L10 Neural networks

Greg Ridgeway

2025-04-06

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.

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

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 we learn how badly we did 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.

Figure 1: A linear model as a network

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

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

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

$$\hat{\beta} \leftarrow \hat{\beta} + \lambda \mathbf{X}'(\mathbf{y} - \hat{\mathbf{y}})$$

where in the last line the λ absorbed the 2.

3 Neural networks with hidden layers

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 **??** two continuous inputs, passes them through a hidden layer of four nodes, which passes their output to a final output node.

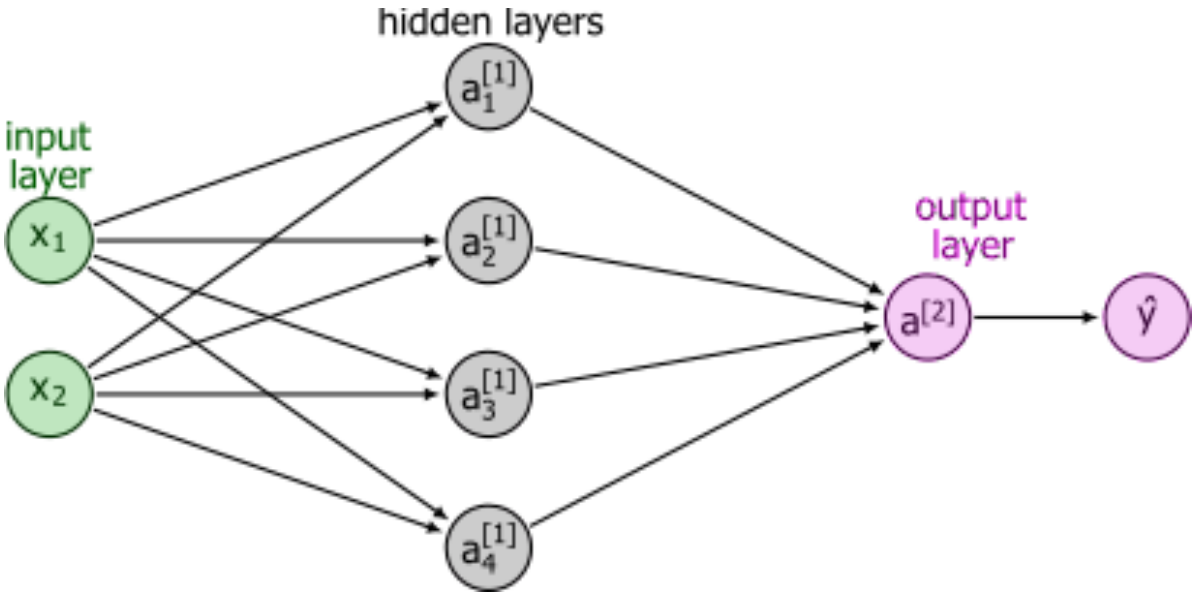

Figure 2: A neural network with a single hidden layer with 4 hidden nodes and a single output

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 **??** shows their shape.

```
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)
```

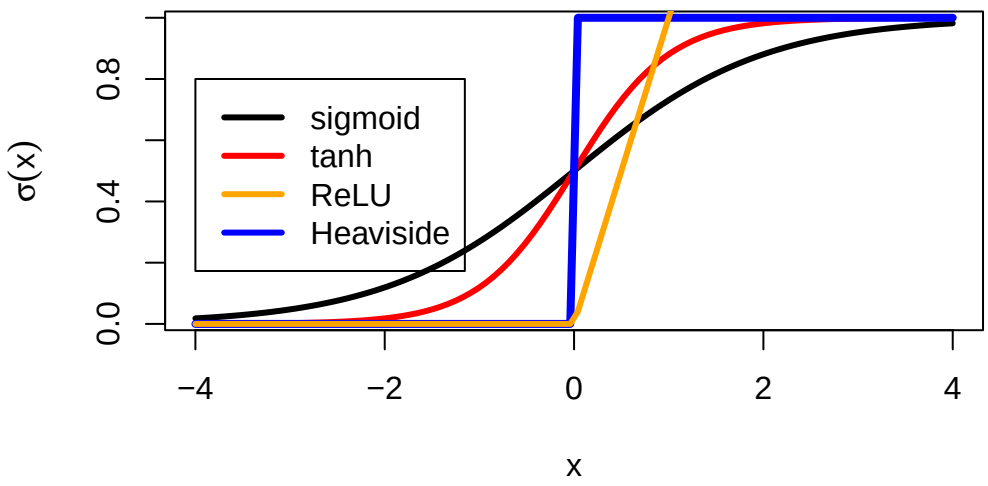

Figure 3: Four common activation functions

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 **??**, 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.

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

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

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