
Epoch 9/20
60000/60000 [==============================] - 7s 113us/step - loss: 0.0397 - acc: 0.9875 - val_loss: 0.0439 - val_acc: 0.9856
Epoch 10/20
60000/60000 [==============================] - 7s 113us/step - loss: 0.0381 - acc: 0.9877 - val_loss: 0.0366 - val_acc: 0.9883
Epoch 11/20
60000/60000 [==============================] - 7s 114us/step - loss: 0.0331 - acc: 0.9895 - val_loss: 0.0360 - val_acc: 0.9885
Epoch 12/20
60000/60000 [==============================] - 7s 113us/step - loss: 0.0296 - acc: 0.9906 - val_loss: 0.0376 - val_acc: 0.9879
Epoch 13/20
60000/60000 [==============================] - 7s 114us/step - loss: 0.0266 - acc: 0.9920 - val_loss: 0.0325 - val_acc: 0.9890
Epoch 14/20
60000/60000 [==============================] - 7s 114us/step - loss: 0.0252 - acc: 0.9920 - val_loss: 0.0331 - val_acc: 0.9890
Epoch 15/20
60000/60000 [==============================] - 7s 114us/step - loss: 0.0213 - acc: 0.9930 - val_loss: 0.0317 - val_acc: 0.9897
Epoch 16/20
60000/60000 [==============================] - 7s 114us/step - loss: 0.0233 - acc: 0.9919 - val_loss: 0.0356 - val_acc: 0.9887
Epoch 17/20
60000/60000 [==============================] - 7s 114us/step - loss: 0.0205 - acc: 0.9933 - val_loss: 0.0307 - val_acc: 0.9891
Epoch 18/20
60000/60000 [==============================] - 7s 113us/step - loss: 0.0158 - acc: 0.9952 - val_loss: 0.0315 - val_acc: 0.9895
Epoch 19/20
60000/60000 [==============================] - 7s 114us/step - loss: 0.0162 - acc: 0.9948 - val_loss: 0.0362 - val_acc: 0.9879
Epoch 20/20
60000/60000 [==============================] - 7s 114us/step - loss: 0.0162 - acc: 0.9947 - val_loss: 0.0340 - val_acc: 0.9882
finish
```



- 4th parameter settings: batch_size=512, optimizer='adam', learn_rate=0.01, epochs=20
  The final training acc is 99.62% and testing acc is 98.89%.

Training Log	Acc Performance Curve

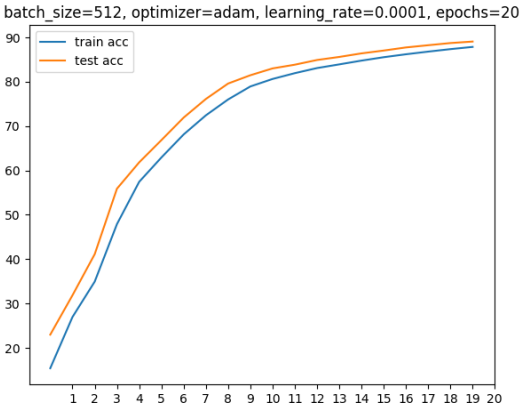
```
(tensorflow_p36) ubuntu@ip-172-31-44-168:~/ee569/hw5$ python p1.py
Using TensorFlow backend.
Train on 60000 samples, validate on 10000 samples
Epoch 1/20
2019-03-28 05:44:09.420412: I tensorflow/core/platform/cpu_feature_guard.cc:141] Your CPU supports instructions that this Tenso
2019-03-28 05:44:09.421204: I tensorflow/core/common_runtime/process_util.cc:69] Creating new thread pool with default inter op
 performance.
