Deep Feed-Forward Neural Networks

Deep feedforward networks, also called **feedforward neural networks**, or **multilayer perceptrons** (MLPs). The goal of a feedforward network is to approximate some function f^*. For example, for a classifier, $y = f^*(\mathbf{x})$ maps an input $\mathbf{x}$ to a category y. A feedforward network defines a mapping $y = f(\mathbf{x}; \boldsymbol{\theta})$ and learns the value of the parameters $\boldsymbol{\theta}$ that result in the best function approximation[1].

Feedforward neural networks are called **networks** because they are typically represented by composing together many different functions. For example, we might have three functions $f^{(1)}, f^{(2)}$, and $f^{(3)}$ connected in a chain, to form $f(\mathbf{x}) = f^{(3)}(f^{(2)}(f^{(1)}(\mathbf{x})))$. In this case, $f^{(1)}$ is called the **first layer** of the network, $f^{(2)}$ is called the **second layer**, and so on. The overall length of the chain gives the **depth** of the model. The name "deep learning" came from this terminology. The final layer of a feedforward network is called the **output** layer. During neural network training, we drive $f(\mathbf{x})$ to match $f^*(\mathbf{x})$. The training data provides us with noisy, approximate examples of $f^*(\mathbf{x})$ evaluated at different training points. Each example x is accompanied by a label $y \approx f^*(\mathbf{x})$. The training examples specify directly what the output layer must do at each point $\mathbf{x}$; it must produce a value that is close to y. The behavior of the other layers is not directly specified by the training data. The learning algorithm must decide how to use those layers to produce the desired output, but the training data do not say what each individual layer should do. Instead, the learning algorithm must decide how to use these layers to best implement an approximation of f^*. Because the training data does not show the desired output for each of these layers, they are called **hidden layers**.

1 Example: Learning XOR

To make the idea of a feedforward network more concrete, we begin with an example of a fully functioning feedforward network on a very simple task: learning the XOR function.

The XOR function ("exclusive or") is an operation on two binary values, x_1 and x_2. When exactly one of these binary values is equal to 1, the XOR function returns 1. Otherwise, it returns 0. The XOR function provides the target function $y = f^*(\mathbf{x})$ that we want to learn. Our model provides a function $y = f(\mathbf{x}; \boldsymbol{\theta})$, and our learning algorithm will adapt the parameters $\boldsymbol{\theta}$ to make f as similar as possible to f^*.

[1]From here on, the bold variables represent vectorial variables.

In this example, We want our network to perform correctly on the four points $\mathbb{X} = \{[0,0],[0,1],[1,0],[1,1]\}$. We will train the network on all four of these points. The only challenge is to fit the training set. We can treat this problem as a regression problem and use a mean squared error (MSE) loss function. Evaluated on our whole training set, the MSE loss function is:

$$J(\boldsymbol{\theta}) = \frac{1}{4}\sum_{\mathbf{x}\in\mathbb{X}}(f^*(\mathbf{x}) - f(\mathbf{x};\boldsymbol{\theta}))^2$$

Now we must choose the form of our model, $f(\mathbf{x};\boldsymbol{\theta})$. Suppose that we choose a linear model, with $\boldsymbol{\theta}$ consisting of $\mathbf{w}$ and b. Our model is defined to be

$$f(\mathbf{x};\mathbf{w},b) = \mathbf{x}^\top\mathbf{w} + b.$$

Similar to regression idea, we can minimize $J(\boldsymbol{\theta})$ in closed form with respect to $\mathbf{w}$ and b using the normal equations. After solving the normal equations, we obtain $\mathbf{w} = \mathbf{0}$ and $b = 0.5$. The linear model simply outputs 0.5 everywhere. Figure 1 shows how a linear model is not able to represent the XOR function. One way to solve this problem is to use a model that learns a different feature space in which a linear model is able to represent the solution.

