Basics of deep learning

Class	Deep learning

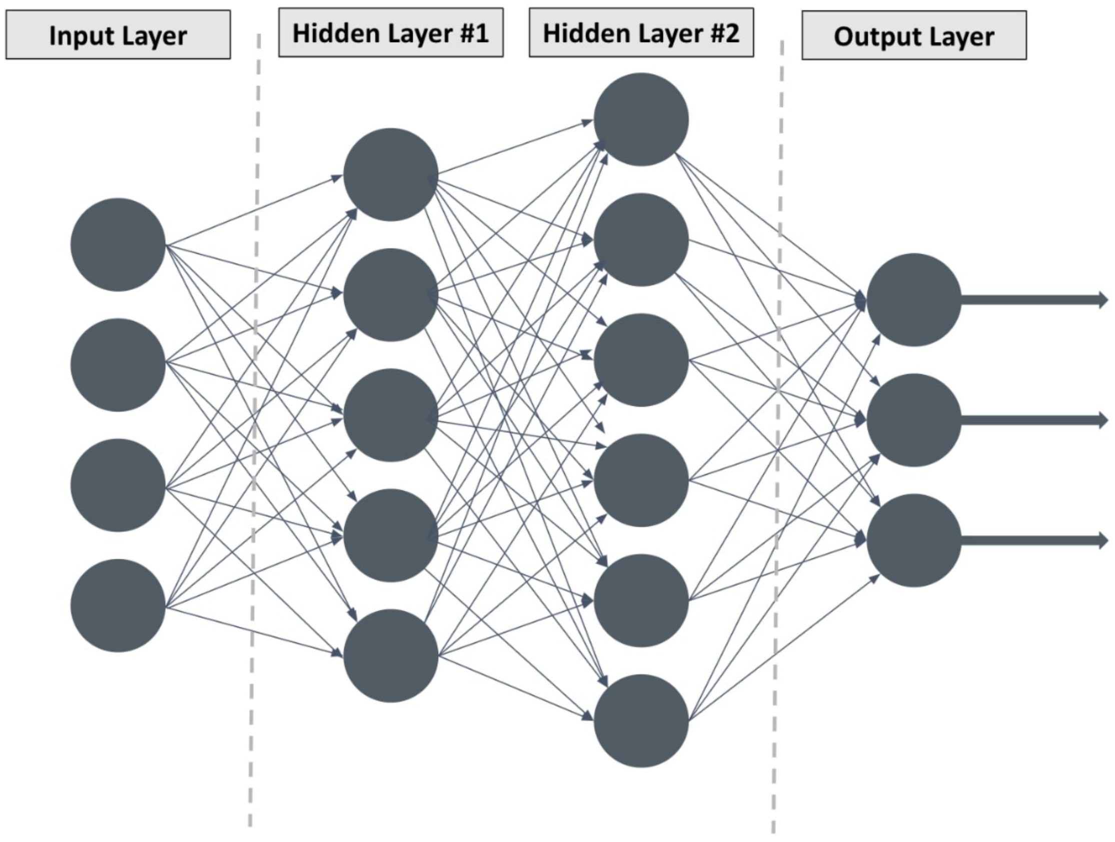
Deep learning is a technique in the programming which mimics the human brain.

Neural network is basically deep learning architecture which consist of set of perceptron in multiple layers which are connected to each other to form a neural network

Neural network can have many layers from 2 to n.

Nn layers:

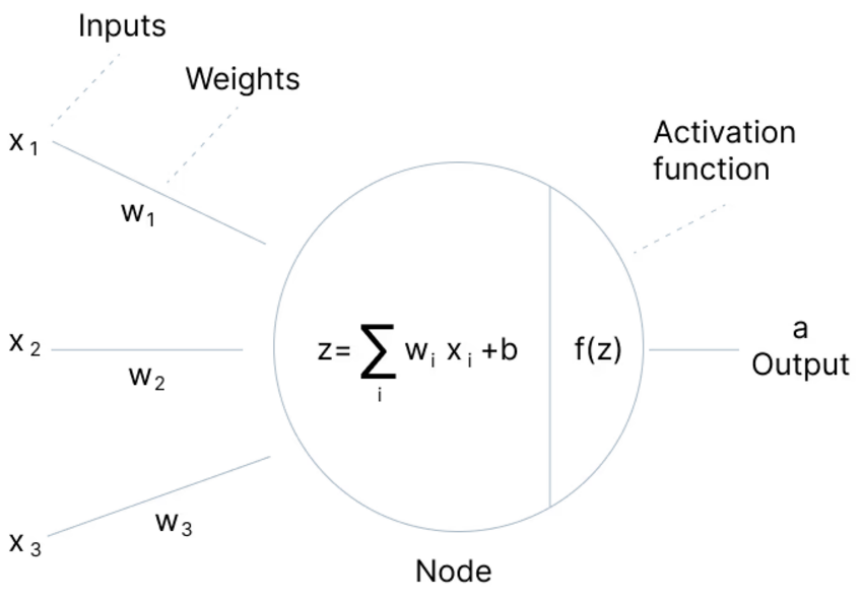
- Input layer: It is the first layer in nn which has each node (perceptron) for each feature.
- hidden layer: This constitutes of 1 or many layers between input layer and output layer. The hidden layer helps in processing and understanding the data. This preceptron are called as neurons and not node, because there is lot of processing happening in this layer
- Output layer: This can have one node or many depending on the output we are expecting for the pirticular use case.



Each neuron is connected with other neurons, and there is some weights assigned to each connection which is crutial for the model. The machine learns with the data means, the machine updates the weights until it converges and makes the right prediction.

As we were discussing in hidden layer that each neuron preforms some processing, So there are 2 types of operation that is taking place in each neuron:

- The summation of weight x input for each connection to the neuron happens
- Activation function is applied on the above result



The above figure shows you a single neuron, its input and the processing that is taking place inside it.

$$z = \sum w_i x_i + b$$

Each input is multiplied with its respective weight and we sum all the inputs and add a bias (i will later cover why we add this bias term)

Now this Z is passed to the activation function. There are many activation function and depending on the requirement we use them.

Activation Functions

An activation function is **a mathematical function applied to the output of a neuron**. Its primary purpose is to introduce non-linearity into the model, allowing the network to learn and represent complex patterns in the data.

Key characteristics of activation functions:

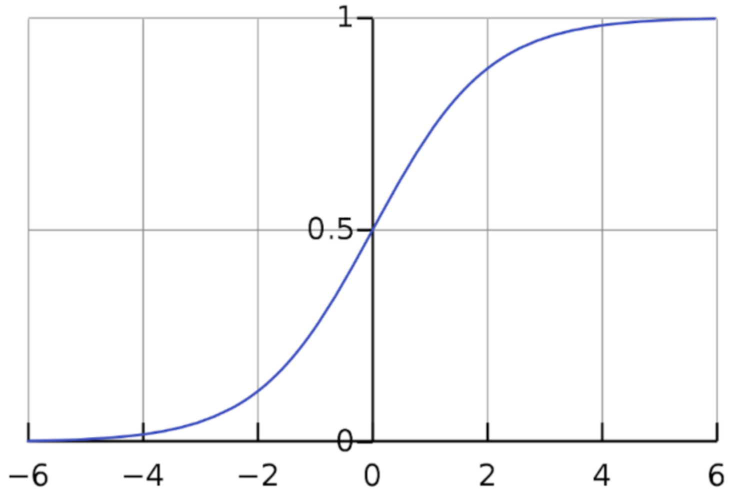
- **Non-linearity**: Activation functions introduce non-linearity into the model, enabling the network to learn complex patterns.

- **Thresholding**: Activation functions can be thought of as a gate that checks if an incoming value is greater than a critical number.
- **Output Range**: Activation functions can have different output ranges, such as binary (0 or 1), continuous (any value between 0 and 1), or unbounded (any real value).

Sigmoid activation function

It is a mathematical function that maps any real-valued number to a value between 0 and 1.