60000/60000 [==============================] - 8s 130us/step - loss: 0.3484 - acc: 0.8938 - val_loss: 0.0694 - val_acc: 0.9790
Epoch 2/20
60000/60000 [==============================] - 7s 114us/step - loss: 0.0626 - acc: 0.9806 - val_loss: 0.0575 - val_acc: 0.9823
Epoch 3/20
60000/60000 [==============================] - 7s 115us/step - loss: 0.0445 - acc: 0.9859 - val_loss: 0.0642 - val_acc: 0.9795
Epoch 4/20
60000/60000 [==============================] - 7s 115us/step - loss: 0.0372 - acc: 0.9883 - val_loss: 0.0377 - val_acc: 0.9867
Epoch 5/20
60000/60000 [==============================] - 7s 116us/step - loss: 0.0363 - acc: 0.9882 - val_loss: 0.0374 - val_acc: 0.9884
Epoch 6/20
60000/60000 [==============================] - 7s 115us/step - loss: 0.0292 - acc: 0.9908 - val_loss: 0.0437 - val_acc: 0.9867
Epoch 7/20
60000/60000 [==============================] - 7s 117us/step - loss: 0.0252 - acc: 0.9919 - val_loss: 0.0454 - val_acc: 0.9874
Epoch 8/20
60000/60000 [==============================] - 7s 116us/step - loss: 0.0208 - acc: 0.9933 - val_loss: 0.0365 - val_acc: 0.9896
Epoch 9/20
60000/60000 [==============================] - 7s 116us/step - loss: 0.0204 - acc: 0.9933 - val_loss: 0.0469 - val_acc: 0.9863
Epoch 10/20
60000/60000 [==============================] - 7s 115us/step - loss: 0.0175 - acc: 0.9944 - val_loss: 0.0533 - val_acc: 0.9854
Epoch 11/20
60000/60000 [==============================] - 7s 115us/step - loss: 0.0182 - acc: 0.9941 - val_loss: 0.0543 - val_acc: 0.9862
Epoch 12/20
60000/60000 [==============================] - 7s 115us/step - loss: 0.0184 - acc: 0.9941 - val_loss: 0.0401 - val_acc: 0.9892
Epoch 13/20
60000/60000 [==============================] - 7s 115us/step - loss: 0.0159 - acc: 0.9950 - val_loss: 0.0449 - val_acc: 0.9897
Epoch 14/20
60000/60000 [==============================] - 7s 115us/step - loss: 0.0141 - acc: 0.9957 - val_loss: 0.0341 - val_acc: 0.9911
Epoch 15/20
60000/60000 [==============================] - 7s 115us/step - loss: 0.0127 - acc: 0.9961 - val_loss: 0.0557 - val_acc: 0.9863
Epoch 16/20
60000/60000 [==============================] - 7s 115us/step - loss: 0.0181 - acc: 0.9942 - val_loss: 0.0533 - val_acc: 0.9868
Epoch 17/20
60000/60000 [==============================] - 7s 115us/step - loss: 0.0175 - acc: 0.9944 - val_loss: 0.0531 - val_acc: 0.9866
Epoch 18/20
60000/60000 [==============================] - 7s 115us/step - loss: 0.0149 - acc: 0.9955 - val_loss: 0.0562 - val_acc: 0.9866
Epoch 19/20
60000/60000 [==============================] - 7s 115us/step - loss: 0.0173 - acc: 0.9947 - val_loss: 0.0510 - val_acc: 0.9876
Epoch 20/20
60000/60000 [==============================] - 7s 115us/step - loss: 0.0115 - acc: 0.9962 - val_loss: 0.0501 - val_acc: 0.9889
finish
```



- 5th parameter settings: batch_size=512, optimizer='adam', learn_rate=0. 00001, epochs=20
  The final training acc is 87.86% and testing acc is 89.07%.

Training Log	Acc Performance Curve

```
(tensorflow_p36) ubuntu@ip-172-31-44-168:~/ee569/hw5$ python p1.py
Using TensorFlow backend.
Train on 60000 samples, validate on 10000 samples
Epoch 1/20
2019-03-28 05:51:13.808626: I tensorflow/core/platform/cpu_feature_guard.cc:141] Your CPU supports instructions that this Tenso
2019-03-28 05:51:13.809488: I tensorflow/core/common_runtime/process_util.cc:69] Creating new thread pool with default inter op
 performance.
