New York University: Machine Learning in Economics

Lecture 9: Neural Networks II - Introduction to Keras

Dr. George Lentzas

What is Deep Learning

In classical programming humans input rules (a program) and data to be processed according to those rules, and out come answers. With Machine Leanring (ML) humans input data as well as answers expected from this data, and out come the rules.

These rules can then be applied to new data to produce new asnwers. Hence we say that an ML system is trained rather than explicitly programmed.

More specifically, ML transforms its input data into meaningful outputs, a process that is "learned" from exposure to examples of inputs and outputs.

Therefore, one of the central problems in ML is to meaningfully transform the data, i.e. create representations of the input data that get us closer to the output.

Thus, learning, in the context of many but not all ML models, can be thought of as an automatic search for better representations of the input data, through a predefined hypothesis space, using guidance from a feedback signal.

What is Deep Learning

In that sense Deep Learning (DL) is a subfield of ML that puts an emphasis on creating multiple successive meaningful representations of the input data. This approach completely automates what used to be the hardest step in the ML workflow, namely feature engineering.

What is also transformative is that these features are learned jointly via multiple transformations (not greedily, layer by layer). Keep in mind however that DL is not always the right tool for the job, especially if the data is not large enough. For example, for non perceptual data, GBMs are expected to do better than DL.

What is Deep Learning

DL took off massively after 2012 due to mostly three technical factors: (i) hardware, (ii) datasets and benchmarks, and (iii) algorithmic advances.

Hardware: Deep neural networks consist of mostly many small matrix multiplications and are highly parallelizable. The rise of cheap and powerful GPUs for gaming and later DL specific chips has allowed for learnign speed that are orders of magnitudes faster than what is possible with a CPU.

Data: If there is one dataset that has been a catalyst for the rise of DL it is the ImageNet dataset, consisting of 1.4 million images that have been annotated with 1,000 image categories and associated with a famous yearly competition.

Algorithms: Until the late 2000s we were missing a reliable way to train very deep neural networks. The key issue was gradient propagation through deep stacks of layers (the signal would fade away as the number of layers increased). This changed around 2009-2010 with the advent of several algorithmic improvements: (i) better activation functions, (ii) better weight initialization via layer-wise pretraining (quickly abandonded), and (iii) better optimization routines (RMSPRop and Adam). Eventually, during 2014-2016, even more algorithmic improvements were discovered such as (iv) batch normalization, (v) residual connections, and (vi) depth-wise separable convolutions. Today we can train models that are thousands of layers deep!

What is Keras

Keras is a high level and user friendly Machine Learning API, written in Python and capable of running on top of TensorFlow, CNTK, or Theano, that allows extremely fast implementation of Neural Networks. Keras has the following attractive characteristics:

- The same code can run on CPUs or GPUs seamlessly.
- It supports a very large number of models, including Convolutional and Recurrent Nets (or combinations of these).
- It supports arbitrary network architectures.
- It can run on top of most major back-end systems, including Tensorflow and Theano.

Here we will use R to utilize and interact with Keras.

This is of minor importance, you can, if your prefer, choose Python, to interact with Keras with no real difference. First we will install the Keras R package; we will use the package `devtools` to install this directly from Git.

Install Keras and Tensorflow

```
install.packages("devtools") #install from Git
install.packages("magrittr") #pipe operator package
install.packages("ggplot2") #plotting package
install.packages("tibble") #package that allows for "improved" dataframes
devtools::install_github("rstudio/keras")
```

The Keras R interface uses Tensorflow as its default back-end.

Tensorflow is an open source library for low level machne learning operations developed by Google Brain and used for both research and production at Google. It is currently the de facto main neural network back end system.

```
keras::install_keras()
```

This will give you the CPU versions of Keras and Tensorflow. If you want to take advantage of GPU computation you please refer to `https://keras.rstudio.com/reference/install_keras.html`.

Building the First Neural Network

We will start with a simple example of how to build a Neural Network to do handwritten digit recognition from the famous MNIST dataset (`https://en.wikipedia.org/wiki/MNIST_database`).

MNIST consists of greyscale images of handwritten digits of dimension $28x28$ and target variables corresponding to the actual digit, namely 10 categories, 0 to 9. This dataset is included with KERAS so we will load it and we will create features and target variables for both train and test data sets.

We will use the MNIST dataset which has about 60,000 training images and 10,000 testing images and which come preloaded in Keras as training and testing subsets.

```
library(keras)
mnist <- dataset_mnist()
train_images <- mnist$train$x
train_labels <- mnist$train$y
test_images <- mnist$test$x
test_labels <- mnist$test$y
```

The images are 3D arrays and the labels are a 1D array of digits ranging from 0 to 9. The function `str()` is a convenient way to look at the structure of an array.