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

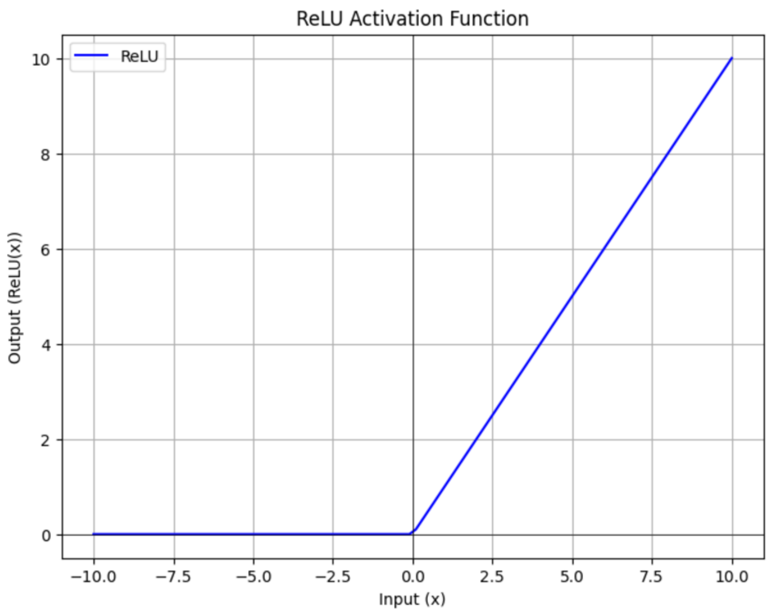


ReLU activation function

It stands for Rectified linear unit.

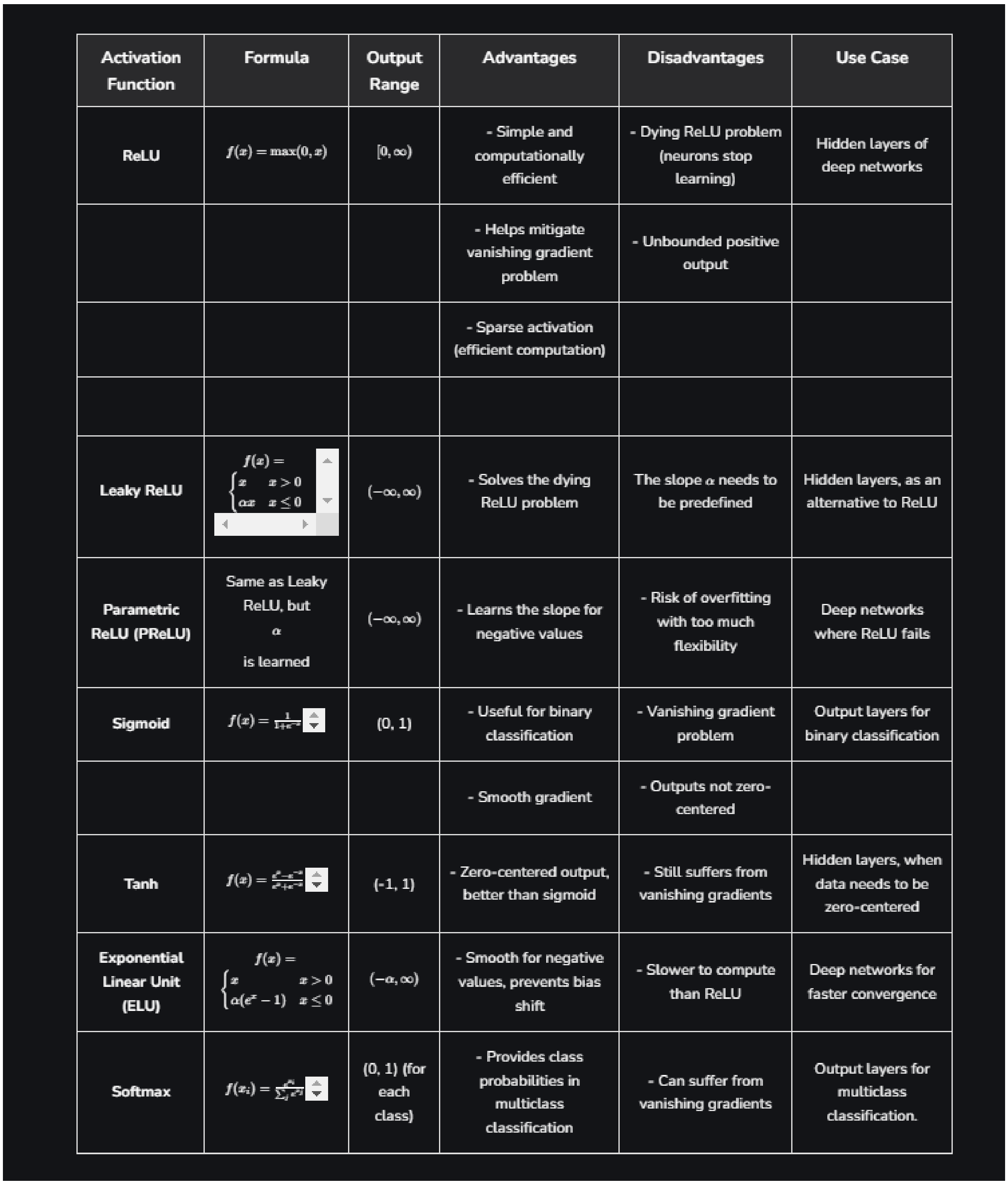
This function is a linear function that outputs zero if its input is negative and directly outputs the input otherwise.

