MNIST Digit Recognition

Rohit Padebettu

6/4/2017

Introduction

The MNIST case is the "Hello World" of image recognition in deep learning. It is a great illustration of what can be acomplished with neural network algorithms. The MNIST(Modern National Institute of Standards and Technology) is a large database of handrwritten digits commonly used in machine learning and image processing systems.

The mnist dataset is large and highly pre-processed, making it easy to get started for image recoginition purposes. The data consists of pixel values for 70,000 handwritten digits 0 - 9 from many scanned documents. 10,000 of the scanned digits are separated out as the test set. It also includes labels for each image, telling us which digit it is. We will use these labels to builda supervised learning algorithm.

In this case we're going to train a model to look at images and predict what digits from 0-9 they are. That makes this a multi-class classification problem. Our goal isn't to train a really elaborate model that achieves state-of-the-art performance, but rather to dip a toe into using Neural Networks in R. We are also going to try to classify the same dataset using some other non-neural network algorithms we have learnt to see how they perform compared to the neural network based algorithm.

More details about the dataset, including algorithms that have been tried on it and their levels of success, can be found at http://yann.lecun.com/exdb/mnist/index.html.

Dataset

The dataset is available at at many places throughout the internet in various forms. In our case let us download the files from pjreddie's website or kaggle, since these two sources provide csv files for train and test set separately which are easily processed.

To download the files to our local machine, we use the following code. the train set is about 105MB and the test set is about 18 MB

```r
# Training Set
download.file(url = "https://pjreddie.com/media/files/mnist_train.csv", method="curl",
              destfile = "./train.csv")
# Testing Set
download.file("https://pjreddie.com/media/files/mnist_test.csv", method="curl",
              destfile = "./test.csv")
```

The data files train.csv and test.csv contain gray-scale images of hand-drawn digits, from zero through nine.

Each image is 28 pixels in height and 28 pixels in width, for a total of 784 pixels in total. Each pixel has a single pixel-value associated with it, indicating the lightness or darkness of that pixel, with higher numbers meaning darker. This pixel-value is an integer between 0 and 255, inclusive.

Both datasets have 785 columns. The first column is an integer indicating the label of the digit drawn by the user. The rest of the columns contain the pixel-values of the associated image.

Installing the required packages

We will introduce couple of new packages as part of this case. We are already familiar with the great `caret` package we used in the `Leaf Classification Case` earlier. Here we will introduce two more very useful packages `data.table` and `h2o`.

DATA.TABLE Package:

The `data.table` package is an incredibly powerful package in R to learn. It provides an enhanced version of R's regular data.frames. Querying and processing records or variables using data.table is orders of maginitude faster than using data.frames or anything else available in R. This feature is even more useful as the data set size grows and complex data manipulations are needed to be performed. All this speed comes at the cost of a slightly complex,cryptic and difficult to master syntax. However it is worth to invest the time to learn the syntax to be able to save more time later on in working with bigger and complex data. `data.table` also provides a very fast file reading function called `fread` which is super fast and very efficient compated to other file readers in R like `read.csv` and `read.table`. To learn more about the `data.table` package type `??data.table` within your R console after installation and go through all the help tutorials.

H2O Package:

The `h2o` package in R is an implementation of the H2O library. H2O is an open source, distributed, java machine learning library. This means this library implements common machine learning algorithms and even cutting edge machine learning algorithms in Java and makes it availabe to us in form of a nice R package. This package is useful because it brings scale and distributed processing to R allowing us to work on some truly big data sets. A good introduction to H2O can be found here

H2O allows us to take greater advantage of our CPU processing power. It can also be connected with clusters at cloud platforms for doing computations at scale in a distributed way. Along with, it uses in-memory compression to handle large data sets even with a small cluster. In addition, H2O provides a nice GUI interface to build store and watch model performance. More on it's performance on some data sets can be found here

To install both these packages and load the `caret` package we do the following

```r
install.packages("h2o")
install.packages("data.table")
```

```r
suppressWarnings(library(caret))
```

```
## Loading required package: lattice
```

```
## Loading required package: ggplot2
```

```r
suppressWarnings(library(h2o))
```

```
##
## ----------------------------------------------------------------------
##
## Your next step is to start H2O:
##     > h2o.init()
##
## For H2O package documentation, ask for help:
##     > ??h2o
##
## After starting H2O, you can use the Web UI at http://localhost:54321
## For more information visit http://docs.h2o.ai
##
```

```
## ----------------------------------------------------------------------
##
## Attaching package: 'h2o'
## The following objects are masked from 'package:stats':
##
##     cor, sd, var
## The following objects are masked from 'package:base':
##
##     &&, %*%, %in%, ||, apply, as.factor, as.numeric, colnames,
##     colnames<-, ifelse, is.character, is.factor, is.numeric, log,
##     log10, log1p, log2, round, signif, trunc
```

```
suppressWarnings(library(data.table))
```

```
##
## Attaching package: 'data.table'
## The following objects are masked from 'package:h2o':
##
##     hour, month, week, year
```

Loading and Exploring the dataset

Once the necessary packages are loaded, we can begin working on our dataset by first loading the data into memory and then exploring it.

