**Project: Classification of the MNIST dataset**

***Mei Eisenbach***

**Introduction**

The MNIST dataset was developed for the evaluation of machine learning models on handwritten digit classification. It is comprised of tens of thousands of images of handwritten numbers from different individuals. Each image is a 28 x 28 pixel square with greyscale values.

This script contains three models:

* Random Forest
* A simple neural network using nnet
* A deep neural network using mxnet

**Step 2: Exploring and preparing the data**

# read in data

library(readr)

train\_orig <- read\_csv("train.csv")

## Parsed with column specification:

## cols(

## .default = col\_integer()

## )

## See spec(...) for full column specifications.

test\_orig <- read\_csv("test.csv")

## Parsed with column specification:

## cols(

## .default = col\_integer()

## )

## See spec(...) for full column specifications.

# save the training labels

train\_orig\_labels <- train\_orig[, 1]

train\_orig\_labels <- as.factor(train\_orig\_labels$label)

summary(train\_orig\_labels)

## 0 1 2 3 4 5 6 7 8 9

## 4132 4684 4177 4351 4072 3795 4137 4401 4063 4188

The training data consists 42,000 observations with a label (0-9) and 784 pixel variables (0-255). This is the distribution of digits in the training data.

**Random Forest**

A Random Forest model will be trained for comparison. Code adapted from <https://www.kaggle.com/benhamner/random-forest-benchmark-1>.

library(randomForest)

numTrees <- 25

# Train on entire training dataset and predict on the test

startTime <- proc.time()

rf <- randomForest(train\_orig[-1], train\_orig\_labels, xtest=test\_orig,

ntree=numTrees)

proc.time() - startTime

## user system elapsed

## 157.21 0.38 158.39

It took 2.5 minutes to train the random forest model.

rf

##

## Call:

## randomForest(x = train\_orig[-1], y = train\_orig\_labels, xtest = test\_orig, ntree = numTrees)

## Type of random forest: classification

## Number of trees: 25

## No. of variables tried at each split: 28

##

## OOB estimate of error rate: 6.28%

## Confusion matrix:

## 0 1 2 3 4 5 6 7 8 9 class.error

## 0 4039 1 7 12 4 13 27 4 21 4 0.02250726

## 1 0 4588 25 20 5 11 5 8 14 8 0.02049530

## 2 27 20 3905 50 27 13 27 51 43 14 0.06511851

## 3 14 11 89 3905 9 124 14 43 96 46 0.10250517

## 4 7 6 19 3 3829 6 27 16 30 129 0.05967583

## 5 31 10 12 116 16 3457 46 11 63 33 0.08906456

## 6 32 10 16 6 20 37 3994 0 19 3 0.03456611

## 7 11 20 60 27 28 7 1 4149 12 86 0.05725971

## 8 20 26 37 88 35 73 29 14 3665 76 0.09795717

## 9 24 5 16 49 108 35 8 55 55 3833 0.08476600

# output predictions for submission

predictions <- data.frame(ImageId=1:nrow(test\_orig),

Label=levels(train\_orig\_labels)[rf$test$predicted])

head(predictions)

## ImageId Label

## 1 1 2

## 2 2 0

## 3 3 9

## 4 4 9

## 5 5 3

## 6 6 7

write\_csv(predictions, "rf\_benchmark.csv")

The submission file has the imageId and the predicted label. The predictions were submitted to Kaggle and received a score of 0.95914.

**Step 3: Training a model on the data**

**Train a neural network using nnet**

library(nnet)

# split the training data into train and test to do local evaluation

set.seed(123)

rows <- sample(1:nrow(train\_orig), as.integer(0.7\*nrow(train\_orig)))

# Get train and test labels

train\_labels <- train\_orig[rows, 1]

test\_labels <- train\_orig[-rows, 1]

# convert the labels to factors

train\_labels <- as.factor(train\_labels$label)

# custom normalization function

normalize <- function(x) {

return(x / 255)