Building the First Neural Network

```
str(train_images)

##  int [1:60000, 1:28, 1:28] 0 0 0 0 0 0 0 0 0 0 ...

str(train_labels)

##  int [1:60000(1d)] 5 0 4 1 9 2 1 3 1 4 ...
```

Now lets build a simple neural network. The network will consist of a sequence of two layers, which are densely (fully) connected, with the second layer returning an array of 10 probability scores.

```
library(magrittr)
network <- keras_model_sequential() %>%
  layer_dense(units = 50, activation = "relu", input_shape = c(28*28)) %>% # 50 hidden units
  layer_dense(units = 10, activation = "softmax")
```

Building the First Neural Network

To make our network ready for training we need to pick three more things as part of the *compilation* step:

1. The Loss Function: how the network measures its performance on the training data.
2. The Optimization Algorithm: how the network will update itself based on the Loss FUnction
3. Metrics to Monitor: what to look at during training and testing

```
network %>% compile(  # note the syntax: this is "modifying in place"
  optimizer = "rmsprop", #note we have set the learning rate manually
  loss = "categorical_crossentropy",
  metrics = c("accuracy")
)
```

Building the First Neural Network

The syntax above uses the pip operator %¿% to highlight an important charactiristic of Keras models. Unlike most R objects, Keras models are modified in place (because they are directed acyclic graphs of layers whose state is update during training). You dont operate on `network` and then return a new `network` object. Placing `network` to the left of %¿% and not saving the results to a new variable signals to the reader that you are modifying in place.
Before training we will reshape the data into the shape that the network expects and scale it so that values are in the $[0, 1]$ interval.

Below we transform a 3D array of shape $(60000, 28, 28)$ of type integer with values in the $[0, 255]$ interval into a doubl arra of shape $(60000, 28 * 28)$ with values between 0 and 1.

```
train_images <- keras::array_reshape(train_images, c(60000, 28*28)) / 255
test_images <- keras::array_reshape(test_images, c(10000, 28*28)) / 255

train_labels <- keras::to_categorical(train_labels)
test_labels <- keras::to_categorical(test_labels)
```

Note that we did not use the `dim()` function to reshape the array.

Building the First Neural Network

We are now ready to train the network which in Keras is done via a call to the network's `fit` method.

```
network %>% fit(train_images, train_labels, epochs = 10, batch_zise = 30)
```

Two quantities are displayed during training: the loss of the network over the training data and the accuracy of the network on the training data. To check how the model performs on the test data wee use `evaluate`

```
metrics <- network %>% evaluate(test_images, test_labels)
metrics

## $loss
## [1] 0.1252879
##
## $accuracy
## [1] 0.9699

network %>% predict_classes(test_images[1:10, ])

##  [1] 7 2 1 0 4 1 4 9 5 9

test_labels[1:10]

##  [1] 0 0 0 1 0 0 0 0 0 0
```

An Introduction to Tensors

Tensors (multidimentional arrays) are the generalization of vectors and matrices to an arbitrary number of dimensions (called axes).

In R we use vectors for 1D tensors, matrices for 2D tensors and arrays for ND tensors. First two simple examples of 1D and 2D tensors.

```
x <- c(1, 2, 3, 4) # a vector example
x

## [1] 1 2 3 4

attributes(x) # no dimension attribute

## NULL

x <- as.array(x)
attributes(x)

## $dim
## [1] 4

dim(x)

## [1] 4
```

An Introduction to Tensors

```
x <- matrix(seq(1,9),3,3) # a vector example
x
```

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

```
attributes(x) # no dimension attribute
```

```
## $dim
## [1] 3 3
```

```
dim(x)
```

```
## [1] 3 3
```

An Introduction to Tensors

Now an example of a 3D tensor, which is essentially a cube of numbers.

```
x <- array(seq(1, 18), dim = c(3,3,2)) # a 3D tensor
x # you can think of this as two 3x3 matrices stacked together
```

```
## , , 1
##
##      [,1] [,2] [,3]
## [1,]    1    4    7
## [2,]    2    5    8
## [3,]    3    6    9
##
## , , 2
##
##      [,1] [,2] [,3]
## [1,]   10   13   16
## [2,]   11   14   17
## [3,]   12   15   18
```

```
dim(x)
```

```
## [1] 3 3 2
```

An Introduction to Tensors

We can keep going, now an example of a 4D tensor.

```
x <- array(seq(1, 16), dim = c(2,2,2,2)) # a 4D tensor
x
```

```
## , , 1, 1
##
##      [,1] [,2]
## [1,]    1    3
## [2,]    2    4
##
## , , 2, 1
##
##      [,1] [,2]
## [1,]    5    7
## [2,]    6    8
##
## , , 1, 2
##
##      [,1] [,2]
## [1,]    9   11
## [2,]   10   12
##
## , , 2, 2
##
##      [,1] [,2]
## [1,]   13   15
## [2,]   14   16
```

Tensors Attributes

Formally, a tensor has the following key attributes:

Rank: the number of axes

Shape: an integer vector that descibes how many dimensions the tensor has along each axis. This is given by `dim()` above.

Data Type: the type of data in the tensor (usually `double`, or `integer`).
Back to the MNIST example.

```
mnist <- dataset_mnist()
train_images <- mnist$train$x; train_labels <- mnist$train$y
test_images <- mnist$test$x; test_labels <- mnist$test$y
rank <- length(dim(train_images)); rank

## [1] 3

shape <- dim(train_images); shape

## [1] 60000    28    28

data_type <- typeof(train_images); data_type

## [1] "integer"
```

A Tensor Example

So this is a 3D tensor and specifically an array of 60000 matrices of 28×28. The data inside each matrix are integers between 0 and 255 representing greyscale values.

