Nothing but NumPy: Understanding & Creating Neural Networks with Computational Graphs from Scratch

<= Previous post
Next post =>

Tags: Backpropagation, Neural Networks, numpy, Python

Entirely implemented with NumPy, this extensive tutorial provides a detailed review of neural networks followed by guided code for creating one from scratch with computational graphs.

comments

By Rafay Khan.

Understanding new concepts can be hard, especially these days when there is an avalanche of resources with only cursory explanations for complex concepts. This blog is the result of a dearth of detailed walkthroughs on how to create neural networks in the form of computational graphs.

In this blog posts, I consolidate all that I have learned as a way to give back to the community and help new entrants. I will be creating common forms of neural networks all with the help of nothing but NumPy.

This blog post is divided into two parts, the first part will be understanding the basics of a neural network and the second part will comprise the code for implementing everything learned from the first part.

Part I: Understanding a Neural Network

Let's dig in

Neural networks are a model inspired by how the brain works. Similar to neurons in the brain, our 'mathematical neurons' are also, intuitively, connected to each other; they take inputs(dendrites), do some simple computation on them and produce outputs(axons).

The best way to learn something is to build it. Let's start with a simple neural network and hand-solve it. This will give us an idea of how the computations flow through a neural network.

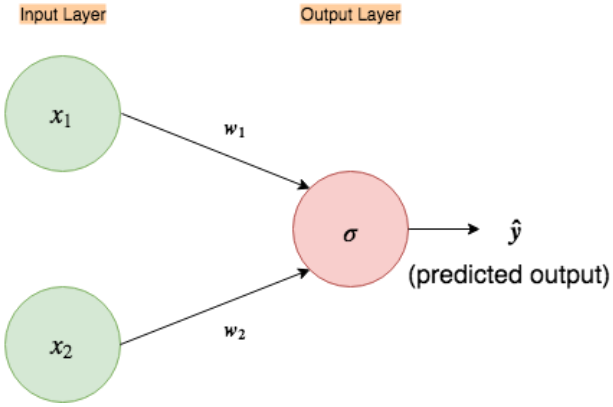


Fig 1. Simple input-output only neural network.

As in the figure above, most of the time you will see a neural network depicted in a similar way. But this succinct and simple looking picture hides a bit of the complexity. Let's expand it out.

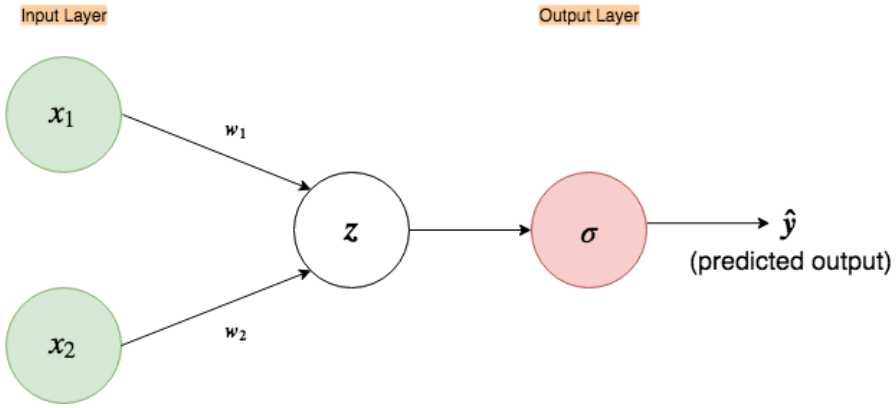


Fig 2. Expanded neural network

Now, let's go over each node in our graph and see what it represents.

Fig 3. Inputs nodes x_1 and x_2

These nodes represent our inputs for our first and second features, x_1 and x_2, that define a single example we feed to the neural network, thus called ***"Input Layer"***

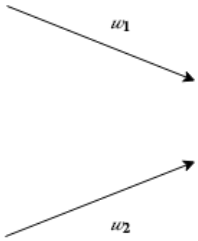


Fig 4. Weights

w_1 and w_2 represent our weight vectors (in some neural network literature it is denoted with the *theta* symbol, θ). Intuitively, these dictate how much influence each of the input features should have in computing the next node. If you are new to this, think of them as playing a similar role to the 'slope' or 'gradient' constant in a linear equation.

Weights are the main values our neural network has to "learn". So initially, we will set them to ***random values*** and let the "*learning algorithm"* of our neural network decide the best weights that result in the correct outputs.

Why random initialization? More on this later.

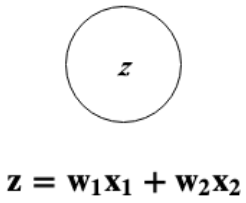


Fig 5. Linear operation

This node represents a linear function. Simply, *it takes all the inputs coming to it and creates a linear equation/combination out of them*. (*By convention, it is understood that a linear combination of weights and inputs is part of each node, except for the input nodes in the input layer, thus this node is often omitted in figures, like in Fig.1.* In this example, I'll leave it in)

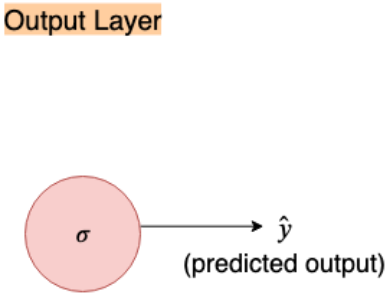
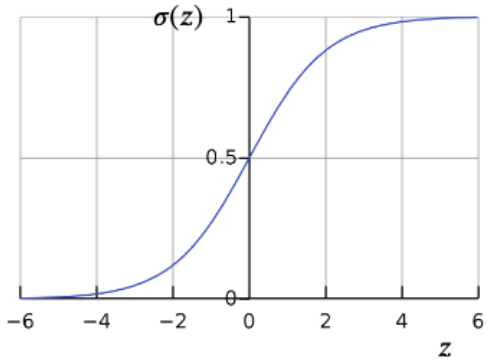


Fig 6. output node

This σ node takes the input and passes it through the following function, called the **sigmoid function**(because of its S-shaped curve), also known as the **logistic function**:



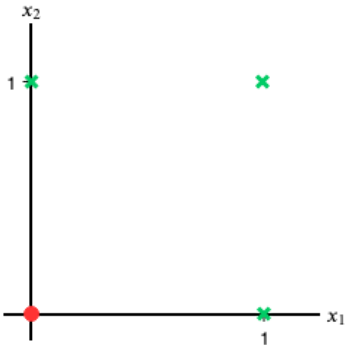
$$\sigma(z) = \frac{1}{1 + e^{-z}}$$

Fig 7. Sigmoid(Logistic) function

Sigmoid is one of the many "activations functions" used in neural networks. The job of an activation function is to change the input to a different range. For example, if $z > 2$ then, $\sigma(z) \approx 1$ and similarly, if $z < -2$ *then,* $\sigma(z) \approx 0$. *So*, the sigmoid function squashes the output range to (0, 1) *(this '()' notation implies exclusive boundaries; never completely outputs 0 or 1 as the function asymptotes, but reaches very close to boundary values)*

In our above neural network since it is the last node, it performs the function of output. The predicted output is denoted by $\hat{y}$. (*Note: in some neural network literature this is denoted by* ***'h(θ)'****, where 'h' is called the hypothesis i.e. this is the hypothesis of the neural network, a.k.a the output prediction, given parameter θ; where θ are weights of the neural networks)*

Now that we know what each and everything represents let's flex our muscles by computing each node by hand on some dummy data.



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

Fig 8. OR gate

The data above represents an **OR** gate(output 1 if any input is 1). Each row of the table represents an *'example'* we want our neural network to learn from. After learning from the given examples we want our neural network to perform the function of an OR gate; given the input features, x_1 and x_2, try to output the corresponding ***y(also called 'label')***. I have also plotted the points on a 2-D plane so that it is easy to visualize(green crosses represent points where the output(***y)*** is ***1*** and the red dot represents the point where the output is ***0***).

This OR-gate data is particularly interesting, as it is ***linearly separable*** i.e. we can draw a straight line to separate the green cross from the red dot.

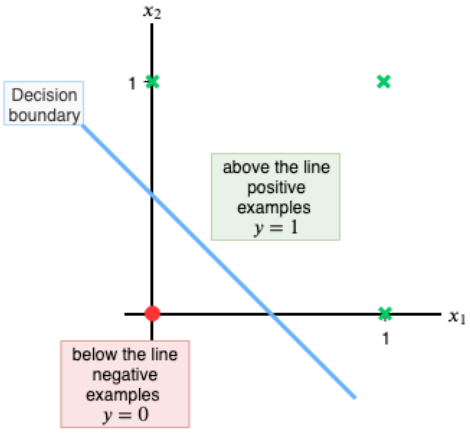
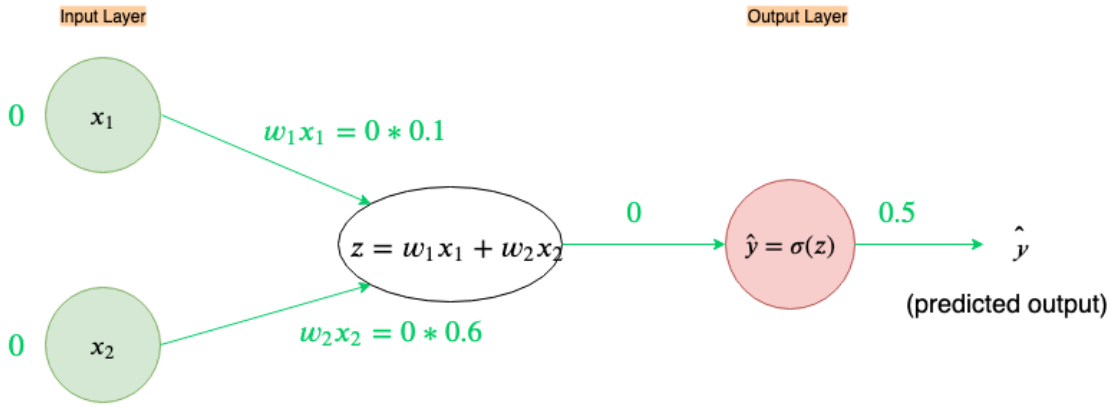


Fig 9. Showing that the OR gate data is linearly separable

We'll shortly see how our simple neural network performs this task.

Data flows from left-to-right in our neural network. In technical terms, this process is called **'forward propagation'**; the computations from each node are forwarded to the next node, it is connected to.

Let's go through all the computations our neural network will perform on given the first example, $x_1=0$, and $x_2=0$. Also, we'll initialize weights $w_1$and w_2 to $w_1=0.1$ and $w_2=0.6$ *(recall, these weights a have been randomly selected)*



The neural network is going through the following computations(forward computations marked in green):

- Our input for first example $x_1 = 0, x_2 = 0$
- Randomly initialized weights $w_1 = 0.1, w_2 = 0.6$
- $z = w_1x_1 + w_2x_2 = 0 * 0.1 + 0 * 0.6 = 0$
- $\hat{y} = \sigma(z) = \frac{1}{1+e^{-z}} = \frac{1}{1+e^{0}} = 0.5$

Fig 10. Forward propagation of the first example from OR table data

With our current weights, $w_1= 0.1$ and $w_2 = 0.6$, our network's output is a bit far from where we'd like it to be. The predicted output, $\hat{y}$, should be $\hat{y}\approx0$for $x_1=0$ and $x_2=0$, right now its $\hat{y}=0.5$.

So, how does one tell a neural network how far it is from our desired output? In comes the ***Loss Function*** to the rescue.

Loss Function

*The **Loss Function** is a simple equation that tells us how far our neural network's predicted output($\hat{y}$) is from our desired output(y), **for ONE example, only.***

*The **derivative** of the loss function dictates whether to increase or decrease weights. A positive derivative would mean decrease the weights and negative would mean increase the weights. **The steeper the slope the more incorrect the prediction was.***

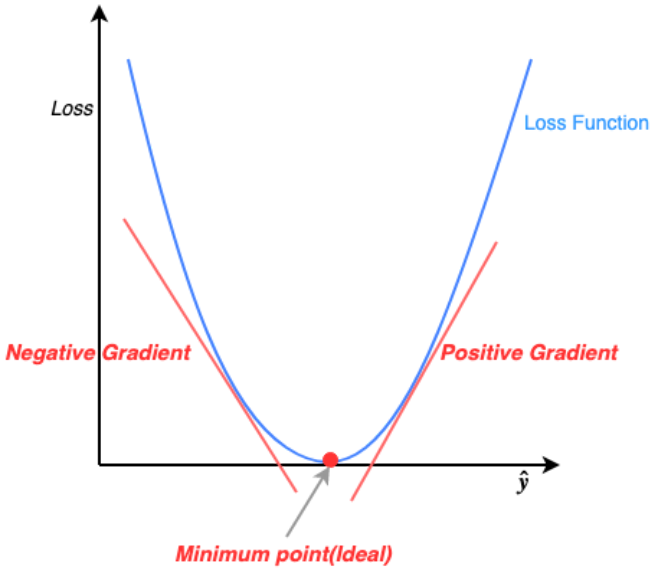


Fig 11. Loss function visualized

The Loss function curve depicted in Figure 11 is an ideal version. In real-world cases, the Loss function may not be so smooth, with some bumps and saddles points along the way to the minimum.

There are many different kinds of loss functions ***each essentially calculating the error between predicted output and desired output***. Here we'll use one of the simplest loss functions, the ***squared-error Loss function.*** Defined as follows:

$$\text{Loss} = L(y, \hat{y}) = \frac{1}{2}(y - \hat{y})^2$$

Where y is the actual desired output, and
$\hat{y}$ is the predicted output

Fig 12. Loss Function. Calculating error for a single example

Taking the **square keeps everything nice and positive** and the **fraction (1/2) is there so that it cancels out when taking the derivative of the squared term** *(it is common among some machine learning practitioners to leave the fraction out)*.

Intuitively, the Squared Error Loss function helps us in minimizing the vertical distance between our predictor line(blue line) and actual data(green dot). Behind the scenes, this predictor line is our ***z***(linear function) node.

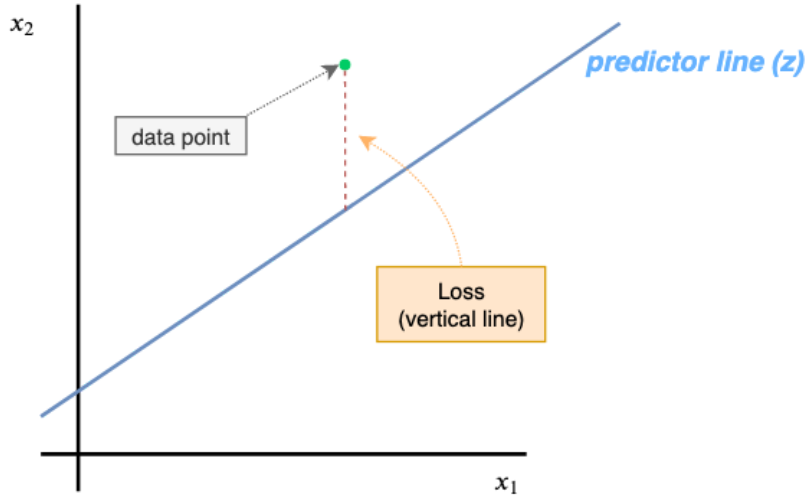


Fig 13. Visualization of the effect of the Loss function

Now that we know the purpose of a Loss function let's calculate the error in our current prediction ***ŷ=0.5,*** given ***y=0***

$$\textit{Loss} = L(\hat{y}, y) = \frac{1}{2}(y - \hat{y})^2 = \frac{1}{2}(0 - 0.5)^2 = \frac{1}{2}(-0.5)^2 = \frac{1}{8} = 0.125$$

- For our current example $\hat{y} = 0.5$ and $y = 0$

Fig 14. Loss calculated for 1st example

As we can see the Loss is ***0.125***. Given this, *we can now use the derivative of the Loss function to check whether we need to increase or decrease our weights.*

This process is called ***backpropagation,*** as we'll be doing the opposite of the *forward* phase. Instead of going from input to output we'll track backward from output to input. Simply, backpropagation allows us to figure out how much of the Loss each part of the neural network was responsible for.

To perform backpropagation we'll employ the following technique: *at each node, we only have our local gradient computed(partial derivatives of that node), then during backpropagation, as we are receiving numerical values of gradients from upstream, we take these and multiply with local gradients to pass them on to their respective connected nodes.*

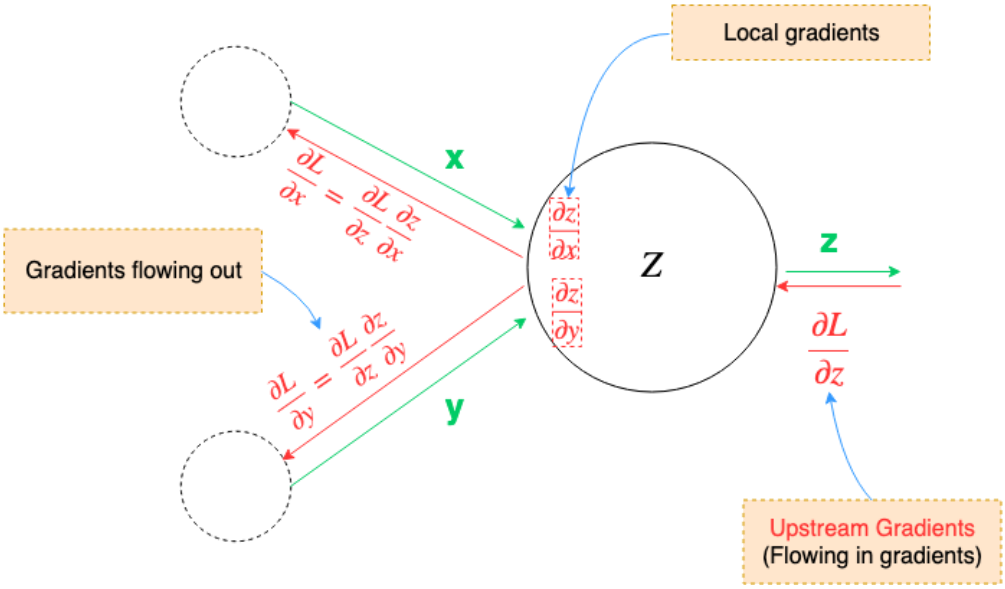


Fig 15. Gradient Flow

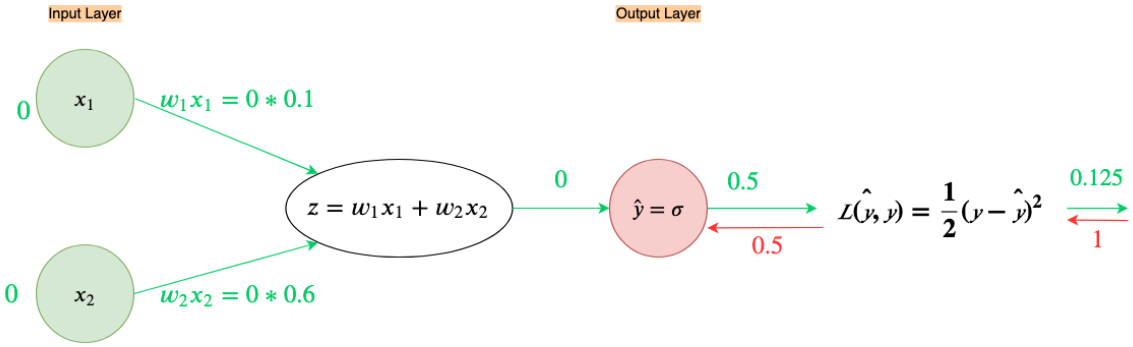
*This is a generalization of the **chain rule** from calculus.*

Since $\hat{y}$(predicted label) dictates our ***Loss*** and y(actual label) is constant, for a single example, *we will take the partial derivative of Loss with respect to* $\hat{y}$

$$\frac{\partial L}{\partial \hat{y}} = \frac{\partial(\frac{1}{2}(y-\hat{y})^2)}{\partial \hat{y}} = -(y-\hat{y})$$

Fig 16. The partial derivative of Loss w.r.t $\hat{y}$

Since the backpropagation steps can seem a bit complicated I'll go over them step by step:



The neural network is going through the following computations(backward computations are marked in red):

- The first backward computation, fot the most part, is redundant, but for the sake of completeness we'll define it.
- $\frac{\partial L}{\partial L} = 1$ *- this forms the first Upstream gradient*
- The *Local gradient* at $L(\hat{y}, y)) = \frac{1}{2}(y - \hat{y})^2$ is:

$$\frac{\partial L}{\partial \hat{y}} = -(y - \hat{y})$$

Recall for current example $\hat{y} = 0.5$ and $y = 0$. So, numercal value of local gradient is:

$$\frac{\partial L}{\partial \hat{y}} = -(0 - 0.5) = 0.5$$

- Finally, we can combine these and send back to the red node:

$$\frac{\partial L}{\partial \hat{y}} = UpstreamGradient * LocalGradient = \frac{\partial L}{\partial L} * \frac{\partial L}{\partial \hat{y}} = 1 * 0.5 = 0.5$$

Fig 17.a. Backpropagation

For the next calculation, we'll need the derivative of the sigmoid function, since it forms the local gradient of the red node. Let's derive that.

Following is the sigmoid function:

$$\hat{y} = \sigma(z) = \frac{1}{1 + e^{-z}}$$

Let $u = 1 + e^{-z}$

So, the equation becomes:

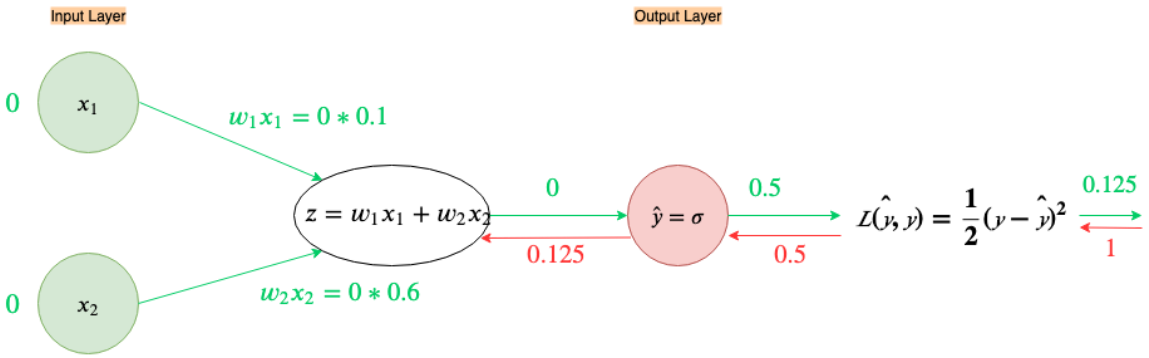
$$\hat{y} = \frac{1}{u}$$

Now, we can use the chain rule to easily derive the derivative:

$$
\begin{aligned}
\frac{d\hat{y}}{dz} &= \frac{d\hat{y}}{du} * \frac{du}{dz} \\
&= \left(-\frac{1}{u^2}\right) * (-e^{-z}) \\
&\text{substitute } u = 1 + e^{-z} \\
&= \left(-\frac{1}{(1+e^{-z})^2}\right) * (-e^{-z}) \\
&= \frac{e^{-z}}{(1+e^{-z})^2} \\
&= \frac{1}{1+e^{-z}} * \frac{e^{-z}}{(1+e^{-z})} \\
&= \frac{1}{1+e^{-z}} * \frac{1+e^{-z}-1}{(1+e^{-z})}, \text{ 1 added and subtracted, overall numerator remains same} \\
&= \frac{1}{1+e^{-z}} * \left(\frac{1+e^{-z}}{1+e^{-z}} - \frac{1}{1+e^{-z}}\right) \\
&= \frac{1}{1+e^{-z}} * \left(1 - \frac{1}{1+e^{-z}}\right) \\
&\text{substitute } \frac{1}{1+e^{-z}} = \hat{y} \\
&= \hat{y} * (1 - \hat{y})
\end{aligned}
$$

Fig18. The derivative of the Sigmoid function.

Let's use this in the next backward calculation



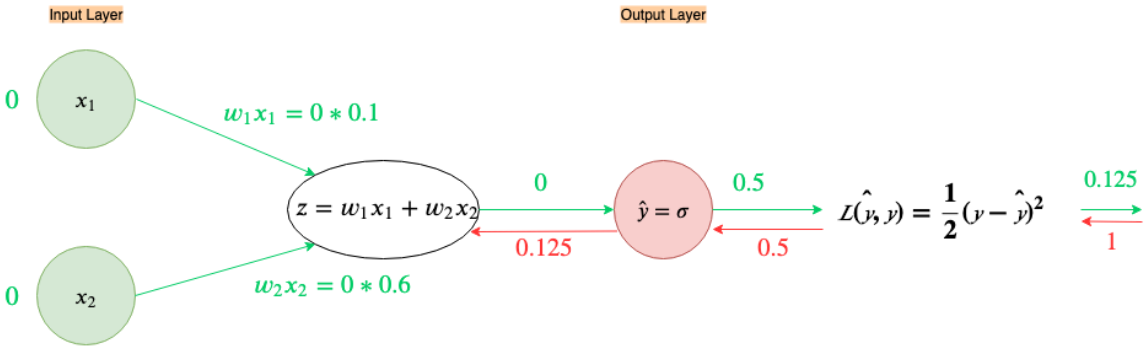
The neural network is going through the following computations(backward computations are marked in red):

- The *Upstream gradient* in this step is $\frac{\partial L}{\partial \hat{y}} = 0.5$
- The *Local gradient* at the red node is: $\frac{\partial \hat{y}}{\partial z} = \hat{y} * (1 - \hat{y}) = 0.5 * (1 - .05) = \frac{1}{4} = 0.25$
- Like previously, we will combine these and send them backwards to the white node:

$$\frac{\partial L}{\partial z} = UpstreamGradient * LocalGradient = \frac{\partial L}{\partial \hat{y}} * \frac{\partial \hat{y}}{\partial \hat{z}} = 0.5 * 0.25 = \frac{1}{8} = 0.125$$

Fig 17.b. Backpropagation

The backward computations should not propagate all the way to inputs as we don't want to change our input data(i.e. red arrows should not go to green nodes). We only want to change the weights associated with inputs.



The neural network is going through the following computations(backward computations are marked in red):

- Finally, we have now propagated the upstream gradient back enough to calculate the derivatives of weights w_1 and w_2.
- The *Upstream gradient* in this step is $\frac{\partial L}{\partial z} = 0.125$
- The *two Local gradients* are:

  1. $\frac{\partial z}{\partial w_1} = \frac{\partial (w_1x_1 + w_2x_2)}{\partial w_1} = x_1 = 0$
  2. $\frac{\partial z}{\partial w_2} = \frac{\partial (w_1x_1 + w_2x_2)}{\partial w_2} = x_2 = 0$

- We will again combine these, but this time not send them back to our input nodes, instead just figure out how much to change the weights w_1 and w_2:

  1. $\frac{\partial L}{\partial w_1} = UpstreamGradient * LocalGradient = \frac{\partial L}{\partial z} * \frac{\partial z}{\partial w_1} = 0.125 * 0 = 0$
  2. $\frac{\partial L}{\partial w_2} = UpstreamGradient * LocalGradient = \frac{\partial L}{\partial z} * \frac{\partial z}{\partial w_2} = 0.125 * 0 = 0$

Fig 17.c. Backpropagation

Notice something weird? *The derivatives to the Loss with respect to the weights,w_1 & w_2, are ZERO*! We can't increase or decrease the weights if their derivatives are zero. So then, how do we get our desired output in this instance if we can't figure out how to adjust the weights? *The key thing to note here is that the local gradients ($\partial z/\partial w_1$ and $\partial z/\partial w_2$) are x_1 and x_2,* both of which, in this example, happens to be zero (i.e. provide no information)

This brings us to the concept of ***bias.***

Bias

Recall equation of a line from your high school days.

