L10 Neural networks

Greg Ridgeway

2025-12-01

Table of contents

1 Reading

Read Hastie, Tibshirani, and Friedman (2001) Chapter 11.

Read James et al. (2021) Chapter 10.

Read Y. LeCun, Y. Bengio, and G. Hinton (2015). “Deep learning,” *Nature* 521, 436–444.

Section 5.6 of Deisenroth, Faisal, and Ong (2020).

2 A simple neural network

Consider a simple neural network that has two inputs x_1 and x_2 that enter a linear transformation, which outputs a prediction. Let’s further assume that we wish to minimize squared error, $J(\mathbf{y},\hat{\mathbf{y}}) = \sum(y_i - \hat{y}_i)^2$. This is the same as ordinary least squares, but I introduce it with a neural network framework to move gradually into more complex neural networks. So, the structure of this model is

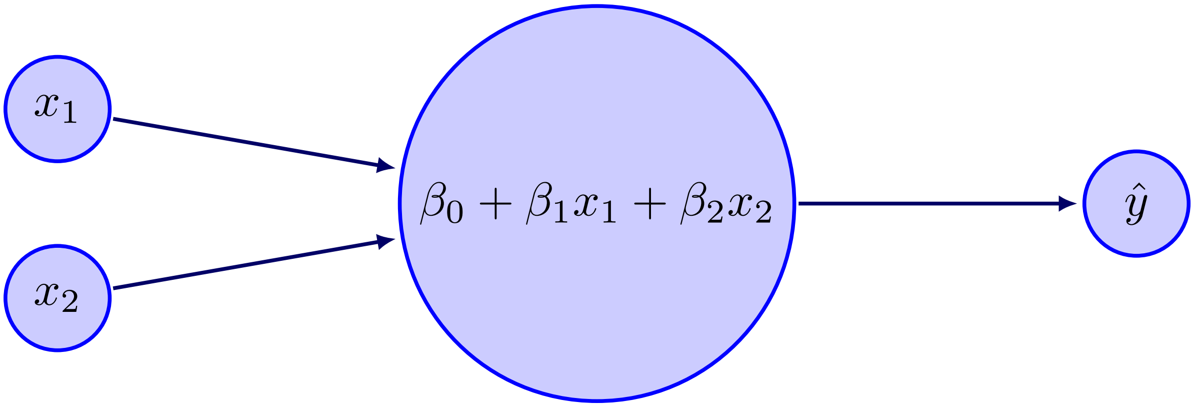

Figure 1: A linear model as a network

We start with some guess for $\hat{\beta}$, perhaps $(0, 0, 0)$. Then, once we pass our data through the neural network, we get predicted values and learn the magnitude of our error by computing $\sum(y_i - \hat{y}_i)^2$. Now we can improve our guess for $\hat{\beta}$ by computing the gradient of J with respect to β. We will chain rule our way to figure out the gradient.

$$
\begin{aligned}
\frac{\partial J}{\partial \hat{y}_i} &= -2(y_i - \hat{y}_i) \\
\frac{\partial \hat{y}_i}{\partial \beta_0} &= 1 \\
\frac{\partial \hat{y}_i}{\partial \beta_1} &= x_{i1} \\
\frac{\partial \hat{y}_i}{\partial \beta_2} &= x_{i2} \\
\frac{\partial J}{\partial \beta_0} &= \frac{\partial J}{\partial \hat{y}_1}\frac{\partial \hat{y}_1}{\partial \beta_0} + ... + \frac{\partial J}{\partial \hat{y}_n}\frac{\partial \hat{y}_n}{\partial \beta_0} \\
&= -2(y_1 - \hat{y}_1)(1) + ... + -2(y_n - \hat{y}_n)(1) \\
&= -2\sum(y_i - \hat{y}_i)
\end{aligned}
$$

So, to reduce the loss function J we need to adjust $\hat{\beta}_0$ as

$$\hat{\beta}_0 \leftarrow \hat{\beta}_0 - \lambda\left(-2\sum(y_i - \hat{y}_i)\right)$$

where λ is the "learning rate." Similarly for $\hat{\beta}_1$ and $\hat{\beta}_2$

$$
\begin{aligned}
\frac{\partial J}{\partial \beta_1} &= \frac{\partial J}{\partial \hat{y}_1}\frac{\partial \hat{y}_1}{\partial \beta_1} + ... + \frac{\partial J}{\partial \hat{y}_n}\frac{\partial \hat{y}_n}{\partial \beta_1} \\
&= -2(y_1 - \hat{y}_1)x_{11} + ... + -2(y_n - \hat{y}_n)x_{n1} \\
&= -2\sum(y_i - \hat{y}_i)x_{i1} \\
\frac{\partial J}{\partial \beta_2} &= -2\sum(y_i - \hat{y}_i)x_{i2}
\end{aligned}
$$

Putting this all together, an algorithm for optimizing this simple neural network would be

$$
\begin{aligned}
\begin{bmatrix}\hat{\beta}_0 \\ \hat{\beta}_1 \\ \hat{\beta}_2\end{bmatrix} &\leftarrow \begin{bmatrix}\hat{\beta}_0 \\ \hat{\beta}_1 \\ \hat{\beta}_2\end{bmatrix} - \lambda \begin{bmatrix}-2\sum(y_i - \hat{y}_i) \\ -2\sum(y_i - \hat{y}_i)x_{i1} \\ -2\sum(y_i - \hat{y}_i)x_{i2}\end{bmatrix} \\
\hat{\beta} &\leftarrow \hat{\beta} + \lambda\mathbf{X}'(\mathbf{y} - \hat{\mathbf{y}})
\end{aligned}
$$

where in the last line the λ absorbed the 2.

3 Computing derivatives in networks

This is a nice example from Deisenroth, Faisal, and Ong (2020) Section 5.6. Let's say you want to compute the derivative of

$$f(x) = \sqrt{x^2 + e^{x^2}} + \cos\left(x^2 + e^{x^2}\right)$$

Using first semester univariate calculus rules, we can compute $f'(x)$ with a little effort.

$$f'(x) = \frac{2x + 2xe^{x^2}}{2\sqrt{x^2 + e^{x^2}}} - \sin\left(x^2 + e^{x^2}\right)\left(2x + 2xe^{x^2}\right)$$

How could we go about turning this into something that a computer could handle? Let's restructure the calculation by listing what we need to compute. In the second column I have compute the derivatives of each of these expressions.

$$\begin{array}{llll}
a & = x^2 & \frac{\partial a}{\partial x} & = 2x \\
b & = e^a & \frac{\partial b}{\partial a} & = e^a \\
c & = a + b & \frac{\partial c}{\partial a} & = 1 \\
 & & \frac{\partial c}{\partial b} & = 1 \\
d & = \sqrt{c} & \frac{\partial d}{\partial c} & = \frac{1}{2\sqrt{c}} \\
e & = \cos(c) & \frac{\partial e}{\partial c} & = -\sin c \\
f & = d + e & \frac{\partial f}{\partial d} & = 1 \\
 & & \frac{\partial f}{\partial e} & = 1
\end{array}$$

In the last row, the value of f is $f(x)$. We can conceptualize the calculation of $f(x)$ with the network shown in Figure 2.

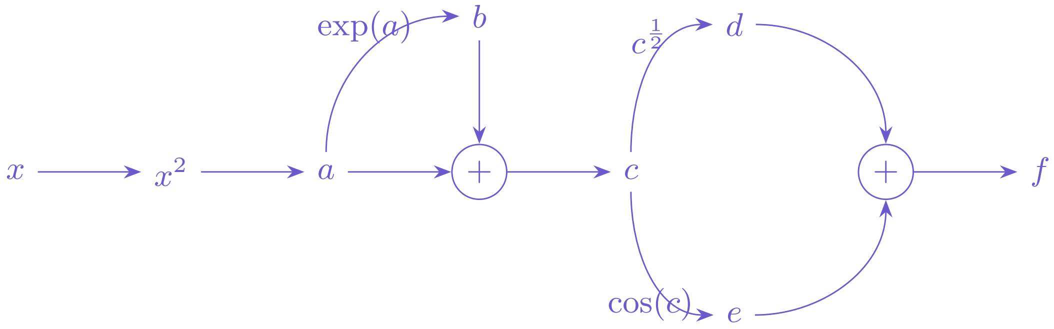

Figure 2: An illustration of evaluating a function structured as a network

Our goal is to compute $\frac{\partial f}{\partial x}$. We will work backwards through Figure 2. We already have $\frac{\partial f}{\partial e}$ and $\frac{\partial f}{\partial d}$. Let's move the derivative one step further to get $\frac{\partial f}{\partial c}$. There are two paths to get to c, one through the square root and the other through the cosine. In standard mathematical notation, we are computing

$$\frac{\partial f}{\partial c} = \frac{\partial}{\partial c}\left(\sqrt{c} + \cos(c)\right)$$

Clearly the derivative is $\frac{1}{2\sqrt{c}} - \sin(c)$. Referring to the network, we can see that this means that we sum over the derivatives associated with the paths through d and e, like

$$\begin{aligned}\frac{\partial f}{\partial c} &= \frac{\partial f}{\partial d}\frac{\partial d}{\partial c} + \frac{\partial f}{\partial e}\frac{\partial e}{\partial c} \\ &= 1 \times \frac{1}{2\sqrt{c}} + 1 \times -\sin(c) \\ &= \frac{1}{2\sqrt{c}} - \sin(c)\end{aligned}$$

The more general rule is that as we move backwards through the network, the chain rule translates to

$$\frac{\partial f}{\partial x_i} = \sum_{x_j \in \text{children}(x_i)} \frac{\partial f}{\partial x_j}\frac{\partial x_j}{\partial x_i}$$

Since d and e are the "children" of c, we sum over them.

Let's move another step back in the network to b applying this approach (note that a is not a child of b). We've already done the work to get $\frac{\partial f}{\partial c}$, so we can plug that in.

$$\begin{aligned}\frac{\partial f}{\partial b} &= \frac{\partial f}{\partial c}\frac{\partial c}{\partial b} \\ &= \left(\frac{1}{2\sqrt{c}} - \sin(c)\right) \times 1\end{aligned}$$

And another step back to a (note that b and c are children of a).

$$\begin{aligned}\frac{\partial f}{\partial a} &= \frac{\partial f}{\partial b}\frac{\partial b}{\partial a} + \frac{\partial f}{\partial c}\frac{\partial c}{\partial a} \\ &= \left(\frac{1}{2\sqrt{c}} - \sin(c)\right) \times e^a + \left(\frac{1}{2\sqrt{c}} - \sin(c)\right) \times 1\end{aligned}$$

And one more step to get us back to x.

$$\begin{aligned}\frac{\partial f}{\partial x} &= \frac{\partial f}{\partial a}\frac{\partial a}{\partial x} \\ &= \left(\left(\frac{1}{2\sqrt{c}} - \sin(c)\right) e^a + \left(\frac{1}{2\sqrt{c}} - \sin(c)\right)\right) \times 2x\end{aligned}$$

If in Figure 2, we choose a specific value for x and propagate it forward through the network, we obtain all the intermediate values (a, b, ...). As we chain rule backwards we use those values to obtain the values of the derivatives. By the time we are back to the beginning we can have already calculated all the necessary terms to obtain our final result.

Let's make this really concrete. Set $x = 2$ and evaluate $f(2)$ and $f'(2)$. Using the definition of $f(x)$ and our traditional hand-calculated derivative we get:

$$\begin{aligned} f(x) &= \sqrt{x^2 + e^{x^2}} + \cos\left(x^2 + e^{x^2}\right) \\ &= 7.19433 \\ f'(x) &= \frac{2x + 2xe^{x^2}}{2\sqrt{x^2 + e^{x^2}}} - \sin\left(x^2 + e^{x^2}\right)\left(2x + 2xe^{x^2}\right) \\ &= -182.8698 \end{aligned}$$

Now let's confirm that we indeed get $f(2) = 7.19433$ when we propagate forward through the network and get $f'(2) = -182.8698$ when we move backwards through the network. In the forward pass, we can compute all of the intermediate values and derivatives of each intermediate value with respect to its "parents."

$$\begin{aligned} a &= x^2 = 4 & \tfrac{\partial a}{\partial x} &= 2x = 4 \\ b &= e^a = e^4 = 54.59815 & \tfrac{\partial b}{\partial a} &= e^a = 54.59815 \\ c &= a + b = 58.59815 & \tfrac{\partial c}{\partial a} &= 1 \\ & & \tfrac{\partial c}{\partial b} &= 1 \\ d &= \sqrt{c} = 7.654943 & \tfrac{\partial d}{\partial c} &= \tfrac{1}{2\sqrt{c}} = 0.06531727 \\ e &= \cos(c) = -0.4606132 & \tfrac{\partial e}{\partial c} &= -\sin c = -0.887601 \\ f &= d + e = 7.19433 & \tfrac{\partial f}{\partial d} &= 1 \\ & & \tfrac{\partial f}{\partial e} &= 1 \end{aligned}$$

In our forward pass we have values computed for every component. Now we move backwards to get the derivative.

$$\begin{aligned} \frac{\partial f}{\partial c} &= \frac{\partial f}{\partial d}\frac{\partial d}{\partial c} + \frac{\partial f}{\partial e}\frac{\partial e}{\partial c} \\ &= 1 \times 0.06531727 + 1 \times -0.887601 \\ &= -0.8222837 \\ \frac{\partial f}{\partial b} &= \frac{\partial f}{\partial c}\frac{\partial c}{\partial b} \\ &= -0.8222837 \times 1 \\ &= -0.8222837 \\ \frac{\partial f}{\partial a} &= \frac{\partial f}{\partial b}\frac{\partial b}{\partial a} + \frac{\partial f}{\partial c}\frac{\partial c}{\partial a} \\ &= -0.8222837 \times 54.59815 + -0.8222837 \times 1 \\ &= -45.71745 \\ \frac{\partial f}{\partial x} &= \frac{\partial f}{\partial a}\frac{\partial a}{\partial x} \\ &= -45.71745 \times 4 \\ &= -182.8698 \end{aligned}$$

Exactly what we computed by hand. This is the general concept of the backpropagation algorithm for fitting neural networks. We will apply simple transformations to linear combinations of inputs and pass them forward in a network. Then to compute the gradient so that we can improve the fit of the model, we do a backwards pass (the backpropagation step). Computers are very good at keeping track of which nodes are children and which ones are parents and can efficiently compute the gradients by calculating and storing all of the intermediate calculations from the forward pass.

