Introduction to Machine Learning

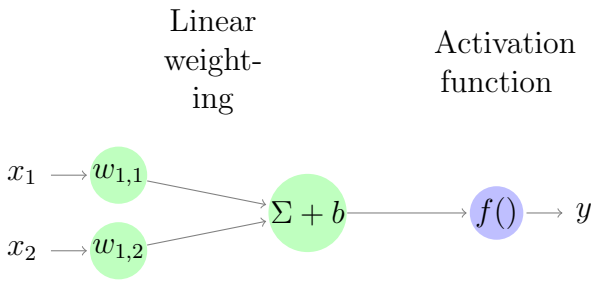
August 22, 2020

0.1 Network structure

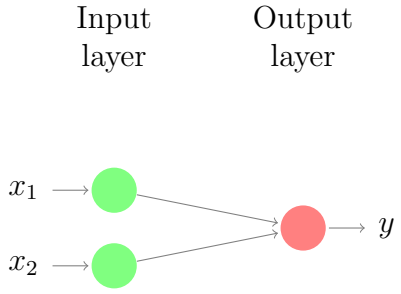
Let's start simply with trying to learn an AND gate. In short, we want to input one of the following pairs of x_1, x_2 and output y. The truth table (which should be familiar) is

x_1	x_2	y
0	0	0
1	0	0
0	1	0
1	1	1

This data can be learned using a single layer network which looks like the following



NOTE: A more confusing yet popular way to draw this is shown below. In the common literature, the linear weighting and activation function is often combined into a single layer. For the sake of being explicit, we will be referring to the *above* drawing.



0.2 Forward Propagation

In order to understand what is happening when we *predict* or *infer* using the neural network, we need to express the drawing in terms of mathematics.

The linear weighting multiplies $x_1 * w_{11}$ and $x_2 * w_{12}$, then these are summed together with an offset bias, b. In equation form this looks like the following.

$$W = \begin{bmatrix} w_{11} & w_{12} \end{bmatrix}$$
$$X = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}$$

$$g = WX + b$$

This is the output after the two sets of green neurons; multiplication by the weights w_{11}, w_{12} and addition with an offset, b.

The activation function simply takes g and applies a non-linear mapping to output values. For example, we know that our outputs can take on values of 0 or 1. Therefore, we are interested in using an activation function that outputs only values between 0 and 1. Here the sigmoid function will do the trick.

$$y = f(g) = \frac{1}{1 + e^{-g}}$$

This is our ***forward*** propagation. Notice it is nothing special! Just matrix multiplication and a mathematical function. To see how it works, assume our weights are $w_{11} = w_{12} = 0.5$ and our bias $b = 0$. Work through the math to get the outputs. (It may be easier to use MATLAB or Python).

0.3 Gradient Descent

0.3.1 1D

Having worked through the math for each input pair, you should have noticed that the output is not what it needs to be from the truth table! We can reconcile this by changing the weights and retrying. The ultimate question

is to figure out how to change our weights. For example, I could arbitrarily choose new weights and ask you to resolve the forward propagation. Do this over and over until you find a set of weights and biases that work, but that will take a long time. Instead, we want to update our weights *beneficially*. We can accomplish this using the method of gradient descent.

For simplicity, let's assume we are dealing with a 1 dimensional problem. Furthermore, let's ignore the activation function for now and only deal with the following equation.

$$y = wx + b$$

The question is, "given an input, x, and an output, y, find w and b." This is a simple algebra problem, but we're going to solve it in a scaleable manner that can be applied to a multi-dimensional problem.

The first step is to define an error metric. This *cost* function will describe how far away our guess for w is. Furthermore, our cost function should not care whether we are above or below the correct w meaning it should be a *distance* measure. A common solution is to use the mean squared error (MSE). Here's how it works.

1. Make a guess for w and b.
2. Using our known input, x, *forward* propagate through the equation to end up with an estimate for y.
3. Figure out how far away our estimate of y is from the true value.