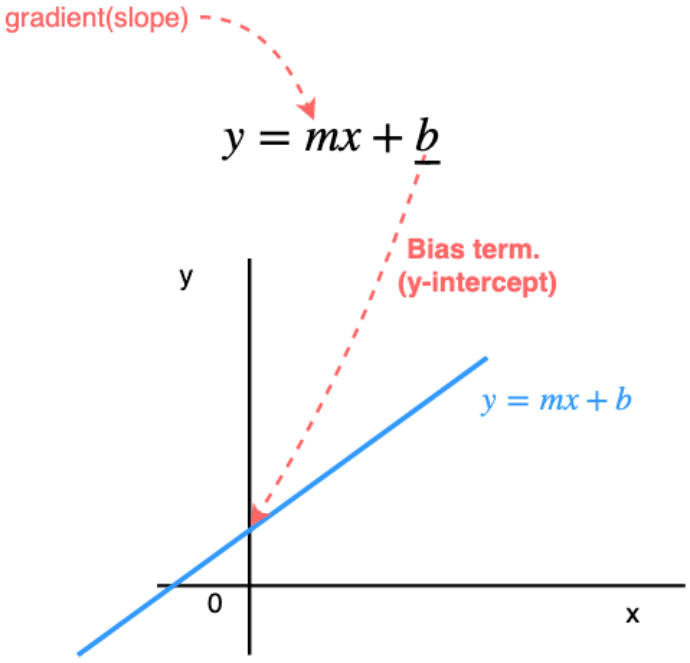


Fig 19. Equation of a Line

Here ***b*** is the bias term. Intuitively, the bias tells us that all outputs computed with ***x****(independent variable)* should have an additive bias of ***b.***So, when ***x=0***(no information coming from the *independent variable) the output should be biased to just* ***b.***

Note that without the bias term a line can only pass through the origin(0, 0) and the only differentiating factor between lines would then be the gradient ***m.***

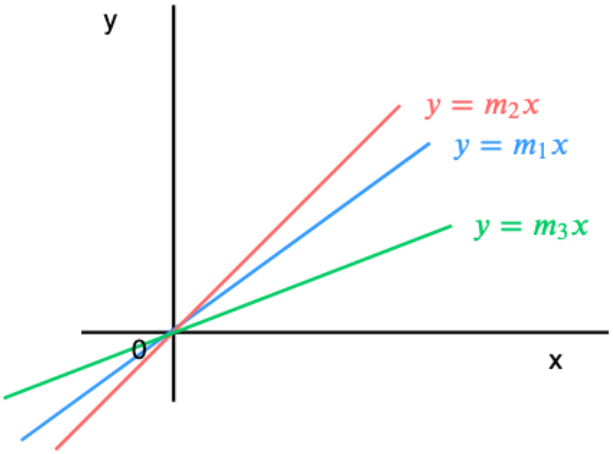


Fig 20. Lines from origin

So, using this new information let's add another node to a neural network; the bias node. (*In neural network literature, every layer, except the input layer, is assumed to have a bias node, just like the linear node, so this node is also often omitted in figures.*)

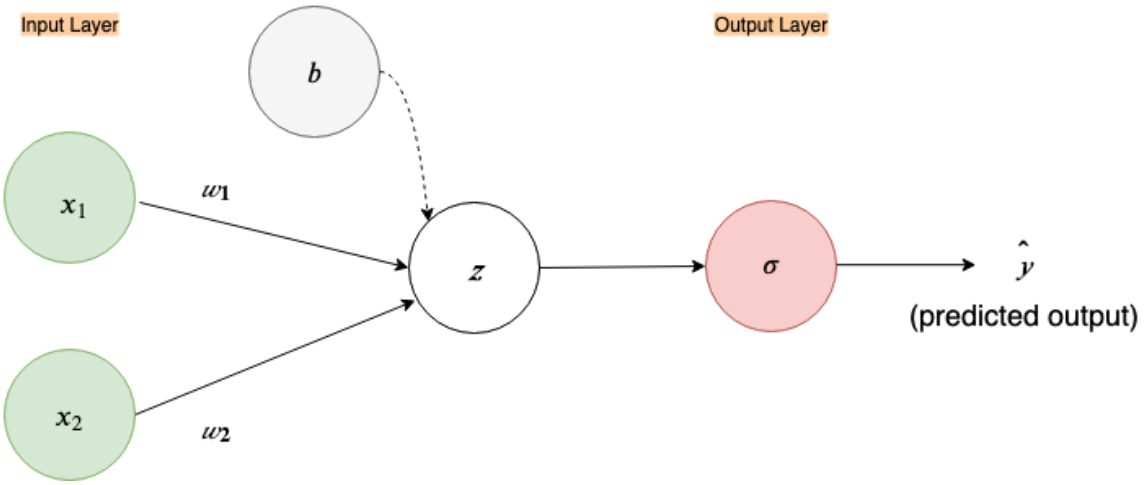


Fig 21. Expanded neural network with a bias node

Now let's do a forward propagation with the same example, $x_1=0$, $x_2=0$, $y=0$ and let's set bias, $b=0$ *(initial bias is always set to zero, rather than a random number)*, and let the backpropagation of Loss figure out the bias.

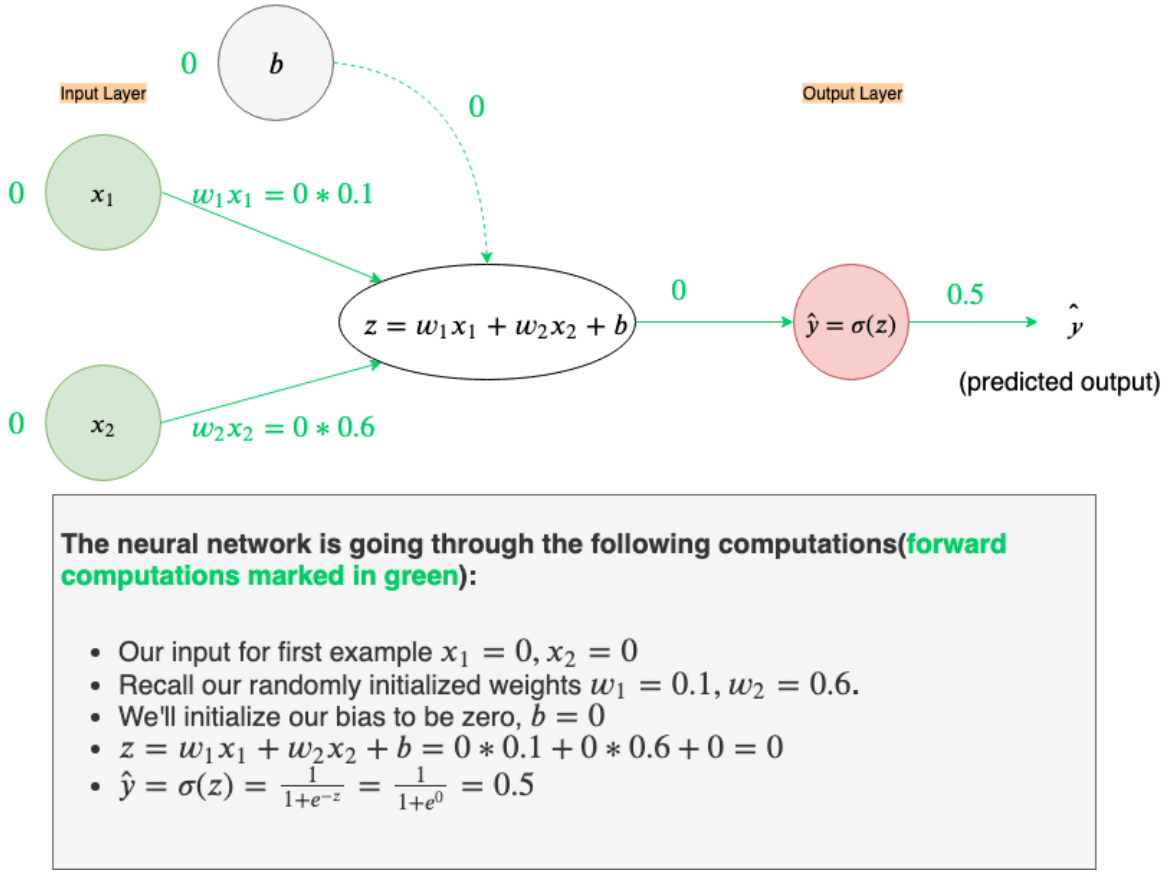
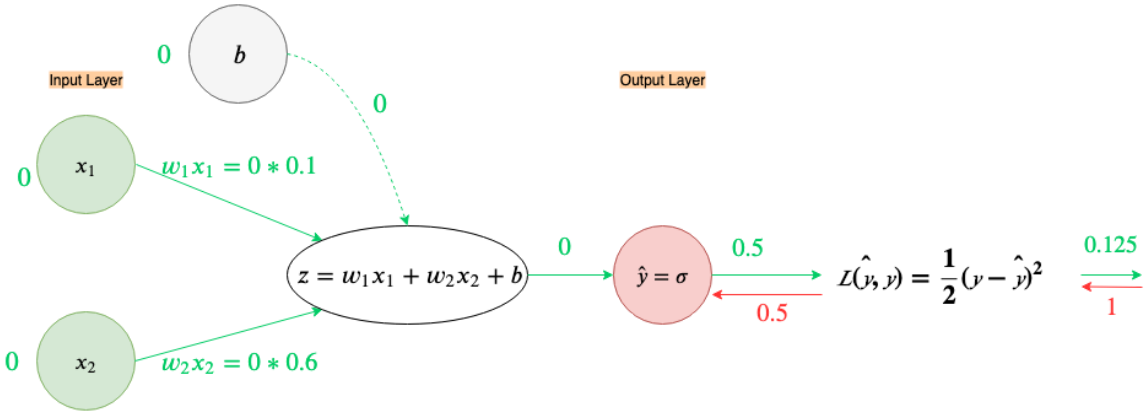


Fig 22. Forward propagation of the first example from OR table data with a bias unit

Well, the forward propagation with a bias of "***b=0***" didn't change our output at all, but let's do the backward propagation before we make our final judgment.

As before let's go through backpropagation in a step by step manner.



The neural network is going through the following computations(backward computations are marked in red):

- Again the first backward computation is redundant $\frac{\partial L}{\partial L} = 1$ *- this is still the first Upstream gradient*
- The *Local gradient* at $L(\hat{y}, y)) = \frac{1}{2}(y - \hat{y})^2$ still remains the same:

$$\frac{\partial L}{\partial \hat{y}} = -(y - \hat{y})$$

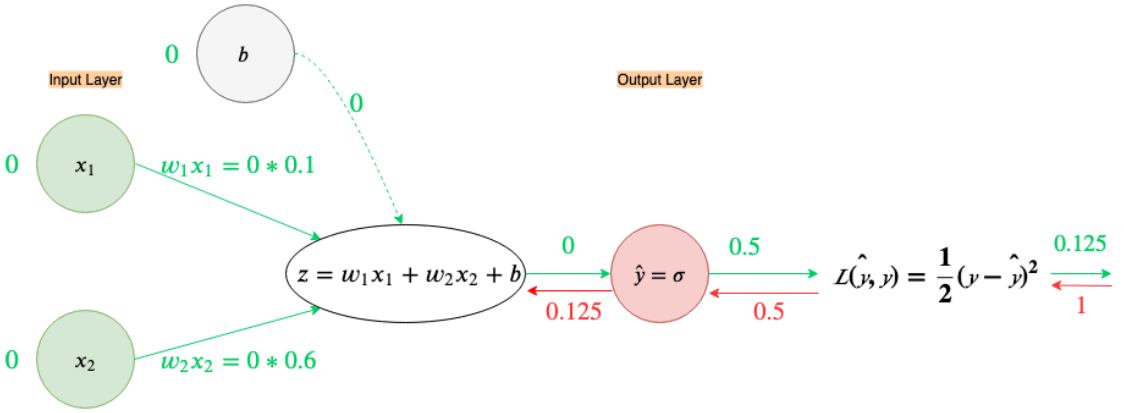
Recall, $\hat{y}$ for current example is $\hat{y} = 0.5$ and $y = 0$. So, numercal vale of local gradient also remains same:

$$\frac{\partial L}{\partial \hat{y}} = -(0 - 0.5) = 0.5$$

- As before, we'll combine these and send back to the red node:

$$\frac{\partial L}{\partial \hat{y}} = UpstreamGradient * LocalGradient = \frac{\partial L}{\partial L} * \frac{\partial L}{\partial \hat{y}} = 1 * 0.5 = 0.5$$

Fig 23.a. Backpropagation with bias

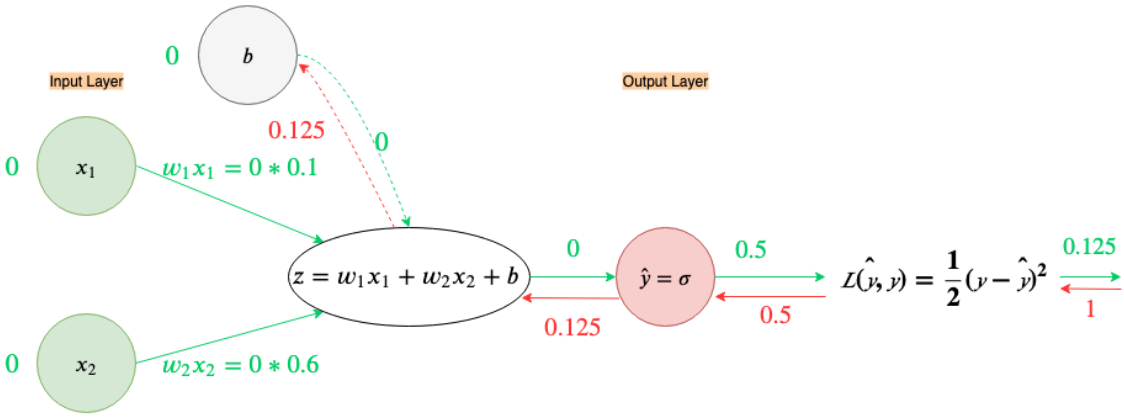


The neural network is going through the following computations(backward computations are marked in red):

- The computations in this step remain the same as before.
- The *Upstream gradient* in this step is $\frac{\partial L}{\partial \hat{y}} = 0.5$
- The *Local gradient* at the red node is: $\frac{\partial \hat{y}}{\partial z} = \hat{y} * (1 - \hat{y}) = 0.5 * (1 - .05) = \frac{1}{4} = 0.25$
- Like previously, we will combine these and send them backwards to the white node:

$$\frac{\partial L}{\partial z} = UpstreamGradient * LocalGradient = \frac{\partial L}{\partial \hat{y}} * \frac{\partial \hat{y}}{\partial \hat{z}} = 0.5 * (0.5 - (1 - 0.5)) = \frac{1}{8} = 0.125$$

Fig 23.b. Backpropagation with bias



The neural network is going through the following computations(backward computations are marked in red):

- FInally, we have now propogated the upstream gradient back enough to calcalute w_1 ,w_2 and our bias b
- The *Upstream gradient* in this step is $\frac{\partial L}{\partial z} = 0.125$
- The *three Local gratients* are:

1. $\frac{\partial z}{\partial w_1} = \frac{\partial (w_1x_1 + w_2x_2 + b)}{\partial w_1} = x_1 = 0$
2. $\frac{\partial z}{\partial w_2} = \frac{\partial (w_1x_1 + w_2x_2 + b)}{\partial w_2} = x_2 = 0$
3. $\frac{\partial z}{\partial b} = \frac{\partial (w_1x_1 + w_2x_2 + b)}{\partial b} = 1$

- We will again combine these, but this time not send them back to our input nodes, instead just figure out how much to change the weights w_1 , w_2 and b:

1. $\frac{\partial L}{\partial w_1} = UpstreamGradient * LocalGradient = \frac{\partial L}{\partial z} * \frac{\partial z}{\partial w_1} = 0.125 * 0 = 0$
2. $\frac{\partial L}{\partial w_2} = UpstreamGradient * LocalGradient = \frac{\partial L}{\partial z} * \frac{\partial z}{\partial w_2} = 0.125 * 0 = 0$
3. $\frac{\partial L}{\partial b} = UpstreamGradient * LocalGradient = \frac{\partial L}{\partial z} * \frac{\partial z}{\partial b} = 0.125 * 1 = 0.125$

Fig 23.c. Backpropagation with bias

Hurrah! we just figured out how much to adjust the bias. Since the derivative of bias(**∂L/∂b**) is positive 0.125, we will need to adjust the bias by moving in the negative direction of the gradient(recall the curve of the Loss function from before). This is technically called ***gradient descent***, as we are "descending" away from the sloping region to a flat region using the direction of the gradient. Let's do that.

To calculate new bias we do the following:

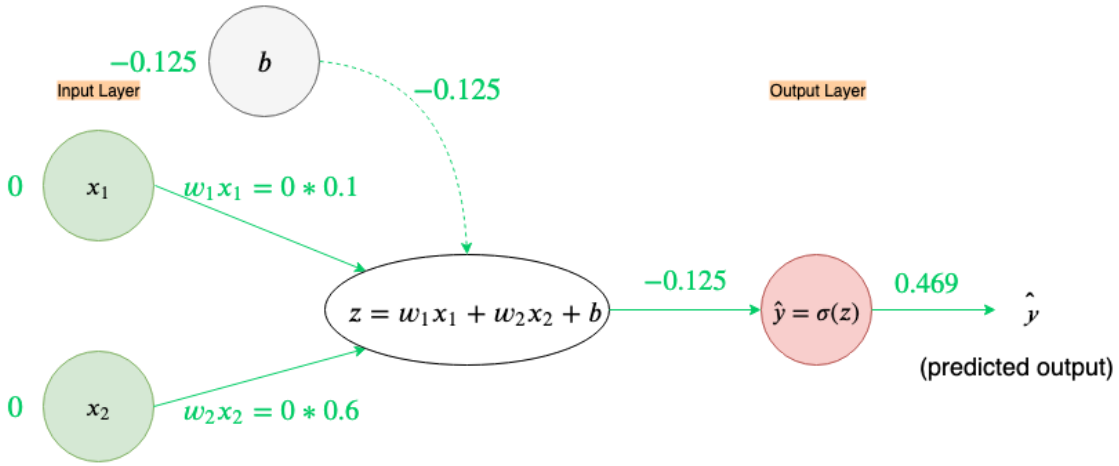
Recall, current bias, $b = 0$ and $\frac{\partial L}{\partial b} = 0.125$

The new bias is:

$$b = b - \frac{\partial L}{\partial b} = 0 - 0.125 = \mathbf{-0.125}$$

Fig 24. Calculated new bias using gradient descent

Now, that we've slightly adjusted the bias to ***b=-0.125,*** let's test if we've done the right thing by doing a **forward propagation** and **checking the new Loss.**



The neural network is going through the following computations(forward computations marked in green):

- Our input for first example $x_1 = 0, x_2 = 0$
- Randomly initialized weights and zero initialized bias $w_1 = 0.1, w_2 = 0.6, b = -0.125$
- $z = w_1x_1 + w_2x_2 + b = 0 * 0.1 + 0 * 0.6 + (-0.125) = -0.125$
- $\hat{y} = \sigma(z) = \frac{1}{1+e^{-z}} = \frac{1}{1+e^{0}} = 0.4687906... \approx \mathbf{0.469}$

Fig 25. Forward propagation with newly calculated bias

$$Loss = L(\hat{y}, y) = \frac{1}{2}(y - \hat{y})^2 = \frac{1}{2}(0 - 0.469)^2 = \frac{1}{2}(-0.469)^2 = \mathbf{0.10998005}$$

- For our current example $\hat{y} \approx \mathbf{0.469}$ and $y = \mathbf{0}$

Fig 26. Loss after newly calculated bias

Now our predicted output is ***ŷ≈0.469****(rounded to 3 decimal places)****,*** that's a slight improvement from the previous 0.5 and Loss is down from 0.125 to around ***0.109***. This slight correction is something that the neural network has 'learned' just by comparing its predicted output with the desired output, ***y***, and then moving in the direction opposite of the gradient**.** Pretty cool, right?

Now you may be wondering, this is only a small improvement from the previous result and how do we get to the minimum Loss. Two things come into play: ***a)*** **how many iterations of 'training' we perform** (each training cycle is forward propagation followed by backward propagation and updating the weights through gradient descent). ***b)*** **the learning rate.**

Learning rate??? What's that? Let's talk about it.

Learning Rate

Recall, how we calculated the new bias, above, by moving in the direction opposite of the gradient(i.e. ***gradient descent***).

$$b = b - \frac{\partial L}{\partial b}$$

Fig 27. The equation for updating bias

Notice that when we updated the bias we moved ***1 step in the opposite direction of the gradient.***

$$b = b - 1\frac{\partial L}{\partial b}$$

1 step

Fig 28. The equation for updating bias showing "step"

We could have moved 0.5, 0.9, 2, 3 or whatever fraction of steps we desired in the opposite direction of the gradient. This '*number of steps' is what we define as the* ***learning rate***, often denoted with ***α***(alpha)**.**

General Equation for Gradient Descent

$$w = w - \alpha\frac{\partial L}{\partial w}$$

Learning Rate

Fig 29. The general equation for gradient descent

Learning rate defines how quickly we reach the minimum loss. Let's visualize below what the learning rate is doing:

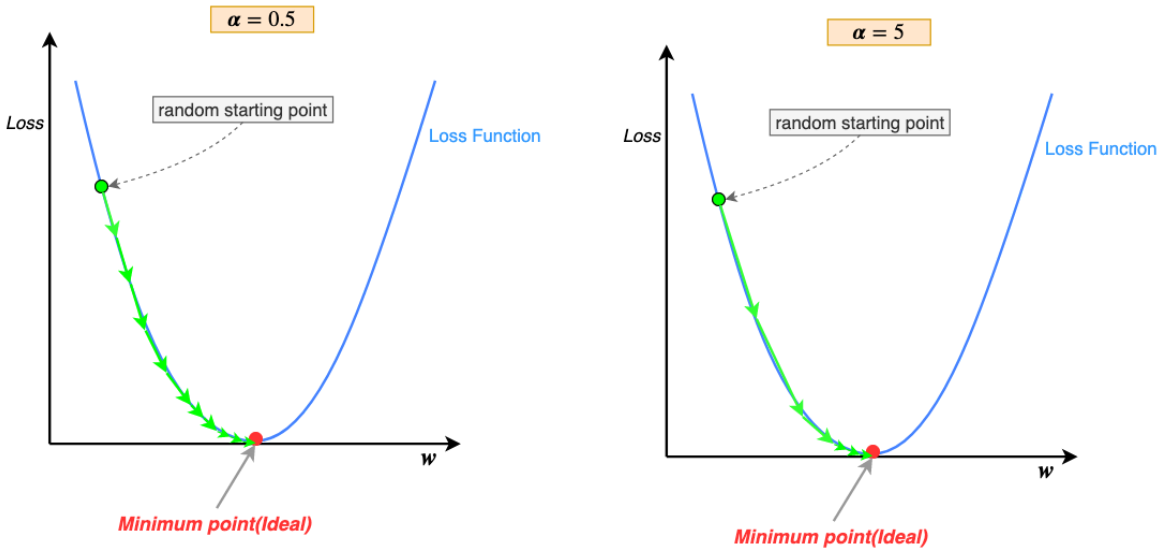


Fig 30. Visualizing the effect of learning rate.

As you can see with a lower learning rate(α=0.5) our descent along the curve is slower and we take many steps to reach the minimum point. On the other hand, with a higher learning rate(α=5) we take much bigger steps and reach the minimum point much faster.

The keen-eyed may have noticed that gradient descent steps(green arrows) keep getting smaller as we get closer and closer to the minimum, why is that? Recall, that the learning rate is being multiplied by the gradient at that point along the curve; as we descend away from sloping regions to flatter regions of the u-shaped curve, near the minimum point, the gradient keeps getting smaller and smaller, thus the steps also get smaller. Therefore, changing the

learning rate during training is not necessary(some variations of gradient descent start with a high learning rate to descend quickly down the slope and then reduce it gradually, this is called "annealing the learning rate")

So what's the takeaway? Just set the learning rate as high possible and reach the optimum loss quickly. NO. Learning rate can be a double-edged sword. Too high a learning rate and the parameters(weights/biases) don't reach the optimum instead start to diverge away from the optimum. To small a learning rate and the parameters take too long to converge to the optimum.

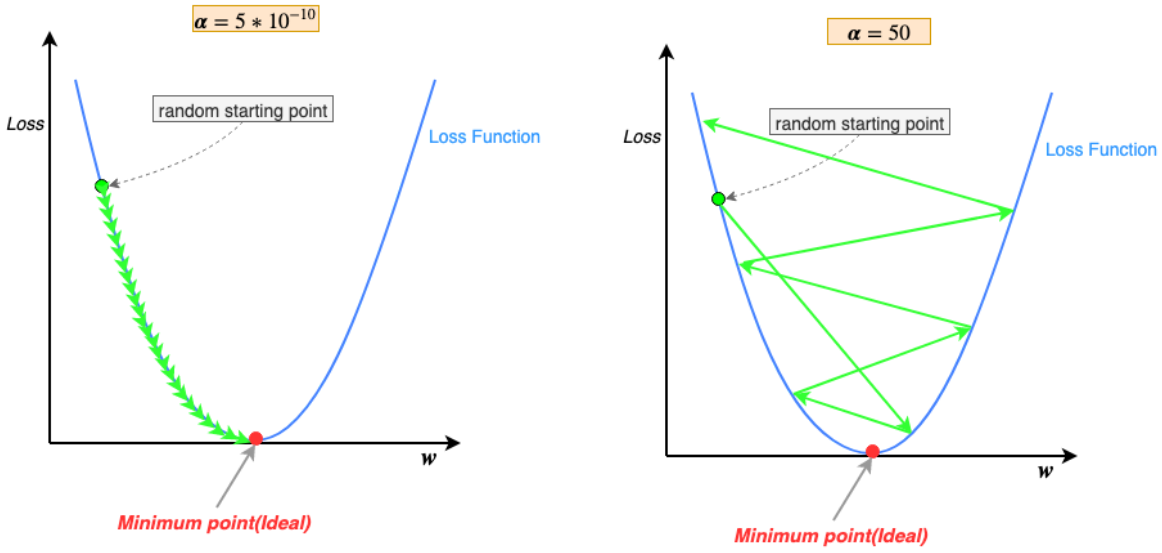


Fig 31. Visualizing the effect of very low vs. very high learning rate.

Small learning rate($\alpha=5*10^{-10}$) resulting is numerous steps to reach the minimum point is self-explanatory; multiply gradient with a small number(α) results in a proportionally small step.

Large learning rate($\alpha=50$) causing gradient descent to diverge may be confounding, but the answer is quite simple; note that at each step gradient descent approximates its path downward by moving in straight lines(green arrows in the figures), in short, it estimates its path downwards. When the learning rate is too high we force gradient descent to take larger steps. Larger steps tend to overestimate the path downwards and shoot past the minimum point, then to correct the bad estimate gradient descent tries to move towards the minimum point but again overshoots past the minimum due to the large learning rate. This cycle of continuous overestimates eventually cause the results to diverge(Loss after each training cycle increase, instead of decrease).

Learning rate is what's called a ***hyper-parameter***. Hyper-parameters are parameters that the neural network can't essentially learn through backpropagation of gradients, they have to be hand-tuned according to the problem and its dataset, by the creator of the neural network model. *(The choice of the Loss function, above, is also hyper-parameter)*

In short, the goal is not the find the "perfect learning rate " but instead a learning rate large enough so that the neural network trains successfully and efficiently without diverging.

So, far we've only used one example(***$x_1=0$*** and ***$x_2=0$***) to adjust our weights and bias(*actually, only our bias up till now*☹) and that reduced the loss on one example from our entire dataset(OR gate table). But we have more than one example to learn from and we want to reduce our loss across all of them. **Ideally, in one training iteration, we would like to reduce our loss across all the training examples**. This is called **Batch Gradient Descent**(or full batch gradient descent), as we use the entire batch of training examples per training iteration to improve our weights and biases. *(Others forms are **mini-batch gradient descent**, where we use a subset of the data set in each iteration and **stochastic gradient descent**, where we only use one example per training iteration as we've done so far).*

*A training iteration where the neural network goes through all the training examples is called an **Epoch**. If using mini-batches than an epoch would be complete after the neural network goes through all the mini-batches, similarly for stochastic gradient descent where a batch is just one example.*

Before we proceed further we need to define something called a ***Cost Function***.

Cost Function

When we perform "*batch gradient descent"* we need to slightly change our Loss function to accommodate not just one example but all the examples in the batch. This adjusted Loss function is called the **Cost Function**.

Also, note that the curve of the Cost Function is similar to the curve of the Loss function(same U-Shape).

Instead of calculating the Loss on one example the cost function calculates average Loss across ALL the examples.

$$
\begin{aligned}
Cost &= C(L(y^{(i)}, \hat{y}^{(i)})) \\
&= \frac{1}{m}\sum_{i=1}^{m} L(y^{(i)}, \hat{y}^{(i)}) \\
&= \frac{1}{m}\sum_{i=1}^{m} \frac{1}{2}(y^{(i)} - \hat{y}^{(i)})^2 \\
&= \frac{1}{2m}\sum_{i=1}^{m} (y^{(i)} - \hat{y}^{(i)})^2
\end{aligned}
$$

$i \rightarrow$ is the i^{th} training example
$m \rightarrow$ is the total number of training examples
$L(y^{(i)}, \hat{y}^{(i)}) \rightarrow$ is the loss in the i^{th} training example

Fig 32. Cost function

Intuitively, the Cost function is expanding out the capability of the Loss function. Recall, how the Loss function was helping to minimize the vertical distance between a *single* data point and the predictor line(z). **The Cost function is helping to minimize the vertical distance(Squared Error Loss) between multiple data points, concurrently.**

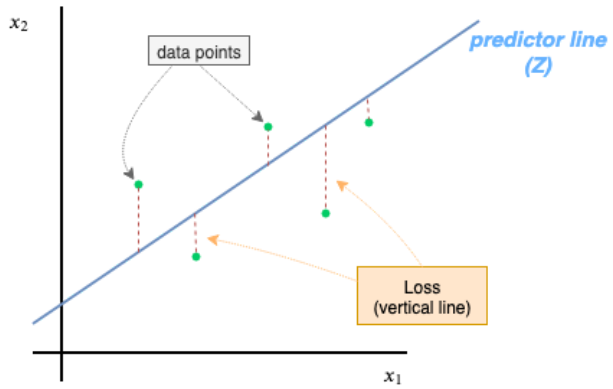


Fig 33. Visualization of the effect of the Cost function

During batch gradient descent we'll use the derivative of the Cost function, instead of the Loss function, to guide our path to minimum cost across all examples. *(In some neural network literature, the Cost Function is at times also represented with the letter **'J'**.)*

Let's take a look at how the derivative equation of the Cost function differs from the plain derivative of the Loss function.