You are now ready to apply this to a neural network with hidden layers.

4 Backpropagation for a neural network with a hidden layer

The neural network in the previous section has no better features or capacity than ordinary least squares. To get more capacity to capture interesting shapes and boundaries we will make two changes:

1. add a hidden layer of nodes and
2. add non-linear transformations of the input features.

The network in Figure 3 takes two continuous inputs, passes them through a hidden layer of four nodes, which passes their output to a final output node.

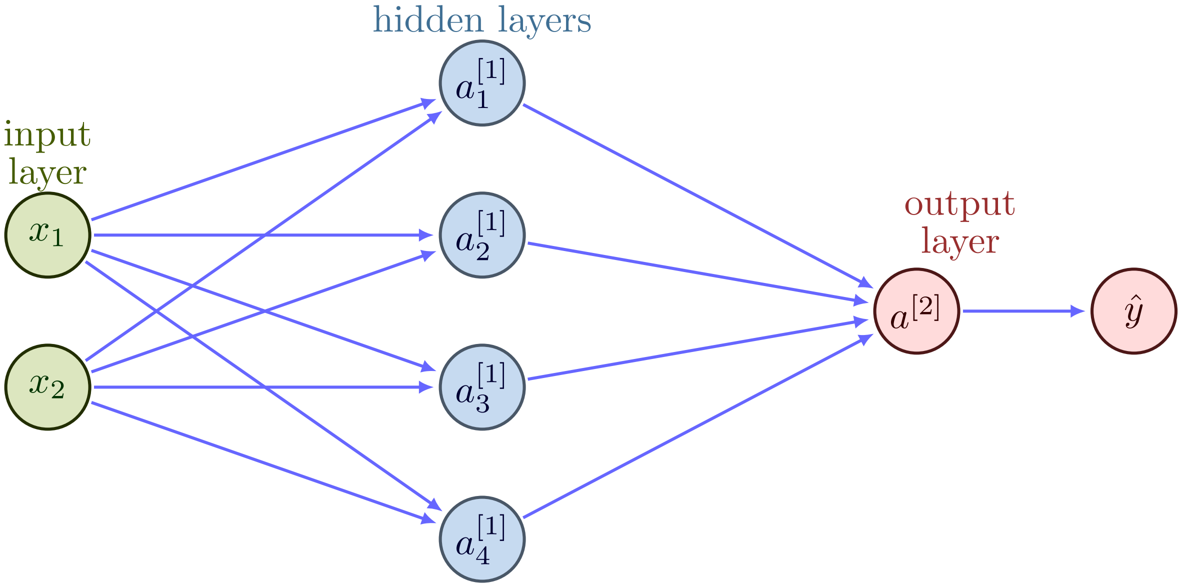

Figure 3: A neural network with a single hidden layer with 4 hidden nodes and a single output

If all the nodes just did linear transformations, then the output node would also just be a linear transformation of x_1 and x_2, offering no improvement over ordinary least squares. Instead, neural networks apply a non-linear "activation function" to the linear combination of features from the previous node. With a single hidden layer and a non-linear activation function, a neural network can capture any shape decision boundary. It is a "universal approximator." The success of deep learning comes from how expressive neural networks can be with layers of these simple functions on top of each other.

The most common activation functions are the sigmoid (also known as the inverse logit transform or expit), the hyperbolic tangent, the rectified linear unit (ReLU… pronounced ray-loo), and the Heaviside function. Figure 4 shows their shape.

```r
x <- seq(-4,4,length.out=100)
# sigmoid, 1/(1+exp(-x))
plot(x, 1/(1+exp(-x)), type="l", lwd=3,
     ylab=expression(sigma(x)))
# hyperbolic tangent
lines(x, (1+tanh(x))/2, col="red", lwd=3)
# Heaviside function, perceptron
lines(x, x>0, col="blue", lwd=4)
# ReLU - Rectified Linear Unit
#   gradient is 0 or 1
lines(x, pmax(0,x), col="orange", lwd=3)
legend(-4,0.8,legend = c("sigmoid","tanh","ReLU","Heaviside"),
       col=c("black","red","orange","blue"),
       lwd=3)
```

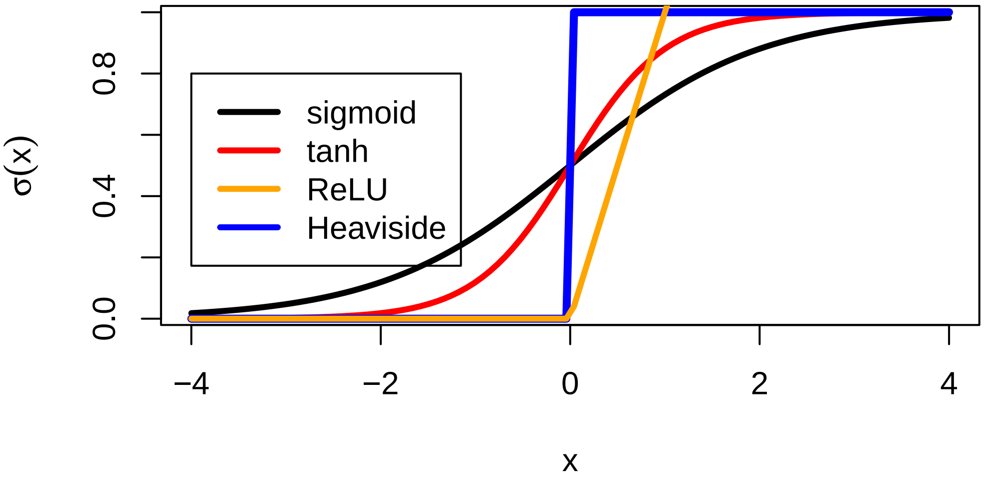

Figure 4: Four common activation functions

Each node applies one of these activation functions to the linear combination of inputs from the previous layer. Used in this manner, they are called "ridge" activation functions.

Returning to the network in Figure 3, the hidden nodes marked with a [1] superscript in them all have the form

$$\mathbf{a}^{[1]} = \begin{bmatrix} a_1^{[1]} \\ a_2^{[1]} \\ a_3^{[1]} \\ a_4^{[1]} \end{bmatrix} = \begin{bmatrix} \sigma\left(\mathbf{w}_1^{[1]\prime}\mathbf{x}\right) \\ \sigma\left(\mathbf{w}_2^{[1]\prime}\mathbf{x}\right) \\ \sigma\left(\mathbf{w}_3^{[1]\prime}\mathbf{x}\right) \\ \sigma\left(\mathbf{w}_4^{[1]\prime}\mathbf{x}\right) \end{bmatrix}$$

In some neural network discussions you will see a constant term separate from the linear transform of the $\mathbf{x}$, called the "bias". Instead here we assume that $\mathbf{x}$ includes a constant 1 as its first element so that $w_{11}^{[1]}$, for example, is that constant term.

The output layer will also apply an activation function to the inputs from the hidden layer.

$$\begin{aligned} \hat{y} = \mathbf{a}^{[2]} &= \sigma\left(\mathbf{w}^{[2]\prime}\mathbf{a}^{[\mathbf{1}]}\right) \\ &= \sigma\left(w_0^{[2]} + w_1^{[2]}a_1^{[1]} + w_2^{[2]}a_2^{[1]} + w_3^{[2]}a_3^{[1]} + w_4^{[2]}a_4^{[1]}\right) \end{aligned}$$

In its full, expanded form it can get rather messy.

$$\hat{y} = \sigma\left(w_0^{[2]} + w_1^{[2]}\sigma\left(\mathbf{w}_1^{[1]\prime}\mathbf{x}\right) + w_2^{[2]}\sigma\left(\mathbf{w}_2^{[1]\prime}\mathbf{x}\right) + w_3^{[2]}\sigma\left(\mathbf{w}_3^{[1]\prime}\mathbf{x}\right) + w_4^{[2]}\sigma\left(\mathbf{w}_4^{[1]\prime}\mathbf{x}\right)\right)$$

It gets a lot more complicated when there are several layers and a lot more nodes in each layer.

Fitting a neural network means optimizing the values of $\mathbf{w}_j^{[1]}$ and $\mathbf{w}_j^{[2]}$ to minimize a loss function.

This time for a loss function we will minimize the negative Bernoulli log-likelihood (sometimes referred to as cross-entropy for binary outputs in the neural network literature).

$$J(\mathbf{y},\hat{\mathbf{y}}) = -\sum_{i=1}^{n} y_i \log(\hat{y}_i) + (1-y_i)\log(1-\hat{y}_i)$$

We will set $\sigma(\cdot)$ to be the sigmoid function. Now we need to compute the gradient with respect to all the parameters. Let's start at the output and work backwards. Recall that the derivative of the sigmoid function is $\sigma'(x) = \sigma(x)(1-\sigma(x))$.

$$\begin{aligned}\frac{\partial J}{\partial \hat{y}_i} &= -\frac{y_i}{\hat{y}_i} + \frac{1-y_i}{1-\hat{y}_i} \\ &= -\frac{y_i - \hat{y}_i}{\hat{y}_i(1-\hat{y}_i)}\end{aligned}$$

Now we need to move to the next level of parameters, the $\mathbf{w}^{[2]}$. Note that

$$\begin{aligned}\frac{\partial \hat{y}_i}{\partial w_j^{[2]}} &= \frac{\partial}{\partial w_j^{[2]}}\sigma\left(\mathbf{w}^{[2]\prime}\mathbf{a}_i^{[1]}\right) \\ &= \sigma\left(\mathbf{w}^{[2]\prime}\mathbf{a}_i^{[1]}\right)\left(1-\sigma\left(\mathbf{w}^{[2]\prime}\mathbf{a}_i^{[1]}\right)\right)a_{ij}^{[1]} \\ &= \hat{y}_i(1-\hat{y}_i)a_{ij}^{[1]}\end{aligned}$$

We can chain rule these together to get the gradient for the output layer weights.

$$\begin{aligned}\frac{\partial J}{\partial w_j^{[2]}} &= \frac{\partial J}{\partial \hat{y}_1}\frac{\partial \hat{y}_1}{\partial w_j^{[2]}} + ... + \frac{\partial J}{\partial \hat{y}_n}\frac{\partial \hat{y}_n}{\partial w_j^{[2]}} \\ &= -\frac{y_1 - \hat{y}_1}{\hat{y}_1(1-\hat{y}_1)}\hat{y}_1(1-\hat{y}_1)a_{1j}^{[1]} - ... - \frac{y_n - \hat{y}_n}{\hat{y}_n(1-\hat{y}_n)}\hat{y}_n(1-\hat{y}_n)a_{nj}^{[1]} \\ &= -(y_1 - \hat{y}_1)a_{ij}^{[1]} - ... - (y_n - \hat{y}_n)a_{nj}^{[1]} \\ &= -\sum (y_i - \hat{y}_i)a_{ij}^{[1]}\end{aligned}$$

A rather complicated idea becomes a rather simple expression. To adjust the output layer weights to make the model perform a little better we need to update as

$$\mathbf{w}^{[2]} \leftarrow \mathbf{w}^{[2]} + \lambda \mathbf{A}^{[1]\prime}(\mathbf{y} - \hat{\mathbf{y}})$$

Now that we have the algorithm for updating the output layer parameters, we can move backwards through the network to update the next layer's parameters.

Reminders

$$\hat{y}_i = a_i^{[2]} = \sigma\left(\mathbf{w}^{[2]\prime}\mathbf{a}_i^{[1]}\right) = \sigma\left(w_1^{[2]}a_{i1}^{[1]} + \ldots + w_4^{[2]}a_{i4}^{[1]}\right)$$

$$a_{ij}^{[1]} = \sigma\left(\mathbf{w}_j^{[1]\prime}\mathbf{x}_i\right) = \sigma\left(w_1^{[1]}x_{i1} + w_2^{[1]}x_{i2}\right)$$

Here we look at layer [1] and adjust the weight on input k associated with hidden node j.

$$\begin{aligned}
\frac{\partial J}{\partial w_{jk}^{[1]}} &= \sum_{i=1}^{n} \frac{\partial J}{\partial \hat{y}_i}\frac{\partial \hat{y}_i}{\partial a_{ij}^{[1]}}\frac{\partial a_{ij}^{[1]}}{\partial w_{jk}^{[1]}} \\
&= \sum_{i=1}^{n} -\frac{y_i - \hat{y}_i}{\hat{y}_i(1-\hat{y}_i)}\hat{y}_i(1-\hat{y}_i)w_j^{[2]}a_{ij}^{[1]}(1-a_{ij}^{[1]})x_{ik} \\
&= -w_j^{[2]}\sum_{i=1}^{n}(y_i - \hat{y}_i)a_{ij}^{[1]}(1-a_{ij}^{[1]})x_{ik}
\end{aligned}$$

Then we update as

$$w_{jk}^{[1]} \leftarrow w_{jk}^{[1]} + \lambda w_j^{[2]}\sum_{i=1}^{n}(y_i - \hat{y}_i)a_{ij}^{[1]}(1-a_{ij}^{[1]})x_{ik}$$

The components of this expression were already computed when we passed forward through the network to make predictions. Now we work backward through the network applying the chain rule over and over to compute the gradients for each layer's parameters. This process of working backwards computing the gradient is the "backpropagation" algorithm (Rosenblatt 1962). The particular nice property of the algorithm is that the gradient is (relatively) easy to calculate knowing the derivatives from the next layer. So, working backwards to get the updates results is reasonably efficient optimization.