In keeping with estimation theory notation, our estimate of y will be denoted by $\hat{y}$. Thus, using MSE, the error between the estimate $\hat{y} = wx + b$ and the true value of y will be

$$E = (\hat{y} - y)^2 = (wx + b - y)^2$$

The foundation of gradient descent is formed from the cost function having a global minimum where $\hat{y} = y$. You may be inclined to minimize the cost function by setting $\frac{\partial E}{\partial \hat{y}} = 0$ and solving for $\hat{y}$. Unfortunately, we don't have control over $\hat{y}$, we can only choose w and b. Furthermore, there is clearly more than one solution for w and b such that $\hat{y} = y$.

Instead, we notice that the derivative $\frac{\partial E}{\partial w}$ tells me the direction of greatest change of the error if I change w. Thus, moving in the negative $\frac{\partial E}{\partial w}$ will decrease the error. The same argument can be used for b.

Then

$$\frac{\partial E}{\partial w} = 2(wx + b - y)x$$
$$\frac{\partial E}{\partial b} = 2(wx + b - y)$$

And each iteration n, we should update our w and b guess such that

$$w_n = w_{n-1} - \frac{\partial E}{\partial w_{n-1}} = w_{n-1} - 2(w_{n-1}x + b - y)x$$
$$b_n = b_{n-1} - \frac{\partial E}{\partial b_{n-1}} = b_{n-1} - 2(w_{n-1}x + b - y)$$

0.3.2 2D

In the case of a multi-dimensional equation (with matrices), we need to figure out the update for each weight and bias. Still ignoring the activation function, we find the prediction $\hat{y}$ for our 2D case to be

$$\hat{y} = w_{11}x_1 + w_{12}x_2 + b$$

Giving an error of

$$E = (\hat{y} - y)^2 = (w_{11}x_1 + w_{12}x_2 + b - y)^2$$

For the first weight, we find

$$\frac{\partial E}{\partial w_{11}} = 2(w_{11}x_1 + w_{12}x_2 + b - y)x_1$$

And for the second, we get

$$\frac{\partial E}{\partial w_{12}} = 2(w_{11}x_1 + w_{12}x_2 + b - y)x_2$$

while the bias error is

$$\frac{\partial E}{\partial b} = 2(w_{12}x_1 + w_{12}x_2 + b - y)$$

Let's pause here and see if we notice a pattern. It looks like all of the gradients have a similar form. This shouldn't be surprising since it's just the chain rule from differential calculus. Notice we can rewrite the above gradients as

$$\begin{aligned}\frac{\partial E}{\partial w_{11}} &= \frac{\partial E}{\partial \hat{y}}\frac{\partial \hat{y}}{\partial w_{11}} = \frac{\partial E}{\partial \hat{y}}x_1 \\ \frac{\partial E}{\partial w_{12}} &= \frac{\partial E}{\partial \hat{y}}\frac{\partial \hat{y}}{\partial w_{12}} = \frac{\partial E}{\partial \hat{y}}x_2 \\ \frac{\partial E}{\partial b} &= \frac{\partial E}{\partial \hat{y}}\frac{\partial \hat{y}}{\partial b}\end{aligned} \tag{1}$$

Writing in this way makes it a lot easier to add our activation function back in. We realize all we have to do is add another derivative. Remember, we split it above into

$$\begin{aligned} g &= wx + b \\ \hat{y} &= \frac{1}{1+e^{-g}}\end{aligned}$$

$$\frac{\partial E}{\partial w_{11}} = \frac{\partial E}{\partial \hat{y}}\frac{\partial \hat{y}}{\partial g}\frac{\partial g}{\partial w_{11}} = \frac{\partial E}{\partial \hat{y}}\frac{\partial \hat{y}}{\partial g}x_1 \tag{2}$$

$$\frac{\partial E}{\partial w_{12}} = \frac{\partial E}{\partial \hat{y}}\frac{\partial \hat{y}}{\partial g}\frac{\partial g}{\partial w_{12}} = \frac{\partial E}{\partial \hat{y}}\frac{\partial \hat{y}}{\partial g}x_2 \tag{3}$$

$$\frac{\partial E}{\partial b} = \frac{\partial E}{\partial \hat{y}}\frac{\partial \hat{y}}{\partial g}\frac{\partial g}{\partial b}$$

All that's left to do is subtract these gradients from each weight/bias on each iteration. Keep in mind the chain rule since that will come in handy during back propagation.