The derivative of Cost Function

So the cost equation is :

$$Cost(\vec{\mathbf{y}}, \vec{\hat{\mathbf{y}}}) = \frac{1}{2m}\sum_{i=1}^{m} (y^{(i)} - \hat{y}^{(i)})^2$$

where $\vec{\mathbf{y}}$ and $\vec{\hat{\mathbf{y}}}$ represent input vectors

Fig 34. Cost function showing it takes input vectors

Taking the derivative of this Cost function, which takes vectors as inputs and sums them, can be a bit dicey. So, let's start out on a simple example before we generalize the derivative.

Let's say, for example, our vectors $\vec{\mathbf{y}}$ and $\vec{\hat{\mathbf{y}}}$ are:

$$\vec{\mathbf{y}} = \begin{bmatrix} y^{(1)} & y^{(2)} \end{bmatrix} \text{ and } \vec{\hat{\mathbf{y}}} = \begin{bmatrix} \hat{y}^{(1)} & \hat{y}^{(2)} \end{bmatrix}$$

where $y^{(i)}$ or $\hat{y}^{(i)}$ is the i^{th} example in the vector. The number of examples, m, here is 2.

Now, let's calculate the ***Cost.***

$$\begin{aligned} Cost(\vec{\mathbf{y}}, \vec{\hat{\mathbf{y}}}) &= \frac{1}{2m} \sum_{i=1}^{m} (y^{(i)} - \hat{y}^{(i)})^2 \\ &= \frac{1}{2m} \sum (\vec{\mathbf{y}} - \vec{\hat{\mathbf{y}}}) \\ &= \frac{1}{2m} \sum (\begin{bmatrix} y^{(1)} & y^{(2)} \end{bmatrix} - \begin{bmatrix} \hat{y}^{(1)} & \hat{y}^{(2)} \end{bmatrix})^{\circ 2} \\ &= \frac{1}{2m} \sum (\begin{bmatrix} (y^{(1)} - \hat{y}^{(1)}) & (y^{(2)} - \hat{y}^{(2)}) \end{bmatrix})^{\circ 2} \\ &= \frac{1}{2m} \sum (\begin{bmatrix} (y^{(1)} - \hat{y}^{(1)})^2 & (y^{(2)} - \hat{y}^{(2)})^2 \end{bmatrix}) \\ &= \frac{1}{2m} [(y^{(1)} - \hat{y}^{(1)})^2 + (y^{(2)} - \hat{y}^{(2)})^2] \end{aligned}$$

Element-wise square

Fig 35. Calculation of Cost on a simple vectorized example

Nothing new here in the calculation of the Cost. Just as expected the Cost, in the end, is the average of the Loss, but the implementation is now vectorized*(we performed vectorized subtraction followed by element-wise exponentiation, called Hadamard exponentiation)*. Let's derive the partial derivatives.

Vector $\vec{\hat{\mathbf{y}}}$ has two examples in it $\hat{y}^{(1)}$ and $\hat{y}^{(2)}$.
To take $\frac{\partial Cost}{\partial \vec{\hat{\mathbf{y}}}}$, we'll have to take two partial derivatives; one with respect to each example.

The result is a vector of partial derivatives, called Jacobian. *(Jacobian is just a fancy name for a vector/ matrix full of derivatives):*

$$\frac{\partial Cost}{\partial \vec{\hat{\mathbf{y}}}} = \begin{bmatrix} \frac{\partial Cost}{\partial \hat{y}^{(1)}} & \frac{\partial Cost}{\partial \hat{y}^{(2)}} \end{bmatrix}$$

Let's do this in two parts so that we can understand how these simple derivatives are being computed:

$$\frac{\partial Cost}{\partial} \quad \left(\frac{1}{} \right.$$

$$
\begin{aligned}
\frac{\partial Cost}{\partial \hat{y}^{(1)}} &= \frac{\partial}{\partial \hat{y}^{(1)}}\left(\frac{1}{2m}[(y^{(1)}-\hat{y}^{(1)})^2+(y^{(2)}-\hat{y}^{(2)})^2]\right) \\
&= \frac{1}{2m}\left[\frac{\partial}{\partial \hat{y}^{(1)}}\left((y^{(1)}-\hat{y}^{(1)})^2\right)+\frac{\partial}{\partial \hat{y}^{(1)}}\left((y^{(2)}-\hat{y}^{(2)})^2\right)\right] \\
&= \frac{1}{2m}\left[-2(y^{(1)}-\hat{y}^{(1)})+0\right] \\
&= -\frac{\mathbf{1}}{\mathbf{m}}(\mathbf{y^{(1)}}-\hat{\mathbf{y}}^{\mathbf{(1)}})
\end{aligned}
$$

$$
\begin{aligned}
\frac{\partial Cost}{\partial \hat{y}^{(2)}} &= \frac{\partial}{\partial \hat{y}^{(2)}}\left(\frac{1}{2m}[(y^{(1)}-\hat{y}^{(1)})^2+(y^{(2)}-\hat{y}^{(2)})^2]\right) \\
&= \frac{1}{2m}\left[\frac{\partial}{\partial \hat{y}^{(2)}}\left((y^{(1)}-\hat{y}^{(1)})^2\right)+\frac{\partial}{\partial \hat{y}^{(2)}}\left((y^{(2)}-\hat{y}^{(2)})^2\right)\right] \\
&= \frac{1}{2m}\left[0+\left(-2(y^{(2)}-\hat{y}^{(2)})\right)\right] \\
&= -\frac{\mathbf{1}}{\mathbf{m}}(\mathbf{y^{(2)}}-\hat{\mathbf{y}}^{\mathbf{(2)}})
\end{aligned}
$$

In the end, the Jacobian simply looks like this:

$$
\begin{aligned}
\frac{\partial Cost}{\partial \vec{\hat{\mathbf{y}}}} &= \begin{bmatrix} \frac{\partial Cost}{\partial \hat{y}^{(1)}} & \frac{\partial Cost}{\partial \hat{y}^{(2)}} \end{bmatrix} \\
&= \begin{bmatrix} -\frac{1}{m}\left(y^{(1)}-\hat{y}^{(1)}\right) & -\frac{1}{m}\left(y^{(2)}-\hat{y}^{(2)}\right) \end{bmatrix} \\
&= -\frac{\mathbf{1}}{m}\begin{bmatrix} \left(y^{(1)}-\hat{y}^{(1)}\right) & \left(y^{(2)}-\hat{y}^{(2)}\right) \end{bmatrix}
\end{aligned}
$$

Fig 36. Calculation of Jacobian on the simple example

From this, we can generalize the partial derivative equation.

Generalized derivative of the Cost(with squared error Loss) :

$$
\frac{\partial Cost}{\partial \hat{y}^{(i)}} = -\frac{\mathbf{1}}{m}\left(y^{(i)}-\hat{y}^{(i)}\right)
$$

Fig 37. Generalized partial derivative equation

Right now we should take a moment to note how the derivative of the Loss is different for the derivative of the Cost.

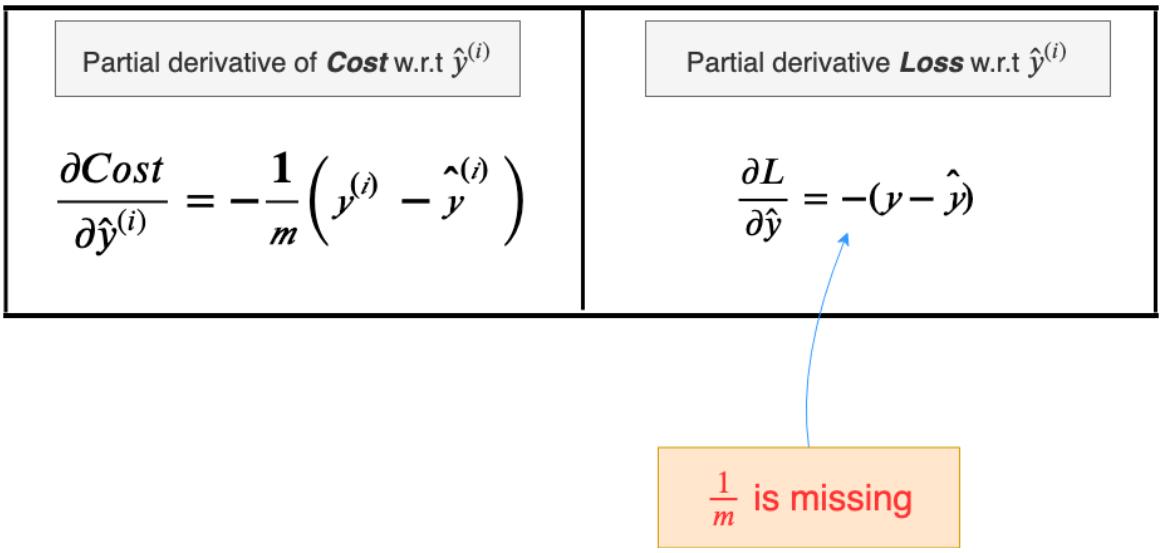


Fig 38. Comparison between the partial derivative of Loss and Cost with respect to(w.r.t) $\hat{\mathbf{y}}^{(i)}$

We'll later see how this small change manifests itself in the calculation of the gradient.

Back to batch gradient descent.

There are two ways to perform batch gradient descent:

1. For each training iteration create separate temporary variables(capital deltas, Δ) that will accumulate the gradients(small deltas, δ) for the weights and biases from each of the ***"m"*** examples in our training set, then at the end of the iteration update the weights using the average of the accumulated gradients. This is a slow method. *(for those familiar time complexity analysis you may notice that as the training data set grows this becomes a polynomial-time algorithm, $O(n^2)$)*

Batch Gradient Descent - Slow method

Start the training loop for an arbitrary number of iterations, let's say 500

loop 500 times:

$\Delta w_1 = 0, \Delta w_2 = 0, \Delta b = 0$ →Define temporary gradient accumulator variables for our weights and bias with capital delta(Δ) as prefix

$\Delta C = 0$ → Temporary variable to accumulate all the losses, so that we can caluclate Cost at the end

Now loop over all, "m" training examples

foreach training example:

Perform Forward-propagation

Claculate Loss(L) on example

$\Delta C = \Delta C + L$ →accumulate loss of example in ΔC,

Perform Backpropagation

$\Delta w_1 = \Delta w_1 + \delta w_1$ → accumulate gradient of $w1(\delta w_1)$

$\Delta w_2 = \Delta w_2 + \delta w_2$ →accumulate gradient of $w2(\delta w_2)$

$\Delta b = \Delta b + \delta b$→accumulate gradient of $b(\delta b)$

Calculate the Cost(which is just the average loss across all examples)

$Cost = \frac{1}{m}\Delta C$

finally, perform gradeint descent for each parameter, recall α is the *learning rate* and m is the *total number of training examples*

$w_1 = w_1 - \frac{\alpha}{m}\Delta w_1$

$w_2 = w_2 - \frac{\alpha}{m}\Delta w_2$

$b = b - \frac{\alpha}{m}\Delta b$

NOTE: diving by m gives us the average of the accumulated gradients in each case.

Fig 39. Batch Gradient Descent slow method

2. The quicker method is similar to above but instead uses vectorized computations to calculate all the gradients for all the training examples in one go, so the inner loop is removed. Vectorized computations run much quicker on computers. This is the method employed by all the popular neural network frameworks and the one we'll follow for the rest of this blog.

For vectorized computations, we'll make an adjustment to the "Z" node of the neural network computation graph and use the Cost function instead of the Loss function.

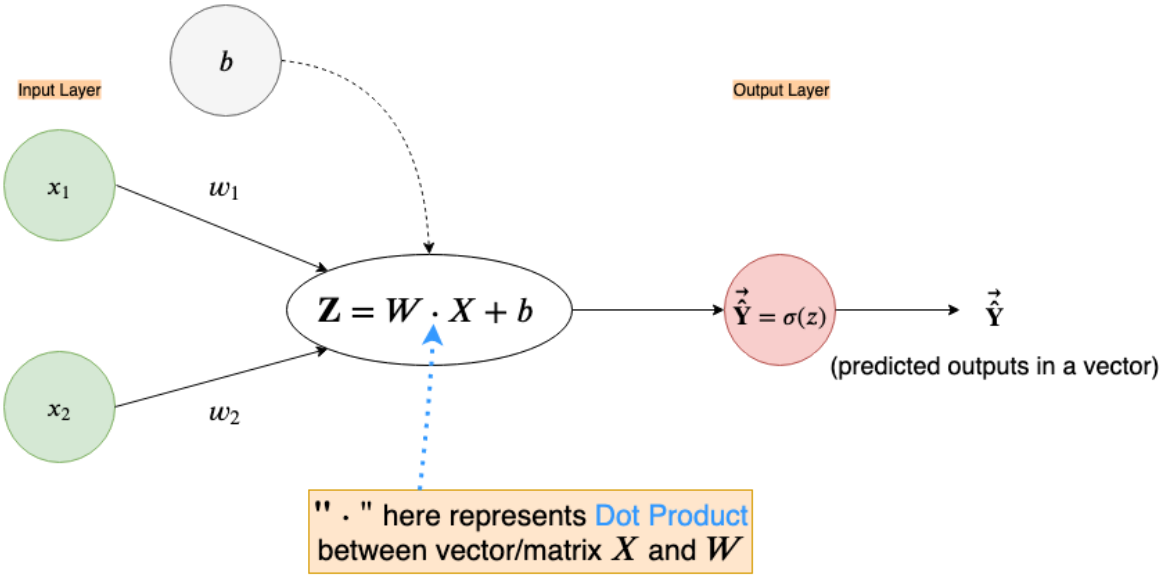


Fig 40. Vectorized implementation of Z node

Note that in the figure above we take **dot-product** between ***W*** and ***X*** which can be either an appropriate size matrix or vector. The bias, ***b***, is still a single number*(a scalar quantity)* here and will be added to the output of the dot product in an element-wise fashion. The predicted output will not be just a number, but instead a vector, $\hat{Y}$, where each element is the predicted output of their respective example.

Let's set up out data(***X, W, b & Y***) before doing forward and backward propagation.

Data Setup

$\mathbf{W} = \begin{bmatrix} w_1 & w_2 \end{bmatrix} = \begin{bmatrix} 0.1 & 0.6 \end{bmatrix}$, this makes $\mathbf{W}$ a (1×2) matrix

$$X = \begin{bmatrix} x_1^{(1)} & x_2^{(1)} \\ x_1^{(2)} & x_2^{(2)} \\ x_1^{(3)} & x_2^{(3)} \\ x_1^{(4)} & x_2^{(4)} \end{bmatrix} = \begin{bmatrix} 0 & 0 \\ 0 & 1 \\ 1 & 0 \\ 1 & 1 \end{bmatrix},$$

here each row represents an example with x_1 and x_2 as its features. **X** is a (4×2) matrix.

Data set up in this way where each row of the matrix represents an individual example is called a Design Matrix

$$\text{Similarly, } Y = \begin{bmatrix} y^{(1)} \\ y^{(2)} \\ y^{(3)} \\ y^{(4)} \end{bmatrix} = \begin{bmatrix} 0 \\ 1 \\ 1 \\ 1 \end{bmatrix},$$

each row represents the desired output for the respective example. **Y** is a (4×1) matrix

For dot-product we need to make sure that the matrices/vectors **W** and **X** are in the correct orientation/shape:

Matrix_A . Matrix_B = ResultMatrix_C

$(a \times b)$. $(b \times c)$ = $(a \times c)$

Need to match

Right now, **W** is (1×2) and **X** is (4×2). So, we would need to fix the

shape of our ***data*** (**X** and **Y**) to align with our computation of the dot-product.

So, we'll take the "transpose" of **X** and **Y**, which is simply flipping the matrix around its diagonal, so that rows become columns of the transposed matrix/vector.

$$X_{train} = X^T = \begin{bmatrix} x_1^{(1)} & x_1^{(2)} & x_1^{(3)} & x_1^{(4)} \\ x_2^{(1)} & x_2^{(2)} & x_2^{(3)} & x_2^{(4)} \end{bmatrix} = \begin{bmatrix} 0 & 0 & 1 & 1 \\ 0 & 1 & 0 & 1 \end{bmatrix},$$

So, the data we'll train with, X_{train}, now has the shape (2×4)

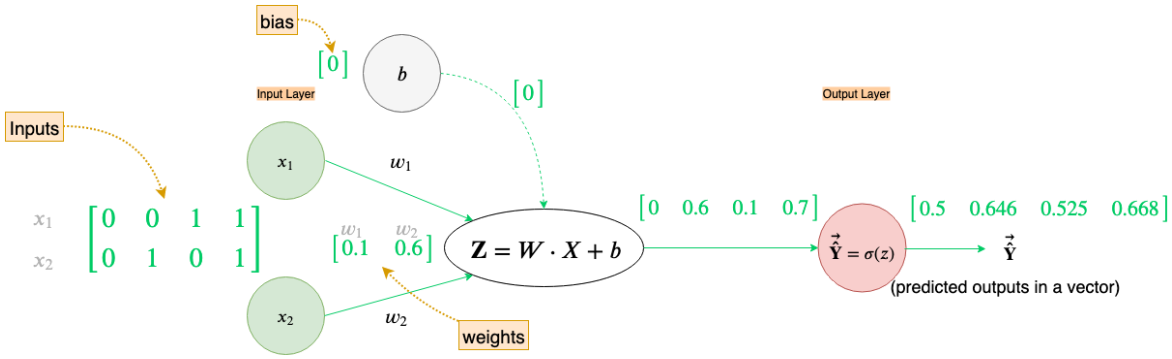
Similarly, $Y_{train} = Y^T = \begin{bmatrix} y^{(1)} & y^{(2)} & y^{(3)} & y^{(4)} \end{bmatrix} = \begin{bmatrix} 0 & 1 & 1 & 1 \end{bmatrix}$,
Y_{train}, now has the shape (1×4)

Bias, **b**, is simply, $b = \begin{bmatrix} b_1 \end{bmatrix} = \begin{bmatrix} 0 \end{bmatrix}$, a (1×1) matrix

Fig 41. Setup data for vectorized computations.

We are now finally ready to perform forward and backward propagation using $\mathbf{X}_{\text{train}}$, $\mathbf{Y}_{\text{train}}$, **W,** and **b**.

(NOTE: All the results below are rounded to 3 decimal points, just for brevity)



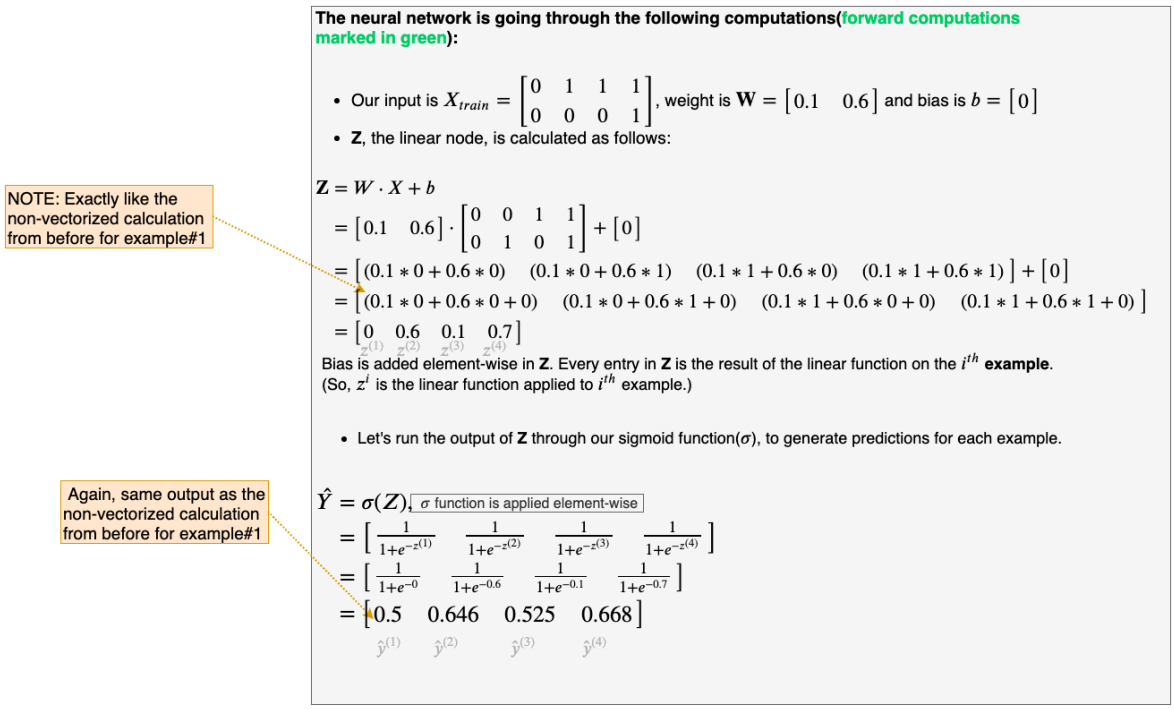


Fig 42. Vectorized Forward Propagation on OR gate dataset

How cool is that we calculated all the forward propagation steps for all the examples in our data set in one go, just by vectorizing our computations.

We can now calculate the **Cost** on these output predictions. *(We'll go over the calculation in detail, to make sure there is no confusion)*

$$Cost(\mathbf{Y}, \hat{\mathbf{Y}}) = \frac{1}{2m} \sum_{i=1}^{m} (y^{(i)} - \hat{y}^{(i)})^2$$

Element-wise square (Hadamard Exponentiation)

$$= \frac{1}{2m} \sum \left(Y - \hat{Y}\right)^{\circ 2}$$

$$= \frac{1}{2(4)} \sum \left(\begin{bmatrix} 0 & 1 & 1 & 1 \end{bmatrix} - \begin{bmatrix} 0.5 & 0.646 & 0.525 & 0.668 \end{bmatrix} \right)^{\circ 2}$$

Same calculation as Loss for example#1 from above

$$= \frac{1}{2(4)} \sum \begin{bmatrix} (0-0.5) & (1-0.646) & (1-0.525) & (1-0.668) \end{bmatrix}^{\circ 2}$$

$$= \frac{1}{8} \sum \begin{bmatrix} (0-0.5)^2 & (1-0.646)^2 & (1-0.525)^2 & (1-0.668)^2 \end{bmatrix}$$

$$= \frac{1}{8} \sum \begin{bmatrix} (-0.5)^2 & (0.354)^2 & (0.475)^2 & (0.332)^2 \end{bmatrix}$$

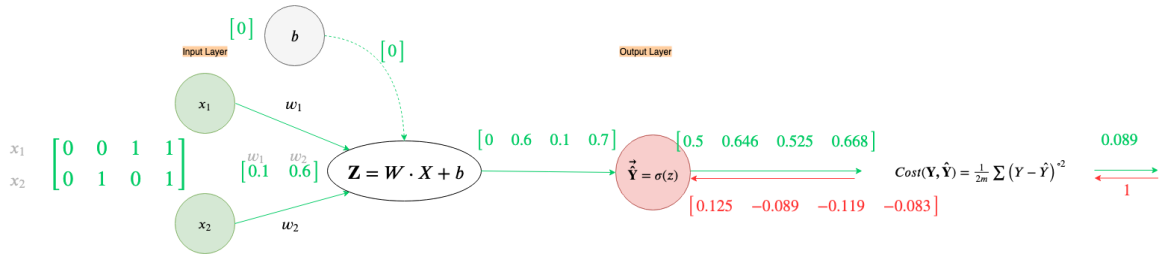
$$= \frac{1}{8} \left((-0.5)^2 + (0.354)^2 + (0.475)^2 + (0.332)^2 \right)$$

$$= \frac{1}{8} (0.711)$$

$$= \mathbf{0.089}$$

Fig 43. Calculation of Cost on the OR gate data

Our **Cost** with our current weights, **W**, turns out to be **0.089**. Our Goal now is to reduce this cost using backpropagation and gradient descent. As before we'll go through backpropagation in a step by step manner



The neural network is going through the following computations(backward computations are marked in red):

- Again, the first backward computation is redundant, $\frac{\partial Cost}{\partial Cost} = 1$ - *this is the first* ***Upstream Gradient***
- Recall the derivative of the **Cost Function** $Cost(\mathbf{Y}, \hat{\mathbf{Y}}) = \frac{1}{2m} \sum_{i=1}^{m} (y^{(i)} - \hat{y}^{(i)})^2$ we calculated above:

$$\frac{\partial Cost}{\partial \hat{y}^{(i)}} = -\frac{1}{m}\left(y^{(i)} - \hat{y}^{(i)}\right)$$

- We can calculate the ***Local Gradient*** in one go, by also vectorizing the $\frac{Cost}{\partial \hat{y}^{(i)}}$ computation as below:

$$\frac{\partial Cost}{\partial \hat{Y}} = -\frac{1}{m}(Y - \hat{Y})$$

Same as the Loss calculation for example# 1 above

$$= -\frac{1}{4}\left(\begin{bmatrix}0 & 1 & 1 & 1\end{bmatrix} - \begin{bmatrix}0.5 & 0.646 & 0.525 & 0.668\end{bmatrix}\right)$$

$$= -\frac{1}{4}\left(\begin{bmatrix}-0.5 & 0.354 & 0.475 & 0.332\end{bmatrix}\right)$$

$$= \begin{bmatrix}0.125 & -0.089 & -0.119 & -0.083\end{bmatrix}$$

- As before we'll combine the Local and the upstream gradient and send it back to the red node:

$$\frac{\partial Cost}{\partial \hat{Y}} = UpstreamGradient * LocalGradient$$

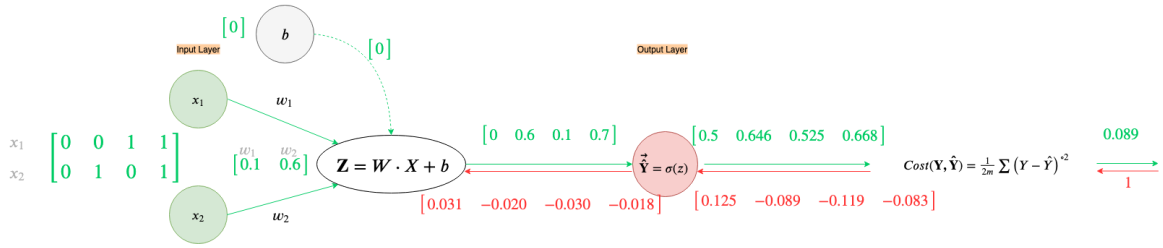
$$= \frac{\partial Cost}{\partial Cost} * \frac{\partial Cost}{\partial \hat{Y}}$$

$$= 1 * \begin{bmatrix}0.125 & -0.089 & -0.119 & -0.083\end{bmatrix}$$

$$= \begin{bmatrix}0.125 & -0.089 & -0.119 & -0.083\end{bmatrix}$$

$$\begin{bmatrix}\frac{\partial Cost}{\partial \hat{y}^{(1)}} & \frac{\partial Cost}{\partial \hat{y}^{(2)}} & \frac{\partial Cost}{\partial \hat{y}^{(3)}} & \frac{\partial Cost}{\partial \hat{y}^{(4)}}\end{bmatrix}$$

Fig 44.a. Vectorized Backward on OR gate data



The neural network is going through the following computations(backward computations are marked in red):

- Our ***Upstream Gradient*** in this step is :

$$\frac{\partial Cost}{\partial \hat{Y}} = \begin{bmatrix} 0.125 & -0.089 & -0.119 & -0.083 \end{bmatrix}$$
$$\begin{bmatrix} \frac{\partial Cost}{\partial \hat{y}^{(1)}} & \frac{\partial Cost}{\partial \hat{y}^{(2)}} & \frac{\partial Cost}{\partial \hat{y}^{(3)}} & \frac{\partial Cost}{\partial \hat{y}^{(4)}} \end{bmatrix}$$

- Recall the derivative of the sigmoid/logistic function(σ): $\frac{\partial \hat{y}}{\partial z} = \hat{y} - (1 - \hat{y})$
- We'll use a vectorized version of the derivative of the sigmoid function as our ***Local Gradient***:

Hadamard Product (element-wise multiplication)

