Handwriting Recognition
Neural Network and Deep Learning
FOM=0.0710

By Group BEAR
Ya-Hsuan Chuo (yc3238),
Barbara Lu (xl2627),
Sui Ping Suen (ss5202),
Helen Tao (jt2956)

Catalogue

Introduction

Handwriting recognition is now a heated topic and has been widely used by various industries including banks, postal services, and government. It is a well-studied subject in computer science and in machine learning. In this project, our group is going to analyze the MNIST dataset of handwritten digits which contains 60K training examples and 10K testing examples. Our goal is to recognize handwritten digits and achieve the highest accuracy with the least number of training data. We employ two machine learning techniques, Fully-Connected Deep Neural Network (DNN) and Convolutional Neural Network (CNN). The quality of the results is measured by the figure-of-merit (FOM), which is defined as the sum of half the percentage of training data being used and the testing error rate. By building two models, tuning parameters, training the model with various sizes of training data, and comparing the FOMs, we achieved a smallest FOM of 0.0710 with Convolutional Neural Network Model.

Data Exploration

Data Summary

- MNIST database of handwritten digits (URL: http://yann.lecun.com/exdb/mnist/)
- 60K training data, 10K testing data
- Four files
  - train-images-idx3-ubyte.gz: training set images (9912422 bytes)
  - train-labels-idx1-ubyte.gz: training set labels (28881 bytes)
  - t10k-images-idx3-ubyte.gz: test set images (1648877 bytes)
  - t10k-labels-idx1-ubyte.gz: test set labels (4542 bytes)

Data Processing

First, we load the four data files into the RStudio. The format of these files are not csv, so we use following code to load them.

```
load_image_file <- function(filename) {
 ret = list()
 f = file(filename,'rb')
 readBin(f,'integer',n=1,size=4,endian='big')
 ret$n = readBin(f,'integer',n=1,size=4,endian='big')
 nrow = readBin(f,'integer',n=1,size=4,endian='big')
 ncol = readBin(f,'integer',n=1,size=4,endian='big')
 x = readBin(f,'integer',n=ret$n*nrow*ncol,size=1,signed=F)
 ret$x = matrix(x, ncol=nrow*ncol, byrow=T)
 close(f)
 ret
}
```

```
load_label_file <- function(filename) {
  f = file(filename,'rb')
  readBin(f,'integer',n=1,size=4,endian='big')
  n = readBin(f,'integer',n=1,size=4,endian='big')
  y = readBin(f,'integer',n=n,size=1,signed=F)
  close(f)
  y
}
train_data <- load_image_file("train-images-idx3-ubyte")
test_data <- load_image_file('t10k-images-idx3-ubyte')
train_label <- load_label_file('train-labels-idx1-ubyte')
test_label <- load_label_file('t10k-labels-idx1-ubyte')
```

Next, We process the training dataset and the testing dataset and check the distribution of the training dataset.

```
train <- data.matrix(train_data[[2]])
test <- data.matrix(test_data[[2]])

train <- as.data.frame(train)
train_label <- as.data.frame(train_label)
table(train_label)
```

```
> table(train_label)
train_label
   0    1    2    3    4    5    6    7    8    9
5923 6742 5958 6131 5842 5421 5918 6265 5851 5949
```

Then, we combine the training data image and the training data label, so we can select various percentages of the whole training data and use the sub-training data to train the models.

```
train_all <- cbind(train_label,train)
set.seed(101)
indexes <- sample(1:nrow(train_all), size=0.1*nrow(train_all))
#0.1 is an example of the percentage we select
train_part <- train_all[indexes,]
train_part_x <- train_part[,-1]
train_part_y <- train_part[,1]
train_part_x <- data.matrix(train_part_x)
train_part_y <- as.integer(train_part_y)
```

Finally, we scale the dataset. The grayscale of each image falls in the range [0, 255], so we linearly transform it into [0,1].

```
train.x <- t(train_part_x/255)
test <- t(test/255)
```