5 Examples in two-dimensions

5.1 Classes separated with a curve, but we try a linear neural network

We are going to be using the sigmoid a lot, so I am going to go ahead and make a helper function.

```r
sigmoid <- function(x) {1/(1+exp(-x))}
```

Let's generate a dataset with a non-linear boundary, one for which a logistic regression model would not be ideal.

```
set.seed(20240402)
d <- data.frame(x0=1,
                x1=rnorm(1000),
                x2=rnorm(1000))
d$y <- as.numeric(d$x2 > 2*(sigmoid(4*d$x1)-0.5))
# show class separation
plot(d$x1, d$x2, col=col1and2[d$y+1],
     xlab="x1", ylab="x2")
```

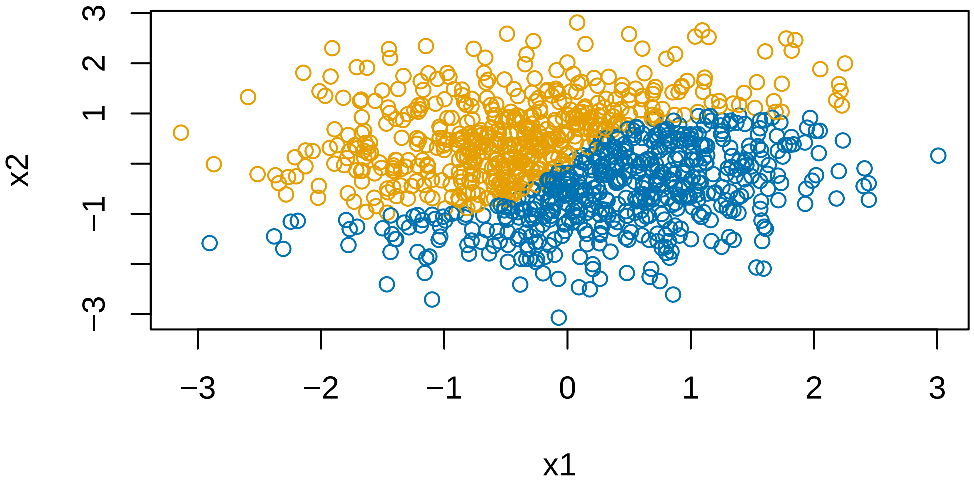

Figure 5: Simulated data with a sigmoid separation boundary

We will use the algorithm we developed earlier to fit a "single-layer linear perceptron" model, like the one shown in Figure 1.

```
# set some starting values
beta <- c(0,-1,0)
learnRate <- 0.0001
X <- as.matrix(d[,c("x0","x1","x2")])

par(mfrow=c(2,3))
for(i in 1:100)
{
  yPred <- as.matrix(d[,1:3]) %*% beta |> as.numeric()
  beta <- beta + learnRate*(t(X) %*% (d$y-yPred))

  if(i %in% c(1,2,5,10,50,100))
  {
    plot(d$x1, d$x2, col=col1and2[(yPred>0.5) + 1],
         xlab="x1", ylab="x2",
         main=paste("Iteration:",i))
    lines(seq(-3,3,length.out=100),
          2*(sigmoid(4*seq(-3,3,length.out=100))-0.5))
  }
}
```

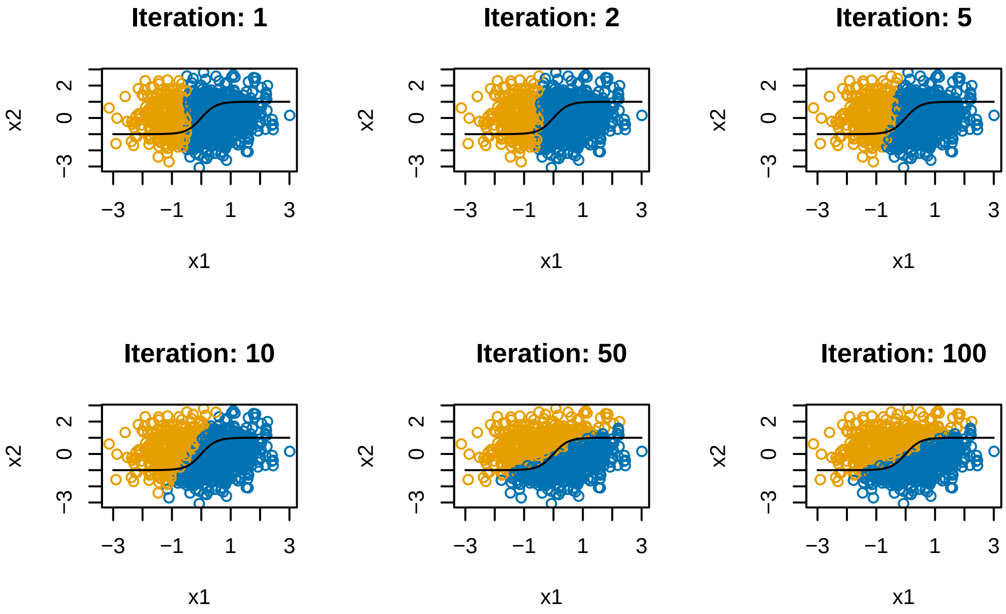

Figure 6: Predicted values from linear perceptron model

In Figure 6, the black curve shows the optimal decision boundary. The blue and orange points show the predicted values. The decision boundary is as best as you can get with a linear model. With a single-layer perceptron in which all relationships are linear, this is the best we can do.

5.2 A circular decision boundary in two dimensions

5.2.1 Searching for a linear boundary

Let's consider a new decision boundary with which a linear neural network will really struggle.

```
d$y <- with(d, as.numeric(x1^2+x2^2 > 1))
plot(d$x1, d$x2, col=col1and2[d$y+1],
     xlab="x1", ylab="x2")
```

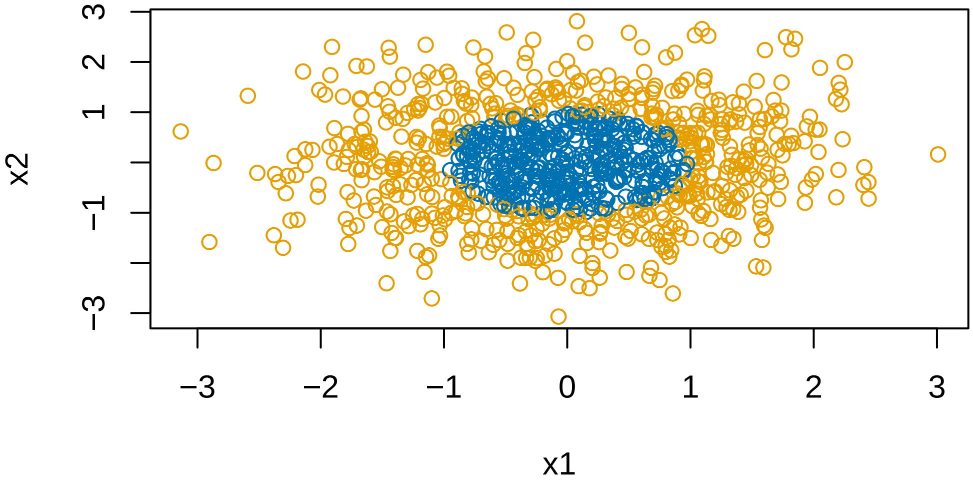

Figure 7: Simulated circular classification boundary

And let's again try the same algorithm

```
# learning algorithm for a single-layer perceptron
beta <- c(0,-1,0)
learnRate <- 0.0001
yPred <- as.matrix(d[,1:3]) %*% beta |> as.numeric()

X <- as.matrix(d[,c("x0","x1","x2")])
```

```r
par(mfrow=c(2,3))
for(i in 1:50)
{
  yPred <- as.matrix(d[,1:3]) %*% beta |> as.numeric()
  beta <- beta + learnRate*(t(X) %*% (d$y-yPred))

  if(i %in% c(1,20,25,30,40,50))
  {
    plot(d$x1, d$x2, col=col1and2[(yPred>0.5) + 1],
         xlab="x1", ylab="x2",
         main=paste("Iteration:",i))
    lines(cos(seq(0,2*pi,length.out=100)),
          sin(seq(0,2*pi,length.out=100)))
  }
}
```

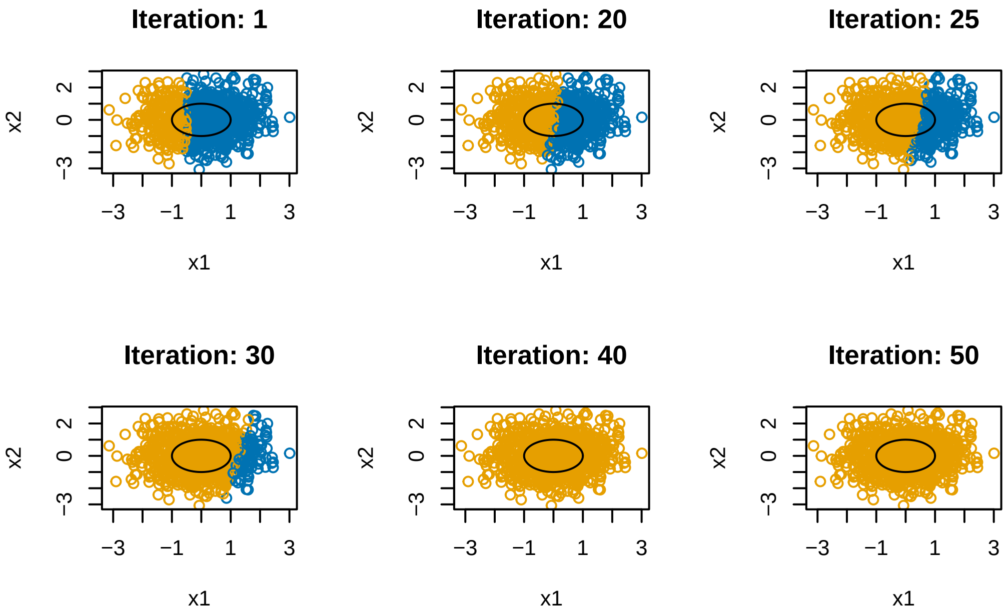

Figure 8: Predicted values from linear perceptron model for circular decision boundary

The mean of y for this simulation is 0.583. The final estimate of β is 0.5800297, 0.0052191, 9.2792344×10^{-4}. In the end, the linear restrictions we have put on the neural network makes it abandon trying to separate the blue and the orange and just predicts all points to be in the majority class, orange.

6 Implementing backpropagation “by hand” for a neural network with a hidden layer

To remedy the problem with the linear neural network, we are going to fit the model shown in Figure 3, with a hidden layer and using the sigmoid activation function. I borrowed this code structure from this blog post on Building A Neural Net from Scratch Using R. The code on the website has several bugs that have been fixed here.

We start by setting up some functions to help us along the way. In this code, the “bias” terms (the “intercept” terms) `b1` and `b2` are stored separately from the weights.

```
# a function to get the number of nodes at each level, input, hidden, and output
getLayerSize <- function(X, y, hidden_neurons) {
  n_x <- nrow(X)
  n_h <- hidden_neurons
  n_y <- nrow(y)

  return( list(n_x=n_x, n_h=n_h, n_y=n_y) )
}

# set up some starting values by randomly selecting numbers between -1 and 1
initializeParameters <- function(X, list_layer_size)
{
  m <- ncol(X)

  n_x <- list_layer_size$n_x
  n_h <- list_layer_size$n_h
  n_y <- list_layer_size$n_y

  W1 <- matrix(runif(n_h * n_x,-1,1),
               nrow = n_h, ncol = n_x, byrow = TRUE) #* 0.01
  b1 <- matrix(rep(0, n_h), nrow = n_h)
  W2 <- matrix(runif(n_y * n_h,-1,1),
               nrow = n_y, ncol = n_h, byrow = TRUE) #* 0.01
  b2 <- matrix(rep(0, n_y), nrow = n_y)

  return( list(W1 = W1, b1 = b1, W2 = W2, b2 = b2) )
}

# run all the input data forward through the network to get predictions
forwardPropagation <- function(X, params, list_layer_size)
{
  m <- ncol(X)
  n_h <- list_layer_size$n_h
  n_y <- list_layer_size$n_y

  W1 <- params$W1
  b1 <- params$b1
  W2 <- params$W2
  b2 <- params$b2

  b1_new <- matrix(rep(b1, m), nrow = n_h)
  b2_new <- matrix(rep(b2, m), nrow = n_y)

  Z1 <- W1 %*% X + b1_new
  A1 <- sigmoid(Z1)
  Z2 <- W2 %*% A1 + b2_new
  A2 <- sigmoid(Z2)

  return( list(Z1 = Z1, A1 = A1, Z2 = Z2, A2 = A2) )
}

# compute the average negative Bernoulli log likelihood
computeCost <- function(X, y, cache)
{
  m <- ncol(X)
  A2 <- cache$A2
  logprobs <- y*log(A2) + (1-y)*log(1-A2)
  cost <- -sum(logprobs/m)
  return (cost)
}

# apply the chain rule working backwards through the network to update weights
backwardPropagation <- function(X, y, cache, params, list_layer_size)
{
  m <- ncol(X)

  n_x <- list_layer_size$n_x
  n_h <- list_layer_size$n_h
  n_y <- list_layer_size$n_y

  A2 <- cache$A2
  A1 <- cache$A1
  W2 <- params$W2
  dZ2 <- A2 - y
  dW2 <- 1/m * (dZ2 %*% t(A1))
  db2 <- matrix(1/m * sum(dZ2), nrow = n_y)
  db2_new <- matrix(rep(db2, m), nrow = n_y)

#  dZ1 <- (t(W2) %*% dZ2) * (1 - A1^2)
  dZ1 <- (t(W2) %*% dZ2) * A1*(1 - A1)
  dW1 <- 1/m * (dZ1 %*% t(X))
  db1 <- matrix(1/m * rowSums(dZ1), nrow = n_h)
  db1_new <- matrix(rep(db1, m), nrow = n_h)

  return( list(dW1 = dW1, db1 = db1, dW2 = dW2, db2 = db2) )
}

# take a gradient descent step
updateParameters <- function(grads, params, learning_rate)
{
  W1 <- params$W1
  b1 <- params$b1
  W2 <- params$W2
  b2 <- params$b2

  dW1 <- grads$dW1
  db1 <- grads$db1
  dW2 <- grads$dW2
  db2 <- grads$db2

  W1 <- W1 - learning_rate * dW1
  b1 <- b1 - learning_rate * db1
  W2 <- W2 - learning_rate * dW2
  b2 <- b2 - learning_rate * db2

  return( list(W1 = W1, b1 = b1, W2 = W2, b2 = b2) )
}
```