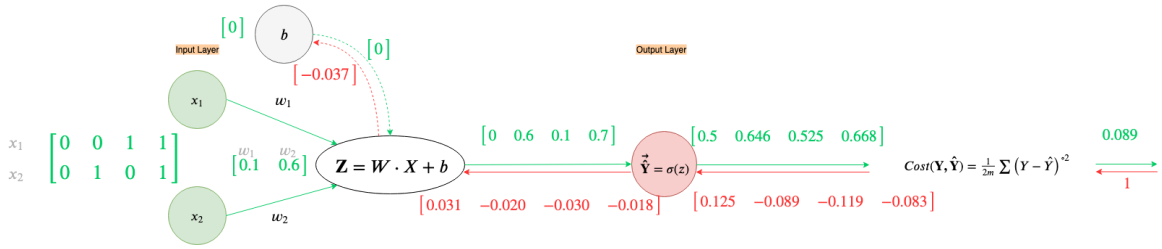
$$\begin{aligned} \frac{\partial \hat{Y}}{\partial Z} &= \hat{\mathbf{Y}}(1 - \hat{\mathbf{Y}}) \\ &= \begin{bmatrix} 0.5 & 0.646 & 0.525 & 0.668 \end{bmatrix} \odot \left(1 - \begin{bmatrix} 0.5 & 0.646 & 0.525 & 0.668 \end{bmatrix}\right) \\ &= \begin{bmatrix} 0.5 & 0.646 & 0.525 & 0.668 \end{bmatrix} \odot \begin{bmatrix} 0.5 & 0.354 & 0.475 & 0.332 \end{bmatrix} \\ &= \begin{bmatrix} (0.5 * 0.5) & (0.646 * 0.354) & (0.525 * 0.475) & (0.668 * 0.332) \end{bmatrix} \\ &= \begin{bmatrix} 0.25 & 0.229 & 0.249 & 0.222 \end{bmatrix} \end{aligned}$$

Same local gradient calculation as example#1, above

- We'll combine the upstream and local gradient and send them back to the white node(Z):

$$\begin{aligned} \frac{\partial Cost}{\partial Z} &= UpstreamGradient * LocalGradient \\ &= \frac{\partial Cost}{\partial \hat{Y}} * \frac{\partial \hat{Y}}{\partial Z} \\ &= \begin{bmatrix} 0.125 & -0.089 & -0.119 & -0.083 \end{bmatrix} \odot \begin{bmatrix} 0.25 & 0.229 & 0.249 & 0.222 \end{bmatrix} \\ &= \begin{bmatrix} (0.125 * 0.25) & (-0.089 * 0.229) & (-0.119 * 0.249) & (-0.083 * 0.222) \end{bmatrix} \\ &= \begin{bmatrix} 0.031 & -0.020 & -0.030 & -0.018 \end{bmatrix} \end{aligned}$$
$$\begin{bmatrix} \frac{\partial Cost}{\partial z^{(1)}} & \frac{\partial Cost}{\partial z^{(2)}} & \frac{\partial Cost}{\partial z^{(3)}} & \frac{\partial Cost}{\partial z^{(4)}} \end{bmatrix}$$

Fig 44.b. Vectorized Backward on OR gate data



The neural network is going through the following computations(backward computations are marked in red):

- We have propagated the upstream gradient back enough the calculate the gradient with respect to our weights W and bias b.
- Our ***Upstream Gradient*** in this step is :

$$\frac{\partial Cost}{\partial Z} = \begin{bmatrix} 0.031 & -0.020 & -0.030 & -0.018 \end{bmatrix}$$

$$\begin{bmatrix} \frac{\partial Cost}{\partial z^{(1)}} & \frac{\partial Cost}{\partial z^{(2)}} & \frac{\partial Cost}{\partial z^{(3)}} & \frac{\partial Cost}{\partial z^{(4)}} \end{bmatrix}$$

- This time our ***Z*** node is the vectorized implementation of a linear function: $Z = W \cdot X + b$, where W(weights) and X(data) are being dotted(dot product) with bias added to each element of the dot product.
- The **Local Gradients** of this vectorized function are:

1. $\frac{\partial Z}{\partial W} = X^T = X^T_{train} = \begin{bmatrix} 0 & 0 \\ 0 & 1 \\ 1 & 0 \\ 1 & 1 \end{bmatrix}$
2. $\frac{\partial Z}{\partial b} = 1$

*(Though we will not use this, as we don't want to change our input data, but the local gratient with respect to **X** is :*
$\frac{\partial Z}{\partial X} = W^T = \begin{bmatrix} 0.1 \\ 0.6 \end{bmatrix}$)

- We'll combine local and upstream gradients to figure out how much to change our weights and bias.

$$\begin{aligned}
\frac{\partial Cost}{\partial W} &= UpstreamGradient * LocalGradient \\
&= \frac{\partial Cost}{\partial Z} \cdot \frac{\partial Z}{\partial W} \\
&= \begin{bmatrix} 0.031 & -0.02 & -0.03 & -0.018 \end{bmatrix} \cdot \begin{bmatrix} 0 & 0 \\ 0 & 1 \\ 1 & 0 \\ 1 & 1 \end{bmatrix} \\
&= \begin{bmatrix} (0*0.031 + 0*-0.02 + 1*-0.03 + 1*-0.018) & (0*-0.031 + 1*-0.02 + 0*-0.03 + 1*-0.018) \end{bmatrix} \\
&= \begin{bmatrix} -0.048 & -0.038 \end{bmatrix} \\
&\quad \begin{bmatrix} \frac{\partial Cost}{\partial w_1} & \frac{\partial Cost}{\partial w_2} \end{bmatrix} \\
\frac{\partial Cost}{\partial b} &= \sum UpstreamGradient * LocalGradient \\
&= \sum \frac{\partial Cost}{\partial Z} * \frac{\partial Z}{\partial b} \\
&= \sum \begin{bmatrix} 0.031 & -0.02 & -0.03 & -0.018 \end{bmatrix} * 1 \\
&= \sum \begin{bmatrix} -0.031 & -0.02 & -0.03 & -0.018 \end{bmatrix} \\
&= [(0.031) + (-0.02) + (-0.03) + (-0.018)] \\
&= \mathbf{[-0.037]}
\end{aligned}$$

Fig 44.c. Vectorized Backward on OR gate data

Voila, we used a vectorized implementation of *batch gradient descent* to calculate all the gradients in one go.

(Those with a keen eye may be wondering how are the local gradients and the final gradients are being calculated in this last step. Don't worry, I'll explain the derivation of the gradients in this last step, shortly. For now, its suffice to say that the gradients defined in this last step are an optimization over the naive way of calculating ∂Cost/∂W and ∂Cost/∂b)

Let's update the weights and bias, keeping learning rate same as the non-vectorized implementation from before i.e. ***α=1.***

Gradient Descent Update

To calculate new weights(*W*) and bias(*b*) we move in the negative direction of the gradient

Recall, our current Weight vector is $W = \begin{bmatrix} 0.1 & 0.6 \end{bmatrix}$, $\alpha = 1$ and $\frac{\partial Cost}{\partial W} = \begin{bmatrix} -0.048 & -0.038 \end{bmatrix}$

The new Weights are:

$$\begin{aligned} W &= W - \alpha \frac{\partial Cost}{\partial W} \\ &= \begin{bmatrix} 0.1 & 0.6 \end{bmatrix} - \left(1 * \begin{bmatrix} -0.048 & -0.038 \end{bmatrix}\right) \\ &= \begin{bmatrix} 0.1 & 0.6 \end{bmatrix} - \begin{bmatrix} -0.048 & -0.038 \end{bmatrix} \\ &= \begin{bmatrix} 0.1 + 0.048 & 0.6 + 0.038 \end{bmatrix} \\ &= \begin{bmatrix} 0.148 & 0.638 \end{bmatrix} \end{aligned}$$

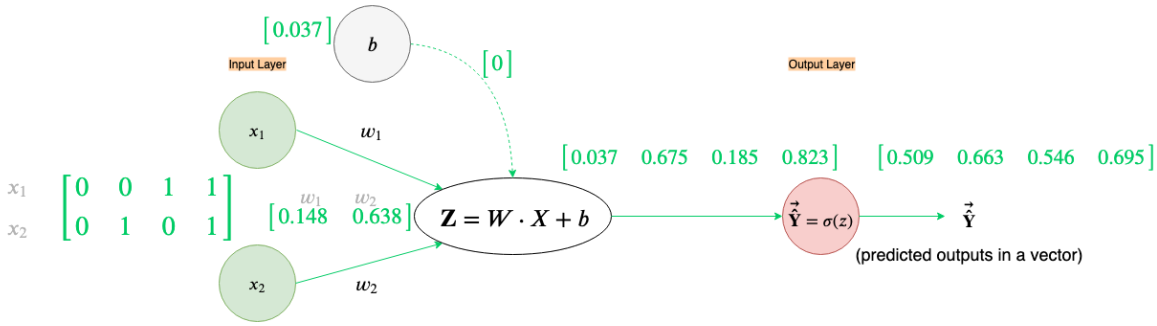
Our current Bias vector is $b = \begin{bmatrix} 0 \end{bmatrix}$, $\alpha = 1$ and $\frac{\partial Cost}{\partial b} = [-0.037]$

The new Bias is:

$$\begin{aligned} b &= b - \alpha \frac{\partial Cost}{\partial b} \\ &= \begin{bmatrix} 0 \end{bmatrix} - \left(1 * \begin{bmatrix} -0.037 \end{bmatrix}\right) \\ &= \begin{bmatrix} 0 \end{bmatrix} - \begin{bmatrix} -0.037 \end{bmatrix} \\ &= \begin{bmatrix} 0 + 0.037 \end{bmatrix} \\ &= \begin{bmatrix} 0.037 \end{bmatrix} \end{aligned}$$

Fig 45. Calculated new Weights and Bias

Now that we have updated the weights and bias lets do a **forward propagation** and **calculate the new Cost** to check if we've done the right thing.



The neural network is going through the following computations(forward computations marked in green):

- Our input is $X_{train} = \begin{bmatrix} 0 & 1 & 1 & 1 \\ 0 & 0 & 0 & 1 \end{bmatrix}$, weight is $\mathbf{W} = \begin{bmatrix} 0.148 & 0.638 \end{bmatrix}$ and bias is $b = [0.099]$

$$\mathbf{Z} = W \cdot X + b$$
$$= \begin{bmatrix} 0.148 & 0.638 \end{bmatrix} \cdot \begin{bmatrix} 0 & 0 & 1 & 1 \\ 0 & 1 & 0 & 1 \end{bmatrix} + \begin{bmatrix} 0.037 \end{bmatrix}$$
$$= \begin{bmatrix} (0.148*0+0.638*0) & (0.148*0+0.638*1) & (0.148*1+0.638*0) & (0.148*1+0.638*1) \end{bmatrix} + \begin{bmatrix} 0.037 \end{bmatrix}$$
$$= \begin{bmatrix} (0.148*0+0.638*0+0.037) & (0.148*0+0.638*1+0.037) & (0.148*1+0.638*1+0.037) & (0.148*1+0.638*1+0.037) \end{bmatrix}$$
$$= \begin{bmatrix} 0.037 & 0.675 & 0.185 & 0.823 \end{bmatrix}$$
$$\begin{bmatrix} z^{(1)} & z^{(2)} & z^{(3)} & z^{(4)} \end{bmatrix}$$

Bias is added element-wise in **Z**. Every entry in **Z** is the result of the linear function on the i^{th} **example**.
(So, $z^{(i)}$ is the linear function applied to i^{th} example.)

- Let's run the output of **Z** through our sigmoid function(σ), to generate predictions for each example.

$$\hat{Y} = \sigma(Z), \quad \sigma \text{ function is applied element-wise}$$
$$= \begin{bmatrix} \frac{1}{1+e^{-z^{(1)}}} & \frac{1}{1+e^{-z^{(2)}}} & \frac{1}{1+e^{-z^{(3)}}} & \frac{1}{1+e^{-z^{(4)}}} \end{bmatrix}$$
$$= \begin{bmatrix} \frac{1}{1+e^{-0.037}} & \frac{1}{1+e^{-0.675}} & \frac{1}{1+e^{-0.185}} & \frac{1}{1+e^{-0.823}} \end{bmatrix}$$
$$= \begin{bmatrix} 0.509 & 0.663 & 0.546 & 0.695 \end{bmatrix}$$

Fig 46. Vectorized Forward Propagation with updated weights and bias

$$\begin{aligned}
Cost(\mathbf{Y}, \hat{\mathbf{Y}}) &= \frac{1}{2m} \sum_{i=1}^{m} (y^{(i)} - \hat{y}^{(i)})^2 \\
&= \frac{1}{2m} \sum \left(Y - \hat{Y}\right)^{\circ 2} \\
&= \frac{1}{2(4)} \sum \left(\begin{bmatrix} 0 & 1 & 1 & 1 \end{bmatrix} - \begin{bmatrix} 0.509 & 0.663 & 0.546 & 0.695 \end{bmatrix} \right)^{\circ 2} \\
&= \frac{1}{2(4)} \sum \begin{bmatrix} (0-0.509) & (1-0.663) & (1-0.546) & (1-0.695) \end{bmatrix}^{\circ 2} \\
&= \frac{1}{8} \sum \begin{bmatrix} (0-0.509)^2 & (1-0.663)^2 & (1-0.546)^2 & (1-0.695)^2 \end{bmatrix} \\
&= \frac{1}{8} \sum \begin{bmatrix} (-0.509)^2 & (0.337)^2 & (0.454)^2 & (0.305)^2 \end{bmatrix} \\
&= \frac{1}{8} \left((-0.509)^2 + (0.337)^2 + (0.454)^2 + (0.305)^2 \right) \\
&= \frac{1}{8} (0.672) \\
&= \mathbf{0.084}
\end{aligned}$$

Fig 47. New Cost after updated parameters

So, we *reduced our Cost(Average Loss across all examples)* from an initial Cost of around ***0.089*** to **0.084**. We will need to do multiple training iterations before we can converge to a low Cost.

At this point, I would recommend that you perform backpropagation step yourself. The result of that should be (rounded to 3 decimal places): ***∂Cost/∂W = [-0.044, -0.035]*** and ***∂Cost/∂b = [-0.031].***

Recall, before we trained the neural network, how we predicted the neural network can separate the two classes in Figure 9, well after about 5000 Epochs(full batch training iterations) Cost steadily decreases to about ***0.0005*** and we get the following decision boundary ***:***

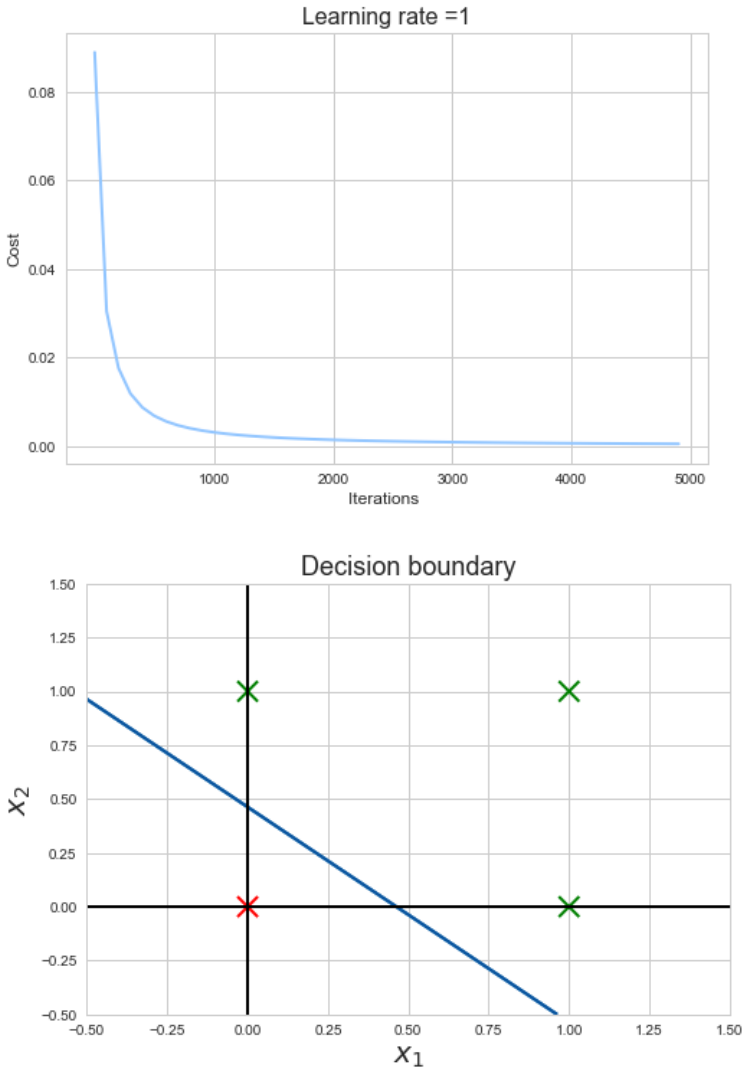


Fig 48. Cost curve and Decision boundary after 5000 epochs

The **Cost curve** is basically the value of Cost plotted after a certain number of iterations(epochs). Notice that the Cost curve flattens after about 3000 epochs this means that the weights and bias of the neural network have converged, so further training will only slightly improve our weights and bias. Why? Recall the u-shaped Loss curve, as we descend closer and closer the minimum point(flat region) the gradients become smaller and smaller thus the steps gradient descent takes are very small.

The **Decision Boundary** shows at the line along which the decision of the neural network changes from one output to the other. We can better visualize this by coloring the area below and above the decision boundary.

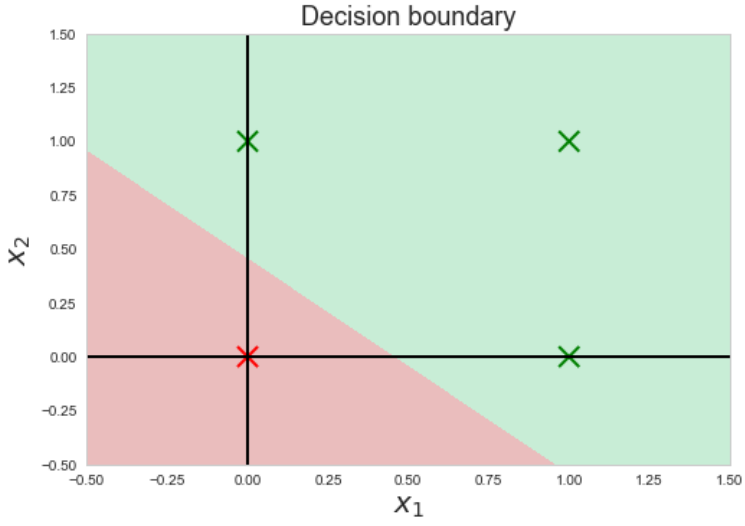


Fig 49. Decision boundary visualized after 5000 epochs

This makes it much clearer. The red shaded area is the area below the decision boundary and everything below the decision boundary has an output($\hat{y}$) of **0**. Similarly, everything above the decision boundary, shaded green, has an output of **1**. In conclusion, our simple neural network has learned a decision boundary by looking at the training data and figuring out how to separate its two output classes(**y=1** and **y=0**)🏠. Now the output neuron fires up🔥(produces 1) whenever x_1 or x_2 or both are 1.

Now would be a good time to see how the "**1/m**" ("**m**" is the total number of examples in the training dataset) in the Cost function manifested in the final calculation of the gradients.

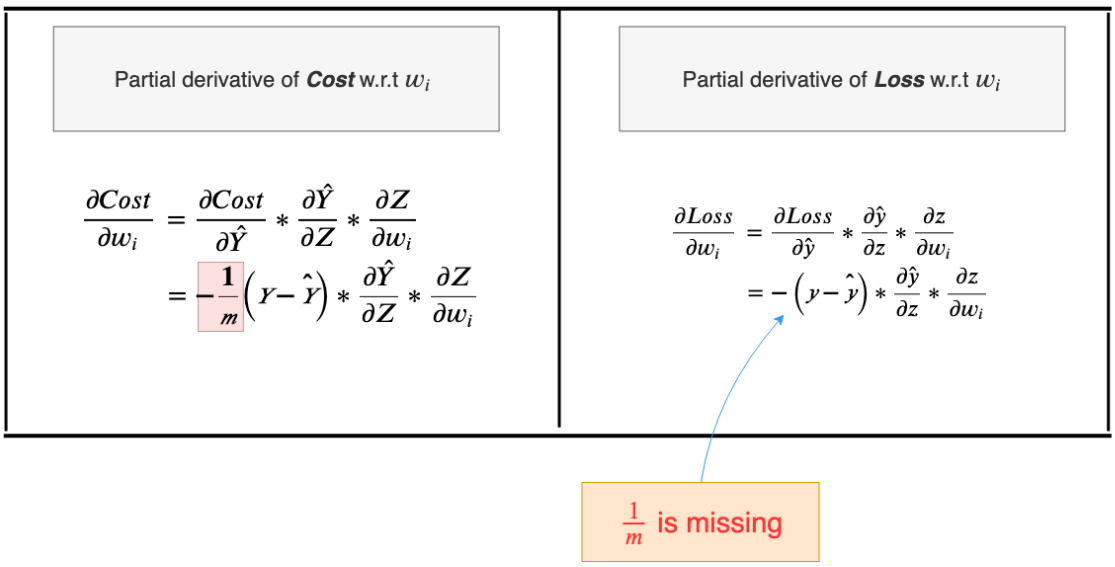


Fig 50. Comparing the effect of derivative w.r.t Cost and Loss on parameters of the neural network

From this, the most important point to know is that the gradient that is used to update our weights, using the Cost function, is the average of all the gradients calculated during a training iteration; same applies to bias. You may want to confirm this yourself by checking the vectorized calculations yourself.

Taking the average of all the gradients has some benefits. Firstly, it gives us a less noisy estimate of the gradient. Second, the resultant learning curve is smooth helping us easily determine if the neural network is learning or not. Both of these features come in very handy when training neural networks on much trickier datasets, such as those with wrongly labeled examples.

This is great and all but how did you calculate the gradients ∂Cost/∂W and ∂Cost/∂b ?

Neural network guides and blog posts I learned from often omitted complex details or gave very vague explanations for them. Not in this blog we'll go over everything leaving no stone unturned.

First, we'll tackle ∂Cost/∂b. Why did we sum the gradients?

To explain this I employ our computational graph technique on three very simple equations.

Let's define three equations:

$$e = c * d,$$
$$c = a * b,$$
$$d = b + 5$$

where $a = 2$ and $b = 5$

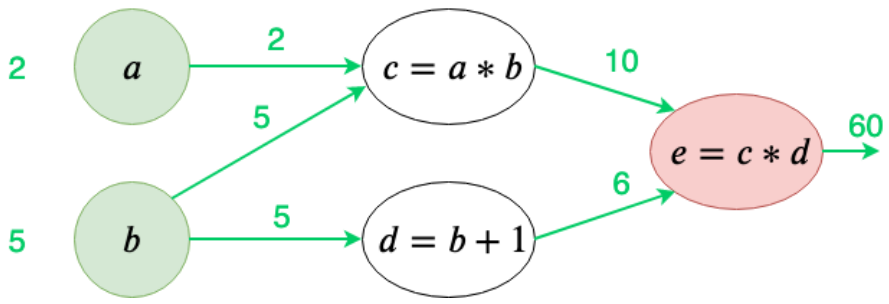


Fig 51. Computational graph of simple equations

I am particularly interested in the ***b*** node, so let's do backpropagation on this.

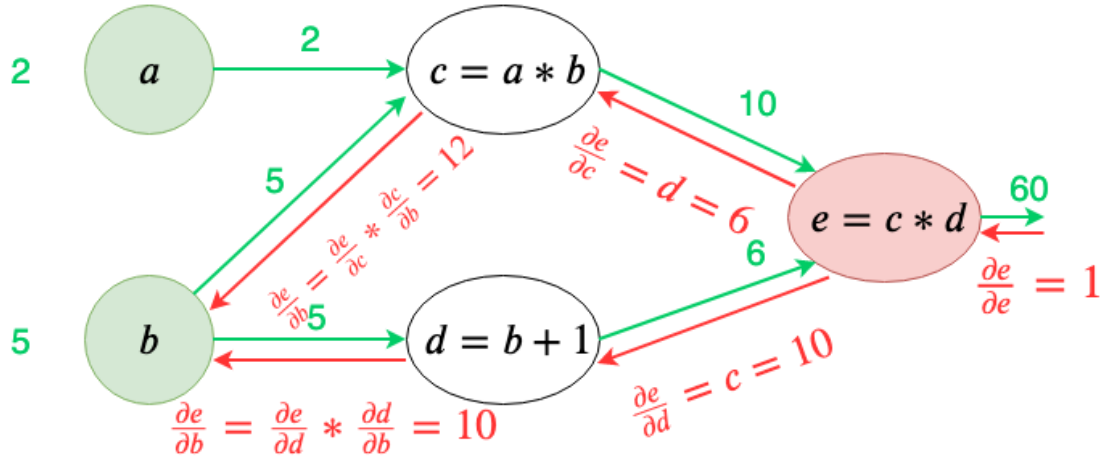


Fig 52. Backpropagation on the computational graph of simple equations

Note that the ***b*** node is receiving gradients from **two** other nodes. So the total of the gradients flowing into node ***b*** is the *sum* of the two gradients flowing in.

$$
\begin{aligned}
\frac{\partial e}{\partial b} &= \left(\frac{\partial e}{\partial d} * \frac{\partial d}{\partial b}\right) + \left(\frac{\partial e}{\partial c} * \frac{\partial c}{\partial b}\right) \\
&= (c * 1) + (d * a) \\
&= (10 * 1) + (6 * 2) \\
&= 10 + 12 \\
&= \mathbf{22}
\end{aligned}
$$

Fig 53. Sum of gradients flowing into node **b**

From this example, we can generalize the following rule: ***Sum all the incoming gradients to a node, from all the possible paths.***

Let's visualize how this rule is used in the calculation of the **bias**. *Our neural network can be seen as doing* ***independent*** *calculations for each of our examples* but using shared parameters for weights and bias, during a training iteration. Below bias(***b***) is visualized as a shared parameter for all individual calculations our neural network performs.

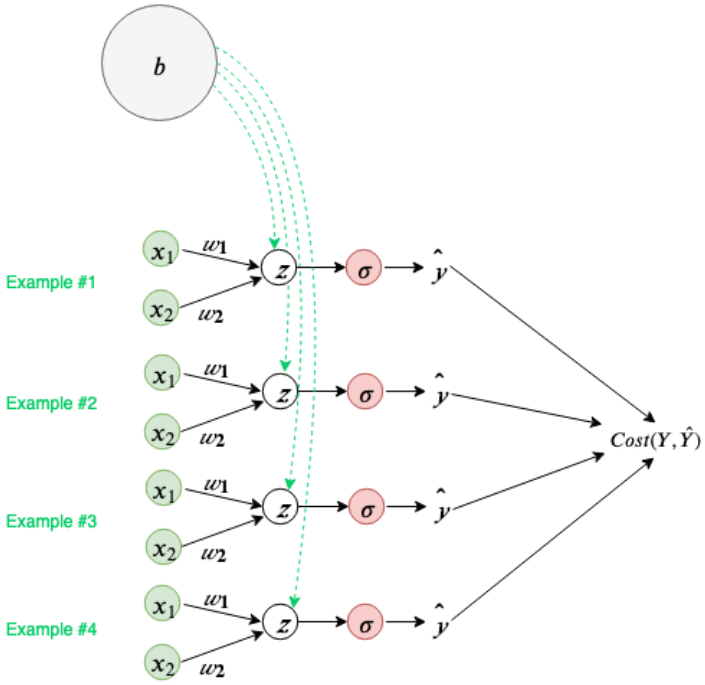


Fig 54. Visualizing bias parameter being shared across a training epoch.

Following the general rule defined above, we will sum all the incoming gradients from all the possible paths to the bias node, **b**.

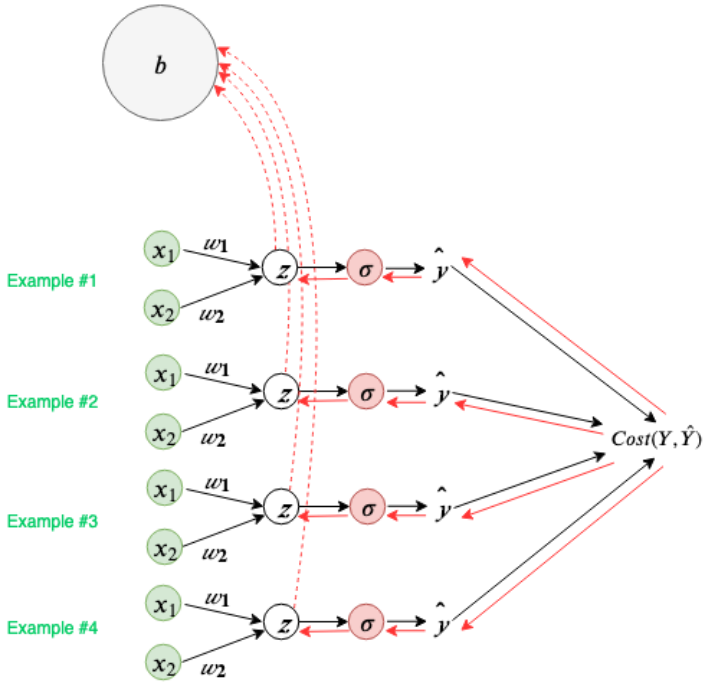


Fig 55. Visualizing all possible backpropagation paths to shared bias parameter

Since the ∂Z/∂b (local gradient at the Z node) is equal to **1**, the total gradient at ***b*** is the sum of gradients from each example with respect to the Cost.

