Neural Network Exercises

Andrew L Beam, PhD

10/25/2017

Exercise 1a

Estimated time: 10 - 15 minutes

In your browser, go to the following page:

http://playground.tensorflow.org/

In the data pane on the left-hand side, select the spiral dataset. Using the options at the top:

- Learning rate
- Activation
- Regularization
- Regularization Rate

and the following architectural options:

- Number of hidden layers
- Number of neurons per layer

construct a neural network that gets the smallest possible testing error. Use this tool to build your intuition about how the various knobs affect the performance of the network.

Exercise 1b

Estimated time: 15 - 30 minutes

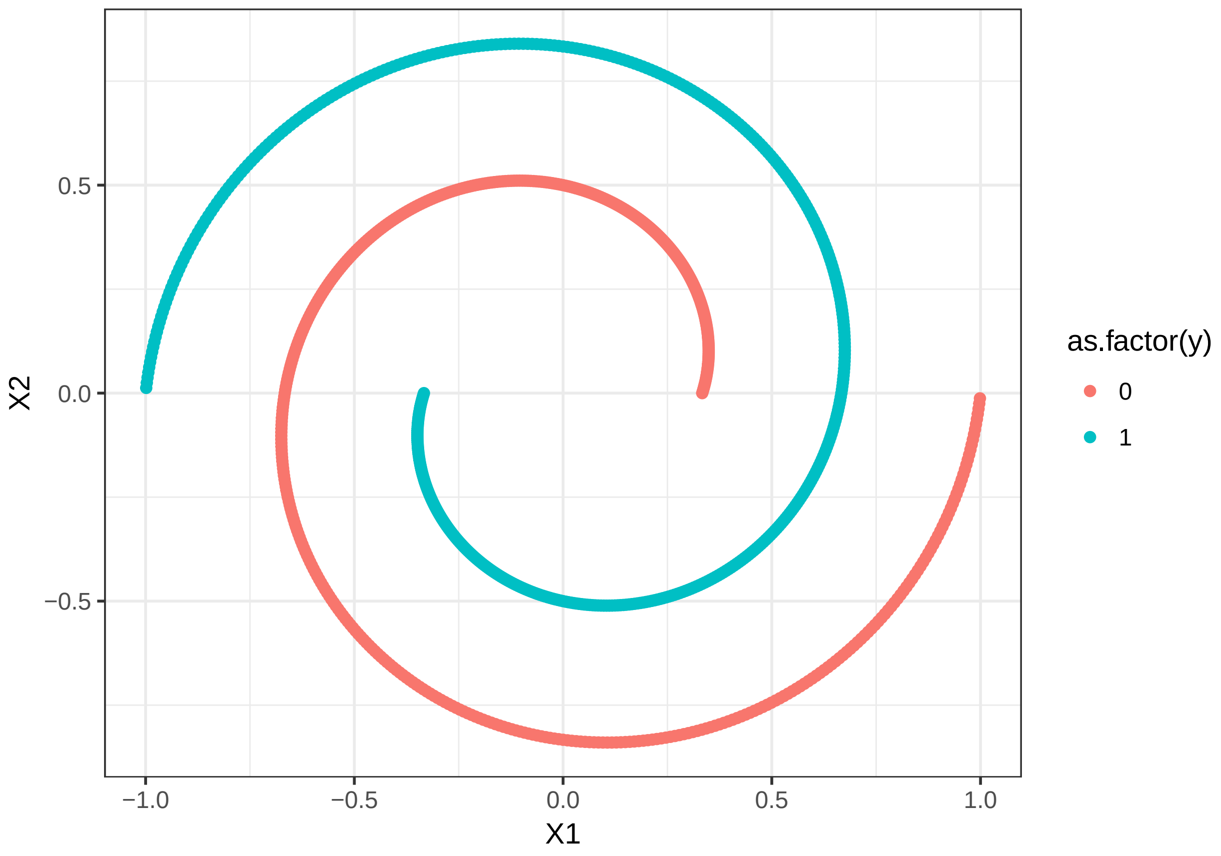
Now we are going to try to implement the neural net you created in your browser in code. Note that the exact details of the two versions of the neural nets might be different due to the fact that we're using a slightly different dataset and a different software framework.

