Neural-Networks-1

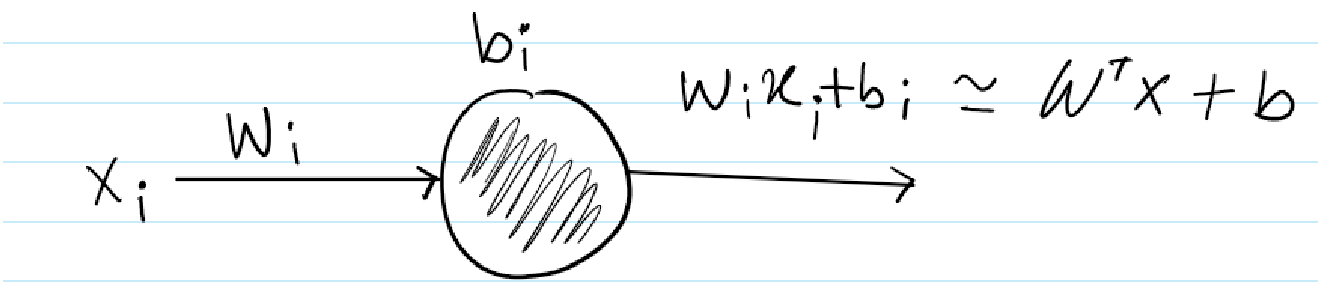
Before diving into topic
Let's solve and learn algebra implmentation with pytorch and numpy side by side #pytorch #numpy #linear-algebra
Problem Statement: Link
Notebook: Unsolved
Solved Notebook (Caution) : Link
Few Problem solving in Numpy : Problems Solution

Now let's move to Topic,
Let's learn a little theory first.

for guidance here is the raw notes of NN : Link
and below is my interpretation

Linear regression interpretation

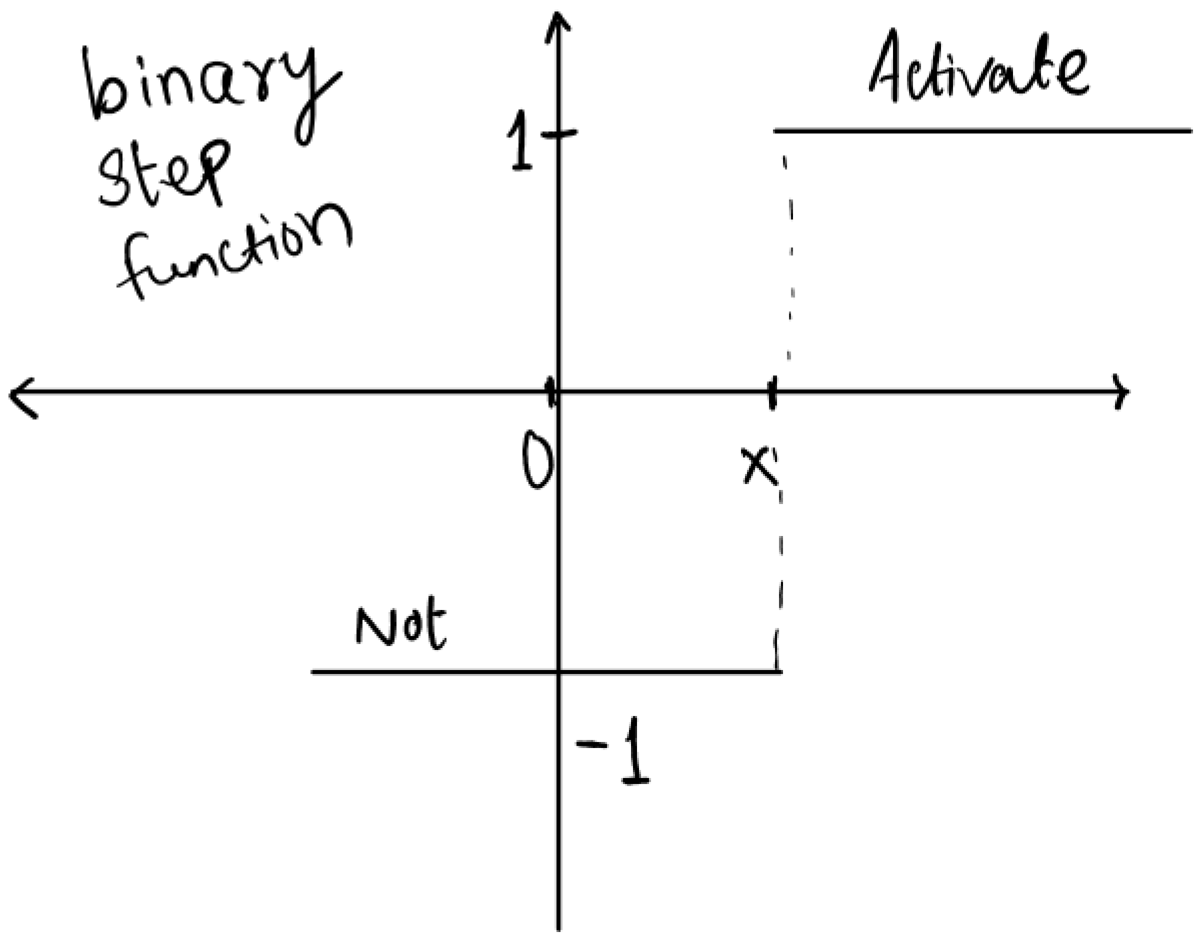
A Neural network with a single neuron and no activation function is noting but a Linear regression think of it as



where
W is the weight vector with a single vector w1,
X is a input vector with single vector x1 and
b is a bias vector with a single vector b1

the line
"Basically a single neuron will calculate weighted sum of input($W.T * X$) and then we can set a threshold to predict output in a perceptron. If weighted sum of input cross the threshold, perceptron fires and if not then perceptron doesn't predict."

