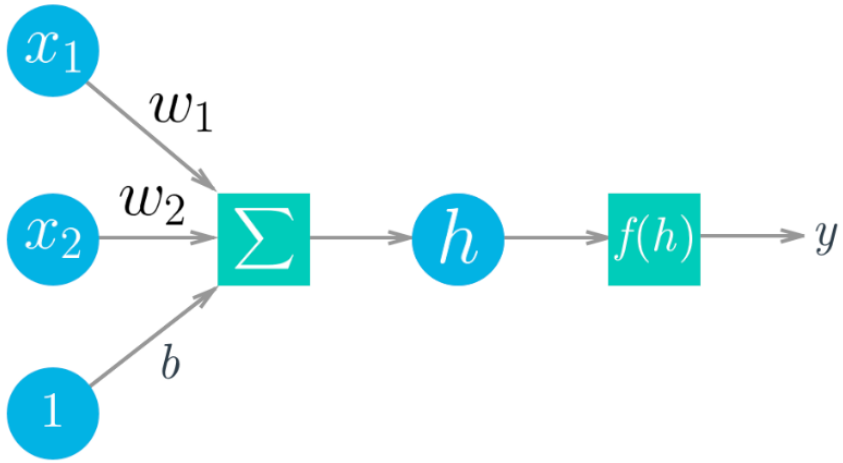
Neural Networks

Deep Learning is based on artificial neural networks which have been around in some form since the late 1950s. The networks are built from individual parts approximating neurons, typically called units or simply "neurons." Each unit has some number of weighted inputs. These weighted inputs are summed together (a linear combination) then passed through an activation function to get the unit's output.



Mathematically this looks like:

$$y = f(w_1 x_1 + w_2 x_2 + b)$$

$$y = f\left(\sum_i w_i x_i + b\right)$$

With vectors this is the dot/inner product of two vectors:

$$h = \begin{bmatrix} x_1 \, x_2 \cdots x_n \end{bmatrix} \cdot \begin{bmatrix} w_1 \\ w_2 \\ \vdots \\ w_n \end{bmatrix}$$

Let us see some tensors

Tensors

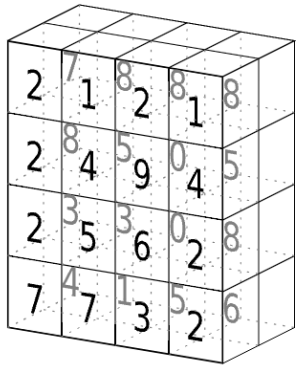
It turns out neural network computations are just a bunch of linear algebra operations on *tensors*, a generalization of matrices. A vector is a 1-dimensional tensor, a matrix is a 2-dimensional tensor, an array with three indices is a 3-dimensional tensor (RGB color images for example). The fundamental data structure for neural networks are tensors and PyTorch (as well as pretty much every other deep learning framework) is built around tensors.

't'
'e'
'n'
's'
'o'
'r'

tensor of dimensions [6]
(vector of dimension 6)

3	1	4	1
5	9	2	6
5	3	5	8
9	7	9	3
2	3	8	4
6	2	6	4

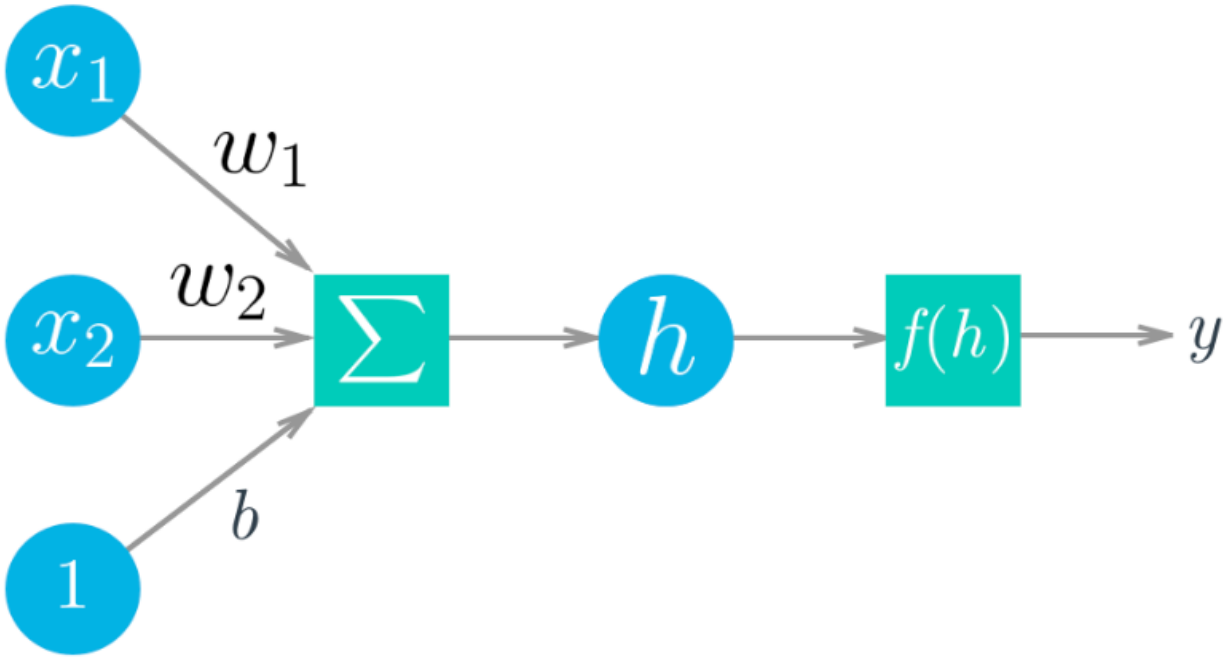
tensor of dimensions [6,4]
(matrix 6 by 4)