So the total of the gradient flowing into node ***b***(bias) is:

$$\begin{aligned}
\frac{\partial Cost}{\partial b} &= \sum_{i=1}^{m} \left(\text{gradient from } i^{th} \text{example}\right) \\
&= \left(\text{gradient from } 1^{st} \text{example}\right) + \left(\text{gradient from } 2^{nd} \text{example}\right) + \left(\text{gradient from } 3^{rd} \text{example}\right) + \left(\text{gradient from } 4^{th} \text{example}\right) \\
&= \left(\frac{\partial Cost}{\partial Z^{(1)}} * \frac{\partial Z^{(1)}}{\partial b}\right) + \left(\frac{\partial Cost}{\partial Z^{(2)}} * \frac{\partial Z^{(2)}}{\partial b}\right) + \left(\frac{\partial Cost}{\partial Z^{(3)}} * \frac{\partial Z^{(3)}}{\partial b}\right) + \left(\frac{\partial Cost}{\partial Z^{(4)}} * \frac{\partial Z^{(4)}}{\partial b}\right) \\
&= \left(\frac{\partial Cost}{\partial Z^{(1)}} * [1]\right) + \left(\frac{\partial Cost}{\partial Z^{(2)}} * [1]\right) + \left(\frac{\partial Cost}{\partial Z^{(3)}} * [1]\right) + \left(\frac{\partial Cost}{\partial Z^{(4)}} * [1]\right) \\
&= \left[\frac{\partial Cost}{\partial Z^{(1)}} + \frac{\partial Cost}{\partial Z^{(2)}} + \frac{\partial Cost}{\partial Z^{(3)}} + \frac{\partial Cost}{\partial Z^{(4)}}\right]
\end{aligned}$$

Which is the same as taking the sum of the Upstream Gradient from ***Fig 44.c***

Fig 56. Proof that ∂Cost/∂b is the sum of upstream gradients

Now that we've got derivative of the bias figured out let's move on to derivative of weights, and more importantly the local gradient with respect to weights.

How is the local gradient(∂Z/∂W) equal to transpose of the input training data(X_train)?

This can be answered in a similar way to the above calculation for bias, but the main complication here is the calculating the derivative of the dot product between the weight matrix(***W***) and the data matrix(X_{train}), which forms our local gradient.

Recall the equation of the vectorized linear function:

$$Z = W \cdot X + b$$

We need to take the derivative of $\boldsymbol{W}$ with respect to $\mathbf{Z}$. Recall in our OR gate example $\boldsymbol{W}$ is a (1×2) matrix, $\boldsymbol{X}$ is a (2×4) matrix and the resultant $\boldsymbol{Z}$ is a (1×4) matrix.

$$\begin{aligned}
\frac{\partial Z}{\partial W} &= \frac{\partial(W \cdot X + b)}{\partial W} \\
&= \frac{\partial(W \cdot X)}{\partial W} + \frac{\partial(b)}{\partial W} \\
&= \frac{\partial(W \cdot X)}{\partial W} + 0 \\
&= \frac{\mathbf{\partial}(W \cdot X)}{\mathbf{\partial} W}
\end{aligned}$$

So, what is $\frac{\mathbf{\partial}(W \cdot X)}{\mathbf{\partial} W}$?

Fig 57.a. Figuring out the derivative of the dot product.

This derivative of the dot product is a bit complicated as **we are no longer working with scalar quantities,** instead, both $\boldsymbol{W}$ and $\boldsymbol{X}$ are matrices and the result of $\boldsymbol{W} \cdot \boldsymbol{X}$ is also a matrix. Let's dive a bit deeper using a simple example first and then generalizing from it.

Let,

$$A = W \cdot X$$

Where $W = \begin{bmatrix} w_1 & w_2 \end{bmatrix}$, and

$$X = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}$$

(Note: In ***X*** the features, x_1 and x_2, are arranged in column-wise fashion just is in X_{train})

Now, we can calculate our dot product:

$$\begin{aligned} A &= W \cdot X \\ &= \begin{bmatrix} w_1 & w_2 \end{bmatrix} \cdot \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} \\ &= \begin{bmatrix} w_1 x_1 + w_2 x_2 \end{bmatrix} \\ &= \begin{bmatrix} a_1 \end{bmatrix} \\ &\text{where } a_1 = w_1 x_1 + w_2 x_2 \end{aligned}$$

Fig 57.b. Figuring out the derivative of the dot product.

Let's calculate the derivative of the ***A*** with respect to ***W***.

$$\begin{aligned} \frac{\partial A}{\partial W} &= \begin{bmatrix} \frac{a_1}{\partial W} \end{bmatrix} \\ &= \begin{bmatrix} \frac{a_1}{\partial w_1} & \frac{a_1}{\partial w_2} \end{bmatrix} \end{aligned}$$

Note: that W is a (1×2) matrix, so the derivative also results in a (1×2) matrix.

$$\begin{aligned} &= \begin{bmatrix} \frac{\partial(w_1 x_1 + w_2 x_2)}{\partial w_1} & \frac{\partial(w_1 x_1 + w_2 x_2)}{\partial w_2} \end{bmatrix} \\ &= \begin{bmatrix} x_1 & x_2 \end{bmatrix} \end{aligned}$$

From this derivative note that we input the examples in vector form, $\begin{bmatrix} x_1 \\ x_2 \end{bmatrix}$, and the derivative of dot product w.r.t ***W*** results in row vectors of the input, $\begin{bmatrix} x_1 & x_2 \end{bmatrix}$.

In general, the derivative of the dot product with respect to W results in the transpose of X

Fig 57.c. Figuring out the derivative of the dot product.

Let us visualize this in case of a training iteration where multiple examples are being processed at the same time. *(Note that input examples are column vectors.)*

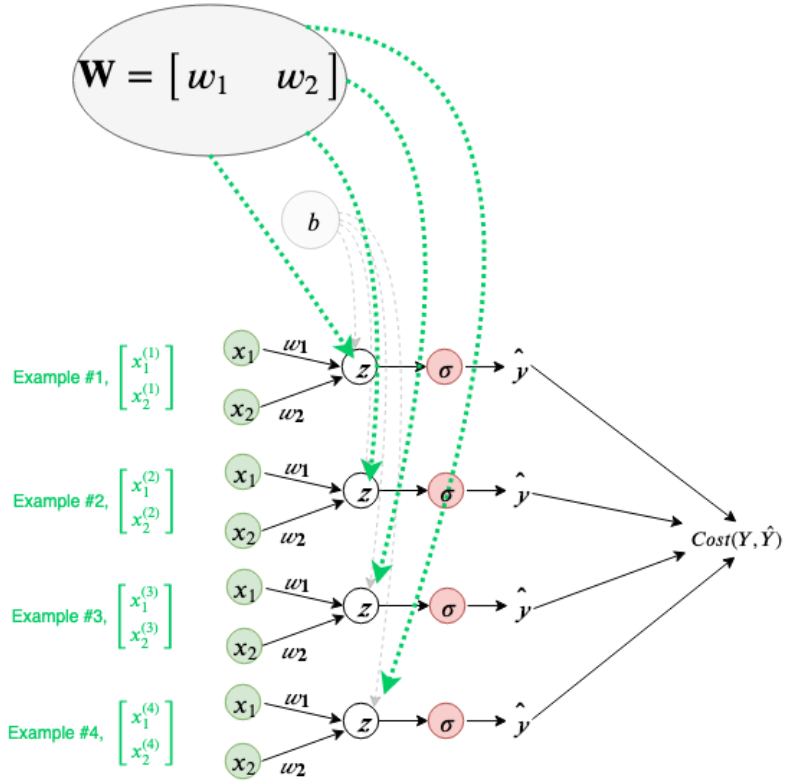


Fig 58. Visualizing weights being shared across a training epoch

Just as the bias (**b**) was being shared across each calculation in a training iteration, weights (***W***) are also being shared. We can also visualize the gradient flowing back to the weights, as follows*(note that the local derivative of each example w.r.t to **W** results in a row vector of the input example i.e. transpose of input)*:

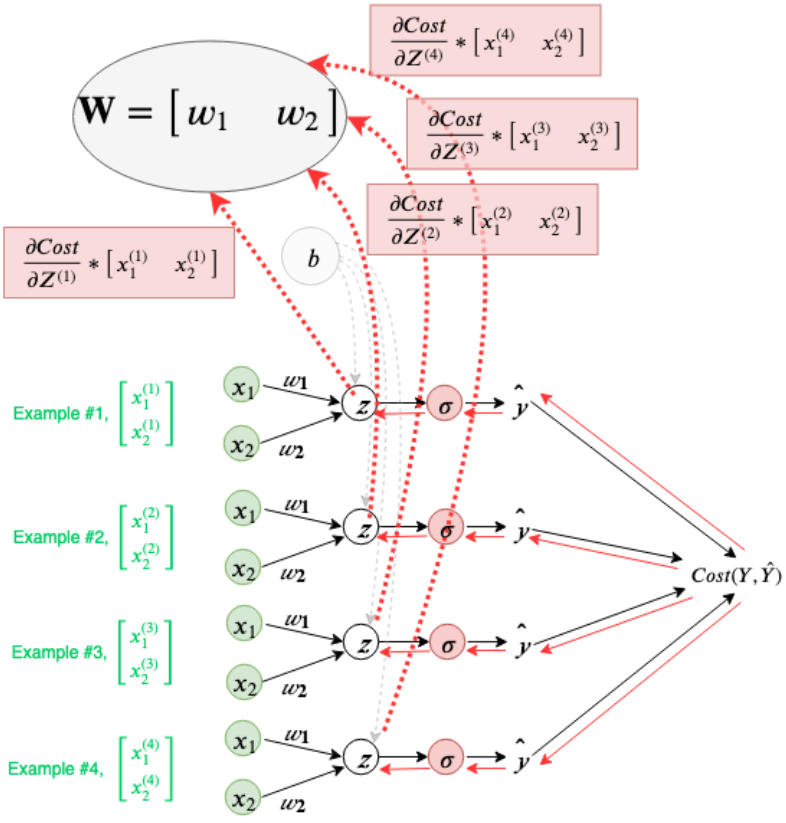


Fig 59. Visualizing all possible backpropagation paths to shared weights parameter

Again, following the general rule defined above, we will sum all the incoming gradients from all the possible paths to the weights node, **W**.

So the total of the gradient flowing into node ***W***(weights) is:

$$
\begin{aligned}
\frac{\partial Cost}{\partial W} &= \sum_{i=1}^{m} \left(\text{gradient from } i^{th}\text{example}\right) \\
&= \left(\text{gradient from } 1^{st}\text{example}\right) + \left(\text{gradient from } 2^{nd}\text{example}\right) + \left(\text{gradient from } 3^{rd}\text{example}\right) + \left(\text{gradient from } 4^{th}\text{example}\right) \\
&= \left(\frac{\partial Cost}{\partial Z^{(1)}} * \frac{\partial Z^{(1)}}{\partial W}\right) + \left(\frac{\partial Cost}{\partial Z^{(2)}} * \frac{\partial Z^{(2)}}{\partial W}\right) + \left(\frac{\partial Cost}{\partial Z^{(3)}} * \frac{\partial Z^{(3)}}{\partial W}\right) + \left(\frac{\partial Cost}{\partial Z^{(4)}} * \frac{\partial Z^{(4)}}{\partial W}\right) \\
&= \left(\frac{\partial Cost}{\partial Z^{(1)}} * \begin{bmatrix} x_1^{(1)} & x_2^{(1)} \end{bmatrix}\right) + \left(\frac{\partial Cost}{\partial Z^{(2)}} * \begin{bmatrix} x_1^{(2)} & x_2^{(2)} \end{bmatrix}\right) + \left(\frac{\partial Cost}{\partial Z^{(3)}} * \begin{bmatrix} x_1^{(3)} & x_2^{(3)} \end{bmatrix}\right) + \left(\frac{\partial Cost}{\partial Z^{(4)}} * \begin{bmatrix} x_1^{(4)} & x_2^{(4)} \end{bmatrix}\right) \\
&= \begin{bmatrix} \frac{\partial Cost}{\partial Z^{(1)}} x_1^{(1)} & \frac{\partial Cost}{\partial Z^{(1)}} x_2^{(1)} \end{bmatrix} + \begin{bmatrix} \frac{\partial Cost}{\partial Z^{(2)}} x_1^{(2)} & \frac{\partial Cost}{\partial Z^{(2)}} x_2^{(2)} \end{bmatrix} + \begin{bmatrix} \frac{\partial Cost}{\partial Z^{(3)}} x_1^{(3)} & \frac{\partial Cost}{\partial Z^{(3)}} x_2^{(3)} \end{bmatrix} + \begin{bmatrix} \frac{\partial Cost}{\partial Z^{(4)}} x_1^{(4)} & \frac{\partial Cost}{\partial Z^{(4)}} x_2^{(4)} \end{bmatrix} \\
&= \begin{bmatrix} \left(\frac{\partial Cost}{\partial Z^{(1)}} x_1^{(1)} + \frac{\partial Cost}{\partial Z^{(2)}} x_1^{(2)} + \frac{\partial Cost}{\partial Z^{(3)}} x_1^{(3)} + \frac{\partial Cost}{\partial Z^{(4)}} x_1^{(4)}\right) & \left(\frac{\partial Cost}{\partial Z^{(1)}} x_2^{(1)} + \frac{\partial Cost}{\partial Z^{(2)}} x_2^{(2)} + \frac{\partial Cost}{\partial Z^{(3)}} x_2^{(3)} + \frac{\partial Cost}{\partial Z^{(4)}} x_2^{(4)}\right) \end{bmatrix} \\
&\qquad \underbrace{\qquad\qquad}_{\frac{\partial Cost}{\partial w_1}} \qquad\qquad \underbrace{\qquad\qquad}_{\frac{\partial Cost}{\partial w_2}}
\end{aligned}
$$

So, this also exactly like the result for gradient of ***W*** in ***Fig 44.c.***

Fig 60. Derivation of ∂Cost/∂W after visualization.

Up till now what we've done to calculate, ***∂Cost/∂W***, though is correct and serves as a good explanation however, it is not an optimized calculation. We can vectorize this calculation, too. Let's do that next

Recall that, in our vectorized OR gate example the gradient flowing backward to the ***Z*** node was a (1×4) matrix $\left(\begin{bmatrix} \frac{\partial Cost}{\partial Z^{(1)}} & \frac{\partial Cost}{\partial Z^{(2)}} & \frac{\partial Cost}{\partial Z^{(3)}} & \frac{\partial Cost}{\partial Z^{(4)}} \end{bmatrix}\right)$ and the input X_{train} was a (2×4) matrix $\left(\begin{bmatrix} x_1^{(1)} & x_1^{(2)} & x_1^{(3)} & x_1^{(4)} \\ x_2^{(1)} & x_2^{(2)} & x_2^{(3)} & x_2^{(4)} \end{bmatrix}\right)$

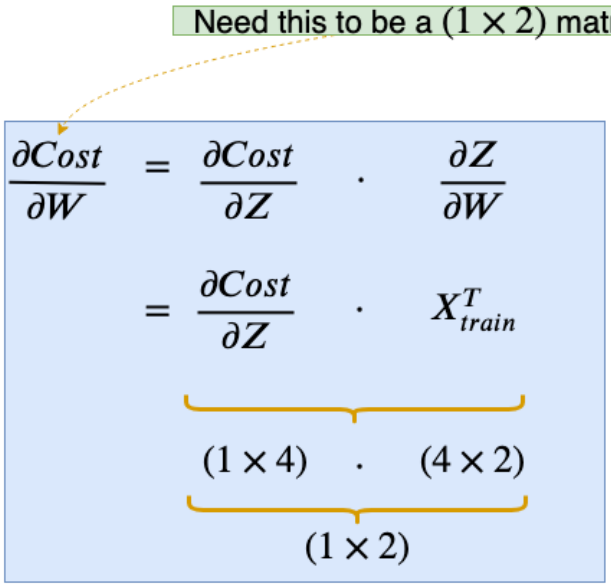
$$
\begin{aligned}
\frac{\partial Cost}{\partial W} &= UpstreamGradient \cdot LocalGradeint \\
&= \frac{\partial Cost}{\partial Z} \cdot \frac{\partial Z}{\partial W} \\
&= \frac{\partial Cost}{\partial Z} \cdot X_{train}^{T} \\
&= \begin{bmatrix} \frac{\partial Cost}{\partial Z^{(1)}} & \frac{\partial Cost}{\partial Z^{(2)}} & \frac{\partial Cost}{\partial Z^{(3)}} & \frac{\partial Cost}{\partial Z^{(4)}} \end{bmatrix} \cdot \begin{bmatrix} x_1^{(1)} & x_2^{(1)} \\ x_1^{(2)} & x_2^{(2)} \\ x_1^{(3)} & x_2^{(3)} \\ x_1^{(4)} & x_2^{(4)} \end{bmatrix} \\
&= \begin{bmatrix} \left(\frac{\partial Cost}{\partial Z^{(1)}} x_1^{(1)} + \frac{\partial Cost}{\partial Z^{(2)}} x_1^{(2)} + \frac{\partial Cost}{\partial Z^{(3)}} x_1^{(3)} + \frac{\partial Cost}{\partial Z^{(4)}} x_1^{(4)}\right) & \left(\frac{\partial Cost}{\partial Z^{(1)}} x_2^{(1)} + \frac{\partial Cost}{\partial Z^{(2)}} x_2^{(2)} + \frac{\partial Cost}{\partial Z^{(3)}} x_2^{(3)} + \frac{\partial Cost}{\partial Z^{(4)}} x_2^{(4)}\right) \end{bmatrix} \\
&\qquad \underbrace{\qquad\qquad}_{\frac{\partial Cost}{\partial w_1}} \qquad\qquad \underbrace{\qquad\qquad}_{\frac{\partial Cost}{\partial w_2}}
\end{aligned}
$$

Fig 61. Proof that ∂Cost/∂W is the dot product between the Upstream gradient and the transpose of $\boldsymbol{X_{train}}$

Is there an easier way of figuring this out, without the math?

Yes! Use ***dimension analysis***.

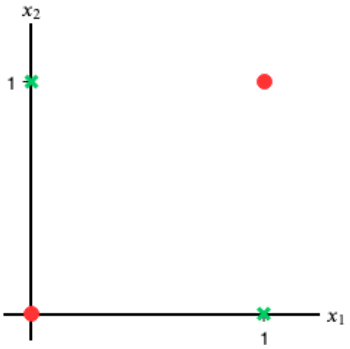
In our OR gate example we know that the gradient flowing into node ***Z*** is a (1 × 4) matrix, X_{train} is a (2 × 4) matrix and the derivative of Cost with respect to the ***W*** needs to be of the same size as ***W***, which is (1 × 2). So, the only way to generate a (1 × 2) matrix would be to take the dot product of between ***Z*** and transpose of $\boldsymbol{X_{train}}$.



Similarly, knowing that bias, ***b***, is a simple (1 × 1) matrix and the gradient flowing into node Z is (1 × 4), using dimension analysis we can be sure that the gradient of Cost w.r.t ***b***, also needs to be a (1 × 1) matrix. The only way we can achieve this, given the local gradient(***∂Z/∂b***) is just equal to ***1***, is by summing up the upstream gradient.

On a final note when deriving derivative expressions work on small examples and then generalize from there. For example here, while calculating the derivative of the dot product w.r.t to ***W,*** *we used a single column vector as a test case and generalized from there, if we would have used the entire data matrix then the derivative would have resulted in a (4 × 1 × 2) tensor (multidimensional matrix), calculation on which can get a bit hairy.*

Before concluding this section lets go over a slightly more complicated example.



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

Fig 62. XOR gate data

Figure 62, above, represents an XOR gate data. Looking at it note that the label, ***y***, is equal to ***1*** only when one of the values x_1 or x_2 is equal to ***1***, *not both*. This makes it a particularly challenging dataset as the data is not linearly separable, i.e. there is no single straight line decision boundary that can successfully separate the two classes(***y=1*** and ***y=0***) in the data. XOR used to be the bane of earlier forms of artificial neural networks.

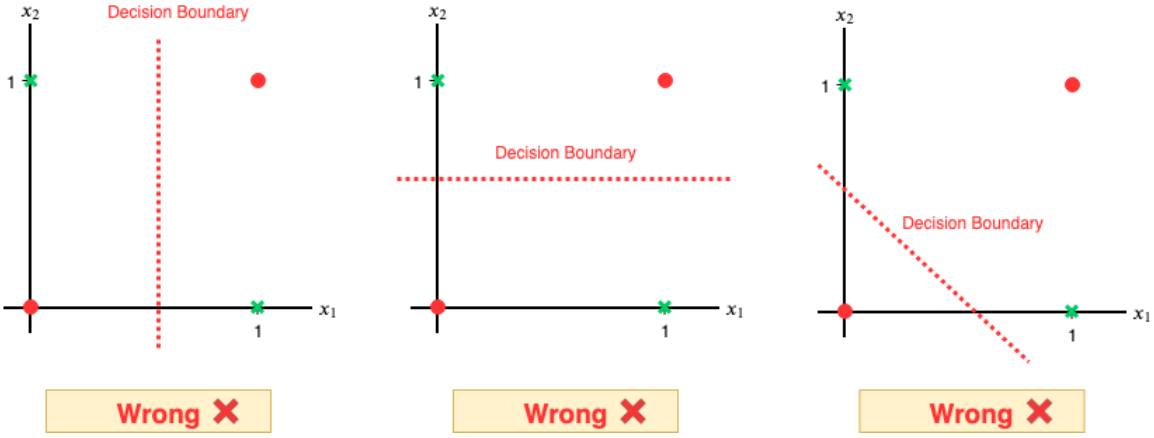


Fig 63. Some linear decision boundaries that are wrong

Recall that our current neural network was successful only because it could figure out the straight line decision boundary that could successfully separate the two classes of the OR gate dataset. A straight line won't cut it here. So, how do we get a neural network to figure this one out?

Well, we can do two things:

1. Amend the data itself, so that in addition to features x_1 and x_2 a third feature provides some additional information to help the neural network decide on a good decision boundary. This process is called ***feature engineering***.
2. Change the architecture of the neural network, making it deeper.

Let's go over both and see which one is better.

Feature Engineering

Let's look at a dataset similar looking to the XOR data that will help us in making an important realization.

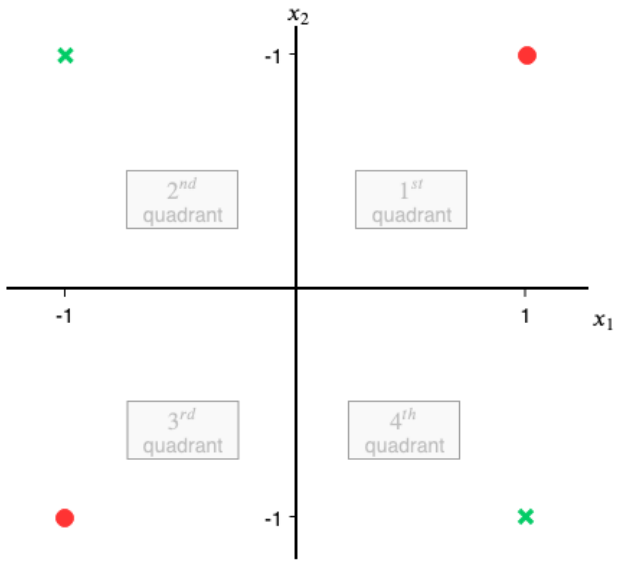


Fig 64. XOR-like data in different quadrants

The data in Figure 64 is exactly like the XOR data except each data point is spread out in different quadrants. Notice that in the **1st and 3rd quadrant all the values are positive** and in the **2nd and 4th all the values are negative.**

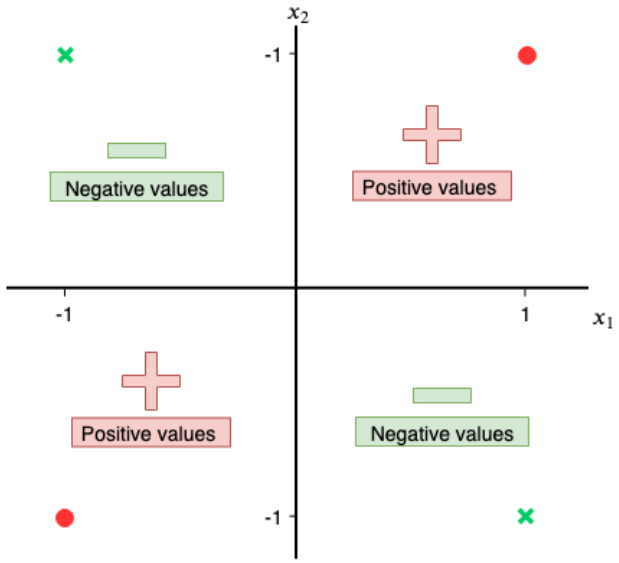


Fig 65. Positive and negative quadrants

Why is that? In the **1st** and **3rd** quadrants **the signs of values are being squared,** while in the **2nd** and **4th** quadrants **the values are a simple product between a negative and positive number resulting in a negative number.**

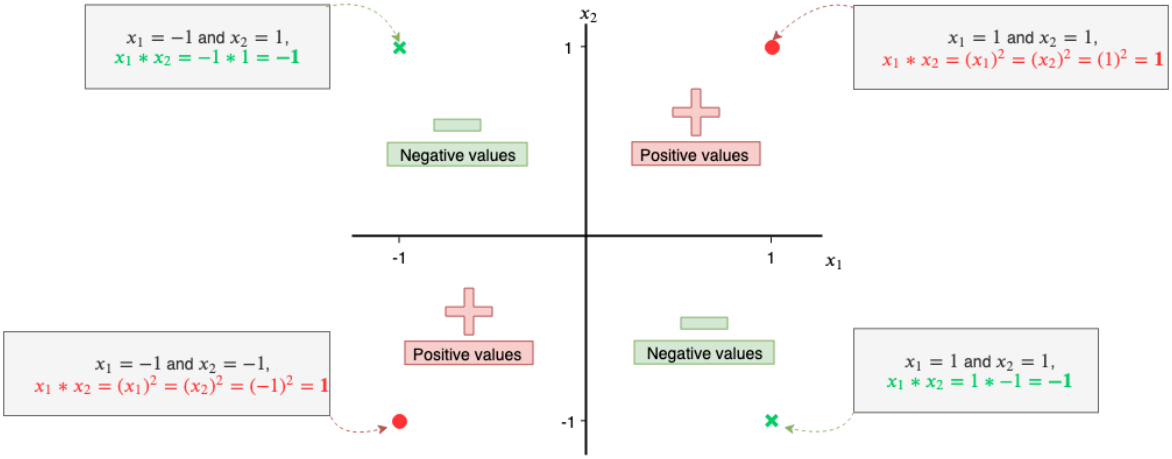


Fig 66. Result of the product of features

So this gives us a pattern to work with using the product of two features. We can even see a similar pattern in the XOR data, where each quadrant can be identified in a similar way.

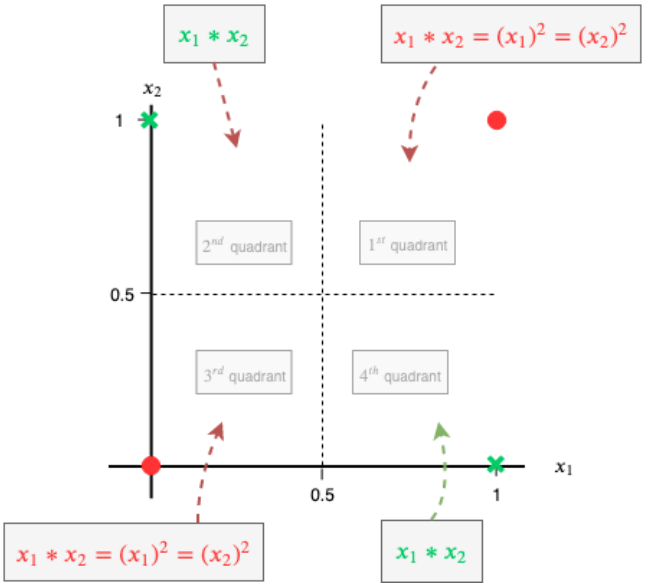


Fig 67. Quadrant-pattern in XOR data plot

Therefore, a good third feature, x_3, would be the product of features x_1 and x_2(i.e. $x_1 * x_2$).

