14. Deep Learning

Simply put, deep learning involves neural networks with many hidden layers. Deep learning was made possible by increased computing power and availability of more data. However it became a hot trend because it improved performance on many hard-to-solve problems involving pattern recognition with big data. Another reason that deep-learning became popular is that it automated feature engineering. In previous chapters we discussed a few techniques for shaping raw data into features that are useful for machine learning algorithms. With deep learning, the algorithm itself can learn features from the inputs at the same time it is learning the target function for the data.

There are several R packages for deep learning:

- MXNet - an R interface the the MXNet deep learning library
- darch - build on code from Hinton and others
- Keras - R interface to Keras that can run on top of TensorFlow, CNTK, or Theano

14.1 Keras

In this chapter we will focus on Keras running on top of Tensorflow. Keras was designed primarily by Francois Chollet from Google. It is an interface that makes building a deep network relatively painless. The website `https://keras.rstudio.com/index.html` provides a wealth of resources on R Keras.

14.1.1 Installing Keras and Tensorflow

Keras is available in CRAN so you can install it just like any other R package, but we need to also have TensorFlow:

```
install.packages("keras")  # install R Keras
library(keras)     # load Keras
install_keras() # install the library and TensorFlow
```

This is the default CPU-based installation of both Keras and TensorFlow. If you are lucky enough to have a computer with an NVidia GPU and have installed CUDA and cuDNN libraries, you can install the GPU version by adding a GPU argument to the install command:

```
install_keras(tensorflow = "gpu")
```

Having the GPU version can make processing 5 to 10 times faster, but if you are patient the CPU version will get there. If you have a need for speed but no GPU, you can use AWS EC2 CPU or Google Cloud. Both can run RStudio Server so the environment in the cloud should be familiar. The down side of course is that if you plan to do a lot of deep learning this will get expensive.

Keras can run on Windows but it is not recommended. A better option is to have a dual-boot Ubuntu set up if you are on a Windows machine. A linux machine or mac should have no problems.

14.2 Data Representation in TensorFlow

The fundamental data representation in TensorFlow is a *tensor*, which is a generalization of vectors and matrices to an arbitrary number of dimensions, called axes. In an R context, a 1D tensor is like a vector, a 2D tensor is like a matrix and an nD matrix is like an array. The n is sometimes called its rank, but the terminology can be inconsistently used. Scalars are 0D vectors, but R doesn't have scalars, just vectors of length 1 which serve the same purpose. A tensor is defined by:

- Rank - the number of axes
- Shape - in n dimensions
- Type - integer or double

For data that fits into a vector framework, 2D tensors of shape (samples, features) are sufficient. Time series or sequential data requires 3D tensors of shape (samples, timesteps, features). Images require 4D tensors of shape (samples, height, width, channels) or some variation, where channel is 1 for greyscale and 3 for RGB data. Video will require 5D tensors with the 5th axis for frames and the others similar to images.

14.3 Keras in R Example

The website `https://keras.rstudio.com/index.html` has learning resources, reference material and examples. We are going to look at a simple example next from the website. The code is available here: `https://keras.rstudio.com/`.

First, we load the library and set up a few constants used in this example.

```
library(keras)
batch_size <- 128
num_classes <- 10
epochs <- 30
```

The work to be done is divided into 3 phases: (1) prepare the data, (2) build the model, and (3) train and evaluate. First, data preparation.

14.3.1 Data Preparation

The MNIST handwriting recognition data set is built into the Keras package and is already randomly divided into train and test sets. Each observation is a representation of a handwritten digit. The data will take a couple of minutes to load. Once it is loaded you can see in the Environment pane that y_test and y_train are vectors of length 10K and 60K, with integer values representing the true digit label. The train and test x arrays are of dimension 60Kx28x28 and 10Kx28x28. The 28x28 represents the original handwritten images transformed into this grid. Preparing the data involves several steps as shown in the code below.

