Machine Learning for Finance Applications - TensorFlow for R

Brian Clark

2020

Contents

1 Introduction

This document provides an overview of how to install and run TensorFlow using R. Other options are available using Python.

Start by going to https://tensorflow.rstudio.com/installation/ which is the TensorFLor for R installation guide. As described in teh documentation, the installation will use the R `install_tensorflow()` function.

1.1 Install TensorFlow

Install the tensorflow R package:

```
rm(list=ls())
# install.packages("tensorflow")  # Uncomment the first time you run
```

Next, you will need a working version of Anaconda if you are using Windows. Go to https://www.anaconda.com/distribution/ to install. I installed the Python 3.7 version on my laptop.

Next call the library and install tensorflow using the R wrapper. Note that I answered "Y" when prompted if I wanted to install Miniconda.

```
library("tensorflow")
```

```
## Warning: package 'tensorflow' was built under R version 3.6.3
```

```
# install_tensorflow() # Uncomment the first time you run
```

Confirm that it succeeded...

```
tf$constant("Hello TensorFLow!")
```

```
## tf.Tensor(b'Hello TensorFLow!', shape=(), dtype=string)
```

Next, we will also need the `keras` package,

```
# install.packages("keras") # Uncomment the first time you run
```

1.2 Using TensorFlow

Next, we will go through a few of the examples on the tutorials tab https://tensorflow.rstudio.com/tutorials/ starting with the beginners link.

We will first follow the example provided by R Studio and then try a similar finance application.

1.2.1 Example 1 - MNIST Dataset (Images)

The first example uses a dataset containing image data. The data is structured such that each image is a graph of 28x28 pixels and teh color is given by an integer between 0 and 255. They are greyscale images.

First download the data.

```
library("keras")
```

```
## Warning: package 'keras' was built under R version 3.6.3
```

```
mnist <- dataset_mnist()
```

Let's look at one of the images.

```
tmp <- mnist$train$x[1,,]
# View(tmp)
```

The above is the first training example. The integers give the color (0 is white and 255 is black). We want to first convert the data to floats between 0 and 1 by dividing by 255.

```
mnist$train$x <- mnist$train$x/255
mnist$test$x <- mnist$test$x/255
```

The next step is to define a Keras model.

```
model <- keras_model_sequential() %>%
  layer_flatten(input_shape = c(28, 28)) %>%
  layer_dense(units = 128, activation = "relu") %>%
  layer_dropout(0.2) %>%
  layer_dense(10, activation = "softmax")

summary(model)
```

```
## Model: "sequential"
## _________________________________________________________________________
## Layer (type)                    Output Shape                  Param #
## =========================================================================
## flatten (Flatten)               (None, 784)                   0
## _________________________________________________________________________
## dense (Dense)                   (None, 128)                   100480
## _________________________________________________________________________
## dropout (Dropout)               (None, 128)                   0
## _________________________________________________________________________
## dense_1 (Dense)                 (None, 10)                    1290
## =========================================================================
## Total params: 101,770
## Trainable params: 101,770
## Non-trainable params: 0
## _________________________________________________________________________
```

A few notes on the above model.

1. **layer_flatten()** forces the input layer which is a $28x28$ pixel image into a single vector of length $28^2 = 784$.
2. **layer_dense()** adds a densely connected layer to a NN model. Recall that densely connected means that each node in layer L is connected to each node in layer $L - 1$.
   (a) **units** is the dimension of the layer, in this case 128 means there are 128 nodes in this layer.
   (b) **activation** defines the activation function. Activtion can be one of the following: relu, elu, softmax, selu, softplus, softsign, tanh, sigmoid, hard_sigmoid, exponential, linear. See https://keras.io/activations/ for a full description.
3. **layer_dropout()** defines teh dropout rate (between 0 and 1) to prevent over-fitting.
4. **layer_dense()** on the last line creates a new layer with 10 nodes Think of this as the classes (i.e., we will classify images into 10 categories).

The next step is to compile the model. Importantly, this step involves selecting the loss function, optimizer, and objective metric.

```
model %>%
  compile(
    loss = "sparse_categorical_crossentropy",
    optimizer = "adam",
```

```
    metrics = "accuracy"
)
```

A few notes:

1. **loss** is the loss function you want to use. Possible loss functions can be viewed at https://keras.io/losses/ and include:
    - mean_squared_error
    - mean_absolute_error
    - mean_absolute_percentage_error
    - mean_squared_logarithmic_error
    - squared_hinge
    - hinge
    - categorical_hinge
    - logcosh
    - huber_loss
    - categorical_crossentropy
    - sparse_categorical_crossentropy
    - binary_crossentropy
    - kullback_leibler_divergence
    - poisson
    - cosine_proximity
    - is_categorical_crossentropy
2. **optimizer** is the optimizer you are using and can be viewed at https://keras.io/optimizers/ and include:
    - SGD
    - RMSprop
    - Adagrad
    - Adadelta
    - Adam
    - Adamax
    - Nadam
3. **metrics** is the performance metric you are using and can be viewed at https://keras.io/metrics/ and include:
    - accuracy
    - binary_accuracy
    - categorical_accuracy
    - sparse_categorical_accuracy
    - top_k_categorical_accuracy
    - sparse_top_k_categorical_accuracy

- cosine_proximity
- clone_metric
- clone_metrics
- Custom metrics (you can define your own performance metric)

Finally, we can fit our model to the data.

```
model %>%
  fit(
    x = mnist$train$x, y = mnist$train$y,
    epochs = 5,
    validation_split = 0.3,
    verbose = 2
  )
```

Note that we need to specify the training features and labels data, x and y. The other inputs are:

1. **epochs** is the number of epochs (recall that an epoch is one full loop thorugh the data in terms of the optimization - e.g., 42K SGD iterations would be an epoch because there are 42K observations in the data)
2. **validation_split** is the test/train validation split.
3. **verbose** controls the level of output.

Finally, we can predict the outcome based on the test data.

```
model %>%
  evaluate(mnist$test$x, mnist$test$y, verbose = 0)
```

```
## $loss
## [1] 0.09078996
##
## $accuracy
## [1] 0.9733
```

Note that the accuracy is about 97.5%

To save the model, you must do so explicitly.

```
save_model_tf(object = model, filepath = "model")
```

To reload the model, you could

```
reloaded_model <- load_model_tf("model")

reloaded_model %>%
  evaluate(mnist$test$x, mnist$test$y, verbose = 0)
```

```
## $loss
## [1] 0.09079025
##
## $accuracy
## [1] 0.9733
```

```
y.hat <- predict(reloaded_model, mnist$test$x)
```

Note that the reloaded model results are exactly the same as the original model.

1.2.2 Example 2 - Regression

This example follows another example from teh Rstudio-TensorFlow website https://tensorflow.rstudio.com/tutorials/beginners/basic-ml/tutorial_basic_regression/. The goal is to predict house prices. We will use the Boston Housing data.

```
library("keras")
library("tfdatasets")
```

```
## Warning: package 'tfdatasets' was built under R version 3.6.3
```

Download the Boston Housing Prices dataset:

```
bh <- dataset_boston_housing()

train.x <- bh$train$x
train.y <- bh$train$y
test.x <- bh$test$x
test.y <- bh$test$y

print(dim(train.x))
```

```
## [1] 404  13
```

```
print(dim(test.x))
```

```
## [1] 102  13
```

There are 13 features in the data and 506 total observations - 404 training and 202 test. The features are as follows:

1. Per capital crime rate
2. The proportion of residential land zoned for lots over 25,000 square feet.
3. The proportion of non-retail business acres per town.
4. Charles River dummy variable (= 1 if tract bounds river; 0 otherwise).
5. Nitric oxides concentration (parts per 10 million).
6. The average number of rooms per dwelling.
7. The proportion of owner-occupied units built before 1940.
8. Weighted distances to five Boston employment centers.
9. Index of accessibility to radial highways.
10. Full-value property-tax rate per $10,000.
11. Pupil-teacher ratio by town.
12. $1000 * (Bk - 0.63)^2$ where Bk is the proportion of Black people by town.
13. Percentage lower status of the population.