implies, we are talking about activation function here.
EX:



Here activation function threshold is `x`. if we somehow get more than that then perceptron fire else it don't.
Mathematically it is,

$$f(y) = \begin{cases} 0, & \text{if } y < x \\ 1, & \text{if } y \geq x \end{cases}$$

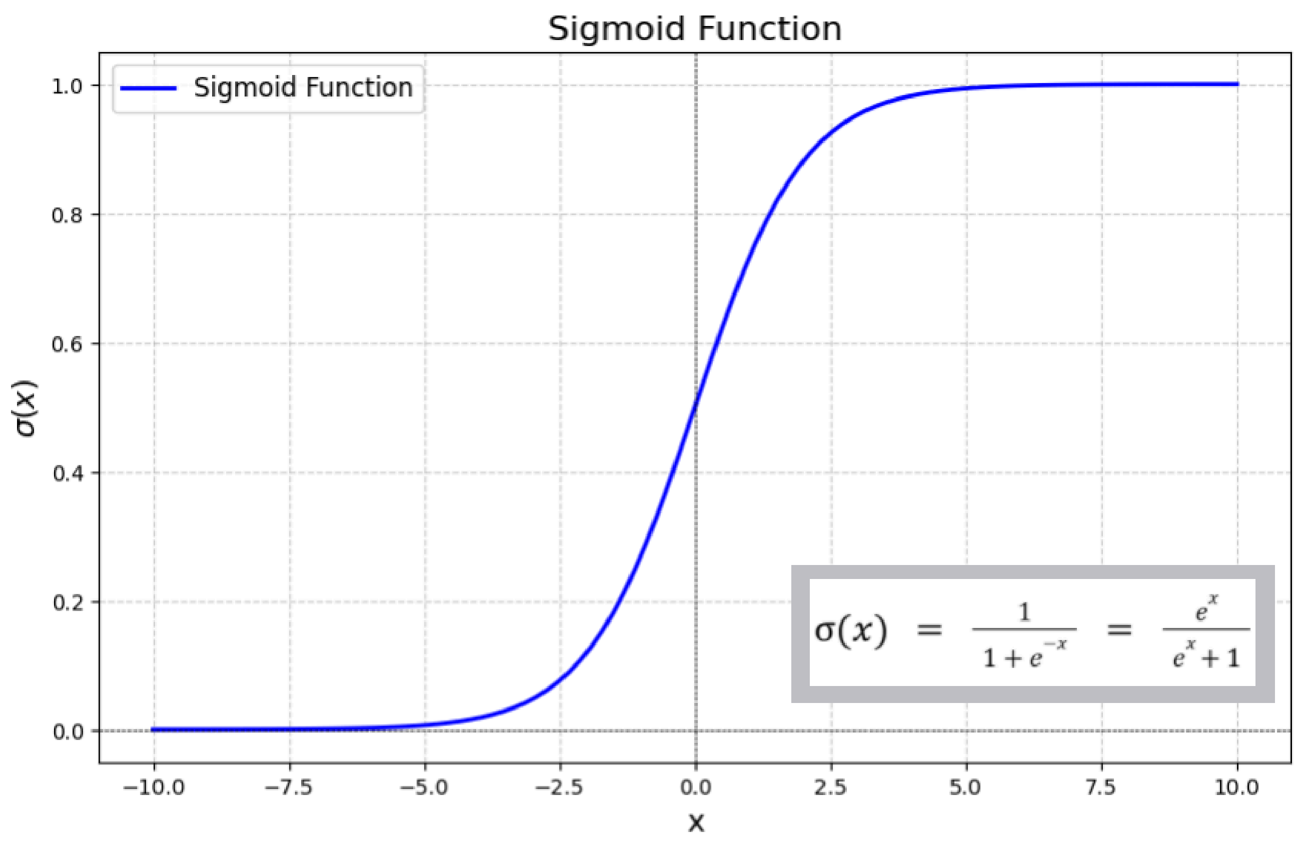
Here are some of the limitations of binary step function:

- It cannot provide multi-value outputs—for example, it cannot be used for multi-class classification problems.
- The gradient of the step function is zero, which causes a hindrance in the backpropagation process.

"Perceptron can take real values input or boolean values."
we will see a example of both #implementation