The first thing we will need to do is to load the data and the `keras` library:

```r
library(keras)
load(url("https://www.dropbox.com/s/gaiwn3xvke3vdy3/data.RData?dl=1"))
df <- data.frame(x,y)
x <- as.matrix(x)
```

The `load()` command will load two data frames into your local environment, `x`, `y`. Let's plot the data to see what it looks like:

```r
library(ggplot2)
ggplot(df,aes(x=X1, y=X2, color=as.factor(y))) +
  geom_point() +
  theme_bw()
```



(note you'll need the `ggplot2` library installed to plot the data)

Now let's build a simple neural net with 1 hidden layers using keras just to get a feel for the syntax. The first thing we need to do is to write some boilerplate code to initialize the model:

```
model <- keras_model_sequential()
```

A note about pipes

We are going to use *pipes* which were first made popular by the magrittr package. If you haven't seen this before it can bee a little confusing, but the syntax looks like this: `left_arg %>% right_function` where `left_arg` is some kind of data (such as a data frame that could have itself been generated by a function) that will be passed as the *first* argument to `right_function`. The equivalent functional version would be `right_function(left_arg)`. Piping makes it easier to chain to together a long series of transformations, especially when you have a lot of operations, because the corresponding functional version would have a lot of nested calls that would be difficult to read.

The primary layer we will be using for now is the `layer_dense`. The main parameter for `layer_dense` is the number of hidden units (`units`) it should have. Each unit will perform a weighted combination of all the inputs from the previous layer followed by the application of an activation function (`activation`). In the case of our perceptron, this means we just want one hidden unit followed by a sigmoid activation. It looks something like this:

```
model %>%
  layer_dense(units = 10, input_shape = c(2), activation = 'relu') %>%
  layer_dense(units = 1, activation = 'sigmoid')
```

Note that in the first layer of the network, we needed to tell it how variables to expect (2 in this case). Notice that we don't have to tell it how many samples to expect (this will be automatically inferred), but only how many features we have. The next step is to *compile* the model. This is where the computational graph specified by our model will be compiled along with a loss function so that gradients can be computed. We also must specify how we would like to perform the optimization. For now we will use vanilla stochastic gradient descent, but check the reference for other options.

```
model %>% compile(
  loss = 'binary_crossentropy',
  optimizer = optimizer_sgd(lr = 0.01),
  metrics = c('accuracy')
)
```

Now we are finally ready fit the model:

```
model %>% fit(
  as.matrix(x), y, verbose=1,
  validation_split = 0.2,
  epochs = 100, batch_size = 128
)
```

Note that `model` is modified *in place* so we do not need to assign the result to a variable.

Now it's time for some *grad student descent*. Your goal is to get the best value for the loss function on the *validation* data. Here are some suggestions to get you started:

- Add more hidden layers.
- Change the parameters for the hidden layers. Here are some ones to start with:
- Try different activation functions. Some suggestions include 'relu' and 'elu' but there are many others.
- Add regularization (such as *L1* or *L2*) to the `kernel_regularizer`.
- Check out `layer_dense` for more options.
- Add *dropout* in between your dense layers.
- Use a different optimizer and/or a different learning rate

Check out the end of this blog post for some suggestions of sensible defaults for the above parameters.

Fill in the template below to improve the model:

```
model <- keras_model_sequential()
model %>%
  ## Insert your improvments here

  layer_dense(units = 1, activation = 'sigmoid') %>%
  compile(
    loss = 'binary_crossentropy',
    optimizer = optimizer_sgd(lr = 0.01),
    metrics = c('accuracy')
  ) %>%
  fit(
    x, y, verbose=1,
    validation_split=0.2,
    epochs = 100, batch_size = 128
  )
```

Exercise 2

Estimated time: 30-60 mins

In this exercise we will build a convolutional neural net that can learn to distinguish between dogs and cats. The follow code will load 4 data frames: `x_train`, `y_train`, `x_test`, `y_test`. Each example will be a `64x64x3` tensor, where the last dimensions are the red, green, blue (RGB) color channels.

```
library(keras)
load(url("https://www.dropbox.com/s/ybyqvdb2csh1wq2/cat_dog_data.RData?dl=1"))
## What shape is the data?
dim(x_train)
```

```
## [1] 2000   64   64    3
```

Let's take a look at a few of the pictures:

```
library(magick)
```

```
## Warning: package 'magick' was built under R version 3.4.2
```

```
## Linking to ImageMagick 6.9.9.18
## Enabled features: cairo, fontconfig, freetype, fftw, pango, rsvg, webp
## Disabled features: ghostscript, lcms, x11
```

```
plot(as.raster(x_train[2,,,]))
```

```
plot(as.raster(x_train[1002,,,]))
```

(note you will to install the `magik` package to see the images)