Loading the data

To load the data into memory we use `data.table's` fast `fread` function as below

```
train<-fread(input = "train.csv")
```

```
##
Read 66.7% of 60000 rows
Read 60000 rows and 785 (of 785) columns from 0.102 GB file in 00:00:03
```

```
test<-fread(input = "test.csv")
```

Exploring the data

We can begin exploring the data by first looking at the dimensions of the dataset, the structure of the dataset and viewing the first few rows of the data set

Dimensions of the data

```
dim(train)
```

```
## [1] 60000   785
```

```
dim(test)
```

```
## [1] 10000   785
```

We can see that the `train` set has 60,000 rows(images) spread across 785 columns and the `test` set has 10,000 rows(images) spread across similar 785 columns

Structure of the data

```
str(train)
str(test)
```

You can load the stucture of any data set using the `str` function and observe how the dataset is organized

Sample Data

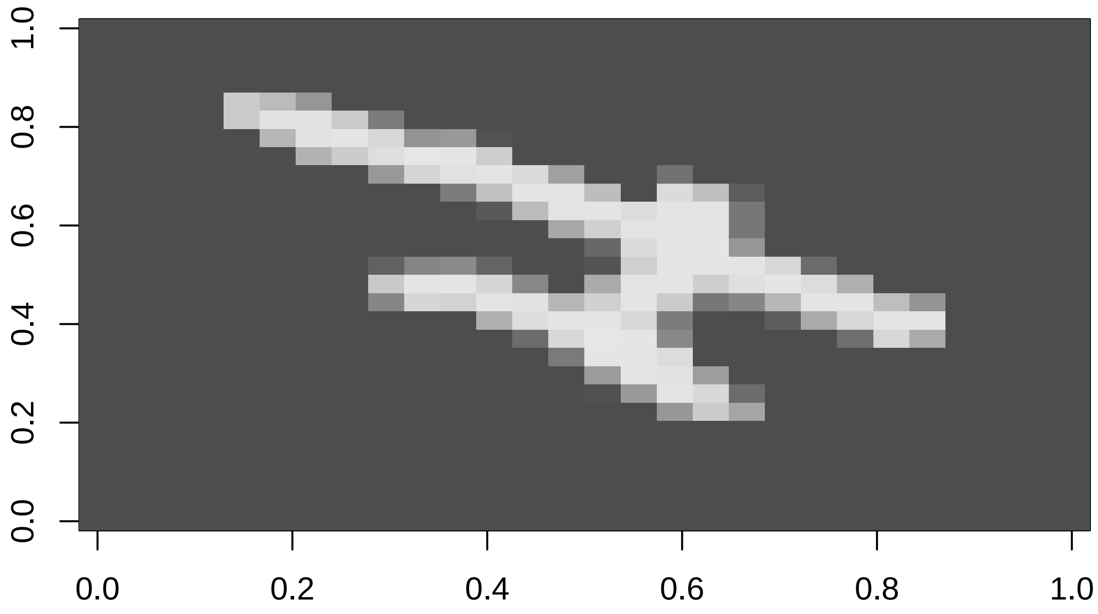
```
head(train)
```

The first column `V1` in the data is the **label** and the rest of the 784 columns indicate the pixel density

---

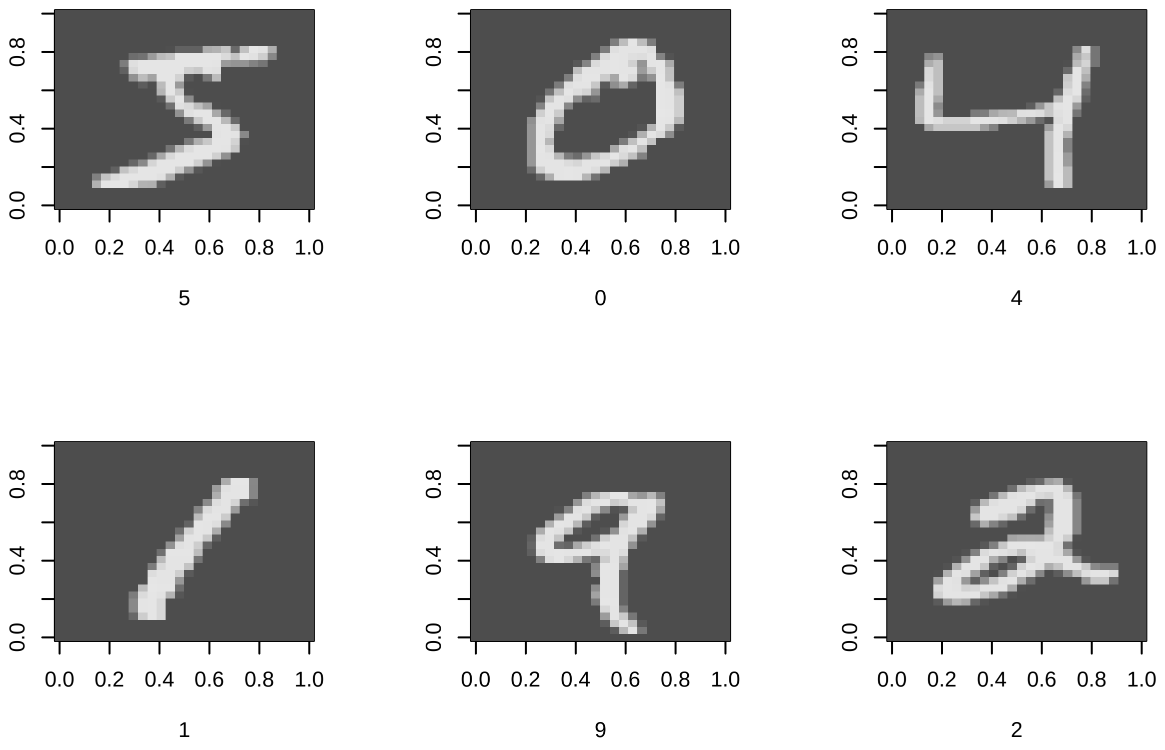
Plot the Data Since these are known to be images, we can plot the first few images in the `train` set to see how they appear

```
# Plot that matrix
image(m,col=grey.colors(255))
```



```
# reverses (rotates the matrix)
rotate <- function(x) t(apply(x, 2, rev))
```

```
# Plot some of images
par(mfrow=c(2,3))
plt<-lapply(1:6,
       function(x) image(
         rotate(matrix(unlist(train[x,-1]),nrow = 28, byrow = TRUE)),
         col=grey.colors(255),
         xlab=train[x,1]
       )
)
```



```r
par(mfrow=c(1,1)) # set plot options back to default
```