Note that the scale of the variables is not standardized and the data s not labeled. We can add some

```
library(dplyr)
```

```
##
## Attaching package: 'dplyr'
```

```
## The following objects are masked from 'package:stats':
##
##     filter, lag

## The following objects are masked from 'package:base':
##
##     intersect, setdiff, setequal, union
```

```
column_names <- c('CRIM', 'ZN', 'INDUS', 'CHAS', 'NOX', 'RM', 'AGE',
                  'DIS', 'RAD', 'TAX', 'PTRATIO', 'B', 'LSTAT')

train_df <- train.x %>%
  as_tibble(.name_repair = "minimal") %>%
  setNames(column_names) %>%
  mutate(label = train.y)

test_df <- test.x%>%
  as_tibble(.name_repair = "minimal") %>%
  setNames(column_names) %>%
  mutate(label = test.y)
```

Note the above code is following the online example. We could have done something very similar by simply creating data frames out of the training and test data.

```
train.df <- as.data.frame(cbind(train.x,train.y))
names(train.df) <- c(column_names,"label")
head(train.df)
```

```
##      CRIM   ZN INDUS CHAS   NOX    RM   AGE    DIS RAD TAX PTRATIO      B
## 1 1.23247  0.0  8.14    0 0.538 6.142  91.7 3.9769   4 307    21.0 396.90
## 2 0.02177 82.5  2.03    0 0.415 7.610  15.7 6.2700   2 348    14.7 395.38
## 3 4.89822  0.0 18.10    0 0.631 4.970 100.0 1.3325  24 666    20.2 375.52
## 4 0.03961  0.0  5.19    0 0.515 6.037  34.5 5.9853   5 224    20.2 396.90
## 5 3.69311  0.0 18.10    0 0.713 6.376  88.4 2.5671  24 666    20.2 391.43
## 6 0.28392  0.0  7.38    0 0.493 5.708  74.3 4.7211   5 287    19.6 391.13
##   LSTAT label
## 1 18.72  15.2
## 2  3.11  42.3
## 3  3.26  50.0
## 4  8.01  21.1
## 5 14.65  17.7
## 6 11.74  18.5
```

```
test.df <- as.data.frame(cbind(test.x,test.y))
names(test.df) <- c(column_names,"label")
head(test.df)
```

```
##       CRIM ZN INDUS CHAS   NOX    RM   AGE    DIS RAD TAX PTRATIO      B
## 1 18.08460  0 18.10    0 0.679 6.434 100.0 1.8347  24 666    20.2  27.25
## 2  0.12329  0 10.01    0 0.547 5.913  92.9 2.3534   6 432    17.8 394.95
## 3  0.05497  0  5.19    0 0.515 5.985  45.4 4.8122   5 224    20.2 396.90
## 4  1.27346  0 19.58    1 0.605 6.250  92.6 1.7984   5 403    14.7 338.92
## 5  0.07151  0  4.49    0 0.449 6.121  56.8 3.7476   3 247    18.5 395.15
## 6  0.27957  0  9.69    0 0.585 5.926  42.6 2.3817   6 391    19.2 396.90
##   LSTAT label
## 1 29.05   7.2
## 2 16.21  18.8
```

```
## 3  9.74  19.0
## 4  5.50  27.0
## 5  8.44  22.2
## 6 13.59  24.5
```

Next, we need to normalize the data.

```
spec <- feature_spec(train_df, label ~ . ) %>%
  step_numeric_column(all_numeric(), normalizer_fn = scaler_standard()) %>%
  fit()

spec
```

```
## -- Feature Spec ----------------------------------------------------------------------------------
## A feature_spec with 13 steps.
## Fitted: TRUE
## -- Steps -----------------------------------------------------------------------------------------
## The feature_spec has 1 dense features.
## StepNumericColumn: CRIM, ZN, INDUS, CHAS, NOX, RM, AGE, DIS, RAD, TAX, PTRATIO, B, LSTAT
## -- Dense features --------------------------------------------------------------------------------
```

The above code uses the `feature\_spec` interface in the `tfdatasets` package. You can read further about it here https://tensorflow.rstudio.com/guide/tfdatasets/feature_spec/. However, for now, it suffices that eh data is normalized and transformed into a usable format for the tensor flow operations.

To view the transformed trianing data:

```
layer <- layer_dense_features(
  feature_columns = dense_features(spec),
  dtype = tf$float32
)
layer(train_df)
```

```
## tf.Tensor(
## [[ 0.81205493  0.44752213 -0.2565147  ... -0.1762239  -0.59443307
##   -0.48301655]
##  [-1.9079947   0.43137115 -0.2565147  ...  1.8920003  -0.34800112
##    2.9880793 ]
##  [ 1.1091131   0.2203439  -0.2565147  ... -1.8274226   1.563349
##   -0.48301655]
##  ...
##  [-1.6359899   0.07934052 -0.2565147  ... -0.3326088  -0.61246467
##    0.9895695 ]
##  [ 1.0554279  -0.98642045 -0.2565147  ... -0.7862657  -0.01742171
##   -0.48301655]
##  [-1.7970455   0.23288251 -0.2565147  ...  0.47467488 -0.84687555
##    2.0414166 ]], shape=(404, 13), dtype=float32)
```

Alternatively, we could have done the following to our custom data frames. Note that scaler_standard() normalizes the data to have a mean = 0 and standard deviaiton = 1.

```
train.df.std <- cbind(apply(train.df[,c(1:13)],2,function(x){(x-mean(x))/sd(x)}),train.df[,14])
head(train.df.std)
```

```
##             CRIM         ZN      INDUS       CHAS        NOX         RM
## [1,] -0.27190919 -0.4830166 -0.4352220 -0.2565147 -0.1650220 -0.1762241
## [2,] -0.40292691  2.9880792 -1.3322597 -0.2565147 -1.2136770  1.8920002
## [3,]  0.12478548 -0.4830166  1.0270523 -0.2565147  0.6278635 -1.8274222
```

```
## [4,] -0.40099633 -0.4830166 -0.8683253 -0.2565147 -0.3611120 -0.3241557
## [5,] -0.00562732 -0.4830166  1.0270523 -0.2565147  1.3269669  0.1534520
## [6,] -0.37455796 -0.4830166 -0.5468011 -0.2565147 -0.5486763 -0.7876746
##             AGE        DIS        RAD        TAX    PTRATIO         B
## [1,]  0.8120550  0.1165538 -0.6254735 -0.5944330  1.1470781 0.4475222
## [2,] -1.9079948  1.2460402 -0.8554019 -0.3480010 -1.7160613 0.4313711
## [3,]  1.1091131 -1.1859686  1.6738104  1.5633491  0.7835049 0.2203441
## [4,] -1.2351404  1.1058088 -0.5105093 -1.0933074  0.7835049 0.4475222
## [5,]  0.6939476 -0.5778555  1.6738104  1.5633491  0.7835049 0.3893995
## [6,]  0.1893067  0.4831160 -0.5105093 -0.7146437  0.5108249 0.3862118
##           LSTAT
## [1,]  0.8241983 15.2
## [2,] -1.3275563 42.3
## [3,] -1.3068796 50.0
## [4,] -0.6521177 21.1
## [5,]  0.2631706 17.7
## [6,] -0.1379572 18.5
```

Note that the data is actually equivalent. However, the tensor flow data has been re-ordered alphabetically such that the first column is "AGE" followed by "B", etc.

Now we can build the model and test it.

First we will follow the example and use the Keras function API.

```
input <- layer_input_from_dataset(train_df %>% select(-label))

output <- input %>%
  layer_dense_features(dense_features(spec)) %>%
  layer_dense(units = 64, activation = "relu") %>%
  layer_dense(units = 64, activation = "relu") %>%
  layer_dense(units = 1)

model <- keras_model(input, output)

summary(model)
```

```
## Model: "model"
## _________________________________________________________________________
## Layer (type)            Output Shape     Param #  Connected to
## =========================================================================
## AGE (InputLayer)        [(None,)]        0
## _________________________________________________________________________
## B (InputLayer)          [(None,)]        0
## _________________________________________________________________________
## CHAS (InputLayer)       [(None,)]        0
## _________________________________________________________________________
## CRIM (InputLayer)       [(None,)]        0
## _________________________________________________________________________
## DIS (InputLayer)        [(None,)]        0
## _________________________________________________________________________
## INDUS (InputLayer)      [(None,)]        0
## _________________________________________________________________________
## LSTAT (InputLayer)      [(None,)]        0
## _________________________________________________________________________
## NOX (InputLayer)        [(None,)]        0
```

```
## ____________________________________________________________________________
## PTRATIO (InputLayer)    [(None,)]        0
## ____________________________________________________________________________
## RAD (InputLayer)        [(None,)]        0
## ____________________________________________________________________________
## RM (InputLayer)         [(None,)]        0
## ____________________________________________________________________________
## TAX (InputLayer)        [(None,)]        0
## ____________________________________________________________________________
## ZN (InputLayer)         [(None,)]        0
## ____________________________________________________________________________
## dense_features_1 (Dense (None, 13)       0        AGE[0][0]
##                                                   B[0][0]
##                                                   CHAS[0][0]
##                                                   CRIM[0][0]
##                                                   DIS[0][0]
##                                                   INDUS[0][0]
##                                                   LSTAT[0][0]
##                                                   NOX[0][0]
##                                                   PTRATIO[0][0]
##                                                   RAD[0][0]
##                                                   RM[0][0]
##                                                   TAX[0][0]
##                                                   ZN[0][0]
## ____________________________________________________________________________
## dense_2 (Dense)         (None, 64)       896      dense_features_1[0][0]
## ____________________________________________________________________________
## dense_3 (Dense)         (None, 64)       4160     dense_2[0][0]
## ____________________________________________________________________________
## dense_4 (Dense)         (None, 1)        65       dense_3[0][0]
## ============================================================================
## Total params: 5,121
## Trainable params: 5,121
## Non-trainable params: 0
## ____________________________________________________________________________
```