Now lets use tensor slicing to select a mini batch of size 20.

```
mini_batch <- train_images[5:24, , ]
# the first axis here is called the batch axis or batch dimension
dim(mini_batch)

## [1] 20 28 28
```

Now lets select a specific element in the tensor (something called *tensor slicing*).

A Tensor Example

```
plot(as.raster(train_images[1, , ], max = 255))
```

Data as Tensors

Lets look at some examples of data tensors:

Vector Data: 2D tensors of shape

```
(sample values, features)
```

Each data point can be thought of as a vector of features with the data set as a 2D tensor (a matrix).

Time Series Data: 3D tensors of shape

```
(sample values, timesteps, features)
```

Timeseries have an explicit time axis and each sample is encoded as a sequence of vectors (a 2D tensor) with the data set now being a 3D tensor. An example is a data set showing some features for a stock (e.g. the stock return, average bid-offer, volume traded) all from time t-1 to time t for all the stocks in the Dow Jones Index (a 2D tensor) over the course of the last 1000 minutes (the data being a 3D tensor). This would be a 3D tensor of shape

```
(time, asset_name, asset_features)
```

Data as Tensors

Image Data: 4D tensors of shape

```
(sample values, height, width, channels)
```

or

```
(samples, channels, height, width)
```

Images typically have three dimensions: height, width and color depth. Greyscale images have one color channel hence we used a 2D tensor for each image (recall our mini batch example above). To add color you need an additional axis, hence representing the Red, Green, and Blue colors. In Tensorflow the shape of an image dataset would look like

```
(samples, height, width, color_depth)
```

Video Data: 5D tensors of shape

```
(sample values, height, width, channels)
```

Video is a sequence of frames so we need an extra axis to represent time. The dataset would now look like

```
(sample values, frames, height, width, color_depth)
```

Tensor Operations

Transformations learnt by DL can be reduced to a handful of tensor operations (e.g. addition, multiplication, etc.) applied to numeric tensors. For example you can think of a layer as a series of operations on 2D tensors:

```
output = max((dot(W,input) + b), 0)
```

Element Wise Operations These are applied to each element in the tensor independently (and are parallelizable in software implementations). Examples include addition and the relu operator above. These are usually delegated to a BLAS (Basic Linear Algebra Subprograms) implementation which if you are using R you can see by running `sessionInfo()`. You can change the default option to optimize performance (see for example: `http://brettklamer.com/diversions/statistical/faster-blas-in-r/`).

Operations on Tensors with Different Dimensions What happens for example with addition, when the shapes of two tensors differ? The R `sweep()` function allows you to perform such operations:

```
sweep(x,2,y,`+`)
```

The second argument denotes the dimension through which sweeping is done. (This is called "broadcasting" in Python).

Tensor Operations

An example of sweeping is given below:

```
x <- array(seq(-5, 180), dim = c(4,3,2)) # a 3D tensor
x
```

```
## , , 1
##
##      [,1] [,2] [,3]
## [1,]   -5   -1    3
## [2,]   -4    0    4
## [3,]   -3    1    5
## [4,]   -2    2    6
##
## , , 2
##
##      [,1] [,2] [,3]
## [1,]    7   11   15
## [2,]    8   12   16
## [3,]    9   13   17
## [4,]   10   14   18
```

```
y <- c(100, 100, 100)
y
```

```
## [1] 100 100 100
```

Tensor Operations

An example of sweeping is given below (cont.):

```
sweep(x, 2, y, `+`)
```

```
## , , 1
##
##      [,1] [,2] [,3]
## [1,]   95   99  103
## [2,]   96  100  104
## [3,]   97  101  105
## [4,]   98  102  106
##
## , , 2
##
##      [,1] [,2] [,3]
## [1,]  107  111  115
## [2,]  108  112  116
## [3,]  109  113  117
## [4,]  110  114  118
```

Tensor Operations

Tensor Dot Dot products use the \%*\% operator. Tensors generalize the idea of matrix multiplication to higher dimensions (as long as the internal dimensions are compatible):

`(a, b, c, d) %*% (d, e) -> (a, b, c, e)`

Tensor Reshaping This is typically used for data pre-processing. You should alwyas use `array_reshape()` from the keras package so that the data is compatible with Keras based row-major semantics. Lets look at an example

```
x <- matrix(seq(1, 4), 2, 2)
x
```

```
##      [,1] [,2]
## [1,]    1    3
## [2,]    2    4
```

```
keras::array_reshape(x, dim = c(4, 1))
```

```
##      [,1]
## [1,]    1
## [2,]    3
## [3,]    2
## [4,]    4
```

Tensor Operations

Geometric Interpretation of DL Because tensors can be interpreted as coordinates of points in a geometrical space, all tensor operations also have a geometric interpretation.

Thus, since a neural network consists of a series of tensor transformations, you can interpret a neural network as a complicated geometric transformation in a high dimensional space, implemented by doing a long series of simple steps.