60000/60000 [==============================] - 8s 132us/step - loss: 2.3108 - acc: 0.1541 - val_loss: 2.2865 - val_acc: 0.2298
Epoch 2/20
60000/60000 [==============================] - 7s 116us/step - loss: 2.2652 - acc: 0.2696 - val_loss: 2.2398 - val_acc: 0.3186
Epoch 3/20
60000/60000 [==============================] - 7s 116us/step - loss: 2.2126 - acc: 0.3494 - val_loss: 2.1753 - val_acc: 0.4113
Epoch 4/20
60000/60000 [==============================] - 7s 116us/step - loss: 2.1349 - acc: 0.4790 - val_loss: 2.0789 - val_acc: 0.5589
Epoch 5/20
60000/60000 [==============================] - 7s 117us/step - loss: 2.0250 - acc: 0.5744 - val_loss: 1.9504 - val_acc: 0.6183
Epoch 6/20
60000/60000 [==============================] - 7s 117us/step - loss: 1.8827 - acc: 0.6293 - val_loss: 1.7871 - val_acc: 0.6684
Epoch 7/20
60000/60000 [==============================] - 7s 116us/step - loss: 1.7080 - acc: 0.6812 - val_loss: 1.5962 - val_acc: 0.7192
Epoch 8/20
60000/60000 [==============================] - 7s 116us/step - loss: 1.5163 - acc: 0.7243 - val_loss: 1.3990 - val_acc: 0.7612
Epoch 9/20
60000/60000 [==============================] - 7s 116us/step - loss: 1.3260 - acc: 0.7601 - val_loss: 1.2120 - val_acc: 0.7960
Epoch 10/20
60000/60000 [==============================] - 7s 116us/step - loss: 1.1518 - acc: 0.7894 - val_loss: 1.0475 - val_acc: 0.8147
Epoch 11/20
60000/60000 [==============================] - 7s 115us/step - loss: 1.0021 - acc: 0.8063 - val_loss: 0.9102 - val_acc: 0.8301
Epoch 12/20
60000/60000 [==============================] - 7s 116us/step - loss: 0.8810 - acc: 0.8194 - val_loss: 0.8030 - val_acc: 0.8385
Epoch 13/20
60000/60000 [==============================] - 7s 115us/step - loss: 0.7854 - acc: 0.8309 - val_loss: 0.7180 - val_acc: 0.8491
Epoch 14/20
60000/60000 [==============================] - 7s 115us/step - loss: 0.7093 - acc: 0.8392 - val_loss: 0.6500 - val_acc: 0.8560
Epoch 15/20
60000/60000 [==============================] - 7s 115us/step - loss: 0.6481 - acc: 0.8477 - val_loss: 0.5956 - val_acc: 0.8640
Epoch 16/20
60000/60000 [==============================] - 7s 115us/step - loss: 0.5979 - acc: 0.8554 - val_loss: 0.5508 - val_acc: 0.8704
Epoch 17/20
60000/60000 [==============================] - 7s 115us/step - loss: 0.5564 - acc: 0.8620 - val_loss: 0.5134 - val_acc: 0.8774
Epoch 18/20
60000/60000 [==============================] - 7s 115us/step - loss: 0.5215 - acc: 0.8680 - val_loss: 0.4821 - val_acc: 0.8824
Epoch 19/20
60000/60000 [==============================] - 7s 115us/step - loss: 0.4918 - acc: 0.8736 - val_loss: 0.4557 - val_acc: 0.8873
Epoch 20/20
60000/60000 [==============================] - 7s 115us/step - loss: 0.4662 - acc: 0.8786 - val_loss: 0.4326 - val_acc: 0.8907
finish
```



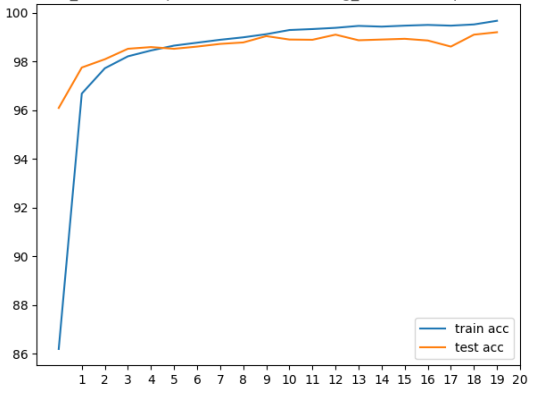
Now, we compute the mean and variance of training accuracy and testing accuracy of above 5 parameter settings.