Product of features is called a ***feature cross*** and results in a new ***synthetic feature***. *Feature crosses can be either the feature itself(eg. $x_1^2, x_1^3, \ldots$), a product of two or more features(eg. $x_1 * x_2, x_1 * x_2 * x_3, \ldots$) or even a combination of both(eg. $x_1^2 * x_2$). For example, in a housing dataset where the input features are the width and length of houses in yards and label is the location of the house on the map, a better predictor for this location could be the feature cross between width and length of houses, giving us a new feature of "size of house in square yards".*

Let's add the new synthetic feature to our training data, X_{train}.

x_1	x_2	$x_3 = x_1 * x_2$	y
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

$$X_{train} = X^T = \begin{matrix} \text{example \#1} & \text{example \#2} & \text{example \#3} & \text{example \#4} \\ \begin{bmatrix} 0 & 0 & 1 & 1 \\ 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 \end{bmatrix} & \begin{matrix} x_1 \\ x_2 \\ x_3 \end{matrix} \end{matrix}$$

Fig 68. New training data

Using this feature cross we can now successfully learn a decision boundary without changing the architecture of the neural network significantly. We only need to add an input node for x_3 and a corresponding weight(*randomly set to 0.2*) to the input layer.

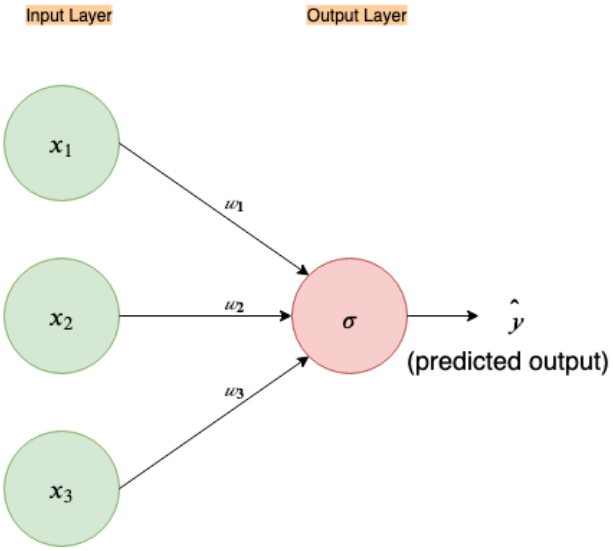


Fig 69. Neural Network with feature cross(x_3) as input

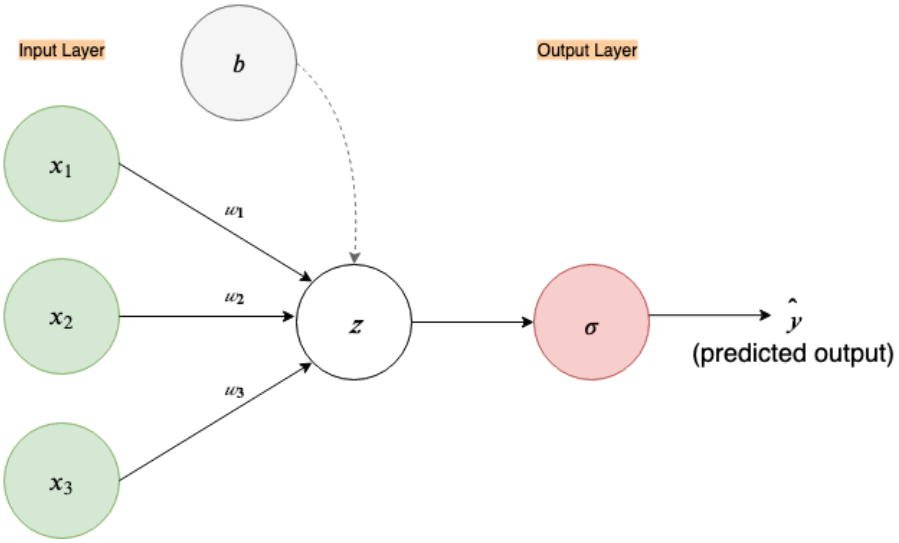


Fig 70. Expanded neural network with feature cross(x_3) as input

Given below is the *first* training iteration of the neural network, you may go through the computations yourself and confirm them as they make for a good exercise. Since we are already familiar with this neural network architecture, I will not go through all the computations in a step-by-step by step manner, as before.

(All calculations below are rounded to 3 decimal places)

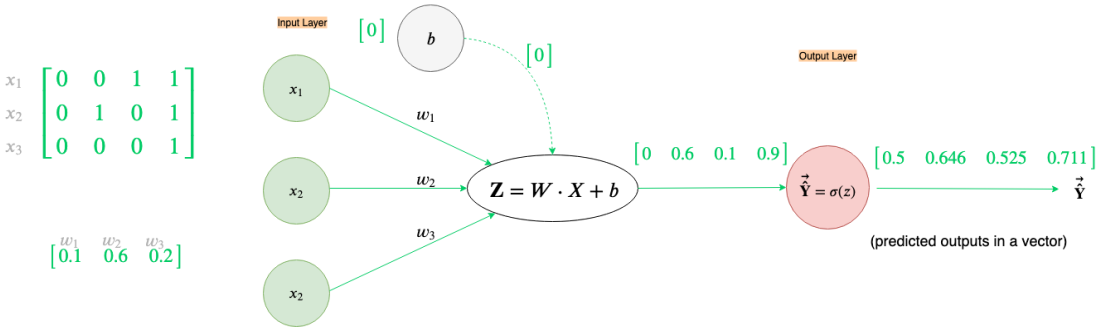


Fig 71. Forward Propagation in the first training iteration

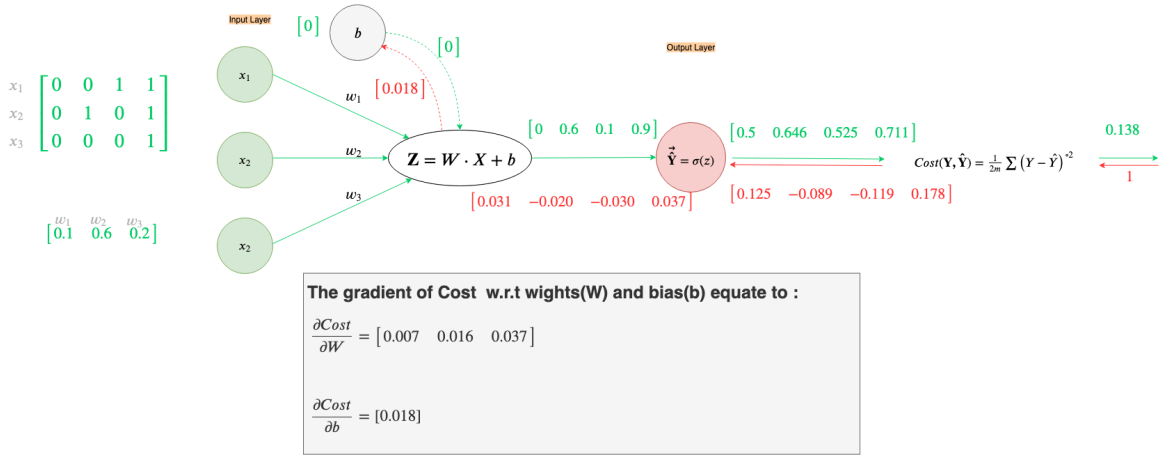


Fig 72. Backpropagation in the first training iteration

Gradient Descent Update

To calculate new weights(*W*) and bias(*b*) we move in the negative direction of the gradient

Our current Weight vector is $W = [0.1 \quad 0.6 \quad 0.2]$, $\alpha = 1$ and $\frac{\partial Cost}{\partial W} = [0.007 \quad 0.016 \quad 0.037]$

The new Weights are:

$$\begin{aligned} W &= W - \alpha \frac{\partial Cost}{\partial W} \\ &= [0.1 \quad 0.6 \quad 0.2] - (1 * [0.007 \quad 0.016 \quad 0.037]) \\ &= [0.1 \quad 0.6 \quad 0.2] - [0.007 \quad 0.016 \quad 0.037] \\ &= [0.1 - 0.007 \quad 0.6 - 0.016 \quad 0.2 - 0.037] \\ &= [0.093 \quad 0.584 \quad 0.163] \end{aligned}$$

Our current Bias vector is $b = [0]$, $\alpha = 1$ and $\frac{\partial Cost}{\partial b} = [0.018]$

The new Bias is:

$$\begin{aligned} b &= b - \alpha \frac{\partial Cost}{\partial b} \\ &= [0] - (1 * [0.018]) \\ &= [0] - [0.018] \\ &= [0 - 0.018] \\ &= [-0.018] \end{aligned}$$

Fig 73. Gradient descent update for new weights and bias, in the first training iteration

After 5000 epochs, the learning curve, and the decision boundary look as follows:

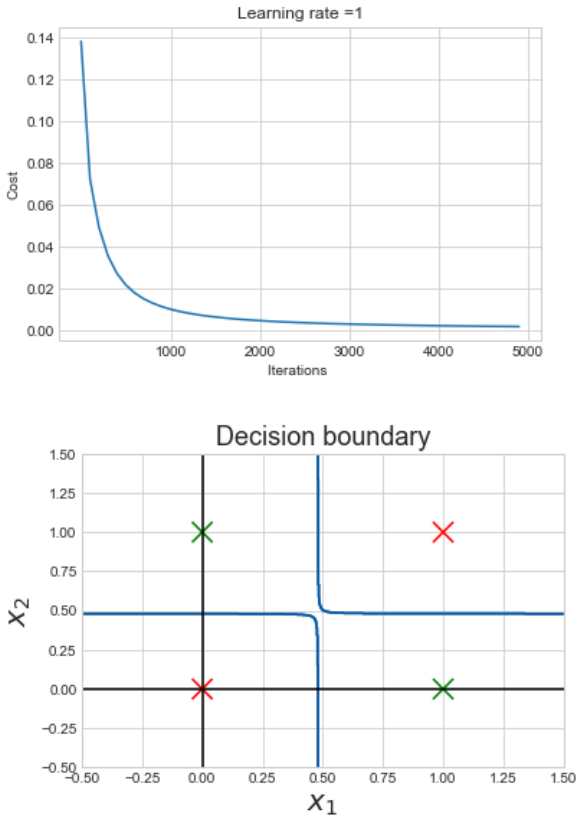


Fig 74. Learning Curve and Decision Boundary of the neural net with a feature cross

As before, to visualize better we can shade the regions where the decision of the neural network changes from one to the other.

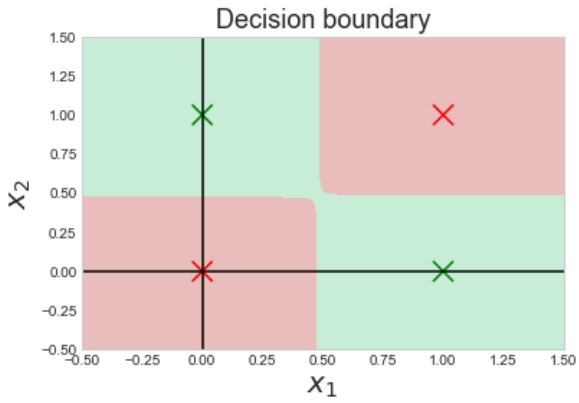


Fig 75. Shaded Decision Boundary for better visualization

Note that feature engineering allowed us to create a decision boundary that is nonlinear. How did it do that? We just need to take a look at what function the ***Z*** node is computing.

From the training data recall that the third feature(x_3) is the feature cross of x_1 and x_2:

$$X = \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} x_1 \\ x_2 \\ x_1 * x2 \end{bmatrix},$$

and the weight matrix is a row vector, $W = \begin{bmatrix} w_1 & w_2 & w_3 \end{bmatrix}$.

Let's take a look at what the Z node is computing:

$$\begin{aligned} Z &= W \cdot X + b \\ &= \begin{bmatrix} w_1 & w_2 & w_3 \end{bmatrix} \cdot \begin{bmatrix} x_1 \\ x_2 \\ x_1 * x2 \end{bmatrix} + b \\ &= w_1 x_1 + w_2 x_2 + w_3 (x_1 * x_2) + b \end{aligned}$$

This feature cross makes the Z equation a polynomial. It is no longer a simple linear function

Fig 76. Node **Z** is computing a polynomial after adding a feature cross

Thus, feature cross helped us to create complex non-linear decision boundary.

This is a very powerful idea!

Changing Neural Network Architecture

This is the more interesting approach as it allows us to bypass the feature engineering ourselves and ***lets the neural network figure out the feature crosses itself!***

Let's take a look at the following neural network:

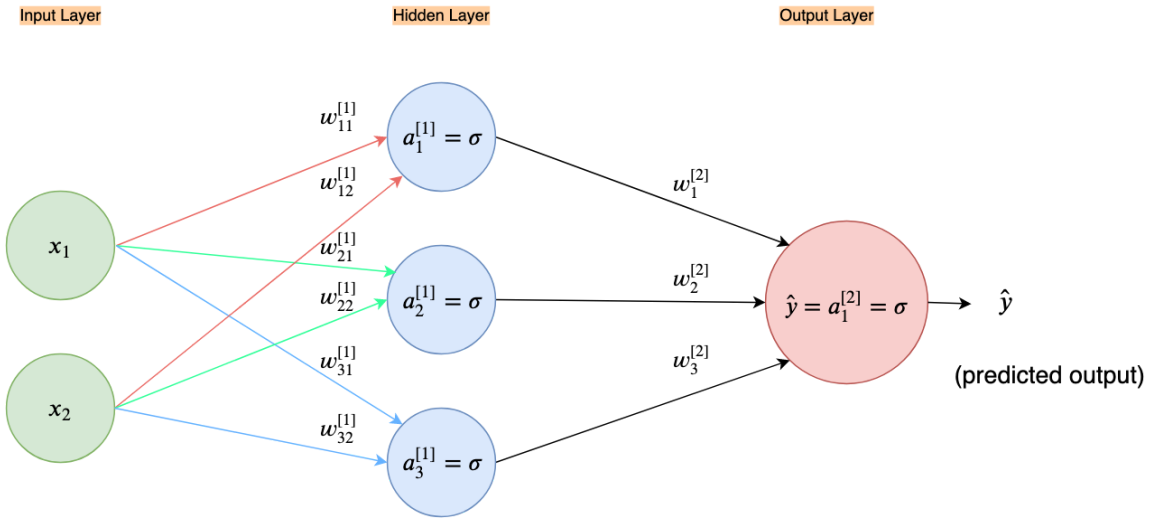


Fig 77. Neural network with one hidden layer.

So we've added a bunch of new nodes in the middle of our neural network architecture from the OR gate example, keeping the input layer and the output layer the same. This column of new nodes in the middle is called a ***hidden layer.*** *Why hidden layer? Because after defining it we don't have any direct*

control over how the neurons in the hidden layers learn, unlike the input and output layer which we can change by changing the data; also since the hidden layers neither constitute as the output or the input of the neural network they are in essence hidden from the user.

We can have an arbitrary number of hidden layers with an arbitrary number of neurons in each layer. This structure needs to be defined by the creator of the neural network. *Thus, the* ***number of hidden layers and the number of neurons in each of the layers are also hyper-parameters. The more hidden layers we add the deeper our neural network architecture becomes and the more neurons we add in the hidden layers the wider the network architecture becomes. The depth of a neural net model is where the term "Deep learning" comes from.***

The architecture in Figure 77 with one hidden layer of three sigmoid neurons, was selected after some experimentation.

Since this is a new architecture I'll go over the computations step-by-step.

First, let's expand out the neural network.

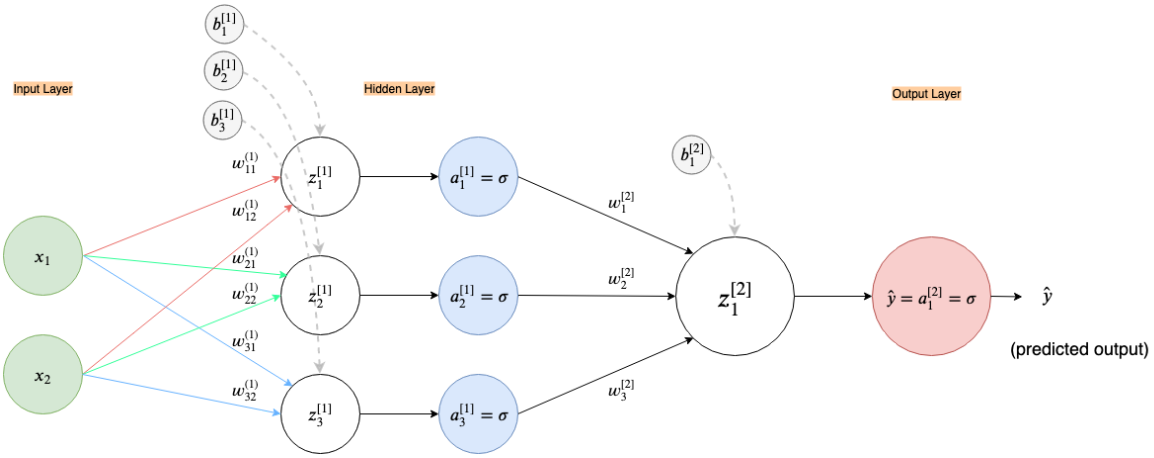
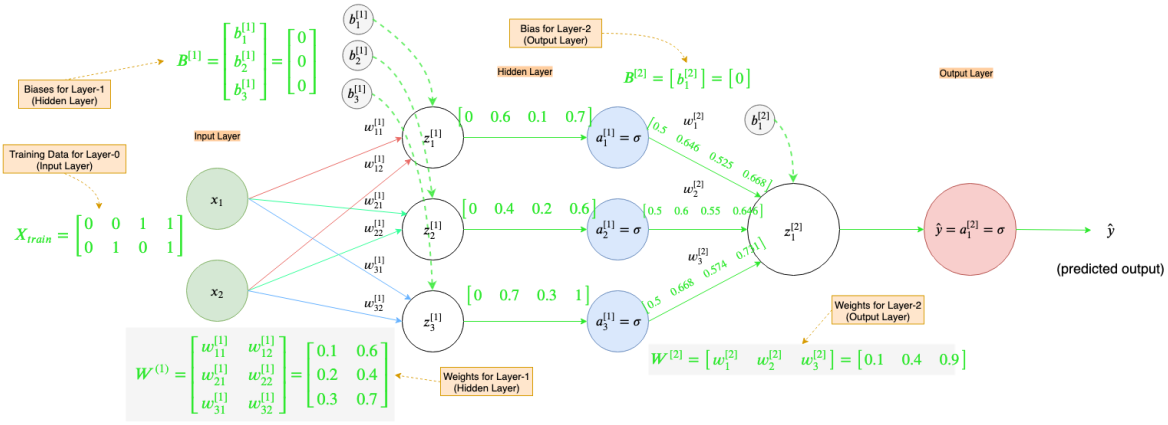


Fig 78. Expanded neural network with one hidden layer

Now let's perform a ***forward propagation:***



The neural network is going through the following computations(forward computations marked in green):

- The first layer(input layer) is passed our data $X_{train} = \begin{bmatrix} 0 & 0 & 1 & 1 \\ 0 & 1 & 0 & 1 \end{bmatrix}$ and randomly chosen weights, $W^{[1]} = \begin{bmatrix} 0.1 & 0.6 \\ 0.2 & 0.4 \\ 0.4 & 0.7 \end{bmatrix}$ and zero initialized bias, $B^{[1]} = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix}$.
- In our neural network, the ***input layer is "layer-0", the hidden layer is "layer 1" and the output layer is "layer-2".*** So the superscript "**[1]**" denotes that the data element belongs to ***layer 1,*** and connects elements from layer-0 to layer-1.

- All the $z^{[1]}$ nodes are calculated in one go in a vectorized calculation, as follows:

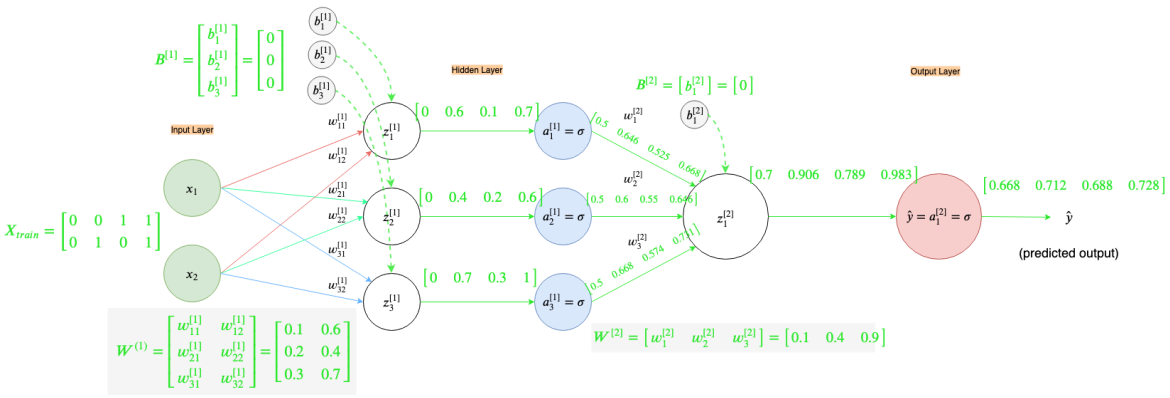
$$
\begin{aligned}
Z^{[1]} &= W^{[1]} \cdot X_{trian} + B^{[1]} \\
&= \begin{bmatrix} w_{11}^{[1]} & w_{12}^{[1]} \\ w_{21}^{[1]} & w_{22}^{[1]} \\ w_{31}^{[1]} & w_{32}^{[1]} \end{bmatrix} \cdot \begin{bmatrix} x_1^{(1)} & x_1^{(2)} & x_1^{(3)} & x_1^{(4)} \\ x_2^{(1)} & x_2^{(2)} & x_2^{(3)} & x_2^{(4)} \end{bmatrix} \\
&= \begin{bmatrix} 0.1 & 0.6 \\ 0.2 & 0.4 \\ 0.3 & 0.7 \end{bmatrix} \cdot \begin{bmatrix} 0 & 0 & 1 & 1 \\ 0 & 1 & 0 & 1 \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix} \\
&= \begin{bmatrix} 0 & 0.6 & 0.1 & 0.7 \\ 0 & 0.4 & 0.2 & 0.6 \\ 0 & 0.7 & 0.3 & 1 \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix} \\
&= \begin{bmatrix} 0 & 0.6 & 0.1 & 0.7 \\ 0 & 0.4 & 0.2 & 0.6 \\ 0 & 0.7 & 0.3 & 1 \end{bmatrix} \begin{matrix} z_1^{[1]} \\ z_2^{[1]} \\ z_3^{[1]} \end{matrix} = \begin{bmatrix} z_1^{[1](1)} & z_1^{[1](2)} & z_1^{[1](3)} & z_1^{[1](4)} \\ z_2^{[1](1)} & z_2^{[1](2)} & z_2^{[1](3)} & z_3^{[1](4)} \\ z_3^{[1](1)} & z_3^{[1](2)} & z_3^{[1](3)} & z_3^{[1](4)} \end{bmatrix}
\end{aligned}
$$

example #1 example #2 example #3 example #4

- Similar to all the z nodes in layer-1 all the activations of layer one are also calculated in one go, as follows:

$$
\begin{aligned}
A^{[1]} &= \sigma(Z^{[1]}) \\
&= \begin{bmatrix} \sigma(z_1^{[1](1)}) & \sigma(z_1^{[1](2)}) & \sigma(z_1^{[1](3)}) & \sigma(z_1^{[1](4)}) \\ \sigma(z_2^{[1](1)}) & \sigma(z_2^{[1](2)}) & \sigma(z_2^{[1](3)}) & \sigma(z_3^{[1](4)}) \\ \sigma(z_3^{[1](1)}) & \sigma(z_3^{[1](2)}) & \sigma(z_3^{[1](3)}) & \sigma(z_3^{[1](4)}) \end{bmatrix} \\
&= \begin{bmatrix} \sigma(0) & \sigma(0.6) & \sigma(0.1) & \sigma(0.7) \\ \sigma(0) & \sigma(0.4) & \sigma(0.2) & \sigma(0.6) \\ \sigma(0) & \sigma(0.7) & \sigma(0.3) & \sigma(1) \end{bmatrix} \\
&= \begin{bmatrix} 0.5 & 0.646 & 0.525 & 0.668 \\ 0.5 & 0.6 & 0.55 & 0.646 \\ 0.5 & 0.668 & 0.574 & 0.731 \end{bmatrix} = \begin{bmatrix} a_1^{[1](1)} & a_1^{[1](2)} & a_1^{[1](3)} & a_1^{[1](4)} \\ a_2^{[1](1)} & a_2^{[1](2)} & a_2^{[1](3)} & a_2^{[1](4)} \\ a_3^{[1](1)} & a_3^{[1](2)} & a_3^{[1](3)} & a_3^{[1](4)} \end{bmatrix}
\end{aligned}
$$

Fig 79.a. Forward propagation on the neural net with a hidden layer



The neural network is going through the following computations(forward computations marked in green):

- Now we'll propagate the values from the ***Hidden Layer(layer-1)*** to the ***Output Layer(layer-2).***
- **The values following into $z_1^{[2]}$ are the activations from the previous layer(the Hidden Layer, in this case).**

$$A_{prev} = A^{[1]} = \begin{bmatrix} 0.5 & 0.646 & 0.525 & 0.668 \\ 0.5 & 0.6 & 0.55 & 0.646 \\ 0.5 & 0.668 & 0.574 & 0.731 \end{bmatrix}$$

- As before we'll perform all the calculations for all the examples in one go.

$$\begin{aligned} Z^{[2]} &= W^{[2]} \cdot A^{[1]} + B^{[2]} \\ &= \begin{bmatrix} w_1^{[2]} & w_2^{[2]} & w_3^{[2]} \end{bmatrix} \cdot \begin{bmatrix} a_1^{[1](1)} & a_1^{[1](2)} & a_1^{[1](3)} & a_1^{[1](4)} \\ a_2^{[1](1)} & a_2^{[1](2)} & a_2^{[1](3)} & a_3^{[1](4)} \\ a_3^{[1](1)} & a_3^{[1](2)} & a_3^{[1](3)} & a_3^{[1](4)} \end{bmatrix} + \begin{bmatrix} b_1^{[2]} \end{bmatrix} \\ &= \begin{bmatrix} 0.1 & 0.4 & 0.9 \end{bmatrix} \cdot \begin{bmatrix} 0.5 & 0.646 & 0.525 & 0.668 \\ 0.5 & 0.6 & 0.55 & 0.646 \\ 0.5 & 0.668 & 0.574 & 0.731 \end{bmatrix} + \begin{bmatrix} 0 \end{bmatrix} \\ &= \begin{bmatrix} 0.7 & .906 & 0.789 & 0.983 \end{bmatrix} + \begin{bmatrix} 0 \end{bmatrix} \\ &= \begin{bmatrix} 0.7 & .906 & 0.789 & 0.983 \end{bmatrix} = \begin{bmatrix} z_1^{[2](1)} & z_1^{[2](2)} & z_1^{[2](3)} & z_1^{[2](4)} \end{bmatrix} \end{aligned}$$

- Now we can calculate the activations of the Output Layer(layer-2):

$$\begin{aligned} \hat{Y} = A^{[2]} &= \sigma(Z^{[2]}) \\ &= \begin{bmatrix} \sigma(z_1^{[2](1)}) & \sigma(z_1^{[2](2)}) & \sigma(z_1^{[2](3)}) & \sigma(z_1^{[2](4)}) \end{bmatrix} \\ &= \begin{bmatrix} \sigma(0.7) & \sigma(0.906) & \sigma(0.789) & \sigma(0.983) \end{bmatrix} \\ &= \begin{bmatrix} 0.668 & 0.712 & 0.688 & 0.728 \end{bmatrix} = \begin{bmatrix} a_1^{[2](1)} & a_1^{[2](2)} & a_1^{[2](3)} & a_1^{[2](4)} \end{bmatrix} = \begin{bmatrix} \hat{y}^{(1)} & \hat{y}^{(2)} & \hat{y}^{(3)} & \hat{y}^{(4)} \end{bmatrix} \end{aligned}$$

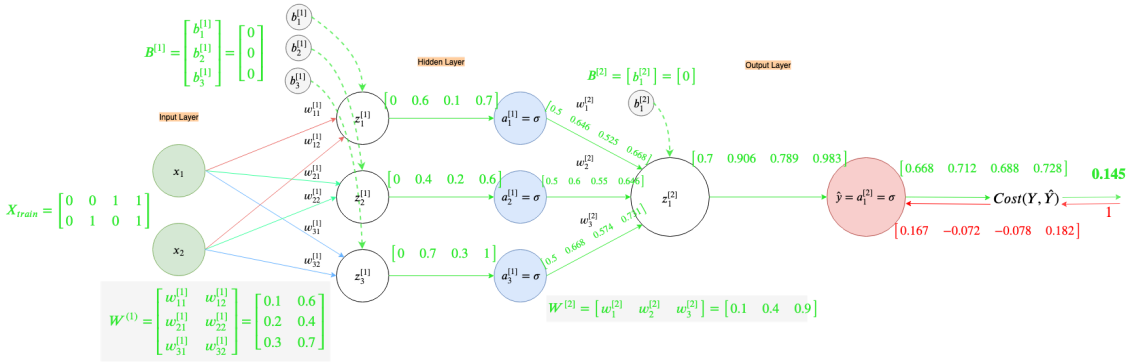
Fig 79.b. Forward propagation on the neural net with a hidden layer

We can now calculate the Cost:

$$\begin{aligned} Cost(Y, \hat{Y}) &= \frac{1}{2m} \sum_{i=1}^{m} (y^{(i)}, \hat{y}^{(i)})^2 \\ &= \frac{1}{2m} \sum (Y - \hat{Y})^{\circ 2} \\ &= \frac{1}{2(4)} \sum (\begin{bmatrix} 0 & 1 & 1 & 0 \end{bmatrix} - \begin{bmatrix} 0.668 & 0.712 & 0.688 & 0.728 \end{bmatrix})^{\circ 2} \\ &= \frac{1}{8} \sum \begin{bmatrix} (0 - 0.668) & (1 - 0.712) & (1 - 0.688) & (0 - 0.728) \end{bmatrix}^{\circ 2} \\ &= \frac{1}{8} \sum \begin{bmatrix} (-0.668)^2 & (0.288)^2 & (0.312)^2 & (-0.728)^2 \end{bmatrix} \\ &= \frac{1}{8} [(-0.668)^2 + (0.288)^2 + (0.312)^2 + (-0.728)^2] \\ &= \mathbf{0.145} \end{aligned}$$

Fig 80. Cost of the neural net with one hidden layer after **first** forward propagation

After the calculation of Cost, we can now do our backpropagation and improve the weights and biases.