Example 1: A Better MNIST Model

Recall that we use the MNIST dataset which has about 60,000 training images and 10,000 testing images and which come preloaded in Keras as training and testing subsets. We re-state the code from above for convenience:

```
library(ggplot2) # we will use this to create some nice graphs

mnist   <- keras::dataset_mnist()

x_train <- mnist$train$x
y_train <- mnist$train$y
x_test  <- mnist$test$x
y_test  <- mnist$test$y
```

Example 1: A Better MNIST Model

The x data is a 3 dimensional array (figures x width x height) of greayscale values. We will start by reshaping the data by a process called *flattening*: each 28 x 28 matrix representing an image is converted to an 1-d vector of lenght of 784 (square of 28). Then the grayscale values, which are integers from 0 to 255, are rescaled to numbers between 0 and 1.

```
# flattening
x_train <- keras::array_reshape(x_train, c(nrow(x_train), 784))
x_test  <- keras::array_reshape(x_test, c(nrow(x_test), 784))
# rescaling
x_train <- x_train / 255
x_test  <- x_test / 255
```

We used the `array_reshape()` (and not the standard R `dim<-()` function) to reshape the array.

This is so that the data is interpreted using row-major semantics (instead of Rs default column-major semantics), which is compatible with the way that Keras interprets array dimensions.

Example 1: A Better MNIST Model

The y data is a vector of integers from 0 to 9. We will process the data by a technique called in ML *one-hot encoding* to transform the vectors into binary class matrices using the Keras `to_categorical()` function. This is nothing more than creating Dummy Variables.

```
y_train <- keras::to_categorical(y_train, 10)
y_test  <- keras::to_categorical(y_test, 10)
```

The core structure in Keras is a *model*. The simplest model is a *sequential* model, a linear stack of layers.

```
model <- keras::keras_model_sequential()
model %>%
  layer_dense(units = 256, activation = 'relu', input_shape = c(784)) %>%
  layer_dropout(rate = 0.3) %>%
  layer_dense(units = 128, activation = 'relu') %>%
  layer_dropout(rate = 0.3) %>%
  layer_dense(units = 10, activation = 'softmax')
```

Example 1: A Better MNIST Model

A summary of our model

```
summary(model)

## Model: "sequential_1"
## _________________________________________________________________________
## Layer (type)                 Output Shape              Param #
## =========================================================================
## dense_2 (Dense)              (None, 256)               200960
## _________________________________________________________________________
## dropout (Dropout)            (None, 256)               0
## _________________________________________________________________________
## dense_3 (Dense)              (None, 128)               32896
## _________________________________________________________________________
## dropout_1 (Dropout)          (None, 128)               0
## _________________________________________________________________________
## dense_4 (Dense)              (None, 10)                1290
## =========================================================================
## Total params: 235,146
## Trainable params: 235,146
## Non-trainable params: 0
## _________________________________________________________________________
```

Example 1: A Better MNIST Model

Next we will compile the model:

```
model %>% compile(loss = 'categorical_crossentropy',
                  optimizer = optimizer_rmsprop(),
                  metrics = c('accuracy')
                  )
```

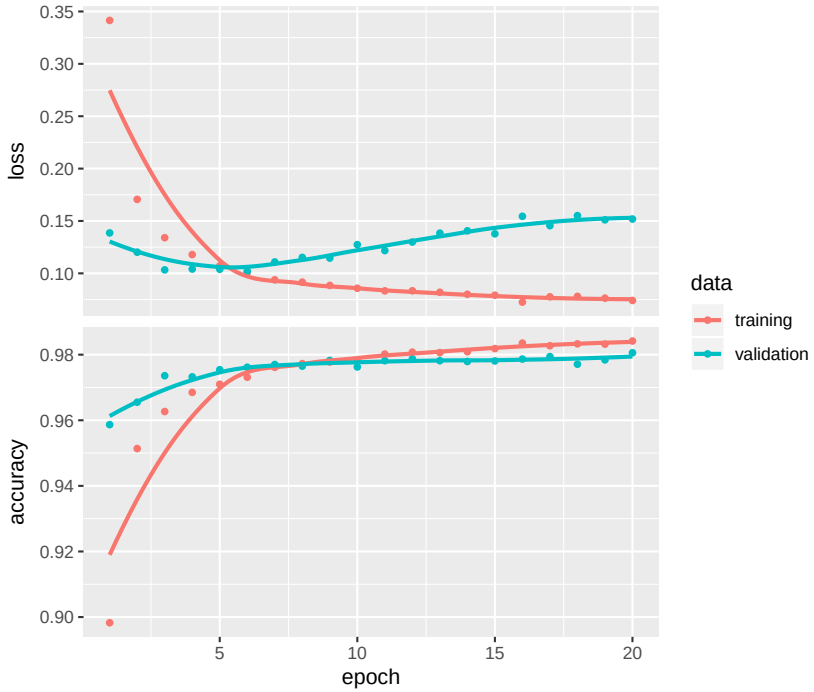
We will use the `fit()` function to train the model for 20 epochs using a mini batch size of 50 and a validation split of 20%.

```
fitted_model <- model %>% fit(x_train, y_train,
                              epochs = 20,
                              batch_size = 50,
                              validation_split = 0.2)
```

The object returned by `fit()` includes loss and accuracy metrics for the training andf validation data.