Now that all those functions are set up, let's put all the steps in order and do one backpropagation step.

```
# set up the data
X <- as.matrix(cbind(d$x1, d$x2))
y <- as.matrix(d$y, ncol=1)
```

```
X <- t(X)
y <- t(y)
```

Step 0. Set up the network

```
layer_size <- getLayerSize(X, y, hidden_neurons = 4)
layer_size
```

```
$n_x
[1] 2

$n_h
[1] 4

$n_y
[1] 1
```

```
curr_params <- initializeParameters(X, layer_size)
curr_params
```

```
$W1
           [,1]       [,2]
[1,]  0.1086112  0.3794711
[2,] -0.9294748 -0.7302107
[3,]  0.3111992  0.6331077
[4,]  0.8354082  0.1209820

$b1
     [,1]
[1,]    0
[2,]    0
[3,]    0
[4,]    0

$W2
            [,1]       [,2]       [,3]        [,4]
[1,] -0.07868139 0.04723443 0.05089913 -0.02138735

$b2
     [,1]
[1,]    0
```

Step 1. Make predictions moving forward, storing key intermediate values

```
fwd_prop <- forwardPropagation(X, curr_params, layer_size)
# the linear combinations of inputs
fwd_prop$Z1[,1:5]
```

```
           [,1]         [,2]       [,3]        [,4]        [,5]
[1,] -0.4632792  0.001896368  0.3991996  0.05738323 -0.06177666
[2,]  0.1696516 -0.416297471 -2.0773103  0.11969689 -0.21375627
[3,] -0.6426956  0.077616547  0.9022255  0.05421843 -0.04305224
[4,]  0.6545844  0.459247219  1.5823260 -0.23747315  0.35001233
```

```
# sigmoid transform of the linear combinations of inputs
fwd_prop$A1[,1:5]
```

```
          [,1]      [,2]      [,3]      [,4]      [,5]
[1,] 0.3862082 0.5004741 0.5984954 0.5143419 0.4845607
[2,] 0.5423115 0.3974031 0.1113218 0.5298885 0.4467635
[3,] 0.3446375 0.5193944 0.7114066 0.5135513 0.4892386
[4,] 0.6580428 0.6128356 0.8295337 0.4409091 0.5866206
```

```
# linear transform of hidden layer
fwd_prop$Z2[1:5]
```

```
[1] -0.001303673 -0.007277098 -0.023363776  0.001269283 -0.004667737
```

```
# sigmoid transform of linear combo from hidden layer
fwd_prop$A2[1:5]
```

```
[1] 0.4996741 0.4981807 0.4941593 0.5003173 0.4988331
```

Step 2. Evaluate loss function

```
cost <- computeCost(X, y, fwd_prop)
cost
```

```
[1] 0.6933894
```

Step 3. Compute gradient moving back through the network

```
back_prop <- backwardPropagation(X, y, fwd_prop, curr_params, layer_size)
# gradient for the w1
back_prop$dW1
```

```
              [,1]          [,2]
[1,]  2.706173e-04  3.478153e-05
[2,] -1.091090e-04  1.241427e-05
[3,] -1.524197e-04 -1.168966e-05
[4,]  7.629919e-05 -1.840025e-05
```

```
# gradient for the b1
back_prop$db1
```

```
              [,1]
[1,]  1.366440e-03
[2,] -5.919621e-05
[3,] -5.751186e-04
[4,]  1.765536e-04
```

```
# gradient for the w2
back_prop$dW2
```

```
            [,1]        [,2]        [,3]        [,4]
[1,] -0.04223389 -0.03869427 -0.04296879 -0.04469599
```

```
# gradient for the b2
back_prop$db2
```

```
            [,1]
[1,] -0.08326909
```

Step 4. Gradient descent step

```
curr_params <- updateParameters(back_prop, curr_params, learning_rate = 0.01)
curr_params
```

```
$W1
           [,1]       [,2]
[1,]  0.1086085  0.3794707
[2,] -0.9294737 -0.7302108
[3,]  0.3112008  0.6331078
[4,]  0.8354074  0.1209822

$b1
              [,1]
[1,] -1.366440e-05
[2,]  5.919621e-07
[3,]  5.751186e-06
[4,] -1.765536e-06

$W2
            [,1]       [,2]       [,3]        [,4]
[1,] -0.07825906 0.04762137 0.05132882 -0.02094039

$b2
             [,1]
[1,] 0.0008326909
```

And that's one step of the backpropagation algorithm. Repeating Steps 1, 2, 3, and 4 will push the parameters towards values that improve the fit of the neural network to the data.

It is possible to create marvelously complex neural networks and apply the chain rule like crazy to tune the parameters of the neural network to fit the data. We use gradient descent, just like we did for optimizing logistic regression models, to learn the neural network. Most implementations of backpropagation today use some variation of *stochastic gradient descent*. Stochastic gradient descent looks largely the same as described here, except that the gradient is computed using a subsample of the data rather than the entire dataset. This can greatly speed up the computation and avoid locally minima, but can require more iterations to converge.

Let's wrap these gradient steps into a `trainModel()` function that will iterate for `num_iterations` with a learning rate of `lr`.

```
trainModel <- function(X, y, num_iteration, hidden_neurons, lr)
{
  layer_size <- getLayerSize(X, y, hidden_neurons)
  init_params <- initializeParameters(X, layer_size)
  cost_history <- rep(NA, num_iteration)
  for (i in 1:num_iteration) {
    fwd_prop <- forwardPropagation(X, init_params, layer_size)
    cost <- computeCost(X, y, fwd_prop)
    back_prop <- backwardPropagation(X, y, fwd_prop, init_params, layer_size)
    update_params <- updateParameters(back_prop, init_params, learning_rate = lr)
    init_params <- update_params
    cost_history[i] <- cost

    if (i %% 10000 == 0) cat("Iteration", i, " | Cost: ", cost, "\n")
  }

  model_out <- list("updated_params" = update_params,
                    "cost_hist" = cost_history)
  return (model_out)
}


makePrediction <- function(X, y, modNN, hidden_neurons)
{
  layer_size <- getLayerSize(X, y,
                             nrow(modNN$updated_params$W1))
  params <- modNN$updated_params
  fwd_prop <- forwardPropagation(X, params, layer_size)

  return(fwd_prop$A2)
}
```

Let's first try fitting the neural net to the "circle" data, first using a network with two hidden nodes.

```
train_model <- trainModel(X, y,
                          hidden_neurons = 2, # are two hidden nodes enough?
                          num_iteration = 50000, lr = 0.1)
```

```
Iteration 10000  | Cost:  0.4590469
Iteration 20000  | Cost:  0.449592
Iteration 30000  | Cost:  0.4373529
Iteration 40000  | Cost:  0.4355029
Iteration 50000  | Cost:  0.4343475
```

```
plot(train_model$cost_hist, type="l",
     xlab="Iteration", ylab="Negative Bernoulli log likelihood")
```

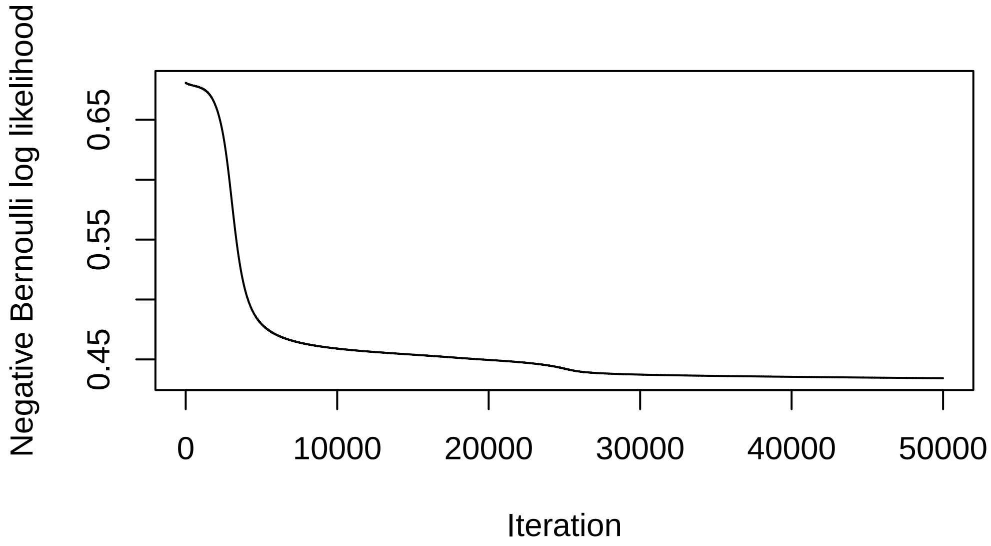

Figure 9: Negative Bernoulli log likelihood by gradient descent iteration, neural network with 2 hidden nodes

The plot shows the reduction in the negative Bernoulli log likelihood with each iteration. But did it fit the data well?

```
y_pred <- makePrediction(X, y, train_model)

plot(X[1,], X[2,], col=col1and2[(y_pred[1,]>0.5) + 1],
     xlab="x1", ylab="x2")
lines(cos(seq(0,2*pi,length.out=100)),
      sin(seq(0,2*pi,length.out=100)))
```

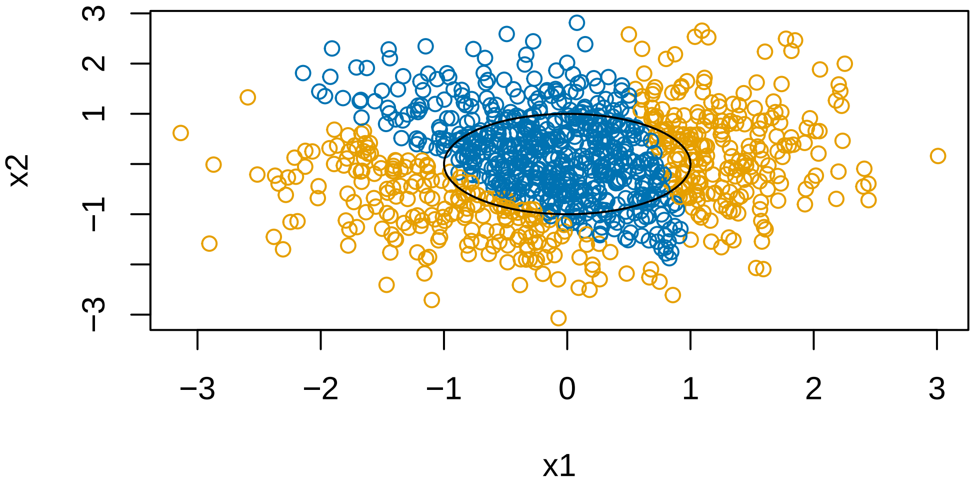

Figure 10: Predictions from a neural network with two hidden nodes

Clearly with two hidden nodes, the neural network does not have sufficient complexity or capacity to capture the elliptical boundary. So, let's try again with a network with four hidden nodes.

```r
train_model <- trainModel(X, y,
                          hidden_neurons = 4,
                          num_iteration = 50000, lr = 0.1)
```

```
Iteration 10000  | Cost:  0.138251
Iteration 20000  | Cost:  0.08347349
Iteration 30000  | Cost:  0.06709865
Iteration 40000  | Cost:  0.05971714
Iteration 50000  | Cost:  0.05542842
```

```r
plot(train_model$cost_hist, type="l")
```

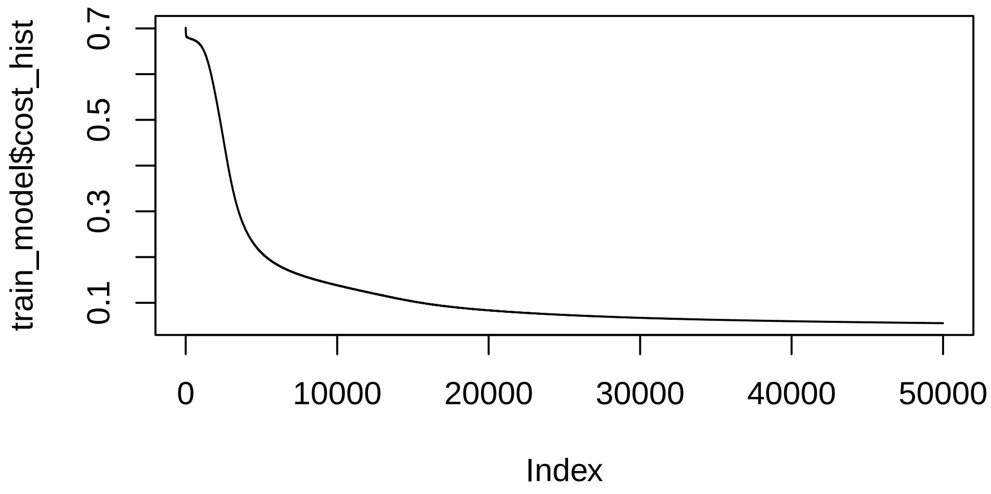

Figure 11: Negative Bernoulli log likelihood by gradient descent iteration, neural network with 4 hidden nodes

We seem to get better predictive performance here since we got the cost much lower. Did we successfully capture the elliptical boundary?

```
y_pred <- makePrediction(X, y, train_model)

dAll <- expand.grid(x1=seq(-4.1,4.1,length.out=100),
                    x2=seq(-4.1,4.1,length.out=100))
dAll$y <- 0
dAll$y_pred <- makePrediction(t(as.matrix(dAll[,1:2])),
                              matrix(dAll$y,nrow=1),
                              train_model) |> as.numeric()



plot(dAll$x1, dAll$x2, col=col1and2transparent[(dAll$y_pred>0.5) + 1],
     xlab="x1", ylab="x2", pch=15)
points(X[1,], X[2,], col=col1and2[(y[1,]>0.5) + 1])
```

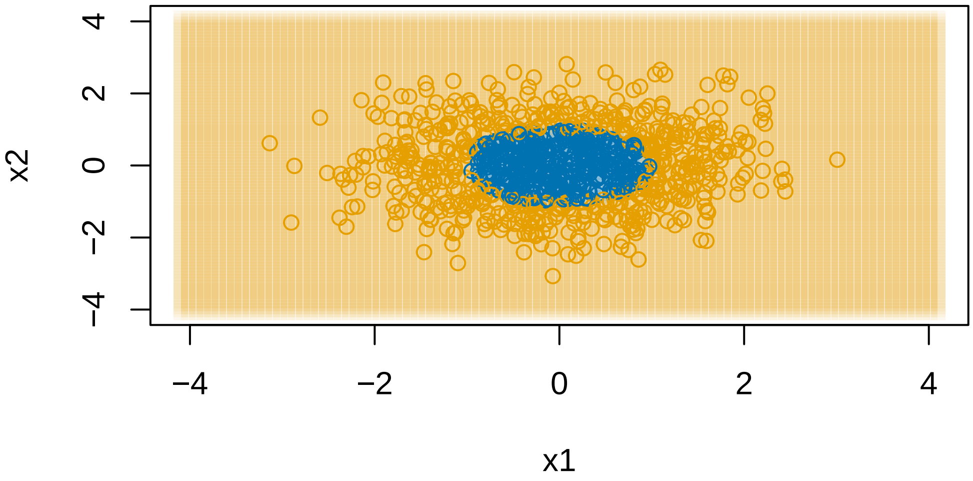

Figure 12: Predictions from a neural network with four hidden nodes

Yes! With four hidden nodes, our neural network has enough "brain power" to capture the concept of an elliptical boundary separating the blue and orange points.

