# Course Materials for GEN-AI

*Northeastern University*

These materials have been prepared and sourced for the course **GEN-AI** at Northeastern University. Every effort has been made to provide proper citations and credit for all referenced works.

If you believe any material has been inadequately cited or requires correction, please contact me at:

**Instructor: Ramin Mohammadi**
r.mohammadi@northeastern.edu

*Thank you for your understanding and collaboration.*

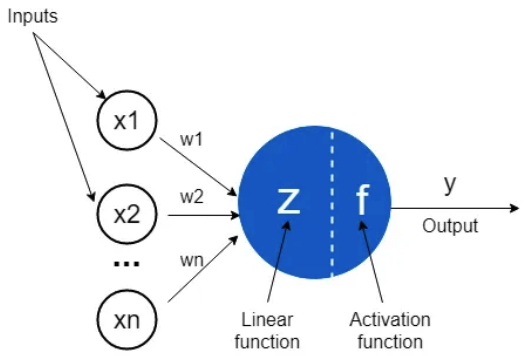
---

Content sourced from multiple references - see Reference section for details

# Neural Networks

## Perceptron

A **Perceptron** is a simple artificial neural network unit that takes multiple inputs, performs a weighted sum of these inputs, and applies a threshold function to produce an output.
It's essentially a basic building block of neural networks, serving as a simplified model of a biological neuron.




- **Inputs**: A neuron takes multiple inputs, often represented as $x_1, x_2, \ldots, x_n$ each of which is associated with a weight $w_1, w_2, \ldots, w_n$. A bias term $b$ is usually added as well.
- **Weighted Sum:** The neuron calculates a weighted sum of the inputs $z$, using the formula:

$$z = \sum_{i=1}^{n} w_i x_i + b$$

- **Activation Function**: The neuron then applies an activation function to this weighted sum to introduce non-linearity into the model.

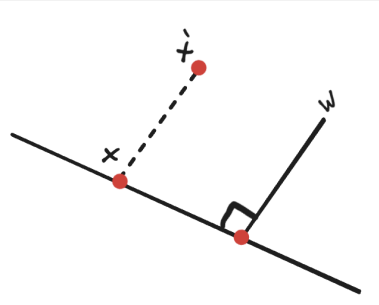
$$a = f(z)$$

  Common activation functions include the sigmoid, ReLU, and tanh functions.
- **Output**: The result, $\alpha$ is the neuron's output, which is passed to the next layer in the network.

**Reminders**

- $w^t x + b = 0$


- $w^t x + b = 0 \Rightarrow b = -w^t x$
- $w^t x + b = 0$




1. $x' - x || w \Rightarrow x' - x = \alpha w \Rightarrow x = x' - \alpha w$
2. $w^T x + b = 0 \Rightarrow w^T [x' - \alpha w] + b = 0 \Rightarrow w^T \ x' - \alpha w^T w = -b$
   $\alpha = \frac{w^T x' + b}{||w||_2^2} \Rightarrow x - x' = \frac{|w^T x' + b|}{||w||_2}$

# Problem

**Given D = {$(x_1, y_1), ..., (x_n, y_n)$}, find f(w, b) so that distance between misclassified data and Decision Boundary is min.**

To minimize the distance between misclassified data and the decision boundary, we need to derive the decision function $f(w, b)$. Lets such that our loss function $J(w)$,

$$J(\mathbf{w}) = \sum_{i=1}^{m} -y_i(\mathbf{w}^T \mathbf{x}_i + b) \tag{1}$$

The decision boundary is typically defined by the condition $w^T x + b$, which separates the data points classified as positive $y_i = 1$ and negative $y_i = -1$. The loss function aims to maximize the margin between the positive and negative class examples. It does this by penalizing cases where the product of the label and the linear prediction is negative.
To minimize this loss function, we can use gradient descent or its variants.

- **Initialize**: Start with random initial values for $\mathbf{w}$ and b.
- **Compute Gradient**: Calculate the gradient of the loss function with respect to $\mathbf{w}$ and b:

$$\frac{\partial J}{\partial \mathbf{w}} = -\sum_{i=1}^{m} y_i \mathbf{x} i \frac{\partial J}{\partial \mathbf{w}} = -\sum_{i=1}^{m} y_i x_i \tag{2}$$

$$\frac{\partial J}{\partial \mathbf{b}} = -\sum_{i=1}^{m} y_i \tag{3}$$

- **Update Parameters**: Update $\mathbf{w}$ and b using the gradient descent update rule:

$$\mathbf{w} := \mathbf{w} - \alpha \frac{\partial J}{\partial \mathbf{w}} := \mathbf{w} - \alpha \sum_{i=1}^{m} yixi \tag{4}$$

$$b := b - \alpha \frac{\partial J}{\partial b} := b - \alpha \sum_{i=1}^{m} y_i \tag{5}$$

  where $\alpha$ is the learning rate.
- **Repeat**: Repeat steps 2 and 3 until convergence or a maximum number of iterations.