Example 1: A Better MNIST Model

```
plot(fitted_model)
```



Example 1: A Better MNIST Model

Next lets evaluate the models performance and make predictions, both on the test data.

```
model %>% evaluate(x_test, y_test)

## $loss
## [1] 0.1375173
##
## $accuracy
## [1] 0.9794

head(model %>% predict_classes(x_test), 10)

##  [1] 7 2 1 0 4 1 4 9 5 9
```

Example 2: Housing Prices Prediction

In this example we will build a Neural Network to predict the median price of homes in a Boston suburb in the mid-1970s. The features are characteristics of the suburb, such as the crime rate, the pupil-teacher ratio by town, the local property tax rate, etc.

The Boston Housing Prices dataset is accessible directly from Keras. The data has 506 total data points that are split between 404 training data points and 102 test data points.

```
boston_housing <- keras::dataset_boston_housing()
c(train_data, train_targets) %<-% boston_housing$train
c(test_data, test_targets) %<-% boston_housing$test
str(train_data)

##  num [1:404, 1:13] 1.2325 0.0218 4.8982 0.0396 3.6931 ...

str(train_targets)

##  num [1:404(1d)] 15.2 42.3 50 21.1 17.7 18.5 11.3 15.6 15.6 14.4 ...
```

Example 2: Housing Prices Prediction

We will add columns names and use the package `tibble` for convenience. The target variable is in thousands of dollars.

```
column_names <- c('CRIM', 'ZN', 'INDUS', 'CHAS', 'NOX', 'RM', 'AGE',
                  'DIS', 'RAD', 'TAX', 'PTRATIO', 'B', 'LSTAT')
colnames(train_data) <- column_names
train_df <- tibble::as_tibble(train_data)
head(train_df)
```

```
## # A tibble: 6 x 13
##     CRIM    ZN INDUS  CHAS   NOX    RM   AGE   DIS   RAD   TAX PTRATIO     B
##    <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl>   <dbl> <dbl>
## 1 1.23     0    8.14     0 0.538  6.14  91.7  3.98     4   307    21    397.
## 2 0.0218  82.5  2.03     0 0.415  7.61  15.7  6.27     2   348    14.7  395.
## 3 4.90     0   18.1      0 0.631  4.97 100    1.33    24   666    20.2  376.
## 4 0.0396   0    5.19     0 0.515  6.04  34.5  5.99     5   224    20.2  397.
## 5 3.69     0   18.1      0 0.713  6.38  88.4  2.57    24   666    20.2  391.
## 6 0.284    0    7.38     0 0.493  5.71  74.3  4.72     5   287    19.6  391.
## # ... with 1 more variable: LSTAT <dbl>
```

```
head(train_targets)
```

```
## [1] 15.2 42.3 50.0 21.1 17.7 18.5
```

Example 2: Housing Prices Prediction

We then scale the features to help model optimization. Please note that the test data is not used when calculating the mean and standard deviation but that the test data is scaled using the mean and standard deviation from the training data.

In R these are retrieved as attributes of the scaled `train_data` object.

```
# Normalize training data
train_data <- scale(train_data)

# Use means and standard deviations from training set to normalize test set ***
col_means_train <- attr(train_data, "scaled:center")
col_stddevs_train <- attr(train_data, "scaled:scale")
# Below we use the training sdata mean and standard deviation to scale ther test data!
test_data <- scale(test_data, center = col_means_train, scale = col_stddevs_train)

round(head(train_data[ ,1:5]),2) # Lets check the normalized data
```

```
##       CRIM    ZN INDUS  CHAS   NOX
## [1,] -0.27 -0.48 -0.44 -0.26 -0.17
## [2,] -0.40  2.99 -1.33 -0.26 -1.21
## [3,]  0.12 -0.48  1.03 -0.26  0.63
## [4,] -0.40 -0.48 -0.87 -0.26 -0.36
## [5,] -0.01 -0.48  1.03 -0.26  1.33
## [6,] -0.37 -0.48 -0.55 -0.26 -0.55
```

Example 2: Housing Prices Prediction

Now we will use a 2 layer Neural Network to makes predictions. Here we wrap the model in a function to add dropout later.

```
build_model <- function(dropout = 0) {

  model <- keras::keras_model_sequential()

  model %>%
        layer_dense(units = 64, activation = "relu", input_shape = dim(train_data)[2]) %>%
        layer_dropout(dropout) %>%
        layer_dense(units = 64, activation = "relu") %>%
        layer_dropout(dropout) %>%
        layer_dense(units = 1)

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

Example 2: Housing Prices Prediction

Lets look at the model summary.