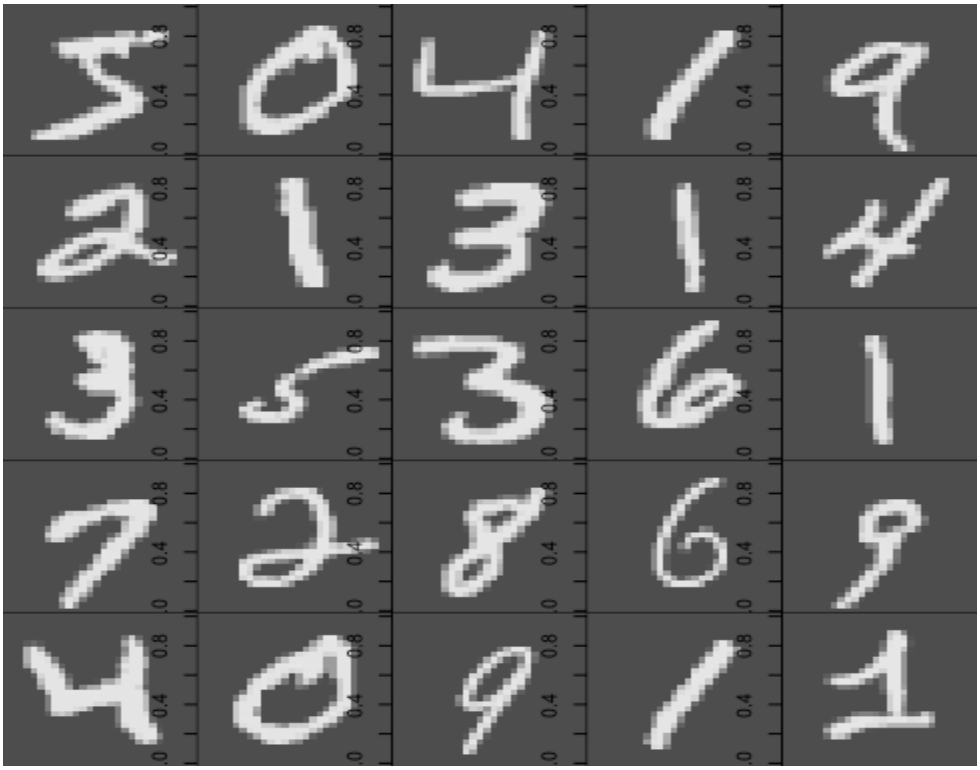
Now, both the training set and testing set are ready to use.

Note: We also researched on artificially generating more training data. However, because the code is very complicated, we didn't employ it to achieve the final result. Thus, we will discuss the topic in the last section of the report, "Data Deformation", and this will be the area we will explore further into.

Data Visualization

Before working on the models, we did a data visualization with the following code.

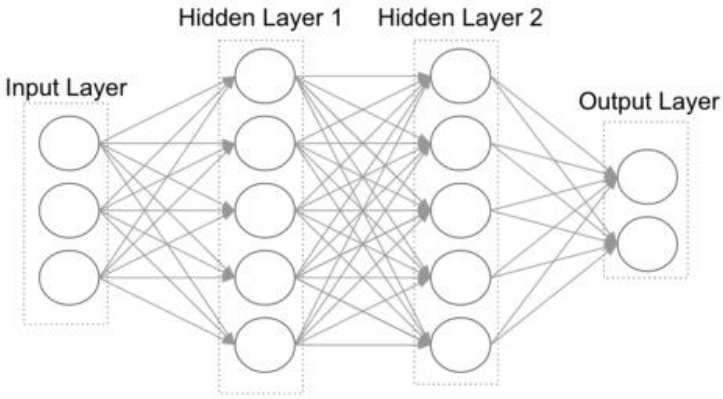
```
par(mfrow=c(5,5),mar=c(0,0,0,0))
flip <- function(x) t(apply(x, 1, rev))
for (i in 1:25)
{
 digit <- flip(matrix(as.numeric(train_data[[2]][i,1:784]),nrow=28))
 image(digit,col=grey.colors(255))
}
```



Fully-Connected Deep Neural Network Model

Model Introduction

The Fully-Connected Deep Neural Network model we use consists of 2 hidden layers of computational units, interconnected in a feed-forward way. Neurons between two adjacent layers are fully pairwise connected, but neurons within a single layer share no connections.
For the activation function, we apply ReLU, which stands for the Rectified Linear Unit. It computes the function f(x)=max(0,x). In other words, the activation is thresholded at zero.
This layered architecture enables very efficient evaluation of Neural Networks based on matrix multiplications interwoven with the application of the activation function.
We have 784 (28*28 pixels) neurons at the input layer and 10 (0 to 9) neurons at the output layer. For the hidden layers, we start with the model introduced by R-bloggers which has 2 hidden layers with 128 neurons and 64 neurons at each layer. We will then modify the number of hidden layers and their neuron sizes to optimize the FOM.



Feedforward neural network with 2 hidden layers

Building Model

We use the MXNet package for this project. The initial model we use is from R-bloggers (https://www.r-bloggers.com/deep-learning-with-mxnetr/). We modify the model in the following section to achieve a lower FOM.

```
#Configure the network
data <- mx.symbol.Variable("data")
fc1 <- mx.symbol.FullyConnected(data, name="fc1", num_hidden=128)
act1 <- mx.symbol.Activation(fc1, name="relu1", act_type="relu")
```

```
fc2 <- mx.symbol.FullyConnected(act1, name="fc2", num_hidden=64)
act2 <- mx.symbol.Activation(fc2, name="relu2", act_type="relu")
fc3 <- mx.symbol.FullyConnected(act2, name="fc3", num_hidden=10)
softmax <- mx.symbol.SoftmaxOutput(fc3, name="sm")
```