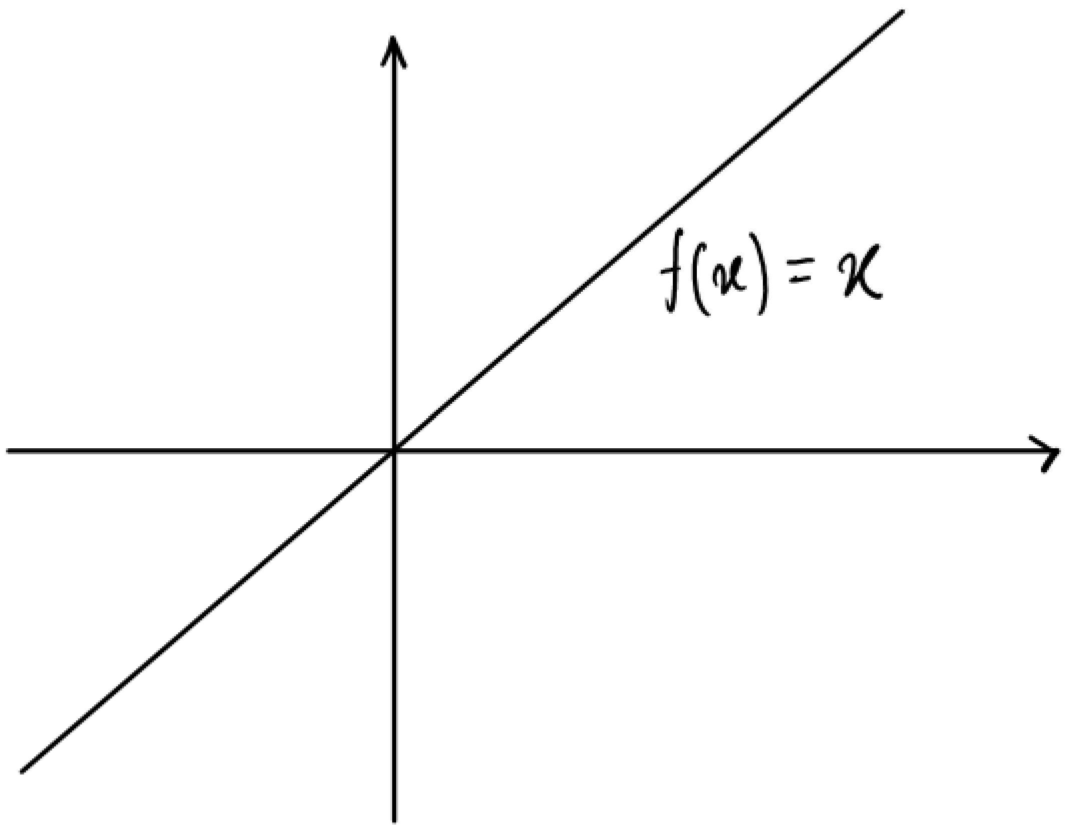
"Disadvantage of perceptron is that it only output binary values and if we try to give small change in weight and bais then perceptron can flip the output. We need some system which can modify the output slightly according to small change in weight and bias. Here comes sigmoid function in picture."

Here we are talking about the activation function again where binary step function has some limitation. so there comes the sigmoid function.



credit: GeekforGeeks

but before that let's talk about why not linear activation function ?
where we have



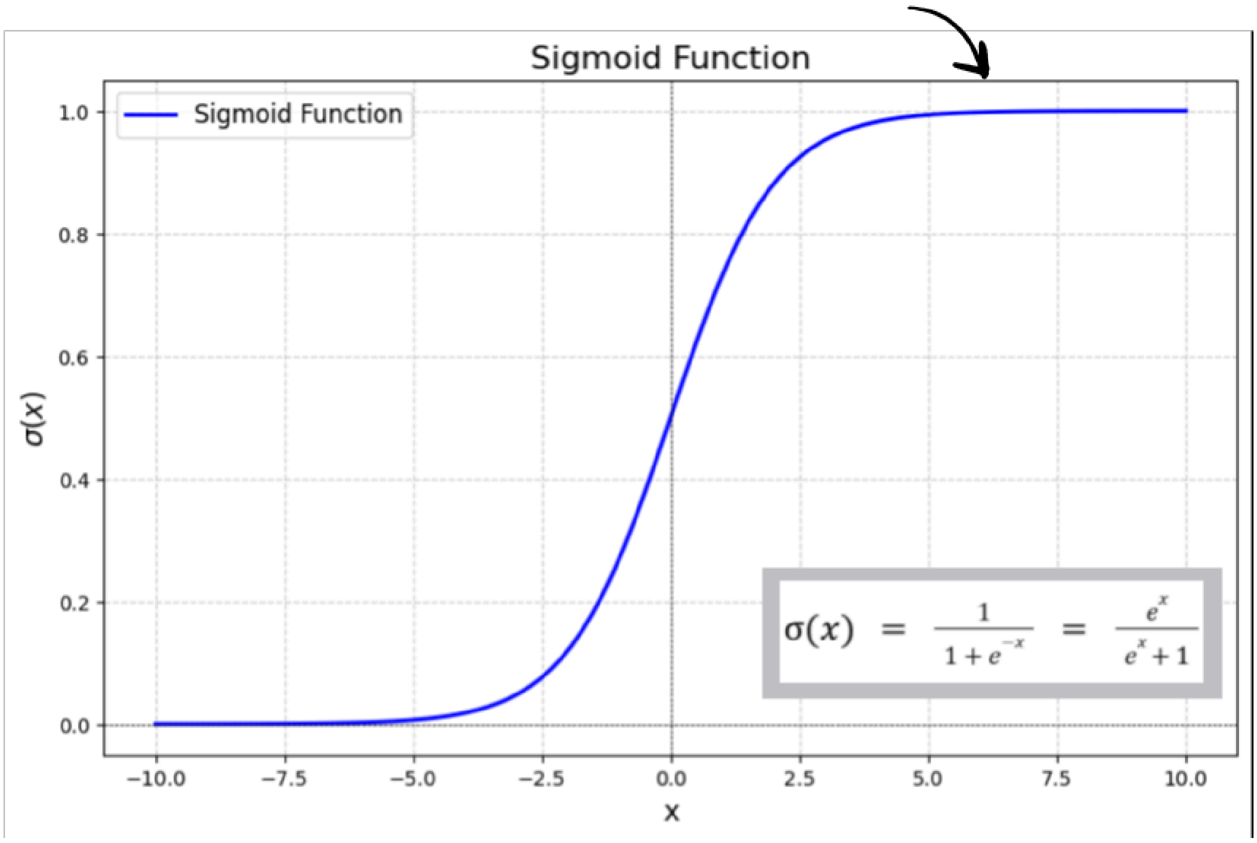
where the limitation is

- It’s not possible to use backpropagation as the derivative of the function is a constant (i.e. 1) and has no relation to the input x.

