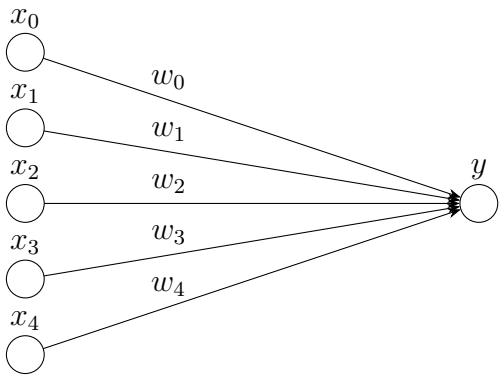
Chapter 1

Introduction

1.1 Neural Networks

Neural networks represent a mathematical tool in machine learning that is useful for performing function approximation; they can be thought of as a generalization of linear regression. The power of a neural network stems from three main properties that we will go over: nonliterary, differentiability, and hidden layers. These key properties allow neural networks to be able to improve its approximation of a set of values via "training."

To kick things off we will start with a simple example of a neural network. In the simplest form, a neural network consists of a **vector input** $\vec{x} \in \mathbb{R}^n$, a set of **weights** $w_i \in \mathbb{R}$, and a final **output** $y \in \mathbb{R}$. Below is a graphical representation of a such network.



In such a graphical representation, the set of x_i nodes are called the **input layer** (or simply the first layer) and the y node is called the **output layer**. In the above diagram, the output layer consists of one node, but as we will see it can also consist of multiple nodes.

The output is obtained as a function of the input and weights as below.

$$y = \sum_i w_i x_i$$

From a statistics perspective, this is actually just a **linear model**. In statistics, one "trains" such a linear model via linear regression on some dataset. If you have taken a statistics course, you might remember that this strategy works on simple examples (e.g., a suspiciously-already-linear weather

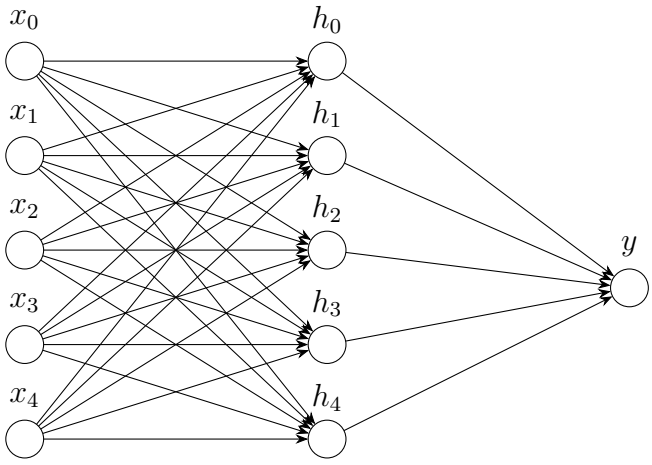
dataset in a Pearson textbook), but linear models do not generalize and often fail to capture complex behavior.

As we will see, neural networks are different from linear models since they add properties of hidden layers and nonlinearity.

1.2 Hidden Layers

Neural networks extend our previous notion of a linear model via **hidden layers**, which can be defined as one or more layers between the input and output layer. Below, we have a neural network which has one hidden layer. The hidden layer can have a variable number of nodes, but in our example below we have five.

In this example, each input node x_i connects to each node h_j in the hidden layer via a weight w_{ij}. These weights are illustrated by the arrows, although we are choosing to suppress the w_{ij} notation in the diagram below to not over complicate the figure.