The next step is to compile the model and define the loss function, optimizer, and performance metrics.

```
model %>%
  compile(
    loss = "mse",
    optimizer = optimizer_rmsprop(),
    metrics = list("mean_absolute_error")
  )
```

FOllowing the example, we could also put the above few steps into a function.

```
build_model <- function() {
  input <- layer_input_from_dataset(train_df %>% select(-label))

  output <- input %>%
    layer_dense_features(dense_features(spec)) %>%
    layer_dense(units = 64, activation = "relu") %>%
    layer_dense(units = 64, activation = "relu") %>%
    layer_dense(units = 1)
```

```
  model <- keras_model(input, output)

  model %>%
    compile(
      loss = "mse",
      optimizer = optimizer_rmsprop(),
      metrics = list("mean_absolute_error")
    )

  model
}
```

Finally, train the model.

```
# Display training progress by printing a single dot for each completed epoch.
print_dot_callback <- callback_lambda(
  on_epoch_end = function(epoch, logs) {
    if (epoch %% 80 == 0) cat("\n")
    cat(".")
  }
)

model <- build_model()

history <- model %>% fit(
  x = train_df %>% select(-label),
  y = train_df$label,
  epochs = 500,
  validation_split = 0.2,
  verbose = 0,
  callbacks = list(print_dot_callback)
)
```

```
##
## ................................................................................
## ................................................................................
## ................................................................................
## ................................................................................
## ................................................................................
## ................................................................................
## ....................
```

Note that the above "print_dot_callback" prints the dots above. It prints a dot for each epoch, with 80 on a line (if (epoch 80 == 0) then add a new line). Finally, the results are saved in history.

```
str(history)
```

```
## List of 2
##  $ params :List of 7
##   ..$ batch_size   : int 32
##   ..$ epochs       : int 500
##   ..$ steps        : num 11
##   ..$ samples      : int 323
##   ..$ verbose      : int 0
##   ..$ do_validation: logi TRUE
##   ..$ metrics      : chr [1:4] "loss" "mean_absolute_error" "val_loss" "val_mean_absolute_error"
```

```
##  $ metrics:List of 4
##   ..$ loss                    : num [1:500] 549 485 415 336 258 ...
##   ..$ mean_absolute_error     : num [1:500] 21.6 20 18.2 16.1 13.8 ...
##   ..$ val_loss                : num [1:500] 581 510 429 345 259 ...
##   ..$ val_mean_absolute_error: num [1:500] 22.2 20.6 18.7 16.3 13.7 ...
##  - attr(*, "class")= chr "keras_training_history"
```

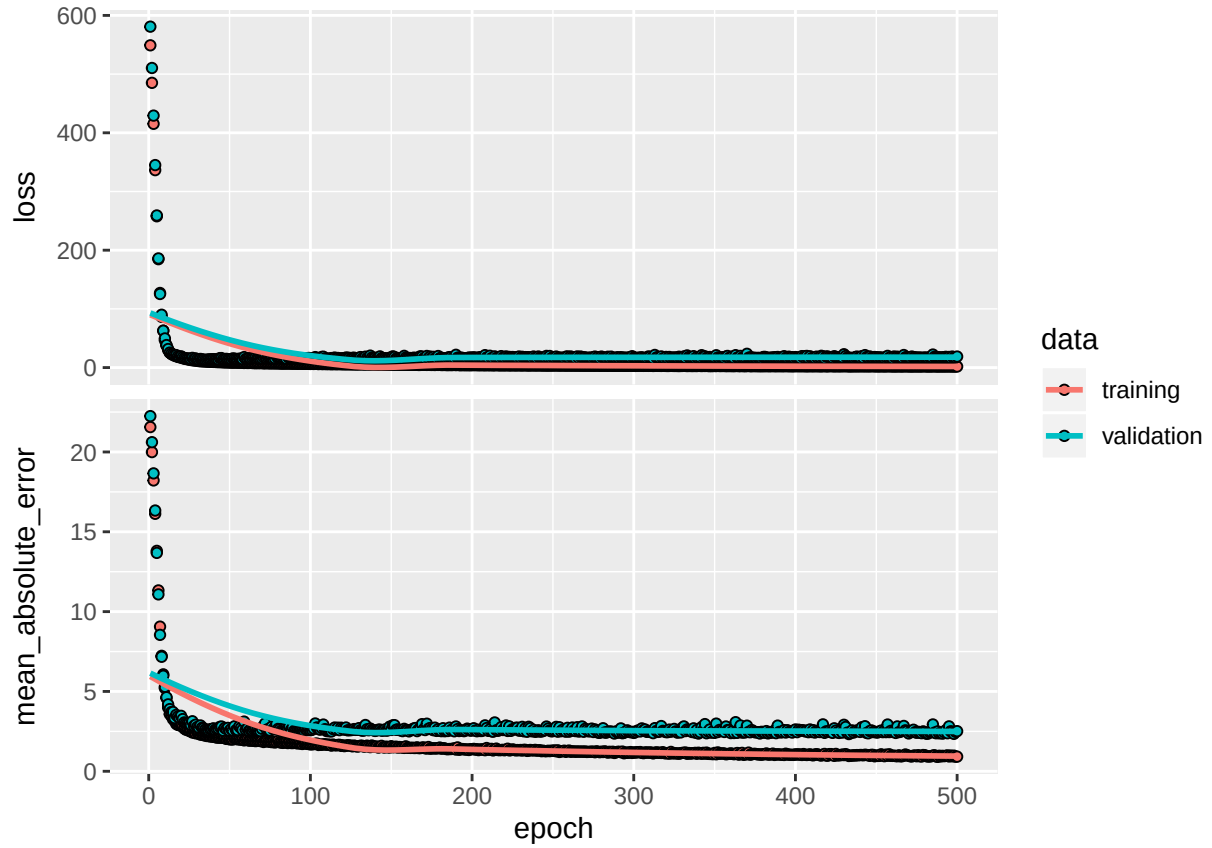
The history object saves a lot of relevant information on the model. The $params object describes the parameters used to fit the model. The $metrics object can be used to assess the fit.

```
library("ggplot2")
```

```
##
## Attaching package: 'ggplot2'
```

```
## The following object is masked _by_ '.GlobalEnv':
##
##     layer
```

```
plot(history)
```

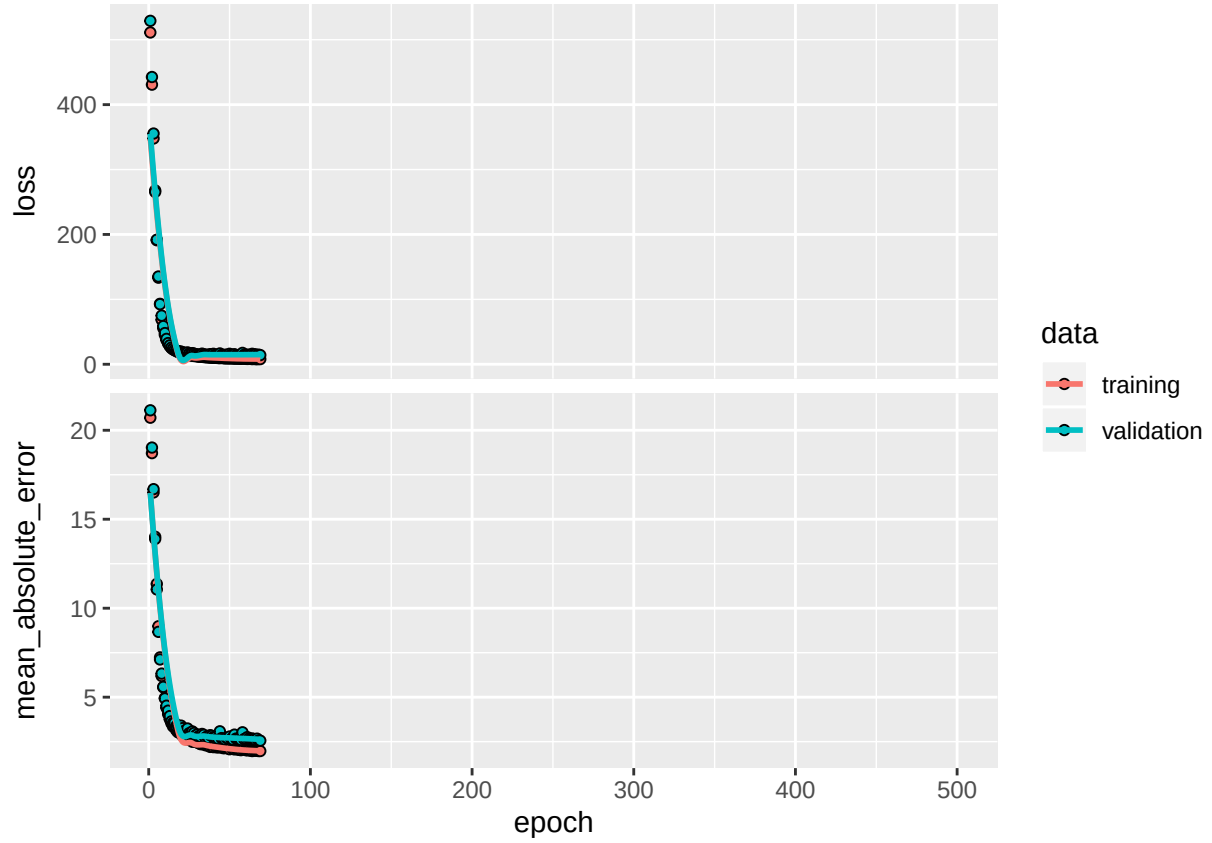


The above plots show the loss and MAE for both the training and validation samples by epoch (i.e., the iterations of the model fitting). Note that the model seems to stop improving after a 100 or so epochs. Thus, we may want to stop training early.