1. We use its own data type symbol to configure the network. “Data” represents the input layer.
2. Then we set the first hidden layer by fc1 <- mx.symbol.FullyConnected(data, name="fc1", num_hidden=128). This layer has data as the input, its name and the number of hidden neurons.
3. The activation is set by act1 <- mx.symbol.Activation(fc1, name="relu1", act_type="relu"). The activation function takes the output from the first hidden layer fc1.
4. The second hidden layer takes the result from act1 as the input, with its name as "fc2" and the number of hidden neurons as 64.
5. The second activation is almost the same as act1, except we have a different input source and name.
6. Here comes the output layer. Since there's only 10 digits, we set the number of neurons to 10.
7. Finally we set the activation to softmax to get a probabilistic prediction.

```
#Choose CPU as the device
device.cpu <- mx.cpu()
#Train the neural network
mx.set.seed(0)
model1 <- mx.model.FeedForward.create(softmax, X=train.x, y=train_part_y,
                                      ctx=device.cpu, num.round=10, array.batch.size=100,
                                      learning.rate=0.07, momentum=0.9,  eval.metric=mx.metric.accuracy,
                                      initializer=mx.init.uniform(0.07),
                                      epoch.end.callback=mx.callback.log.train.metric(100))
#Make the prediction 1
preds1 <- predict(model1, test)
pred.label1 <- max.col(t(preds1)) - 1

#Confusion matrix and accuracy
table(pred.label1,test_label)
correct1 <- sum(pred.label1==test_label)
accuracy1 <- correct1/ncol(test)
print(accuracy1)

FOM <- 0.1/2+(1-accuracy1)
print(FOM)
```

Optimizing Model

In this project, our aim is to minimize FOM, so we want both our training data size and test error to be small. Based on this goal, we first take a look at our initial model's test accuracy and FOM by using different training data size, from 10% to 100% of the whole training data.

	Initial Model	
Training data size	Test Accuracy	FOM
10%	0.9326	0.1174
20%	0.958	0.142
30%	0.9638	0.1862
40%	0.9685	0.2315
50%	0.9676	0.2824
60%	0.9662	0.3338
70%	0.9716	0.3784
80%	0.9659	0.4341
90%	0.9762	0.4738
100%	0.9776	0.5224

From the table above, we can see that test accuracy tends to be stable (around 97%) after we use more than 30% of the training data. FOM keeps increasing. When we use less than 20% of training data, the FOM value is relatively small. Therefore, to find out the smallest FOM, we take a look at our test accuracy and FOM again by using 1% to 20% of our training data.

	Initial Model	
Training data size	Test	FOM

	Accuracy	
1%	0.7883	0.2167
2%	0.796	0.214
3%	0.878	0.137
4%	0.8975	0.1225
5%	0.9056	0.1194
6%	0.91	0.12
7%	0.9321	0.1029
8%	0.9312	0.1088
9%	0.9439	0.1011
10%	0.9326	0.1174
11%	0.9345	0.1205
12%	0.9456	0.1144
13%	0.9451	0.1199
14%	0.9501	0.1199
15%	0.952	0.123
16%	0.9473	0.1327
17%	0.9582	0.1268
18%	0.9457	0.1443
19%	0.956	0.139
20%	0.958	0.142

From the table above, we can see that FOM values are relatively small when using 4% to 14% of training data. The minimum FOM occurs at using 9% of training data, which is 0.1011. We are going to tune our model to further minimize the FOM. In the table below, it summarizes our process of tuning parameters for our fully-connected model.

	Initial Model 1	Model 1 4 hidden layers	Model 1 num_hidden=10	Model 1 round.num 20	Model 1 learning.rate=0.1

					24,512,10					
	Test Accuracy	FOM	Test Accuracy	FOM	Test Accuracy	FOM	Test Accuracy	FOM	Test Accuracy	FOM
10%	0.9326	0.1174	0.8765	0.1735	0.9594	0.0906	0.955	0.095	0.9451	0.1049
20%	0.958	0.142	0.9264	0.1736	0.9695	0.1305	0.9661	0.1339	0.9594	0.1406
30%	0.9638	0.1862	0.9465	0.2035	0.975	0.175	0.9701	0.1799	0.9664	0.1836
40%	0.9685	0.2315	0.9587	0.2413	0.9773	0.2227	0.9722	0.2278	0.9699	0.2301
50%	0.9676	0.2824	0.9633	0.2867	0.9774	0.2726	0.9751	0.2749	0.9655	0.2845
60%	0.9662	0.3338	0.9647	0.3353	0.9774	0.3226	0.9723	0.3277	0.9621	0.3379
70%	0.9716	0.3784	0.9674	0.3826	0.9801	03699	0.9755	0.3745	0.9687	0.3813
80%	0.9659	0.4341	0.9724	0.4276	0.9823	0.4177	0.9757	0.4243	0.9696	0.4304
90%	0.9762	0.4738	0.9719	0.4781	0.981	0.469	0.9737	0.4763	0.9694	0.4806
100%	0.9776	0.5224	0.9698	0.5302	0.9859	0.5141	0.979	0.521	0.9771	0.5229