The calculation of a hidden layer is simply

$$h_i = \sum_j w_{ij} x_i$$

Intuitively, this means that each input value x_i makes a weighted contribution of w_{ij} to the value h_j. Something to observe at this point is that we can summarize the entire hidden layer calculation as a matrix equation:

$$\vec{h} = \begin{bmatrix} h_1 \\ h_2 \\ h_3 \\ h_4 \\ h_5 \end{bmatrix} = \begin{bmatrix} \sum_i w_{i1} x_i \\ \sum_i w_{i2} x_i \\ \sum_i w_{i3} x_i \\ \sum_i w_{i4} x_i \\ \sum_i w_{i5} x_i \end{bmatrix} = \begin{bmatrix} w_{11} & w_{12} & w_{13} & w_{14} & w_{15} \\ w_{21} & w_{22} & w_{23} & w_{24} & w_{25} \\ w_{31} & w_{32} & w_{33} & w_{34} & w_{35} \\ w_{41} & w_{42} & w_{43} & w_{44} & w_{45} \\ w_{51} & w_{52} & w_{53} & w_{54} & w_{55} \end{bmatrix} \cdot \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \end{bmatrix} = W\vec{x} \tag{1.1}$$

This suggest the concept of a **weight matrix** W, which is the key to calculating the hidden layer $\vec{h}$ from the input $\vec{x}$.

As we can see, between each layer we have a matrix of weights. We can denote U as the matrix of the weights between the output layer and the hidden layer, so that $y = U\vec{h}$.

Often times with neural networks it is useful to introduce a **bias** in each layer. This is because later, we will be interested in seeing how each individual weight in each layer contributes to the final output. If the value of a hidden node is simply zero, it will be harder to judge how the weights (which contributed to that hidden node) are contributing to the final output. Hence, adding a bias helps prevent the value of a hidden node from becoming just zero.

If we introduce bias in our above network, the calculation becomes

$$h_i = \sum_j w_{ij} x_i + b_i$$

where b_i denotes the bias assigned to the hidden layer node h_i.

If we assemble the the bias values b_i into a vector $\vec{b}$, our calculation of $\vec{h}$ becomes

$$\vec{h} = W\vec{x} + \vec{b}$$

and if the we let c (in this case, it's just a scalar instead of a vector) denote the bias in the final layer we have

$$y = U\vec{h} + c$$

However, note that we're not really doing much mathematically by adding a hidden layer. Observe that we can rewrite y as

$$y = U\vec{h} + c = U(W\vec{x} + \vec{b}) + c = W'\vec{x} + c'$$

where $W' = UW$ and $c' = U\vec{b} + c$. This reduces our above network, with three layers, to a boring network, with two layers (similar to the one we started with), just with a different weight matrix W' and bias value c'. The reason is because our network is linear, which means no matter how many layers we add it will always reduce to the same boring network we started with. Thus in order to get something interesting with hidden layers we need to add some nonlinearity.

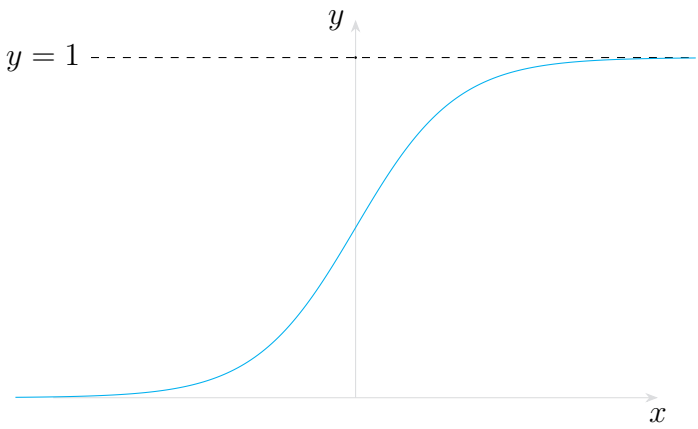
1.3 Nonlinearity

Neural networks achieve our desired property of nonlinearity via usage of **activation fuctions**. A few common examples of such functions are the sigmoid (also known as logistic), tanh or RELU functions, and the choice of an activation function depends on what one is trying to achieve with a neural network.

The sigmoid function is given by

$$\sigma(x) = \frac{1}{1 + e^{-x}}$$

The graph of the sigmoid function is given below.