Therefore, we will insert an early stopping criteria to speed things up.

```
# The patience parameter is the amount of epochs to check for improvement.
early_stop <- callback_early_stopping(monitor = "val_loss", patience = 20)
```

```
model.early <- build_model()

history.early <- model.early %>% fit(
  x = train_df %>% select(-label),
  y = train_df$label,
  epochs = 500,
  validation_split = 0.2,
  verbose = 0,
  callbacks = list(early_stop)
)

plot(history.early)
```



To see how the model performed, let's see the MAE.

```
MAE.val <- history.early$metrics$val_mean_absolute_error
message(sprintf("MAE is $%5.2f.",MAE.val[length(MAE.val)]*1000))
```

```
## MAE is $2556.09.
```

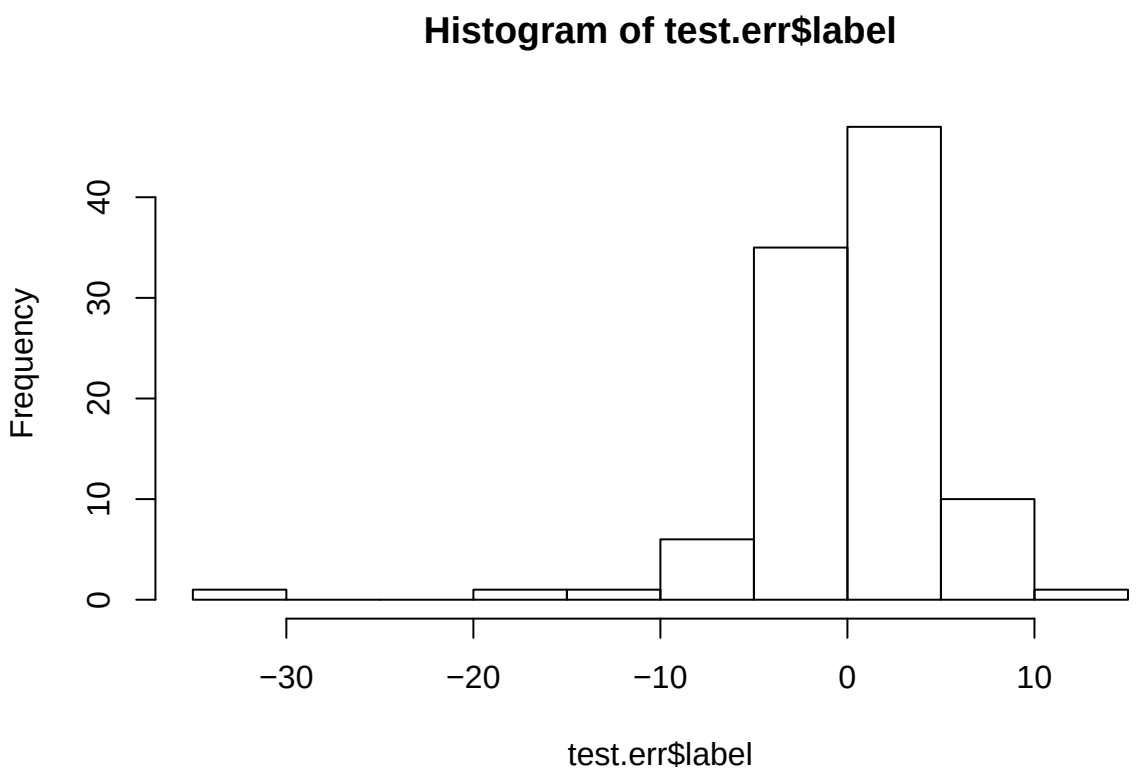
However, we really want the predictions on the test data.

```
c(loss, MAE.test) %<-% (model.early %>% evaluate(test_df %>% select(-label), test_df$label, verbose = 0)


message(sprintf("MAE is $%5.2f on the test sample.",MAE.test*1000))
```

```
## MAE is $3102.13 on the test sample.
```

Is the error too much? Let's look at the relative errors.

```
test_predictions <- model.early %>% predict(test_df %>% select(-label))
test.err <- test_df[,14] - test_predictions
hist(test.err$label)
```




```
rel.test.err <- test.err / test_df[,14]
hist(rel.test.err$label)
```

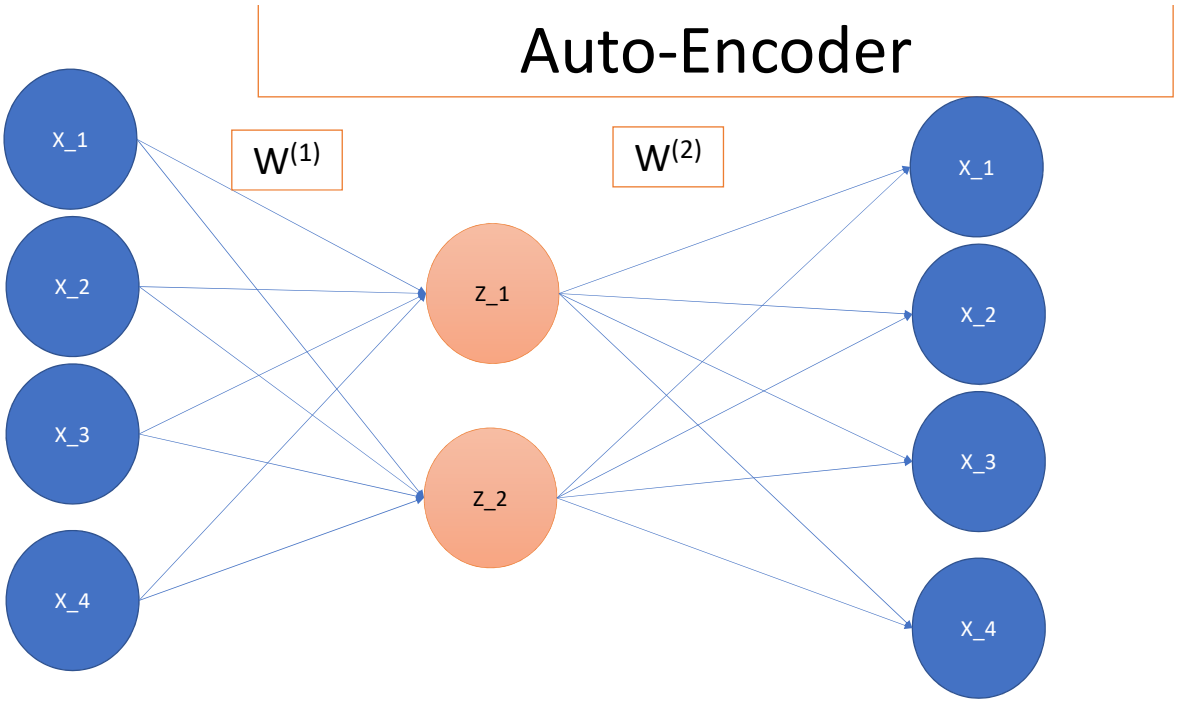

```
RMAE <- mean(abs(rel.test.err$label))
message(sprintf("RMAE is %5.2f%% on the test sample.",RMAE*100))
```

```
## RMAE is 15.57% on the test sample.
```

This means that you missed the house price by roughly 15%.

1.2.3 Example 3 - Auto-Encoder

Auto encoders are deep learning models that are useful for dimension reduction. The basic idea is very similar to a PCA in that you essentially are trying to find a reduced diensional space to explain a higher dimension space. Practically, you want to map $X \leftarrow F \leftarrow Y$, where Y is your original data. It's easiest understood with a picture.



From the above picture, it can be easily seen that we are tyring to find a set of encoding weights $W^{(1)}$ that map the input data onto itself. The $W^{(2)}$ weights are a set of decoding weights that decipher the encoded mapping back to the original values. The hidden layer $Z_1, ..., Z_p$ are similar to the PCA factors. We could also make the network deep in the sense that we could add mutliple layers in a bottleneck orientation so the central layer has the number of dimensions to which we want to reduce.

Let's go back to our interest rate data that we used for the PCA analysis and see if we can fit an auto-encoder to the data. A similar approach could also be used in a portfolio formation problem. FOr example, if you wanted to track the S&P 500 index, you could reduce the dimension of the 500 stocks in the index to a small set that explains the majority of the variance in the index.