From the table above, we can see that first, we try to add two more hidden layers. We implement this tuning by modifying our initial code as following:

```
fc1 <- mx.symbol.FullyConnected(data, name="fc1", num_hidden=128)
act1 <- mx.symbol.Activation(fc1, name="relu1", act_type="relu")
fc2 <- mx.symbol.FullyConnected(act1, name="fc2", num_hidden=64)
act2 <- mx.symbol.Activation(fc2, name="relu2", act_type="relu")
fc3 <- mx.symbol.FullyConnected(act2, name="fc3", num_hidden=32)
act3 <- mx.symbol.Activation(fc3, name="relu3", act_type="relu")
fc4 <- mx.symbol.FullyConnected(act3, name="fc4", num_hidden=16)
act4 <- mx.symbol.Activation(fc4, name="relu4", act_type="relu")
fc5 <- mx.symbol.FullyConnected(act4, name="fc5", num_hidden=10)
softmax <- mx.symbol.SoftmaxOutput(fc5, name="sm")
```

However, we can see that both test accuracy and FOM are getting worse. Thus, we are not going to add more hidden layers.

Second, we try to increase the size of each hidden layer by increasing the number of neurons. We increase the number of neurons in first hidden layer from 128 to 1024, and we increase the number of neurons in second hidden layer from 64 to 512. We implement this tuning by modifying our initial code as following:

```
fc1 <- mx.symbol.FullyConnected(data, name="fc1", num_hidden=1024)
act1 <- mx.symbol.Activation(fc1, name="relu1", act_type="relu")
fc2 <- mx.symbol.FullyConnected(act1, name="fc2", num_hidden=512)
act2 <- mx.symbol.Activation(fc2, name="relu2", act_type="relu")
```

As a result, we can see that both our test accuracy and FOM get improved by increasing the size of hidden layers.

Third, we increase the number of iterations over training data to train model. Considering the model training duration time, we only double the number of iterations by modifying our initial code as following:

```
model1 <- mx.model.FeedForward.create(softmax, X=train.x, y=train_part_y,
                                      ctx=device.cpu, num.round=20, array.batch.size=100,
                                      learning.rate=0.07, momentum=0.9,  eval.metric=mx.metric.accuracy,
                                      initializer=mx.init.uniform(0.07),
                                      epoch.end.callback=mx.callback.log.train.metric(100))
```

From the table above, we can see that increasing the training iterations definitely improve both test accuracy and FOM values. Last, we tune the learning rate by modifying our initial code as following:

```
model1 <- mx.model.FeedForward.create(softmax, X=train.x, y=train_part_y,
                                      ctx=device.cpu, num.round=10, array.batch.size=100,
                                      learning.rate=0.1, momentum=0.9,  eval.metric=mx.metric.accuracy,
                                      initializer=mx.init.uniform(0.1),
                                      epoch.end.callback=mx.callback.log.train.metric(100))
```

From the summarized table, we can see that both test accuracy and FOM get slight improvement when we use less than 40% of training data. Since we mainly focus on using from 1% to 20% of training data, the result of tuning learning rate can be acceptable.

Finally, after tuning parameters, we are going to find out where the minimum FOM resides. The following table shows the comparison between our initial model and optimized model with tuned parameters.

	Model 1		Model 1 optimized	
	Test Accuracy	FOM	Test	FOM
1%	0.7883	0.2167	0.8909	0.1141
2%	0.796	0.214	0.9197	0.0903
3%	0.878	0.137	0.9338	0.0812

4%	0.8975	0.1225	0.9434	0.0766
5%	0.9056	0.1194	0.9473	0.0777
6%	0.91	0.12	0.953	0.077
7%	0.9321	0.1029	0.9552	0.0798
8%	0.9312	0.1088	0.9573	0.0827
9%	0.9439	0.1011	0.959	0.086
10%	0.9326	0.1174	0.9609	0.0891
11%	0.9345	0.1205	0.9639	0.0911
12%	0.9456	0.1144	0.9644	0.0956
13%	0.9451	0.1199	0.9673	0.0977
14%	0.9501	0.1199	0.967	0.103
15%	0.952	0.123	0.9682	0.1068
16%	0.9473	0.1327	0.9679	0.1121
17%	0.9582	0.1268	0.9699	0.1151
18%	0.9457	0.1443	0.9702	0.1198
19%	0.956	0.139	0.971	0.124
20%	0.958	0.142	0.9715	0.1285

From the table above, we can see that in our initial model, FOM values are relatively small in the range of using from 4% to 14% of training data. In the optimized model, the small range fall into 4% to 6% of training data. Also, in the optimized model, the test accuracy gets better sooner, and the minimum FOM occurs earlier.