Code 14.3.1 — Data Preparation. MNIST Data.

```
c(c(x_train, y_train), c(x_test, y_test)) %<-% dataset_mnist()
x_train <- array_reshape(x_train, c(nrow(x_train), 784))
x_test <- array_reshape(x_test, c(nrow(x_test), 784))
x_train <- x_train / 255
x_test <- x_test / 255
y_train <- to_categorical(y_train, num_classes)
y_test <- to_categorical(y_test, num_classes)
```

14.3.2 Build the Model

Before training can start, we have to design the network and then compile it. The following code starts with a sequential model and adds 2 hidden layers with 256 and 128 nodes and a final output layer with 10 nodes, one for each possible digit classification. The hidden layers have relu activation function and the output layer has softmax. These terms are described in detail below. For now, just follow the steps.

In the compile step we specify the loss function, the optimizer and the metrics used to evaluate. The loss function is categorical crossentropy, the optimizer is root mean squared back propagation, and the metric is accuracy.

Code 14.3.2 — Build the Model. MNIST Data.

```
model <- keras_model_sequential()
model %>%
  layer_dense(units = 256, activation = 'relu',
              input_shape = c(784)) %>%
  layer_dropout(rate = 0.4) %>%
  layer_dense(units = 128, activation = 'relu') %>%
  layer_dropout(rate = 0.3) %>%
  layer_dense(units = 10, activation = 'softmax')

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

14.3.3 Train and Evaluate

The fit() method trains the model with the data and parameters that have been previously specified. We requested 30 epochs with verbose output and a batch size of 128. The fit method evaluates itself as it goes through each epoch using 20% of the training data as held-out validation data. The history variable is keeping track of information we can examine later. This process took about 3 minutes on a 2013 MacBook with 8G of RAM.

The results are:

```
Test loss: 0.1028771
Test accuracy: 0.9834
```

Code 14.3.3 — Train and Evaluate. MNIST Data.

```
history <- model %>% fit(
  x_train, y_train,
  batch_size = batch_size,
  epochs = epochs,
  verbose = 1,
  validation_split = 0.2
)