By minimizing the loss function using gradient descent, you can find a set of weights and bias that effectively classify the data points according to their labels.
In 1960, a book named "Perceptron" was published that shattered the dream behind Perceptron.
**It proved that it cannot solve "XOR"/non linear problems.**
The idea then was to stack multiple perceptrons forming a layer.

## Tensors

In the realm of machine learning and deep learning, a tensor is a multi-dimensional array of numbers. It's a generalization of matrices (2D arrays) to higher dimensions.

$$T = (i_N, i_W, i_H, i_C)$$

where $i_N, i_W, i_H, i_C$ represents **dimensions** of a tensor.

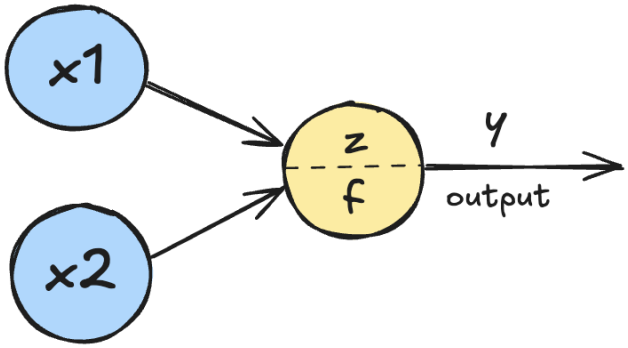
- $\mathbf{i_N}$**(Batch Size)**: This dimension represents the number of samples in a batch. For instance, if you're training a neural network on a dataset of 100 images, you might process them in batches of 32.
- $\mathbf{i_W}$**(Width)**: This dimension typically refers to the horizontal axis of an image.
- $\mathbf{i_H}$**(Height)**: This dimension refers to the vertical axis of an image.
- $\mathbf{i_C}$**(Channel)**: This dimension represents the number of color channels. For example, a color image has 3 channels (RGB), while a grayscale image has 1 channel.

## Neural Networks

The main idea behind Neural Networks (NNs) is to mimic how the human brain processes information. NNs consist of layers of interconnected nodes (or neurons) that work together to learn patterns from data. Each neuron receives inputs, processes them with a weighted sum, applies an activation function to introduce non-linearity, and passes the result to the next layer. Through training, the network adjusts its weights to minimize prediction errors, allowing it to learn complex relationships in data and make accurate predictions or classifications.

## Neuron or Node

In a neural network, a **neuron/node** is a fundamental unit that mimics the functioning of a biological neuron. It receives one or more inputs, processes them, and produces an output. The process is as follows:




- **Inputs**: A neuron takes multiple inputs, often represented as $x_1, x_2, \ldots, x_n$ each of which is associated with a weight $w_1, w_2, \ldots, w_n$. A bias term $b$ is usually added as well.
- **Weighted Sum:** The neuron calculates a weighted sum of the inputs $z$, using the formula:

$$z = \sum_{i=1}^{n} w_i x_i + b$$

- **Activation Function**: The neuron then applies an activation function to this weighted sum to introduce non-linearity into the model.

$$a = f(z)$$

  Common activation functions include the sigmoid, ReLU, and tanh functions.

- **Output**: The result, a is the neuron's output, which is passed to the next layer in the network.

Why are activation functions needed?
Without an activation function, the output of a neuron would simply be a linear combination of its inputs, making the entire network behave like a single-layer linear model.

Activation functions introduce **non-linearity**, enabling neural networks to learn and model more complex relationships in data.

This is crucial for solving problems like image classification or language processing where the data has intricate patterns. They also help neural networks learn hierarchical and complex patterns in the data.