```
model <- build_model(); model %>% summary()

## Model: "sequential_2"
## _________________________________________________________________________
## Layer (type)                 Output Shape              Param #
## =========================================================================
## dense_5 (Dense)              (None, 64)                896
## _________________________________________________________________________
## dropout_2 (Dropout)          (None, 64)                0
## _________________________________________________________________________
## dense_6 (Dense)              (None, 64)                4160
## _________________________________________________________________________
## dropout_3 (Dropout)          (None, 64)                0
## _________________________________________________________________________
## dense_7 (Dense)              (None, 1)                 65
## =========================================================================
## Total params: 5,121
## Trainable params: 5,121
## Non-trainable params: 0
## _________________________________________________________________________
```

Example 2: Housing Prices Prediction

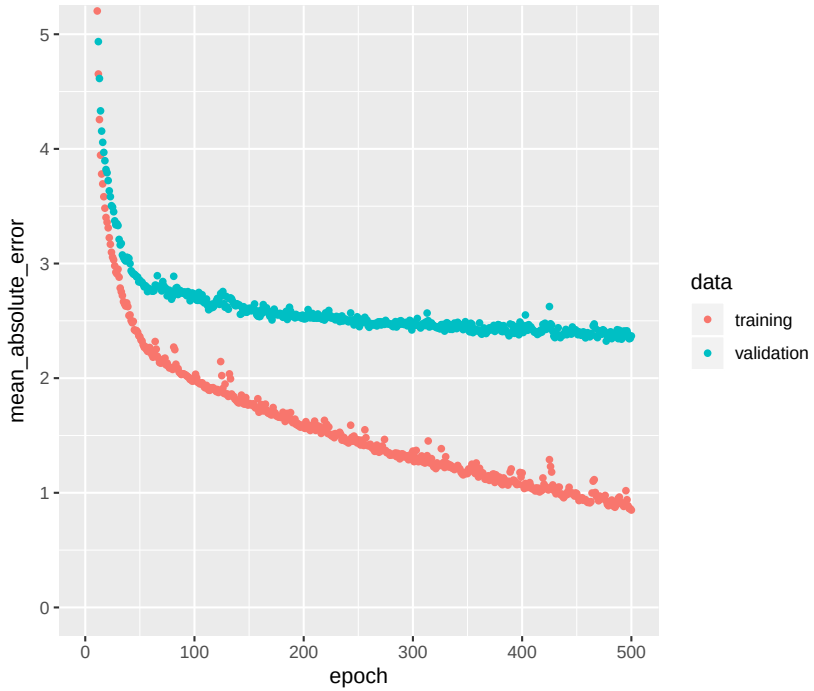
The model is trained for 500 epochs.

```
# 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(".")
  }
)

epochs <- 500
# Fit the model and store training stats
fitted_model_hp <- model %>% fit(train_data,
                         train_targets,
                         epochs = epochs,
                         validation_split = 0.2,
                         verbose = 0,
                         callbacks = list(print_dot_callback)
                         )

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

Example 2: Housing Prices Prediction

We visualize the models training progress using the `fitted_model_hp` variable.

```
plot(fitted_model_hp, metrics = "mean_absolute_error", smooth = FALSE) +
  coord_cartesian(ylim = c(0, 5))
```



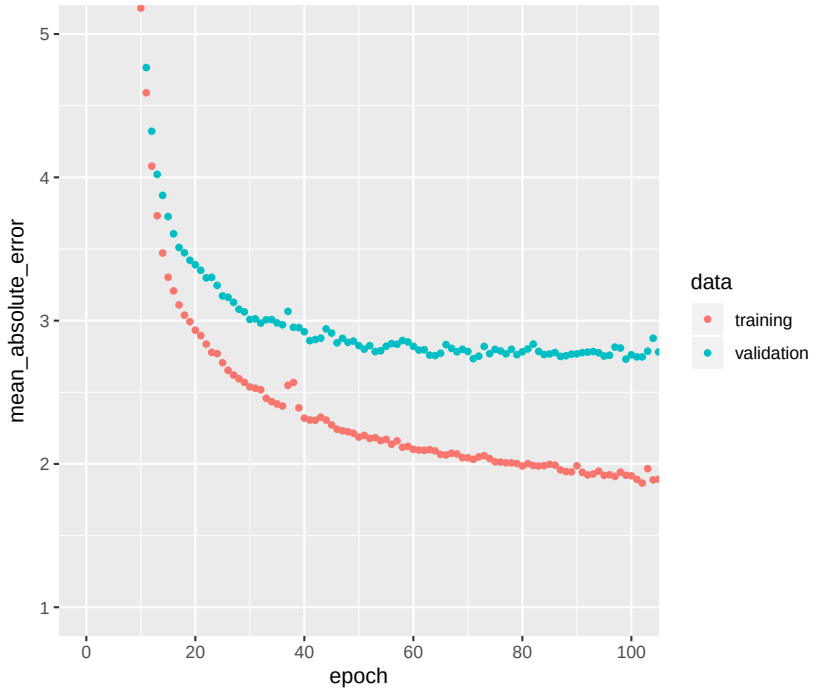
Example 2: Housing Prices Prediction

Validation shows little improvement in out of sample predictive performance after about 200 epochs. Lets update the fit method to stop training when the validation score doesnt improve over a set of epochs.

```
# The patience parameter is the amount of epochs to check for improvement.
early_stop <- callback_early_stopping(monitor = "val_loss", patience = 50)
epochs <- 200
model <- build_model()
fitted_model_hp <- model %>% fit(train_data,
                                 train_targets,
                                 epochs = epochs,
                                 validation_split = 0.2,
                                 verbose = 0,
                                 callbacks = list(early_stop, print_dot_callback)
                                 )

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

Example 2: Housing Prices Prediction

```
plot(fitted_model_hp, metrics = "mean_absolute_error", smooth = FALSE) +
              coord_cartesian(xlim = c(0, 100), ylim = c(1, 5))
```



Example 2: Housing Prices Prediction

Lets check how the model performs on the test set. How does this compare with the MNIST example?