tensor of dimensions [4,4,2]

With the basics covered, it's time to explore how we can use PyTorch to build a simple neural network.

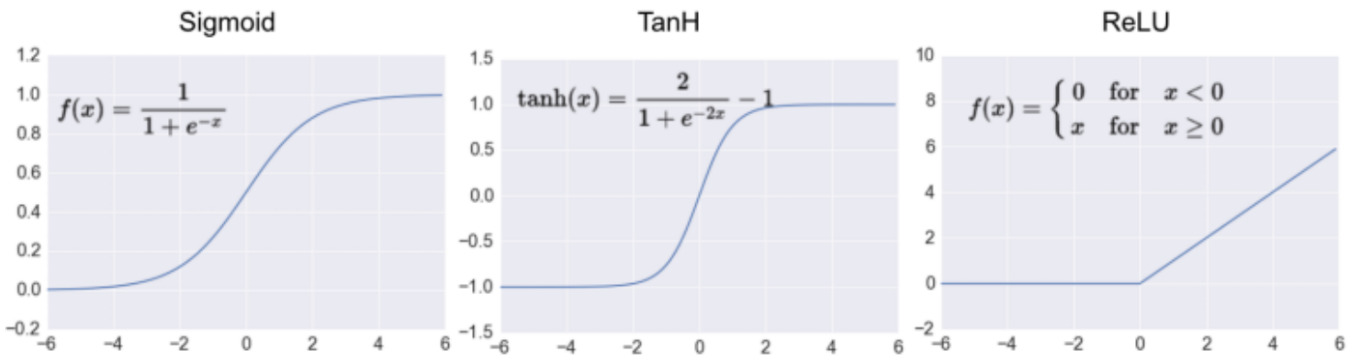
Let us implement - computation of h



Let us implement - activation function

Activation functions

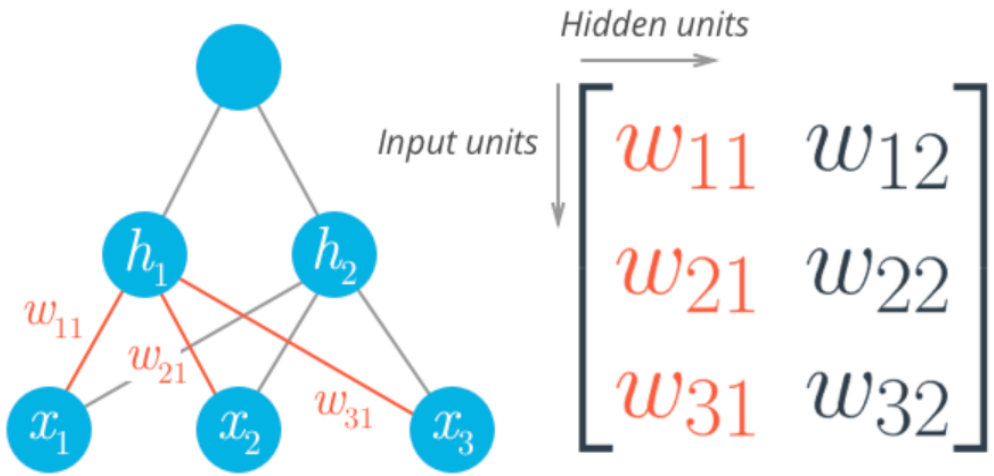
So far we've only been looking at the sigmoid and softmax activation functions, but in general any function can be used as an activation function. The only requirement is that for a network to approximate a non-linear function, the activation functions must be non-linear. Here are a few more examples of common activation functions: Tanh (hyperbolic tangent), and ReLU (rectified linear unit).



