Multi-Layer Perceptrons and MNIST

Zachary Blanks

```
library(keras)
```

How do Neural Networks work?

Before we tackle our MNIST problem, I think it is important that we get an intuitive understanding of how neural networks function. Specifically we are going to investigate the role of gradients and learning rate – some of the key portions of the algorithm. To begin suppose we are trying to predict the test scores of a student by only looking at the number of hours he or she studied. We propose that the grade function takes the form

$$f(x) = \frac{100}{1 + \exp(-0.25x)}.$$

First we will generate some data to work with this function and then talk about what happens when we are actually trying to fit the curve.

```
# We need to set the seed to reproduce our results
# since we're going to inroduce some randomness
set.seed(17)

# We'll determine the number of samples for our model --
# this is a parameter we can change later if we'd like
n_sample = 1000

# First we're going to sample our x values between
# 0 to 20 hours of studying
x <-  matrix(data = sample(0:20, replace = TRUE,
                           size = n_sample),
             nrow = n_sample, ncol = 1)

# We're also going to introduce some randomness to the test scores
eps <- sample(-5:5, replace = TRUE, size = n_sample)

# Now we can generate our test scores
y <- 100 / (1 + exp(-0.25 * x)) + eps

# We're also going to split our data into a training
# and test split so that we can evaluate our model's performance
idx = c(1:n_sample) <= 0.75 * dim(x)[1]
x_train = matrix(x[idx, ])
y_train = y[idx]
x_test = matrix(x[!idx, ])
y_test = y[!idx]
```

Now that we have generated our data, let's tackle this problem epoch by epoch to see how our loss is affected by the various choices that we make. However before we can do that, I would like to show you what the model, is doing on the zeroth epoch – i.e. before it has seen any data.

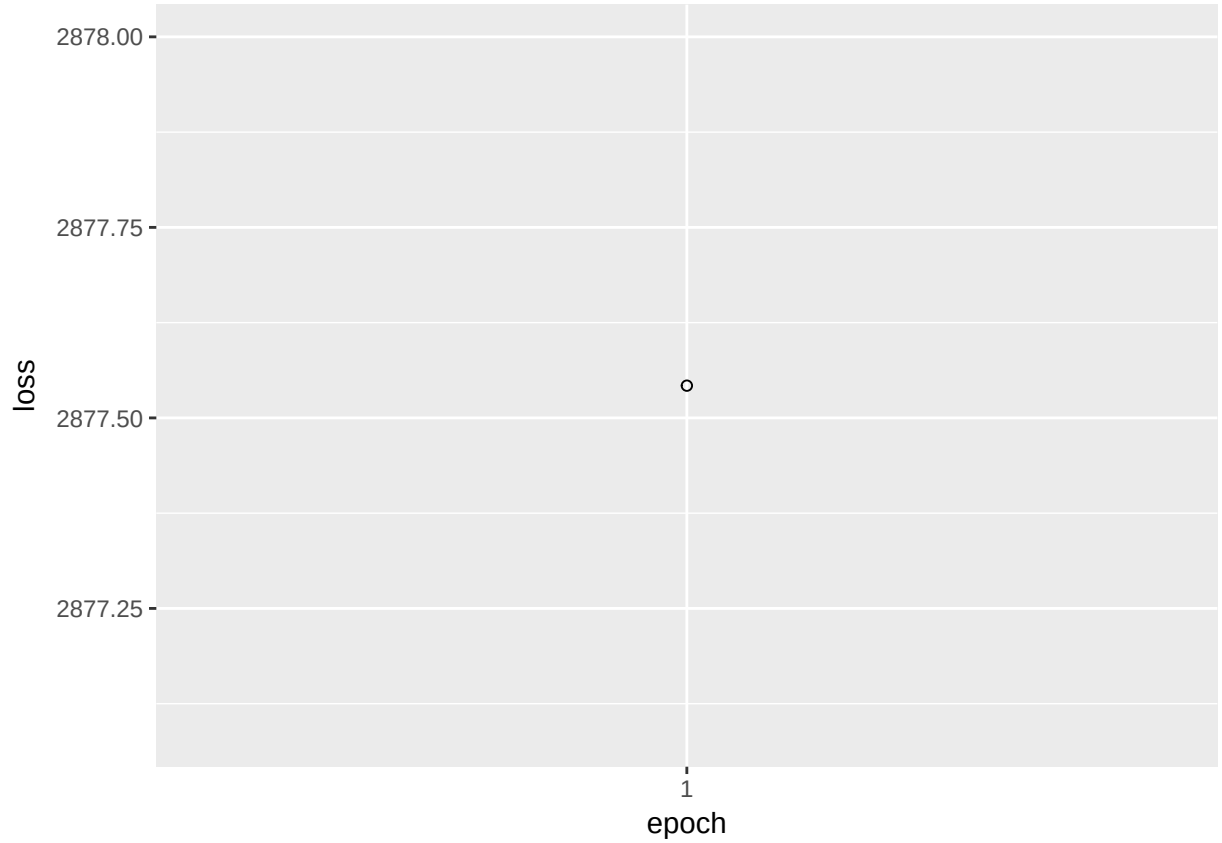
Before we start training our neural network, we set all of the parameters to random values, which in effect correspond to random guesses. From this we get a loss value which we can then use in tandem with the gradient to inform us what direction we need to proceed. Let's see how we would do if we just guessed randomly.

```
random_guess <- runif(n = length(y_train), min = 50, max = 100)
ssr <- sum(y_train - random_guess)^2
cat("The loss on the 0th epoch is: ", ssr)
```

```
## The loss on the 0th epoch is:  67872571
```