You are welcome to add an additional parameter called a *learning rate*. This is a scalar value to be multiplied by the gradient that controls how large a step in the gradient direction. The learning rate is our first **hyperparameter**: a parameter we have control over during the training phase.

0.3.3 Matrix form

Now that we understand gradient descent for multiple dimensions, we are inclined to write the weight updates in matrix form.

$$\begin{bmatrix} w_{11} & w_{12} \end{bmatrix} = \begin{bmatrix} w_{11} & w_{12} \end{bmatrix} - \begin{bmatrix} \frac{\partial E}{\partial w_{11}} & \frac{\partial E}{\partial w_{12}} \end{bmatrix}$$

Expanding the gradients out like equation 2, we can re-write the updates as

$$\begin{bmatrix} w_{11} & w_{12} \end{bmatrix} = \begin{bmatrix} w_{11} & w_{12} \end{bmatrix} - \frac{\partial E}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial g} \begin{bmatrix} x_1 & x_2 \end{bmatrix}$$

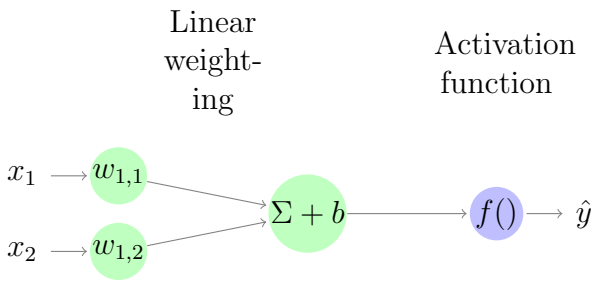
In matrix form, this is

$$W = W - \frac{\partial E}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial g} X^T \tag{4}$$

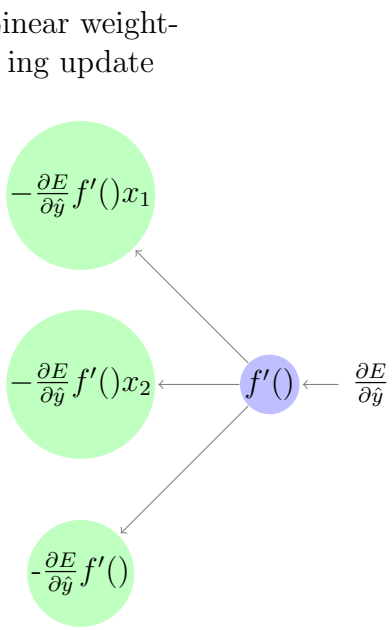
$$b = b - \frac{\partial E}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial g} \tag{5}$$

0.4 Backpropagation

Let's return briefly to our network graph



Now that we understand what is needed to update our weights, let's draw a *backward* graph. After predicting $\hat{y}$, our update equations (4, 5) indicate that we need to propagate $\frac{\partial E}{\partial \hat{y}}$ backwards through the network.



Why is this important? Well it tells us each layer we implement will forward propagate to produce an output. At the end of that process, we need to implement a cost function that calculates what our current cost is, and then ***propagates the error*** **backwards** ***to the layer above***. This is the foundation of backpropagation and learning in neural networks. In the drawing above, this is illustrated as such.

1. The error function will pass $\frac{\partial E}{\partial \hat{y}}$ to the activation layer.
2. The activation layer will multiply $\frac{\partial E}{\partial \hat{y}} \times f'()$ where $f'()$ is the derivative of the activation function chosen.
3. The linear layer receives the *error from layer below* (activation layer) and uses this to update it's weights following equations 4 and 5

While we derived this for a simple single layer network, the exact same equations hold for any number of layers! When using more layers, we will have multiple linear layers that need to be trained using gradient descent. In order to achieve this, we need to continue to back propagate an error to the next layer. I encourage you do work through the math the same way we did for a single layer network, but the equation is shown in the section below.

Before we summarize the equations and the classes for implementation, we need to understand what GPU's have to do with this.

0.5 Batch, Stochastic, Mini Batch Training

I want to emphasize there are inherent trade offs of each of these training methods and there are no correct solution. It depends heavily on the application and the dataset so don't assume one of these is better than another.