# evaluate
score <- model %>% evaluate(
  x_test, y_test,
  verbose = 0
)

# Output metrics
cat('Test loss:', score[[1]], '\n')
cat('Test accuracy:', score[[2]], '\n')
```

While the model is training, it creates two plots shown in Figure 14.1. The top plot shows training data loss and validation set loss decreasing over the 30 epochs. The bottom plot shows that accuracy on the training data and validation data increase rapidly through the first few epochs and then become fairly stable. The test data accuracy was 98%, similar to what we see for the training and validation data so the model did not overfit the data.

14.4 Deep Learning Basics

Now that we have run through an example of building a network, training, and evaluating, let's step back and discuss some of the terminology that we just used. We have only used a couple of layers in our examples, so we could call this shallow learning instead of deep learning, but the principles and terminology remain the same as we add more layers. The basic algorithm is the same as discussed in the previous chapter on neural networks: the data feeds forward through the network, at each layer being multiplied by the weights, until the output value is calculated. The output has an error which we measure with a loss function, also called the objective function. This acts as a feedback signal to modify

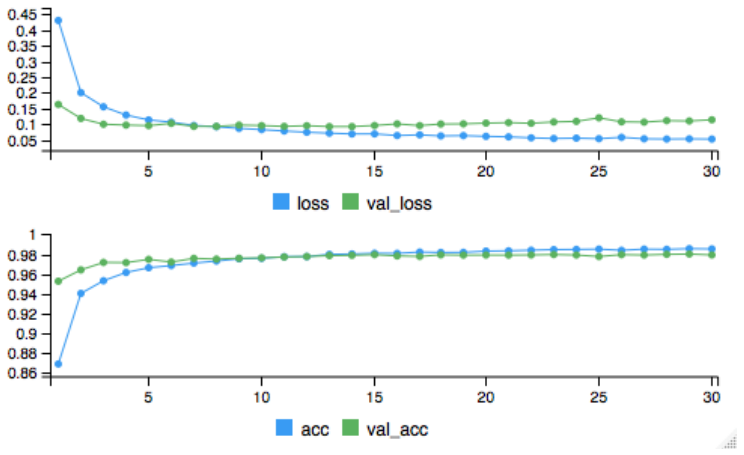

Figure 14.1: Progress Plot while Training on MNIST Data

the weights. This modification is done by the optimizer, typically some variation of the back propagation algorithm we discussed in Chapter 13. Training proceeds forward and backward through hundreds or thousands of iterations. Training stops when the error is below a certain threshold.

So what advantages do deep networks have over shallow ones? The most obvious answer to that is that many layers help learn more complex functions. Beyond that, deep architectures help with feature engineering. In a shallow network the input data must already have been manually manipulated into a good representation for the network. Deep learning can partially automate this by learning features as it learns the function in many layers. According to Challot in his book *Deep Learning with R*, these are the two key principles that make deep learing unique and powerful:

1. increasingly complex representations are learned layer by layer over many iterations
2. the intermediate representations are learned jointly so that as one feature changes, all the others are adjusted automatically

Deep architectures have a lot of moving parts: activation functions, optimization schemes, metrics, and more. We discuss this below but first we discuss the concept of a batch. Deep learning frameworks often don't work on all the data at a time, but process it in batches. The number of batches will be a parameter we can select. As a consequence of handling data in batches, the optimization function is mini-batch stochastic gradient descent. SGD is called stochastic because the selection of observations in a batch are random.

14.4.1 Layers and Units

Designing the optimum number of layers, and units within layers, is not yet a science but more of a trial-and-error approach that you will develop intuitions for as you gain experience. A few guidelines are:

- You want the number of units in an intermediate layer to be greater than or equal to the number of units in the subsequent layer, otherwise you create an information bottleneck.
- For small data sets, avoid overfitting by having a smaller number of hidden layers.

- To search for the best architecture for a given problem, start with a simple model on validation data. Iteratively increase the complexity of a model until you get diminished improvements at which point you may be overfitting.
- Add weight regularization to the hidden layers if overfitting is a concern. This constrains the weights to smaller absolute values. If you recall, we discussed regularization way back in the linear regression chapter. L1 regularization uses the L1 norm which is calculated from the absolute values of the coefficients (weights). L2 uses the L2 norm which is the square of the weights. L2 regularization is sometimes called weight decay. This regularization is only done during training.
- Another regularization technique is dropout, which only occurs in training. This randomly sets a few output weights to 0. Although this seems like an arbitrary approach, it has been shown to reduce overfitting by randomly eliminating the impact of different units that may be tuning into "noise" instead of the function you want them to learn.

14.4.2 Activation Functions

Each layer can have its own activation function. If no activation function is specified, linear activation is assumed. This is typically the choice for the output layer of regression tasks. The sigmoid activation function will output a probability between 0 and 1, so it is often chosen as the activation function for the output layer of a binary classification task. For multi-class classification, the softmax activation is often used because it outputs probabilities for each class. A popular choice for intermediate layers is the relu (rectified linear unit). The relu is linear for positive values but makes negative values 0.

14.4.3 Optimization Schemes and Loss Functions

The optimizer determines how learning proceeds and is associated with a loss function that measures how well it is learning. Although there are a bewildering array of options for all these parameters, there are a few choices that are standard. For regression problems, mean squared error is the standard loss function. For 2-class classification, binary crossentropy is standard, while for multi-class classification, categorical crossentropy is typically used. Crossentropy is a measure of how far off the predictions are from the true values.

14.5 Bring Your Own Data

The previous example showed how to run Keras on some data that is included in the package. What if we want to use our own data? The Check Your Understanding below walks you through the process on the Boston housing data from another package. Note that Keras does include Boston housing data, preprocessed, but the point of this exercise is to see how to use external data with Keras. The following shows how to load the data and arrange it for Keras. First we load the packages we need and take a look at the data. Then we set up some variables that will be useful as we go: N is the size of the data, p is the number of columns, and t identifies the target column. Then we create X and Y, predictors and labels. X is just a data frame and Y is just a regular vector. We set a seed and get a vector of indices we will use to identify training observations as usual. Y_train and Y_test are vectors. X_train and X_test are coerced into matrices for Keras.