The neural network is going through the following computations(backward computations are in red):

- As before the first **Upstream Gradient** in redundant. $\frac{\partial Cost}{\partial Cost} = 1$

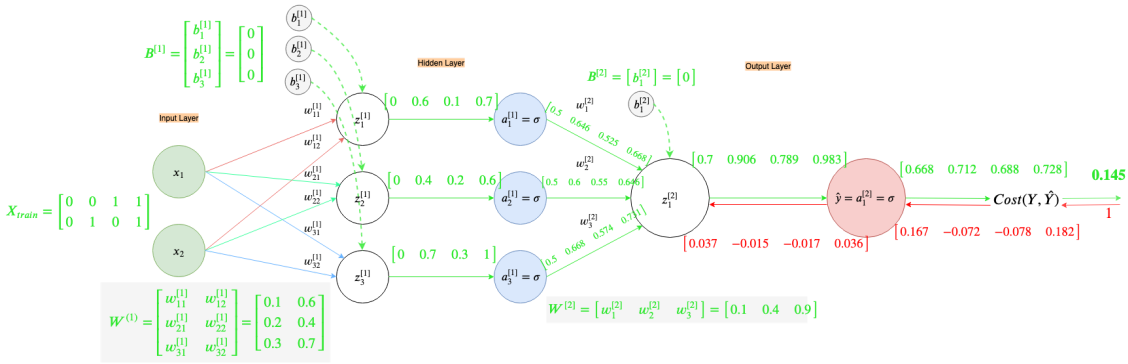
- The **Local Gradient** is :

$$\begin{aligned}
\frac{\partial Cost}{\partial \hat{Y}} &= -\frac{1}{m}(Y - \hat{Y}) \\
&= -\frac{1}{4}([0 \quad 1 \quad 1 \quad 0] - [0.668 \quad 0.712 \quad 0.688 \quad 0.728]) \\
&= -\frac{1}{4}([-0.668 \quad 0.288 \quad 0.312 \quad -0.728]) \\
&= [0.167 \quad -0.072 \quad -0.078 \quad 0.182] = \left[\frac{\partial Cost}{\partial \hat{y}^{(1)}} \quad \frac{\partial Cost}{\partial \hat{y}^{(2)}} \quad \frac{\partial Cost}{\partial \hat{y}^{(3)}} \quad \frac{\partial Cost}{\partial \hat{y}^{(4)}}\right]
\end{aligned}$$

- We'll combine the upstream and local gradient and send it back to the red node of the Output Layer:

$$\begin{aligned}
\frac{\partial Cost}{\partial \hat{Y}} &= UpstreamGradeint * LocalGradient \\
&= \frac{\partial Cost}{\partial Cost} * \frac{\partial Cost}{\partial \hat{Y}} \\
&= 1 * \left[\frac{\partial Cost}{\partial \hat{y}^{(1)}} \quad \frac{\partial Cost}{\partial \hat{y}^{(2)}} \quad \frac{\partial Cost}{\partial \hat{y}^{(3)}} \quad \frac{\partial Cost}{\partial \hat{y}^{(4)}}\right] \\
&= 1 * [0.167 \quad -0.072 \quad -0.078 \quad 0.182] \\
&= [0.167 \quad -0.072 \quad -0.078 \quad 0.182] = \left[\frac{\partial Cost}{\partial \hat{y}^{(1)}} \quad \frac{\partial Cost}{\partial \hat{y}^{(2)}} \quad \frac{\partial Cost}{\partial \hat{y}^{(3)}} \quad \frac{\partial Cost}{\partial \hat{y}^{(4)}}\right]
\end{aligned}$$

Fig 81.a. Backpropagation on the neural net with a hidden layer



The neural network is going through the following computations(backward computations are in red):

- The **Upstream Gradient** in this step is:

$$\frac{\partial Cost}{\partial \hat{Y}} = \begin{bmatrix} 0.167 & -0.072 & -0.078 & 0.182 \end{bmatrix}$$

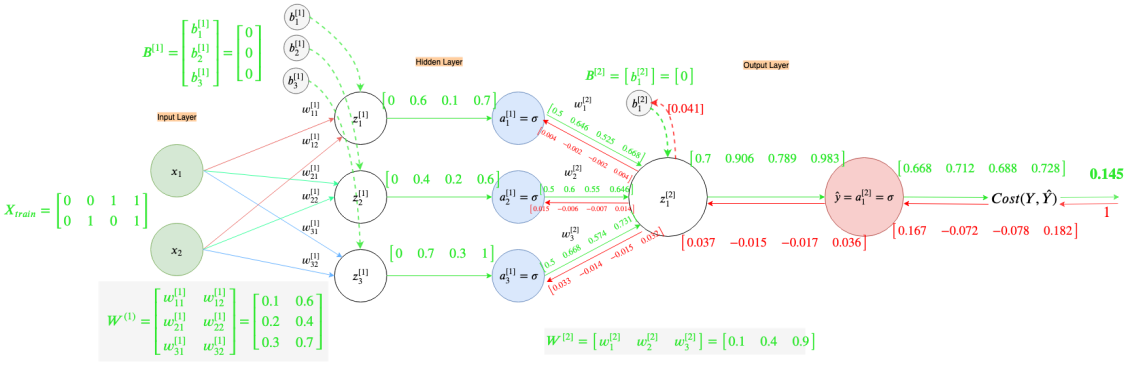
- The **Local Gradient** is :

$$\begin{aligned} \frac{\partial \hat{Y}}{\partial Z^{(2)}} &= \hat{Y}\left(1-\hat{Y}\right) \\ &= \begin{bmatrix} 0.668 & 0.712 & 0.688 & 0.728 \end{bmatrix} * \left(1 - \begin{bmatrix} 0.668 & 0.712 & 0.688 & 0.728 \end{bmatrix}\right) \\ &= \begin{bmatrix} 0.668 & 0.712 & 0.688 & 0.728 \end{bmatrix} * \begin{bmatrix} 0.332 & 0.288 & 0.312 & 0.272 \end{bmatrix} \\ &= \begin{bmatrix} 0.222 & 0.205 & 0.215 & 0.198 \end{bmatrix} = \begin{bmatrix} \frac{\partial y^{(1)}}{\partial z_1^{[2](1)}} & \frac{\partial y^{(2)}}{\partial z_1^{[2](2)}} & \frac{\partial y^{(3)}}{\partial z_1^{[2](3)}} & \frac{\partial y^{(4)}}{\partial z_1^{[2](4)}} \end{bmatrix} \end{aligned}$$

- We'll combine the upstream and local gradient and send it back to the white, ***z,*** node of the Output Layer:

$$\begin{aligned} \frac{\partial Cost}{\partial Z^{(2)}} &= UpstreamGradeint * LocalGradient \\ &= \frac{\partial Cost}{\partial \hat{Y}} * \frac{\partial \hat{Y}}{\partial Z^{(2)}} \\ &= \begin{bmatrix} \frac{\partial Cost}{\partial \hat{y}^{(1)}} & \frac{\partial Cost}{\partial \hat{y}^{(2)}} & \frac{\partial Cost}{\partial \hat{y}^{(3)}} & \frac{\partial Cost}{\partial \hat{y}^{(4)}} \end{bmatrix} * \begin{bmatrix} \frac{\partial y^{(1)}}{\partial z_1^{[2](1)}} & \frac{\partial y^{(2)}}{\partial z_1^{[2](2)}} & \frac{\partial y^{(3)}}{\partial z_1^{[2](3)}} & \frac{\partial y^{(4)}}{\partial z_1^{[2](4)}} \end{bmatrix} \\ &= \begin{bmatrix} 0.167 & -0.072 & -0.078 & 0.182 \end{bmatrix} * \begin{bmatrix} 0.222 & 0.205 & 0.215 & 0.198 \end{bmatrix} \\ &= \begin{bmatrix} 0.037 & -0.015 & -0.017 & 0.036 \end{bmatrix} = \begin{bmatrix} \frac{\partial Cost}{\partial z_1^{[2](1)}} & \frac{\partial Cost}{\partial z_1^{[2](2)}} & \frac{\partial Cost}{\partial z_1^{[2](3)}} & \frac{\partial Cost}{\partial z_1^{[2](4)}} \end{bmatrix} \end{aligned}$$

Fig 81.b. Backpropagation on the neural net with a hidden layer



The neural network is going through the following computations(backward computations are in red):

- The **Upstream Gradient** in this step is:

$$\frac{\partial Cost}{\partial Z^{(2)}} = \begin{bmatrix} 0.037 & -0.015 & -0.017 & 0.036 \end{bmatrix}$$

- The ***three*** **Local Gradients** associated with the $Z^{[2]}$ node are :

1. $\frac{\partial Z^{[2]}}{\partial W^{[2]}} = \left(A^{[1]}\right)^T = \begin{bmatrix} 0.5 & 0.5 & 0.5 \\ 0.646 & 0.6 & 0.668 \\ 0.525 & 0.55 & 0.574 \\ 0.668 & 0.646 & 0.731 \end{bmatrix}$
2. $\frac{\partial Z^{[2]}}{\partial A^{[1]}} = \left(W^{[2]}\right)^T = \begin{bmatrix} 0.1 \\ 0.4 \\ 0.9 \end{bmatrix}$
3. $\frac{\partial Z^{[2]}}{\partial B^{[2]}} = [1]$

- Finally, we'll combine all the upstream gradients with the local ones and send them back to their respective nodes, or store them for the gradient descent update after finishing backpropagation:

1. $W^{[2]}$ is a (1×3) row vector so the derivative of Cost w.r.t $W^{[2]}$ should also be a (1×3) row vector:

$$
\begin{aligned}
\frac{\partial Cost}{\partial W^{[2]}} &= UpstreamGradient \cdot LocalGradient \\
&= \frac{\partial Cost}{\partial Z^{[2]}} \cdot \frac{\partial Z^{[2]}}{\partial W^{[2]}} \\
&= \begin{bmatrix} \frac{\partial Cost}{\partial z_1^{[2](1)}} & \frac{\partial Cost}{\partial z_1^{[2](2)}} & \frac{\partial Cost}{\partial z_1^{[2](3)}} & \frac{\partial Cost}{\partial z_1^{[2](4)}} \end{bmatrix} \cdot \begin{bmatrix} \frac{\partial z^{[2](1)}}{\partial w_1^{[2]}} & \frac{\partial z^{[2](1)}}{\partial w_2^{[2]}} & \frac{\partial z^{[2](1)}}{\partial w_3^{[2]}} \\ \frac{\partial z^{[2](2)}}{\partial w_1^{[2]}} & \frac{\partial z^{[2](2)}}{\partial w_2^{[2]}} & \frac{\partial z^{[2](2)}}{\partial w_3^{[2]}} \\ \frac{\partial z^{[2](3)}}{\partial w_1^{[2]}} & \frac{\partial z^{[2](3)}}{\partial w_2^{[2]}} & \frac{\partial z^{[2](3)}}{\partial w_3^{[2]}} \\ \frac{\partial z^{[2](4)}}{\partial w_1^{[2]}} & \frac{\partial z^{[2](4)}}{\partial w_2^{[2]}} & \frac{\partial z^{[2](4)}}{\partial w_3^{[2]}} \end{bmatrix} \\
&= \begin{bmatrix} 0.037 & -0.015 & -0.017 & 0.036 \end{bmatrix} \cdot \begin{bmatrix} 0.5 & 0.5 & 0.5 \\ 0.646 & 0.6 & 0.668 \\ 0.525 & 0.55 & 0.574 \\ 0.668 & 0.646 & 0.731 \end{bmatrix} \\
&= \begin{bmatrix} 0.024 & 0.023 & 0.025 \end{bmatrix} = \begin{bmatrix} \frac{\partial Cost}{\partial w_1^{[2]}} & \frac{\partial Cost}{\partial w_2^{[2]}} & \frac{\partial Cost}{\partial w_3^{[2]}} \end{bmatrix}
\end{aligned}
$$

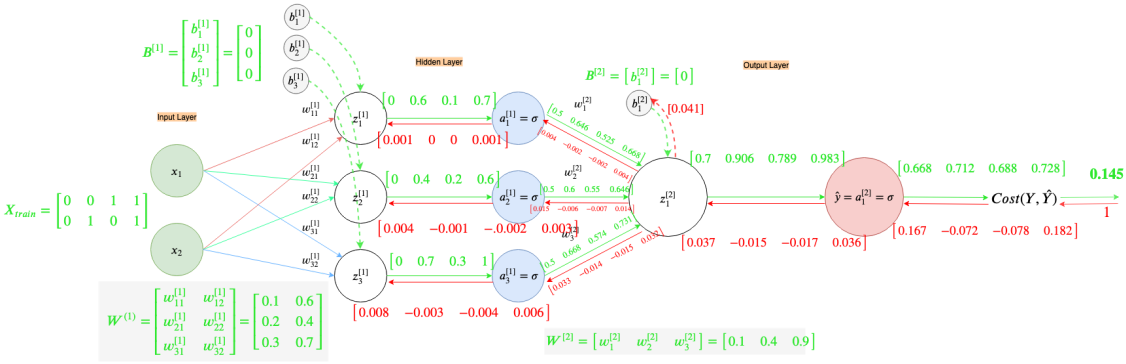
2. $B^{[2]}$ is a (1×1) matrix(*actually, just a scalar*) so the derivative of Cost w.r.t $B^{[2]}$ should also have the shape (1×1):

$$
\begin{aligned}
\frac{\partial Cost}{\partial B^{[2]}} &= \sum UpstreamGradient * LocalGradient \\
&= \sum \frac{\partial Cost}{\partial Z^{[2]}} * \frac{\partial Z^{[2]}}{\partial B^{[2]}} \\
&= \sum \begin{bmatrix} \frac{\partial Cost}{\partial z_1^{[2](1)}} & \frac{\partial Cost}{\partial z_1^{[2](2)}} & \frac{\partial Cost}{\partial z_1^{[2](3)}} & \frac{\partial Cost}{\partial z_1^{[2](4)}} \end{bmatrix} * \begin{bmatrix} \frac{\partial z^{[2](1)}}{\partial b_1^{[2]}} \end{bmatrix} \\
&= \sum \begin{bmatrix} 0.037 & -0.015 & -0.017 & 0.036 \end{bmatrix} * \begin{bmatrix} 1 \end{bmatrix} \\
&= [0.037 - 0.015 - 0.017 + 0.036] \\
&= [0.041] = \begin{bmatrix} \frac{\partial Cost}{\partial b_1^{[2]}} \end{bmatrix}
\end{aligned}
$$

3. $A^{[1]}$ has the shape (3×4) so the derivative of Cost w.r.t $A^{[1]}$ should also have the shape (3×4):

$$
\begin{aligned}
\frac{\partial Cost}{\partial A^{[1]}} &= LocalGradient \cdot UpstreamGradient \\
&= \frac{\partial Z^{[2]}}{\partial A^{[1]}} \cdot \frac{\partial Cost}{\partial Z^{[2]}} \\
&= \begin{bmatrix} \frac{\partial z_1^{[2]}}{\partial a_1^{[1]}} \\ \frac{\partial z_1^{[2]}}{\partial a_2^{[1]}} \\ \frac{\partial z_1^{[2]}}{\partial a_3^{[1]}} \end{bmatrix} \cdot \begin{bmatrix} \frac{\partial Cost}{\partial z_1^{[2](1)}} & \frac{\partial Cost}{\partial z_1^{[2](2)}} & \frac{\partial Cost}{\partial z_1^{[2](3)}} & \frac{\partial Cost}{\partial z_1^{[2](4)}} \end{bmatrix} \\
&= \begin{bmatrix} 0.1 \\ 0.4 \\ 0.9 \end{bmatrix} \cdot \begin{bmatrix} 0.037 & -0.015 & -0.017 & 0.036 \end{bmatrix} \\
&= \begin{bmatrix} 0.004 & -0.002 & -0.002 & 0.004 \\ 0.015 & -0.006 & -0.007 & 0.014 \\ 0.033 & -0.014 & -0.015 & 0.032 \end{bmatrix} = \begin{bmatrix} \frac{\partial Cost}{\partial a_1^{[1](1)}} & \frac{\partial Cost}{\partial a_1^{[1](2)}} & \frac{\partial Cost}{\partial a_1^{[1](3)}} & \frac{\partial Cost}{\partial a_1^{[1](4)}} \\ \frac{\partial Cost}{\partial a_2^{[1](1)}} & \frac{\partial Cost}{\partial a_2^{[1](2)}} & \frac{\partial Cost}{\partial a_2^{[1](3)}} & \frac{\partial Cost}{\partial a_2^{[1](4)}} \\ \frac{\partial Cost}{\partial a_3^{[1](1)}} & \frac{\partial Cost}{\partial a_3^{[1](2)}} & \frac{\partial Cost}{\partial a_3^{[1](3)}} & \frac{\partial Cost}{\partial a_3^{[1](4)}} \end{bmatrix}
\end{aligned}
$$

Fig 81.c. Backpropagation on the neural net with a hidden layer



The neural network is going through the following computations(backward computations are in red):

- The **Upstream Gradient** in this step is:

$$\frac{\partial Cost}{\partial A^{[1]}} = \begin{bmatrix} 0.004 & -0.002 & -0.002 & 0.004 \\ 0.015 & -0.006 & -0.007 & 0.014 \\ 0.033 & -0.014 & -0.015 & 0.032 \end{bmatrix}$$

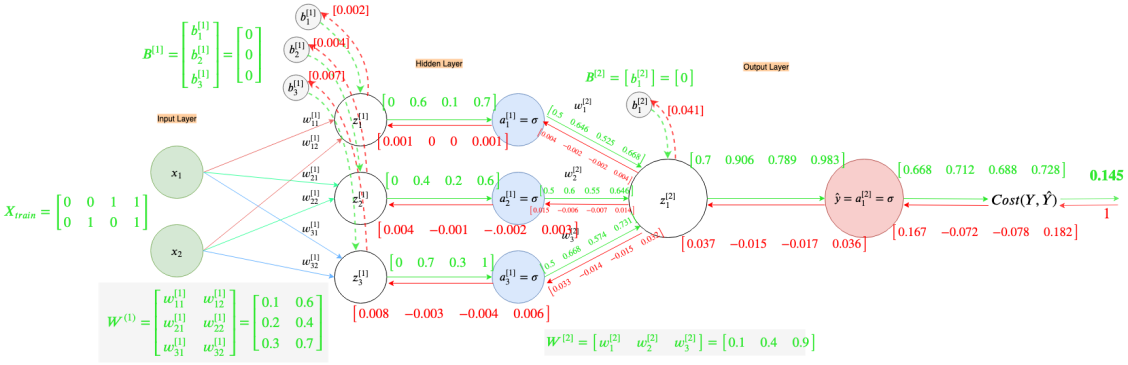
- The **Local Gradient** is the derivative of the sigmoid($a^{[1]}$ nodes):

$$\begin{aligned} \frac{\partial A^{[1]}}{\partial Z^{[1]}} &= A^{[1]}(1 - A^{[1]}) \\ &= \begin{bmatrix} 0.5 & 0.646 & 0.525 & 0.668 \\ 0.5 & 0.6 & 0.55 & 0.646 \\ 0.5 & 0.668 & 0.574 & 0.731 \end{bmatrix} \left(1 - \begin{bmatrix} 0.5 & 0.646 & 0.525 & 0.668 \\ 0.5 & 0.6 & 0.55 & 0.646 \\ 0.5 & 0.668 & 0.574 & 0.731 \end{bmatrix} \right) \\ &= \begin{bmatrix} 0.5 & 0.646 & 0.525 & 0.668 \\ 0.5 & 0.6 & 0.55 & 0.646 \\ 0.5 & 0.668 & 0.574 & 0.731 \end{bmatrix} * \begin{bmatrix} 0.5 & 0.354 & 0.475 & 0.332 \\ 0.5 & 0.4 & 0.45 & 0.354 \\ 0.5 & 0.332 & 0.426 & 0.296 \end{bmatrix} \\ &= \begin{bmatrix} 0.25 & 0.229 & 0.249 & 0.222 \\ 0.25 & 0.24 & 0.248 & 0.229 \\ 0.25 & 0.222 & 0.254 & 0.197 \end{bmatrix} \end{aligned}$$

- Let's combine these two and send them back to the z nodes of the Hidden Layer:

$$\begin{aligned} \frac{\partial Cost}{\partial Z^{[1]}} &= UpstreamGradient * LocalGradient \\ &= \frac{\partial Cost}{\partial A^{[1]}} * \frac{\partial A^{[1]}}{\partial Z^{[1]}} \\ &= \begin{bmatrix} 0.004 & -0.002 & -0.002 & 0.004 \\ 0.015 & -0.006 & -0.007 & 0.014 \\ 0.033 & -0.014 & -0.015 & 0.032 \end{bmatrix} * \begin{bmatrix} 0.25 & 0.229 & 0.249 & 0.222 \\ 0.25 & 0.24 & 0.248 & 0.229 \\ 0.25 & 0.222 & 0.254 & 0.197 \end{bmatrix} \\ &= \begin{bmatrix} 0.001 & 0 & 0 & 0.001 \\ 0.004 & -0.001 & -0.002 & 0.003 \\ 0.008 & -0.003 & -0.004 & 0.006 \end{bmatrix} = \begin{bmatrix} \frac{\partial Cost}{\partial z_1^{[1](1)}} & \frac{\partial Cost}{\partial z_1^{[1](2)}} & \frac{\partial Cost}{\partial z_1^{[1](3)}} & \frac{\partial Cost}{\partial z_1^{[1](4)}} \\ \frac{\partial Cost}{\partial z_2^{[1](1)}} & \frac{\partial Cost}{\partial z_2^{[1](2)}} & \frac{\partial Cost}{\partial z_2^{[1](3)}} & \frac{\partial Cost}{\partial z_2^{[1](4)}} \\ \frac{\partial Cost}{\partial z_3^{[1](1)}} & \frac{\partial Cost}{\partial z_3^{[1](2)}} & \frac{\partial Cost}{\partial z_3^{[1](3)}} & \frac{\partial Cost}{\partial z_3^{[1](4)}} \end{bmatrix} \end{aligned}$$

Fig 81.d. Backpropagation on the neural net with a hidden layer



The neural network is going through the following computations(backward computations are in red):

- The **Upstream Gradient** in this last step is:

$$\frac{\partial Cost}{\partial Z^{[1]}} = \begin{bmatrix} 0.001 & 0 & 0 & 0.001 \\ 0.004 & -0.001 & -0.002 & 0.003 \\ 0.008 & -0.003 & -0.004 & 0.006 \end{bmatrix}$$

- We have propagated the gradients back enough to calculate the weights and biases associated with the hidden layer the **two Local Gradients** are:

1. $\frac{\partial Z^{[1]}}{\partial W^{[1]}} = X_{train}^T = \begin{bmatrix} 0 & 0 \\ 0 & 1 \\ 1 & 0 \\ 1 & 1 \end{bmatrix}$
2. $\frac{\partial Z^{[1]}}{\partial B^{[1]}} = \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix}$

Again, note that we will not claulate the local gradient w.r.t X_{train} as we do not intend to change the training data through our gradient descet update.

- Finally, let's combine the upstream and local gradients:

1. $W^{[1]}$ is a (3×2) matrix so the derivative of Cost w.r.t $W^{[1]}$ should also be of the same shape:

$$\begin{aligned} \frac{\partial Cost}{\partial W^{[1]}} &= UpstreamGradient \cdot LocalGradient \\ &= \frac{\partial Cost}{\partial Z^{[1]}} \cdot \frac{\partial Z^{[1]}}{\partial W^{[1]}} \\ &= \begin{bmatrix} 0.001 & 0 & 0 & 0.001 \\ 0.004 & -0.001 & -0.002 & 0.003 \\ 0.008 & -0.003 & -0.004 & 0.006 \end{bmatrix} \cdot \begin{bmatrix} 0 & 0 \\ 0 & 1 \\ 1 & 0 \\ 1 & 1 \end{bmatrix} \\ &= \begin{bmatrix} 0.001 & 0.001 \\ 0.001 & 0.002 \\ 0.002 & 0.003 \end{bmatrix} = \begin{bmatrix} \frac{\partial Cost}{\partial w_{11}^{[1]}} & \frac{\partial Cost}{\partial w_{12}^{[1]}} \\ \frac{\partial Cost}{\partial w_{21}^{[1]}} & \frac{\partial Cost}{\partial w_{22}^{[1]}} \\ \frac{\partial Cost}{\partial w_{31}^{[1]}} & \frac{\partial Cost}{\partial w_{32}^{[1]}} \end{bmatrix} \end{aligned}$$

2. The bias vector, $B^{[1]}$, has the shape (3×1) so its derivative should also have the same shape:

$$\begin{aligned} \frac{\partial Cost}{\partial B^{[1]}} &= \sum (UpstreamGradient \cdot LocalGradient, \, axis = 1) \\ &= \sum \left(\frac{\partial Cost}{\partial Z^{[1]}} \odot \frac{\partial Z^{[1]}}{\partial B^{[1]}}, \, axis = 1 \right) \\ &= \sum \left(\begin{bmatrix} \frac{\partial Cost}{\partial z_1^{[1](1)}} & \frac{\partial Cost}{\partial z_1^{[1](2)}} & \frac{\partial Cost}{\partial z_1^{[1](3)}} & \frac{\partial Cost}{\partial z_1^{[1](4)}} \\ \frac{\partial Cost}{\partial z_2^{[1](1)}} & \frac{\partial Cost}{\partial z_2^{[1](2)}} & \frac{\partial Cost}{\partial z_2^{[1](3)}} & \frac{\partial Cost}{\partial z_2^{[1](4)}} \\ \frac{\partial Cost}{\partial z_3^{[1](1)}} & \frac{\partial Cost}{\partial z_3^{[1](2)}} & \frac{\partial Cost}{\partial z_3^{[1](3)}} & \frac{\partial Cost}{\partial z_3^{[1](4)}} \end{bmatrix} \odot \begin{bmatrix} \frac{\partial Z^{[1]}}{\partial b_1^{[1]}} \\ \frac{\partial Z^{[1]}}{\partial b_2^{[1]}} \\ \frac{\partial Z^{[1]}}{\partial b_3^{[1]}} \end{bmatrix}, \, axis = 1 \right) \end{aligned}$$

$$
\begin{aligned}
&= \sum\left(\begin{bmatrix} 0.001 & 0 & 0 & 0.001 \\ 0.004 & -0.001 & -0.002 & 0.003 \\ 0.008 & -0.003 & -0.004 & 0.006 \end{bmatrix} \odot \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix}, axis = 1\right) \\
&= \sum\left(\begin{bmatrix} 0.001 & 0 & 0 & 0.001 \\ 0.004 & -0.001 & -0.002 & 0.003 \\ 0.008 & -0.003 & -0.004 & 0.006 \end{bmatrix}, axis = 1\right) \\
&= \begin{bmatrix} 0.001 + 0 + 0 + 0.001 \\ 0.004 - 0.001 - 0.002 + 0.003 \\ 0.008 - 0.003 - 0.004 + 0.006 \end{bmatrix} \\
&= \begin{bmatrix} 0.002 \\ 0.004 \\ 0.007 \end{bmatrix} = \begin{bmatrix} \frac{\partial Cost}{\partial b_1^{[1]}} \\ \frac{\partial Cost}{\partial b_2^{[1]}} \\ \frac{\partial Cost}{\partial b_3^{[1]}} \end{bmatrix}
\end{aligned}
$$

"⊙" is element-wise product, also known as Hadamard product
"axis=1" here means to sum across rows, it is a NumPy convention. ("axis=0" means sum across coulums.)

Fig 81.e. Backpropagation on the neural net with a hidden layer

Whew! That was a lot, but it did a great deal to improve our understanding. Let's perform the gradient descent update:

Gradient Descent Update

To calculate new weights and biases we will move in the negative direction of the gradient.

Recall, our learning rate is $\alpha = 1$.