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



ReLU is computationally efficient as it involves only a thresholding operation. This simplicity makes it easy to implement and compute, which is crucial when training deep neural networks with millions of parameters. Although it seems like a piecewise linear function, ReLU is still a non-linear function. This allows the model to learn more complex data patterns and model intricate relationships between features.

ReLU's ability to output zero for negative inputs introduces sparsity in the network, meaning that only a fraction of neurons activate at any given time. This can lead to more efficient and faster computation. ReLU offers computational advantages in terms of backpropagation, as its derivative is simple—either 0 (when the input is negative) or 1 (when the input is positive). This helps to avoid the vanishing gradient problem, which is a common issue with sigmoid or tanh activation functions.



Activation Function	Formula	Output Range	Advantages	Disadvantages	Use Case
ReLU	$f(x) = \max(0, x)$	$[0, \infty)$	- Simple and computationally efficient	- Dying ReLU problem (neurons stop learning)	Hidden layers of deep networks
			- Helps mitigate vanishing gradient problem	- Unbounded positive output	
			- Sparse activation (efficient computation)		
Leaky ReLU	$f(x) = \begin{cases} x & x > 0 \\ \alpha x & x \leq 0 \end{cases}$	$(-\infty, \infty)$	- Solves the dying ReLU problem	The slope α needs to be predefined	Hidden layers, as an alternative to ReLU
Parametric ReLU (PReLU)	Same as Leaky ReLU, but α is learned	$(-\infty, \infty)$	- Learns the slope for negative values	- Risk of overfitting with too much flexibility	Deep networks where ReLU fails
Sigmoid	$f(x) = \frac{1}{1+e^{-x}}$	(0, 1)	- Useful for binary classification	- Vanishing gradient problem	Output layers for binary classification
			- Smooth gradient	- Outputs not zero-centered	
Tanh	$f(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$	(-1, 1)	- Zero-centered output, better than sigmoid	- Still suffers from vanishing gradients	Hidden layers, when data needs to be zero-centered
Exponential Linear Unit (ELU)	$f(x) = \begin{cases} x & x > 0 \\ \alpha(e^x - 1) & x \leq 0 \end{cases}$	$(-\alpha, \infty)$	- Smooth for negative values, prevents bias shift	- Slower to compute than ReLU	Deep networks for faster convergence
Softmax	$f(x_i) = \frac{e^{x_i}}{\sum_j e^{x_j}}$	(0, 1) (for each class)	- Provides class probabilities in multiclass classification	- Can suffer from vanishing gradients	Output layers for multiclass classification.

You can refer the above table for advantages and disadvantages of each activation function. There is specific cases where each function performs best. Based on this we need to select which activation function we need for each layer.

For hidden layer ReLU is best

for output layer if its classification problem : binary means sigmoid is best , multiclass classification means softmax is best.