The images in this dataset are highly processed using some other image manipulation softwares to scale and center these images exactly in order to be consistently read by the machine. If we are to use our own handwritten examples to train our model, we would also need to perform similar sort of image manipulations involving conversion to grayscale, centering, cropping, rotating and scaling.

Modeling the data

Now that we understand how the data is structured and stored in our tables, we can begin building machine learning models to learn the patterns in the `train` data. These models can later be used to predict and compare with the labels in the `test` data.

Random Forest Model

As a first step let us model this data using the `randomforest` algorithm in the previously learnt `CARET` package. Tree based algorithms usually work one variable at a time. `randomforest` algo improves the accuracy of the tree based algorithms by creating an ensemble of diverse trees. However these are known to be no very good for image classification tasks especially using just the default variables without any feature engineering.

```r
inTrain = data.frame(y=as.factor(unlist(train[,1])), train[,-1])
inTest = data.frame(y=as.factor(unlist(test[,1])), test[,-1])

# Converting the first label column into factors
inTrain$y <- as.factor(inTrain$y)
inTest$y <- as.factor(inTest$y)
```

Also we will only be using the first 1000 images to classify because this algo can take a long time to run.

```r
# Random Forest algorithm
set.seed(1000)

# Setting up the parameter search grid for rf
rfGrid <-  expand.grid( mtry = c(10,30,100))
model.rf <- train(y ~ ., # classify Y as a function of all other variables
             data = head(inTrain, 1000), # Pick the top 1000 images
             method = 'rf',
             tuneGrid = rfGrid, # Tune the model for different values of mtry
             ntree = 300, # number of trees to grow
             metric = 'Accuracy', # metric to use for selecting model
             allowParallel = TRUE # allow trees to be built parallely
             )

saveRDS(model.rf,"RF Model")
```

NOTE: The above model would take some time to run on most machines

```r
model.rf<-readRDS("RF Model")
model.rf
```

```
## Random Forest
##
## 1000 samples
##  784 predictor
##   10 classes: '0', '1', '2', '3', '4', '5', '6', '7', '8', '9'
##
## No pre-processing
## Resampling: Bootstrapped (25 reps)
## Summary of sample sizes: 1000, 1000, 1000, 1000, 1000, 1000, ...
## Resampling results across tuning parameters:
##
##   mtry  Accuracy   Kappa
##    10   0.8777691  0.8638654
##    30   0.8781625  0.8643125
##   100   0.8734280  0.8590278
##
## Accuracy was used to select the optimal model using  the largest value.
## The final value used for the model was mtry = 30.
```