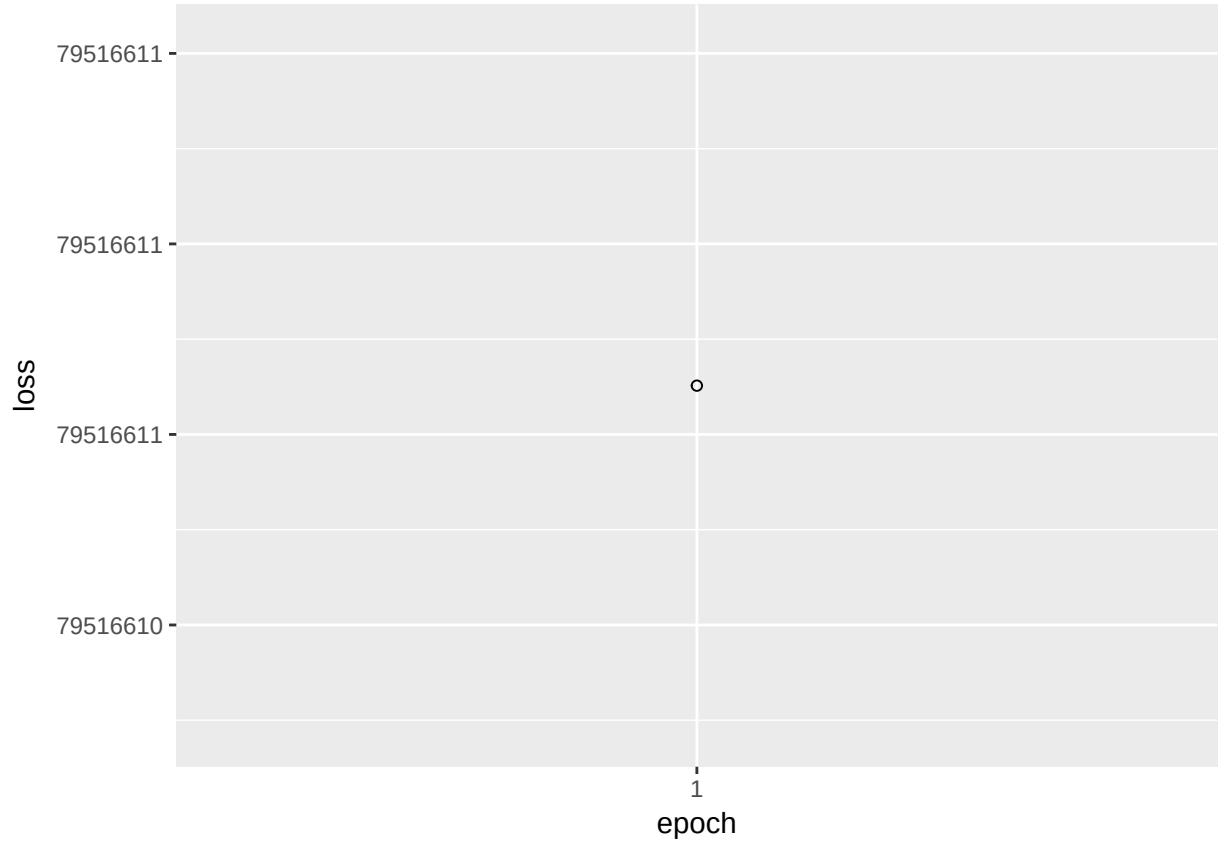
That is an enormous loss! However we were just guessing randomly so it's not particularly surprising that we did not do very well. However even though we did not do well, we have a loss and the gradient can be computed. We can then use these pieces of information to know which direction to proceed. Now let's use the Keras API to see what happens have the first epoch. I won't explain the API too much yet because I do that much more in the MNIST section.

```
# Set the constants for this block -- the learning
# rate and the number of epochs
n_epoch <- 1
learning_rate <- 1e-3

# This is a generic model to help us fit to our data
# don't worry too much about the details yet --
# we will get to that part
model <- keras_model_sequential() %>%
  layer_dense(units = 1, activation = 'sigmoid',
              input_shape = c(NULL, 1)) %>%
  layer_dense(units = 100, activation = 'sigmoid') %>%
  layer_dense(units = 1)

model %>% compile(
  loss = 'mean_squared_error',
  optimizer = optimizer_sgd(lr = learning_rate)
)

history <- model %>% fit(
  x = x_train, y = y_train,
  epochs = n_epoch
)
```