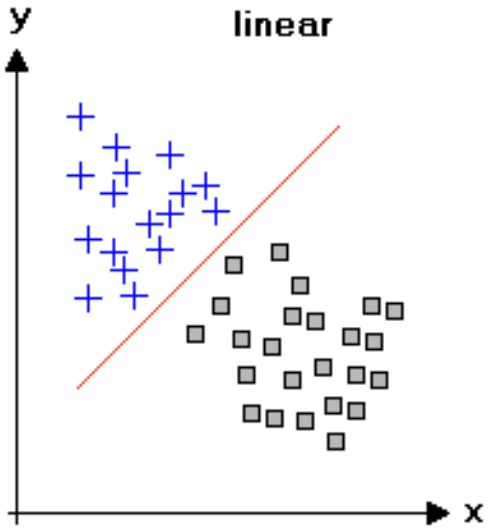



Figure 1: Linear Data

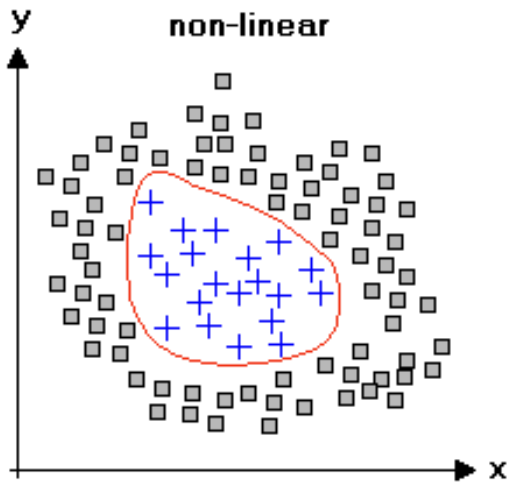


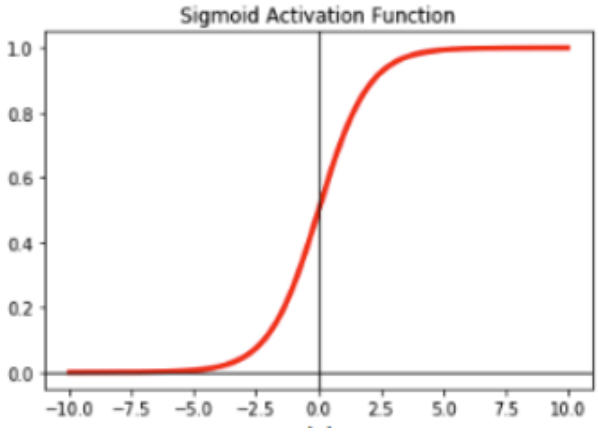

Figure 2: Non-Linear Data

Some common activation functions:

- Sigmoid function

$$f(z) = \frac{1}{1 + e^{-z}} \tag{6}$$

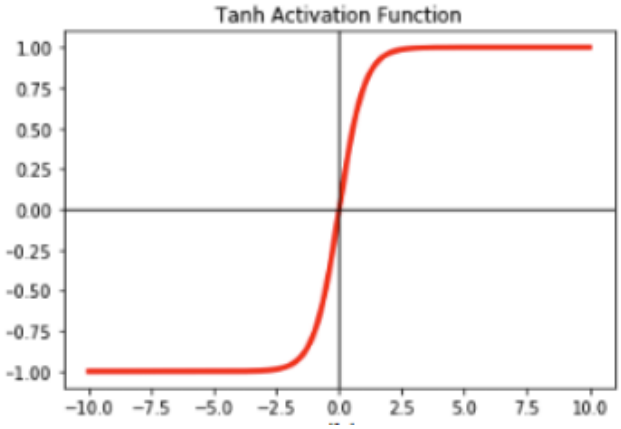
$$f'(z) = f(z)(1 - f(z)) \tag{7}$$




- Tanh function

$$f(z) = \frac{e^{z} - e^{-z}}{e^{z} + e^{-z}} \tag{8}$$

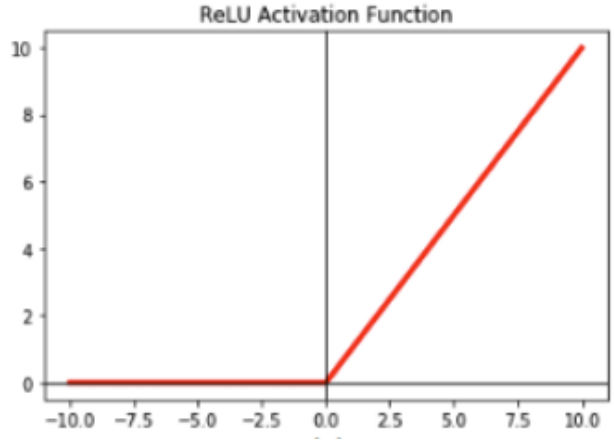
$$f'(z) = 1 - f(z)^2 \tag{9}$$




- ReLU function

$$f(z) = \max(0, z) \tag{10}$$

$$f'(z) = \begin{cases} 1 & \text{if } z > 0 \\ 0 & \text{if } z < 0 \\ \text{undefined} & \text{if } z = 0 \end{cases} \tag{11}$$




- GAUSSIAN ERROR LINEAR UNITS (GELUS)

  a high-performing neural network activation function. The GELU activation function is $x\Phi(x)$, where $\Phi(x)$ the standard Gaussian cumulative distribution function. The GELU nonlinearity weights inputs by their value, rather than gates inputs by their sign as in ReLUs ($x1_x > 0$).