Prediction using Test set

Next we look at how the model performed againt some of the test set

```r
suppressWarnings(library(randomForest))
```

```
## randomForest 4.6-12

## Type rfNews() to see new features/changes/bug fixes.

##
## Attaching package: 'randomForest'

## The following object is masked from 'package:ggplot2':
##
##     margin
```

```r
results <- predict.train(model.rf, newdata = inTest)
confusionMatrix(results, inTest$y)
```

```
## Confusion Matrix and Statistics
##
##           Reference
## Prediction    0    1    2    3    4    5    6    7    8    9
##          0  955    0   11    8    1   18   16    2    9    8
##          1    0 1111    3    2    1   13    5   10    4    6
##          2    0    4  920   26    2    5   28   23   14    7
##          3    0    1    9  824    1   15    0    1   20   10
##          4    1    0   15    1  836   23   41    9   18   46
##          5    4    2    4  101    1  742   25    1   32   11
##          6    9    2   16    1   26   16  838    1   17    3
##          7    6    1   38   27    4   14    0  939   11   21
##          8    4   14   16   12    6   12    5    4  796    3
##          9    1    0    0    8  104   34    0   38   53  894
##
## Overall Statistics
##
##                Accuracy : 0.8855
##                  95% CI : (0.8791, 0.8917)
##     No Information Rate : 0.1135
##     P-Value [Acc > NIR] : < 2.2e-16
##
##                   Kappa : 0.8727
##  Mcnemar's Test P-Value : NA
##
## Statistics by Class:
##
##                      Class: 0 Class: 1 Class: 2 Class: 3 Class: 4 Class: 5
## Sensitivity            0.9745   0.9789   0.8915   0.8158   0.8513   0.8318
## Specificity            0.9919   0.9950   0.9878   0.9937   0.9829   0.9801
## Pos Pred Value         0.9290   0.9619   0.8941   0.9353   0.8444   0.8039
## Neg Pred Value         0.9972   0.9973   0.9875   0.9796   0.9838   0.9835
## Prevalence             0.0980   0.1135   0.1032   0.1010   0.0982   0.0892
## Detection Rate         0.0955   0.1111   0.0920   0.0824   0.0836   0.0742
## Detection Prevalence   0.1028   0.1155   0.1029   0.0881   0.0990   0.0923
## Balanced Accuracy      0.9832   0.9869   0.9397   0.9048   0.9171   0.9060
##                      Class: 6 Class: 7 Class: 8 Class: 9
## Sensitivity            0.8747   0.9134   0.8172   0.8860
## Specificity            0.9899   0.9864   0.9916   0.9735
## Pos Pred Value         0.9020   0.8850   0.9128   0.7898
## Neg Pred Value         0.9868   0.9900   0.9805   0.9870
## Prevalence             0.0958   0.1028   0.0974   0.1009
## Detection Rate         0.0838   0.0939   0.0796   0.0894
## Detection Prevalence   0.0929   0.1061   0.0872   0.1132
## Balanced Accuracy      0.9323   0.9499   0.9044   0.9298
```

As can be seen in the results above, while the OOB predicted error rate of the model from the training set alone was about 10.9%, the error rate on the test set was about 11.38%. We can imporve upon these results by using the full dataset for training and optimizing tuning parameters available to us, but that would probably take computational time totalling in hours. We will leave that exercise to those interested in such an effort.

H2O based Models

The first step is to initiate a H2O session. We will also ask our H2O session to use all the cores available on our machine. If we have a multicore machine, R usually uses only one core to perform it's computations. We can speed things up by utilizing all the computational power available to us through H2O.

```
# Initialize h2o on port 54321. The ntreads argument indicates how many CPU's to use (-1 meaning ALL av
h2o.init(port = 54321,nthreads = -1)
```

```
##  Connection successful!
##
## R is connected to the H2O cluster:
##     H2O cluster uptime:         20 hours 43 minutes
##     H2O cluster version:        3.10.2.1
##     H2O cluster version age:    5 months and 13 days !!!
##     H2O cluster name:           H2O_started_from_R_rohitpittu_qcf307
##     H2O cluster total nodes:    1
##     H2O cluster total memory:   2.78 GB
##     H2O cluster total cores:    8
##     H2O cluster allowed cores:  8
##     H2O cluster healthy:        TRUE
##     H2O Connection ip:          localhost
##     H2O Connection port:        54321
##     H2O Connection proxy:       NA
##     R Version:                  R version 3.3.2 (2016-10-31)

## Warning in h2o.clusterInfo():
## Your H2O cluster version is too old (5 months and 13 days)!
## Please download and install the latest version from http://h2o.ai/download/
```

```
## optional: connect to a running H2O cluster
#h2o.init(ip="mycluster", port=54321)
```

You can view the h2o session at (http://localhost:54321/flow/index.html) in the browser of your local machine

The next step is to **load the data** into H2O, which is accomplished as follows

```
train_h2o<-h2o.importFile("train.csv")
```

```
##
  |
  |                                                                 |   0%
  |
  |=====================================================            |  81%
  |
  |=================================================================| 100%
```

```
test_h2o<-h2o.importFile("test.csv")
```

```
##
  |
  |                                                                 |   0%
  |
  |=================================================================| 100%
```

If you are running this in RStudio, you can observe in the `Environment` pane (usually on the right top corner by default) that the size of these two H2O frames is minimal compared to original data. H2O has actually loaded the data into its Java Virtual Machine and is only making a reference object to it available within the R session. Data loaded into H2O can again be observed in the GUI at (http://localhost:54321/flow/index.html) using the `getFrames` function there or alternatively navigating to `Data -> List all Frames` in the menu bar at the top of the page. Since H2O compresses the objects, the size of the `train` and `test` data frame in the H2O session are 22MB and 5MB respectively

---

Building models within H2O

Let us begin by building a **Random Forest Model** again within H2O and see how quickly we can model and how accurately we can predict. This time it would be called a **Distributed Random Forest** model given H2O's scalable architecture.