First load the data. See https://tensorflow.rstudio.com/tutorials/beginners/load/load_csv/ for an example. This example will follow closely with https://statslab.eighty20.co.za/posts/autoencoders_keras_r/.

```
rm(list=ls())

TRAIN_DATA_LOC <- "C:/Users/clarkb2/Documents/Classes/2020-Spring/AIML/ML Course"
TEST_DATA_LOC <- "C:/Users/clarkb2/Documents/Classes/2020-Spring/AIML/ML Course"

train_file_path <- get_file("rates_2020.csv", TRAIN_DATA_LOC)
test_file_path <- get_file("rates_2020.csv", TEST_DATA_LOC)

train.df <- read.csv(train_file_path)
train.df <- train.df[,c(6:13)]
```

```
train.df <- apply(train.df,2,function(x){(x-mean(x))/sd(x)})
split_ind <- train.df[,1] %>% caret::createDataPartition(p = 0.8,list = FALSE)

train <- train.df[split_ind,]
test <- train.df[-split_ind,]
```

The train and test datasets are created using the caret splitting function - createDataPartition(). Essentially, the data is randomly split into a training and test sample. This example is straight forward in the sense that all the data is numerical.

```
train_X <- train %>% as.matrix()

test_X <- test %>% as.matrix()
```

If we had categorical data (e.g., a default 1/0 label), we could do something like the following.

```
# train_y <- train[,5] %>%
#   keras::to_categorical()
```

Now the next step is to build the model. We will start with a simple structure that has only one hidden layer. In a sense this will be similar to a PCA.

Define the first layer as the input layer.

```
input_layer <- layer_input(shape=c(8))
```

Note that 12 is the number of features we have.

The next step is to define the encoder layer(s). For now, we will only have one encoder layer. We will use a hidden layer with three (3) nodes.

```
encoder <-
  input_layer %>%
  layer_dropout(rate = 0.2) %>%
  layer_dense(units = 3,activation = 'relu')
```

Next we build the decoder layer, which maps the hidden dimensions back to the original data.

```
decoder <-
  encoder %>%
  layer_dense(units = 8) # 8 dimensions for the original 8 variables
```

Finally, we can compile and train the model using the Keras model function.

```
autoencoder_model <- keras_model(inputs = input_layer, outputs = decoder)

autoencoder_model %>% compile(
  loss='mean_squared_error',
  optimizer='adam',
  metrics = c('mean_squared_error')
)

summary(autoencoder_model)
```

```
## Model: "model_3"
## _________________________________________________________________________
## Layer (type)                 Output Shape              Param #
## =========================================================================
## input_1 (InputLayer)         [(None, 8)]               0
```

```
## ________________________________________________________________________
## dropout_1 (Dropout)              (None, 8)                     0
## ________________________________________________________________________
## dense_11 (Dense)                 (None, 3)                     27
## ________________________________________________________________________
## dense_12 (Dense)                 (None, 8)                     32
## ========================================================================
## Total params: 59
## Trainable params: 59
## Non-trainable params: 0
## ________________________________________________________________________
```

Notice that we have three layers with 12, 3, and 12 nodes. The weights on the first layer will represent the factors which are comparable to principal components.

Solve the model. Note that we will use the train_X and test_X dta only. That is, $Y = X$.

```
history <-
  autoencoder_model %>%
  keras::fit(train_X,
             train_X,
             epochs=50,
             shuffle=TRUE,
             validation_data= list(test_X, test_X)
  )
```

We could have stopped the model after about 50 epochs or so but this only takes a few seconds to run so we keep the full 100. The final MSE is stored in the history object as history$metrics$val_mean_squared_error.

```
head(history$metrics$val_mean_squared_error,5)
```

```
## [1] 1.2958705 1.1949503 1.1134884 1.0483962 0.9929737
```

```
tail(history$metrics$val_mean_squared_error,5)
```

```
## [1] 0.3850777 0.3796295 0.3744399 0.3690245 0.3634291
```

The MSE is about 0.03.

We can also recover the predicted points using the Keras predict_on_batch() funciton.

```
reconstructed_points <-
  autoencoder_model %>%
  keras::predict_on_batch(x = train_X)
```

We could compare the differences and see how we did.

```
library("rioja")
```

```
## Warning: package 'rioja' was built under R version 3.6.2
```

```
## This is rioja 0.9-21
```

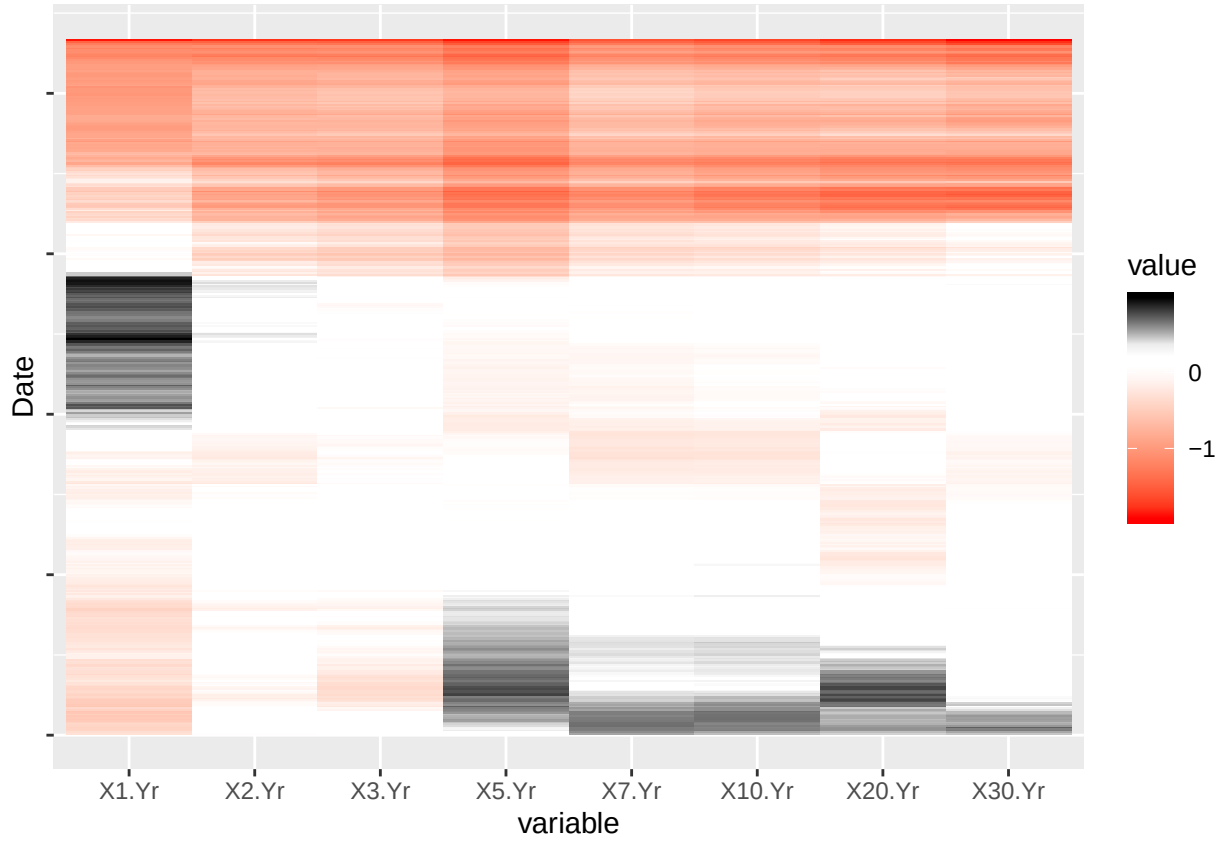
```
library("reshape2")
library("ggplot2")
library("scales")

train_hat <- as.matrix(reconstructed_points)

df.errs <- as.data.frame(train_X - train_hat)
df.errs$Date <- c(1:nrow(df.errs))
```

```
df.melt <- melt(df.errs,id.vars="Date")

N <- nlevels(df.melt$value)

p <- ggplot(df.melt, aes(x=variable,y=Date)) +
  geom_tile(aes(fill=value)) +
  scale_fill_gradientn(colours=c("red","white","white", "white","black"),
                       values=rescale(c(min(df.melt$value),0-.Machine$double.eps,0,0+.Machine$double.eps
  theme(axis.text.y = element_blank())

print(p)
```

```
## Warning in regularize.values(x, y, ties, missing(ties)): collapsing to
## unique 'x' values
```

```
## Warning in regularize.values(x, y, ties, missing(ties)): collapsing to
## unique 'x' values
```

```
## Warning in regularize.values(x, y, ties, missing(ties)): collapsing to
## unique 'x' values
```



The above plot shows how well we recovered our original values. The white areas are perfect predictions

where as the dark black or red are errors.