0.5.1 Batch

Until now, we've been training one input data point at a time. The idea behind *batch* training is to pass all the training data into the network at the same time and execute gradient descent. The next iteration, we continue to pass all the data and update the weights and continue to convergence. Above, we assumed passing one input at a time. Here let's pass all of them.

$$W = \begin{bmatrix} w_{11} & w_{12} \end{bmatrix}$$
$$X = \begin{bmatrix} 0 & 0 & 1 & 1 \\ 0 & 1 & 0 & 1 \end{bmatrix}$$

$$G = WX + b$$

Notice that the output G is now a vector! You can follow the same method as above to figure out our backpropagation and gradient descent equations in matrix form. These are summarized in section 0.6.

One of big differences now is that our cost function will have to change. Instead of

$$E = (\hat{y} - y)^2$$

we'll need to sum over all m predictions; four in our case (one for each of our input data)

$$E = \frac{1}{m} \sum_{i=0}^{m-1} (\hat{y}[i] - y)^2$$

And the derivative $\frac{\partial E}{\partial \hat{y}[i]}$ needs to exist for every prediction (4 in this case)

$$E'(\hat{y}[i]) = \frac{2}{m}(\hat{y}[i] - y)$$

The error we propagate backwards is now a *row* vector.

To summarize, instead of passing a single input each epoch, we pass all the data and update the weights based on all the data. We still need to run for many epochs, but we're able to train on multiple data points simultaneously; a perfect job for GPUs.

0.5.2 Stochastic

In the *stochastic* method, often called Stochastic Gradient Descent (SGD), we return to passing the data one input at a time. The difference here is that we randomize the input data going into the system. The algorithm looks like

1. Randomly choose an input/output pair from the truth table.
2. Forward propagate through the network to get a prediction
3. Backpropagate the error from the prediction to the true output and update the weights.
4. Repeat

It might not make much sense for such a small dataset, but this is an important change for large datasets because it will help avoid local minimum when training with randomized input data.

0.5.3 Mini-batch

This method is exactly what it sounds like. Instead of training on the entire dataset each iteration (like batch training), we can split the dataset into a lot of mini batches to pass to the network simultaneously. The size of the mini-batch is another hyperparameter.

Furthermore, we are free to randomize the dataset before splitting into smaller dataset (essentially combining SGD and mini-batch training). This should give you an idea of how complicating training can become.

0.6 Implementation

We will be implementing the Batch gradient descent, but it can be used for mini-batch gradient descent. The only difference is how we structure our

data (whether we pass 4 inputs at a time or 2 inputs at a time). Below are the matrix equations we need to implement.

0.6.1 Cost Function

- *cost():* Calculates the cost function of the given weights:

$$output = E(input)$$

- *dCost():* Calculates the derivative of the cost function:

$$output = E'(input)$$

NOTES: Below are the definitions of the cost function for MSE in batch mode. Notice that E' has i terms indicating it's a vector. For sake of continuity, we will assume it's a *row* vector. This means $E'(y[i])$ is the i^{th} column of E'

$$E(\hat{Y}) = \frac{1}{m}\sum_{i=0}^{m}(\hat{y}[i] - y[i])^2$$

$$E'(\hat{y}[i]) = 2(\hat{y}[i] - y[i])$$

0.6.2 Activation

- *forward():* Performs the forward activation function:

$$output = f(input)$$

- *backprop():* Takes the error from the layer below (E_B) and calculates the error to feed to the layer above (E_A):

$$E_A = E_B * f'(input)$$

NOTES: Below are the definitions of the sigmoid function

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

$$f'(g) = f(g) * (1 - f(g))$$

0.6.3 Linear Layer

- *forward():* Performs the forward matrix multiply: $output = WX + b$
- *backprop():* Takes the error from the layer below (E_B) and calculates the error to feed to the layer above (E_A):

$$E_A = W^T E_B$$

- *updateWeights():* Takes the error from the layer below (E_B) and updates the weights:

$$\frac{1}{m} E_B X^T$$

- *updateBias():* Takes the error from the layer below (E_B) and updates the bias:

$$\frac{1}{m} \sum_{i=0}^{m} E_B^{(i)}$$

NOTES: m is the batch size. $E_B^{(i)}$ is the i^{th} column of the batch. In batch mode, E_B will be a matrix (see Cost Function).