Before we begin building our model, we need to factorize the labels in our dataset

```
train_h2o[,1]<-as.factor(train_h2o[,1])
test_h2o[,1]<-as.factor(test_h2o[,1])
```

Once that is done, setting up a **Distributed Random Forest** model is as simple as below. We will again use only the first 1000 images for training purpose to save time computing the model. The parameters for the model would be almost the same as the best model previously built in `CARET` but this time we will also perform a 5-fold cross validation to get a good estimate of the error of the model.

```
model.rf.h2o <- h2o.randomForest(x = 2:785,  # column numbers for predictors
                   y = 1,   # column number for label
                   training_frame  = train_h2o[1:1000,], # data in H2O format
                   nfolds = 5, # number of cross validation folds
                   ntrees = 300, # number of trees to grow
                   seed =1000, # setting the seed for randomness
                   mtries = 50 # number of variables to consider at each split
                   )

h2o.saveModel(model.rf.h2o,getwd())
```

You can follow the progress of the model at the GUI above at (http://localhost:54321/flow/index.html) navigating to `Admin-> Jobs` and clicking on the `DRF`(Distributed Random Forest) model. Training for this dataset with the 5 fold cross validation took about 2 minutes on my machine. Using more data would take longer time, but likely improve accuracy

The parameters and information about the model generated can be accessed either via the GUI or within RStudio as follows

```
model.rf.h2o<-h2o.loadModel(paste0(getwd(),"/DRF_model_R_1496539619488_2"))
model.rf.h2o
```

```
## Model Details:
## ==============
##
## H2OMultinomialModel: drf
## Model ID:  DRF_model_R_1496539619488_2
## Model Summary:
##   number_of_trees number_of_internal_trees model_size_in_bytes min_depth
## 1             300                     3000             1386335         5
##   max_depth mean_depth min_leaves max_leaves mean_leaves
## 1        20    9.77067         14         53    31.59900
```

```
##
##
## H2OMultinomialMetrics: drf
## ** Reported on training data. **
## ** Metrics reported on Out-Of-Bag training samples **
##
## Training Set Metrics:
## =====================
##
## MSE: (Extract with `h2o.mse`) 0.1799862
## RMSE: (Extract with `h2o.rmse`) 0.4242478
## Logloss: (Extract with `h2o.logloss`) 0.5828052
## Mean Per-Class Error: 0.0918264
## Confusion Matrix: Extract with `h2o.confusionMatrix(<model>,train = TRUE)`)
## =========================================================================
## Confusion Matrix: vertical: actual; across: predicted
##          0   1  2  3   4  5  6   7  8  9  Error         Rate
## 0       94   0  0  0   0  0  2   0  1  0 0.0309 =     3 / 97
## 1        0 112  0  1   0  1  1   1  0  0 0.0345 =    4 / 116
## 2        1   3 87  0   2  0  3   1  1  1 0.1212 =    12 / 99
## 3        2   1  2 83   1  1  0   1  0  2 0.1075 =    10 / 93
## 4        1   1  0  1  94  0  2   1  0  5 0.1048 =   11 / 105
## 5        0   0  1  3   1 82  2   0  1  2 0.1087 =    10 / 92
## 6        1   0  0  0   3  1 89   0  0  0 0.0532 =     5 / 94
## 7        0   3  1  0   4  0  0 106  0  3 0.0940 =   11 / 117
## 8        0   1  2  3   1  2  0   0 78  0 0.1034 =     9 / 87
## 9        2   0  1  1   6  1  0   5  0 84 0.1600 =   16 / 100
## Totals 101 121 94 92 112 88 99 115 81 97 0.0910 = 91 / 1,000
##
## Hit Ratio Table: Extract with `h2o.hit_ratio_table(<model>,train = TRUE)`
## =======================================================================
## Top-10 Hit Ratios:
##     k hit_ratio
## 1   1  0.909000
## 2   2  0.957000
## 3   3  0.978000
## 4   4  0.986000
## 5   5  0.989000
## 6   6  0.992000
## 7   7  0.995000
## 8   8  0.995000
## 9   9  0.996000
## 10 10  1.000000
##
##
##
## H2OMultinomialMetrics: drf
## ** Reported on cross-validation data. **
## ** 5-fold cross-validation on training data (Metrics computed for combined holdout predictions) **
##
## Cross-Validation Set Metrics:
## =====================
##
## MSE: (Extract with `h2o.mse`) 0.2032367
## RMSE: (Extract with `h2o.rmse`) 0.4508178
## Logloss: (Extract with `h2o.logloss`) 0.6096152
## Mean Per-Class Error: 0.106565
## Hit Ratio Table: Extract with `h2o.hit_ratio_table(<model>,xval = TRUE)`
## =======================================================================
## Top-10 Hit Ratios:
##     k hit_ratio
## 1   1  0.895000
## 2   2  0.950000
## 3   3  0.974000
## 4   4  0.983000
## 5   5  0.989000
## 6   6  0.991000
## 7   7  0.996000
## 8   8  0.998000
## 9   9  0.998000
## 10 10  1.000000
##
##
## Cross-Validation Metrics Summary:
##                                mean           sd cv_1_valid  cv_2_valid
## accuracy                  0.8948093  0.016087035 0.90430623    0.919598
## err                       0.1051907  0.016087035 0.09569378  0.08040201
## err_count                      21.0     3.065942       20.0        16.0
## logloss                  0.60982275  0.037726358 0.61370134  0.53229785
## max_per_class_error      0.26003578  0.042721555 0.18181819      0.3125
## mean_per_class_accuracy  0.89369303  0.013193592  0.9023471   0.9174735
## mean_per_class_error    0.106306985  0.013193592 0.09765291 0.082526505
## mse                      0.20325053  0.013141033 0.21037532   0.1769977
## r2                       0.97536546 0.0019522152 0.97300535   0.9794656
## rmse                       0.450366  0.014508653  0.4586669  0.42071095
##                          cv_3_valid cv_4_valid cv_5_valid
## accuracy                  0.8960396  0.9020619  0.8520408
## err                     0.103960395 0.09793814 0.14795919
## err_count                      21.0       19.0       29.0
## logloss                   0.6025902  0.6008425  0.6996819
## max_per_class_error       0.1904762 0.30769232 0.30769232
## mean_per_class_accuracy  0.89624727 0.89171755  0.8606797
## mean_per_class_error     0.10375274 0.10828246 0.13932028
## mse                      0.19697784 0.19827393 0.23362784
## r2                         0.977574 0.97462755 0.97215486
## rmse                     0.44382185  0.4452796 0.48335063
```