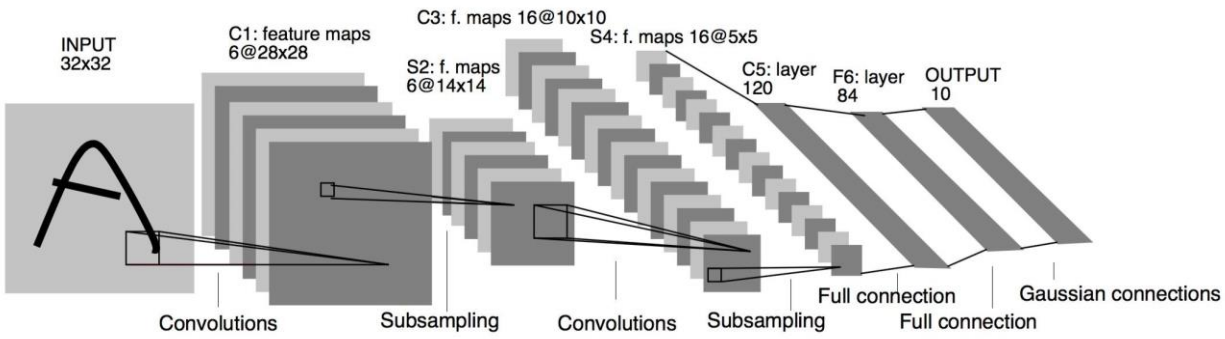
In conclusion, for our Fully-Connected Deep Neural Network model, the minimum FOM value we got is 0.0766. We use 4% of training data, which are 2400 examples. The test accuracy is 94.34%. Our model contains 2 hidden layers. The first hidden layer contains 1024 neurons, and the second hidden layer contains 512 neurons. We train the model with 20 iterations, and the learning rate is 0.1.

Convolutional Neural Network Model

Model Introduction

LeNet-5

LeNet-5 was designed by Yann LeCun in 1994, which is a family of convolutional neural network. Convolutional neural network uses sequence of 3 layers: convolution, pooling, non-linearity. The lower-layers are composed to alternating convolution and max-pooling layers. The upper-layers are fully-connected and multi-layer neural network (hidden layer + logistic regression).



Building Model

In our final model, we have two convolutional layers and two fully connected layers. Between each convolutional layers, we also apply max-pooling layers. Convolutional layers can capture spatial information with the size of local receptive (equivalently this is the filter size). One local receptive field represent a neuron. The hidden layers displayed with different neurons are creating a feature maps. We can also determine how many feature maps we need for learning. With more feature maps, it can learn different features thus the accuracy will rise. We define 5x5 local receptive fields. The kernel size and the number of feature maps are the hyperparameters.
1. In the first convolutional layers, we set 100 feature maps. After the first hidden layer, we apply the subsampling by using max-pooling method with a 2x2 input region.

```
data <- mx.symbol.Variable('data')
# first convolution layer
conv1 <- mx.symbol.Convolution(data=data, kernel=c(5,5), num_filter=100)
tanh1 <- mx.symbol.Activation(data=conv1, act_type="tanh")
pool1 <- mx.symbol.Pooling(data=tanh1, pool_type="max", kernel=c(2,2), stride=c(2,2))
```

2. We add another convolutional layer and a max-pooling layer. The second convolutional layer is using 5x5 local receptive fields, and 50 feature layers.

3. Because the output from the convolutional layers represents high-level features in the data, adding fully-connected layers can flatten and connect to our output layer. In the two fully connected layers, the number of hidden layers are 500 and 10.
4. The last hidden layer must be 10 because it correspond to our classifiers, which is the number of 0-9.

```
# second convolutional layer
conv2 <- mx.symbol.Convolution(data=pool1, kernel=c(5,5), num_filter=50)
tanh2 <- mx.symbol.Activation(data=conv2, act_type="tanh")
pool2 <- mx.symbol.Pooling(data=tanh2, pool_type="max",
                           kernel=c(2,2), stride=c(2,2))
# first fully connected
flatten <- mx.symbol.Flatten(data=pool2)
fc2.1 <- mx.symbol.FullyConnected(data=flatten, num_hidden=500)
tanh3 <- mx.symbol.Activation(data=fc2.1, act_type="tanh")
# second fullclly conncected
fc2.2 <- mx.symbol.FullyConnected(data=tanh3, num_hidden=10)
lenet <- mx.symbol.SoftmaxOutput(data=fc2.2)
```

Optimizing model

Adding layers

In this model, we have two convolutional layers and two fully connected layers.
We have tried adding more layers, either convolutional or fully connected, however, the results of accuracy are not improved. It also shows that adding another layer couldn't correctly classify the numbers. Thus, we choose not to add an additional layer and also we didn't report the results in a table because the FOM is too high. We conclude that the two convolutional layers are the optimized size of CNN model.

Changing the local receptive fields

We think that the smaller size of local receptive fields can train more better, thus we try to change 5x5 to 3x3 in the second convolutional layer. In the training data size around 5% to 10%, the FOM are all larger than the other. Thus the changning of kernel size may not be helpful.