- Mean of Training Acc: m_train_acc = 96.976%; Variance of Training Acc: var_train_acc = 20.96%
- Mean of Testing Acc: m_test_acc = 96.854%; Variance of Testing Acc: var_test_acc = 15.20%

- The best parameter settings is as below. I chose batch size=256, optimizer=adam, learning rate=0.01 and trained the model for 20 epochs. The final training accuracy is 99.68% and the final test accuracy is 99.21%, which are really good.

Training Log	Acc Performance Curve

```
(tensorflow_p36) ubuntu@ip-172-31-44-168:~/ee569/hw5$ python p1.py
Using TensorFlow backend.
Train on 60000 samples, validate on 10000 samples
Epoch 1/20
2019-03-28 08:10:52.779804: I tensorflow/core/platform/cpu_feature_guard.cc:141] Your CPU supports instructions that this Tensor
2019-03-28 08:10:52.780697: I tensorflow/core/common_runtime/process_util.cc:69] Creating new thread pool with default inter op
 performance.
60000/60000 [==============================] - 9s 147us/step - loss: 0.4660 - acc: 0.8620 - val_loss: 0.1296 - val_acc: 0.9610
Epoch 2/20
60000/60000 [==============================] - 8s 131us/step - loss: 0.1069 - acc: 0.9669 - val_loss: 0.0679 - val_acc: 0.9776
Epoch 3/20
60000/60000 [==============================] - 8s 131us/step - loss: 0.0725 - acc: 0.9773 - val_loss: 0.0594 - val_acc: 0.9810
Epoch 4/20
60000/60000 [==============================] - 8s 132us/step - loss: 0.0583 - acc: 0.9822 - val_loss: 0.0448 - val_acc: 0.9853
Epoch 5/20
60000/60000 [==============================] - 8s 131us/step - loss: 0.0488 - acc: 0.9846 - val_loss: 0.0415 - val_acc: 0.9860
Epoch 6/20
60000/60000 [==============================] - 8s 132us/step - loss: 0.0427 - acc: 0.9866 - val_loss: 0.0453 - val_acc: 0.9853
Epoch 7/20
60000/60000 [==============================] - 8s 132us/step - loss: 0.0376 - acc: 0.9878 - val_loss: 0.0408 - val_acc: 0.9862
Epoch 8/20
60000/60000 [==============================] - 8s 131us/step - loss: 0.0346 - acc: 0.9890 - val_loss: 0.0369 - val_acc: 0.9873
Epoch 9/20
60000/60000 [==============================] - 8s 132us/step - loss: 0.0305 - acc: 0.9900 - val_loss: 0.0346 - val_acc: 0.9879
Epoch 10/20
60000/60000 [==============================] - 8s 132us/step - loss: 0.0264 - acc: 0.9913 - val_loss: 0.0302 - val_acc: 0.9905
Epoch 11/20
60000/60000 [==============================] - 8s 132us/step - loss: 0.0224 - acc: 0.9930 - val_loss: 0.0335 - val_acc: 0.9891
Epoch 12/20
60000/60000 [==============================] - 8s 133us/step - loss: 0.0207 - acc: 0.9934 - val_loss: 0.0311 - val_acc: 0.9890
Epoch 13/20
60000/60000 [==============================] - 8s 132us/step - loss: 0.0194 - acc: 0.9939 - val_loss: 0.0281 - val_acc: 0.9911
Epoch 14/20
60000/60000 [==============================] - 8s 132us/step - loss: 0.0168 - acc: 0.9947 - val_loss: 0.0362 - val_acc: 0.9888
Epoch 15/20
60000/60000 [==============================] - 8s 132us/step - loss: 0.0170 - acc: 0.9944 - val_loss: 0.0372 - val_acc: 0.9891
Epoch 16/20
60000/60000 [==============================] - 8s 131us/step - loss: 0.0152 - acc: 0.9948 - val_loss: 0.0340 - val_acc: 0.9894
Epoch 17/20
60000/60000 [==============================] - 8s 132us/step - loss: 0.0141 - acc: 0.9951 - val_loss: 0.0361 - val_acc: 0.9887
Epoch 18/20
60000/60000 [==============================] - 8s 131us/step - loss: 0.0145 - acc: 0.9948 - val_loss: 0.0475 - val_acc: 0.9862
Epoch 19/20
60000/60000 [==============================] - 8s 131us/step - loss: 0.0145 - acc: 0.9953 - val_loss: 0.0312 - val_acc: 0.9911
Epoch 20/20
60000/60000 [==============================] - 8s 131us/step - loss: 0.0099 - acc: 0.9968 - val_loss: 0.0276 - val_acc: 0.9921
finish
```



4. Discussion

- Learning rate: learning rate determines the speed the model learns. When we set the learning rate too small, as shown in the 5th parameter settings (with a lr=0.00001), the model cannot reach its best performance after training, i.e. it learns too slow. However, when the learning rate is too large, the model cannot converge to its global minimum, which means the model is not stable. It is easy to understand, suppose the learning rate is the step we take every time and the goal is to step for 10 meters. When the step is too small, it will take a long time to reach the finish line; but, when we take a step that is too large, we will wonder around finish line and never really reach the finish line.