New $W^{[2]}$:

$$
\begin{aligned}
W^{[2]} &= W^{[2]} - \alpha \frac{\partial Cost}{\partial W^{[2]}} \\
&= \begin{bmatrix} 0.1 & 0.4 & 0.9 \end{bmatrix} - \left(1 * \begin{bmatrix} 0.024 & 0.023 & 0.025 \end{bmatrix}\right) \\
&= \begin{bmatrix} 0.1 & 0.4 & 0.9 \end{bmatrix} - \begin{bmatrix} 0.024 & 0.023 & 0.025 \end{bmatrix} \\
&= \begin{bmatrix} (0.1 - 0.024) & (0.4 - 0.023) & (0.9 - 0.025) \end{bmatrix} \\
&= \begin{bmatrix} 0.076 & 0.377 & 0.875 \end{bmatrix}
\end{aligned}
$$

New $B^{[2]}$:

$$
\begin{aligned}
B^{[2]} &= B^{[2]} - \alpha \frac{\partial Cost}{\partial B^{[2]}} \\
&= \begin{bmatrix} 0 \end{bmatrix} - (1 * [0.041]) \\
&= \begin{bmatrix} -0.041 \end{bmatrix}
\end{aligned}
$$

New $W^{[1]}$:

$$
\begin{aligned}
W^{[1]} &= W^{[1]} - \alpha \frac{\partial Cost}{\partial W^{[1]}} \\
&= \begin{bmatrix} 0.1 & 0.6 \\ 0.2 & 0.4 \\ 0.3 & 0.7 \end{bmatrix} - \left(1 * \begin{bmatrix} 0.001 & 0.001 \\ 0.001 & 0.002 \\ 0.002 & 0.003 \end{bmatrix}\right)
\end{aligned}
$$

$$
\begin{bmatrix} 0.1 & 0.6 \end{bmatrix} \quad \begin{bmatrix} 0.001 & 0.001 \end{bmatrix}
$$

$$= \begin{bmatrix} 0.2 & 0.4 \\ 0.3 & 0.7 \end{bmatrix} - \begin{bmatrix} 0.001 & 0.002 \\ 0.002 & 0.003 \end{bmatrix}$$
$$= \begin{bmatrix} (0.1 - 0.001) & (0.6 - 0.001) \\ (0.2 - 0.001) & (0.4 - 0.002) \\ (0.3 - 0.002) & (0.7 - 0.003) \end{bmatrix}$$
$$= \begin{bmatrix} 0.099 & 0.599 \\ 0.199 & 0.398 \\ 0.298 & 0.697 \end{bmatrix}$$

New $B^{[1]}$:

$$B^{[1]} = B^{[1]} - \alpha \frac{\partial Cost}{\partial B^{[1]}}$$
$$= \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix} - \left(1 * \begin{bmatrix} 0.002 \\ 0.004 \\ 0.007 \end{bmatrix} \right)$$
$$= \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix} - \begin{bmatrix} 0.002 \\ 0.004 \\ 0.007 \end{bmatrix}$$
$$= \begin{bmatrix} -0.002 \\ -0.004 \\ -0.007 \end{bmatrix}$$

Fig 82. Gradient descent update for the neural net with a hidden layer

At this point, I would encourage all readers to perform one training iteration themselves. The resultant gradients should be approximately(rounded to 3 decimal places):

Derivatives w.r.t Cost on 2nd training Iteration

1. $\frac{\partial Cost}{\partial W^{[2]}} = \begin{bmatrix} 0.023 & 0.022 & 0.024 \end{bmatrix}$
2. $\frac{\partial Cost}{\partial B^{[2]}} = \begin{bmatrix} 0.039 \end{bmatrix}$
3. $\frac{\partial Cost}{\partial W^{[1]}} = \begin{bmatrix} 0 & 0 \\ 0.001 & 0.002 \\ 0.002 & 0.003 \end{bmatrix}$
4. $\frac{\partial Cost}{\partial B^{[1]}} = \begin{bmatrix} 0.001 \\ 0.003 \\ 0.007 \end{bmatrix}$

Fig 83. Derivatives computed during 2nd training iteration

After 5000 epochs the Cost steadily decreases to about ***0.0009*** and we get the following Learning Curve and Decision Boundary:

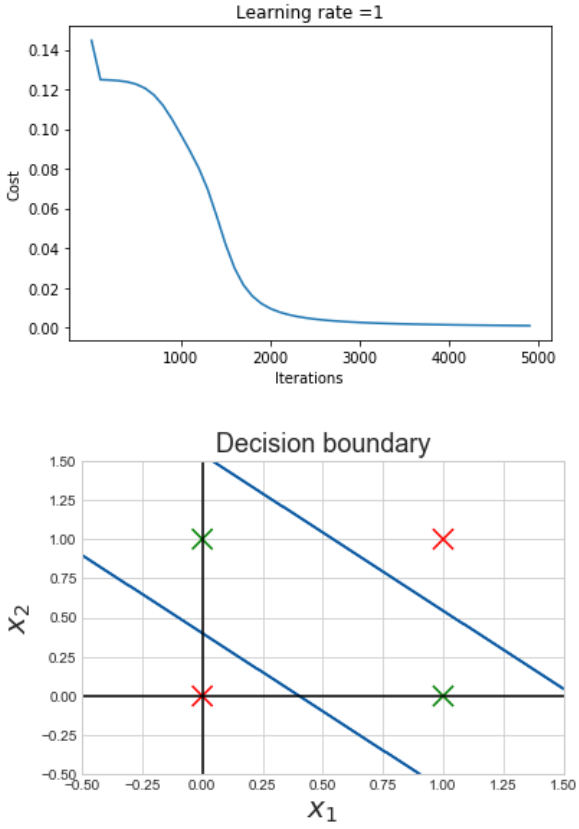


Fig 84. Learning Curve and Decision boundary of the neural net with one hidden layer

Let's also visualize where the decision of the neural network changes from 0(red) to 1(green):

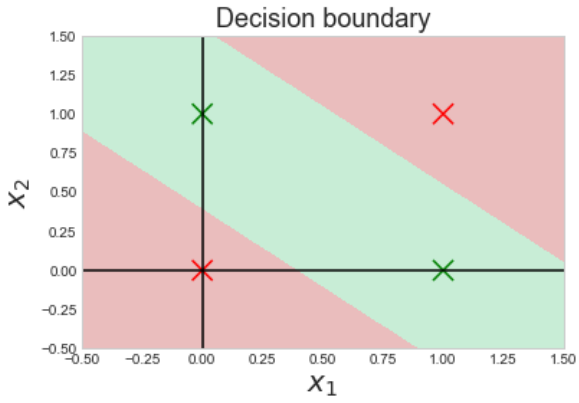


Fig 85. Shaded decision boundary of the neural net with one hidden layer

This shows that the neural network has in fact learned where to fire-up(output 1) and where to lay dormant(output 0).

If we add another hidden layer with maybe 2 or 3 sigmoid neurons we can get an even more complex decision boundary that may fit our data even more tightly, but let's leave that for the coding section.

Before we conclude this section I want to answer some remaining questions:

1- So, which one is better Feature Engineering or a Deep Neural Network?

Well, the answer depends on many factors. Generally, if we have a lot of training data we can just use a deep neural net to achieve acceptable accuracy, but if data is limited we may need to perform some feature engineering to extract more performance out of our neural network. As you saw in the feature engineering example above, to make good feature crosses one needs to have intimate knowledge of the dataset they are working with.

Feature engineering along with a deep neural network is a powerful combination.

2- How to count the number of layers in a Neural Network?

By convention, we don't count layers without tunable weights and bias. Therefore, though the input layer is a separate "layer" we don't count it when specifying the depth of a neural network.

So, our last example was a "*2 layer neural network*" (one hidden + output layer) and all the examples before it just a "*1 layer neural network*" (output layer, only).

3- Why use Activation Functions?

Activation functions are nonlinear functions and add nonlinearity to the neurons. The feature crosses are a result of stacking the activation functions in hidden layers. The combination of a bunch of activation functions thus results in a complex non-linear decision boundary. In this blog, we used the sigmoid/logistic activation function, but there are many other types of activation functions(*ReLU being a popular choice for hidden layers*) each providing a certain benefit. ***The choice of the activation function is also a hyper-parameter when creating neural networks.***

Without activations functions to add nonlinearity, no matter how many linear functions we stack up the result of them will still be linear.Consider the following:

Let's consider the following equations, where x_1 and x_2 are in the inputs:

$$A = w_1 x_1 + w_2 x_2$$

$$B = w_3 A$$

$$C = w_4 B$$

So, the B and C equations are like stacking linear layers on top of a linear layer A. Let's see what the final layer, layer C, is calculating:

First, we'll expand out B:

$$\begin{aligned} B &= w_3 A \\ &= w_3(w_1 x_1 + w_2 x_2) \\ &= w_3 w_1\ x_1 + w_3 w_2\ x_2 \end{aligned}$$

Now we'll expand out C:

$$\begin{aligned} C &= w_4 B \\ &= w_4(w_3 w_1\ x_1 + w_3 w_2\ x_2) \\ &= w_4 w_3 w_1\ x_1 + w_4 w_3 w_2\ x_2 \end{aligned}$$

Notice that the inputs x_1 and x_2 aren't being crossed or squared at all, the final function C is still just a linear function:

$$\begin{aligned} C &= w_4 w_3 w_1\ x_1 + w_4 w_3 w_2\ x_2 \\ &= \text{someWeight} * x_1 + \text{someWeight} * x_2 \end{aligned}$$

Fig 86. Showing that stacking linear layers/functions results in a linear layer/function

You may use any nonlinear function as an activation function. Some researchers have used even ***cos*** and **sin** functions. Preferably the activation function should be a continuous i.e. no breaks in the domain of the function.

4- Why Random Initialization of Weights?

This question is much easier to answer now. Note that if we had set all the weights in a layer to the same value than the gradient that passes through each node would be the same. In short, all the nodes in the layer would learn the same feature about the data. Setting the weights to ***random values helps in breaking the symmetry of weights*** so that each node in a layer has the opportunity to learn a unique aspect of the training data

There are many ways to set weights randomly in neural networks. For small neural networks, it is ok to set the weights to small random values. For larger networks, we tend to use "Xavier" or "He" initialization methods(*will be in the coding section*). Both these methods still set weights to random values but control their variance. *For now, its suffice to say use these methods when the network does not seem to converge and the Cost becomes static or reduces very slowly when using the "plain" method of setting weights to small random values.* Weight initialization is an active research area and will be a topic for a future "Nothing but Numpy" blog.

Biases can be randomly initialized, too. But in practice, it does not seem to have much of an effect on the performance of a neural network. Perhaps this is because the number of bias terms in a neural network is much fewer than the weights.

The type of neural network we created here is called a "***fully-connected feedforward network***" or simply a "***feedforward network***".

This concludes Part I.

Part II: Coding a Modular Neural Network

The implementation in this part follows OOP principals.

Let's first see the **Linear Layer** class. The constructor takes as arguments: the shape of the data coming in(`input_shape`), the number of neurons the layer outputs(`n_out`) and what type of random weight initialization need to be performed(`ini_type="plain"`, default is "plain" which is just small random gaussian numbers).

The `initialize_parameters` is a helper function used to define weights and bias. We'll look at it separately, later.

Linear Layer implements the following functions:

- `forward(A_prev)` : This function allows the linear layer to take in activations from the previous layer(the input data can be seen as activations from the input layer) and performs the linear operation on them.
- `backward(upstream_grad)`: This function computes the derivative of Cost w.r.t weights, bias, and activations from the previous layer(`dW`, `db`&`dA_prev`, respectively)
- `update_params(learning_rate=0.1)` : This function performs the gradient descent update on weights and bias using the derivatives computed in the `backward` function. The default learning rate(α) is 0.1

```
import numpy as np  # import numpy library
from util.paramInitializer import initialize_parameters  # import function to initialize weights and biases


class LinearLayer:
    """
        This Class implements all functions to be executed by a linear layer
        in a computational graph
        Args:
            input_shape: input shape of Data/Activations
            n_out: number of neurons in layer
            ini_type: initialization type for weight parameters, default is "plain"
                      Opitons are: plain, xavier and he
        Methods:
            forward(A_prev)
            backward(upstream_grad)
            update_params(learning_rate)
    """

    def __init__(self, input_shape, n_out, ini_type="plain"):
        """
        The constructor of the LinearLayer takes the following parameters
        Args:
            input_shape: input shape of Data/Activations
            n_out: number of neurons in layer
            ini_type: initialization type for weight parameters, default is "plain"
        """

        self.m = input_shape[1]  # number of examples in training data
        # `params` store weights and bias in a python dictionary
        self.params = initialize_parameters(input_shape[0], n_out, ini_type)  # initialize weights and bias
        self.Z = np.zeros((self.params['W'].shape[0], input_shape[1]))  # create space for resultant Z output

    def forward(self, A_prev):
        """
        This function performs the forwards propagation using activations from previous layer
        Args:
            A_prev:  Activations/Input Data coming into the layer from previous layer
        """

        self.A_prev = A_prev  # store the Activations/Training Data coming in
        self.Z = np.dot(self.params['W'], self.A_prev) + self.params['b']  # compute the linear function

    def backward(self, upstream_grad):
        """
        This function performs the back propagation using upstream gradients
        Args:
            upstream_grad: gradient coming in from the upper layer to couple with local gradient
        """

        # derivative of Cost w.r.t W
        self.dW = np.dot(upstream_grad, self.A_prev.T)

        # derivative of Cost w.r.t b, sum across rows
        self.db = np.sum(upstream_grad, axis=1, keepdims=True)

        # derivative of Cost w.r.t A_prev
        self.dA_prev = np.dot(self.params['W'].T, upstream_grad)

    def update_params(self, learning_rate=0.1):
        """
        This function performs the gradient descent update
        Args:
            learning_rate: learning rate hyper-param for gradient descent, default 0.1
        """

        self.params['W'] = self.params['W'] - learning_rate * self.dW  # update weights
```

```
        self.params['W'] = self.params['W'] - learning_rate * self.dW  # update weights
        self.params['b'] = self.params['b'] - learning_rate * self.db  # update bias(es)
```

Fig 87. Linear Layer Class

Now let's see the **Sigmoid Layer** class, its constructor takes in as an argument the shape of data coming in(`input_shape`) from a Linear Layer preceding it.

Sigmoid Layer implements the following functions:

- `forward(Z)` : This function allows the sigmoid layer to take in the linear computations(`Z`) from the previous layer and perform the sigmoid activation on them.
- `backward(upstream_grad)`: This function computes the derivative of Cost w.r.t $\boldsymbol{Z}$(`dZ`).

```
import numpy as np  # import numpy library

class SigmoidLayer:
    """
    This file implements activation layers
    inline with a computational graph model
    Args:
        shape: shape of input to the layer
    Methods:
        forward(Z)
        backward(upstream_grad)
    """

    def __init__(self, shape):
        """
        The consturctor of the sigmoid/logistic activation layer takes in the following arguments
        Args:
            shape: shape of input to the layer
        """
        self.A = np.zeros(shape)  # create space for the resultant activations

    def forward(self, Z):
        """
        This function performs the forwards propagation step through the activation function
        Args:
            Z: input from previous (linear) layer
        """
        self.A = 1 / (1 + np.exp(-Z))  # compute activations

    def backward(self, upstream_grad):
        """
        This function performs the  back propagation step through the activation function
        Local gradient => derivative of sigmoid => A*(1-A)
        Args:
            upstream_grad: gradient coming into this layer from the layer above
        """
        # couple upstream gradient with local gradient, the result will be sent back to the Linear layer
        self.dZ = upstream_grad * self.A*(1-self.A)
```

Fig 88. Sigmoid Activation Layer class

The `initialize_parameters` function is used only in the Linear Layer to set weights and biases. Using the size of input(`n_in`) and output(`n_out`) it defines the shape the weight matrix and bias vector need to be in. This helper function then returns both the weight(W) and bias(b) in a python dictionary to the respective Linear Layer.

```
def compute_cost(Y, Y_hat):
    """
    This function computes and returns the Cost and its derivative.
    The is function uses the Squared Error Cost function -> (1/2m)*sum(Y - Y_hat)^.2
    Args:
        Y: labels of data
        Y_hat: Predictions(activations) from a last layer, the output layer
    Returns:
        cost: The Squared Error Cost result
        dY_hat: gradient of Cost w.r.t the Y_hat
    """
    m = Y.shape[1]

    cost = (1 / (2 * m)) * np.sum(np.square(Y - Y_hat))
    cost = np.squeeze(cost)  # remove extraneous dimensions to give just a scalar

    dY_hat = -1 / m * (Y - Y_hat)  # derivative of the squared error cost function

    return cost, dY_hat
```

Fig 89. Helper function to set weights and bias

Finally, the Cost function `compute_cost(Y, Y_hat)` takes as argument the activations from the last layer(`Y_hat`) and the true labels(`Y`) and computes and returns the Squared Error Cost(`cost`) and its derivative(`dY_hat`).

```
def compute_cost(Y, Y_hat):
    """
    This function computes and returns the Cost and its derivative.
    The is function uses the Squared Error Cost function -> (1/2m)*sum(Y - Y_hat)^.2
    Args:
        Y: labels of data
        Y_hat: Predictions(activations) from a last layer, the output layer
    Returns:
        cost: The Squared Error Cost result
        dY_hat: gradient of Cost w.r.t the Y_hat
    """
    m = Y.shape[1]

    cost = (1 / (2 * m)) * np.sum(np.square(Y - Y_hat))
    cost = np.squeeze(cost)  # remove extraneous dimensions to give just a scalar

    dY_hat = -1 / m * (Y - Y_hat)  # derivative of the squared error cost function

    return cost, dY_hat
```

Fig 90. Function to compute Squared Error Cost and derivative.

*At this point, you should open the **2_layer_toy_network_XOR** Jupyter notebook from this **repository** in a separate window and go over this blog and the notebook side-by-side.*

Now we are ready to create our neural network. Let's use the architecture defined in *Figure 77* for XOR data.

```
# define training constants
learning_rate = 1
number_of_epochs = 5000

np.random.seed(48) # set seed value so that the results are reproduceable
                   # (weights will now be initailzaed to the same pseudo-random numbers, each time)


# Our network architecture has the shape:
#                   (input)--> [Linear->Sigmoid] -> [Linear->Sigmoid] -->(output)

#------ LAYER-1 ----- define hidden layer that takes in training data
Z1 = LinearLayer(input_shape=X_train.shape, n_out=3, ini_type='xavier')
A1 = SigmoidLayer(Z1.Z.shape)

#------ LAYER-2 ----- define output layer that take is values from hidden layer
Z2= LinearLayer(input_shape=A1.A.shape, n_out=1, ini_type='xavier')
A2= SigmoidLayer(Z2.Z.shape)
```

Fig 91. Defining the layers and training parameters

Now we can start the main training loop:

```
costs = [] # initially empty list, this will store all the costs after a certian number of epochs

# Start training
for epoch in range(number_of_epochs):

    # ------------------------- forward-prop -------------------------
    Z1.forward(X_train)
    A1.forward(Z1.Z)

    Z2.forward(A1.A)
    A2.forward(Z2.Z)

    # ---------------------- Compute Cost ----------------------------
    cost, dA2 = compute_cost(Y=Y_train, Y_hat=A2.A)

    # print and store Costs every 100 iterations.
    if (epoch % 100) == 0:
        #print("Cost at epoch#" + str(epoch) + ": " + str(cost))
        print("Cost at epoch#{}: {}".format(epoch, cost))
        costs.append(cost)

    # ------------------------- back-prop ----------------------------
    A2.backward(dA2)
    Z2.backward(A2.dZ)

    A1.backward(Z2.dA_prev)
    Z1.backward(A1.dZ)

    # ----------------------- Update weights and bias ----------------
    Z2.update_params(learning_rate=learning_rate)
    Z1.update_params(learning_rate=learning_rate)
```

Fig 92. The training loop

Running the loop in the notebook we see that the Cost decreases to about 0.0009 after 4900 epochs

```
...
Cost at epoch#4600: 0.001018305488651183
Cost at epoch#4700: 0.000983783942124411
Cost at epoch#4800: 0.0009514180100050973
Cost at epoch#4900: 0.0009210166430616655
```

The Learning curve and Decision Boundaries look as follows:

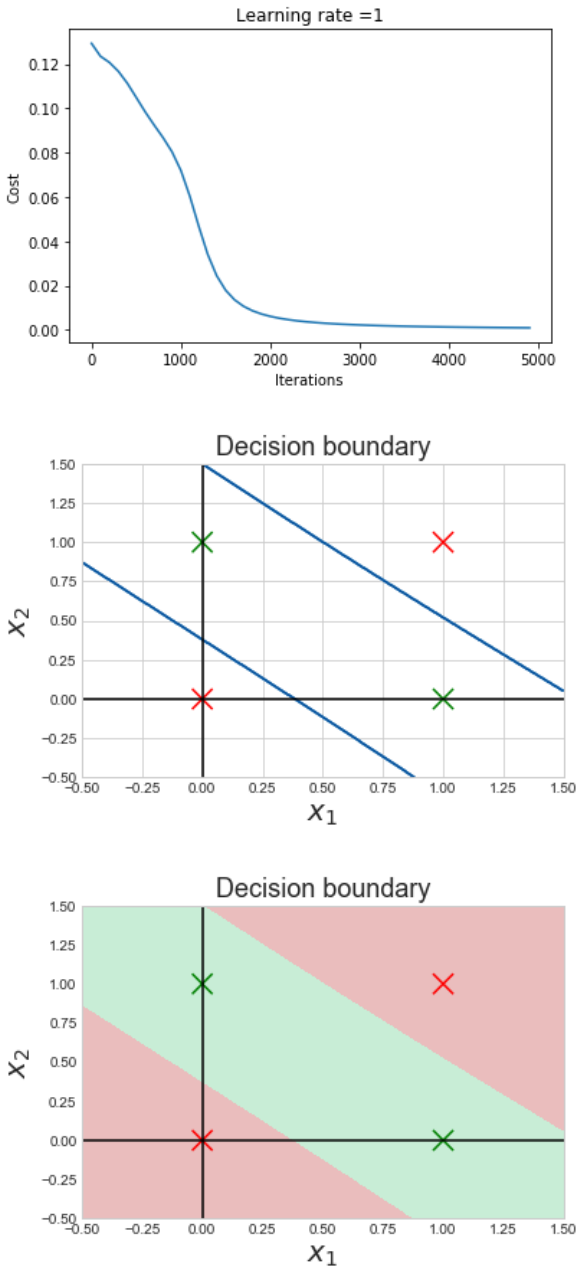


Fig 93. The Learning Curve, Decision Boundary, and Shaded Decision Boundary.

The predictions our trained neural network produces are accurate.

```
The predicted outputs:
 [[ 0.  1.  1.  0.]]
The accuracy of the model is: 100.0%
```

Make sure to check out the other notebooks in the ***repository***. We'll be building upon the things we learned in this blog in future Nothing but NumPy blogs, therefore, it would behoove you to create the layer classes from memory as an exercise and try recreating the OR gate example from ***Part I***.

This concludes the blog. I hope you enjoyed.

For any questions feel free to reach out to me on twitter @RafayAK

This blog would not have been possible without following resources and people:

- Andrej Karpathy's (@karpathy) Stanford course
- Christopher Olah's (@ch402) blogs
- Andrew Trask's (@iamtrask) blogs
- Andrew Ng (@AndrewYNg) and his Coursera courses on deep learningand machine learning

- Terence Parr (@the_antlr_guy) and Jeremy Howard (@jeremyphoward)(https://explained.ai/matrix-calculus/index.html)
- Ian Goodfellow (@goodfellow_ian) and his amazing book
- Finally, Hassan-uz-Zaman (@OKidAmnesiac) and Hassan Tauqeer for invaluable feedback.

Original. Reposted with permission.

Related:

- Neural Networks with Numpy for Absolute Beginners: Introduction
- Building Convolutional Neural Network using NumPy from Scratch
- Artificial Neural Network Implementation using NumPy and Image Classification

What do you think?

26 Responses

Upvote	Funny	Love	Surprised	Angry	Sad
14	0	10	2	0	0

6 Comments **KDnuggets** **Disqus' Privacy Policy** **Login**

Recommend 1 Tweet Share **Sort by Best**

Join the discussion…

LOG IN WITH OR SIGN UP WITH DISQUS

Name

Luiz Abbadia • a year ago

Oh my Lord! Touched everything!!! (that I currently know) a worderful lesson composed of several normal lessons, thanks for this page !!!!

1 • Reply • Share ›

Rafay Khan Luiz Abbadia • a year ago

I'm glad you enjoyed it Luiz

• Reply • Share ›

John • 8 months ago • edited

Hi. Thanks for this nice tutorial/blog post. I've enjoyed it a lot.
Fig 89 should be: helper function to set weights and bias, however, it is also the cost calculation function, as in Fig 90. I saw it in your github repo, thought you'd like to update it above.
Thanks again.

• Reply • Share ›

Yev • 8 months ago

Ah finally someone explained the math behind all of that code nonsense =)) Thank you, very well written. I especially enjoyed those bits where you didn't skip the calculation parts and rather jumped straight into it.

This was very much needed and exactly what I was looking for. Please make more of these types of blogs, with a hands on explanation of the math behind the things.

I'd love to see the math behind convolution neural network for object detection.

• Reply • Share ›

Rafay Khan • a year ago

Author here.
As I am fleshing out this series there have a few updates to the Github repo, following is the new link:
https://github.com/RafayAK/...

• Reply • Share ›

Alex Smola • a year ago

If you want to do this and much more in NumPy check out what a framework will do.
http://numpy.d2l.ai and for documentation see http://numpy.mxnet.io

• Reply • Share ›

Subscribe Add Disqus to your siteAdd DisqusAdd Do Not Sell My Data

---

<= Previous post
Next post =>

Top Stories Past 30 Days

Most Popular

1. **How to become a Data Scientist: a step-by-step guide**
2. **DIY Election Fraud Analysis Using Benford's Law**
3. **The Best Data Science Certification Youve Never Heard Of**
4. **Top Python Libraries for Data Science, Data Visualization & Machine Learning**
5. **How to Acquire the Most Wanted Data Science Skills**
6. **Learn to build an end to end data science project**
7. **How to Get Into Data Science Without a Degree**

Most Shared

1. **Top Python Libraries for Data Science, Data Visualization & Machine Learning**
2. **Top 5 Free Machine Learning and Deep Learning eBooks Everyone should read**
3. **How to Explain Key Machine Learning Algorithms at an Interview**
4. **Pandas on Steroids: End to End Data Science in Python with Dask**
5. **From Y=X to Building a Complete Artificial Neural Network**
6. **How to Get Into Data Science Without a Degree**
7. **How to Acquire the Most Wanted Data Science Skills**

Latest News

- Is Your Machine Learning Model Likely to Fail?
- The 4 Stages of Being Data-driven for Real-life Businesses
- Learn Deep Learning with this Free Course from Yann Lecun
- How to Know if a Neural Network is Right for Your Machi...
- Cartoon: Thanksgiving and Turkey Data Science
- Better data apps with Streamlit's new layout options

Top Stories
Last Week

Most Popular

1. **How to Get Into Data Science Without a Degree**

2. **Top Python Libraries for Deep Learning, Natural Language Processing & Computer Vision**
3. **Is Data Science for Me? 14 Self-examination Questions to Consider**
4. **Facebook Open Sourced New Frameworks to Advance Deep Learning Research**
5. **AI and Automation meets BI**
6. **How to Acquire the Most Wanted Data Science Skills**
7. **Learn to build an end to end data science project**

Most Shared

1. **How to Get Into Data Science Without a Degree**
2. **Top Python Libraries for Deep Learning, Natural Language Processing & Computer Vision**
3. **Facebook Open Sourced New Frameworks to Advance Deep Learning Research**
4. **5 Most Useful Machine Learning Tools every lazy full-stack data scientist should use**
5. **How Machine Learning Works for Social Good**
6. **Is Data Science for Me? 14 Self-examination Questions to Consider**
7. **Adversarial Examples in Deep Learning – A Primer**

More Recent Stories

- Better data apps with Streamlit's new layout options
- Essential Math for Data Science: Integrals And Area Under The ...
- How to Incorporate Tabular Data with HuggingFace Transformers
- Simple Python Package for Comparing, Plotting & Evaluatin...
- TabPy: Combining Python and Tableau
- Fraud through the eyes of a machine
- How Data Professionals Can Add More Variation to Their Resumes
- Top Stories, Nov 16-22: How to Get Into Data Science Without a...
- 15 Exciting AI Project Ideas for Beginners
- Know-How to Learn Machine Learning Algorithms Effectively
- The Rise of the Machine Learning Engineer
- Computer Vision at Scale With Dask And PyTorch
- How Machine Learning Works for Social Good
- Top 6 Data Science Programs for Beginners
- Adversarial Examples in Deep Learning – A Primer
- How Data Scientists Can Avoid 'Lost in Translation'...
- Cellular Automata in Stream Learning
- The top courses for aspiring data scientists

- AI and Automation meets BI
- Compute Goes Brrr: Revisiting Sutton's Bitter Lesson for AI