Increase the feature maps

We try to increase the number of filters in the first convolutional models. The original model is using 20 feature maps and 50 feature maps in the corresponding convolutional layers. We assume that increase the number of feature maps can have huge difference on FOM and accuracy. Therefore, we increase from 20 to 100, and the local receptive field remain 5x5 in the

two convolutional layers. The result of FOM shows that the more feature maps can have a lower FOM. Since the CNN model with different parameters didn't change a lot in the 10% to 100% training dataset, we can look further in the 5% to 10% size of training data. When we increase the feature maps to 500, the result shows that the FOM comes to the **lowest of 0.71 with 6% of the training dataset.**

Firstly, we run the following model code with training data from 1%, 10% to 100%, and get test accuracy and FOM.

```
device.cpu <- mx.cpu()
mx.set.seed(0)
model2 <- mx.model.FeedForward.create(lenet, X=train.array, y=train_part_y,
                                      ctx=device.cpu, num.round=10, array.batch.size=100,
                                      learning.rate=0.05, momentum=0.9, wd=0.00001,
                                      eval.metric=mx.metric.accuracy,
                                      epoch.end.callback=mx.callback.log.train.metric(100))
```

Comparing each model with different parameters, we found that the range of every model that has good test accuracy and lowest FOM is from 1% to 10%. However, there is a large gap between 10% and 20%. Therefore, we will retrain 1% ~ 20% training data next. Expecially, we will focus FOM performances from 5% to 10%.

The proportion of training data	Model2 (20,50,500,10) (5,5) (5,5)		Model2 Changing the local receptive fields (20,50,500,10) (5,5) **(3,3)**		Model2 Increase the feature maps (**100**,50,500,10) (5,5) (5,5)	
	Test Accuracy	FOM	Test Accuracy	FOM	Test Accuracy	FOM
1%	0.1032	0.9018	0.1032	0.9018	0.1032	0.9018
10%	0.9692	0.0808	0.9646	0.0854	0.9704	0.0796
20%	0.9787	0.1213	0.9793	0.1207	0.9786	0.1214
30%	0.9812	0.1688	0.9834	0.1666	0.9803	0.1697
40%	0.9866	0.2134	0.9858	0.2142	0.9854	0.2146
50%	0.9876	0.2624	0.9878	0.2622	0.988	0.262
60%	0.9875	0.3125	0.9883	0.3117	0.9879	0.3121
70%	0.9895	0.3605	0.9896	0.3604	0.9903	0.3597
80%	0.9902	0.4098	0.99	0.41	0.9902	0.4098

90%	0.9907	0.4593	--	--	0.9895	0.4605
100%	0.9894	0.5106	--	--	0.9895	0.5105

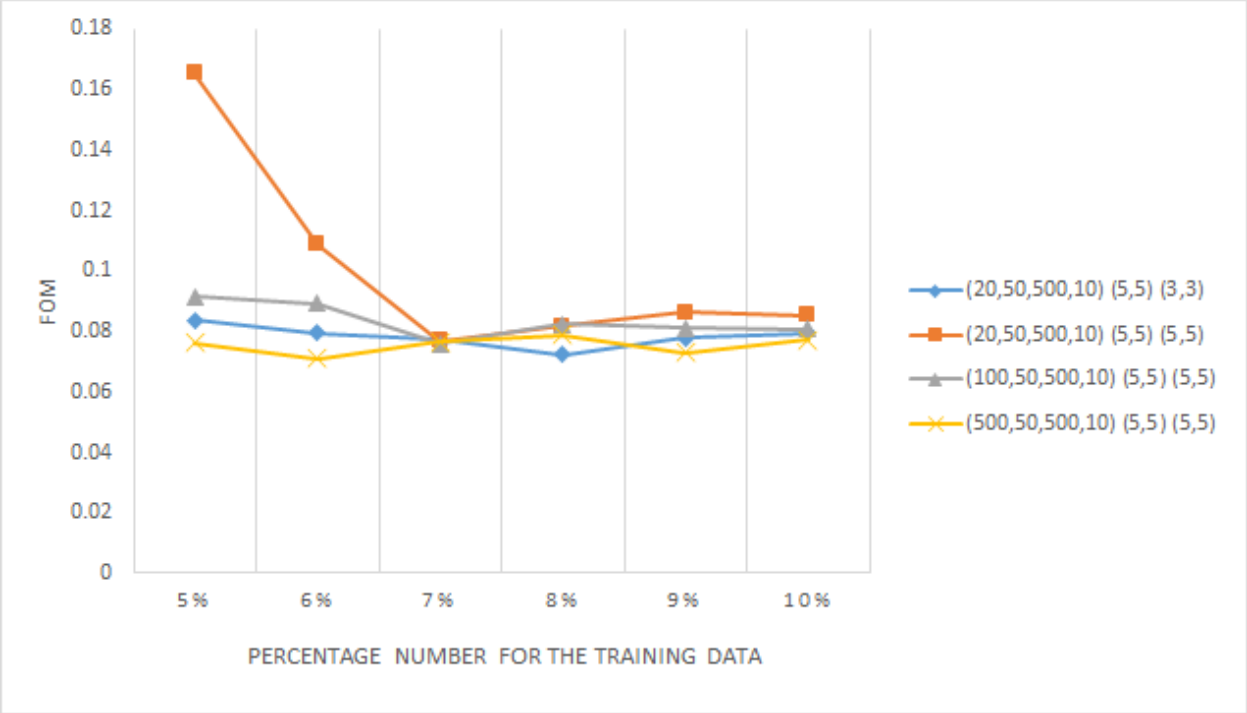
*90% and 100% of training data have large FOM, so we are unnecessary to run them.

