Deep Learning

Taylor Shrode

December 13, 2020

Introduction

Deep Learning performs well in a number of diverse problems due to its ability to provide training stability, generalization, and scalability with big data (Candel & LeDell, 2020). The basic framework of multi-layer neural networks are used to accomplish Deep Learning tasks. This multi-layer, feedforward neural networks framework starts with an input layer that matches the feature space (Candel & LeDell, 2020). This is followed by multiple layers of nonlinearity and ends with a linear regression or classification layer to match the output space. H2O follows the model of multi-layer, feedforward neural networks for predictive modeling (Candel & LeDell, 2020).

The data we will be using to apply the H2O's Deep Learning algorithm is from the MNIST database of handwritten digits. This data consists of a training set of 60,000 examples and a test set of 10,000 examples (Lecun et al, n.d.). The files available include:

1. Training set images: *train-images-idx3-ubyte.gz*
2. Training set labels: *train-labels-idx1-ubyte.gz*
3. Test set images: *t10k-images-idx3-ubyte.gz*
4. Test set labels: *t10k-labels-idx1-ubyte.gz*

Each image is a standardized 28^2 pixel greyscale image of a single handwritten digit (Candel & LeDell, 2020). This data can be loaded into R directly from the website URL and then the files can be unzipped. First, we need to load all the necessary libraries into R.

Load Libraries and Data

The packages needed to complete this assignment are loaded below.

```
library(R.utils)
library(h2o)
library(caret)
```

The **R.utils** package is loaded to unzip our *.gz* files, or *gunzip* (Bengtsson, 2015). The **h2o** package is loaded to perform Deep Learning. Finally, the **caret** package is loaded to evaluate our Deep Learning models.

Now, we can load our data into R. To do this, we begin by downloading the data directly from the URL using the **download.file()** function (Dalpiaz, n.d.).

```
download.file("http://yann.lecun.com/exdb/mnist/train-images-idx3-ubyte.gz",
"train-images-idx3-ubyte.gz")
download.file("http://yann.lecun.com/exdb/mnist/train-labels-idx1-ubyte.gz",
"train-labels-idx1-ubyte.gz")
download.file("http://yann.lecun.com/exdb/mnist/t10k-images-idx3-ubyte.gz",
"t10k-images-idx3-ubyte.gz")
download.file("http://yann.lecun.com/exdb/mnist/t10k-labels-idx1-ubyte.gz",
"t10k-labels-idx1-ubyte.gz")
```

Now, we need to unzip the *.gz* files using the **gunzip()** function (Dalpiaz, n.d.).

```
gunzip("train-images-idx3-ubyte.gz", overwrite = TRUE)
gunzip("train-labels-idx1-ubyte.gz", overwrite = TRUE)
gunzip("t10k-images-idx3-ubyte.gz", overwrite = TRUE)
gunzip("t10k-labels-idx1-ubyte.gz", overwrite = TRUE)
```

To load the image files, we can use the function below. This function has been adapted from Hou (2019). The steps found in the function below are as follows (Hou, 2019):

1. Read Magic Number (A constant numerical or text value used to identify a file format)
2. Read Number of Images
3. Read Number of Rows
4. Read Number of Columns
5. Read pixels of every image, each image has **nrow** x **ncol** pixels
6. Store them in a matrix form for easy visualization

```
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
}
```

Now, we can use the function above to load our image files.

```
trainset <- load_image_file("train-images-idx3-ubyte")
testset <- load_image_file("t10k-images-idx3-ubyte")
```

To load our image labels, we can use the function below, which has been adapted from Hou (2019). The steps found in the function below are as follows (Hou, 2019):

1. Read Magic Number
2. Read Number of Labels
3. Read All the Labels

```
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
}
```

We will now load our labels as a factor, and then add them to our **trainset** and **testset** objects (Dalpiaz, n.d.).

```
trainset_labels <- as.factor(load_label_file("train-labels-idx1-ubyte"))
testset_labels <- as.factor(load_label_file("t10k-labels-idx1-ubyte"))


trainset$labels = trainset_labels
testset$labels  = testset_labels
```

To determine the type of object our data is stored in, we can use the **class()** function (Hou, 2019).

```
class(trainset)

## [1] "list"

class(testset)

## [1] "list"
```

To view the length of each element of a list, we can use the **lengths()** function (Hou, 2019).

```
## Length of trainset: 1 47040000 60000

## Length of testset: 1 7840000 10000
```

Similarly, to determine the dimensions of our data, we can use the commands below (Hou, 2019).

```
dim(as.data.frame(trainset))

## [1] 60000   786

dim(as.data.frame(testset))

## [1] 10000   786
```

Next, we can define a function to display the images.

Display Images

First, we need to define a function to visualize the images in our data (O'Connor, n.d.).