}

# create the train and test datasets and apply normalization

train\_norm <- as.data.frame(lapply(train\_orig[rows, -1], normalize))

test\_norm <- as.data.frame(lapply(train\_orig[-rows,-1], normalize))

# check a random pixel to see if the normalization worked

summary(train\_orig$pixel350)

## Min. 1st Qu. Median Mean 3rd Qu. Max.

## 0.00 0.00 0.00 89.51 228.00 255.00

summary(train\_norm$pixel350)

## Min. 1st Qu. Median Mean 3rd Qu. Max.

## 0.000000 0.000000 0.003922 0.350500 0.890200 1.000000

summary(test\_norm$pixel350)

## Min. 1st Qu. Median Mean 3rd Qu. Max.

## 0.0000 0.0000 0.0000 0.3521 0.9059 1.0000

Neural networks work best on data that is normalized.

# create the class indicator matrix

train\_labels\_matrix = class.ind(train\_labels)

head(train\_labels)

## [1] 2 2 9 2 8 7

## Levels: 0 1 2 3 4 5 6 7 8 9

head(train\_labels\_matrix)

## 0 1 2 3 4 5 6 7 8 9

## [1,] 0 0 1 0 0 0 0 0 0 0

## [2,] 0 0 1 0 0 0 0 0 0 0

## [3,] 0 0 0 0 0 0 0 0 0 1

## [4,] 0 0 1 0 0 0 0 0 0 0

## [5,] 0 0 0 0 0 0 0 0 1 0

## [6,] 0 0 0 0 0 0 0 1 0 0

nnet requires the class labels as a one-of-n normalized matrix.

# train model

set.seed(123)

startTime <- proc.time()

nn = nnet(train\_norm, train\_labels\_matrix,

size = 1,

softmax = TRUE

)

## # weights: 805

## initial value 71631.780715

## iter 10 value 64946.553191

## iter 20 value 57294.824640

## iter 30 value 55912.804141

## iter 40 value 54648.757612

## iter 50 value 53950.781576

## iter 60 value 52927.199756

## iter 70 value 52291.634751

## iter 80 value 51967.602466

## iter 90 value 51774.654787

## iter 100 value 51643.951402

## final value 51643.951402

## stopped after 100 iterations

proc.time() - startTime

## user system elapsed

## 46.97 0.13 47.54

This is just to try out the nnet function. One hidden node is mostly likely not enough for the model. “softmax” should be set to TRUE when performing classification. The default maximum number of iterations is 100. The algorithm did not converge before reaching the maximum. It ran for 46 seconds.

nn

## a 784-1-10 network with 805 weights

## options were - softmax modelling

**Step 4: Evaluating model performance**

# get predictions

pred = predict(nn, test\_norm, type="class")

cbind(head(pred), head(test\_labels))

## head(pred) label

## 1 1 8

## 2 1 3

## 3 1 8

## 4 1 0

## 5 1 3

## 6 7 4

The first six predictions do not look very good.

# evaluate the model

accuracy <- mean(pred == test\_labels)

print(paste('Accuracy:', accuracy))

## [1] "Accuracy: 0.206174113165622"

The accuracy is 20.1% which is pretty bad but this was expected given that there is only one hidden node.

**Step 5: Improving model performance**

**Neural network autotuned with caret**

library(caret)

The caret package allows you to tune model parameters (e.g. number of hidden node). In addition, you can do k-fold cross-validation.

# Enable parallel processing

library(doParallel)

## Loading required package: foreach

## Loading required package: iterators

## Loading required package: parallel

# subtract 1 from detectCores() to reduce the number of cores used

no\_cores <- detectCores()

cl <- makeCluster(no\_cores)

cl

## socket cluster with 4 nodes on host 'localhost'

registerDoParallel(cl)

The automated tuning process is very processor intensive and will take a while. To help with that, use the doParallel package to enable caret to use more than one processor.