The convolutional neural network will take each image that is a `64x64x3` tensor and perform a 2-dimension convolution across the surface of the image. In fact it will perform many quasi-independent convolutions, each of which is commonly referred to as a *filter*, because it scans local patches of the image for patterns while discarding the rest of the image. First, we are going to initialize the model and set up a simple convolutional/max-pooling block:

```
model <- keras_model_sequential()
model %>%
  ## First the convolutional block with:
  # - 16 filters
  # - Looking at 3x3 patches of the image at a time (kernal_size)
  # - Each convolution is followed by a rectified linear activation (relu)
  # - We will zero-pad the image to keep the output tensor the same size as the input tensor
  layer_conv_2d(filters = 16,
                kernel_size = c(3,3),
                activation = 'relu',
                input_shape = c(64,64,3),
                padding = "same") %>%
  layer_max_pooling_2d(pool_size = c(2, 2))
```

This sets up the first layer which is a convoluational layer with 16 filters, each of which has a `3x3` receptive field. Each of these 16 filters is going to look at the entire image `3x3 = 9` pixels at time and perform the weighted sum of these 9 pixels followed by the `relu` activation. The pooling layer is going to down sample the result by a factor of 2, since it pools over 2 pixels in both the x and y directions. Let's look at the model summary:

```
model %>% summary()
```

```
## ___________________________________________________________________________
## Layer (type)                     Output Shape                  Param #
## ===========================================================================
## conv2d_1 (Conv2D)                (None, 64, 64, 16)            448
## ___________________________________________________________________________
## max_pooling2d_1 (MaxPooling2D)   (None, 32, 32, 16)            0
## ===========================================================================
## Total params: 448
## Trainable params: 448
## Non-trainable params: 0
## ___________________________________________________________________________
```

We see that the output shape of the first convolutional layer takes our `64x64x3` input and produces an output tensor with volume `64x64x16`, where the last dimension is the number of filters we specified. Next, the max pooling layer *down samples* this volume by a factor of two by simply taking the maximum activation over 2x2 grids.

Let's finish off the model by flattening the result of the max pool layer (which discards the spatial information) and feeding the result through a dense sigmoid layer:

```
model %>%
  layer_flatten() %>%
  layer_dense(units = 1, activation = 'sigmoid')

## Look at the full model ##
model %>% summary()
```

```
## ___________________________________________________________________________
## Layer (type)                     Output Shape                  Param #
## ===========================================================================
## conv2d_1 (Conv2D)                (None, 64, 64, 16)            448
## ___________________________________________________________________________
## max_pooling2d_1 (MaxPooling2D)   (None, 32, 32, 16)            0
## ___________________________________________________________________________
## flatten_1 (Flatten)              (None, 16384)                 0
## ___________________________________________________________________________
## dense_1 (Dense)                  (None, 1)                     16385
## ===========================================================================
## Total params: 16,833
## Trainable params: 16,833
## Non-trainable params: 0
## ___________________________________________________________________________
```

Now let's compile and train for 10 epochs:

```
model %>%
  compile(
    loss = 'binary_crossentropy',
    optimizer = 'adam',
    metrics = c('accuracy')
  )  %>%
  fit(x_train, y_train,
    batch_size = 128,
    epochs = 10,
    verbose = 1,
    validation_data = list(x_test, y_test)
)
```

Let's evaluate on the test set:

```
results <- model %>% evaluate(x_test,y_test)
print(paste("Validation loss:", results$loss,sep=' '))
```

```
## [1] "Validation loss: 0.61678747844696"
```

```
print(paste("Validation accuracy:", results$acc,sep=' '))
```

```
## [1] "Validation accuracy: 0.645"
```

Already we should be doing much better than any of our previous models, and this model starts making progress much faster than our MLP (which took 100s - 1000s of epochs and overfit pretty badly). This is because CNNs use the properites of images (2D spatial structure) to prevent overfitting.

In class assignment

Now it's your turn again to improve upon this model. Keep track of your best model and validation accuracy, the person/team with the best result will win a prize! Here are some suggestions:

- 2 convolutional layers before the max pooling layer
- More convolutional/maxpooling blocks. Be careful though, this can get computational expensive quickly.
- A dense layer after the flatten layer but before the sigmoid layer
- Dropout at various places
- Regularization

Put your additions in here to improve the model:

```
model <- keras_model_sequential()
model %>%
  ## Your improvments here
```

```r
  layer_dense(units = 1, activation = 'sigmoid') %>%
  compile(
    loss = 'binary_crossentropy',
    optimizer = 'adam',
    metrics = c('accuracy')
  )  %>%
  fit(x_train, y_train,
    batch_size = 128,
    epochs = 25,
    verbose = 1,
    validation_data = list(x_test, y_test)
)
```