```
c(loss, mae) %<-% (model %>% evaluate(test_data, test_targets, verbose = 0))

paste0("Mean absolute error on test set: $", sprintf("%.2f", mae * 1000))

## [1] "Mean absolute error on test set: $3116.79"
```

Example 2: Housing Prices Prediction

Now lets try to regularize the model in order to improve prediction accuracy.

```
dropout_model <- build_model(dropout = 0.3)

dropout_fitted_model_hp<- dropout_model %>% fit(train_data,
                                                train_targets,
                                                epochs = 50,
                                                verbose = 0,
                                                batch_size = 25
                                                )

c(loss, mae) %<-% (dropout_model %>% evaluate(test_data, test_targets, verbose = 0))

paste0("Mean absolute error of dropout model on test set: $", sprintf("%.2f", mae * 1000))

## [1] "Mean absolute error of dropout model on test set: $3230.27"
```

Example 2: Housing Prices Prediction

Alternatively, we may want to use cross-validation to choose the "optimal" number of epochs. First a trivial exmaple to understand how validation indeces work.

```
fake_train_data <- matrix(seq(1,50,5),10,1)
indices <- sample(1:nrow(fake_train_data)); indices # shuffles the data randomly

##  [1]  1  8  5  7  6  3 10  4  9  2

folds <- cut(indices, breaks = 2, labels = FALSE); folds # divides the range of x into intervals

##  [1] 1 2 1 2 2 1 2 1 2 1

# if you are confused with the last line look at what cut() does below
cut(c(1,2,3,4,5,6,7,8,9,10), breaks = 2, labels = FALSE)

##  [1] 1 1 1 1 1 2 2 2 2 2
```

Example 2: Housing Prices Prediction

Now for the real thing, lets do 5-fold cross validation, first using 50 epochs.

```
k <- 5
indices <- sample(1:nrow(train_data)) # shuffles the data randomly
folds <- cut(indices, breaks = k, labels = FALSE) # divides the range of x into intervals
# and codes the values in x according to which interval they fall.

num_epochs <- 50; all_scores <- c()
# the cross validation loop
for (i in 1:k) {

  # create the validation data for iteration i
  val_indices <- which(folds == i, arr.ind = TRUE)
  # "arr.ind": should array indices be returned when x is an array?
  val_data    <- train_data[val_indices,]
  val_targets <- train_targets[val_indices]

  # create the training data for iteration i
  partial_train_data    <- train_data[-val_indices,]
  partial_train_targets <- train_targets[-val_indices]
  model <- build_model()
  model %>% fit(partial_train_data, partial_train_targets,
                epochs = num_epochs, batch_size = 25, verbose = 0)
  # evaluates the model on validation set
  results <- model %>% evaluate(val_data, val_targets, verbose = 0)
  all_scores <- c(all_scores, results$mean_absolute_error)
}
```

Example 2: Housing Prices Prediction

You can see the difference in estimated Mean Absolute Error across the five folds below:

```
all_scores
```

```
## [1] 2.657487 2.386805 2.609158 2.765371 2.507076
```

```
mean(all_scores)
```

```
## [1] 2.585179
```

Example 2: Housing Prices Prediction

Now lets increase the number of epochs to 200 and start saving the per-epoch validation score.

```
num_epochs <- 200
all_mae_histories <- NULL
for (i in 1:k) {
i
  val_indices <- which(folds == i, arr.ind = TRUE)
  val_data <- train_data[val_indices,]
  val_targets <- train_targets[val_indices]

  partial_train_data <- train_data[-val_indices,]
  partial_train_targets <- train_targets[-val_indices]