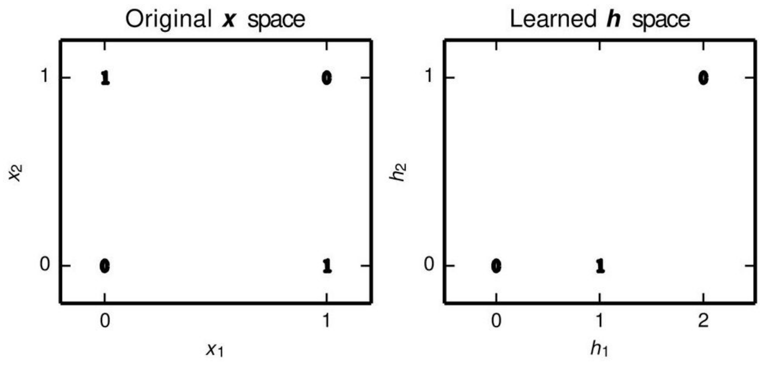

Figure 1: Solving the XOR problem by learning a representation. The bold numbers printed on the plot indicate the value that the learned function must output at each point. (*Left*) A linear model applied directly to the original input cannot implement the XOR function. (*Right*) In the transformed space represented by the features extracted by a neural network, a linear model can now solve the problem. In our example solution, the two points that must have output 1 have been collapsed into a single point in feature space. In other words, the nonlinear features have mapped both $\mathbf{x} = [1,0]$ and $\mathbf{x} = [0,1]$ to a single point in feature space, $\mathbf{h} = [1,0]$. The linear model can now describe the function as increasing in $\mathbf{h}_1$ and decreasing in $\mathbf{h}_2$. In this example, the motivation for learning the feature space is only to make the model capacity greater so that it can fit the training set.

To solve it, we use a simple feedforward network with one hidden layer containing two hidden units as figure 2. This feedforward network has a vector of hidden units $\mathbf{h}$ that are computed by a function $f^{(1)}(\mathbf{x}; \mathbf{W}, \mathbf{c})$. The values of these hidden units are then used as the input for a second layer. The second layer is the output layer of the network. The output layer is still just a linear regression model, but now it is applied to $\mathbf{h}$ rather than to $\mathbf{x}$. The network now contains two functions chained together, $\mathbf{h} = f^{(1)}(\mathbf{x}; \mathbf{W}, \mathbf{c})$ and $y = f^{(2)}(\mathbf{h}; \mathbf{w}, \mathbf{b})$, with the complete model being $f(\mathbf{x}; \mathbf{W}, \mathbf{c}, \mathbf{w}, b) = f^{(2)}(f^{(1)}(\mathbf{x}))$.

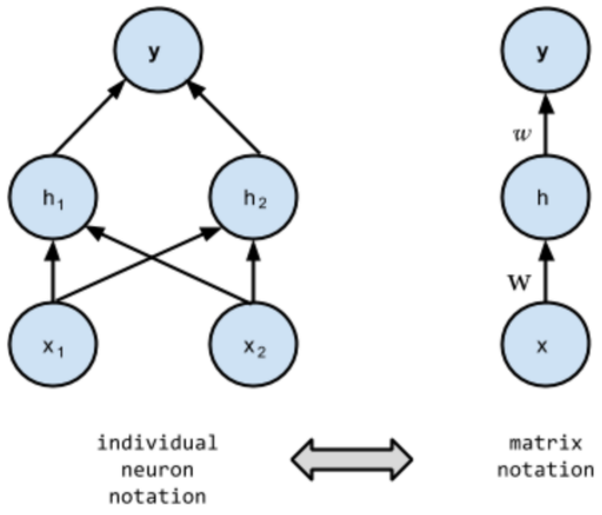

Figure 2: The feedforward network used to solve the XOR example.

What function should $f^{(1)}$ compute? Linear models have served us well so far, and it may be tempting to make f (1) linear as well. Unfortunately, if $f^{(1)}$ were linear, then the feedforward network as a whole would remain a linear function of its input. Ignoring the intercept terms for the moment, suppose $f^{(1)}(\mathbf{x}) = W^\top \mathbf{x}$ and $f^{(2)}(\mathbf{h}) = \mathbf{h}^\top \mathbf{w}$. Then $f(\mathbf{x}) = \mathbf{w}^\top \mathbf{W}^\top \mathbf{x}$. We could represent this function as $f(\mathbf{x}) = \mathbf{x}^\top(\mathbf{W}\mathbf{w})$.

Clearly, we must use a nonlinear function to describe the features. Most neural networks use an affine transformation controlled by learned parameters, followed by a fixed nonlinear function called an **activation function**. We define $\mathbf{h} = g(\mathbf{W}^\top \mathbf{x} + \mathbf{c})$, where $\mathbf{W}$ provides the weights of a linear transformation and $\mathbf{c}$ the biases. We describe an affine transformation from a vector $\mathbf{x}$ to a vector $\mathbf{h}$, so an entire vector of bias parameters is needed. The activation function g is typically chosen to be a function that is applied element-wise, with $h_i = g(\mathbf{x}^\top \mathbf{W}_{:,i} + c_i)$. In modern neural networks, the default recommendation is to use the **rectified linear unit** (ReLU), defined by the activation function $g(z) = \max\{0, z\}$, depicted in figure 3.