Let's try this out on a more complex shape.

```
planar_dataset <- function(){
  set.seed(set.seed(20240402))
  m <- 1000
  N <- m/2
  D <- 2
  X <- matrix(0, nrow = m, ncol = D)
  Y <- matrix(0, nrow = m, ncol = 1)
  a <- 4

  for(j in 0:1){
    ix <- seq((N*j)+1, N*(j+1))
    t <- seq(j*3.12,(j+1)*3.12,length.out = N) + rnorm(N, sd = 0.2)
    r <- a*sin(4*t) + rnorm(N, sd = 0.2)
    X[ix,1] <- r*sin(t)
    X[ix,2] <- r*cos(t)
    Y[ix,] <- j
  }

  d <- as.data.frame(cbind(X, Y))
  names(d) <- c('X1','X2','Y')
  d
}

df <- planar_dataset()
X <- t(as.matrix(df[,1:2]))
y <- t(df[,3])
plot(X[1,], X[2,], col=col1and2[y[1,] + 1],
     xlab="x1", ylab="x2")
```

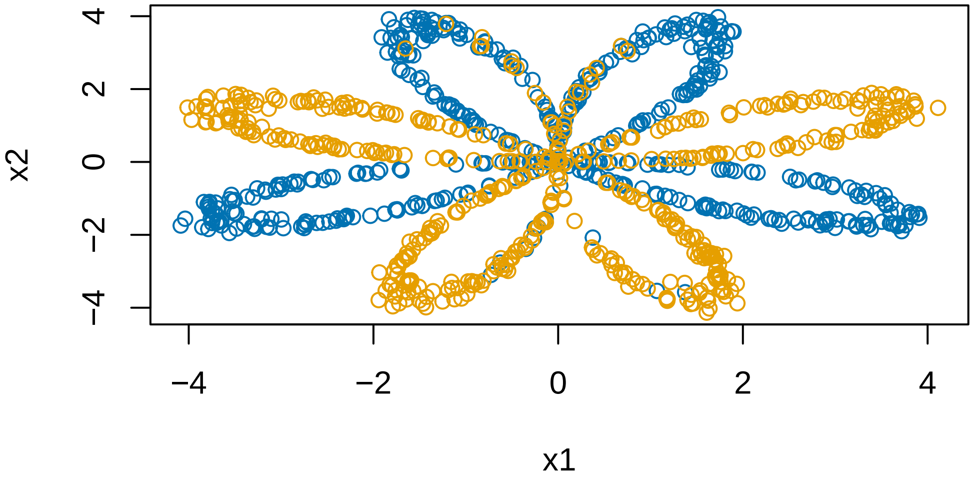

Figure 13: Simulated data with a complex classification boundary

This one has a more complex decision boundary, but let's see if four hidden nodes are sufficient.

```
train_model <- trainModel(X, y,
                          hidden_neurons = 4,
                          num_iteration = 50000, lr = 0.1)
```

```
Iteration 10000  | Cost:  0.2372615
Iteration 20000  | Cost:  0.2138269
Iteration 30000  | Cost:  0.2048732
Iteration 40000  | Cost:  0.1997423
Iteration 50000  | Cost:  0.1962864
```

```
plot(train_model$cost_hist, type="l")
```

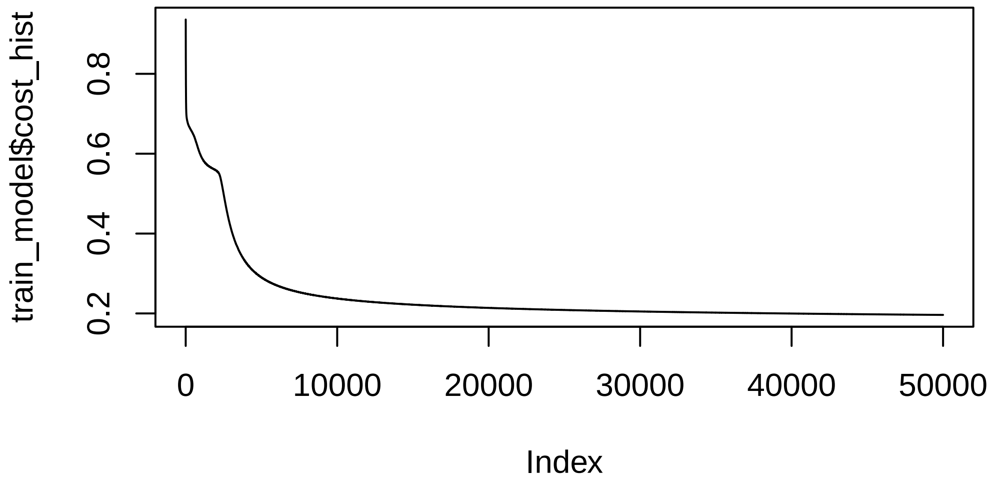

Figure 14: Negative Bernoulli log likelihood by gradient descent iteration, neural network with 4 hidden nodes, “flower shaped” data

```
y_pred <- makePrediction(X, y, train_model)

dAll <- expand.grid(x1=seq(-4.1,4.1,length.out=100),
                    x2=seq(-4.1,4.1,length.out=100))
dAll$y <- 0
dAll$y_pred <- makePrediction(t(as.matrix(dAll[,1:2])),
                              matrix(dAll$y,nrow=1),
                              train_model) |> as.numeric()
plot(dAll$x1, dAll$x2, col=col1and2transparent[(dAll$y_pred>0.5) + 1],
     xlab="x1", ylab="x2", pch=15)
points(X[1,], X[2,], col=col1and2[(y[1,]>0.5) + 1])
```

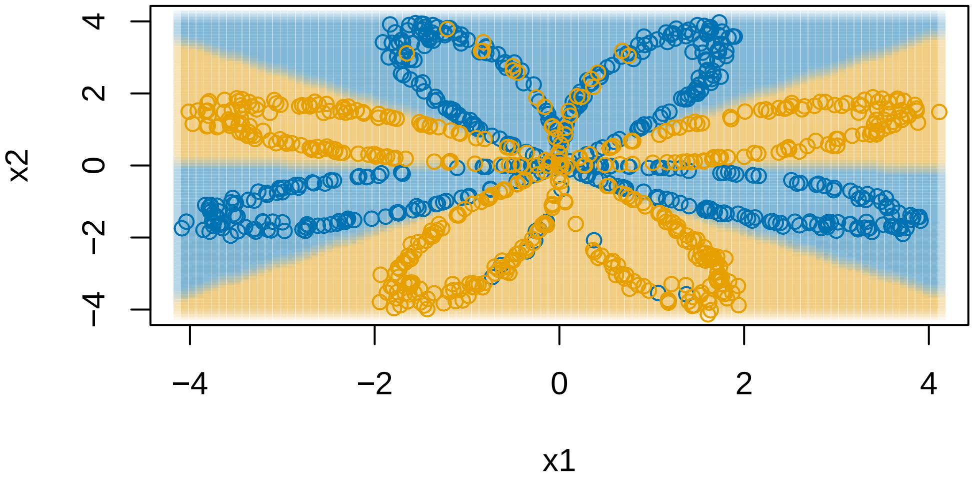

Figure 15: Predictions from a neural network with four hidden nodes

Yes! Four hidden nodes are sufficient to capture this pattern too.

7 R's `neuralnet` package

Now that we have explored fitting a neural network "by hand," in this section we will cover using existing software to fit neural networks.

First, we will walk through using the `neuralnet` package. There is also a `nnet` package, but it allows only one hidden layer. The `neuralnet` package simply allows for more flexibility. After that, we will experiment with Keras, a professional grade neural network system regularly used in scientific analyses.

Start by loading the neuralnet package and revisiting our circular decision boundary dataset.

```
library(neuralnet)

# our circular decision boundary
plot(d$x1, d$x2, col=col1and2[d$y+1],
     xlab="x1", ylab="x2")
```

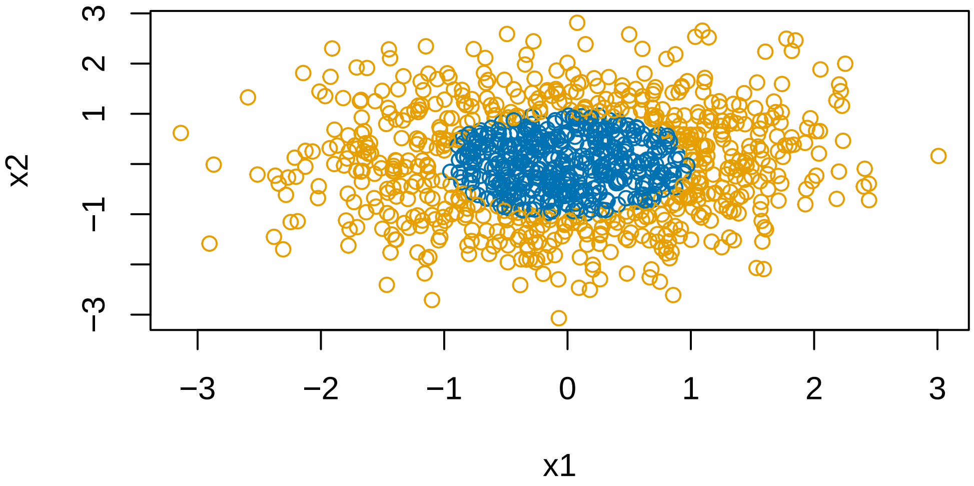

Figure 16: Simulated circular classification boundary

Now we will fit a neural net with a single hidden layer with four nodes using squared error loss and the sigmoid activation function.

```
nn1 <- neuralnet(y ~ x1+x2,
                 data=d,
                 hidden=4,
                 linear.output = FALSE, # apply sigmoid to output
                 stepmax = 1000000,
                 err.fct="sse",         # squared error
                 act.fct="logistic",    # sigmoid
                 lifesign="minimal")    # how much detail to print
```

```
hidden: 4    thresh: 0.01    rep: 1/1    steps:   74792 error: 0.41125  time: 39.35 secs
```

The plot function will draw the network graph for us with the coefficients on all of the edges.

```
plot(nn1,
     show.weights = TRUE,
     information = FALSE,
     col.entry.synapse = col2,
     col.out.synapse = col2,
     col.hidden = col1,
     col.hidden.synapse = "black",
     fill = col1,
     rep="best") # "best" request plot of single best, rather than all
```

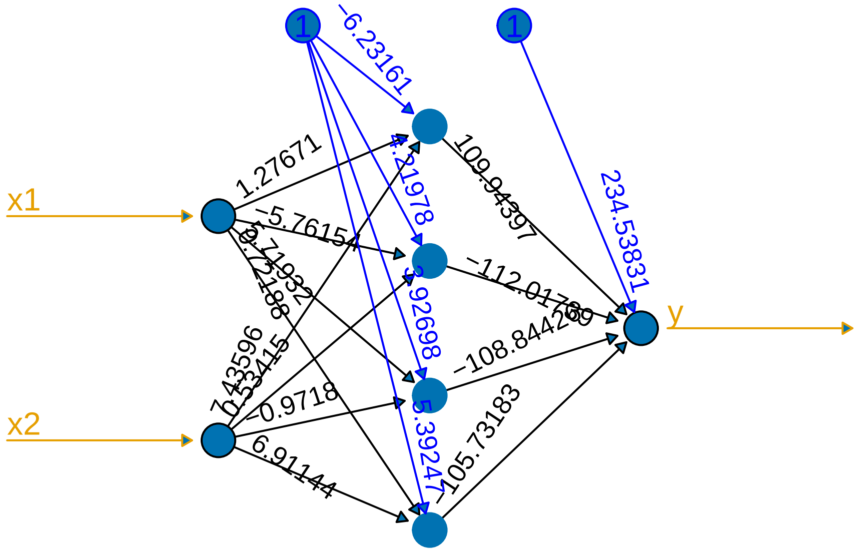

Figure 17: Neural network estimated with `neuralnet()`

Create a plot to show the decision boundary.

```
dAll <- expand.grid(x1=seq(-4.1,4.1,length.out=100),
                    x2=seq(-4.1,4.1,length.out=100))
dAll$y <- 0
dAll$y_pred <- predict(nn1, newdata = dAll)

plot(dAll$x1, dAll$x2, col=col1and2transparent[dAll$y_pred + 1],
     xlab="x1", ylab="x2", pch=15)
points(d$x1, d$x2, col=col1and2[(d$y>0.5) + 1])
```