- All layers of the neural network will collapse into 1 if a linear activation function is used. No matter the number of layers in the neural network, the last layer will still be a linear function of the first layer. So, essentially, a linear activation function turns the neural network into just one layer. 🔥 🔥

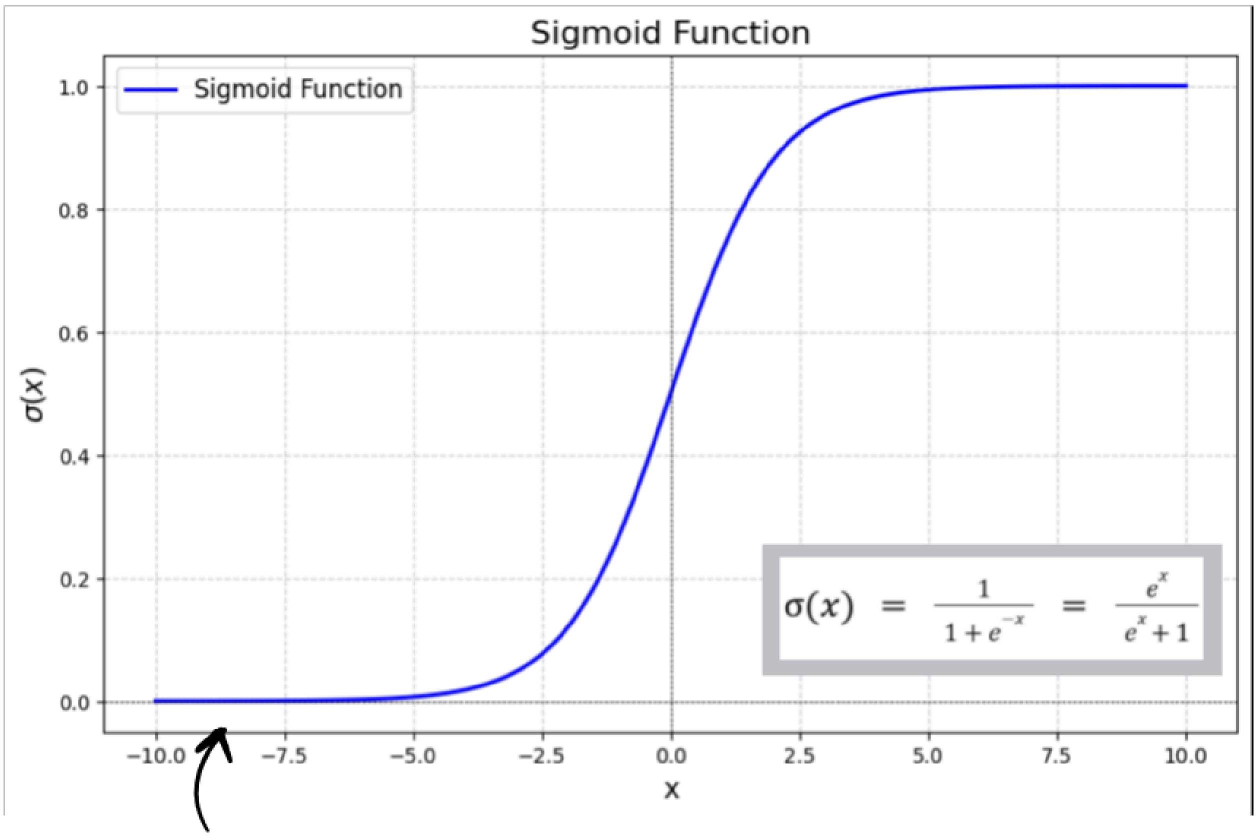
Now let's talk about **Sigmoid function**

1. This function takes any real value as input and outputs values in the range of 0 to 1.



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

2. The larger the input (more positive), the closer the output value will be to 1.0, whereas the smaller the input (more negative), the closer the output will be to 0.0



Here’s why sigmoid/logistic activation function is one of the most widely used functions:

- It is commonly used for models where we have to predict the probability as an output. Since probability of anything exists only between the range of 0 and 1, sigmoid is the right choice because of its range.

$$\boxed{\frac{1}{1+e^{-x}} \simeq \frac{e^{x}}{e^{x}+1}}$$

$$x \in \mathbb{R}$$

$$e^{-x} > 0$$

$$1+e^{-x} > 1$$

$$\frac{1}{1+e^{-x}} > 0 \quad \text{and}$$

If deno > numerator :

$$\frac{1}{1+e^{-x}} < 1$$

so it will map to a number between 0 and 1

- The function is differentiable and provides a smooth gradient, i.e., preventing jumps in output values. This is represented by an S-shape of the sigmoid activation function.
  - as there is no sharp edges in S-shape so it will be differentiable and for smoothness

Proof 1: Differentiability (The First Derivative)

To prove it is differentiable, we must find a valid formula for its slope $\frac{d}{dx}$ for any real number x. We use the **Chain Rule**.

1. Apply Chain Rule:

$$\frac{d}{dx}(1+e^{-x})^{-1} = -1 \cdot (1+e^{-x})^{-2} \cdot \frac{d}{dx}(1+e^{-x})$$

2. Differentiate the inner term:
   The derivative of 1 is 0, and the derivative of e^{-x} is $-e^{-x}$.

$$= -1 \cdot (1+e^{-x})^{-2} \cdot (-e^{-x})$$

3. Simplify:

$$= \frac{e^{-x}}{(1+e^{-x})^2}$$

4. The "Magic" Substitution:
   We can rewrite this fraction to see the relationship with the original function $\sigma(x)$:

$$= \frac{1}{1+e^{-x}} \cdot \frac{e^{-x}}{1+e^{-x}}$$

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

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

$$= \sigma(x) \cdot (1-\sigma(x))$$

   Conclusion: Since $\sigma(x)$ is defined for all real numbers, and this formula $\sigma(x)(1-\sigma(x))$ involves only multiplication and subtraction of defined values, the derivative exists everywhere.