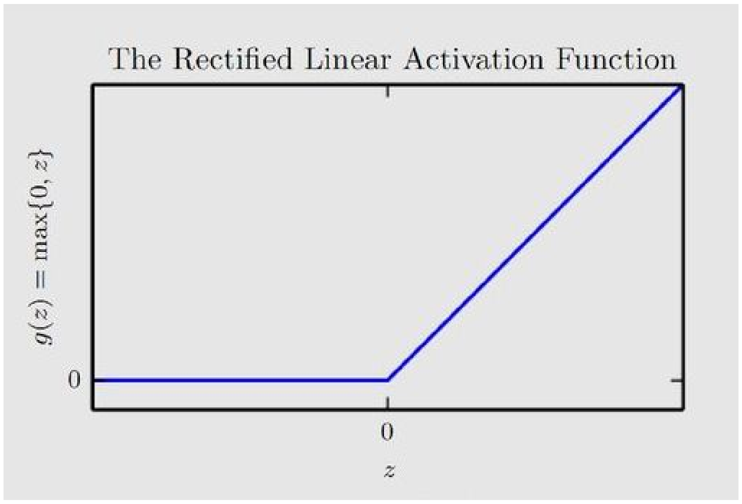

Figure 3: The rectified linear activation function. Applying this function to the output of a linear transformation yields a nonlinear transformation. The function remains very close to linear, however, in the sense that is a piecewise linear function with two linear pieces.

We can now specify our complete network as

$$f(\mathbf{x}; \mathbf{W}, \mathbf{c}, \mathbf{w}, b) = \mathbf{w}^\top \max\{0, \mathbf{W}^\top \mathbf{x} + \mathbf{c}\} + b. \tag{1}$$

We can now specify a solution to the XOR problem by defining
$\mathbf{W} = \begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix}$, $\mathbf{c} = \begin{bmatrix} 0 \\ -1 \end{bmatrix}$, $\mathbf{w} = \begin{bmatrix} 1 \\ -2 \end{bmatrix}$ and $b = 0$.

We will check how the defined weights satisfy our neural network presented in equation 1 for the XOR. Let $\mathbf{X}$ be the design matrix containing all four points in the binary input space, with one example per row:

$$\mathbf{X} = \begin{bmatrix} 0 & 0 \\ 0 & 1 \\ 1 & 0 \\ 1 & 1 \end{bmatrix}$$

The first step in the neural network is to multiply the input matrix by the first layer's weight matrix:

$$\mathbf{XW} = \begin{bmatrix} 0 & 0 \\ 1 & 1 \\ 1 & 1 \\ 2 & 2 \end{bmatrix}$$

Next, we add the bias vector $\mathbf{c}$, to obtain

$$\mathbf{XW} + \mathbf{c} = \begin{bmatrix} 0 & -1 \\ 1 & 0 \\ 1 & 0 \\ 2 & 1 \end{bmatrix}$$

To finish computing the value of $\mathbf{h}$ for each example, we apply the rectified linear transformation:

$$\max\{0, (\mathbf{XW} + \mathbf{c})\} = \begin{bmatrix} 0 & 0 \\ 1 & 0 \\ 1 & 0 \\ 2 & 1 \end{bmatrix}$$

This transformation has changed the relationship between the examples. They no longer lie on a single line. As shown in figure 1, they now lie in a space where a linear model can solve the problem. We finish with multiplying by the weight vector $\mathbf{w}$ and finally adding $b = 0$:

$$\max\{0, (\mathbf{XW} + \mathbf{c})\}\mathbf{w} + b = \begin{bmatrix} 0 \\ 1 \\ 1 \\ 0 \end{bmatrix}$$

In this example, we simply specified the solution, then showed that it obtained zero error. In a real situation, there might be billions of model parameters and billions of training examples, so one cannot simply guess the solution as we did here.

2 Feed-Forward Neural Networks

Figure 4 demonstrates a simple example of a feed-forward neural network. The bottom layer of the network pulls in the input data. The top layer of neurons (output nodes) computes our final answer. The middle layer(s) of neurons are called the hidden layers, and we let $w_{i,j}^k$ be the weight of the connection between the i-th neuron in the k-th layer with the j-th neuron in the $k+1$st layer. These weights constitute our parameter vector, $\boldsymbol{\theta}$, and just as before, our ability to solve problems with neural networks depends on finding the optimal values to plug into $\boldsymbol{\theta}$. We note that in this example, connections only traverse from a lower layer to a higher layer. There are no connections between neurons in the same layer, and there are no connections that transmit data from a higher layer to a lower layer.