  I tried Adam and SGD optimizer for this model. The result is that SGD performs worse than Adam. The learning rate in SGD is invariant with training, which may result in slow training (lr too small) or overshoot (lr too large and miss the global minimum). In contrary, learning rate is reduced with training in Adam, and this help the model reach its best performance quickly and accurately.

- Batch size:
  Batch size is the total number of training examples present in a single batch. Normally, bath size of the powers of 2 work well, i.e. 16, 32, 64, … When we are training on a small dataset (N<10,000), batch size of 32 works well; when we training on a larger dataset, we need a bigger batch size, such as 128, 256, 1024. After my test for different batch size for this LeNet model on MNIST, I found batch size=256 performs best. With a larger batch size, the training will be slow and it cannot reach highest test accuracy; with a smaller batch size, it probably cannot reach the global minimum (best performance).

- Optimizer: I've talked about optimizer above. For SGD, it is the most classical optimizer and it performs worse than Adam or some other optimizers. It is because SGD does not consider variable learning rate and momentum. On the opposite, Adam considers momentum and changeable learning rate, and it performs better.

- Epochs: one epoch is when an entire dataset is passed forward and backward through the network once. It means how many times we've trained our model with the entire dataset. Obviously, larger epochs will lead to better training result, but, larger epochs may also result in over-fitting. When we train our model with the same dataset too many times, the model will become too sensitive and over-fitting. Hence, a suitable epoch number can make the model reach its best performance but not cause over-fitting. Here I choose 20 epochs for my model, because 10 epochs is not enough for the model to reach its best, and >20 epochs is useless in helping improve performance but will cause over-fitting.

(c) Apply Trained Network to Negative Image (30%)

1. Motivation

Although CNN can learn features by itself, usually it performs worse on a new dataset, especially when the new dataset has a new data structure. For example, when CNN is trained on a dataset with black background and white object, this CNN will probably perform bad on a dataset with white background and black object. So, test our CNN model on a negative image dataset is an important criterion to evaluate the CNN model.

2. Approach

A simple way to create a negative image dataset is to do reverse the pixel value, i.e. make black to white

and white to black. We can use the following equation:

$$\mathrm{v(i,j)} = 255 - p(i,j)$$

where the v(i, j) is the new pixel value and p(i, j) is the original pixel value. Human have no difficulty in recognizing a negative image, but, CNN (LeNet) may have problem with this. Let's test the performance of above trained model on a negative image dataset and find out how bad it is.

To test the model on negative dataset, here I only use the testing images in MNIST, i.e. only using the 10,000 images to test. I load the model trained from prob.(b) and convert the 10,000 images to negative images. Then, I evaluate the model with x_test_negative images and y_test labels. The testing performance is shown in *Result* part.

To solve this problem, I propose that we can add negative images to training dataset. The idea is that, since CNN has problem to predict a negative dataset, we can just let it "see" this negative dataset while training it. We can convert every training image in the MNIST dataset to a negative image and then put all the original training images and negative images into one training dataset, i.e. we will have a dataset of 120,000 images, half of them are original training images and half of them are negative images that are derived from original training images.