# Set up training parameters

TrainingParameters <- trainControl(method = "cv", number = 3)

grid\_nn <- expand.grid(.size = c(1, 3, 5, 10),

.decay = 0)

grid\_nn

## .size .decay

## 1 1 0

## 2 3 0

## 3 5 0

## 4 10 0

3-fold cross-validation was used instead of 10 because of time constraints. The “decay” parameter is for the weight decay. This value is typically between 1e-04 and 1e-02 and helps the optimization process and to avoid overfitting. This value was left at 0 because with 10 or less hidden nodes overfitting is not a concern with the amount of data that is being trained on.

# use all of the given training data

train\_norm <- as.data.frame(lapply(train\_orig[, -1], normalize))

startTime <- proc.time()

set.seed(123)

nn2 <- train(train\_norm, train\_orig\_labels,

trControl= TrainingParameters,

method = "nnet",

tuneGrid = grid\_nn,

MaxNWts = 20000

)

## # weights: 7960

## initial value 113869.131729

## iter 10 value 38250.007353

## iter 20 value 22933.929209

## iter 30 value 16712.847863

## iter 40 value 14600.216283

## iter 50 value 12393.910594

## iter 60 value 10789.659868

## iter 70 value 9734.241256

## iter 80 value 8880.801485

## iter 90 value 8278.771459

## iter 100 value 7871.091151

## final value 7871.091151

## stopped after 100 iterations

proc.time() - startTime

## user system elapsed

## 342.38 2.47 924.04

Because we’re doing k-fold cross validation, we do not need to partition the data into training and test. caret will also report the accuracy for each model. Training took about 30 minutes to complete.

nn2

## Neural Network

##

## 42000 samples

## 784 predictor

## 10 classes: '0', '1', '2', '3', '4', '5', '6', '7', '8', '9'

##

## No pre-processing

## Resampling: Cross-Validated (3 fold)

## Summary of sample sizes: 27998, 28002, 28000

## Resampling results across tuning parameters:

##

## size Accuracy Kappa

## 1 0.2330441 0.1431423

## 3 0.5703788 0.5221642

## 5 0.8336191 0.8150801

## 10 0.9002856 0.8891680

##

## Tuning parameter 'decay' was held constant at a value of 0

## Accuracy was used to select the optimal model using the largest value.

## The final values used for the model were size = 10 and decay = 0.

The best model had 10 hidden nodes and an estimated accuracy of 0.89.

# normalize test data and predict on it

test\_norm <- as.data.frame(lapply(test\_orig, normalize))

NNPredictions <-predict(nn2, test\_norm)

# output predictions for submission

predictions <- data.frame(ImageId=1:nrow(test\_orig),

Label=levels(train\_orig\_labels)[NNPredictions])

head(predictions)

## ImageId Label

## 1 1 2

## 2 2 0

## 3 3 9

## 4 4 7

## 5 5 2

## 6 6 7

write\_csv(predictions, "caret\_nnet.csv")

The submission file has the imageId and the predicted label. The file was submitted to Kaggle and received a score of 0.90086, which is lower than the score from the Random Forest model.

# stop the cluster for parallel processing

stopCluster(cl)

**Deep Neural Network using MXNet**

Adapted from tutorial: <http://dmlc.ml/rstats/2015/11/03/training-deep-net-with-R.html>

# load the namespace of mxnet package and attach it

require(mxnet)

## Init Rcpp

# transpose the input matrix to npixel x nexamples, which is the column major

# format accepted by mxnet

train\_matrix <- data.matrix(train\_orig)

test\_matrix <- data.matrix(test\_orig)

train\_matrix.x <- train\_matrix[,-1]

train\_matrix.y <- train\_matrix[,1]

# normalize

train\_matrix.x <- t(train\_matrix.x/255)

test\_matrix <- t(test\_matrix/255)

# check the distribution of the classes

table(train\_matrix.y)

## train\_matrix.y

## 0 1 2 3 4 5 6 7 8 9

## 4132 4684 4177 4351 4072 3795 4137 4401 4063 4188

## Network Configuration

data <- mx.symbol.Variable("data")

fc1 <- mx.symbol.FullyConnected(data, name="fc1", num\_hidden=10)

act1 <- mx.symbol.Activation(fc1, name="relu1", act\_type="relu")

fc2 <- mx.symbol.FullyConnected(act1, name="fc2", num\_hidden=10)

softmax <- mx.symbol.SoftmaxOutput(fc2, name="sm")

This is a neural network with 1 hidden layer of 10 nodes. The parameters were configuered like this for comparison with nnet.