From the `Cross-Validation Metric Summary` table printed above we can see the model is expected to produce an accuracy rate of 89.48% on average, with a standard deviation of 1.6%.

Prediction using Test set

To predict the results from the model, we can use the `h2o.predict` function

```
## Using the RF model for predictions
h2o_rf_yhat_test <- h2o.predict(model.rf.h2o, test_h2o)
```

```
##
  |
  |                                                                      |   0%
  |
  |=================                                                     |  25%
  |
  |====================================================                  |  75%
  |
  |======================================================================| 100%
```

```r
## Converting H2O format into data frame
df_rf_yhat_test <- as.data.frame(h2o_rf_yhat_test)

## Getting test labels
test_labels<-unlist(as.data.frame(test_h2o[,1]))

## Table of predictions
table(test_labels,df_rf_yhat_test[,1])
```

```
##
## test_labels    0    1    2    3    4    5    6    7    8    9
##           0  956    0    0    0    0    2    8    5    9    0
##           1    0 1112    4    1    1    1    5    1   10    0
##           2   14    8  927   10   10    0   16   16   29    2
##           3   10    1   32  841    0   58    8   24   26   10
##           4    1   10    4    0  832    3   16    4   10  102
##           5   15    6    4   20   16  756   23   12   13   27
##           6   12    4   17    0   32   26  856    0   10    1
##           7    2    9   20    4    9    1    0  947    3   33
##           8   11    6   20   23   14   19   13   18  813   37
##           9    9    6    3   13   27    4    1   58    6  882
```

```r
## Accuracy of predictions
Acc<-sum(diag(table(test_labels,df_rf_yhat_test[,1])))/length(test_labels)
print(paste0("Accuracy = ",Acc*100,"%"))
```

```
## [1] "Accuracy = 89.22%"
```

We can see that our accuracy on test data is very close to the estimated accuracy of the model of about 90%. This indicates the model has generalized well.

Neural Network Model on H2O

Our final step would be to build a Neural network model on H2O and see how it performs compared to the other models we have built thus far. A good reference guide for Deep Learning and other neural networks on H2O is this

The code for setting up a Neural Network is very similar to what we did for the *Distributed Random Forest Model* with the difference only being Neural Network specific parameters.

Here, we will construct a 3 hidden layer neural network with 300,150 and 50 cells each. We will also perform a cross validation to get a good estimate of our training error as we did with the *Distributed Random Forest Model* . The activation function used for processing the signal generated by the network is a **Rectifier** which is also known as **ReLu** in some other packages.

Given the complexity of the network, we would need a lot more training data to avoid over-fitting our model to the data. We will therefore be using all the 60,000 images available to us in the `training set` to train this model.

```
model.nn.h2o <-
  h2o.deeplearning(x = 2:785,  # column numbers for predictors
                   y = 1,   # column number for label
                   training_frame  = train_h2o, # data in H2O format
                   activation = "Rectifier", # or 'RectifierWithDropout'
                   #input_dropout_ratio = 0.2, # % of inputs dropout
                   #hidden_dropout_ratios = c(0.5,0.5,0.5), # % for nodes dropout
                   hidden = c(300,150,50), # three layers of 50 nodes
                   epochs = 30,  # number of passes of data
                   variable_importances = TRUE, # variable importance
                   nfolds = 5 ) # cross validation folds

h2o.saveModel(model.nn.h2o,getwd())
```

As before you can observe the training of the model from within the RStudio console or at the GUI (http://localhost:54321/flow/index.html) navigating to `Admin-> Jobs` and clicking on the `Deep Learning` model. Training for this dataset with 5 fold cross validation, using all 60,000 images from the training set took about 20 minutes on my machine, which is significantly faster compared to Random Forest model which took 2 minutes to train on just 1,000 images.

