$NNs\ MATH$

$NOTE: I\ am\ trying\ to\ teach\ myself\ a\ NN\ from\ scratch\ using\ just\ maths,$
$and\ deepen\ my\ understanding\ of\ it\ from\ fundamental\ layer$
$and\ this\ is\ what\ my\ basic\ understanding\ is,\ which\ might\ be\ (and\ I\ am\ sure\ in\ ways\ is)$
$wrong,\ but\ feel\ free\ to\ use\ this\ if\ it\ helps$

1) $Defining\ a\ mathematical\ Perceptron:$

$Let\ our\ inputs\ be\ x1,\ x2,\ x3,\ x4, \ldots xn$

$And\ our\ output\ be\ y$

$Now\ our\ Perceptron's\ value\ will\ be:$

$$\sum_{i=1}^{n} xi * wi\ +\ bi$$

$Then\ we\ apply\ the\ non\ linearity\ to\ it,\ we\ choose\ ReLu\ here$

$Let\ f(x)\ =\ Max(0, x)\ \ =\ ReLu(x)$

$After\ applying\ non\ Linearity:$

$$ReLu\left(\sum_{i=1}^{n} xi * wi\ +\ bi\right)$$

$Why\ do\ we\ need\ a\ non\ linear\ activation\ function?$

$Because\ the\ data\ is\ complex!\ And\ not\ necessarily\ always\ linear,\ we\ use\ the\ linear$
$functions\ as\ legos\ to\ build\ a\ complex\ non\ linear\ function$

2) $A\ Neural\ network\ is\ just\ a\ multi\ Layer\ Perceptron\ \left(NNs\ \rightarrow\ MLP\right)$

$Where\ n(x)\ represent\ the\ number\ of\ input\ layers$

and then we can basically have arbitrary number of layers and depth to it

any layer that is not an input layer, or an output layer is a hidden layer

The more hidden layers the more Activation functions we can have and basically more legos to make a more complex curve.

Basically

$$Number\ of\ Layers \propto Complexity\ of\ curve$$

Now for each perceptron we have the calculations refrenced in (1)

therefore we represent them using matrices, cause that helps us ease the calculations and visualize them simply

for each perceptron in a layer we calclate zi (*where i is in range* (1, *total perceptrons in layer*))

$$\begin{pmatrix} x1 & & & & & \\ x2 & & & & & \\ x3 & * w1 & w2 & w3 & .. & wn \\ .. & & & & & \\ xn & & & & & \end{pmatrix} + \begin{pmatrix} b1 \\ b2 \\ b3 \\ .. \\ bn \end{pmatrix} = zi$$

Then we apply the activation function

$$ReLu(zi)$$

and we repeat this process for each perceptron of each layer until we reach the output nodes

Also a good way to visualize how different ReLus form a complex curve it to think of each ReLu output from previous layer as the input for next Layer and when we finally add them, we basically get the curve using the principle of super position!

The key idea is super position of inputs from previous layers

watch StatQuest's video on activation function to visualize this better

3) *Complexity of a n dimentional curve*:

Now, all our inputs are $x1, x2, x3, x4... xn$

I used to think, how do they reduce an n dimentional curve to a 2 *dimentional one and then compute the loss and etc.*

The truth is that they don't. If you have two inputs x your input space is two dimentional and your output space adding one results in 3 (2 + 1) *dimentional space. We can calculate loss and perform all the other stuf in an n dimentional space and don't necessarily need a* 2 *dimentional space.*

In general, if we have n inputs (in form of $x1, x2, ..., xn$*) and m outputs (in form of* $y1, y2, ..., ym$*), we have a* $(m + n)$ *dimentional space.*

4) *Loss function*

now for a simplified example lets say we are construting a linear function to determine the price of a house (y)

here $x1$ *is out input say the area of the house*

so we take data points and plot them on a graph

and try to minimize the loss:

the loss is equivalent to:

$$Average\ Loss = \left(\sum_{i=1}^{n} (yi - p(xi))^2 \right) / n = AL$$

here $p(xi)$ *represents the predicted value on the linear line*

and now we want to minimize the Average Loss function

5) *Gradient Descent and back prop*

But the function has so many weights and biases, so here we use partial derivates to compute it easily and minimize the Average loss function

now we take partial derivate with respect to each weight (wi) & bias (bi)

$\partial AL/\partial wi$ & $\partial AL/\partial bi$
$for\ i \in [1, n]$

The way partial derivative works is on an assumption that all the other variables have achived their optimum value and are constants.
This is not a specific property of NNs, but of the math involved in partial derivatives in general.

side NOTE: We usually use chain rules to compute partial derivatives

Now when we have the partial derivatives of all our weights, biases and activation functions we need to compute them

It's called backpropagation because it involves going "backwards" through the layers of a computer program to find out how it can be improved. so we compute the partial derivative starting from the last bias:

$\partial AL/\partial bn$

then

$\partial AL/\partial wn$

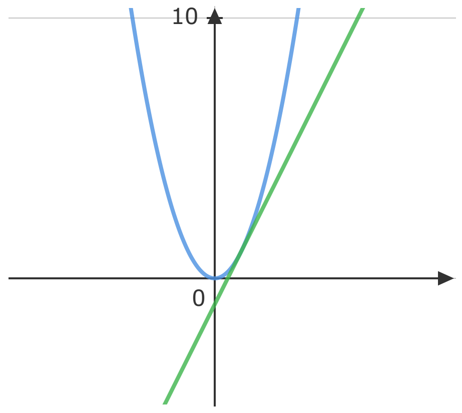
and so on...

Now we train out network and update the weight and biases!

so we basically need a learning rate α *(alpha usually, arbitrary notation)*

Now we use the earlier computed derivative, let's take example of $\partial AL/\partial wi$ *in this example:*

$\partial AL/\partial wi$



$$x\ axis\ =\ AL$$
$$y\ axis\ =\ W1\ (where\ i\ =\ 1,\ arbitrary\ for\ example\ )$$

Now this simple graph is lets say graph of AL wrt weight (at arbitrary point i)

now we take a derivative and if the derivative is positibe that means thats $\partial AL / \partial wi$ *is increasing*

and to minimize it we need to move into the opposite direction

so we update the weights in opposite direction

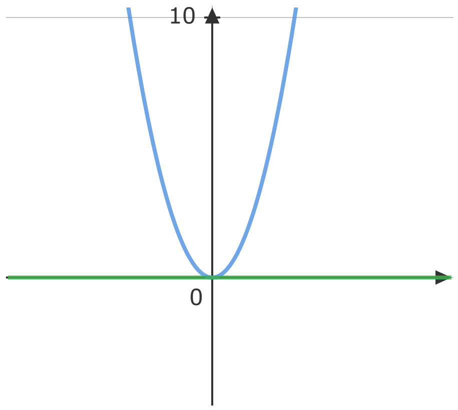
so therefore

$$weight1\ =\ weight1\ -\ alpha(weight1)$$

and then we iteratively repeat this until the point converges to the lowest point

NOTE: the good thing is that if the value of $\partial AL / \partial wi$ *is negative, that means we need to add, so the* $-ve$ *value of* $\partial AL / \partial wi$ *and* $-\alpha$ *together become*

$$weight1\ =\ weight1\ +\ alpha(weight1)$$



and now we repeat this for all the weights and biases

And Ta – Da! *We have the optimum value of weights and biases*!

The NN is trained! *Now test it*! *LLFFGGG*!