Compare the loss we got after the first epoch to the zeroth epoch. It's smaller, right? This is normally what we expect. We use the gradient and the loss to help us find better parameter values. A key part of the puzzle that I haven't talked about yet is the learning rate. How does that affect the results? Let's reset the model and see what it does to the loss.

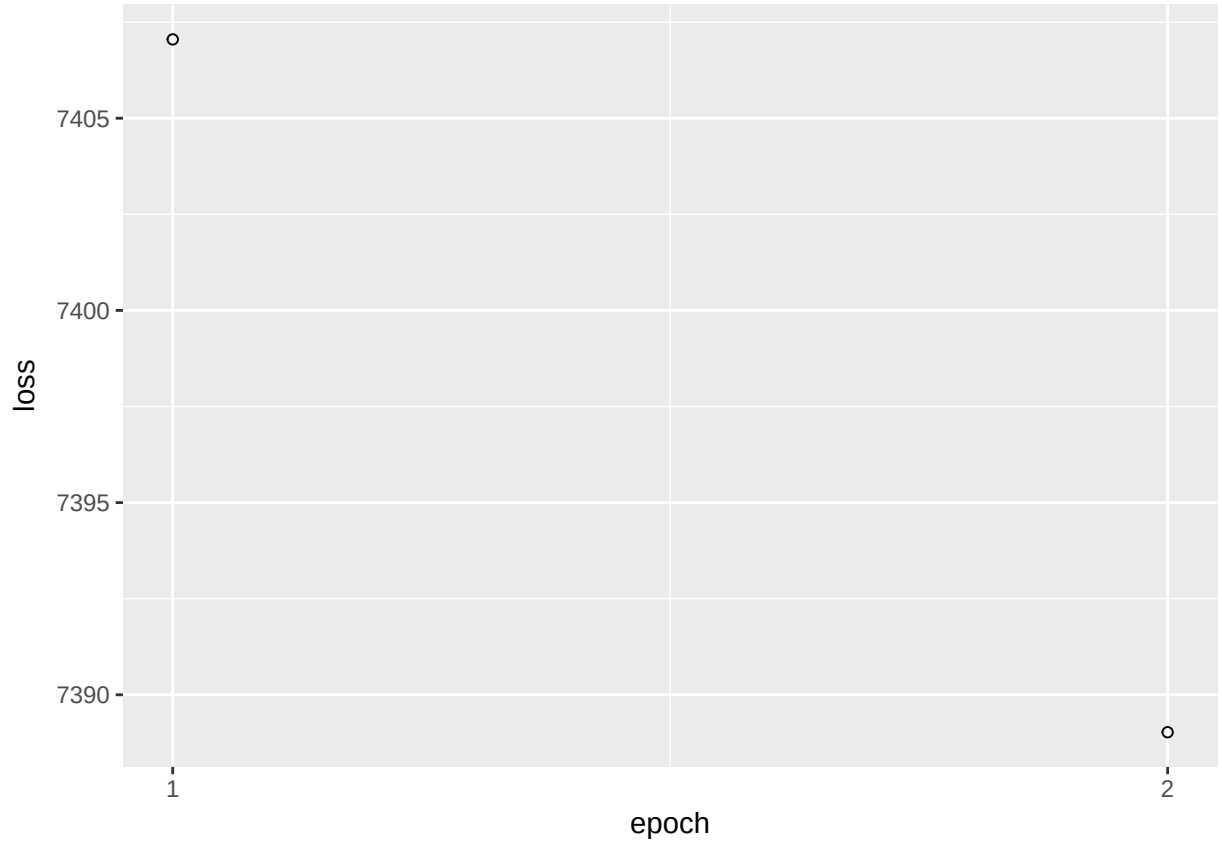
```
n_epoch <- 1

# I want to have a very high learning rate
# to show what it can do
learning_rate <- 1

# We're using the same model as before
model <- keras_model_sequential() %>%
  layer_dense(units = 1, activation = 'sigmoid',
              input_shape = c(NULL, 1)) %>%
  layer_dense(units = 100, activation = 'sigmoid') %>%
  layer_dense(units = 1)

model %>% compile(
  loss = 'mean_squared_error',
  optimizer = optimizer_sgd(lr = learning_rate)
)

history <- model %>% fit(
  x = x_train, y = y_train,
  epochs = n_epoch
)
```



You may have quite a bit of variance on the training loss when compared to the zeroth epoch. Sometimes I got a lower value, sometimes much higher, and in some cases I even got an infinite value. Why is this happening? At a high level, the learning rate controls how much we update our parameters after each epoch with respect to the gradient we computed. Thus a high value means that we are making big changes after every epoch. Go ahead and try running your model for a few epochs and see what happens. On the flip side, let's also investigate what happens when we set our learning rate to a really small value like $1e^{-6}$.

```
n_epoch <- 2

# I want to have a very high learning rate
# to show what it can do
learning_rate <- 1e-6

# We're using the same model as before
model <- keras_model_sequential() %>%
  layer_dense(units = 1, activation = 'sigmoid',
              input_shape = c(NULL, 1)) %>%
  layer_dense(units = 100, activation = 'sigmoid') %>%
  layer_dense(units = 1)

model %>% compile(
  loss = 'mean_squared_error',
  optimizer = optimizer_sgd(lr = learning_rate)
)

history <- model %>% fit(
  x = x_train, y = y_train,
```

```
  epochs = n_epoch
)
```



Look at the difference in the training loss between first and second epoch. It is probably pretty small. You can probably guess why this is happening. As a consequence of only updating our parameter values a tiny bit after each epoch we generally do not get much improvement. While having a low learning rate won't cause wild fluctuations like we saw with a learning rate that is too high, the model will have to run for longer periods of time to get to similar locations on the loss surface when compared to an appropriate learning rate. Like many things in machine learning it's a balancing act.

Now that we have an idea of what the neural network is doing under the hood, let's use them to tackle a classic problem – hand-written digit recognition.