Now, we can also go back and get all the weights on the layers of the network to do the dimension reduction.

```
autoencoder_weights <-
  autoencoder_model %>%
  keras::get_weights()

autoencoder_weights
```

```
## [[1]]
##              [,1]       [,2]       [,3]
## [1,]  0.265797555  0.5786897  0.5967201
## [2,]  0.607714534  0.4636038  0.6537501
## [3,] -0.235804006 -0.2268173  0.5445278
## [4,] -0.209414139  0.2102737  0.2328061
## [5,]  0.366435230  0.8034121  0.3612016
## [6,]  0.009777259  0.3505738  0.3815330
## [7,]  0.504496872  0.6208218 -0.3907457
## [8,]  0.470385641  0.2908080 -0.4443508
##
## [[2]]
## [1] -0.056215107  0.462468415 -0.005542825
##
## [[3]]
##             [,1]      [,2]       [,3]        [,4]        [,5]       [,6]
## [1,] -0.1188178 0.2343228 -0.3662648 -0.03979129  0.04802009  0.3350190
## [2,]  0.3059328 0.2110620  0.7201838  0.09917236  0.46535856  0.3289587
## [3,]  0.3043551 0.2162908 -0.1943834  0.47193488 -0.07650595 -0.1155618
##             [,7]       [,8]
## [1,] -0.4915579  0.4663017
## [2,]  0.4985268  0.4002322
## [3,]  0.2785618 -0.3895994
##
## [[4]]
## [1] -0.5151827 -0.5101747 -0.4958844 -0.2964446 -0.5602314 -0.5126807
## [7] -0.5023491 -0.4842684
```

The weights are in the form of a list. The first item corresponds to the weights in the first layer. These correspond to the PCA weights. We can save teh weights as follows.

```
keras::save_model_weights_hdf5(object = autoencoder_model,filepath = 'autoencoder_weights.hdf5',overwri
```

In this case it is traightforward to return the factor scores, but as we will see below fo rthe case of a deep auto-encoder network this may not be the case. Thus, we will do an example so show how to get the desired new dimesnions.

First, define a new model that is the encoder portion of your network.

```
encoder_model <- keras_model(inputs = input_layer, outputs = encoder)
```

Next, call the model and load the saved weights and compile it.

```
encoder_model %>% keras::load_model_weights_hdf5(filepath = "autoencoder_weights.hdf5",skip_mismatch = '

encoder_model %>% compile(
  loss='mean_squared_error',
  optimizer='adam',
```

```
  metrics = c('mean_squared_error')
)
```

We can finally get the new dimensions.

```
dim_reduction <-
  encoder_model %>%
  keras::predict_on_batch(x = train_X)

dim_reduction
```

```
## tf.Tensor(
## [[0.         0.         0.        ]
##  [0.         0.         0.        ]
##  [0.         0.14740449 0.        ]
##  ...
##  [0.         0.         0.        ]
##  [0.         0.         0.        ]
##  [0.         0.         0.        ]], shape=(433, 3), dtype=float32)
```

```
# View(as.matrix(dim_reduction))
```

"dim_reduction" are the new variables that you would use as an input to a regularized model, akin to what you would use for PCA.

We can also compare our results to a PCA analysis.

```
pre_process <- caret::preProcess(train_X,method = "pca",pcaComp = 3)
pca <- predict(pre_process,train_X)
```

Compare the correaltions across the factors.

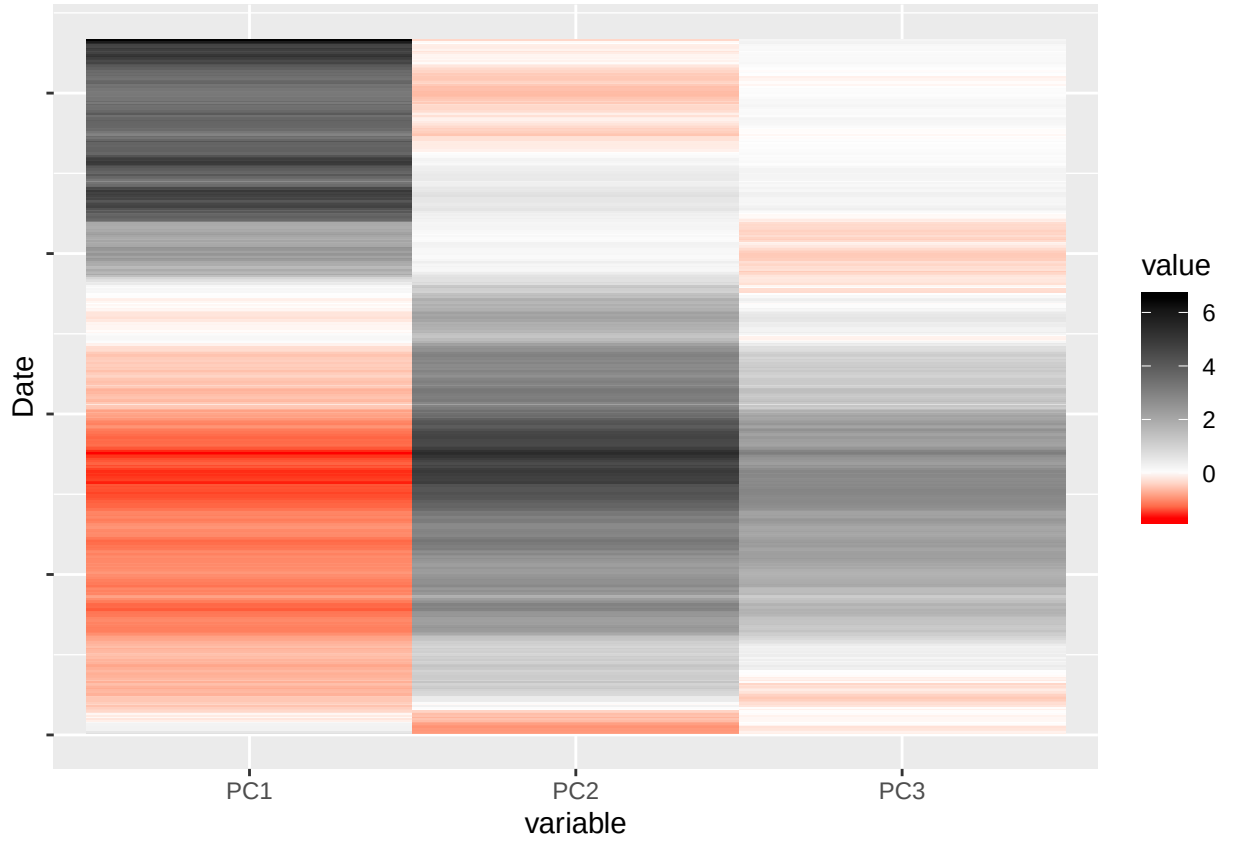
```
auto.enc <- as.matrix(dim_reduction)

rslt <- cbind(pca,auto.enc)
cor(rslt)
```

```
##              PC1           PC2           PC3
## PC1 1.000000e+00  2.747766e-15  2.035003e-15  0.8782586  0.92262058
## PC2 2.747766e-15  1.000000e+00 -6.171889e-17 -0.1403483 -0.06306813
## PC3 2.035003e-15 -6.171889e-17  1.000000e+00 -0.1828760 -0.20542011
##     8.782586e-01 -1.403483e-01 -1.828760e-01  1.0000000  0.98836799
##     9.226206e-01 -6.306813e-02 -2.054201e-01  0.9883680  1.00000000
##     8.281370e-01 -2.654136e-01 -3.570336e-01  0.9557089  0.94476485
##
## PC1  0.8281370
## PC2 -0.2654136
## PC3 -0.3570336
##      0.9557089
##      0.9447648
##      1.0000000
```

Let's see how the new reduced dimenson data comapres.

```
Z.errs <- as.data.frame(auto.enc - pca)
Z.errs$Date <- c(1:nrow(Z.errs))

Z.melt <- melt(Z.errs,id.vars="Date")
```

```
N <- nlevels(Z.melt$value)

p.Z <- ggplot(Z.melt, aes(x=variable,y=Date)) +
  geom_tile(aes(fill=value)) +
  scale_fill_gradientn(colours=c("red","white","white", "white","black"),
                       values=rescale(c(min(Z.melt$value),0-.Machine$double.eps,0,0+.Machine$double.eps
  theme(axis.text.y = element_blank())

print(p.Z)
```



There are some obvious differences between the methods which isn't entirely surprising.

Next, let's try a deep auto-encoder.

1.2.4 Example 4 - Deep Auto-Encoder

Now, we will implement a deep auto-encoder which has many hidden layers with a bottle-neck shape. The bottle neck is the center layer that will be our new dimensions.

We will satrt with teh same data.