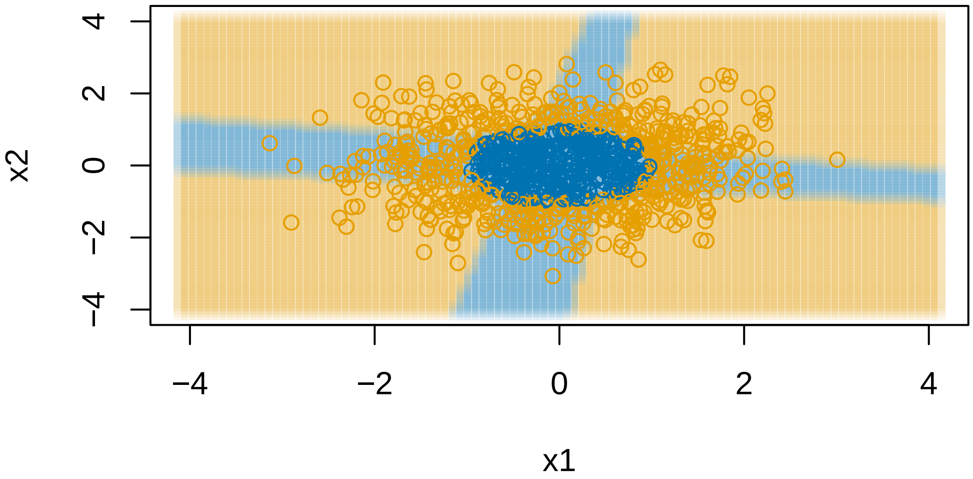

Figure 18: Predictions from a neural network from the `neuralnet` package with four hidden nodes

8 Tensorflow and Keras

Tensorflow is a Google product for efficient machine learning. It allows for computation with GPUs, but current GPU functionality on Windows is broken and removed. Keras is a set of Python tools for communicating what neural network structure you want. It then translates that structure into a neural network model that can be fit with Tensorflow (or with PyTorch, Facebook AI group's Python tools for machine learning). I encourage you to read the documentation and explore demos available for all of these tools.

A lot of the scientific community using deep learning works in Python. For that reason, you will find a lot of Python resources for developing neural networks. Since most social science research is conducted in R, I focused on R for this course. We will still use the R Keras library, which hooks into Python, which hooks into Keras, which hooks into Tensorflow. With all of your variations in computers, operating systems, and settings there is a lot of opportunity for some settings, versions, and options to not be compatible. I will have limited ability to troubleshoot why Keras, Tensorflow, or Python is giving you errors. In this case you will have to learn how to learn to figure these things out.

For the most serious deep learning analysis, that work is done directly in Tensorflow. So, if you really want to learn this area well, start with Keras and then start digging into working with Tensorflow (or PyTorch).

Tensorflow and Keras are usually behind a version or two of Python. Python 3.13 is the current version, but Keras/Tensorflow support up to Python 3.12. This constantly changes. I will be using Python 3.11, because I know it works.

8.1 Tensorflow Playground

Visit the Tensorflow Playground. This let's you freely experiment with datasets with different shapes for their classification boundaries, change the number of hidden layers and the number of nodes in each layer, and see the effect on the network's ability to learn the decision boundary.

Challenge: Using only X_1 and X_2 as inputs, can you alter the number of the hidden layers and nodes that will successful learn the spiral pattern?

8.2 Installing Tensorflow and Keras

First, we will do a one-time installation.

```
install.packages("keras3")
# March 2024, Keras works with 3.11, even though current is 3.14
reticulate::install_python(version = '3.11')
keras3::install_keras(backend = "tensorflow")
```

You will need to restart R one last time after this installation.

Now we can get busy with Keras by loading the library. If all is installed correctly, then this line will run with no errors.

```
library(keras3)
```

8.3 MNIST postal digits data

We are going to experiment with the NIST postal digits data (MNIST database). Why? Because it seems to be a rite of passage for anyone working on neural networks. Everyone has to run the MNIST problem at some point. It is a set of 60,000 28x28 grayscale images handwritten digits. There is also a test set of 10,000 images. The Keras library comes with the MNIST database. We can load the dataset and print out one of the images.

```
numData <- dataset_mnist()

xTrain <- numData$train$x
yTrain <- numData$train$y

xTest <- numData$test$x
yTest <- numData$test$y

i <- 3
img <- t(apply(xTrain[i,,], 2, rev))
image(1:28, 1:28, img,
      col = gray((255:0) / 255),
      axes = FALSE,
      xlab="",ylab="",
      main=yTrain[i])
```

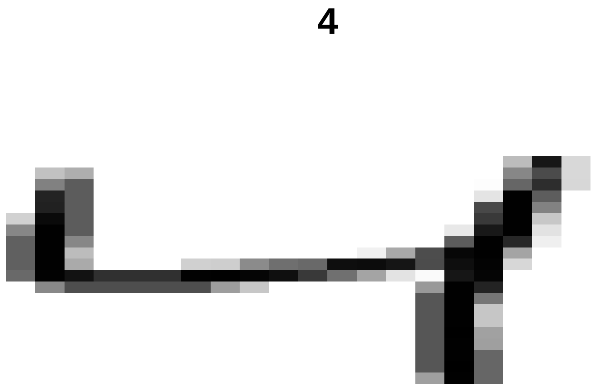

Figure 19: An example number 4 from the MNIST data

That gives us a rough image of a handwritten number 4. Let's take a look at a number of other handwritten digits to get an idea of what these images look like.

```
par(mfrow=c(5,5), mai=0.02+c(0,0,0.5,0))
for(i in 1:25+25)
{
  img <- t(apply(xTrain[i,,], 2, rev))
  image(1:28, 1:28, img,
        col = gray((255:0) / 255),
        xaxt = 'n', yaxt = 'n',
        xlab="",ylab="",
        mar=c(0,0,4,0)+0.01,
        main=yTrain[i])
}
```

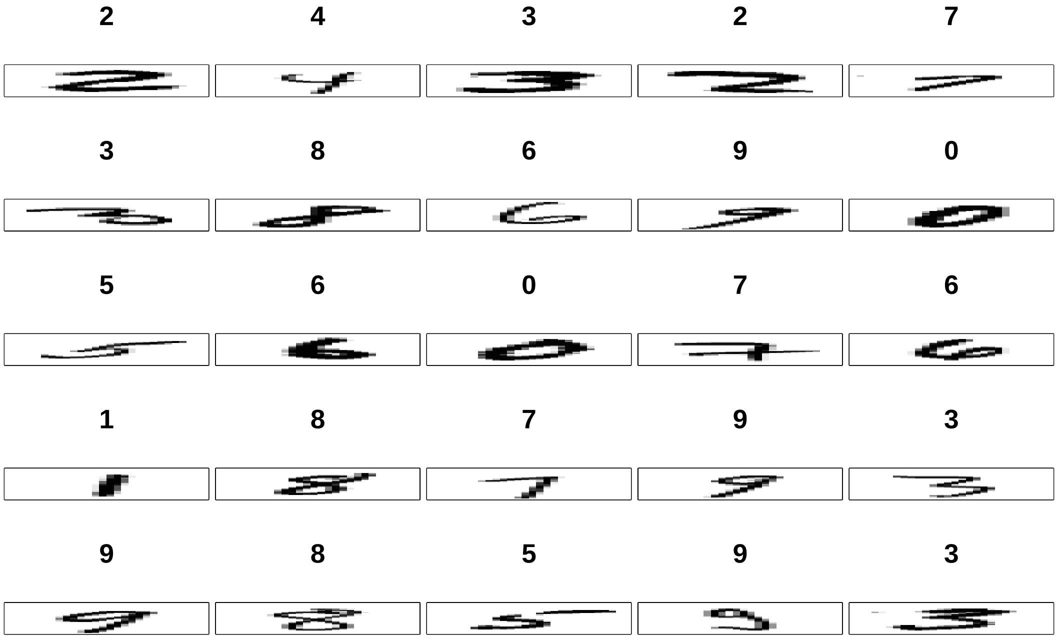

Figure 20: 25 examples from the MNIST data

We need to do a little restructuring of the dataset. Much like we did with the emoji data, we are going to stretch these data out wide, but we need to pay attention to how R stores array data.

I am going to make a little array with three “images.” The first image will have the numbers 1 to 4, the second 5 to 8, and the third 9 to 12.

```
a <- array(NA, dim=c(3,2,2))
a[1,,] <- matrix(1:4, ncol=2,byrow=TRUE)
a[2,,] <- matrix(5:8, ncol=2,byrow=TRUE)
a[3,,] <- matrix(9:12,ncol=2,byrow=TRUE)
a
```

```
, , 1

     [,1] [,2]
[1,]    1    3
[2,]    5    7
[3,]    9   11

, , 2

     [,1] [,2]
[1,]    2    4
[2,]    6    8
[3,]   10   12
```

When we print out a three dimensional array in R, it prints it out so that the last index changes the "fastest." It first shows `a[,,1]` and then `a[,,2]`. To work with Keras, when we stretch the array out wide we need to put the array in "C" format in which the last index changes the fastest.

```
array_reshape(a, c(3, 2*2), order="C") # C = last index changes fastest
```

```
     [,1] [,2] [,3] [,4]
[1,]    1    2    3    4
[2,]    5    6    7    8
[3,]    9   10   11   12
```

Now we can apply this reformatting to our MNISt data.

```
dim(xTrain)
```

```
[1] 60000    28    28
```

```
xTrain <- array_reshape(xTrain, c(60000, 28*28)) / 255
xTest  <- array_reshape(xTest,  c(10000, 28*28)) / 255
```

We also need to convert our outcome values to be 0/1 indicators. So, rather than the outcome being "4," we are going to create a vector with 10 numbers, all which are 0 except for the fourth one, which we will set to 1.

```
# converts outcome to 0/1 coding
yTrain <- to_categorical(yTrain)
yTest <- to_categorical(yTest)
```

Now we will set up the neural network, describing our input data, the number of nodes in the hidden layer (512), setting the activation function (ReLU), and insist that the output predictions sum to one (softmax) so that we have a probability for each number of each image.

```
keras1 <- keras_model_sequential(input_shape = 28*28) |>
  # 512 hidden nodes
  layer_dense(units = 512, activation = "relu") |>
  # randomly set 20% of nodes to 0, supposedly reduces overfitting
  layer_dropout(0.2) |>
  # softmax makes sure outputs sum to 1
  layer_dense(units = 10, activation = "softmax")
```

We can ask Keras to describe the model we are about to fit.

```
summary(keras1)
```

```
Model: "sequential"

 Layer (type)                    Output Shape                    Param #

 dense (Dense)                   (None, 512)                     401,920

 dropout (Dropout)               (None, 512)                           0

 dense_1 (Dense)                 (None, 10)                        5,130

 Total params: 407,050 (1.55 MB)
 Trainable params: 407,050 (1.55 MB)
 Non-trainable params: 0 (0.00 B)
```

Then we tell Keras about how we want it to optimize the model and evaluate its performance. rmsprop is Root Mean Square Propagation. It is a popular variant of backpropagation that regularly adjusts the learning rate parameter. Categorical cross-entropy is the multinomial generalization of the negative Bernoulli log likelihood. We can also have Keras track other metrics, like here I ask it to track accuracy.

```
compile(keras1,
        optimizer = "rmsprop",
        loss = "categorical_crossentropy",
        metrics = "accuracy")
```

Ready to go! It's now time to tell Keras to actually fit the model. The `batch_size` is the number of training observations used in one forward and backward pass through the neural network. This is called "mini-batch gradient descent." `epochs` is the number of full passes through the dataset. Since there are 60,000 images, and we are using 80% of them for training and 20% for validation, and we have a batch size of 128, you will see each epoch require 375 (= 60000*0.8/128) updates within each epoch.

```
fitHx <- fit(keras1,
  xTrain,
  yTrain,
  epochs = 20,
  batch_size = 128,
  validation_split = 0.2
)
```

```
Epoch 1/20
375/375 - 6s - 17ms/step - accuracy: 0.9104 - loss: 0.3094 - val_accuracy: 0.9517 - val_loss
Epoch 2/20
375/375 - 4s - 12ms/step - accuracy: 0.9600 - loss: 0.1351 - val_accuracy: 0.9693 - val_loss
Epoch 3/20
375/375 - 4s - 11ms/step - accuracy: 0.9726 - loss: 0.0929 - val_accuracy: 0.9698 - val_loss
Epoch 4/20
375/375 - 4s - 12ms/step - accuracy: 0.9786 - loss: 0.0711 - val_accuracy: 0.9751 - val_loss
Epoch 5/20
375/375 - 5s - 12ms/step - accuracy: 0.9828 - loss: 0.0577 - val_accuracy: 0.9754 - val_loss
Epoch 6/20
375/375 - 4s - 11ms/step - accuracy: 0.9857 - loss: 0.0465 - val_accuracy: 0.9789 - val_loss
Epoch 7/20
375/375 - 4s - 12ms/step - accuracy: 0.9881 - loss: 0.0388 - val_accuracy: 0.9794 - val_loss
Epoch 8/20
375/375 - 8s - 21ms/step - accuracy: 0.9900 - loss: 0.0329 - val_accuracy: 0.9787 - val_loss
```

```
Epoch 9/20
375/375 - 5s - 12ms/step - accuracy: 0.9914 - loss: 0.0274 - val_accuracy: 0.9790 - val_loss
Epoch 10/20
375/375 - 5s - 14ms/step - accuracy: 0.9931 - loss: 0.0235 - val_accuracy: 0.9812 - val_loss
Epoch 11/20
375/375 - 5s - 13ms/step - accuracy: 0.9939 - loss: 0.0196 - val_accuracy: 0.9812 - val_loss
Epoch 12/20
375/375 - 4s - 12ms/step - accuracy: 0.9942 - loss: 0.0188 - val_accuracy: 0.9809 - val_loss
Epoch 13/20
375/375 - 4s - 12ms/step - accuracy: 0.9953 - loss: 0.0152 - val_accuracy: 0.9810 - val_loss
Epoch 14/20
375/375 - 4s - 11ms/step - accuracy: 0.9960 - loss: 0.0132 - val_accuracy: 0.9801 - val_loss
Epoch 15/20
375/375 - 4s - 11ms/step - accuracy: 0.9965 - loss: 0.0120 - val_accuracy: 0.9809 - val_loss
Epoch 16/20
375/375 - 4s - 10ms/step - accuracy: 0.9967 - loss: 0.0116 - val_accuracy: 0.9808 - val_loss
Epoch 17/20
375/375 - 4s - 10ms/step - accuracy: 0.9976 - loss: 0.0087 - val_accuracy: 0.9821 - val_loss
Epoch 18/20
375/375 - 4s - 10ms/step - accuracy: 0.9974 - loss: 0.0089 - val_accuracy: 0.9822 - val_loss
Epoch 19/20
375/375 - 4s - 10ms/step - accuracy: 0.9976 - loss: 0.0077 - val_accuracy: 0.9813 - val_loss
Epoch 20/20
375/375 - 4s - 10ms/step - accuracy: 0.9979 - loss: 0.0070 - val_accuracy: 0.9822 - val_loss
```