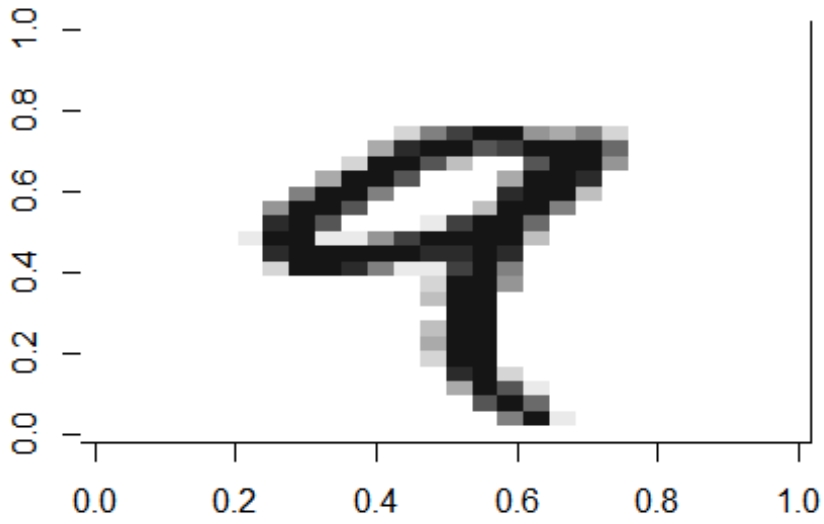
```
show_digit <- function(arr784, col=gray(12:1/12), ...) {
  image(matrix(arr784, nrow=28)[,28:1], col=col, ...)
}
```

Using the function above, we can display the fifth label and it's corresponding image/digit in our **trainset** by using the commands below.

```
trainset$labels[5]

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

show_digit(trainset$x[5,])
```

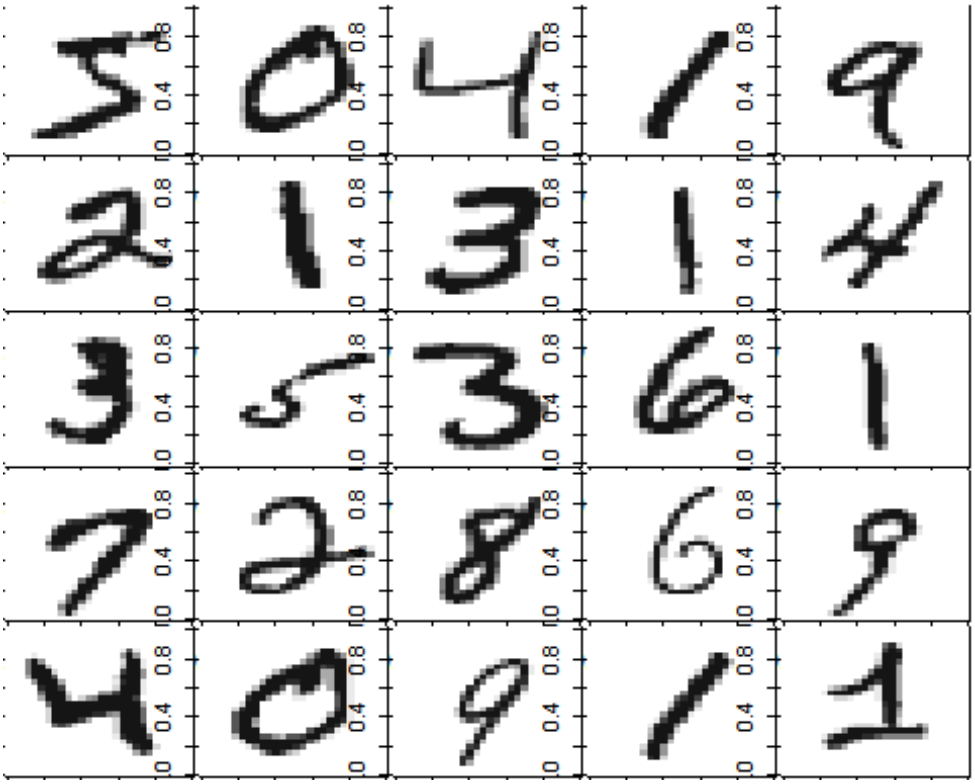


According to the the label output, the image below is the number "9". To display the first 25 labels and images, we can use the commands below (Hou, 2019):

```
label_matrix <- t(matrix(trainset$labels[1:25], 5, 5))
label_matrix

##      [,1] [,2] [,3] [,4] [,5]
## [1,] "5"  "0"  "4"  "1"  "9"
## [2,] "2"  "1"  "3"  "1"  "4"
## [3,] "3"  "5"  "3"  "6"  "1"
## [4,] "7"  "2"  "8"  "6"  "9"
## [5,] "4"  "0"  "9"  "1"  "1"
```

```
par(mfrow=c(5,5))
par(mar=c(0.1,0.1,0.1,0.1))
for(i in 1:25){show_digit(trainset$x[i,])}
```



The output above allows us to easily visualize the images and labels in our **trainset**. Before we continue, we need to reset the **mfrow** parameter.

```
par(mfrow=c(1,1))
```