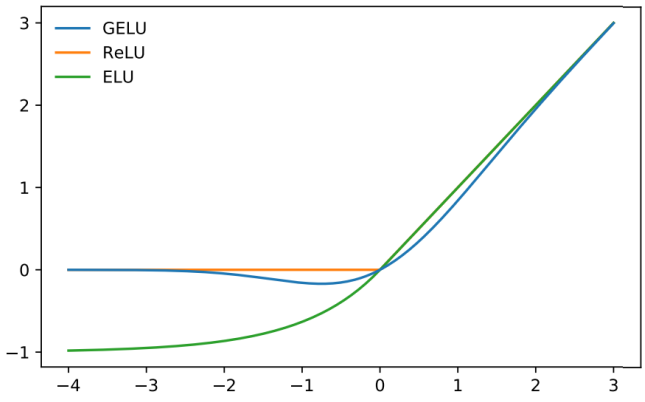
$$\text{GELU}(x) = x\,\Phi(x) = \tfrac{1}{2}x\big(1 + \text{erf}(x/\sqrt{2})\big).$$

$$\frac{d}{dx}\,\text{GELU}(x) = \Phi(x) + x\,\phi(x), \quad \phi(x) = \tfrac{1}{\sqrt{2\pi}}e^{-x^2/2}.$$

Tanh approximation:

$$\text{GELU}_{\tanh}(x) = \tfrac{1}{2}x\Big(1 + \tanh\big(\sqrt{\tfrac{2}{\pi}}(x + 0.044715x^3)\big)\Big),$$

$$\frac{d}{dx}\,\text{GELU}_{\tanh}(x) = \tfrac{1}{2}\Big[1 + \tanh u(x)\Big] + \tfrac{1}{2}x\,\text{sech}^2\,u(x)\cdot u'(x), \quad u'(x) = \sqrt{\tfrac{2}{\pi}}(1 + 3\cdot 0.044715\,x^2).$$



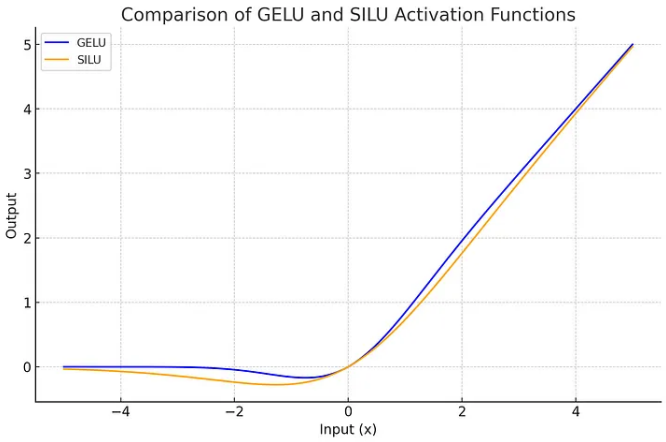

Pros: smooth gating, strong empirical results in Transformers. Cons: slightly heavier than ReLU/SiLU.

- SiLU / Swish

  Swish is a smooth, non-monotonic function that does not consistently increase or decrease; activation function defined as :

$$\mathrm{SiLU}(x) = x\,\sigma(x), \qquad \sigma(x) = \frac{1}{1+e^{-x}}, \qquad \mathrm{Swish}_\beta(x) = x\,\sigma(\beta x).$$

$$\frac{d}{dx}\,\mathrm{SiLU}(x) = \sigma(x)\Big(1 + x[1-\sigma(x)]\Big).$$




Pros: smooth, efficient; common in diffusion. Cons: slightly costlier than ReLU.

- GLU

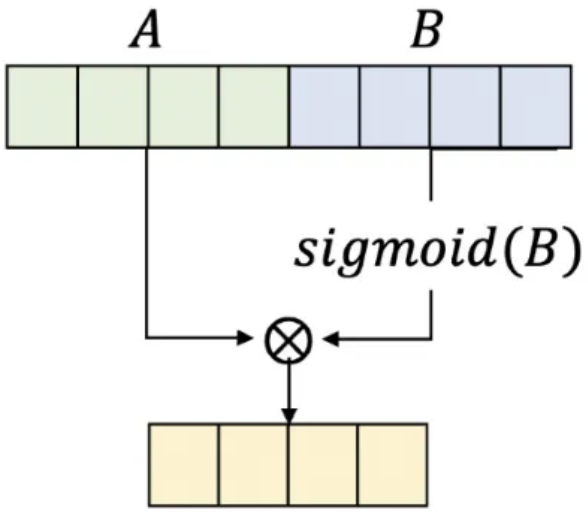
  The idea behind this function is that it takes the output of a linear transformation and splits it into two parts: one part is passed through another linear transformation, while the second is passed through a sigmoid activation function.