Data Preprocessing

The MNIST data is hosted by Yan LeCun on his website and is commonly used as a benchmark for various deep learning applications, so it's easily accessible and a pretty clean set. Actual data sets, as you might imagine, take quite a bit of work to be ready to run in a neural network. However, that is not the focus of the lecture, and so we will ignore that important aspect of the process.

```
# Get the MNIST data
mnist <- dataset_mnist()

# Split into training and testing
x_train <- mnist$train$x
y_train <- mnist$train$y
```

```
x_test <- mnist$test$x
y_test <- mnist$test$y
```

Because we want to use a multi-layer perceptron (MLP) for this particular example and not a convolutional network (we will do this later), we're going to get the data into a 2-d form.

```
x_train <- array_reshape(x_train, dim = c(dim(x_train)[1],
                                          dim(x_train)[2] * dim(x_train)[3]))

x_test <- array_reshape(x_test, dim = c(dim(x_test)[1],
                                        dim(x_test)[2] * dim(x_test)[3]))
```

An important portion of working with images, similar to other machine learning applications, is to normalize the data. By normalizing our images this helps solve three issues: it speeds up computation time, it guards against vanishing or exploding gradients, and typically improves model performance. While there are many ways to normalize data we're going to use the most common way:

$$\tilde{X} = \frac{X - \mu_X}{\sigma_X}$$

where μ_X represents the mean of our data X and σ_X is the standard deviation. When working with larger data sets, it can be the case where you run into memory issues due to the size of X. There are ways around this problem, the most common being batch normalization which involves taking small batches of X and normalizing it in real time, but this will not be covered during this lecture.

```
x <- rbind(x_train, x_test)
x <- (x - mean(x)) / sd(x)
x_train <- x[1:dim(x_train)[1], ]
x_test <- x[(dim(x_train)[1] + 1):dim(x)[1], ]
rm(x)
```

Finally before we start training our model, we need to one-hot encode our $\mathbf{y}$ vectors so that we can use them in our MLP.

```
y_train <- to_categorical(y_train)
y_test <- to_categorical(y_test)
```

Logistic Regression

While this lecture does focus on using neural networks to tackle unstructured data problems, I first need to make the case that there are appropriate gains to be made by using these more complicated techniques versus more traditional methods such as logistic regression. So first we will run a logistic regression model to benchmark our performance.

To start we need to define a "model" object. With the Keras package, the "model" object is central to all of the methods we will use. The sequential model allows us to stack various layers on top of one other. As a consequence we can improve the modularity of our code, and in my opinion, it is much more readable versus TensorFlow code. The trade off, however, is that we have less control and we may not fully understand what is going on with the code because it is such a high level extension of TensorFlow.

```
logit <- keras_model_sequential()
```

After you have instantiated your model, the next step is to build all of the layers that you would like to have. Note that we have not actually done any computations yet. Instead we are building what is known as a "computation graph." Basically, we are defining all of the parts of our model and after it's fully built then we

run it all at once. The advantage of this computing paradigm is that it allows the TensorFlow back-end to allocate resources more efficiently at run-time versus if it was run serially like most computer code.

```
# Since we're only running a logistic regression model, we only have a
# softmax layer
logit %>%
  layer_dense(units = dim(y_train)[2], activation = 'softmax',
              input_shape = c(dim(x_train)[2]))
```

A few notes on the layer that we just created. First, the "units" argument details how many "nodes" we need for this layer. Since this is the final layer we need as many nodes as we have target classes. When dealing with other layers as I will show you in a bit, we can change this as we desire and is one of the many hyper-parameters that the user controls when dealing with neural networks. Second, for every layer you need specify the activation function you would like to use. A list of them can be found on **https://keras.rstudio.com/reference/index.html**. Finally, when dealing with your first layer (which in this case is also our final layer), you need to specify the input dimension of your data. This allows Keras to figure our what shape the $\mathbf{W}$ matrix needs to be. In every other layer you do not have to specify this parameter since it is inferred from the number of hidden units. This is an instance where Keras may be easier to work with since you don't have to worry about these details, but also may lead to a decreased understanding of what is happening with your code.