Proof 2: Smoothness (Infinite Differentiability)

In mathematics, "smooth" usually means C^∞—that the function has derivatives of **all** orders.

We rely on the property we just proved: The derivative is defined in terms of the function itself.

$$\sigma' = \sigma(1-\sigma)$$

The Inductive Logic:

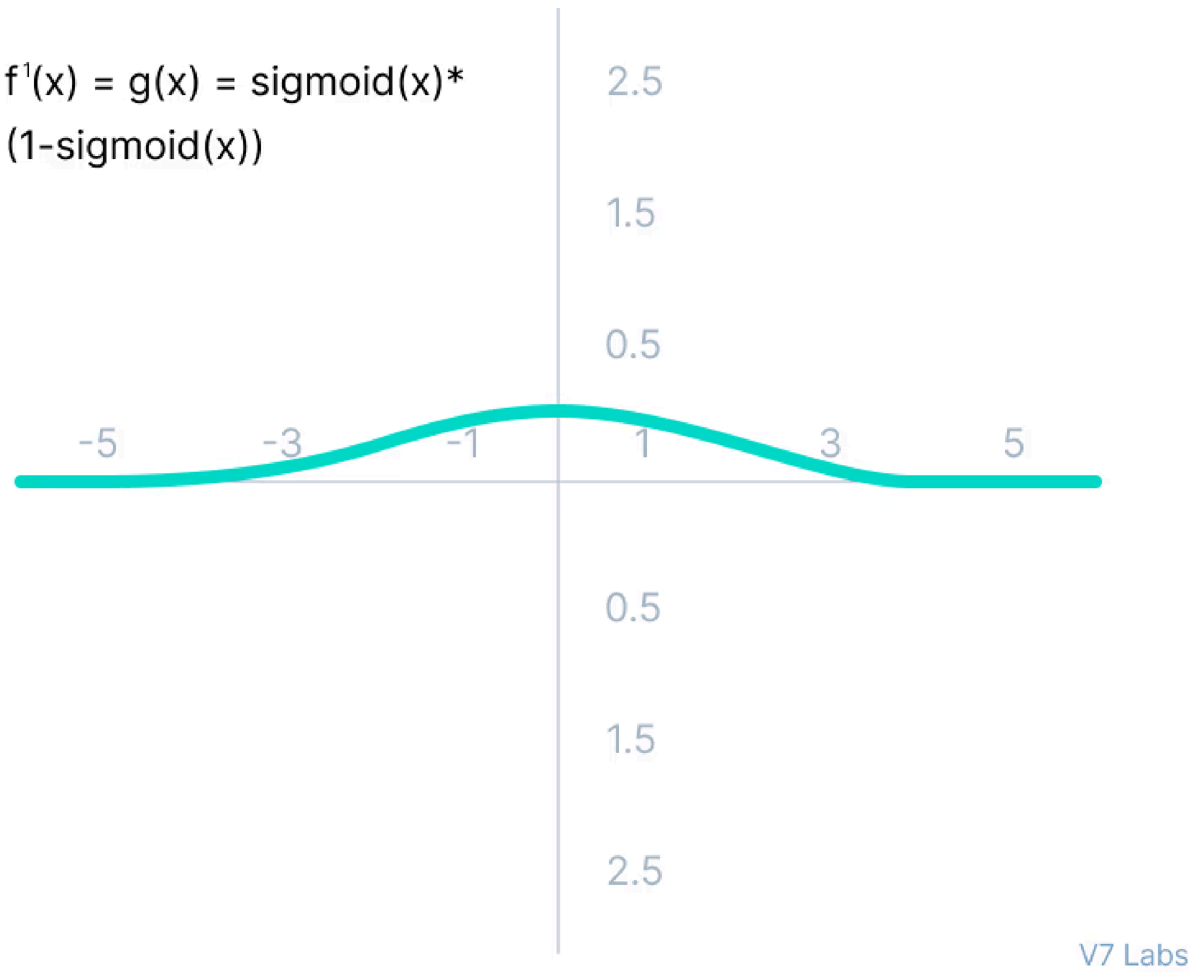
1. **First Derivative (σ'):** It is a polynomial of σ (specifically $\sigma - \sigma^2$). Since σ is differentiable, σ' is differentiable.
2. Second Derivative (σ''):

$$\sigma'' = \frac{d}{dx}(\sigma - \sigma^2) = \sigma' - 2\sigma\sigma' = \sigma'(1-2\sigma)$$

   Since σ' and σ are differentiable, σ'' must also be differentiable.
3. N-th Derivative:

   Every subsequent derivative will just be a sum of products involving lower-order derivatives. Since we never introduce a "discontinuous" operation (like division by zero or a root of a negative number), we can continue this forever

So why we need other activation function. it means there are few disadvantages of Sigmoid function



credit: V7Labs

As we can see from the above Figure, the gradient values are only significant for range -3 to 3, and the graph gets much flatter in other regions.

It implies that for values greater than 3 or less than -3, the function will have very small gradients. As the gradient value approaches zero, the network ceases to learn and suffers from the *Vanishing gradient* problem.

The output of logistic function is not symmetric around zero. So the output of all the neurons will be of the same sign.
which make training of neural network more difficult and stable.