Let $a = xW_a + b_a$, $b = xW_b + b_b$.

$$\text{ReGLU: } y = \text{ReLU}(a) \odot b, \quad \text{GEGLU: } y = \text{GELU}(a) \odot b, \quad \text{SwiGLU: } y = \text{Swish}(a) \odot b.$$

Gradients:

$$\frac{\partial y}{\partial a} = g'(a) \odot b, \quad \frac{\partial y}{\partial b} = g(a), \quad \frac{\partial y}{\partial W_a} = x^\top (g'(a) \odot b), \quad \frac{\partial y}{\partial W_b} = x^\top g(a).$$



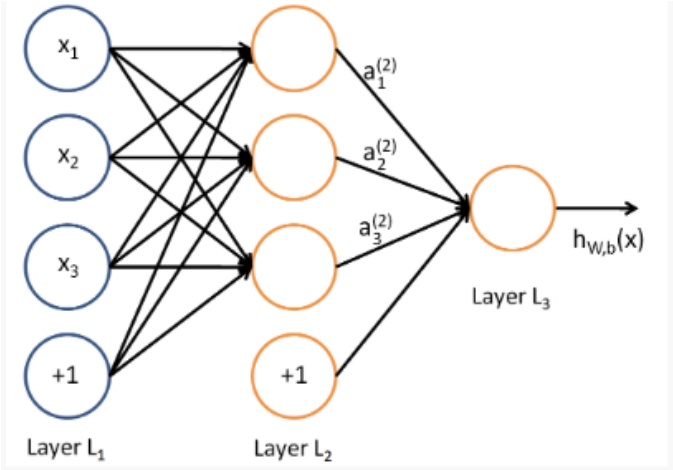

Pros: better perplexity/efficiency; Cons: slight added compute.

For additional clarity, refer to this Understanding Deep Learning.

## Neural network structure

A neural network is created by connecting several nodes so that the output of some nodes serves as the input to others.

In a neural network, the layers of nodes are organized into three main types: the input layer, the hidden layer, and the output layer. The input layer consists of nodes that represent the input features of the data, and it also includes special bias units. The hidden layer is where the network processes the information through weights and biases to learn patterns. This layer's nodes are not directly observed in the data but play a crucial role in transforming the inputs into meaningful outputs. Finally, the output layer provides the final prediction or result of the network. In our example, there are 3 input nodes, 3 hidden nodes, and 1 output node.




### Connections

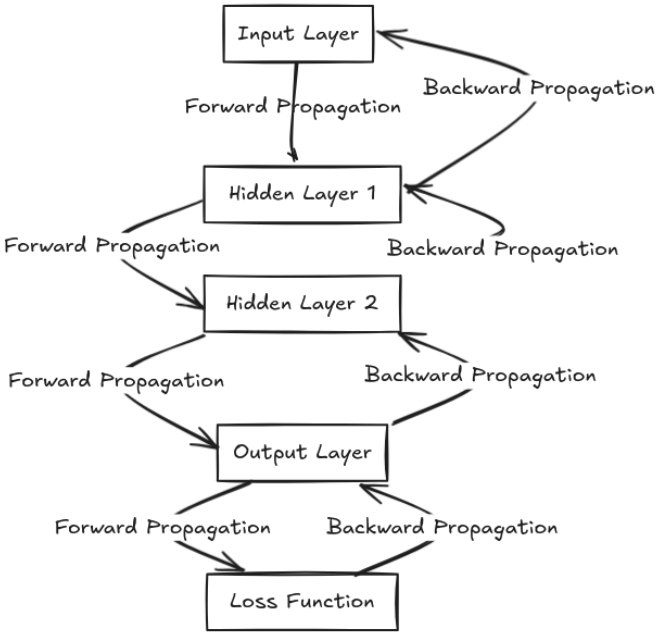
- **Weights:** Each connection between nodes has a weight that controls how much influence one node has on another. Weights are adjusted during training to improve the network's predictions.

$$\begin{bmatrix} w_{11}^L & w_{12}^L & \cdots & w_{1S_L}^L \\ w_{21}^L & w_{22}^L & \cdots & w_{2S_L}^L \\ \vdots & \vdots & \ddots & \vdots \\ w_{S_{L+1}1}^L & w_{S_{L+1}2}^L & \cdots & w_{S_{L+1}S_L}^L \end{bmatrix} \tag{12}$$

- **Bias:**: Each node in the hidden and output layers has an associated bias. Biases help the model fit the data better by allowing the activation function to shift its output.

$$b_L = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_{S_{L+1}} \end{bmatrix} \tag{13}$$

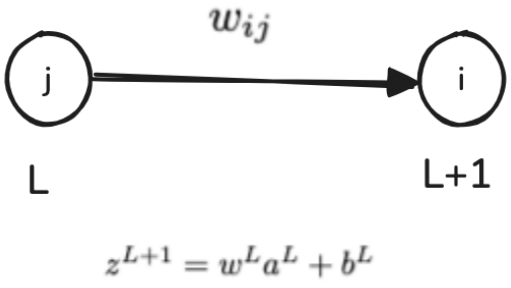
## How does a neural network learn?