```
library(keras)
library(MASS)
df <- Boston[]
str(df)
N <- nrow(df)
p <- ncol(df)
t <- 14   # target column

X <- Boston[, -t]
Y <- Boston[, t]

set.seed(1234)
i <- sample(1:nrow(df), 0.8*nrow(df), replace=FALSE)
X_train <- data.matrix(X[i, -t])
Y_train <- Y[i]
X_test <- data.matrix(X[-i, -t])
Y_test <- Y[-i]
```

We know that we will get better results if we normalize the data. Here is code you can use to normalize the train and test predictors. We are leaving the labels unscaled. First we get a vector of means and a vector of standard deviations from the training data. These are used to scale both the train and test sets. In this way we maintain a firewall between our train and test data.

```
# normalize data
means <- apply(X_train, 2, mean)
stdvs <- apply(X_train, 2, sd)
X_train <- scale(X_train, center=means, scale=stdvs)
X_test <- scale(X_test, center=means, scale=stdvs)
```

Now you are ready to create the model in the Check Your Understanding below.

Check Your Understanding 14.1 — Keras Regression. With Boston Housing Data.

Try the following:

- Create a sequential model.
- Add just one hidden layer with 6 units, relu activation
- Make the input_shape=dim(X_train[[2]]) for the hidden layer
- Add one output layer with one unit.
- Compile the model with loss='mse', optimizer='rmsprop', and metrics='mae'.
- Fit the model on X_train and Y_train, with 10 epochs, batch size of 1 and verbose=1.
- Evaluate with the code shown below.

```
results <- model %>% evaluate(X_test, Y_test, verbose=0)
results$mean_absolute_error
```

Continue experimenting with the model:

- What was your mae above?
- Now change your number of hidden units from 6 to 64. What is the mae now?
- Add another 64-until layer with relu activation between the first hidden layer and the output layer. There is no need to specify an input shape. What is the mae for this model?
- Finally, change your number of epochs from 10 to 100. What is the mae?

In our run of the full model, we got an mae of 2.436 which means our estimate of the home price would be off an average of $2,436, since the medv column is in units of 1000s. What if we want to extract predictions? The following code shows how to extract predictions using predict(). Note we also computed mse and rmse, as well as computed mae manually to confirm that it is the same formula used by Keras.