Deep Learning Model

As stated above, we will be using the **h2o** package to create our Deep Learning models. To begin, we need to initialize the H2O server.

```
h2o.init(nthreads=-1, enable_assertions  = FALSE)
```

```
##  Connection successful!
##
## R is connected to the H2O cluster:
##     H2O cluster uptime:         4 days 4 hours
##     H2O cluster timezone:       America/Denver
##     H2O data parsing timezone:  UTC
##     H2O cluster version:        3.32.0.1
##     H2O cluster version age:    2 months and 5 days
##     H2O cluster name:           H2O_started_from_R_07hoc_iwp664
##     H2O cluster total nodes:    1
##     H2O cluster total memory:   2.37 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
##     H2O Internal Security:      FALSE
##     H2O API Extensions:         Amazon S3, Algos, AutoML, Core V3,
TargetEncoder, Core V4
##     R Version:                  R version 4.0.3 (2020-10-10)
```

The arguments used above include (Initialize and Connect to H2O, n.d.):

1. *nthreads*: Number of threads in the thread pool. This relates very closely to the number of CPUs used. -1 means use all CPUs on the host (Default).
2. *enable_assertions*: A logical value indicating whether H2O should be launched with assertions enabled. Used mainly for error checking and debugging purposes. This value is only used when R starts H2O.

Next, we need to convert our **trainset** and **testset** lists to H2O objects.

```
h2o.no_progress()
h2o_train <- as.h2o(trainset)
h2o_test <- as.h2o(testset)
```

The first $28^2 = 784$ values of each row represent the full image and the final values denotes the digit class (label) (Candel & LeDell, 2020). Before we create our Deep Learning model, we need specify the response and predictor columns.

```
y <- "labels"
x <- setdiff(names(h2o_train),y)

tail(x)

## [1] "x.779" "x.780" "x.781" "x.782" "x.783" "x.784"
```

Above, **y** is our response variable, and **x** contains our predictor columns. Now, we need to encode the response column as categorical for multinomial classification (Candel & LeDell, 2020).

```
h2o_train[,y] <- as.factor(h2o_train[,y])
h2o_test[,y] <- as.factor(h2o_test[,y])
```

Next, we can train a Deep Learning model and validate on our testing data set. The arguments used in the model below include (Deep Learning, 2020):

1. x: Vector containing the names or indices of the predictor variables to use when building the model.
2. y: The column to use as the dependent variable.
3. $training\_frame$: The dataset used to build the model.

4. *distribution*: The distribution (i.e., the loss function). The options are AUTO, bernoulli, multinomial, gaussian, poisson, gamma, laplace, quantile, huber, or tweedie.
    a. Multinomial requires that the response column must be categorical.
5. *activation*: The activation function. Options include: Tanh, Tanh with dropout, Rectifier, Rectifier with dropout, Maxout, Maxout with dropout.
    a. Rectifier and Rectifier with dropout have been known to enhance performance on the MNIST database data (Candel & LeDell, 2020).
6. *hidden*: Specify the hidden layer sizes (e.g., 100,100). The value must be positive. This option defaults to (200,200).
7. *epochs*: Specify the number of times to iterate (stream) the dataset. The value can be a fraction. This option defaults to 10.
8. *input_dropout_ratio*: Specify the input layer dropout ratio to improve generalization. Suggested values are 0.1 or 0.2. Defaults to 0.
9. *l*1: Specify the L1 regularization to add stability and improve generalization; sets the value of many weights to 0 (default).
10. *variable_importance*: Specify whether to compute variable importance.

The H2O Deep Learning model has many parameters, and often it's just the number and sizes of hidden layers, the number of epochs and the activation function and maybe some regularization techniques that need to be changed (Candel, n.d.).

```
h2o.no_progress()
model <- h2o.deeplearning(x = x,
                          y = y,
                          training_frame  = h2o_train,
                          validation_frame = h2o_test,
                          distribution = "multinomial",
                          activation = "RectifierWithDropout",
                          hidden = c(50,50,50),
                          epochs = 10,
                          input_dropout_ratio = 0.2,
                          l1 = 1e-5,
                          variable_importances = TRUE)

## Warning in .h2o.processResponseWarnings(res): Dropping bad and constant
columns: [x.86, x.85, x.84, x.83, x.561, x.760, x.169, x.755, x.756, x.757,
x.758, x.759, x.32, x.31, x.30, x.672, x.112, x.673, x.113, x.674, x.477,
x.700, x.701, x.702, x.1, x.3, x.2, x.5, x.4, x.7, x.23, x.6, x.22, x.9,
x.21, x.8, x.20, x.141, x.142, x.29, x.781, x.28, x.782, x.27, x.783, x.26,
x.784, x.25, x.24, x.12, x.56, x.11, x.55, x.10, x.54, x.53, x.19, x.18,
x.17, n, x.58, x.730, x.57, x.731, x.645, x.646, x.728, x.729].
```

Using our model, we can extract the parameters, examine the scoring process and make predictions on new data.