A neural network learns through a back propagation technique. First, data will be propagate forward through the network which produces output.
The weighted sum of inputs is given by:

$$z_i^{(L+1)} = \sum_{j=1}^{S_L} W_{ij}^{(L)} a_j^{(L)} + b_i^{(L)} = W_i^{(L)} a^{(L)} + b_i^{(L)} \quad (14)$$

$$z^{(L+1)} \in \mathbb{R}^{S_{L+1}} \begin{bmatrix} z_1^{(L+1)} \\ z_2^{(L+1)} \\ \vdots \\ z_{S_{L+1}}^{(L+1)} \end{bmatrix} \quad (15)$$




The weighted sum is then passed through an activation function to introduce non-linearity into the model. This helps the network learn more complex patterns. This process is repeated for each layer in the network until the final output layer is reached. Each layer computes a new weighted sum, applies an activation function, and passes the result to the next layer.

$$a_i^{(L+1)} = f(z_i^{(L+1)}) \quad (16)$$

$$a^{(L+1)} \in \mathbb{R}^{S_{L+1}} \begin{bmatrix} a_1^{(L+1)} \\ a_2^{(L+1)} \\ \vdots \\ a_{S_{L+1}}^{(L+1)} \end{bmatrix} \quad (17)$$

To generalize for a layer $n$, activation function can be given as:

$$a_i^{(L_n)} = f(w_{i,:}^{(L_{n-1})} a^{(L_{n-1})} + b_i^{(L_{n-1})}) = h_{(w,b)}(x) \quad (18)$$

Lets say we're calculating activation function and weighted sum for layer 4 then we have,

$$z^{(4)} = W^{(3)} a^{(3)} + b^{(3)} \quad (19)$$

$$a^{(4)} = f(z^4) \quad (20)$$

The process continues through each layer until the output layer. The output layer provides the final prediction $\hat{y}_i$, which can vary depending on the task:

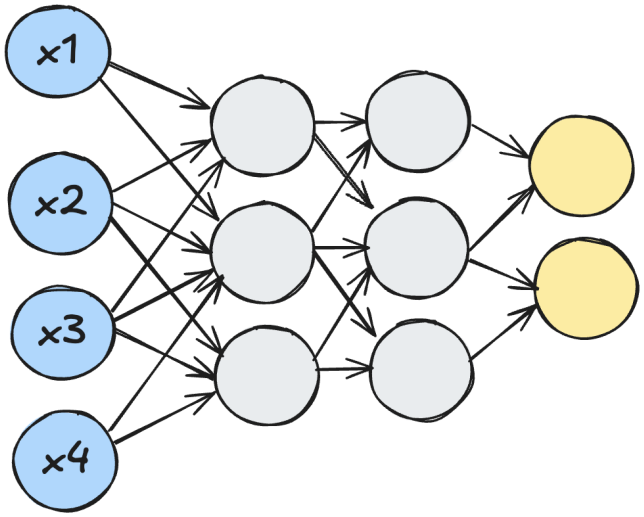
- **Regression**: For regression tasks, the final layer typically has a single neuron (or multiple neurons if multi-output), with no activation or a linear activation (identity function), outputting the predicted value directly.
- **Classification**: For classification tasks, the output layer may have multiple neurons, each representing a class. Typically, a *softmax* activation function is applied in the output layer to convert the raw scores (log-its) into probabilities for each class.

## Loss Calculation

Once the network produces the predicted output $\hat{y}$, the next step in forward propagation is calculating the loss, which measures how far the predictions are from the actual labels.

- **Regression Loss (Mean Squared Error)**: For regression problems, the *Mean Squared Error (MSE)* is a common loss function. The difference between the predicted values $\hat{y}$ and the true labels $y$ is squared and averaged across all data points.
- **Classification Loss (Cross-Entropy Loss)**: For classification tasks, the *Cross-Entropy Loss* (also known as *Log Loss*) is commonly used. It measures the difference between the true label distribution and the predicted probability distribution.

The term "feedforward" refers to the fact that the data flows in a single direction, **forward**, from the input layer through the hidden layers to the output layer. There are no loops or cycles, meaning the output of one layer is only passed to the next layer, without going back to previous layers.




## Back Propagation

Back propagation is a key process in training neural networks, and it's all about helping the network learn by adjusting its internal settings, called **weights**.

A simple way to think about back propagation:

- **Forward Propagation**: First, you give the network some input (like a sentence from a movie review), and it makes a prediction (an output). For example, in a sentiment analysis task, the network might say, "This review seems 85% positive and 15% negative."
- **Error Calculation**: The network's prediction is compared to the correct answer (called the "ground truth"). The difference between what the network predicted and the actual answer is the error (or loss). In our movie review example, if the correct label is 100% positive, the difference between 85% and 100% is the error.
- **Backpropagation**: Now comes the tricky part: *backpropagation*. The goal here is to figure out how to adjust the network's weights so that it makes better predictions next time. To do this:
  - We start from the output layer and move backward through the network.
  - For each connection (between nodes), we calculate how much it contributed to the overall error. This is done using chain rule of calculus.
  - Once we know how each weight contributed to the error, we adjust them to **reduce the error**. The idea is to nudge the weights in a direction that makes future predictions more accurate.
- **Weight Update**: Finally, after determining how much each weight needs to change, we actually update the weights. After this, the network is a little bit better at making predictions.