I trained my model on this combined dataset (combined dataset of original dataset and negative dataset), and do testing on original testing data and negative testing data. The result my proposed method achieved is actually very good. Please check the performance I show in the *Result* part.

(Note that, the parameter settings that I used in this prob.(c) are the same with the best performance model I obtained in prob.(b).

3. Result

The performance result on the negative image dataset is really bad: while the positive dataset (original MNIST test dataset) can reach a testing accuracy at 99.21%, the testing accuracy of negative image dataset is only 27.71%. This testing result shows that CNN have difficulty to predict a new dataset which has different data structure (i.e. negative image dataset).

```
loss = 4.566499
accuracy = 0.277100
```

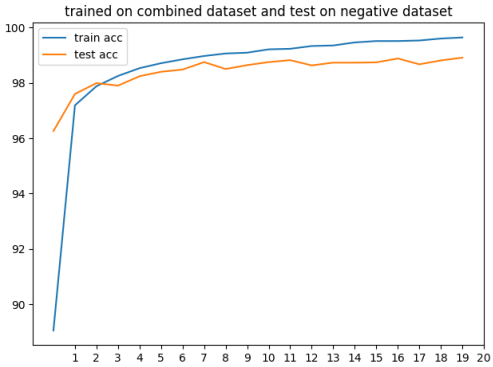
Figure 14 Test Performance of Negative Dataset

Next, I will show the performance of my proposed method. First, I test the model on negative testing data (i.e. the negative images). It reached a training accuracy of 99.64% and a testing accuracy of 98.91%. (Note, the training log figure is obtained by setting (xte_negative, yte) as validation data in training process. The "loss" and "accuracy" is obtained by re-load the model and test on negative data.)

Training Log	Acc Performance Curve

```
(tensorflow_p36) ubuntu@ip-172-31-44-168:~/ee569/hw5$ python prob_c_2.py
Using TensorFlow backend.
Train on 120000 samples, validate on 10000 samples
Epoch 1/20
2019-03-29 19:44:49.154268: I tensorflow/core/platform/cpu_feature_guard.cc:141] Your CPU supports instructions that this TensorFlow
2019-03-29 19:44:49.155075: I tensorflow/core/common_runtime/process_util.cc:69] Creating new thread pool with default inter op set
120000/120000 [==============================] - 16s 133us/step - loss: 0.3684 - acc: 0.8906 - val_loss: 0.1163 - val_acc: 0.9626
Epoch 2/20
120000/120000 [==============================] - 15s 125us/step - loss: 0.0916 - acc: 0.9719 - val_loss: 0.0723 - val_acc: 0.9760
Epoch 3/20
120000/120000 [==============================] - 15s 125us/step - loss: 0.0682 - acc: 0.9788 - val_loss: 0.0629 - val_acc: 0.9799
Epoch 4/20
120000/120000 [==============================] - 15s 125us/step - loss: 0.0561 - acc: 0.9825 - val_loss: 0.0616 - val_acc: 0.9790
Epoch 5/20
120000/120000 [==============================] - 15s 126us/step - loss: 0.0475 - acc: 0.9853 - val_loss: 0.0541 - val_acc: 0.9824
Epoch 6/20
120000/120000 [==============================] - 15s 126us/step - loss: 0.0410 - acc: 0.9871 - val_loss: 0.0500 - val_acc: 0.9840