We can plot the "learning curves," tracing out how much better the neural network became after each epoch.

```r
library(ggplot2)
fitHx |>
  plot() +
  geom_point(size = 3) +
  geom_line(linetype = "dashed")
```

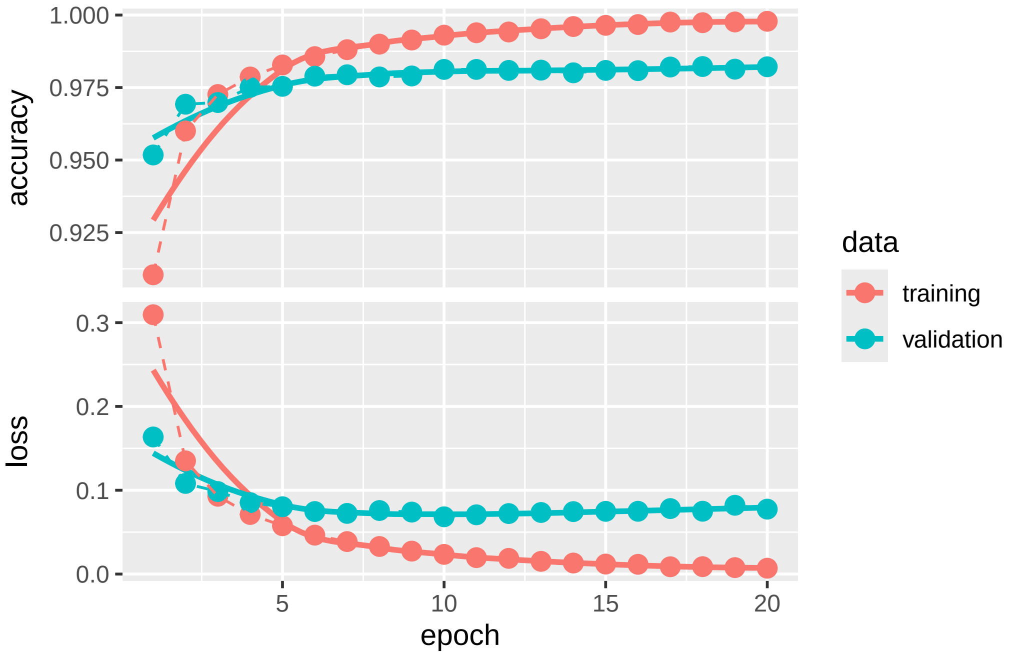

Figure 21: Loss function and accuracy on training and validation MNIST data

We can predict on `xTest`, the held out test dataset and create a confusion matrix to see how our neural network performed.

```
yPredVal <- predict(keras1, xTest) |>
  op_argmax(axis=2, zero_indexed = TRUE) |>
  as.numeric()
```

```
313/313 - 1s - 3ms/step
```

```
yTestVal <- apply(yTest, 1, which.max)-1
table(yPredVal, yTestVal)
```

```
        yTestVal
yPredVal    0    1    2    3    4    5    6    7    8    9
       0  972    0    5    0    1    2    3    2    1    2
       1    0 1126    1    0    0    0    2    4    0    3
       2    1    3 1013    1    3    0    0    8    5    0
       3    0    0    1  990    0    5    1    2    3    3
       4    0    0    3    0  960    2    3    1    2    6
```

```
   5    1    0    0    6    0  874    3    0    1    3
   6    4    2    1    0    3    3  945    0    1    1
   7    1    1    4    4    2    0    0 1003    3    3
   8    1    3    4    3    2    2    1    3  956    0
   9    0    0    0    6   11    4    0    5    2  988
```

For the most part we observe very high counts along the diagonal indicating that the neural network gets a lot of the classifications correctly. We do see a fair number of off diagonal elements as well. For example, note that for 6 images that were actually 9s, the neural network classified them as 4s. That is probably the most common mistake a human would make as well.

Let's examine 12 randomly selected digits that we have misclassified.

```r
set.seed(20240408)
iError <- which(yPredVal != yTestVal) |> sample(size=12)

par(mfrow=c(3,4), mai=0.02+c(0,0,0.5,0))
for(i in iError)
{
  img <- matrix((xTest[i,]), nrow=28)[,28:1]
  image(1:28, 1:28, img,
        col = gray((255:0) / 255),
        xaxt = 'n', yaxt = 'n',
        xlab="",ylab="",
        mar=c(0,0,4,0)+0.1,
        main=paste("Prediction:",yPredVal[i]))
}
```

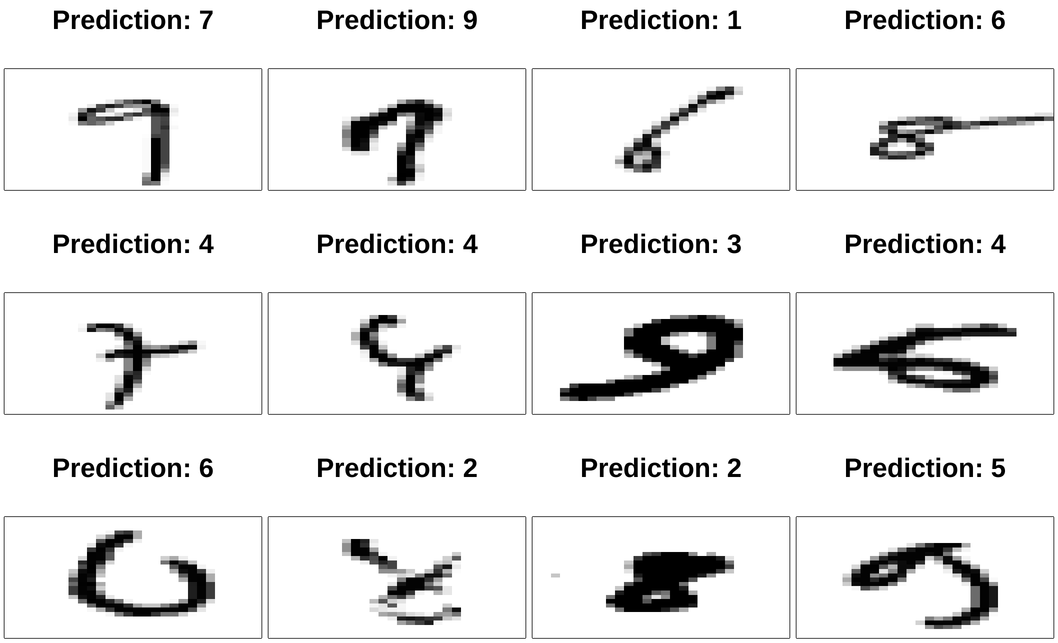

Figure 22: Examples of errors from our neural network. Heading of each image shows the predicted value

`evaluate()` extracts measures of performance on the test dataset.

```
evaluate(keras1, xTest, yTest)
```

```
313/313 - 1s - 4ms/step - accuracy: 0.9827 - loss: 0.0664

$accuracy
[1] 0.9827

$loss
[1] 0.06636865
```

8.4 Convolution layers

Convolutional layers are the main building block for Convolution Neural Networks (CNNs), primarily used in processing data with a grid-like shapes, like images. The utility of convolutional layers comes from their ability to efficiently handle and process spatial information. Unlike layers that treat input data as a flat array, as we did previously, convolutional layers preserve the spatial relationships between pixels or data points by applying filters (or kernels) that scan over the input. This approach enables the network to capture local patterns such as edges, textures, or shapes within the input data. Convolutional layers can significantly reduce the size of networks needed to get good predictive performance, making the training process faster and less prone to overfitting. Convolutional layers learn feature representations, making them especially powerful for tasks involving images, video, and time-series data.

Let $\mathbf{X}$ be an $r \times c$ greyscale image where the element $\mathbf{X}_{ij}$ is between 0 and 1, denoting the greyscale range from black to white. The convolution layer involves a $m \times m$ matrix of parameters, $\mathbf{K}$, that the network will need to estimate from data. Typically, in practice m is small, like 2, 3, or 4.

Assume $m = 2$ so that $\mathbf{X}$ is a 2×2 matrix of the form $\begin{bmatrix} a & b \\ c & d \end{bmatrix}$. The convolution layer will transform every 2×2 section of $\mathbf{X}$ to create a new matrix that will feed into the next layer.

Consider a small greyscale image in the shape of an "X'' that we will conveniently call our $\mathbf{X}$

$$\mathbf{X} = \begin{bmatrix} 0.9 & 0.0 & 0.8 \\ 0.0 & 0.7 & 0.0 \\ 0.8 & 0.0 & 0.9 \end{bmatrix}$$

We run every 2×2 adjacent submatrix of $\mathbf{X}$ this through the convolution layer by multiplying the elements of the submatrix $\mathbf{X}_{(i,j)}$ by $\mathbf{K}$ and adding up the elements. This is equivalent to computing $\mathrm{tr}(\mathbf{K}'\mathbf{X}_{(i,j)})$, where the trace of a matrix is the sum of the diagonal elements.

$$\begin{bmatrix} \mathrm{tr}\left(\begin{bmatrix} a & b \\ c & d \end{bmatrix}' \begin{bmatrix} 0.9 & 0 \\ 0 & 0.7 \end{bmatrix}\right) & \mathrm{tr}\left(\begin{bmatrix} a & b \\ c & d \end{bmatrix}' \begin{bmatrix} 0 & 0.8 \\ 0.7 & 0 \end{bmatrix}\right) \\ \mathrm{tr}\left(\begin{bmatrix} a & b \\ c & d \end{bmatrix}' \begin{bmatrix} 0 & 0.7 \\ 0.8 & 0 \end{bmatrix}\right) & \mathrm{tr}\left(\begin{bmatrix} a & b \\ c & d \end{bmatrix}' \begin{bmatrix} 0.7 & 0 \\ 0 & 0.9 \end{bmatrix}\right) \end{bmatrix} \rightarrow \begin{bmatrix} 0.9a + 0.7d & 0.8b + 0.7c \\ 0.7b + 0.8c & 0.7a + 0.9d \end{bmatrix}$$

Then the components of the result are simply treated as a vector of the form

$$\begin{bmatrix} 0.9a + 0.7d & 0.8b + 0.7c & 0.7b + 0.8c & 0.7a + 0.9d \end{bmatrix}'$$

and sent into the next layer of the neural network, now accepting 4 new inputs rather than the original 9 inputs.

The convolution kernel K can vary by size and shape and "stride," the spacing between each application of the convolution. Also, the input may be a three-dimensional tensor input with the third dimension being a 3-level color channel, the mixture of red, green, and blue, for example. For such cases, we use a 3D kernel for the convolution layer. This transforms both the spatial image and the local color structure to an input vector.

8.5 A convolutional neural network with Keras

Keras easily allows you to add a convolution layer to the neural network. First, we need to reorganize our dataset so that we do not flatten out the images into one long vector of grayscale values, but instead retain their two dimensional structure.

```
xTrain <- numData$train$x
xTest  <- numData$test$x

# 60000 28x28 images with 1 scale color (greyscale)
xTrain <- array_reshape(xTrain, c(60000,28,28,1))
xTest  <- array_reshape(xTest,  c(10000,28,28,1))
```

Next, we need to describe through Keras how to structure the layers of the neural network. We need to indicate the `input_shape` (28x28). We need to indicate the `kernel_size` (3x3). We can have the model consider several kernels so that one might capture lines while another captures a curve of a particular type. Here I set `filters=32` so the neural network will consider 32 different kernels. `padding="same"` adds 0s around the edges of the image so that the image does not shrink due to the image boundary. It lets the kernel straddle the edges of an image.

Note that I have added a second convolution layer after the first convolution layer. In CNNs, stacking multiple convolutional layers is a useful strategy to increase the neural network's ability to identify complex features within the image. The first layer typically learns simple features, such as edges and lines, while subsequent layers combine these to detect more sophisticated patterns. The layered approach expands the "receptive field," allowing the network to perceive larger portions of the input data, but allows the model to capture non-linearity. The approach also uses parameters more efficiently, reducing the risk of overfitting by leveraging spatial hierarchies rather than relying on a vast number of parameters. Deeper convolutional layers improve the neural network's generalization capabilities, making it more adept at recognizing a wide range of features at different levels of complexity.

The maximum 2D pooling layer carves up the inputs from the second convolution layer into a bunch of 2x2 grids and just passes the largest value on to the next layer. This reduces the number of features passed on to the next layer, highlights the most significant features by taking their maximum values, and contributes to the model's efficiency and reduces the risk of overfitting.