## Mathematical representation

Let $(x^{(i)}, y^{(i)})$ for $i = 1...m$ denote our training samples: we want to learn weights

$w^L \in R^{S_{L+1} \times S_L}$ and bias terms $b^L \in R^{S_{L+1}}$ for $L = 1...N-1$ so as to minimize the cost:

$$J(w, b; x^{(i)}, y^{(i)}) \triangleq \frac{1}{2} \| h_{w,b}(x^{(i)}) - y^{(i)} \|_2^2$$

The network computes the error for each training example and then averages it to get an overall measure of how well it is doing. Depending on the final task (classification, regression or etc.) We must scale outputs $y^i$.

For neural networks, the cost function is typically **non-convex** due to complex structure and interaction between multiple layers. Despite the non-convexity, we use gradient descent to find a good set of weights that minimizes the cost function.

$$\omega^{(L)} := \omega^{(L)} - \alpha \frac{\partial J_(\omega, b)}{\partial \omega^{(L)}} \tag{21}$$

$$b^{(L)} := b^{(L)} - \alpha \frac{\partial J_(\omega, b)}{\partial b^{(L)}} \tag{22}$$

Notice that,

$$\frac{\partial J_(\omega, b)}{\partial \omega^{(L)}} = \frac{1}{m} \sum_{i=1}^{m} \frac{\partial J\left(\omega, b, x^{(i)}, y^{(i)}\right)}{\partial \omega^{(L)}} \tag{23}$$

$$\frac{\partial J_(w, b)}{\partial b^{(L)}} = \frac{1}{m} \sum_{i=1}^{m} \frac{\partial J\left(\omega, b, x^{(i)}, y^{(i)}\right)}{\partial b^{(L)}} \tag{24}$$

To compute gradients of $\omega^L$ and $b^L$, we first consider gradient of cost wrt the input of unit i in layer L for L = n, n-1,...2

$$J(\omega, b, x, y) = \frac{1}{2} \left\| h_{w,b}(x) - y \right\|_2^2 = \frac{1}{2} \sum_{k=1}^{S_n} \left( a_k^{(n)} - y_k \right)^2 = \frac{1}{2} \sum_{k=1}^{S_n} \left( f\left( z_k^{(n)} \right) - y_k \right)^2 \tag{25}$$

$$\text{Let } \delta_i^{(n)} \triangleq \frac{\partial J}{\partial z_i^{(n)}} = \left( f\left( z_i^{(n)} \right) - y_i \right) \cdot f'\left( z_i^{(m)} \right) = \left( a_i^{(n)} - y_i \right) f'\left( z_i^n \right); i = 1, 2, \ldots s_n \tag{26}$$

$\triangleq$ is used to mean "defined as" or "by definition"

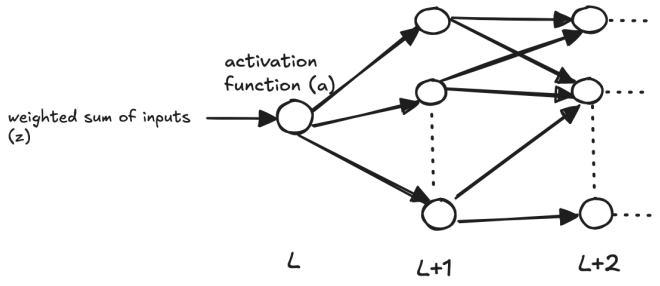



Figure 3: Neural network structure

$$\delta_i^{(L)} \triangleq \frac{dJ}{\partial z_i^{(L)}} = \sum_{k=1}^{S_{L+1}} \frac{\partial J}{\partial z_k^{(L+1)}} \times \frac{\partial z_k^{(L+1)}}{\partial z_i^{(L)}} \tag{27}$$

where we used:

$$z^{(L+1)} = w^{(L)} a^{(L)} + b^{(L)} = w^{(L)} f\left(z^{(L)}\right) + b^{(L)} \tag{28}$$

Taking the derivative we get,

$$\frac{\partial z_k^{L+1}}{\partial z_i^L} = w_{ki}^L f'\left(z_i^{(L)}\right) \tag{29}$$

We can now compute the weights and biases using chain rule of derivatives like so:

$$\frac{\partial J}{\partial w_{ij}^{(L)}} = \frac{\partial J}{\partial z_i^{(L+1)}} \cdot \frac{\partial z_i^{(L+1)}}{\partial w_{ij}^{(L)}} = \delta_i^{(L+1)} \times a_j^{(L)} \tag{30}$$

$$\frac{\partial J}{\partial b_i^{(L)}} = \frac{\partial J}{\partial z_i^{(L+1)}} \cdot \frac{\partial z_i^{(L+1)}}{\partial b_i^{(L)}} = \delta_i^{(L+1)} \tag{31}$$