View Results of Model

First, we can look at the model summary, which examines the performance of the trained model and displays all performance metrics (Candel & LeDell, 2020).

```
model

## Model Details:
## ==============
##
## H2OMultinomialModel: deeplearning
## Model ID:  DeepLearning_model_R_1607552775867_8
## Status of Neuron Layers: predicting labels, 10-class classification,
multinomial distribution, CrossEntropy loss, 41,510 weights/biases, 607.5 KB,
600,000 training samples, mini-batch size 1
##   layer units             type dropout       l1       l2 mean_rate
rate_rms
## 1     1   717            Input 20.00 %       NA       NA        NA
NA
## 2     2    50 RectifierDropout 50.00 % 0.000010 0.000000  0.032859
0.087344
## 3     3    50 RectifierDropout 50.00 % 0.000010 0.000000  0.000292
0.000125
## 4     4    50 RectifierDropout 50.00 % 0.000010 0.000000  0.000542
0.000277
## 5     5    10          Softmax      NA 0.000010 0.000000  0.002966
0.002833
##   momentum mean_weight weight_rms mean_bias bias_rms
## 1       NA          NA         NA        NA       NA
## 2 0.000000    0.049185   0.115816 -0.141996 0.231248
## 3 0.000000   -0.040012   0.135008  0.796043 0.191569
## 4 0.000000   -0.035712   0.141048  0.693208 0.135723
## 5 0.000000   -0.357644   0.828543 -1.630914 0.645160
##
##
## H2OMultinomialMetrics: deeplearning
## ** Reported on training data. **
## ** Metrics reported on temporary training frame with 10011 samples **
##
## Training Set Metrics:
## =====================
##
## MSE: (Extract with `h2o.mse`) 0.05977089
## RMSE: (Extract with `h2o.rmse`) 0.2444809
## Logloss: (Extract with `h2o.logloss`) 0.2262201
## Mean Per-Class Error: 0.06520146
## Confusion Matrix: Extract with `h2o.confusionMatrix(<model>,train =
TRUE)`)
## =========================================================================
## Confusion Matrix: Row labels: Actual class; Column labels: Predicted class
```

```
##           0    1    2    3   4   5   6   7   8   9  Error           Rate
## 0      1001    0    5    4   2   4   6   0   4   1 0.0253 =   26 / 1,027
## 1         0 1059   17    5   2   2   2   4  12   1 0.0408 =   45 / 1,104
## 2         3    2  927   21   9   2  16   7   8   2 0.0702 =     70 / 997
## 3         0    0   18  931   0  27   2  11  10   4 0.0718 =   72 / 1,003
## 4         2    2    8    2 907   2   8   0   5  24 0.0552 =     53 / 960
## 5         5    0    5   28   5 834  15   1   6   5 0.0774 =     70 / 904
## 6        11    1    8    1   0  10 937   1   4   0 0.0370 =     36 / 973
## 7         4    3   20   11   6   1   1 956   1  38 0.0817 =   85 / 1,041
## 8         5   16   11   14   8  22   8   1 895   5 0.0914 =     90 / 985
## 9         4    1    1   15  40  14   1  16  11 914 0.1013 =  103 / 1,017
## Totals 1035 1084 1020 1032 979 918 996 997 956 994 0.0649 = 650 / 10,011
##
## Hit Ratio Table: Extract with `h2o.hit_ratio_table(<model>,train = TRUE)`
## =======================================================================
## Top-10 Hit Ratios:
##     k hit_ratio
## 1   1  0.935071
## 2   2  0.973329
## 3   3  0.984817
## 4   4  0.991210
## 5   5  0.993407
## 6   6  0.995605
## 7   7  0.997303
## 8   8  0.998402
## 9   9  0.999301
## 10 10  1.000000
##
##
## H2OMultinomialMetrics: deeplearning
## ** Reported on validation data. **
## ** Metrics reported on full validation frame **
##
## Validation Set Metrics:
## =====================
##
## Extract validation frame with `h2o.getFrame("RTMP_sid_8a62_6")`
## MSE: (Extract with `h2o.mse`) 0.06152417
## RMSE: (Extract with `h2o.rmse`) 0.2480407
## Logloss: (Extract with `h2o.logloss`) 0.2391414
## Mean Per-Class Error: 0.0673272
## Confusion Matrix: Extract with `h2o.confusionMatrix(<model>,valid =
TRUE)`)
## =========================================================================
## Confusion Matrix: Row labels: Actual class; Column labels: Predicted class
##           0    1    2    3    4   5   6   7   8   9  Error           Rate
## 0       956    0    2    3    1   6  10   1   1   0 0.0245 =     24 / 980
## 1         0 1106    7    5    1   1   3   1  11   0 0.0256 =   29 / 1,135
## 2        10    0  961   14    7   4   8   9  17   2 0.0688 =   71 / 1,032
## 3         0    0   20  941    0  23   1  13   9   3 0.0683 =   69 / 1,010
```

```
## 4         1    1    7    2  937   1  12   3   3  15 0.0458 =     45 / 982
## 5         7    1    5   40    7 797   9   4  16   6 0.1065 =     95 / 892
## 6        11    2    9    0    4   6 923   0   3   0 0.0365 =     35 / 958
## 7         2    8   30    7    2   0   1 938   0  40 0.0875 =    90 / 1,028
## 8         7    4    9   14   10  33  10   8 874   5 0.1027 =    100 / 974
## 9         9    5    2   13   48  12   0   9  10 901 0.1070 =  108 / 1,009
## Totals 1003 1127 1052 1039 1017 883 977 986 944 972 0.0666 = 666 / 10,000
##
## Hit Ratio Table: Extract with `h2o.hit_ratio_table(<model>,valid = TRUE)`
## =======================================================================
## Top-10 Hit Ratios:
##     k hit_ratio
## 1   1  0.933400
## 2   2  0.971400
## 3   3  0.982900
## 4   4  0.990100
## 5   5  0.994300
## 6   6  0.997100
## 7   7  0.998600
## 8   8  0.999300
## 9   9  0.999800
## 10 10  1.000000
```