The parameters and information about the model generated can be accessed either via the GUI or within RStudio as follows

```
model.nn.h2o<-h2o.loadModel(paste0(getwd(),"/DeepLearning_model_R_1496539619488_5"))
model.nn.h2o@model$cross_validation_metrics_summary
```

```
## Cross-Validation Metrics Summary:
##                                mean           sd  cv_1_valid  cv_2_valid
## accuracy                 0.96647036  0.001544542    0.964746  0.96381825
## err                     0.033529624  0.001544542 0.035254013 0.036181744
## err_count                     402.4    18.982098       424.0       434.0
## logloss                  0.15205763  0.004859943  0.14147529  0.15378635
## max_per_class_error     0.057990152 0.0038843818  0.05221932 0.061797753
## mean_per_class_accuracy    0.966173 0.0014169258   0.9645205  0.96385133
## mean_per_class_error    0.033827018 0.0014169258 0.035479467  0.03614867
## mse                     0.028995125 0.0011179982 0.029986542 0.030826071
## r2                        0.9965262 1.3525135E-4   0.9963921   0.9962994
## rmse                     0.17021503 0.0033145242  0.17316623  0.17557356
##                          cv_3_valid  cv_4_valid  cv_5_valid
## accuracy                 0.96622294   0.9700268   0.9675379
## err                      0.03377704 0.029973209 0.032462128
## err_count                     406.0       358.0       390.0
## logloss                  0.16200776   0.1482223  0.15479648
## max_per_class_error      0.06596306  0.05826087  0.05170976
## mean_per_class_accuracy  0.96586597    0.969475  0.96715206
## mean_per_class_error     0.03413402 0.030525008  0.03284793
## mse                     0.029593127 0.026295785 0.028274098
## r2                       0.99646425   0.9968382   0.9966371
## rmse                     0.17202653  0.16215976  0.16814904
```

As can be seen from the Cross Validation Metrics table above, the final model is expected to have an accuracy of 96.65% on average.

Prediction using Test data

To predict the results from this model, we can use the `h2o.predict` function and check if the expected accuracy holds up against unseen `test` data

```
## Using the RF model for predictions
h2o_nn_yhat_test <- h2o.predict(model.nn.h2o, test_h2o)
```

```
##
  |
  |                                                                      |   0%
  |
  |======================================================================| 100%
```

```
## Converting H2O format into data frame
df_nn_yhat_test <- as.data.frame(h2o_nn_yhat_test)

## Getting test labels
test_labels<-unlist(as.data.frame(test_h2o[,1]))

## Table of predictions
table(test_labels,df_nn_yhat_test[,1])
```

```
##
## test_labels    0    1    2    3    4    5    6    7    8    9
##           0  971    0    0    0    0    3    2    1    3    0
##           1    0 1129    2    1    0    0    0    1    2    0
##           2    3    1 1000    7    1    1    4    8    6    1
##           3    0    0    4  992    0    4    0    3    2    5
##           4    0    0    8    1  950    3    4    4    1   11
##           5    2    0    1   10    1  870    2    1    3    2
##           6    3    2    2    1    4   11  932    0    3    0
##           7    0   11   12    4    2    0    0  987    5    7
##           8    0    0    3   13    2    6    4    3  938    5
##           9    4    4    0    5   12   10    1   10    1  962
```

```
## Accuracy of predictions
Acc<-sum(diag(table(test_labels,df_nn_yhat_test[,1])))/length(test_labels)
print(paste0("Accuracy = ",Acc*100,"%"))
```