Define the first layer as the input layer.

```
input_layer <- layer_input(shape=c(8))
```

Note that 12 is the number of features we have. The difference from the previous example is in the following lines of code; the encoder and decoder functions are deeper.

```
encoder <-
  input_layer %>%
  layer_dense(units = 150, activation = "relu") %>%
  layer_batch_normalization() %>%
  layer_dropout(rate = 0.2) %>%
  layer_dense(units = 50, activation = "relu") %>%
  layer_dropout(rate = 0.1) %>%
  layer_dense(units = 25, activation = "relu") %>%
  layer_dense(units = 3) # 2 dimensions for the output layer

decoder <-
  encoder %>%
  layer_dense(units = 150, activation = "relu") %>%
  layer_dropout(rate = 0.2) %>%
  layer_dense(units = 50, activation = "relu") %>%
  layer_dropout(rate = 0.1) %>%
  layer_dense(units = 25, activation = "relu") %>%
  layer_dense(units = 8) # 4 dimensions for the original 4 variables
```

Again, we can compile and train the model using the Keras model function.

```
autoencoder_model.D <- keras_model(inputs = input_layer, outputs = decoder)

autoencoder_model.D %>% compile(
  loss='mean_squared_error',
  optimizer='adam',
  metrics = c('mean_squared_error')
)

summary(autoencoder_model.D)
```

```
## Model: "model_5"
## _________________________________________________________________________
## Layer (type)                    Output Shape                  Param #
## =========================================================================
## input_2 (InputLayer)            [(None, 8)]                   0
## _________________________________________________________________________
## dense_13 (Dense)                (None, 150)                   1350
## _________________________________________________________________________
## batch_normalization (BatchNormal (None, 150)                  600
## _________________________________________________________________________
## dropout_2 (Dropout)             (None, 150)                   0
## _________________________________________________________________________
## dense_14 (Dense)                (None, 50)                    7550
## _________________________________________________________________________
## dropout_3 (Dropout)             (None, 50)                    0
## _________________________________________________________________________
## dense_15 (Dense)                (None, 25)                    1275
## _________________________________________________________________________
## dense_16 (Dense)                (None, 3)                     78
## _________________________________________________________________________
## dense_17 (Dense)                (None, 150)                   600
## _________________________________________________________________________
## dropout_4 (Dropout)             (None, 150)                   0
```

```
## _________________________________________________________________
## dense_18 (Dense)                 (None, 50)                    7550
## _________________________________________________________________
## dropout_5 (Dropout)              (None, 50)                    0
## _________________________________________________________________
## dense_19 (Dense)                 (None, 25)                    1275
## _________________________________________________________________
## dense_20 (Dense)                 (None, 8)                     208
## =================================================================
## Total params: 20,486
## Trainable params: 20,186
## Non-trainable params: 300
## _________________________________________________________________
```

Notice that we have 7 hidden layers layers with 150, 50 25, 3, 25, 50, and 150 nodes. The weights on the first layer will represent the factors which are comparable to principal components.

Solve the model. Note that we will use the train_X and test_X dta only. That is, $Y = X$.

```
history.D <-
  autoencoder_model.D %>%
  keras::fit(train_X,
             train_X,
             epochs=100,
             shuffle=TRUE,
             validation_data= list(test_X, test_X)
  )
```

We could have stopped the model after about 50 epochs or so but this only takes a few seconds to run so we keep the full 100. The final MSE is stored in the history object as history$metrics$val_mean_squared_error.

```
head(history.D$metrics$val_mean_squared_error,5)
```

```
## [1] 0.8713644 0.8094540 0.7365517 0.7223928 0.6649703
```

```
tail(history.D$metrics$val_mean_squared_error,5)
```

```
## [1] 0.01075424 0.01582984 0.02069499 0.01521974 0.02675117
```

The MSE is about 0.012, which is much less than before.

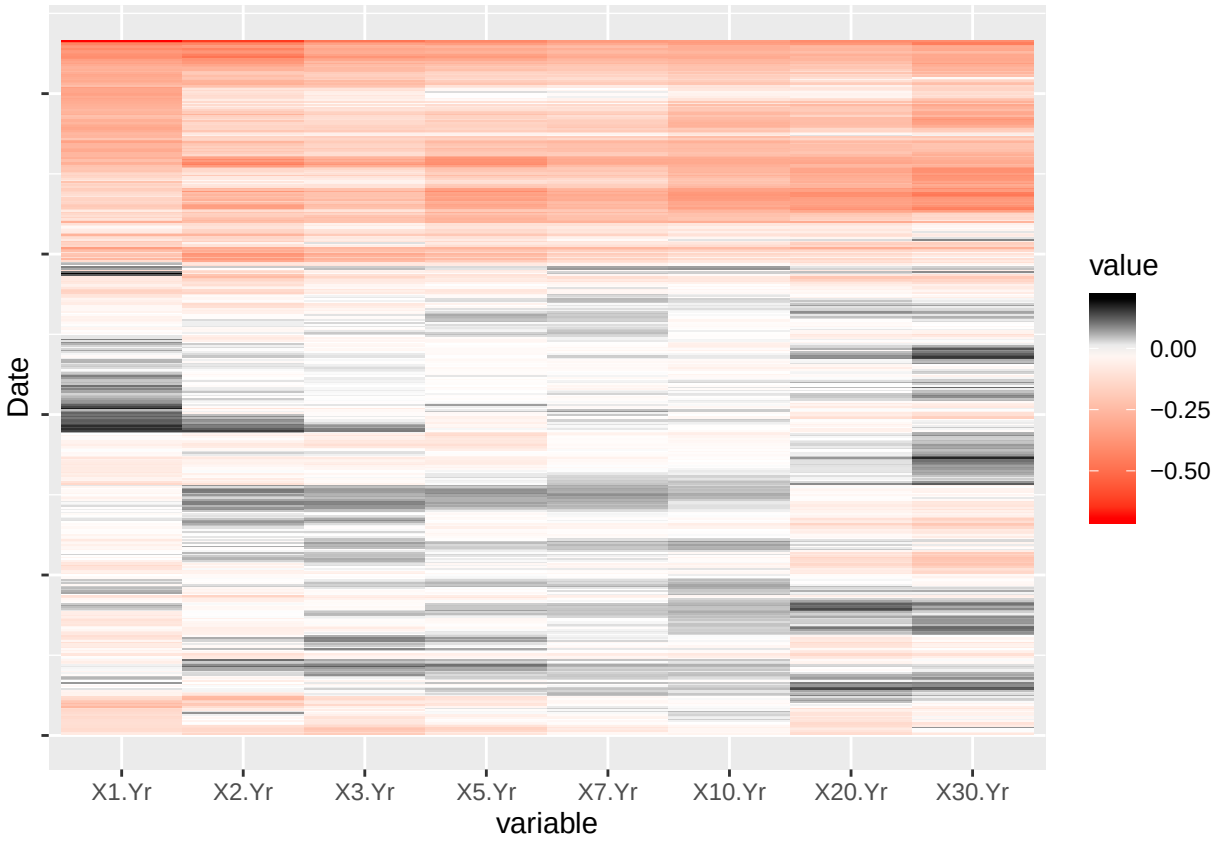
We can also recover the predicted points using the Keras predict_on_batch() funciton.

```
reconstructed_points.D <-
  autoencoder_model.D %>%
  keras::predict_on_batch(x = train_X)
```

We could compare the differences and see how we did.

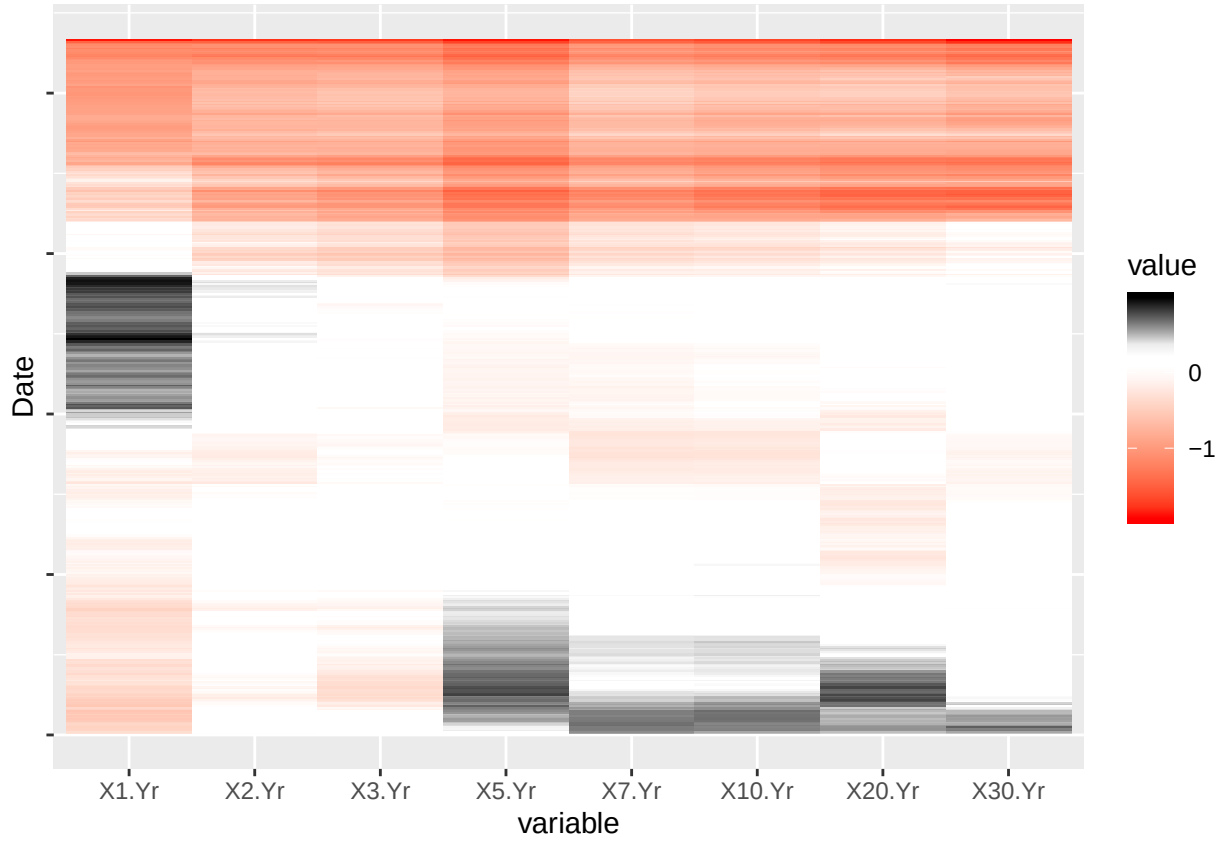
```
library("rioja")
library("reshape2")
library("ggplot2")
library("scales")

train_hat.D <- as.matrix(reconstructed_points.D)

df.errs.D <- as.data.frame(train_X - train_hat.D)
df.errs.D$Date <- c(1:nrow(df.errs.D))
```

```
df.melt.D <- melt(df.errs.D,id.vars="Date")

N <- nlevels(df.melt.D$value)

p.D<- ggplot(df.melt.D, aes(x=variable,y=Date)) +
  geom_tile(aes(fill=value)) +
  scale_fill_gradientn(colours=c("red","white","white", "white","black"),
                       values=rescale(c(min(df.melt.D$value),0-.Machine$double.eps,0,0+.Machine$double.
  theme(axis.text.y = element_blank())

print(p.D)
```




```
print(p)
```

```
## Warning in regularize.values(x, y, ties, missing(ties)): collapsing to
## unique 'x' values

## Warning in regularize.values(x, y, ties, missing(ties)): collapsing to
## unique 'x' values

## Warning in regularize.values(x, y, ties, missing(ties)): collapsing to
## unique 'x' values
```




```
library("gridExtra")
```