To view specified parameters of the Deep Learning model, we can use the **model@parameters** command (Candel & LeDell, 2020). The **h2o.performance(model)** function "returns all pre-computed performance metrics for the training/validation set", which can be found in the summary output above under *Training Set Metrics* (Candel & LeDell, 2020). Using the command **h2o.performance(model, valid = TRUE)** returns the validation metrics, which is located under the *Validation Set Metrics* section in the summary output above.

To get the mean square error (MSE), we can use the **h2o.mse()** function. "The MSE metric measures the average of the squares of the errors or deviations. MSE takes the distances from the points to the regression line (these distances are the "errors") and squaring them to remove any negative signs (Performance and Prediction, n.d.)."

```
h2o.mse(model, train = TRUE)
```

```
## [1] 0.05977089
```

```
h2o.mse(model, valid = TRUE)
```

```
## [1] 0.06152417
```

The output above suggests that our model has an error rate of ≈ 6.1%. This then suggests that our model has an accuracy rate of ≈ 93.9%. We can confirm this with a confusion matrix, which we will create in the next section.

To generate the variable importances, we can use the **h2o.varimp()** function. "This feature allows us to view the absolute and relative predictive strength of each feature in the prediction task (Candel & LeDell, 2020)." Th

```
h2o.varimp(model)
```

```
## Variable Importances:
##   variable relative_importance scaled_importance percentage
## 1    x.590            1.000000          1.000000   0.003332
## 2    x.534            0.977569          0.977569   0.003257
## 3    x.143            0.957341          0.957341   0.003190
## 4    x.198            0.954711          0.954711   0.003181
## 5     x.36            0.953686          0.953686   0.003178
##
## ---
##     variable relative_importance scaled_importance percentage
## 712    x.313            0.185208          0.185208   0.000617
## 713    x.679            0.184548          0.184548   0.000615
## 714    x.285            0.183385          0.183385   0.000611
## 715    x.136            0.178898          0.178898   0.000596
## 716    x.134            0.165955          0.165955   0.000553
## 717    x.690            0.162303          0.162303   0.000541
```

The output above shows the top 5 important variables and the least important variables.

Now, using this model, we can use the **h2o.predict()** function to compute and store predictions on new data (Candel & LeDell, 2020).

View Predictions

As stated above, we can use the **h2o.predict()** function to make predictions using out model. To view the predictions, we can use the **head()** function.

```
pred <- h2o.predict(model, newdata = h2o_test)
head(pred)
```

```
##   predict           p0           p1           p2           p3           p4
## 1       7 2.736164e-04 5.914795e-06 1.928436e-03 1.597784e-03 6.785902e-05
## 2       2 2.215879e-03 7.817808e-03 7.574309e-01 1.038842e-01 2.529168e-03
## 3       1 1.252809e-06 9.988615e-01 7.201478e-05 2.668904e-05 1.618585e-04
## 4       0 9.997911e-01 2.190379e-08 5.127281e-05 1.123193e-06 2.874733e-07
## 5       4 4.375919e-05 5.021726e-05 3.230510e-04 1.191457e-05 9.774495e-01
## 6       1 9.659590e-07 9.989099e-01 6.506568e-05 2.634763e-05 1.520974e-04
##             p5           p6           p7           p8           p9
## 1 2.945361e-04 1.429446e-05 9.865363e-01 6.895781e-05 9.212274e-03
## 2 9.361011e-03 6.078492e-02 2.358355e-02 2.937314e-02 3.019399e-03
## 3 1.045071e-04 2.458213e-05 4.042398e-06 7.393178e-04 4.205403e-06
## 4 1.151793e-04 2.429604e-05 2.156041e-07 1.492926e-05 1.575258e-06
## 5 3.582411e-04 3.183312e-04 1.393430e-04 1.005394e-03 2.030021e-02
## 6 8.550216e-05 1.820877e-05 3.703191e-06 7.341458e-04 4.074732e-06
```

From the output above, we can see the digit that the model predicted for each observation, and the probabilities that the particular observation was a different digit. For example, for the first observation, the model predicted that the image was a "7". The probability that the image is a "7" is 0.9976. The probability that the image is a "1" is 3.3027e-07.