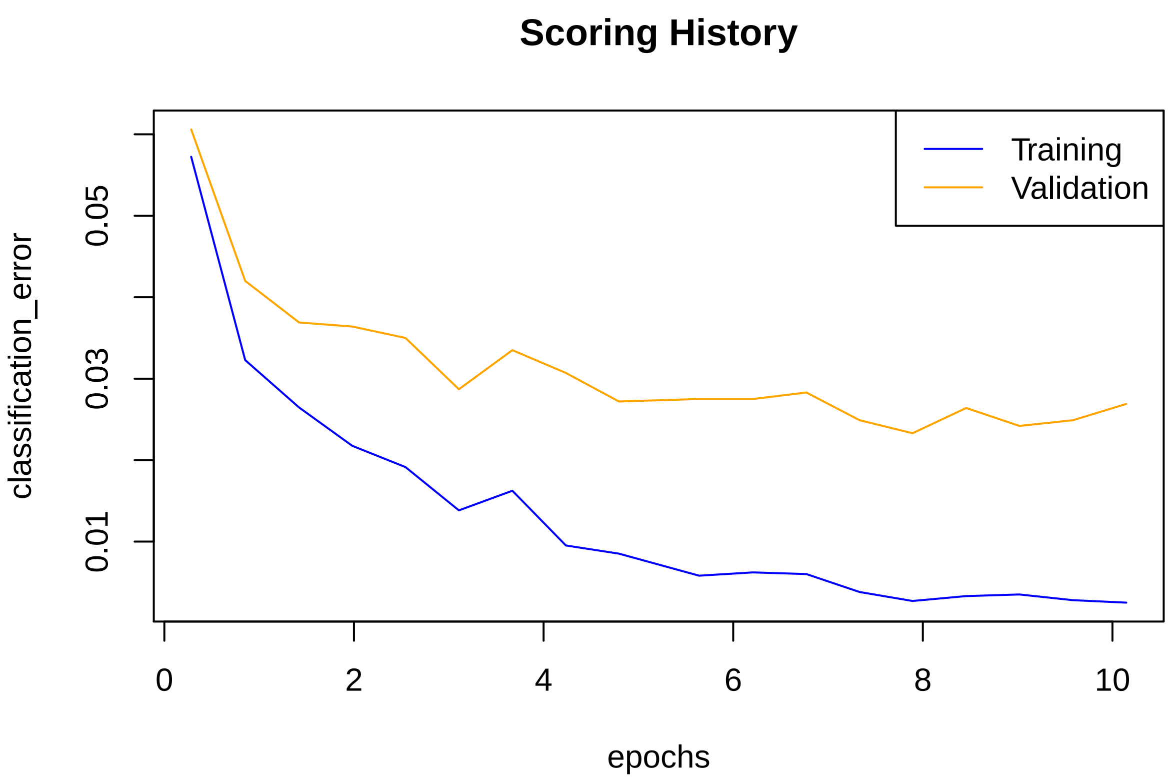
```
## [1] "Accuracy = 97.31%"
```

The `Accuracy` metric from the `test` data above indicates that the model in this case has performed better on our test data than the mean CV metrics predicted. The error on `test` data is 2.69% compared to predicted error of 3.35%. **The world record for error rate on this test dataset is currently 0.83%** It would be a good challenge to modify and train the network to come close and beat the world record!

Finally, H2O based neural networks also allow us to make certain plots

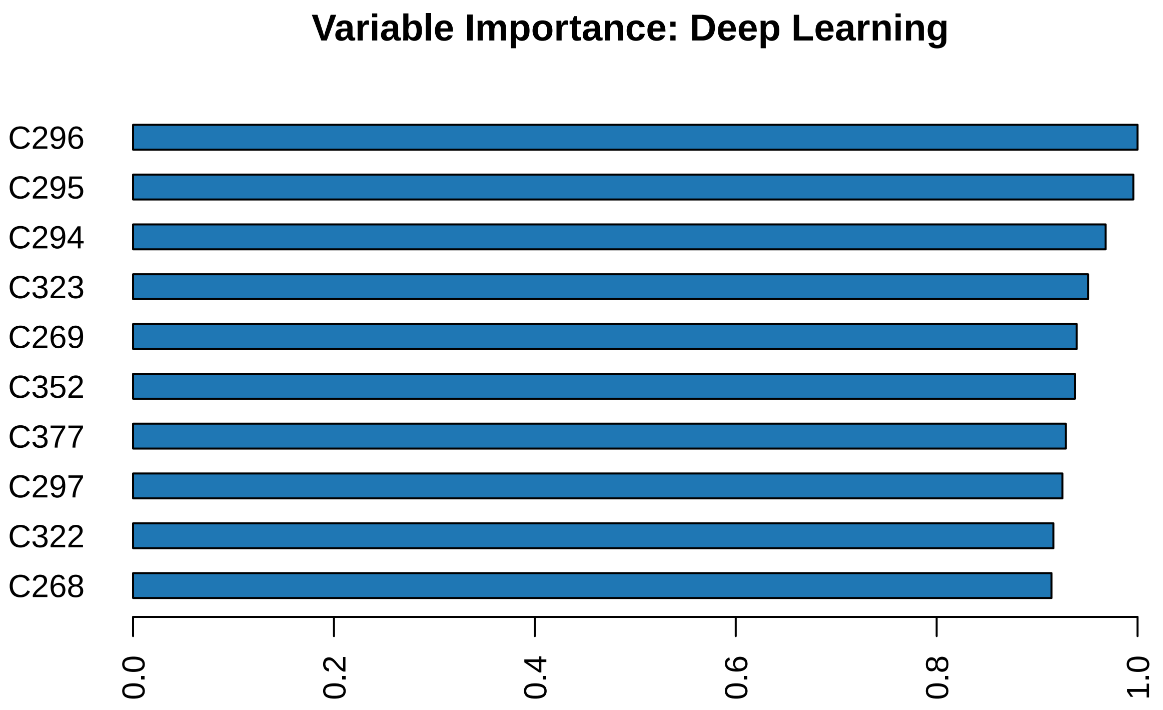
Scoring History

```
plot(model.nn.h2o)
```



Top 10 important variables

```
h2o.varimp_plot(model.nn.h2o,num_of_features = 10)
```



Summary

In this case we looked at classifying a dataset of Handwritten Digits from the MNIST database. This was a multiclass classification problem with 60,000 training images and 10,000 test images.

- We first ran a *Random Forest* model using the `CARET` package. This model had an estimated error of 10.9% but generated an error of 11.38% on our test data. Notably this model was trained only on 1000 training images and it's performance and fit could have been improved with more data and cross validation.
- For our second run, we generated a *Distributed Random Forest* model using the scalable machine learning library provided by `H2O` package. The model error estimate here was 10.52% when it was trained on 1000 training images with a 5 fold cross validation. The error on the test set was also a similar 10.78%.
- For our last run, we generated a *Deep Learning* model using a 3 layer Neural Network constructed again using the scalable machine learning library `H2O`. The model for this run was constructed using all the training data available with a 5 fold cross validation. The estimated model error was 3.35% and the final test set error on this was 2.69%.