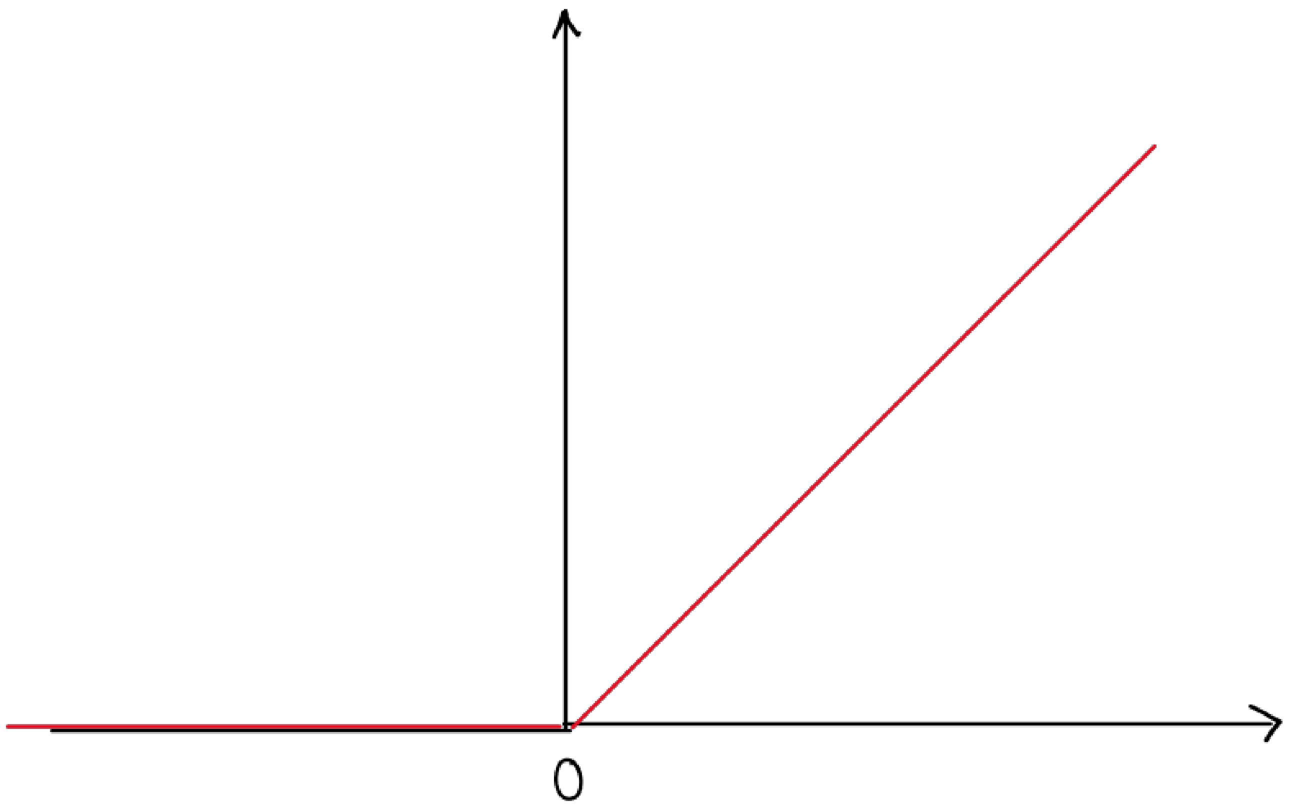
we will see #implementation

There are other activation function but we will be covering them on the go when we will be needing them.

> Applying Sigmoid function to a single neuron will act as Logistic function (Your task to prove)

"RELU stands for rectified linear unit is the most popular activation function right now that makes deep NNs train faster now."

How it is better than all previous activation functions



Gives vibe of Linear activation function for $x > 0$.
Mathematically it is

$$f(x) = max(0, x)$$

The advantages of using ReLU as an activation function are as follows:

- **Since only a certain number of neurons are activated**, the ReLU function is far more computationally efficient when compared to the sigmoid and tanh functions.
- ReLU accelerates the convergence of gradient descent towards the global minimum of the loss function due to its **linear, non-saturating property.**

"Since only a certain number of neurons are activated" : **Concept of Sparsity**
and how ReLU has advantages over sigmoid and tanh functions over this ?

- **The Problem with Sigmoid/Tanh:** These are "dense" activators. Whether the input is -100 or $+100$, the output is non-zero (e.g., 0.00001 or 0.9999). This means **every single neuron** in the network fires a signal, no matter how small, and the computer has to process all those tiny floating-point numbers in the next layer.
- **The ReLU Solution:** If the input is negative ($x < 0$), ReLU outputs a hard **0**. The neuron is completely silent.

"linear, non-saturating property." : This addresses the biggest historical problem in Deep Learning: the **Vanishing Gradient Problem**.
Understanding "Saturation": Look at the Sigmoid curve. At both ends (very high or very low inputs), the curve becomes flat (horizontal).
- In Calculus, a flat line means the **slope (gradient) is Zero**.
- When the network tries to learn via Backpropagation, it multiplies these gradients. If you multiply a bunch of small numbers (e.g., $0.1 \times 0.1 \times 0.1$), the error signal vanishes to zero. The network stops learning because it thinks there is no error to fix.

"Hidden layers predicts connection between inputs automatically, thats what deep learning is good at."

How ?

In technical terms, this is called **Feature Extraction** or **Representation Learning**.
The "How" is not that the network is smart; it's that it is **lazy** and **greedy**. It wants to reduce its error score to zero as fast as possible, and it realizes that the only way to do that is to group the inputs into meaningful patterns.

1. The Mechanism: It starts with Randomness

When you first build a neural network, the hidden layers are not connected to anything meaningful.

Initialization of weights of every connection is random.

2. The Feedback Loop:

This is the engine of the "automatic" learning. Here each time the network makes an error it update the weights of neurons that contribute most to the error.