Now, to view the confusion matrix for our model, we can use the **h2o.confusionMatrix()** function with the model and testing set as arguments (Zalando's images classification using H2O with R, 2017).

```
h2o.confusionMatrix(model, h2o_test)
```

```
## Confusion Matrix: Row labels: Actual class; Column labels: Predicted class
##           0    1    2    3    4   5   6   7   8   9  Error           Rate
## 0       956    0    2    3    1   6  10   1   1   0 0.0245 =     24 / 980
## 1         0 1106    7    5    1   1   3   1  11   0 0.0256 =   29 / 1,135
## 2        10    0  961   14    7   4   8   9  17   2 0.0688 =   71 / 1,032
## 3         0    0   20  941    0  23   1  13   9   3 0.0683 =   69 / 1,010
## 4         1    1    7    2  937   1  12   3   3  15 0.0458 =     45 / 982
## 5         7    1    5   40    7 797   9   4  16   6 0.1065 =     95 / 892
## 6        11    2    9    0    4   6 923   0   3   0 0.0365 =     35 / 958
## 7         2    8   30    7    2   0   1 938   0  40 0.0875 =   90 / 1,028
## 8         7    4    9   14   10  33  10   8 874   5 0.1027 =    100 / 974
## 9         9    5    2   13   48  12   0   9  10 901 0.1070 =  108 / 1,009
## Totals 1003 1127 1052 1039 1017 883 977 986 944 972 0.0666 = 666 / 10,000
```

The output above suggests that the error rate for our model is ≈ 6.75%. This confirms our MSE values that was calculated above. Thus, the accuracy rate is 93.25%.

Another method to creating a confusion matrix is to use the **caret** package and look at the overall accuracy and other statistic values (Charleshsliao, 2017).

```
pred_results <- as.data.frame(pred[,1])
confusionMatrix(unlist(pred_results), testset$labels)$overall
```

```
##       Accuracy          Kappa  AccuracyLower  AccuracyUpper   AccuracyNull
##      0.9334000      0.9259727      0.9283358      0.9382100      0.1135000
## AccuracyPValue  McnemarPValue
##      0.0000000            NaN
```

Notice, this output also confirms our accuracy rate of ≈ 93.25%. This output also provides the Cohen's Kappa Statistic. Recall from the Week 2 Assignment, this value is used to measure the agreement of two raters or methods rating on categorical scales. Kappa values range from 0 to a maximum of 1, which indicates a perfect agreement between the model's predictions and the true values. The value above indicates very good agreement/near perfect agreement.

Cartesian Hyper-Parameter Grid Search

For the final portion of this assignment, we will be using a Cartesian grid search for model comparison. "Grid search provides more subtle insights into the model tuning and selection

process by inspecting and comparing our trained models after the grid search process is complete (Candell & LeDell, 2020)." To perform a Cartesian grid search, we need to create the hyperparameters. For this, we will create a list of control parameters for the number of hidden layer sizes and the L1 regularization.

```
hidden_opt <- list(c(50,50), c(100,100), c(200,200))
l1_opt <- c(1e-4, 1e-5)
hyper_params <- list(hidden = hidden_opt, l1 = l1_opt)
```

Now, we use the **h2o.grid()** function to launch a grid search. The arguments we will be using below include (H2o. Grid function, n.d.):

1. *algorithm*: Name of algorithm to use in grid search (gbm, randomForest, kmeans, glm, deeplearning, naivebayes, pca).
2. *grid_id*: ID for resulting grid search.
3. *hyper_params*: List of lists of hyper parameters
4. *score_interval*, *stopping_rounds*, *stopping_tolerance*, *stopping_metric*: List of control parameters for smarter hyperparameter search.

```
grid_search <- h2o.grid(algorithm = "deeplearning",
                        grid_id = "mygrid",
                        x = x,
                        y = y,
                        hyper_params = hyper_params,
                        training_frame  = h2o_train,
                        validation_frame = h2o_test,
                        distribution = "multinomial",
                        score_interval = 2,
                        epochs = 10,
                        stopping_rounds = 3,
                        stopping_tolerance = 0.05,
                        stopping_metric = "misclassification")
grid_search
```

```
## H2O Grid Details
## ================
##
## Grid ID: mygrid
## Used hyper parameters:
##   -  hidden
##   -  l1
## Number of models: 24
## Number of failed models: 0
##
## Hyper-Parameter Search Summary: ordered by increasing logloss
##       hidden     l1       model_ids             logloss
## 1 [200, 200] 1.0E-4  mygrid_model_3 0.07434812550851004
## 2 [200, 200] 1.0E-4 mygrid_model_15  0.0803907671587165
## 3 [200, 200] 1.0E-4 mygrid_model_21 0.08597297674501649
## 4 [100, 100] 1.0E-4 mygrid_model_14 0.08802445190969467
```

```
## 5 [100, 100] 1.0E-4 mygrid_model_20 0.08841444593521085
##
## ---
##        hidden     l1       model_ids             logloss
## 19 [100, 100] 1.0E-5 mygrid_model_17 0.14211806701397514
## 20 [100, 100] 1.0E-5 mygrid_model_23 0.14342010895373078
## 21   [50, 50] 1.0E-5 mygrid_model_16 0.15662723158731137
## 22   [50, 50] 1.0E-5  mygrid_model_4  0.1626738685423568
## 23   [50, 50] 1.0E-5 mygrid_model_10 0.16508876887424984
## 24   [50, 50] 1.0E-5 mygrid_model_22  0.1728066514656209
```