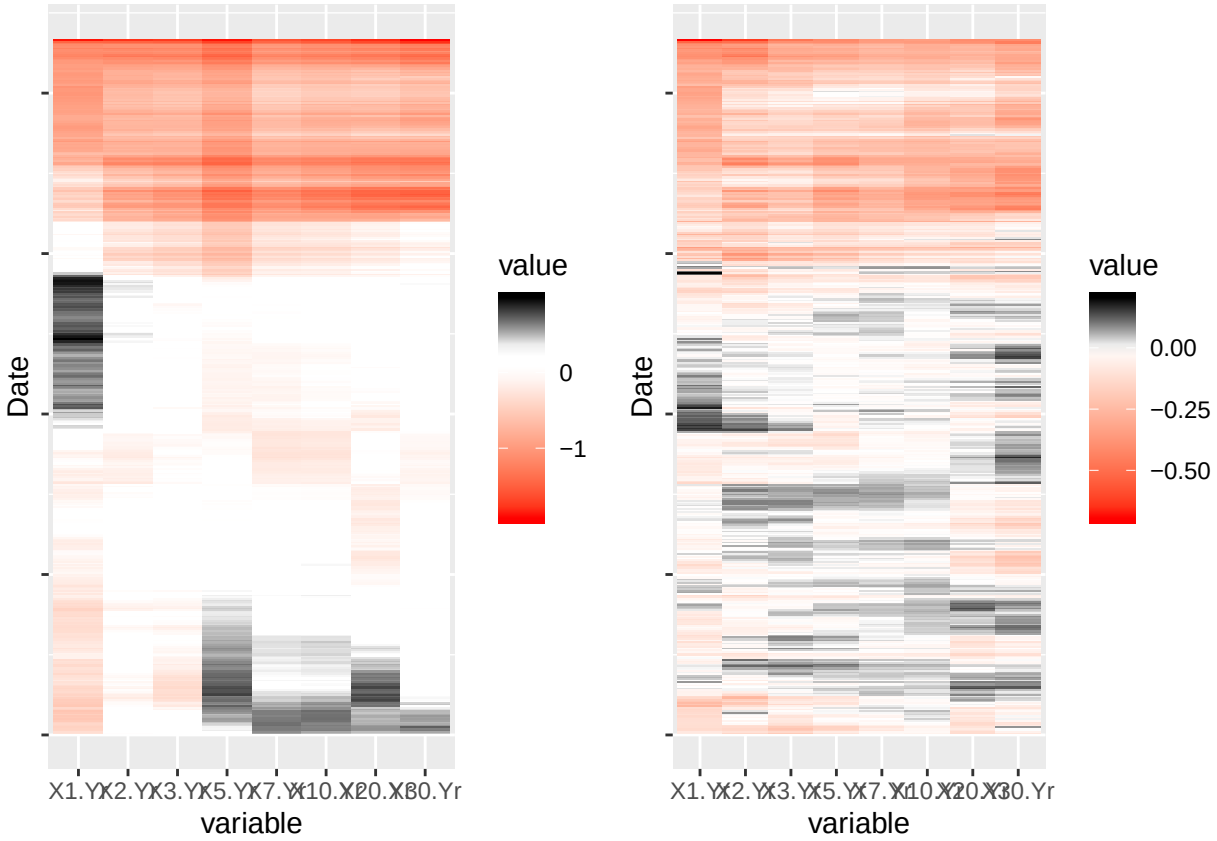
```
## Warning: package 'gridExtra' was built under R version 3.6.2
```

```
##
## Attaching package: 'gridExtra'
```

```
## The following object is masked from 'package:dplyr':
##
##     combine
```

```
grid.arrange(p,p.D,ncol=2,nrow=1)
```

```
## Warning in regularize.values(x, y, ties, missing(ties)): collapsing to
## unique 'x' values

## Warning in regularize.values(x, y, ties, missing(ties)): collapsing to
## unique 'x' values

## Warning in regularize.values(x, y, ties, missing(ties)): collapsing to
## unique 'x' values
```



The above plots show how well we recovered our original values. The right side plot is the deep network and the left plot is the shallow network. There are two main differences. One, the errors are much smaller on the deep auto-encoder and two, they are much more non-linear. Notice that there are clear patterns in the data on the left in terms of big chucks of red or black.

Now, we can also go back and get all the weights on the layers of the network to do the dimension reduction.

```
autoencoder_weights.D <-
  autoencoder_model.D %>%
  keras::get_weights()
```

```
keras::save_model_weights_hdf5(object = autoencoder_model,filepath = 'autoencoder_weights_DEEP.hdf5',ov
```

Now that the network is more complicated, we can still use the same code to get our factor scores.

First, define a new model that is the encoder portion of your network.

```
encoder_model.D <- keras_model(inputs = input_layer, outputs = encoder)
```

Next, call the model and load the saved weights and compile it.

```
encoder_model.D %>% keras::load_model_weights_hdf5(filepath = "autoencoder_weights_DEEP.hdf5",skip_misma

encoder_model.D %>% compile(
  loss='mean_squared_error',
  optimizer='adam',
  metrics = c('mean_squared_error')
)
```

We can finally get the new dimensions.

```
dim_reduction.D <-
  encoder_model.D %>%
  keras::predict_on_batch(x = train_X)

dim_reduction.D
```

```
## tf.Tensor(
## [[-0.08242412 -0.64130414 -0.8674446 ]
##  [-0.03852785 -0.77328163 -0.9315891 ]
##  [ 0.12752551 -1.2381805  -1.012443  ]
##  ...
##  [ 0.7321412   2.8322449  -3.7937033 ]
##  [ 0.8635161   3.223953   -4.38526   ]
##  [ 1.3267087   3.5469306  -5.329883  ]], shape=(433, 3), dtype=float32)
```

```
# View(as.matrix(dim_reduction))
```

"dim_reduction" are the new variables that you would use as an input to a regularized model, akin to what you would use for PCA.

We can also compare our results to a PCA analysis.

```
pre_process <- caret::preProcess(train_X,method = "pca",pcaComp = 3)
pca <- predict(pre_process,train_X)
```

Compare the correaltions across the factors.

```
auto.enc.D <- as.matrix(dim_reduction.D)

rslt <- cbind(pca,auto.enc.D)
cor(rslt)
```

```
##                PC1           PC2           PC3
## PC1  1.000000e+00  2.747766e-15  2.035003e-15 -0.6936221 -0.90954516
## PC2  2.747766e-15  1.000000e+00 -6.171889e-17 -0.3481742 -0.32183815
## PC3  2.035003e-15 -6.171889e-17  1.000000e+00  0.3574900 -0.09538901
##     -6.936221e-01 -3.481742e-01  3.574900e-01  1.0000000  0.72479795
##     -9.095452e-01 -3.218381e-01 -9.538901e-02  0.7247980  1.00000000
##      8.989248e-01 -2.833631e-01  4.832354e-02 -0.5208225 -0.80427414
##
## PC1  0.89892477
## PC2 -0.28336305
## PC3  0.04832354
##     -0.52082245
##     -0.80427414
##      1.00000000
```

2 References