Configuration variables:

* data represents the input layer
* fc1 is the first hidden layer
* act1 is the first activation function
* fc2 is the output layer, the number of neurons is set to 10 (the number of classes we have)
* softmax is the activation function for fc2, which will produce a probabilistic prediction

# set the device to the cpu

devices <- mx.cpu()

# train the model

mx.set.seed(0)

model <- mx.model.FeedForward.create(softmax, X=train\_matrix.x, y=train\_matrix.y,

ctx=devices, 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))

## Warning in mx.model.select.layout.train(X, y): Auto detect layout input matrix, use colmajor..

## Start training with 1 devices

## [1] Train-accuracy=0.834105011933174

## [2] Train-accuracy=0.91352380952381

## [3] Train-accuracy=0.923238095238096

## [4] Train-accuracy=0.929738095238096

## [5] Train-accuracy=0.933714285714286

## [6] Train-accuracy=0.933666666666667

## [7] Train-accuracy=0.935523809523809

## [8] Train-accuracy=0.936880952380952

## [9] Train-accuracy=0.938761904761904

## [10] Train-accuracy=0.939357142857142

The training accuracy after 10 iterations is 0.94. This took only a few seconds to run.

## Network Configuration

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")

Now we can see what kind of accuracy we get with two hidden layers. The first hidden layer has 128 nodes and the second has 64 nodes.

# train the model

mx.set.seed(0)

model <- mx.model.FeedForward.create(softmax, X=train\_matrix.x, y=train\_matrix.y,

ctx=devices, 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))

## Warning in mx.model.select.layout.train(X, y): Auto detect layout input matrix, use colmajor..

## Start training with 1 devices

## [1] Train-accuracy=0.859832935560859

## [2] Train-accuracy=0.958309523809525

## [3] Train-accuracy=0.971000000000004

## [4] Train-accuracy=0.977119047619051

## [5] Train-accuracy=0.982571428571431

## [6] Train-accuracy=0.986904761904765

## [7] Train-accuracy=0.98816666666667

## [8] Train-accuracy=0.990428571428574

## [9] Train-accuracy=0.990738095238098

## [10] Train-accuracy=0.990809523809527

Accuracy: 0.99

# get predictions

preds <- predict(model, test\_matrix)

## Warning in mx.model.select.layout.predict(X, model): Auto detect layout input matrix, use colmajor..

dim(preds)

## [1] 10 28000

pred.label <- max.col(t(preds)) - 1

table(pred.label)

## pred.label

## 0 1 2 3 4 5 6 7 8 9

## 2927 3163 2734 2889 2723 2453 2764 2848 2721 2778

# output submission

submission <- data.frame(ImageId=1:ncol(test\_matrix), Label=pred.label)

write.csv(submission, file='mxnet.csv', row.names=FALSE, quote=FALSE)

Kaggle score: 0.97014. This is higher than the Random Forest score. The accuracy on the test dataset is lower than the training accuracy, which is to be expected.

**Conclusion**

| **Model** | **Accuracy** |
| --- | --- |
| Random Forest | 95.914% |
| nnet | 90.086% |
| MXNet | 97.014% |

The MXNet model with 2 hidden layers (128 and 64 hidden nodes) performed the best out of all the models tested. The model trained using MXNet was a Multilayer Perceptron. There are many other types of neural network models available and it’s also possible to do the processing on a GPU. Currently, the best result reported for neural networks is 99.79% accuracy.

Random Forest performed surprisingly well, was easy to train, and fairly quick. The limitation with Random Forest is when datasets become larger.

The nnet package is not really usuable for large datasets as it is extremely slow. But it is good as an introduction to neural networks because it’s simpler.