Using the function below, we can print out the test MSE for all of the models and their model ID's (Candel & LeDell, 2020).

```
for (model_id in grid_search@model_ids) {
  mse <- h2o.mse(h2o.getModel(model_id), valid = TRUE)
  print(sprintf("Test set MSE: %f", mse))
  print(sprintf("Model ID: %s", model_id))
  }
```

```
## [1] "Test set MSE: 0.019274"
## [1] "Model ID: mygrid_model_3"
## [1] "Test set MSE: 0.019676"
## [1] "Model ID: mygrid_model_15"
## [1] "Test set MSE: 0.021812"
## [1] "Model ID: mygrid_model_21"
## [1] "Test set MSE: 0.020904"
## [1] "Model ID: mygrid_model_14"
## [1] "Test set MSE: 0.021727"
## [1] "Model ID: mygrid_model_20"
## [1] "Test set MSE: 0.022561"
## [1] "Model ID: mygrid_model_2"
## [1] "Test set MSE: 0.022772"
## [1] "Model ID: mygrid_model_8"
## [1] "Test set MSE: 0.018032"
## [1] "Model ID: mygrid_model_12"
## [1] "Test set MSE: 0.024475"
## [1] "Model ID: mygrid_model_9"
## [1] "Test set MSE: 0.024623"
## [1] "Model ID: mygrid_model_7"
## [1] "Test set MSE: 0.024521"
## [1] "Model ID: mygrid_model_1"
## [1] "Test set MSE: 0.020091"
## [1] "Model ID: mygrid_model_24"
## [1] "Test set MSE: 0.026184"
## [1] "Model ID: mygrid_model_13"
## [1] "Test set MSE: 0.021792"
## [1] "Model ID: mygrid_model_5"
## [1] "Test set MSE: 0.026478"
## [1] "Model ID: mygrid_model_19"
```

```
## [1] "Test set MSE: 0.020185"
## [1] "Model ID: mygrid_model_6"
## [1] "Test set MSE: 0.021728"
## [1] "Model ID: mygrid_model_18"
## [1] "Test set MSE: 0.022885"
## [1] "Model ID: mygrid_model_11"
## [1] "Test set MSE: 0.025376"
## [1] "Model ID: mygrid_model_17"
## [1] "Test set MSE: 0.025036"
## [1] "Model ID: mygrid_model_23"
## [1] "Test set MSE: 0.028468"
## [1] "Model ID: mygrid_model_16"
## [1] "Test set MSE: 0.029359"
## [1] "Model ID: mygrid_model_4"
## [1] "Test set MSE: 0.028007"
## [1] "Model ID: mygrid_model_10"
## [1] "Test set MSE: 0.028386"
## [1] "Model ID: mygrid_model_22"
```

To get the model with the lowest MSE, we can use the commands below because the grid search automatically sorts the models by MSE (Candel, n.d.).

```
best_model <- grid_search@model_ids[[1]]
h2o.mse(h2o.getModel(best_model))

## [1] 0.0103964
```

Using the "best model", we can make predictions. Similar to above.

```
pred_search <- h2o.predict(h2o.getModel(best_model), newdata = h2o_test)
head(pred_search)

##   predict           p0           p1           p2           p3           p4
## 1       7 4.128290e-09 1.373566e-08 8.044667e-08 3.690622e-07 2.882428e-10
## 2       2 5.328029e-07 4.191872e-07 9.997953e-01 7.472790e-08 2.800911e-12
## 3       1 8.251455e-06 9.987975e-01 1.440151e-04 3.580470e-06 4.696438e-05
## 4       0 9.999414e-01 1.494900e-06 8.341984e-06 1.493985e-06 5.202007e-08
## 5       4 3.019286e-06 6.470066e-07 3.115631e-07 8.710120e-11 9.980734e-01
## 6       1 2.023899e-06 9.997637e-01 2.636763e-06 2.565462e-07 2.884491e-05
##             p5           p6           p7           p8           p9
## 1 6.175897e-09 4.844054e-10 9.999961e-01 6.994240e-09 3.452315e-06
## 2 2.189492e-08 1.344172e-06 2.198129e-15 2.023220e-04 3.909108e-12
## 3 2.558474e-06 4.247720e-04 2.208426e-04 3.509739e-04 5.166662e-07
## 4 2.050785e-06 2.893183e-06 3.434190e-05 2.632551e-08 7.952373e-06
## 5 1.375254e-09 1.051897e-06 4.235667e-05 6.302066e-08 1.879143e-03
## 6 4.206026e-08 2.475412e-05 1.475798e-04 3.011120e-05 8.309110e-08
```