3. The Emergence of Features

After showing a good amount of data the weights stop being random. They settle into a configuration that minimizes error. Crucially, they settle into a **Hierarchy**.

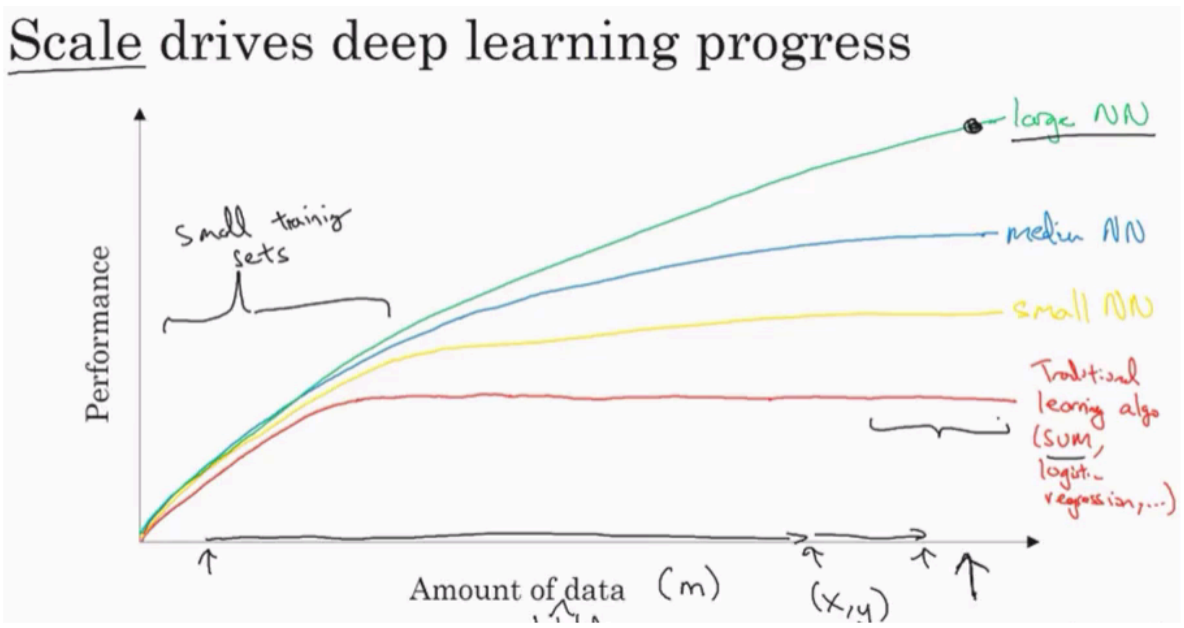
Ex:
Imagine we are training a network to recognize **Faces**. We don't teach it what an eye or a nose is. It figures it out automatically:

- **Layer 1 (The Edge Detectors):** The neurons in the first layer realize: "I can lower the error if I only fire when I see a **vertical line**." Another neuron decides to only fire for a **horizontal line**.
  - *Why?* Because every image is made of lines. It's the most basic way to compress the data.

- **Layer 2 (The Shape Detectors):** These neurons don't look at the raw pixels anymore; they look at Layer 1. A neuron in Layer 2 realizes: "Hey, if the 'vertical line' neuron AND the 'horizontal line' neuron from Layer 1 fire at the same time, that makes a **corner**."
    - Suddenly, Layer 2 is detecting corners, curves, and circles.
- **Layer 3 (The Object Detectors):** These neurons look at Layer 2. A neuron here realizes: "If I see two 'circles' and a 'triangle' roughly in the middle, that usually means a **Face**."
    - This neuron becomes a "Face Detector."

"Deep NN consists of more hidden layers (Deeper layers)" : the name says it all

Read the Portion: Link



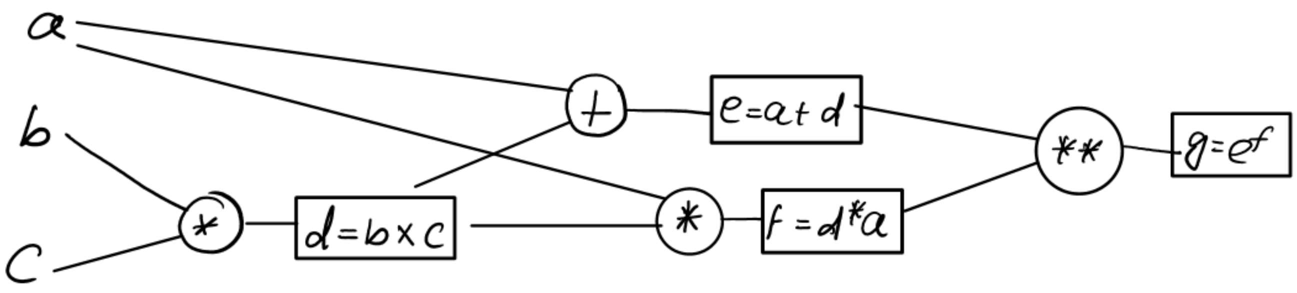
for larger data, as the size of model increases, the performance increases

but at the same time for smaller datasets, Neural network can perform as Linear regression or SVM (Support vector machine)