`layer_flatten()` turns the 2D inputs into one long vector containing all the features from the previous layer. These go into a hidden layer wih 1000 nodes with the ReLU activation function. `layer_dropout(0.5)` randomly sets half of the inputs to 0 during training. This is believed to reduce the risk of overfitting and encourage the optimizer to explore other "versions" to find parameters that perform well. When predicting on future observations, all nodes are activated. The final layer, `layer_dense(10, activation = "softmax")`, is the output layer set to 10 nodes, one for each of the digits we are trying to predict. `softmax` forces the output values to sum to 1 so that they represent a probability distribution over the possible predictions.

```
inputs <- layer_input(shape = c(28, 28, 1))

outputs <- inputs |>
  layer_conv_2d(filters = 32, kernel_size = c(3,3), padding = "same") |>
  layer_activation("relu") |>
  layer_conv_2d(filters = 16, kernel_size = c(2,2),
                dilation_rate = c(1,1), activation = "softplus",
                padding = "same") |>
  layer_max_pooling_2d(pool_size = c(2,2)) |>
  layer_flatten() |>
  layer_dense(1000, activation = "relu") |>
  layer_dropout(0.5) |>
  layer_dense(10, activation = "softmax")

kerasCNN <- keras_model(inputs = inputs, outputs = outputs)

summary(kerasCNN)
```

```
Model: "functional_3"

 Layer (type)                      Output Shape                  Param #

 input_layer_1 (InputLayer)        (None, 28, 28, 1)                   0

 conv2d (Conv2D)                   (None, 28, 28, 32)                320

 activation (Activation)           (None, 28, 28, 32)                  0

 conv2d_1 (Conv2D)                 (None, 28, 28, 16)              2,064

 max_pooling2d (MaxPooling2D)      (None, 14, 14, 16)                  0

 flatten (Flatten)                 (None, 3136)                        0

 dense_2 (Dense)                   (None, 1000)                3,137,000

 dropout_1 (Dropout)               (None, 1000)                        0

 dense_3 (Dense)                   (None, 10)                     10,010
 Total params: 3,149,394 (12.01 MB)
 Trainable params: 3,149,394 (12.01 MB)
 Non-trainable params: 0 (0.00 B)
```

As before, we then compile the model to optimize the cross-entropy.

```
compile(kerasCNN,
  loss = "categorical_crossentropy",
  optimizer = "rmsprop",
  metrics = "accuracy")
```

Then we train the model as before and observe its predictive performance over iterations.

```
fitHx <- fit(kerasCNN,
             xTrain,
             yTrain,
             epochs = 20,
             batch_size = 128,
             validation_split = 0.2)
```

```
Epoch 1/20
375/375 - 26s - 69ms/step - accuracy: 0.8876 - loss: 1.9728 - val_accuracy: 0.9744 - val_los
Epoch 2/20
375/375 - 16s - 42ms/step - accuracy: 0.9694 - loss: 0.1076 - val_accuracy: 0.9827 - val_los
Epoch 3/20
375/375 - 11s - 31ms/step - accuracy: 0.9799 - loss: 0.0691 - val_accuracy: 0.9852 - val_los
Epoch 4/20
375/375 - 13s - 34ms/step - accuracy: 0.9844 - loss: 0.0568 - val_accuracy: 0.9848 - val_los
Epoch 5/20
375/375 - 13s - 35ms/step - accuracy: 0.9865 - loss: 0.0474 - val_accuracy: 0.9852 - val_los
Epoch 6/20
375/375 - 14s - 36ms/step - accuracy: 0.9889 - loss: 0.0406 - val_accuracy: 0.9862 - val_los
Epoch 7/20
375/375 - 13s - 36ms/step - accuracy: 0.9905 - loss: 0.0352 - val_accuracy: 0.9861 - val_los
Epoch 8/20
375/375 - 14s - 37ms/step - accuracy: 0.9913 - loss: 0.0341 - val_accuracy: 0.9864 - val_los
Epoch 9/20
375/375 - 14s - 37ms/step - accuracy: 0.9920 - loss: 0.0302 - val_accuracy: 0.9862 - val_los
Epoch 10/20
375/375 - 14s - 39ms/step - accuracy: 0.9925 - loss: 0.0292 - val_accuracy: 0.9876 - val_los
```

```
Epoch 11/20
375/375 - 14s - 38ms/step - accuracy: 0.9923 - loss: 0.0322 - val_accuracy: 0.9875 - val_loss
Epoch 12/20
375/375 - 14s - 38ms/step - accuracy: 0.9932 - loss: 0.0275 - val_accuracy: 0.9861 - val_loss
Epoch 13/20
375/375 - 15s - 41ms/step - accuracy: 0.9932 - loss: 0.0279 - val_accuracy: 0.9876 - val_loss
Epoch 14/20
375/375 - 17s - 44ms/step - accuracy: 0.9940 - loss: 0.0260 - val_accuracy: 0.9867 - val_loss
Epoch 15/20
375/375 - 17s - 46ms/step - accuracy: 0.9935 - loss: 0.0273 - val_accuracy: 0.9865 - val_loss
Epoch 16/20
375/375 - 19s - 52ms/step - accuracy: 0.9950 - loss: 0.0208 - val_accuracy: 0.9873 - val_loss
Epoch 17/20
375/375 - 1478s - 4s/step - accuracy: 0.9947 - loss: 0.0198 - val_accuracy: 0.9872 - val_loss
Epoch 18/20
375/375 - 13s - 35ms/step - accuracy: 0.9951 - loss: 0.0223 - val_accuracy: 0.9872 - val_loss
Epoch 19/20
375/375 - 20s - 52ms/step - accuracy: 0.9946 - loss: 0.0236 - val_accuracy: 0.9862 - val_loss
Epoch 20/20
375/375 - 12s - 32ms/step - accuracy: 0.9955 - loss: 0.0215 - val_accuracy: 0.9843 - val_loss
```

```r
plot(fitHx)
```

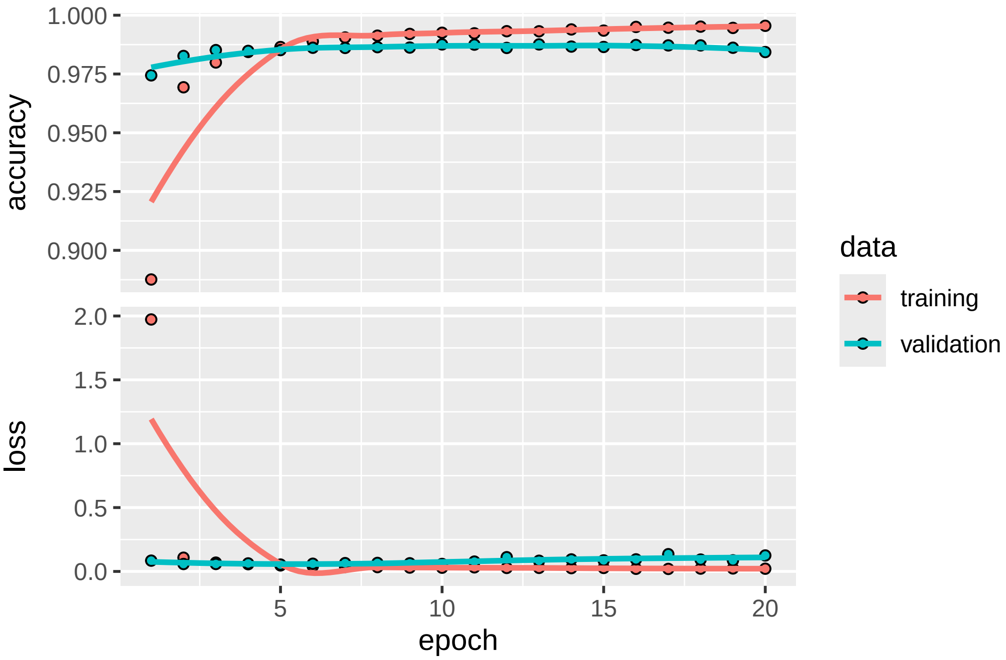

Figure 23: Loss function and accuracy on training and validation MNIST data with a convolution layer

Lastly, we predict on the test dataset.

```r
# predict on test dataset
yPredVal <- predict(kerasCNN, xTest) |>
  op_argmax(axis=2, zero_indexed = TRUE) |>
  as.numeric()
```

```
313/313 - 1s - 5ms/step
```

```r
yTestVal <- yTest |>
  op_argmax(axis=2, zero_indexed = TRUE) |>
  as.numeric()
table(yPredVal, yTestVal)
```

```
        yTestVal
yPredVal    0    1    2    3    4    5    6    7    8    9
       0  974    0    3    0    1    1    3    0    5    1
       1    1 1128    2    1    2    0    2    1    1    4
```

```
    2    0    2 1017    2    2    0    0    5    3    0
    3    0    2    0  994    1    4    0    1    3    0
    4    0    0    0    0  939    0    0    0    0    1
    5    1    1    0    7    0  883    6    1    4    6
    6    2    2    0    0    7    4  947    0    3    0
    7    1    0    9    3    5    0    0 1015    5    9
    8    1    0    1    0    3    0    0    1  945    1
    9    0    0    0    3   22    0    0    4    5  987
```

Again, we can check to see what kind of errors the model makes. Frankly, they would be quite difficult for a human to distinguish too.

```r
iError <- which(yPredVal != yTestVal) |> sample(size=12)

par(mfrow=c(3,4), mai=0.02+c(0,0,0.5,0))
for(i in iError)
{
  img <- matrix(xTest[i,,,], ncol=28, byrow=TRUE)[,28:1]
  image(1:28, 1:28, img,
        col = gray((255:0) / 255),
        xaxt = 'n', yaxt = 'n',
        xlab="",ylab="",
        mar=c(0,0,4,0)+0.1,
        main=paste("Prediction:",yPredVal[i]))
}
```

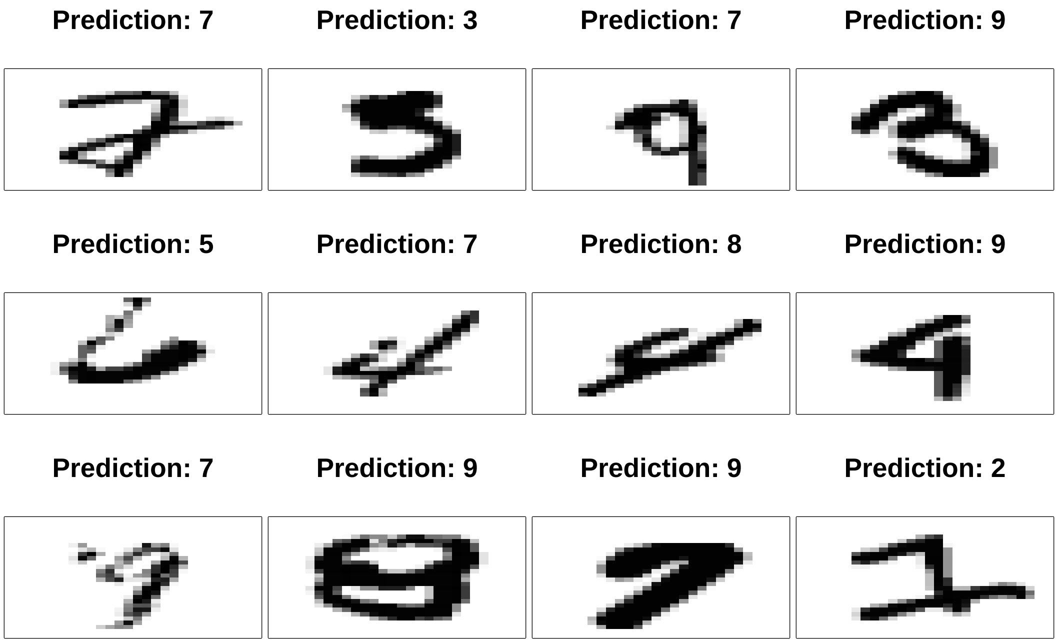

Figure 24: Examples of errors from our neural network with a convolution layer. Heading of each image shows the predicted

Let's check the overall performance.

```
evaluate(kerasCNN, xTest, yTest)
```

```
313/313 - 1s - 5ms/step - accuracy: 0.9829 - loss: 0.1427
```

```
$accuracy
[1] 0.9829

$loss
[1] 0.142709
```

We have squeezed out just a tiny bit more predictive performance with this model. However, this model had 3149394 parameters while our simpler one without convolution layers had 407050 parameters. We had to spend a lot of additional parameters to get a tiny gain in predictive performance. And when we check what we are getting wrong, we are at the stage where humans would have a hard time getting them correct. For comparison, today's large language models have over 100,000,000,000 (100 billion) parameters. GPT4 weights in at 1,800,000,000,000 (1.8 trillion) parameters.

Developing a neural network is more an art than a science. There are no general theories that tell us what the right way of combining layers are, how best to use convolutional layers, or how best to optimize parameters to minimize generalization error. In practice, we try a range of layers, parameters, and alterations to our dataset to see which one might get us some better performance. There are a variety of rules-of-thumb that have been adopted and I have used a lot of those here, but there is no reason to think that these are the best choices, and they certainly are not the best for all applications. With this I hope you have a start on how to use Keras to assemble a neural network for whatever problem you want to try to solve.

Deisenroth, Marc Peter, A. Aldo Faisal, and Cheng Soon Ong. 2020. *Mathematics for Machine Learning*. Cambridge University Press. https://mml-book.com/.

Hastie, T., R. Tibshirani, and J. H. Friedman. 2001. *The Elements of Statistical Learning: Data Mining, Inference, and Prediction*. Springer-Verlag.

James, Gareth, Daniela Witten, Trevor Hastie, and Robert Tibshirani. 2021. *An Introduction to Statistical Learning: With Applications in r*. 2nd ed. Springer Texts in Statistics. New York, NY: Springer. https://www.statlearning.com/.

Rosenblatt, F. 1962. *Principles of Neurodynamics: Perceptrons and the Theory of Brain Mechanisms*. Cornell Aeronautical Laboratory. Report No. VG-1196-g-8. Spartan Books. https://books.google.com/books?id=7FhRAAAAMAAJ.