Next, we view the confusion matrix for our model.

```
h2o.confusionMatrix(h2o.getModel(best_model), h2o_test)
```

```
## Confusion Matrix: Row labels: Actual class; Column labels: Predicted class
##          0    1    2    3   4   5   6    7   8    9  Error           Rate
## 0      969    0    1    2   1   1   0    1   1    4 0.0112 =     11 / 980
## 1        0 1124    2    2   0   0   2    0   5    0 0.0097 =   11 / 1,135
## 2        3    1 1006    3   3   0   0    9   7    0 0.0252 =   26 / 1,032
## 3        0    0    2  992   0   4   0    4   8    0 0.0178 =   18 / 1,010
## 4        0    1    3    1 947   0   6    3   5   16 0.0356 =     35 / 982
## 5        3    0    0   13   0 862   5    2   5    2 0.0336 =     30 / 892
## 6        5    2    0    0  13   3 930    0   5    0 0.0292 =     28 / 958
## 7        1    4    8    5   0   0   0 1000   2    8 0.0272 =   28 / 1,028
## 8        3    0    2    6   2   3   0    5 950    3 0.0246 =     24 / 974
## 9        3    1    0    4   6   1   0    3   3  988 0.0208 =   21 / 1,009
## Totals 987 1133 1024 1028 972 874 943 1027 991 1021 0.0232 = 232 / 10,000
```

Notice, our model's error rate went down to ≈ 2.3%. To print the accuracy rate, we can use the command below.

```
pred_search_results <- as.data.frame(pred_search[,1])
confusionMatrix(unlist(pred_search_results), testset$labels)$overall
```

```
##       Accuracy          Kappa  AccuracyLower  AccuracyUpper   AccuracyNull
##      0.9768000      0.9742116      0.9736569      0.9796610      0.1135000
## AccuracyPValue  McnemarPValue
##      0.0000000            NaN
```

Using the Cartesian grid search method and selecting the "best model", our accuracy increased from ≈ 93.33% to ≈ 97.683%.

Conclusion

For this assignment, we used H2O's Deep Learning algorithm for classification of handwritten numerals. The data used was from the MNIST database that consisted of of 60,000 training images and 10,000 test images. To load our data, we used the **R.utils** package to "gunzip" our files. For our first Deep Learning model, we mostly used the default settings as our parameters. After computing and storing our predictions made on our testing data, the confusion matrix showed our model had an accuracy rate of ≈ 93%. To improve the performance of our model, we performed a Cartesian grid search which allowed us to train several models using combinations of our specified parameters. Choosing the model with the lowest MSE value and using this model to make predictions, the confusion matrix showed that this model had an accuracy rate of ≈ 97%. There are many other parameters that could be adjusted and applied. However, without distortions, convolutions, or other advanced image processing techniques, the best-ever published test set error for the MNIST dataset is 0.83% by Microsoft (Candel & LeDell, 2020). This was accomplished after training for 8,000 epochs on ten nodes, which took about ten hours. Typically, the default values will result in good performance for most problems (Candel & LeDell, 2020).

Resources

Bengtsson, H. (2015, February 11). [R] How to unzip a .gz file. https://stat.ethz.ch/pipermail/r-help/2015-February/425709.html

Candel , A., & LeDell, E. (2020). Deep Learning with H2O. http://h2o-release.s3.amazonaws.com/h2o/rel-zermelo/2/docs-website/h2o-docs/booklets/DeepLearningBooklet.pdf

Candel, A. (n.d.). Classification and Regression with H2O Deep Learning. GitHub. Retrieved December 13, 2020, from https://github.com/h2oai/h2o-world-2014-training

Charleshsliao. (2017, April 15). A h2o fnn model for mnist. Charles' Hodgepodge. https://charleshsliao.wordpress.com/2017/04/15/a-h2o-fnn-model-for-mnist/

Dalpiaz, D. (n.d.). Load the MNIST handwritten digits dataset into R as a tidy data frame. Gist. Retrieved December 13, 2020, from https://gist.github.com/daviddalpiaz/ae62ae5ccd0bada4b9acd6dbc9008706

Deep Learning . (2020, November 17). http://docs.h2o.ai/h2o/latest-stable/h2o-docs/data-science/deep-learning.html

H2o. Grid function. (n.d.). Retrieved December 13, 2020, from https://www.rdocumentation.org/packages/h2o/versions/3.32.0.1/topics/h2o.grid

Hou, J. (2019, February 7). Load the mnist dataset. https://rpubs.com/JanpuHou/465274

Initialize and Connect to H2O. (n.d.). Retrieved December 13, 2020, from http://docs.h2o.ai/h2o/latest-stable/h2o-r/docs/reference/h2o.init.html

Lecun, Y., Cortes, C., & Burges, C. (n.d.). Mnist handwritten digit database. Retrieved December 13, 2020, from http://yann.lecun.com/exdb/mnist/

O'Connor, B. (n.d.). Load the mnist data set in r. Gist. Retrieved December 13, 2020, from https://gist.github.com/brendano/39760

Performance and Prediction. (n.d.). Retrieved December 14, 2020, from http://docs.h2o.ai/h2o/latest-stable/h2o-docs/performance-and-prediction.html

Zalando's images classification using H2O with R. (2017, September 12). Appsilon | End To End Data Science Solutions. https://appsilon.com/zalandos-images-classification-using-h2o-with-r/