In practice, the ReLU function is used almost exclusively as the activation function for hidden layers.

Stack them up!

That's how you can calculate the output for a single neuron. The real power of this algorithm happens when you start stacking these individual units into layers and stacks of layers, into a network of neurons. The output of one layer of neurons becomes the input for the next layer. With multiple input units and output units, we now need to express the weights as a matrix.



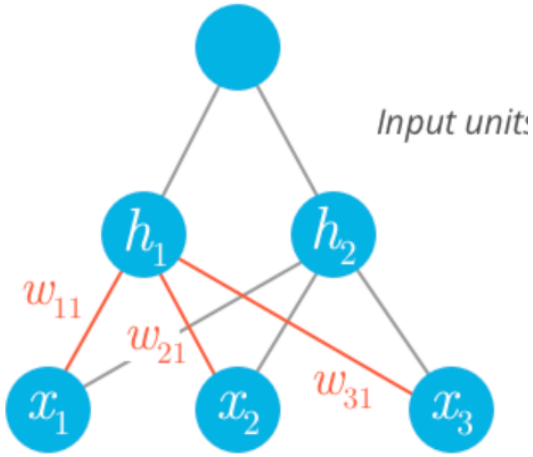
The first layer shown on the bottom here are the inputs, understandably called the **input layer**. The middle layer is called the **hidden layer**, and the final layer (on the right) is the **output layer**. We can express this network mathematically with matrices again and use matrix multiplication to get linear combinations for each unit in one operation. For example, the hidden layer (h_1 and h_2 here) can be calculated

$$\vec{h} = [h_1\, h_2] = [x_1\, x_2 \cdots x_n] \cdot \begin{bmatrix} w_{11} & w_{12} \\ w_{21} & w_{22} \\ \vdots & \vdots \\ w_{n1} & w_{n2} \end{bmatrix}$$

The output for this small network is found by treating the hidden layer as inputs for the output unit. The network output is expressed simply

$$y = f_2\big(\, f_1\big(\vec{x}\, \mathbf{W_1}\big)\mathbf{W_2}\big)$$

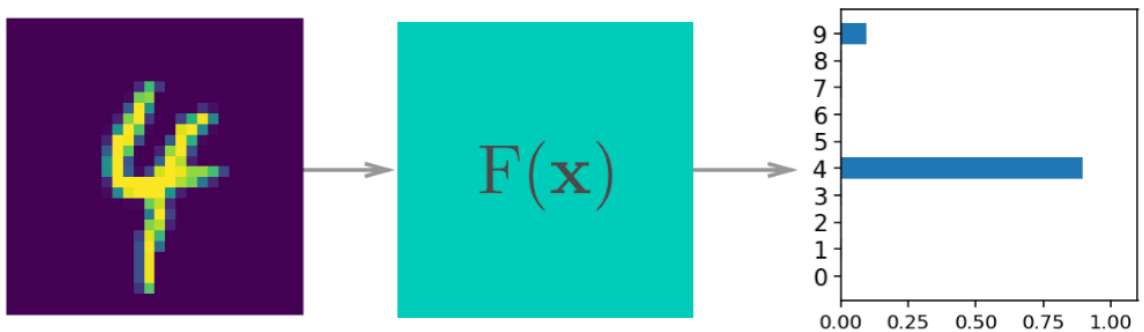
Let us implement



Let us solve MNIST

Now we're going to build a larger network that can solve a (formerly) difficult problem, identifying text in an image. Here we'll use the MNIST dataset which consists of greyscale handwritten digits. Each image is 28x28 pixels, you can see a sample below