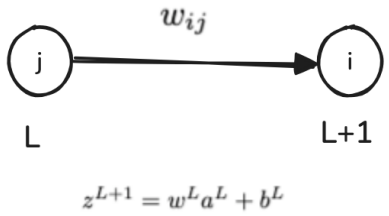



Figure 4: Weight between node j and i

We can now write back propagation in the matrix-vector form as:

$$\delta^n = \left(a^{(n)} - y\right) \odot f'\left(z^{(n)}\right) \tag{32}$$

The symbol $\odot$ represents the Hadamard product (also known as the element-wise product)

$$\delta^L = \left(w^{(L)^\top} \cdot \delta^{(L+1)}\right) \odot f'\left(z^{(L)}\right); L = n-1, \ldots, 2 \tag{33}$$

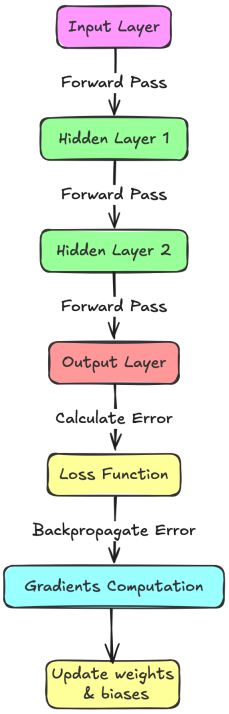
$$\frac{dJ}{d_w^{(L)}} = \delta^{(L+1)} \cdot a^{(L)^\top} \tag{34}$$

$$\frac{\partial J}{\partial b^{(L)}} = \delta^{(L+1)} \tag{35}$$

Notice that, to obtain updates we need to first run the forward propagation with the current weights to obtain the activation and inputs $a^L, z^L$

## Steps involved in back propagation algorithm

- Initialize all weights $W_{i,j}^{(L)}$ and biases $b_j^{(L)}$ with random values $\sim \mathcal{N}\left(0, 0.01^2\right)$
- For $t = 1, \ldots, T$ , repeat:
  - Set $\Delta W^{(L)} = 0$, $\Delta b^{(L)} = 0$, $\forall L = 1, \ldots, n-1$
  - For $i = 1, \ldots, m$ , repeat:
    - Apply forward propagation to compute $z^{(L)}(x_i)$ and $a^{(L)}(x_i)$
    - Use backpropagation to compute $\nabla_{w^{(L)}} J(w, b, x_i, y_i)$ , $\nabla_{b^{(L)}} J(w, b, x_i, y_i)$
    - Set $\Delta W^{(L)} := \Delta W^{(L)} + \nabla_{w^{(L)}} J(w, b, x_i, y_i)$
    - Set $\Delta b^{(L)} := \Delta b^{(L)} + \nabla_{b^{(L)}} J(w, b, x_i, y_i)$
  - Update parameters:
    - $w^{(L)} := w^{(L)} - \alpha\left(\frac{1}{m}\Delta W^{(L)}\right)$
    - $b^{(L)} := b^{(L)} - \alpha\left(\frac{1}{m}\Delta b^{(L)}\right)$




Thus,

$$J_\lambda(w, b, x_i, y_i) = \frac{1}{N}\sum_{i=1}^{N} J(x_i, y_i) + \frac{\lambda}{2}\sum_{L}\left\|w^{(L)}\right\|_F^2 \tag{36}$$

## Stochastic Gradient Descent (SGD)

SGD computes the gradient using a single data point (or a small batch) and updates the parameters more frequently, leading to faster but noisier updates. SGD is more efficient for large datasets and often generalizes better due to its inherent randomness, while GD can be computationally expensive and prone to overfitting.

It follows similar steps to GD, but unlike GD which repeats step 2 For i=1, . . . , N , repeat: , SGD does for i in minibatch of size S

## Gradient checking

We can always check whether in our implementation, gradient $\nabla_{w^{(L)}} J_\lambda^{(w,b)} = \frac{1}{m}\Delta w^{(L)} + \lambda w^{(L)}$ and $\nabla_{b^{(L)}} J_\lambda^{(w,b)} = \frac{1}{m}\Delta b^{(L)}$ are correctly implemented.

To do so, compute $\frac{J_\lambda\left(\omega^{(l)} + \varepsilon E_{ij}\right) - J_\lambda\left(w^{(L)} - \varepsilon E_{ij}\right)}{2\varepsilon}$ with say $\varepsilon \approx 1^{-3}$

Here, we fix all other weight matrices and biases and only change $w_{ij}^L$ to compute $\frac{\partial J_\lambda}{\partial \omega_{ij}^l}$. The result of this should be close to implementation $\nabla_{w^L} J_\lambda^{(w,b)}$ to say 4 digits.

There are other approaches to optimizing $J_\lambda(\cdot)$:

- L-BFGS: Compute an approximate Hessian and use a step size that gives the highest reduction of $J_\lambda(\cdot)$