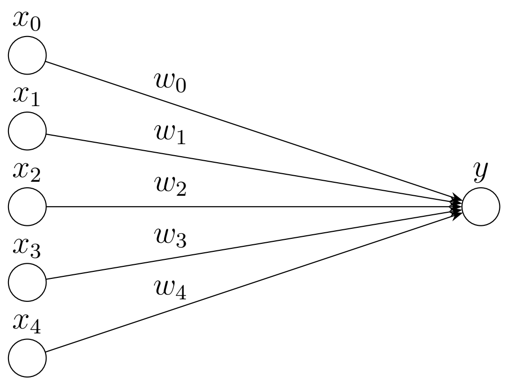
Chapter 1

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.

$$y = \sum_i w_i x_i$$

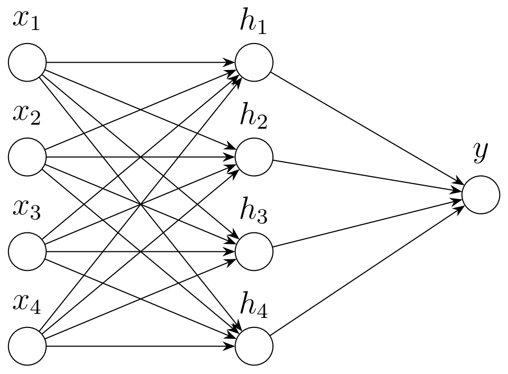
From a statistics perspective, this is actually just a **linear model**. In statistics, one "trains" such a linear model via linear regression on some dataset. If you have taken a statistics course, you might remember that this strategy works on simple examples (e.g., a suspiciously-already-linear weather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

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

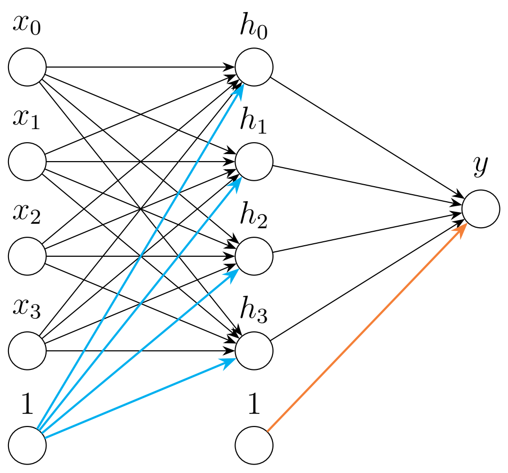
Intuitively, this means that each input value x_i makes a weighted contribution of w_{ij} to the value h_j. Something to observe at this point is that we can summarize the entire hidden layer calculation as a matrix equation:

$$\vec{h} = \begin{bmatrix} h_1 \\ h_2 \\ h_3 \\ h_4 \end{bmatrix} = \begin{bmatrix} \sum_i w_{1i} x_i \\ \sum_i w_{2i} x_i \\ \sum_i w_{3i} x_i \\ \sum_i w_{4i} x_i \end{bmatrix} = \begin{bmatrix} w_{11} & w_{12} & w_{13} & w_{14} \\ w_{21} & w_{22} & w_{23} & w_{24} \\ w_{31} & w_{32} & w_{33} & w_{34} \\ w_{41} & w_{42} & w_{43} & w_{44} \end{bmatrix} \cdot \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix} = W\vec{x} \tag{1.1}$$

This suggest the concept of a **weight matrix** W, which is the key to calculating the hidden layer $\vec{h}$ from the input $\vec{x}$. For a neural network that has multiple hidden layers, there are multiple corresponding weight matrices. In fact, a neural network with N layers will have $(N-1)$-many weight matrices.

For our current example, we will let U denote 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 necessary to introduce a **bias** in each layer. A bias is an extra node, assigned a value of 1, that we can add to a layer which will be used to contribute to the calculation of the next layer. Below we illustrate the network we'd obtain by adding bias to our previous network.



With this neural network architecture, we have the following weight matrices:

$$W = \begin{bmatrix} w_{11} & w_{12} & w_{13} & w_{14} & w_{15} \\ w_{21} & w_{22} & w_{23} & w_{24} & w_{25} \\ w_{31} & w_{32} & w_{33} & w_{34} & w_{35} \\ w_{41} & w_{42} & w_{43} & w_{44} & w_{45} \end{bmatrix} \qquad U = \begin{bmatrix} u_{11} & u_{12} & u_{13} & u_{14} & w_{15} \end{bmatrix}$$

Thus, adding a bias to a layer is equivalent to adding a new column to the weight matrix. Our input $[x_1, x_2, x_3, x_4]$ is still in $\mathbb{R}^4$, but we now instead feed the neural network a value of $[x_1, x_2, x_3, x_4, 1]^T \in \mathbb{R}^5$.