Epoch 7/20
120000/120000 [==============================] - 15s 126us/step - loss: 0.0369 - acc: 0.9885 - val_loss: 0.0422 - val_acc: 0.9848
Epoch 8/20
120000/120000 [==============================] - 15s 126us/step - loss: 0.0329 - acc: 0.9897 - val_loss: 0.0396 - val_acc: 0.9875
Epoch 9/20
120000/120000 [==============================] - 15s 126us/step - loss: 0.0295 - acc: 0.9906 - val_loss: 0.0448 - val_acc: 0.9850
Epoch 10/20
120000/120000 [==============================] - 15s 127us/step - loss: 0.0271 - acc: 0.9909 - val_loss: 0.0429 - val_acc: 0.9864
Epoch 11/20
120000/120000 [==============================] - 15s 126us/step - loss: 0.0247 - acc: 0.9921 - val_loss: 0.0387 - val_acc: 0.9875
Epoch 12/20
120000/120000 [==============================] - 15s 126us/step - loss: 0.0233 - acc: 0.9923 - val_loss: 0.0387 - val_acc: 0.9882
Epoch 13/20
120000/120000 [==============================] - 15s 126us/step - loss: 0.0204 - acc: 0.9933 - val_loss: 0.0418 - val_acc: 0.9863
Epoch 14/20
120000/120000 [==============================] - 15s 126us/step - loss: 0.0198 - acc: 0.9935 - val_loss: 0.0428 - val_acc: 0.9873
Epoch 15/20
120000/120000 [==============================] - 15s 126us/step - loss: 0.0165 - acc: 0.9946 - val_loss: 0.0422 - val_acc: 0.9873
Epoch 16/20
120000/120000 [==============================] - 15s 126us/step - loss: 0.0156 - acc: 0.9951 - val_loss: 0.0449 - val_acc: 0.9874
Epoch 17/20
120000/120000 [==============================] - 15s 125us/step - loss: 0.0146 - acc: 0.9951 - val_loss: 0.0450 - val_acc: 0.9888
Epoch 18/20
120000/120000 [==============================] - 15s 126us/step - loss: 0.0136 - acc: 0.9953 - val_loss: 0.0490 - val_acc: 0.9867
Epoch 19/20
120000/120000 [==============================] - 15s 126us/step - loss: 0.0120 - acc: 0.9960 - val_loss: 0.0461 - val_acc: 0.9881
Epoch 20/20
120000/120000 [==============================] - 15s 126us/step - loss: 0.0114 - acc: 0.9964 - val_loss: 0.0444 - val_acc: 0.9891
finish
```

```
loss = 0.044397
accuracy = 0.989100
```

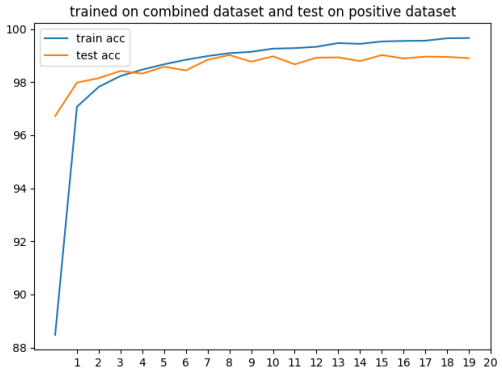


Then, I run the model on positive testing data (i.e. the original MNIST images). It reached a training accuracy of 99.66% and a testing accuracy of 98.90%. (Note, the training log figure is obtained by setting (xte_positive, yte) as validation data in training process. The "loss" and "accuracy" is obtained by re-load the model and test on positve data.)

Training Log	Acc Performance Curve

```
(tensorflow_p36) ubuntu@ip-172-31-44-168:~/ee569/hw5$ python prob_c_2.py
Using TensorFlow backend.