  model <- build_model()
  history <- model %>% fit(
    partial_train_data, partial_train_targets,
    validation_data = list(val_data, val_targets),
    epochs = num_epochs, batch_size = 25, verbose = 0
  )
  mae_history <- history$metrics$val_mean_absolute_error
  all_mae_histories <- rbind(all_mae_histories, mae_history)
}
```

Example 2: Housing Prices Prediction

Now we will average the per-epoch MAE scores across all folds (need to use `apply` to apply the `mean` function to columns).

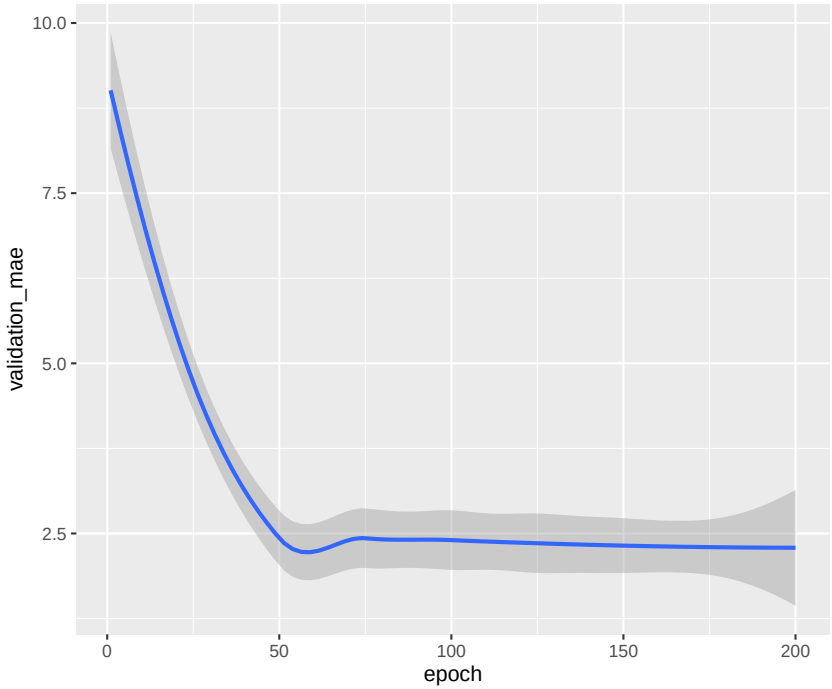
```
average_mae_history <- data.frame(
  epoch = seq(1:ncol(all_mae_histories)),
  validation_mae = apply(all_mae_histories, 2, mean)
)
```

Lets plot the validation MAE (and its smoothed version).

Example 2: Housing Prices Prediction

```
ggplot(average_mae_history, aes(x = epoch, y = validation_mae)) +geom_smooth()
```

```
## `geom_smooth()` using method = 'loess' and formula 'y ~ x'
```



Example 2: Housing Prices Prediction

It looks like the minimum MAE is around 100 epochs so lets use that.

```
model <- build_model()
model %>% fit(train_data, train_targets,
          epochs = 100, batch_size = 25, verbose = 0)
result <- model %>% evaluate(test_data, test_targets)
```

Stochastic Gradient Descent

The weights of the neural network contain the information learned by the network from exposure to the training data.

Initially these weights are filled with small random values. Then the network gradually adjusts those values based on a feedback signal; this adjustment is what we call *training*.

A *training loop* works by repeating the following steps in a loop:

(i) draw a batch of training samples `x` and corresponding targets `y`,

(ii) run the network on `x` (*forward pass*) to obtain predictions `y_pred`,

(iii) compute the loss of the network on the batch (measuring the difference between `y` and `y_pred`), and

(iv) update the weights of the network in a way that slightly reduces the loss on this batch.

A good way to do this (since all the operations in the network are differentiable) is to compute the gradient of the loss with regard to the vector of weights and move these weights in the opposite direction from the gradient to reduce it.

Stochastic Gradient Descent

Recall that a gradient is the derivative of a tensor. It is itself also a tensor and each of its elements indicates the direction and magnitude of the change in the function with respect to which the derivatives are taken when changing an individual element of the tensor.

You can therefore reduce $f(\mathbf{x})$ by moving each element in $f(\mathbf{x})$ in the opposite direction from the gradient. This gives you

$$\Delta x = -\eta \times \text{gradient}$$

where η is a small scaling factor called the *learning rate* due to the fact that the gradient is a local approximation. So the algorithm now becomes: (i) draw a batch of training samples `x` and corresponding targets `y`, (ii) run the network on `x` (*forward pass*) to obtain predictions `y_pred`, (iii) compute the loss of the network on the batch (measure of the difference between `y` and `y_pred`), (iv) compute the gradient of the loss with respect to the network's parameters (backward pass), and (v) move the parameters a little in the opposite direction from he gradient using $\Delta w = -\eta \times \text{gradient}$.

This is the well known *Stochastic Gradient Descent algorithm*. Each iteration over all the training data during mini-batching is called an *epoch*.

Backprobagation Momentum

A common extension to SGD is the application of *momentum*. This helps with two issues: (i) convergence speed and (ii) local minima (where SGD with a small learning rate can easily get stuck).
Previously we assumed that because a function is differentiable we can compute its derivative. This is important because a neural network consists of many tensor operations chained together, each of which has a simple known derivative.

We can thus invoke the chain rule $(f(g(x)))' = f'(g(x)) \times g'(x)$. Applying the chain rule to the computation of the gradient values of a neural network gives rise to an algorithm called *backprobagation*.

Backprobagation starts with he final loss value and works backwards from the top layers to the bottom layers, applying the chain rule to compute the contribution that each parameter has in the loss value.

Tensorflow and other modern DL engines are capable of symbolic differentiation which means that given a series of operations with a known derivative they can compute the gradient function for the chain that maps networks parameter values to gradient values.

Acknowledgements

These notes are based on and follow closely the excellent book "Deep Learning with R" by Franois Chollet (the creator of Keras) with J.J. Allaire.

These notes are also based on and follow closely the excellent tutorials at `wwww.keras.rstudio.tutorials`.

There are a numerous useful Keras related resources online, including at `www.keras.io`.

Additionally, you can refer to the excellent books for more examples and details: "The Deep Learning with R" and "Deep Learning with Python", the latter also by Franois Chollet.

A Deep Learning with Keras cheat sheet is available here `https://github.com/rstudio/cheatsheets/raw/master/keras.pdf`.