If you're curious you can actually see the model details by asking for a summary of the model you've instantiated. This can also act as a good sanity check to make sure you've included everything you want into your model

```
summary(logit)
```

```
## _________________________________________________________________________
## Layer (type)                 Output Shape              Param #
## =========================================================================
## dense_10 (Dense)             (None, 10)                7850
## =========================================================================
## Total params: 7,850
## Trainable params: 7,850
## Non-trainable params: 0
## _________________________________________________________________________
```

We note that there are 7850 parameters to tune and since our images are $\mathbf{X} \in \mathbb{R}^{n \times 784}$ and our target is $\mathbf{y} \in \mathbb{R}^{10}$ and our bias vector is the same dimension of $\mathbf{y}$ we get $784 \times 10 + 10 = 7850$. So everything looks good and we're ready to define additional parts of our model.

The next part of defining our computation graph is to define the loss function, the optimization algorithm, and what metrics to use when evaluating the model. As you might imagine, there are a number of choices for each of these and I encourage you to look at the documentation at the link provided above if you're curious about all your options. We will go with the standard approach of cross-entropy loss, stochastic gradient descent optimization, and accuracy.

```
logit %>% compile(
  loss = 'categorical_crossentropy',
  optimizer = optimizer_sgd(),
  metrics = c('accuracy')
)
```

That's actually everything you need to do to have a model up and running. The best part is that that only took about twenty lines. Moreover, if you want (and we will show shortly), even more complex models you can easily add them with only a few additional lines of code. Now let's run our model.

```
set.seed(17)
history <- logit %>% fit(
  x = x_train, y = y_train,
  epochs = 30, batch_size = 128,
  validation_split = 0.2
)
```

We've saved a "history" object which allows to visualize the training process. A note on some of the arguments we've provided – the "epochs" refers to the number of times we run through the complete data set. For example when the algorithm has gone through every sample and computed and updated the gradient accordingly, it has completed one epoch. The larger this value the longer it takes to finish training and there is usually a point of diminishing return or even over-fitting. The batch size refers to the number of samples you want to use for a given step in the epoch. The larger the batch size the more memory this takes up and the longer it takes to compute the gradient, but you will also get a better estimate. Finally, verbose is just to control how much feedback you want from the algorithm. There are a number of other interesting and useful arguments you can provide, including callbacks which allow you to, among other things, save your model weights after each epoch (trust me this is important when running actual models).

Now we're ready to evaluate our model.

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

```
## $loss
## [1] 0.2774556
##
## $acc
## [1] 0.9202
```

It looks like we got approximately 92% accuracy on the test set. That's pretty impressive for a simple model, but we can do quite a bit better. With just a few simple changes we can get to around 97% accuracy and state of the art at the moment is around 99.8% accuracy.

MLP

We saw that we were able to get 92% accuracy from our logistic regression model. Now let's see if my claim that we can do better with an MLP is true. I'll also showcase how easy it is to scale up the complexity of our model with only a few additional lines of code.

```
mlp <- keras_model_sequential()
mlp %>%
  layer_dense(units = 256, activation = 'relu', input_shape = c(dim(x_train)[2])) %>%
  layer_dense(units = 128, activation = 'relu') %>%
  layer_dense(units = dim(y_train)[2], activation = 'softmax')
```

Let's take a moment to look at what we did there. First, with just two additional lines of code we were able to take our simple logistic model into a two hidden layer highly non-linear MLP. Additionally note that we have created a number of additional hyper-parameters for ourselves – the number of hidden units in each layer, the activation function to use each in each layer, and the number of layers to include. The best way to make these decisions is not easy and I will discuss a way to prioritize hyper-parameter tuning later in the lecture because it can be overwhelming.

We're going to do the same thing we did last time so that we have a fair comparison of our models.

```
mlp %>% compile(
  loss = 'categorical_crossentropy',
  optimizer = optimizer_sgd(),
```

```
  metrics = c('accuracy')
)

mlp %>% fit(
  x = x_train, y = y_train,
  epochs = 30, batch_size = 128,
  validation_split = 0.2,
  verbose = 0
)

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

```
## $loss
## [1] 0.08964297
##
## $acc
## [1] 0.9731
```

Indeed we were able to get around 97% accuracy using the MLP. We can do better than this by making small changes to our code and I encourage you to play around with the settings that made. Change the number of hidden units, add additional layers, try a different optimization method, etc. A big part of working with neural networks involves building an intuition of how certain parameters affect performance. Next we're going to tackle more challenging problems and we will accordingly use more powerful techniques.