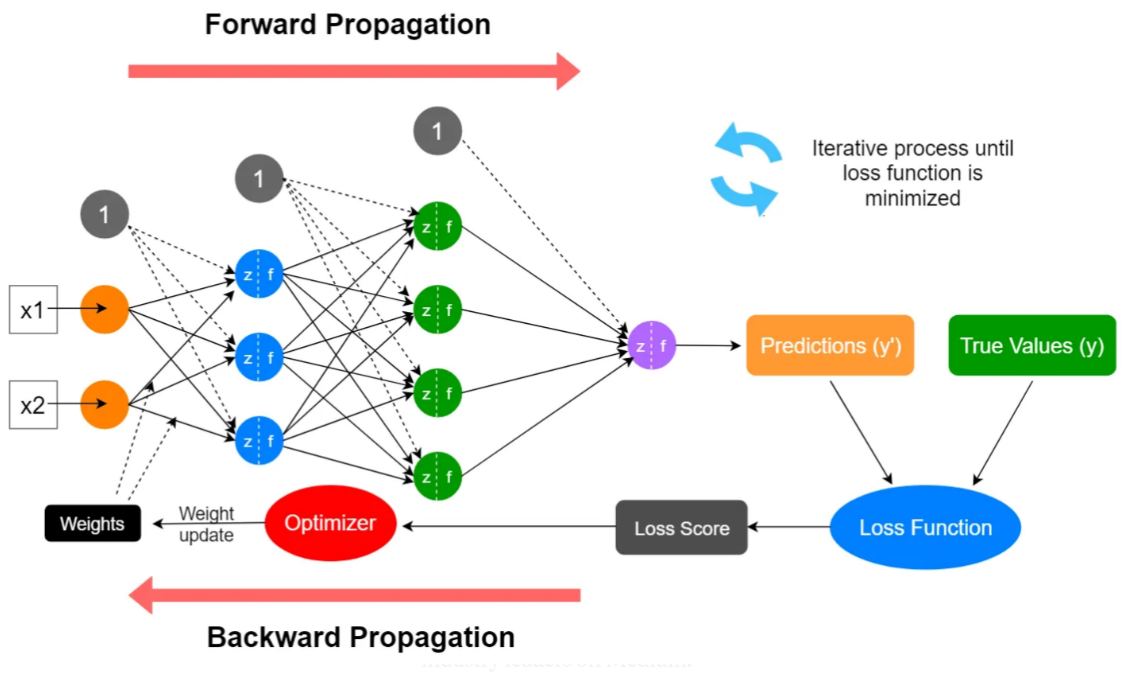
for linear output again relu is best.

Training Neural network

The learning (training) process of a neural network is an iterative process in which the calculations are carried out forward and backward through each layer in the network until the loss function is minimized.

The entire learning process can be divided into three main parts:

- **Forward propagation (Forward pass)**
- **Calculation of the loss function**
- **Backward propagation (Backward pass/Backpropagation)**



Forward propagation is the process where input data flows through the network, layer by layer, until it reaches the output. During this phase, each neuron applies its weights and activation function to produce an output. The calculation of the loss function follows, comparing the network's predictions to the actual target values. This quantifies how well the network is performing.

Finally, backpropagation occurs, where the error is propagated backwards through the network, allowing for the adjustment of weights to minimize the loss in future iterations.

This iterative process of forward propagation, loss calculation, and backpropagation continues until the network's performance reaches a satisfactory level or a predetermined number of iterations (epochs) is completed. The goal is to gradually refine the network's weights and biases, enabling it to make increasingly accurate predictions on the training data and, ideally, generalize well to unseen data.

The actual training of a neural network involves the process of backpropagation, which utilizes the chain rule from calculus to efficiently compute gradients. Here's a detailed look at how this works:

1. Forward Pass

First, we perform a forward pass through the network. For each layer l, we compute:

$$z^{[l]} = W^{[l]}a^{[l-1]} + b^{[l]}$$

$$a^{[l]} = g^{[l]}(z^{[l]})$$

Where W is the weight matrix, b is the bias vector, and g is the activation function.

2. Compute Loss

After the forward pass, we compute the loss L. For example, for binary classification with m training examples, we might use cross-entropy loss:

$$L = -\frac{1}{m}\sum_{i=1}^{m}[y_i \log(a_i) + (1 - y_i)\log(1 - a_i)]$$

3. Backpropagation

Now, we use backpropagation to compute the gradients. We start from the output layer and move backwards:

$$\frac{\partial L}{\partial a^{[L]}} = -(\frac{y}{a^{[L]}} - \frac{1-y}{1-a^{[L]}})$$

For each layer l, we compute:

$$\delta^{[l]} = \frac{\partial L}{\partial z^{[l]}} = \frac{\partial L}{\partial a^{[l]}} \cdot g'^{[l]}(z^{[l]})$$

$$\frac{\partial L}{\partial W^{[l]}} = \delta^{[l]}(a^{[l-1]})^T$$

$$\frac{\partial L}{\partial b^{[l]}} = \delta^{[l]}$$

Here, g' is the derivative of the activation function.