Using our previous network, we can add nonlinearity by defining the computation of a hidden unit to be

$$h_j = \sigma\left(\sum_i w_{ij}x_i + b_i\right)$$

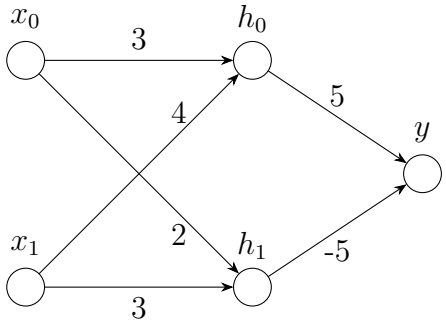
where σ is the activation function of choice.

1.4 Backpropagation: A first stab

At this point, as we have discussed the basic properties of a neural network, we will introduce a concrete example of a neural network and attempt to perform train it to approximate the XOR function, which performs the following mapping on two bit inputs:

$$\begin{bmatrix}1\\0\end{bmatrix} \to 1 \qquad \begin{bmatrix}0\\0\end{bmatrix} \to 0 \qquad \begin{bmatrix}0\\1\end{bmatrix} \to 1 \qquad \begin{bmatrix}1\\1\end{bmatrix} \to 0 \tag{1.2}$$

Below is our proposed network.



Define the biases $\vec{b}_1$ and $\vec{b}_2$ for the first layer and the second layer, respectively, to be

$$\vec{b}_1 = \begin{bmatrix}-2\\-4\end{bmatrix} \qquad \vec{b}_2 = -2$$

This leads us to define weight matrices

$$W = \begin{bmatrix}3 & 4\\2 & 3\end{bmatrix} \qquad U = \begin{bmatrix}5 & -5\end{bmatrix}$$

allowing us to write $\vec{h} = \sigma(W\vec{x})$ and $y = \sigma(U\vec{h})$, or more explicitly

$$h_0 = \sigma(3x_0 + 4x_1 - 2) \tag{1.3}$$
$$h_1 = \sigma(2x_0 + 3x_1 - 4) \tag{1.4}$$
$$y = \sigma(5h_1 - 5h_2 + 2) \tag{1.5}$$

which we can use to calculate the network. Below is a table of how this neural network currently computes the XOR values.

(x_0, x_1)	h_0	h_1	y	target
$(1, 0)$	$\sigma(1) = 0.731$	$\sigma(-2) = 0.119$	$\sigma(1.060) = 0.743$	1
$(0, 0)$	$\sigma(-2) = 0.119$	$\sigma(-4) = 0.018$	$\sigma(-1.495) = 0.183$	0
$(0, 1)$	$\sigma(2) = 0.881$	$\sigma(-1) = 0.269$	$\sigma(1.060) = 0.743$	1
$(1, 1)$	$\sigma(5) = 0.993$	$\sigma(1) = 0.731$	$\sigma(-0.690) = 0.334$	0

Based on the above table, we can see that so far it's not performing that well, but after all it is a first stab. This now begs the question: What does it mean for a model to perform well, and how do we know when it is improving? The answer to this is to introduce a **cost function** which can give a measure of error. There are many possible choices for a cost function, but for simplicity we will use the **least squares** cost function. If we have a dataset of target values t_i, and our model currently approximates this data with a set of values y_i, then the measured loss is

$$L = \frac{1}{2}\sum_i (t_i - y_i)^2$$

The concept of a cost function now leads to the strategy of back propagation: simply change the model's weights in such a way as to minimize the cost function.

In the case of our model, we need to find out what direction, and how much, we should "push" each value of our existing matices U and W, so as to minimize the cost function. In terms of calculus, this means we are interested in the quantities

$$\frac{dL}{du_i} \qquad \frac{dL}{dw_{ij}}$$

Once we obtain these quantities, we can update our weights after reviewing one training example t_i via:

$$u_i' = u_i - \frac{dL}{du_i} \tag{1.6}$$
$$w_{ij}' = w_{ij} - \frac{dL}{dw_{ij}} \tag{1.7}$$

1.5 Backpropagation: More generally