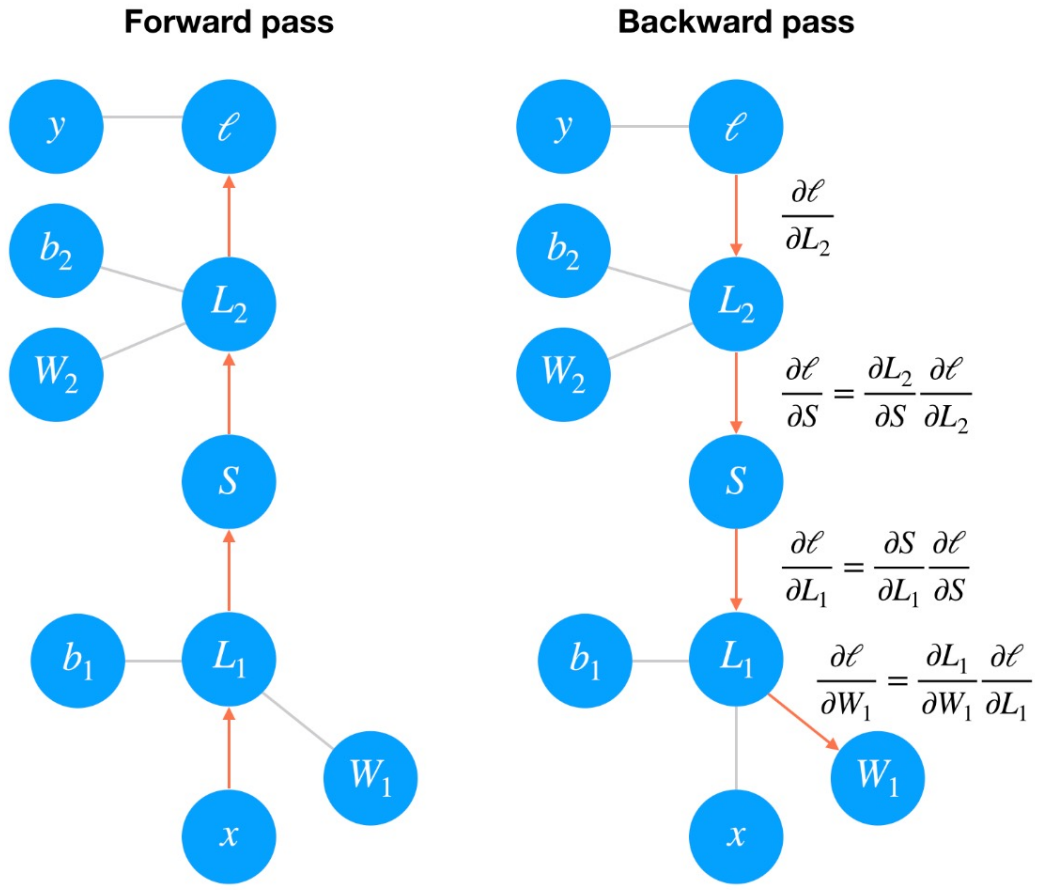
The network we built in the previous part isn't so smart, it doesn't know anything about our handwritten digits. Neural networks with non-linear activations work like universal function approximators. There is some function that maps your input to the output. For example, images of handwritten digits to class probabilities. The power of neural networks is that we can train them to approximate this function, and basically any function given enough data and compute time.



At first the network is naive, it doesn't know the function mapping the inputs to the outputs. We train the network by showing it examples of real data, then adjusting the network parameters such that it approximates this function.

To find these parameters, we need to know how poorly the network is predicting the real outputs. For this we calculate a **loss function** (also called the cost), a measure

Training multilayer networks is done through **backpropagation** which is really just an application of the chain rule from calculus. It's easiest to understand if we convert a two layer network into a graph representation.



In the forward pass through the network, our data and operations go from bottom to top here. We pass the input x through a linear transformation L_1 with weights W_1 and biases b_1. The output then goes through the sigmoid operation S and another linear transformation L_2. Finally we calculate the loss ℓ. We use the loss as a measure of how bad the network's predictions are. The goal then is to adjust the weights and biases to minimize the loss.

To train the weights with gradient descent, we propagate the gradient of the loss backwards through the network. Each operation has some gradient between the inputs and outputs. As we send the gradients backwards, we multiply the incoming gradient with the gradient for the operation. Mathematically, this is really just calculating the gradient of the loss with respect to the weights using the chain rule.

$$\frac{\partial \ell}{\partial W_1} = \frac{\partial L_1}{\partial W_1} \frac{\partial S}{\partial L_1} \frac{\partial L_2}{\partial S} \frac{\partial \ell}{\partial L_2}$$

Optimization step after backpropagation

We update our weights using this gradient with some learning rate α.

$$W_1' = W_1 - \alpha \frac{\partial \ell}{\partial W_1}$$

The learning rate α is set such that the weight update steps are small enough that the iterative method settles in a minimum.

Stability of large value input

$$
\begin{aligned}
log(softmax(x)) &= log(\frac{exp(x_i)}{\sum_{j=1}^{n} exp(x_j)}) = log(exp(x_i)) - log(\sum_{i=1}^{n} exp(x_j)) \\
&= x_j - log(\sum_{i=1}^{n} exp(x_j)) = x_j - log(\sum_{i=1}^{n} exp(x_j - c)exp(c)) \\
&= x_j - log(exp(c)\sum_{i=1}^{n} exp(x_j - c)) = x_j - c - log(\sum_{i=1}^{n} exp(x_j - c))
\end{aligned}
$$