2.1 Output Units

Linear neurons are easy to compute with, but they run into serious limitations. In fact, it can be shown that any feed-forward neural network consisting of only

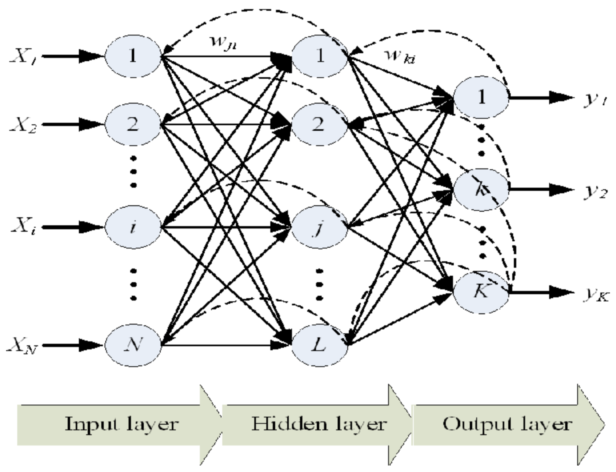

Figure 4: A simple example of a feed-forward neural network with three layers (input, one hidden, and output) and three neurons per layer.

linear neurons can be expressed as a network with no hidden layers. This is problematic because, in order to learn complex relationships, we need to use neurons that employ some sort of nonlinearity.

Sigmoid, Tanh and ReLU Units : There are three major types of outputs that are used in practice that introduce nonlinearities in their computations. The first of these is the **sigmoid neuron**, which uses the function:

$$f(z) = \frac{1}{1 + e^{-Z}}$$

Intuitively, this means that when the logit is very small, the output of a logistic neuron is very close to 0. When the logit is very large, the output of the logistic neuron is close to 1. In-between these two extremes, the neuron assumes an S-shape, as shown in Figure 5.

Tanh units use a similar kind of S-shaped nonlinearity, but instead of ranging from 0 to 1, the output of tanh neurons range from -1 to 1. As one would expect, they use f $z = \tanh(z)$. The resulting relationship between the output y and the logit z is described by Figure 6. When S-shaped nonlinearities are used, the tanh neuron is often preferred over the sigmoid neuron because it is zero-centered.

A different kind of nonlinearity is used by the **restricted linear unit (ReLU)** unit. It uses the function $f(z) = \max(0, z)$, resulting in a characteristic hockey-stick-shaped response, as shown in Figure 3.

Softmax Outyput Layers Oftentimes, we want our output vector to be a probability distribution over a set of mutually exclusive labels. For example,

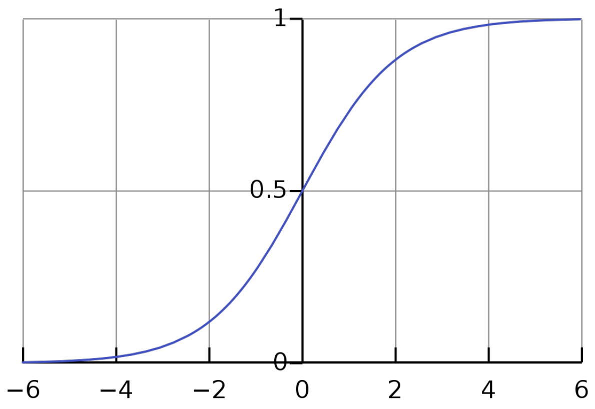

Figure 5: The output of a sigmoid unit as z varies.

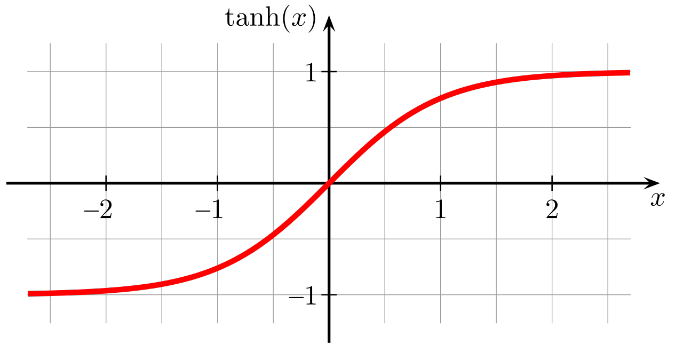

Figure 6: The output of a Tanh unit as z varies.

let's say we want to build a neural network to recognize handwritten digits from the MNIST dataset[2]. Each label (0 through 9) is mutually exclusive, but it's unlikely that we will be able to recognize digits with 100% confidence. Using a probability distribution gives us a better idea of how confident we are in our predictions. As a result, the desired output vector is of the form below, where $\sum_{i=0}^{9} p_i = 1$:

$$[p_0 \; p_1 \; p_2 \; \cdots \; p_9]$$

This is achieved by using a special output layer called a **softmax layer**. Unlike in other kinds of layers, the output of a neuron in a softmax layer depends on the outputs of all the other neurons in its layer. This is because we require the sum of all the outputs to be equal to 1. Letting z_i be the logit of the i-th

[2]Is a dataset containing various images of 10 digits from 0 to 9: `https://en.wikipedia.org/wiki/MNIST_database`

softmax neuron, we can achieve this normalization by setting its output to:

$$y_i = \frac{e^{z_i}}{\sum_j e^{z_j}}$$

A strong prediction would have a single entry in the vector close to 1, while the remaining entries were close to 0. A weak prediction would have multiple possible labels that are more or less equally likely.