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

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

where $W' = UW$. 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'. 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. As we know, linear patterns cannot adequately capture complex patterns. 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**. An activation function is a function f that is called on the summation of a given node in a network, allowing us to modify the calculation for a node h_i as below.

$$h_i = f\left(\sum_j w_{ij}x_j\right)$$

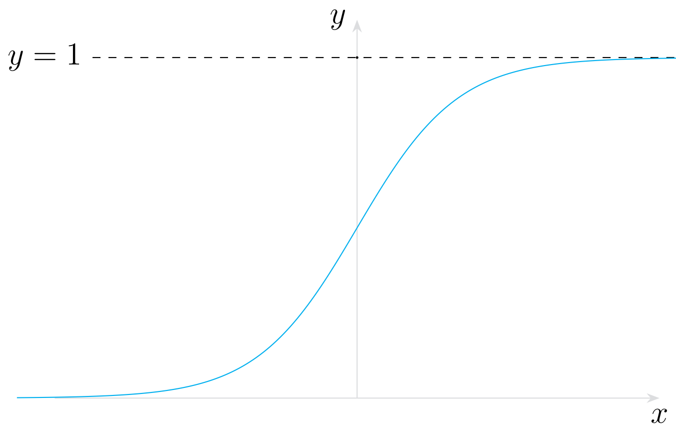
We can introduce nonlinearity into our system if we design f to be nonlinear.

A few common examples of such functions are the sigmoid (also known as logistic), tanh or RELU functions. The machine learning community has gone through several iterations of what is considered to be best practice for an activation function. In the 1990s, the sigmoid function was used very widely. In the later 90s and early 2000s, it was discovered that the tanh function lead to be better training performance. Later, it was the discovered that the ReLU function lead to even better training performance. As a result, most modern neural networks will use this for the activation function.

Let's introduce the activation functions we just discussed. Below, we have the sigmoid function

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

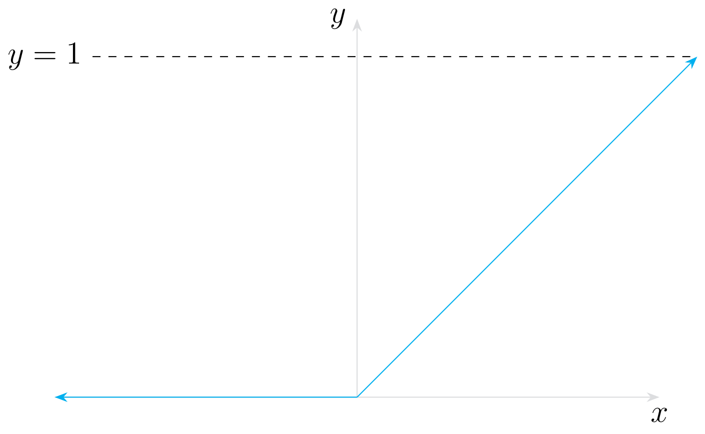
and the graph of the sigmoid function is given below.



Finally, ReLU itself is given by

$$r(x) = \begin{cases} x & \text{if } x > 0 \\ 0 & \text{otherwise} \end{cases}$$

Below we have ReLU, which we will use in our examples as it generally results in better training performance.



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

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

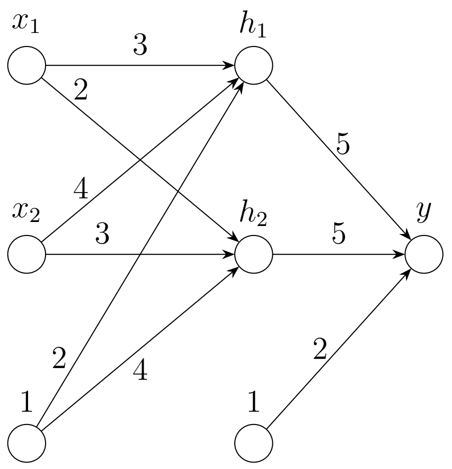
where σ is the activation function of choice.

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 train it to approximate the XOR function. The XOR function performs the following mapping on these 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. We'll use ReLU, denoted as $r(x)$, for the activation function on our nodes.



Note that we have a bias inbetween each layer. The first one has weights 2 and -4, and the second one has weight -2. Thus for this network, the weight matrices are defined to be

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

allowing us to write $\vec{h} = r(W\vec{x})$ and $y = r(U\vec{h})$, or more explicitly, for a given input (x_0, x_1)

$$h_1 = r(3x_1 + 4x_2 + 2) \tag{1.3}$$
$$h_2 = r(2x_1 + 3x_2 + 4) \tag{1.4}$$
$$y = r(5h_1 + 5h_2 + 2) \tag{1.5}$$

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

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

In our case, since the output of our function is in $\mathbb{R}$, we have $L = \frac{1}{2}(t - y)^2$.