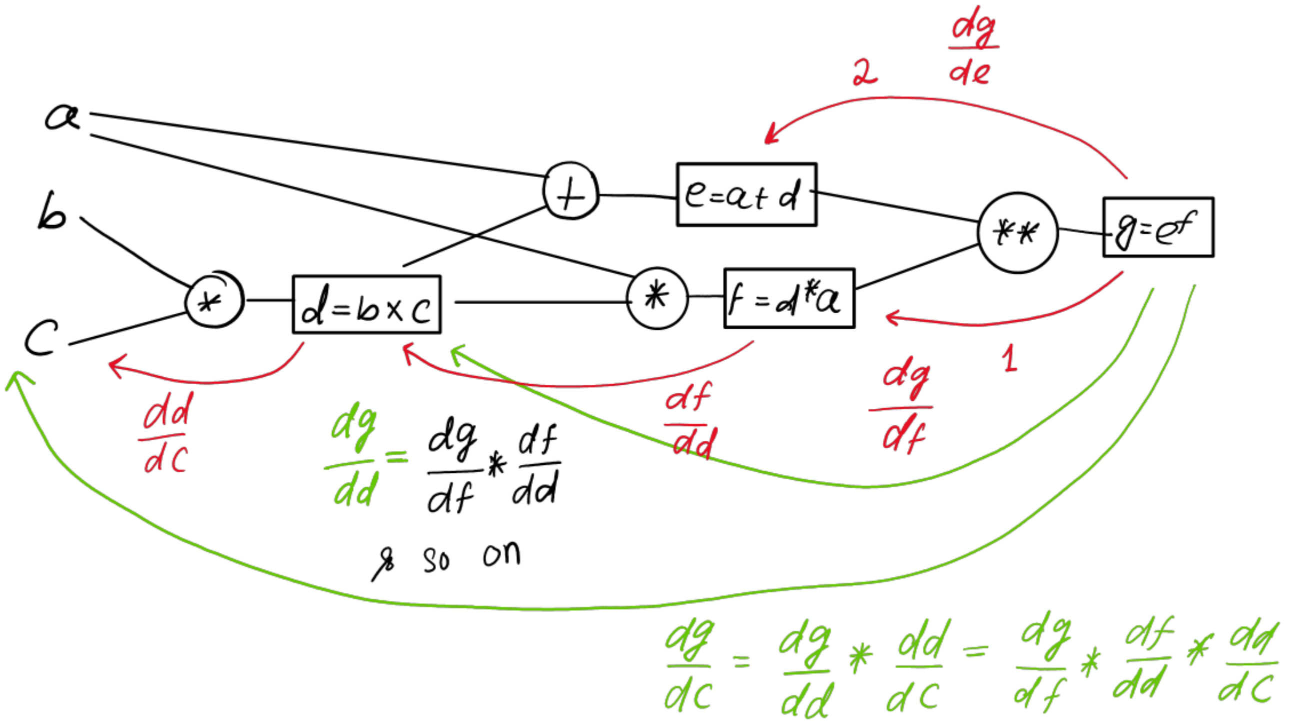
Computation graph

Graph to visualize the computation sequences and backpropagation sequence

Let's say there 3 independent variables a, b, c and few dependent variables
$d = b * c$
$e = a + d$
$f = d * a$
$g = e^f$
so the computation graph will be



Derivative would be



In the resource the logistic regression gradient descent computation graph goes the same. there the input vector consists x_1 and x_2.

Simple funda

- **Training** = Forward Pass + Backward Pass (Backpropagation).
- **Inference** = Forward Pass **only**. Unless it is Online Learning (which is a rare exception where models DO use backpropagation during inference)

- the weights are frozen => No update in weights => No training for those weights
- **Backpropagation's only purpose** is to calculate gradients to update the weights (to learn).

- **Inference's purpose** is to use the existing weights to make a prediction.

Vectorization

for loops are okay for small datasets. To avoid for loop in large datasets we use vectorization which is a feature in many libraries (numpy, pytorch, ..).

Like Dot product which use vectorization by default

"The vectorization can be done on CPU or GPU thought the SIMD operation. But its faster on GPU."

1. What is SIMD? (The Baseline)

SIMD stands for **Single Instruction, Multiple Data**.

- **Without SIMD:** If you want to add `[1, 2, 3, 4]` to `[5, 6, 7, 8]`, the processor does it one by one: `1+5`, then `2+6`, etc. (4 cycles).
- **With SIMD:** The processor grabs the whole chunk and adds them all in **one single cycle**.

Both modern CPUs (using AVX-512 instructions) and GPUs do this.

2. The "Lane Width" Difference (The Scale)

The difference lies in **how much data** they can handle in that single cycle.

The CPU

A CPU is designed to be smart and versatile. It has a few very powerful cores (e.g., 8 to 16 cores).

- **SIMD Capability:** A CPU core might have a "vector unit" that can handle **8 or 16 numbers** at once (e.g., 512 bits).
- **Scenario:** If you have 1,000,000 numbers to add, the CPU takes them in chunks of 16. It has to repeat the cycle thousands of times.

The GPU

A GPU is designed to be "dumb" but massive. It doesn't have 16 smart cores; it has **thousands of tiny, simple cores** (e.g., 10,000+ cores).

- **SIMD Capability:** The GPU doesn't just process a chunk of 16; it effectively processes thousands of data points simultaneously across its massive grid of cores.
- **Scenario:** If you have 1,000,000 numbers, the GPU swallows them in huge gulps. It finishes the job in a fraction of the cycles.

That's why Vectorization is faster on GPU.

See Vectorizing logistic regression : [Link]

the blog has steps for building neural network as

General Notes

- The main steps for building a Neural Network are:

  - Define the model structure (such as number of input features and outputs)
  - Initialize the model's parameters.
  - Loop.
    - Calculate current loss (forward propagation)
    - Calculate current gradient (backward propagation)
    - Update parameters (gradient descent)
- Preprocessing the dataset is important.
- Tuning the learning rate (which is an example of a "hyperparameter") can make a big difference to the algorithm.

Let's see how use full it is with a implementation task.