Train on 120000 samples, validate on 10000 samples
Epoch 1/20
2019-03-29 19:58:45.597363: I tensorflow/core/platform/cpu_feature_guard.cc:141] Your CPU supports instructions that this TensorFl
2019-03-29 19:58:45.598231: I tensorflow/core/common_runtime/process_util.cc:69] Creating new thread pool with default inter op se
120000/120000 [==============================] - 17s 139us/step - loss: 0.3794 - acc: 0.8848 - val_loss: 0.1015 - val_acc: 0.9672
Epoch 2/20
120000/120000 [==============================] - 16s 131us/step - loss: 0.0934 - acc: 0.9707 - val_loss: 0.0677 - val_acc: 0.9798
Epoch 3/20
120000/120000 [==============================] - 16s 131us/step - loss: 0.0688 - acc: 0.9782 - val_loss: 0.0610 - val_acc: 0.9815
Epoch 4/20
120000/120000 [==============================] - 16s 131us/step - loss: 0.0557 - acc: 0.9823 - val_loss: 0.0471 - val_acc: 0.9842
Epoch 5/20
120000/120000 [==============================] - 16s 131us/step - loss: 0.0476 - acc: 0.9847 - val_loss: 0.0505 - val_acc: 0.9832
Epoch 6/20
120000/120000 [==============================] - 16s 132us/step - loss: 0.0417 - acc: 0.9867 - val_loss: 0.0434 - val_acc: 0.9858
Epoch 7/20
120000/120000 [==============================] - 16s 131us/step - loss: 0.0362 - acc: 0.9884 - val_loss: 0.0486 - val_acc: 0.9844
Epoch 8/20
120000/120000 [==============================] - 16s 132us/step - loss: 0.0316 - acc: 0.9897 - val_loss: 0.0342 - val_acc: 0.9884
Epoch 9/20
120000/120000 [==============================] - 16s 132us/step - loss: 0.0283 - acc: 0.9909 - val_loss: 0.0346 - val_acc: 0.9902
Epoch 10/20
120000/120000 [==============================] - 16s 131us/step - loss: 0.0263 - acc: 0.9914 - val_loss: 0.0388 - val_acc: 0.9877
Epoch 11/20
120000/120000 [==============================] - 16s 131us/step - loss: 0.0230 - acc: 0.9926 - val_loss: 0.0336 - val_acc: 0.9897
Epoch 12/20
120000/120000 [==============================] - 16s 131us/step - loss: 0.0211 - acc: 0.9928 - val_loss: 0.0460 - val_acc: 0.9867
Epoch 13/20
120000/120000 [==============================] - 16s 131us/step - loss: 0.0196 - acc: 0.9933 - val_loss: 0.0340 - val_acc: 0.9892
Epoch 14/20
120000/120000 [==============================] - 16s 131us/step - loss: 0.0164 - acc: 0.9947 - val_loss: 0.0369 - val_acc: 0.9893
Epoch 15/20
120000/120000 [==============================] - 16s 130us/step - loss: 0.0165 - acc: 0.9944 - val_loss: 0.0437 - val_acc: 0.9879
Epoch 16/20
120000/120000 [==============================] - 16s 131us/step - loss: 0.0137 - acc: 0.9953 - val_loss: 0.0348 - val_acc: 0.9902
Epoch 17/20
120000/120000 [==============================] - 16s 131us/step - loss: 0.0133 - acc: 0.9955 - val_loss: 0.0406 - val_acc: 0.9889
Epoch 18/20
120000/120000 [==============================] - 16s 130us/step - loss: 0.0128 - acc: 0.9956 - val_loss: 0.0384 - val_acc: 0.9896
Epoch 19/20
120000/120000 [==============================] - 16s 130us/step - loss: 0.0107 - acc: 0.9965 - val_loss: 0.0450 - val_acc: 0.9895
Epoch 20/20
120000/120000 [==============================] - 16s 131us/step - loss: 0.0102 - acc: 0.9966 - val_loss: 0.0420 - val_acc: 0.9890
finish
```

```
loss = 0.041389
accuracy = 0.989000
```



4. Discussion

As you can see from the *Result* part, the model performance on negative dataset has been greatly improved, from 27.71% to 98.91% for testing accuracy. From this we can know that CNN model is very sensitive to dataset structure changes and data changes. When we train the model only on positive dataset (the original 60,000 MNIST training images), the model trained can only be used to predict data that has the same structure, which is black as background and white as object. If the dataset structure is changed, then the performance of the model is bad.

Based on this property, my proposed method adds negative images into the training dataset, which can let CNN model "see" and "learn" both positive images and negative images. Therefore, the performance of the new model has been greatly improved.

One of difficulties in CNN model development and research is actually how to get a "good" training dataset. The "good" means the diversity of the data, the amount of the data, the representative ability of the data, the regularization possibility of the data, and etc. When we have a "good" data, then getting a good model is more likely to happen.

Nowadays, some researchers are also working on how to let CNN conquer this weakness and there are some success, for example Saak transform.

Reference

[1] online: https://becominghuman.ai/back-propagation-in-convolutional-neural-networks-intuition-and-code-714ef1c38199

[2] LeCun Y, Bottou L, Bengio Y, et al. Gradient-based learning applied to document recognition[J]. Proceedings of the IEEE, 1998, 86(11): 2278-2324.