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 via

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

First, let us calculate how the loss is affected by the weights in the final layer (i.e. the entries of the matrix U). Since $y = r(\sum_k u_k h_k)$, we have that

$$\frac{dL}{du_i} = \frac{dL}{dy}\frac{dy}{du_i}$$

Observe that

$$\frac{dL}{dy} = -(t-y)$$

and

$$\frac{dy}{du_i} = r'\left(\sum_k u_k h_k\right) \cdot \frac{d}{du_i}\left(\sum_k u_k h_k\right) = r'\left(\sum_k u_k h_k\right) \cdot h_i$$

If we write $s_y = \sum_k u_k h_k$ (i.e. the summation before applying the activation r which calculates the value of y) then we can write

$$\frac{dL}{du_i} = -(t-y)r'(s_y)h_i.$$

Next, let us calculate how the loss is affected by weights in the first layer (i.e. the entries of the matrix W). Once again we have

$$\frac{dL}{dw_{ij}} = \frac{dL}{dy}\frac{dy}{dw_{ij}} = \frac{dL}{dy}\frac{dy}{dh_i}\frac{dh_i}{dw_{ij}}$$

We already know $\frac{dL}{dy}$. Thus we calculate

$$\frac{dy}{dh_i} = r'\left(\sum_k u_k h_k\right) \cdot \frac{d}{dh_i}\left(\sum_k u_k h_k\right) \tag{1.8}$$

$$= r'\left(\sum_k u_k h_k\right) \cdot u_i \tag{1.9}$$

$$= r'(s_y)u_i \tag{1.10}$$

and since $h_i = r\left(\sum_k w_{ik} x_k\right)$ we have that

$$\frac{dh_i}{dw_{ij}} = r'\left(\sum_k w_{ik} x_i\right) \cdot \frac{d}{dw_{ij}}\left(\sum_k w_{ik} x_k\right) \tag{1.11}$$

$$= r'\left(\sum_k w_{ik} x_i\right) \cdot x_j \tag{1.12}$$

If we write $s_{h_i} = \sum_k w_{ik} h_k$, then we can write

$$\frac{dh_i}{dw_{ij}} = r'(s_{h_i})x_i$$

Combining all of our calculations, we then get that

$$\frac{dL}{dw_{ij}} = -(t-y) \cdot r'(s_y)u_i \cdot r'(s_{h_i})x_j$$

If we define $\delta = -(t-y)r'(s_y)$, interpreting it as an error term, we obtain the following explicit weight update formulas.

$$u'_1 = u_1 - \delta h_1 \tag{1.13}$$
$$u'_2 = u_2 - \delta h_2 \tag{1.14}$$
$$w'_{11} = w_{11} - \delta u_1 r'(s_{h_1}) x_1 \tag{1.15}$$
$$w'_{12} = w_{12} - \delta u_1 r'(s_{h_1}) x_2 \tag{1.16}$$
$$w'_{21} = w_{21} - \delta u_2 r'(s_{h_2}) x_1 \tag{1.17}$$
$$w'_{22} = w_{22} - \delta u_2 r'(s_{h_2}) x_2 \tag{1.18}$$
$$\tag{1.19}$$

Recall that the weights u_3 (in U) and w_{13}, w_{23} (in W) correspond to the bias parameters in our model. Their weight update formulas are much simpler, since the value of their origin node is automatically fixed to 1.

$$u'_3 = u_3 - \delta \tag{1.20}$$
$$w'_{13} = w_{13} - \delta u_1 r'(s_{h_1}) \tag{1.21}$$
$$w'_{23} = w_{23} - \delta u_2 r'(s_{h_2}) \tag{1.22}$$
$$\tag{1.23}$$

This completes our description for how we update our model's weights given one training example. In practice, however, we tend to have many training examples $t_1, t_2, \ldots, t_N$ that we can use to update our model. There are three main approaches as to exactly how we can update our model's weights using all of the training examples.

- **Stochastic Gradient Descent.** Update the model's weights after each training example t_i.
- **Batch Gradient Descent.** Update the model's weights after showing it every training example. In this case, we could achieve this by collecting all of the gradients calculated from each training example and then averaging them.

$$w'_{ij} = w_{ij} - \frac{1}{N} \sum_{t_k} \frac{dL(t_k)}{dw_{ij}}$$

- **Mini-Batch Gradient Descent.** Update the model's weights after showing it $n < N$ many training examples; we'd call n our **batch size**. This can be thought of as a compromise between stochastic and batch gradient descent.

These three methods trade off training speed and model accuracy. As mini-batch is a compromise between speed and accuracy, it tends to be preferred in practice. In any case, in practice we tend to train the model on the training set each time; each iteration is called an **epoch**, and so a training procedure may undergo several epochs. Too few epochs will lead to poor accuracy, but too many epochs will leads to poor generalization outside of the training set, a concept known as **overfitting**.

For this example, if we perform batch gradient descent on the model we presented, using our four training examples of the XOR function, training this for tens of thousands of epochs will allow us to converge on these model weighs

$$W = \begin{bmatrix} 1.52183263 & 1.55282077 & -1.55282077 \\ 1.0637993 & 1.0637993 & 0.39200322 \end{bmatrix} \tag{1.24}$$

$$U = \begin{bmatrix} -1.31420497 & 0.94002694 & -0.36849359 \end{bmatrix} \tag{1.25}$$

which successfully mimic the XOR function.

1.5 Backpropagation: More generally