```
pred <- predict(model, X_test)
cor(pred, Y_test)  # 0.948799
mse <- mean((pred - Y_test)^2)  # mse = 10.47
sqrt(mse)  # rmse = 3.24 (higher than the mae)
mae <- mean(abs(pred - Y_test))  # mae=same as output of Keras
```

Classification

Now that you have practiced using Keras with your own data in a regression problem, let's try a classification problem. This Check Your Understanding uses a wine data set that is available on the github.

Check Your Understanding 14.2 — Keras Classification. Using the Wine Data.

Try the following:

- Load wine_all.csv and examine the data with str().
- Set up variables N, p and t as we did for the regression problem above. Here our target is column 13.
- Set up the X and Y data frames. This time we want to transform Y into an integer. Then replace Y with Y-1 because we want 0-1 values instead of 1-2. Later we will transform it to one-hot encoding.
- Divide X and Y into train and test sets.
- Normalize the X train and test sets.
- Convert Y train and test sets into one-hot encoding with `to_categorical()`.
- Build a model with 16 units in the first hidden layer with relu activation.
- The second layer should have 2 units to match our one-hot encoding of the two-level factor. Give this layer a sigmoid activation.
- Compile the model with binary_crossentropy as the loss function, rmsprop as the optimizer and accuracy as the metric.
- Fit the model.
- What are your results?
- Try changing a few things about your model to see if you can improve the accuracy.

■

In our run of this problem, we got an accuracy of 99.38% with a model of 16 units, relu activation, and running for only 4 epochs. Adding an 8-unit second layer and running for 10 epochs resulted in an accuracy of 99.5%. However, if you get 99% accuracy, you probably don't need to add another layer, it is just going to take a little longer to run. The extra epochs accounted for the slight improvement in accuracy.

14.6 Deep Architecture

The examples in this chapter so far have showed neural networks with 2 hidden layers similar to what we discussed in the last chapter but built with the Keras framework. Where Keras shows its power is in the varieties of architectures that you can build. Next we discuss convolutional neural networks, and recurrent neural networks.

14.6.1 Convolutional Neural Networks

Convolutional neural networks, convnets, work well with image data. The reason is the way these networks learn. A densely connected sequential layer will learn global patterns from the input data. In contrast, the convolutional layer will learn patterns in small 2D windows of the input data. This gives convnets two unique abilities. The first is that once a pattern is learned in one location it will recognize it when translated to another location. The second is that convolutional layers can learn spatial hierarchies. For example, a first convolutional layer could learn local features like edges and subsequent layers could learn how these edges and other features combine to form complex and abstract concepts like faces. This is amazingly powerful. Let's say the initial convolutional layer had an RGB channel from an image input as a 3D tensor. It could learn patterns from a subset of the input space and pass what it learned on to the next layer but now the channel will represent the filter through which it "sees" the input. To make this less abstract let's think of it in terms of the MNIST data where each sample is a single grayscale image of a handwritten digit of shape (28, 28, 1). The first convolution layer takes a small window size like 3x3 and slides this over the input image to create an output shape of (26, 26, 32) which represents a response map of the filter at different locations of the input. As the data passes through convolutional layers it is typically halved at each layer, so the 26x26 will become 13x13, and so on. This is called max pooling and its purpose is to carry forward what is important and leave the rest behind.

14.6.2 Recurrent Neural Networks

Recurrent neural networks can learn from sequential data like time-series or text. Text requires a bit of preprocessing to either one-hot vectors or word embeddings, a discussion of which is beyond our focus here. A recurrent neural network, RNN, has memory, state, which enables it to learn a sequence. The architecture of an RNN contains loops to revisit the state and learn sequences.

A problem with recurrent neural networks is the vanishing gradient problem, a problem that was identified early in the exploration of multi-layer neural networks. As you add layers, the gradient back propagated becomes smaller and smaller so that training is impossible. The Long Short-Term Memory (LSTM) algorithm is one method for avoiding vanishing gradients. The key is keeping the memory data path independent of the back propagation path and giving its own update mechanism.

14.7 Summary

The workflow for Keras involves these steps:

- Define the dimensions and characteristics of the train and test data tensors.
- Define the model.
- Compile the model with an appropraite loss function, optimizer, and metrics.
- Fit the model.
- Evalute the results.

You probably won't hit a home run the first time through these steps so changing some parameters and repeating the above steps may happen multiple times.

14.7.1 Going Further

In this chapter we discussed only one example from the RStudio Keras site. In my experience teaching machine learning to undergraduate students, I find that a significant number of students' personal computers have issues with computation-intensive algorithms like SVM or deep networks. Many overheat or just crash. For this reason, I don't assign a deep learning problem for homework but just discuss the ideas and terminology for students who wish to explore this on their own. The good news is that there is a book that will teach you all you need to know.

The book to read is *Deep Learning with R* by Francois Chollet, the chief developer of Keras, and J. J. Allaire who created the R interface. The book is available here: `https://www.manning.com/books/deep-learning-with-r`.