Comparing each model with different parameters, we highlight lowest FOM in each model.

- (20,50,500,10) (5,5) (3,3), 7%, FOM 0.077
- (20,50,500,10) (5,5) (5,5), 7%, FOM 0.0758
- (100,50,500,10) (5,5) (5,5), 8%, FOM 0.0725
- (20,50,500,10) (5,5) (3,3), 6%, FOM **0.071**

The proportion of training data	Model 2 (20,50,500,10) (5,5) (**3,3**)		Model 2 (20,50,500,10) (5,5) (5,5)		Model 2 (**100**,50,500,10) (5,5) (5,5)		Model 2 (**500**,50,500,10) (5,5) (5,5)	
	Test Accuracy	FOM	Test Accuracy	FOM	Test Accuracy	FOM	Test Accuracy	FOM
1%	0.1032	0.9018	0.1032	0.9018	0.1032	0.9018	--	--
2%	0.1135	0.8965	0.1135	0.8965	0.2182	0.7918	--	--
3%	0.1135	0.9015	0.1544	0.8606	0.8801	0.1349	--	--
4%	0.8232	0.1968	0.9036	0.1164	0.9305	0.0895	--	--
5%	0.8595	0.1655	0.9333	0.0917	0.9314	0.0837	0.9486	0.0764
6%	0.9212	0.1088	0.9407	0.0893	0.9506	0.0794	0.959	**0.071**
7%	0.958	**0.077**	0.9592	**0.0758**	0.9576	0.0774	0.9583	0.0767
8%	0.9583	0.0817	0.9575	0.0825	0.9675	**0.0725**	0.961	0.079
9%	0.9586	0.0864	0.9637	0.0813	0.9671	0.0779	0.9719	0.0731
10%	0.9646	0.0854	0.9692	0.0808	0.9704	0.0796	0.9728	0.0772
11%	0.9609	0.0941	0.9682	0.0868	0.9737	0.0813	--	--
12%	0.9706	0.0894	0.9703	0.0897	0.9741	0.0859	--	--
13%	0.9724	0.0926	0.9755	0.0895	0.9747	0.0903	--	--
14%	0.9721	0.0979	0.9735	0.0965	0.9757	0.0943	--	--
15%	--	--	0.9764	0.0986	0.977	0.098	--	--
16%	--	--	0.9743	0.1057	0.9781	0.1019	--	--

17%	--	--	0.9768	0.1082	0.98	0.105	--	--
18%	--	--	0.9789	0.1111	0.9795	0.1105	--	--
19%	--	--	0.9789	0.1161	0.9773	0.1177	--	--
20%	--	--	0.9786	0.1214	0.9786	0.1214	--	--



As this graph shows, we compare the four different changes in the parameters of the CNN model. The model with yellow curve is the model of 500 filters in first convolutional layer, 50 filters in second convolutional layer, 500 hidden layers in first full connection layer, 10 hidden layer in our output, and 5*5 kernel. It has the lowest FOMin 6% and 9% and 10% percent of training dataset. Therefore, we get our lowest FOM, **0.071**, **with 6% training size (3600 examples).**

Conclusion and Teamwork

We explored two machine learning techniques, Fully-Connected Deep Neural Network (DNN) and Convolutional Neural Network (CNN). Based on the codes introduced by R-bloggers, we trained the models with different training data sizes and tuned the parameters for multiple times. Analyzing the results, we finally choose the CNN model as our best model.
In this project, our team did as great as midterm project. Before building the models, we worked together to find out a way to load data into R. Because the source data provided are not csv files and are rather complicated, we could not follow the normal routine. After a lively discussion, we decided to use DNN and CNN.

Sui Ping Suen and Helen Tao took charge of DNN. After studying and testing the codes from R-bloggers, we builded our initial model and then worked on optimizing it. We first ran the model with training data size from 10% to 100% to see how FOM is related to the training data size. Then, we tuned the model with added hidden layers, changed neuron numbers for each layer, changed round.num, and changed learning rate. After countless number of trials, we achieved the optimized model and ran it with data size from 1% to 20%. Our smallest FOM is 0.0766 using 4% of the whole training data.

Ya-Hsuan Chuo and Barbara Lu are responsible for CNN. Writing code is not a challenge this time. For CNN, our challenge is how to change parameters to get an optimal model. We ran a turn function in midterm project to find the best parameters. However, this time, we have to change parameters manually. After hundreds of run, we found that adding number of running time, adding filters in first convolutional layer and changing kernel worked well, but adding layers had no effects in CNN model. Ya-Hsuan and Barbara shared out the work and cooperated together. Ya-Hsuan ran (100)(5,5),(500)(5,5) model, and Barbara ran (20)(5,5), (20)(3,3) model.

During the process, we also explored on how to use GPU to do the computation and how to employ data deformation to artificially generated more training data. However, with the time constraint, we did not work out a practical way, but the discussion on Data Deformation is provided below. We will further explore on this topic even after the course ends.

Every member of our team contributed a lot of time and efforts to this project, and we really had a great experience studying and exploring together.