4. Chain Rule Application

The chain rule is crucial in backpropagation. It allows us to compute gradients layer by layer. For example:

$$\frac{\partial L}{\partial W^{[l]}} = \frac{\partial L}{\partial a^{[l]}} \cdot \frac{\partial a^{[l]}}{\partial z^{[l]}} \cdot \frac{\partial z^{[l]}}{\partial W^{[l]}}$$

This chain of partial derivatives allows the error to flow backwards through the network.

5. Update Parameters

Finally, we update the weights and biases using an optimization algorithm like gradient descent:

$$W^{[l]} = W^{[l]} - \alpha \frac{\partial L}{\partial W^{[l]}}$$

$$b^{[l]} = b^{[l]} - \alpha \frac{\partial L}{\partial b^{[l]}}$$

Where α is the learning rate.

This process is repeated for many iterations (epochs) until the loss converges or a stopping criterion is met. The backpropagation algorithm, powered by the chain rule, allows for efficient computation of gradients, making it possible to train deep neural networks with many layers.

Common Problems in Deep Neural Networks

1. Vanishing Gradient Problem

The vanishing gradient problem occurs when the gradients become extremely small as they are backpropagated through the network, especially in deep networks. This can lead to very slow learning in the earlier layers of the network.

- **Cause:** Often occurs with activation functions like sigmoid or tanh, where gradients are small for large input values.
- **Effect:** Earlier layers learn very slowly or not at all, leading to poor performance.
- **Solutions:** Use ReLU activation, implement skip connections (as in ResNet), or use gradient clipping.

2. Exploding Gradient Problem

The exploding gradient problem is the opposite of the vanishing gradient problem. It occurs when the gradients become extremely large, causing unstable updates to the network weights.

- **Cause:** Can occur when the product of gradients becomes very large, often in recurrent neural networks.
- **Effect:** Leads to unstable training, with weights updating to very large values, potentially causing overflow.
- **Solutions:** Gradient clipping, weight regularization, or careful initialization of weights.

3. Dead Neuron Problem

The dead neuron problem primarily affects networks using ReLU activation functions. It occurs when a neuron always outputs zero for any input.

- **Cause:** Often happens when a large negative bias is learned, or when the learning rate is too high.
- **Effect:** The neuron stops learning and becomes inactive, reducing the network's capacity.

- **Solutions:** Use Leaky ReLU or other variants of ReLU, careful learning rate selection, or proper weight initialization.

These problems highlight the importance of careful network design, appropriate activation function selection, and proper training techniques in deep learning. Addressing these issues is crucial for developing robust and effective neural networks.

Solutions to Common Deep Learning Problems

Dropout Layers

Dropout is a regularization technique that helps prevent overfitting and can address some of the problems mentioned earlier.

- **How it works:** Randomly "drops out" a certain percentage of neurons during training.
- **Benefits:** Reduces overfitting, improves generalization, and can help with the vanishing gradient problem.
- **Implementation:** Typically applied after activation functions in hidden layers.

Batch Normalization

Batch normalization is a technique that normalizes the inputs to each layer, which can help with training stability and speed.

- **How it works:** Normalizes the output of a previous activation layer by subtracting the batch mean and dividing by the batch standard deviation.
- **Benefits:** Reduces internal covariate shift, allows higher learning rates, and can mitigate the vanishing/exploding gradient problem.

Proper Weight Initialization

Careful initialization of weights can help prevent both vanishing and exploding gradients.

- **Techniques:** Xavier/Glorot initialization, He initialization.
- **Benefits:** Ensures that the variance of activations remains the same across layers, leading to stable training.

Implementing these solutions can significantly improve the training stability and performance of deep neural networks, addressing many of the common problems encountered in deep learning.

xavier normal:

$$w_{ij} = normaldistribution(0, \sigma)$$

where ,

$$\sigma = \sqrt{\frac{2}{fanin + fanout}}$$

fanin=number of input to neuron

fanout=number of output to neuron

Xavier works pretty well with sigmoid activation function

where as He initialization works best for ReLU