Data Deformation

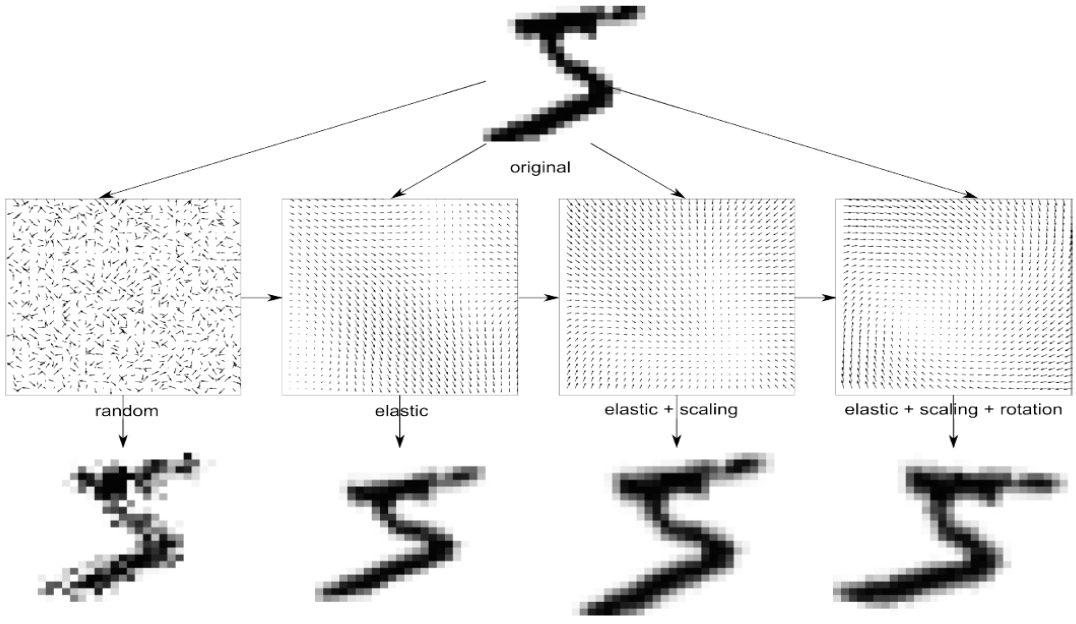
To deform the digit images, we use affine (rotation, scaling and horizontal shearing) and elastic deformations. The distortions are characterized by the respective real-valued parameters:

1. σ and α: for elastic deformations emulating uncontrolled oscillations of hand muscles;
2. β: a random angle from [-β;+β] describes either rotation or horizontal shearing. In case of shearing, tanβ defines the ratio between horizontal displacement and image height;
3. γx and γy: for horizontal and vertical scaling, randomly selected from [1-γ/100 , 1+γ/100].

Each affine deformation is fully defined by the corresponding real-valued parameter that is randomly drawn from a uniform interval.

The elastic deformation was performed by defining a normalized random displacement field u(x,y) that for each pixel location (x, y) in an image specifies a unit displacement vector, such that Rw = Ro + u, where Rw and Ro describe the location of the pixels in the original and warped images respectively. The strength of the displacement in pixels is given by α. The smoothness of the displacement field is controlled by the parameter σ, which is the standard deviation of the Gaussian that is convolved with matrices of uniformly distributed random values that form the x and y dimensions of the displacement field u.

The following picture explains the process.



When we want to employ these deformations, we need to write the code, tune the parameters to achieve an ideal accuracy, and ensure the warped images are still human recognizable. We will explore further into this topic when time is sufficient.

References

Anonymous. (n.d.). Convolutional Neural Networks (LeNet). Retrieved April 27, 2017, from http://deeplearning.net/tutorial/lenet.html

Ciresan, D. C., Meier, U., Gambardella, L. M., & Schmidhuber, J. (2012). Deep Big Multilayer Perceptrons For Digit Recognition. *Tricks of the Trade,* 581-598. Retrieved April 27, 2017, from https://link.springer.com/chapter/10.1007%2F978-3-642-35289-8_31#page-1.

Culurciello, E. (2016, October 25). Neural Network Architectures. Retrieved April 26, 2017, from https://culurciello.github.io/tech/2016/06/04/nets.html

CS231n Convolutional Neural Networks for Visual Recognition. (n.d.). Retrieved April 27, 2017, from http://cs231n.github.io/neural-networks-1/#actfun

Póczos, B., & Singh, A. (2017, April 26). *Deep Learning*. Lecture presented in Carnegie Mellon University. Retrieved from http://www.cs.cmu.edu/~aarti/Class/10701_Spring14/slides/DeepLearning.pdf

R Stats. (2015, November 17). Deep Learning with MXNetR. Retrieved April 27, 2017, from https://www.r-bloggers.com/deep-learning-with-mxnetr/

Wong, S. C., Gatt, A., Stamatescu, V., & McDonnell, M. D. (2016, September 28). Understanding data augmentation for classification: when to warp? Retrieved April 27, 2017, from https://arxiv.org/abs/1609.08764