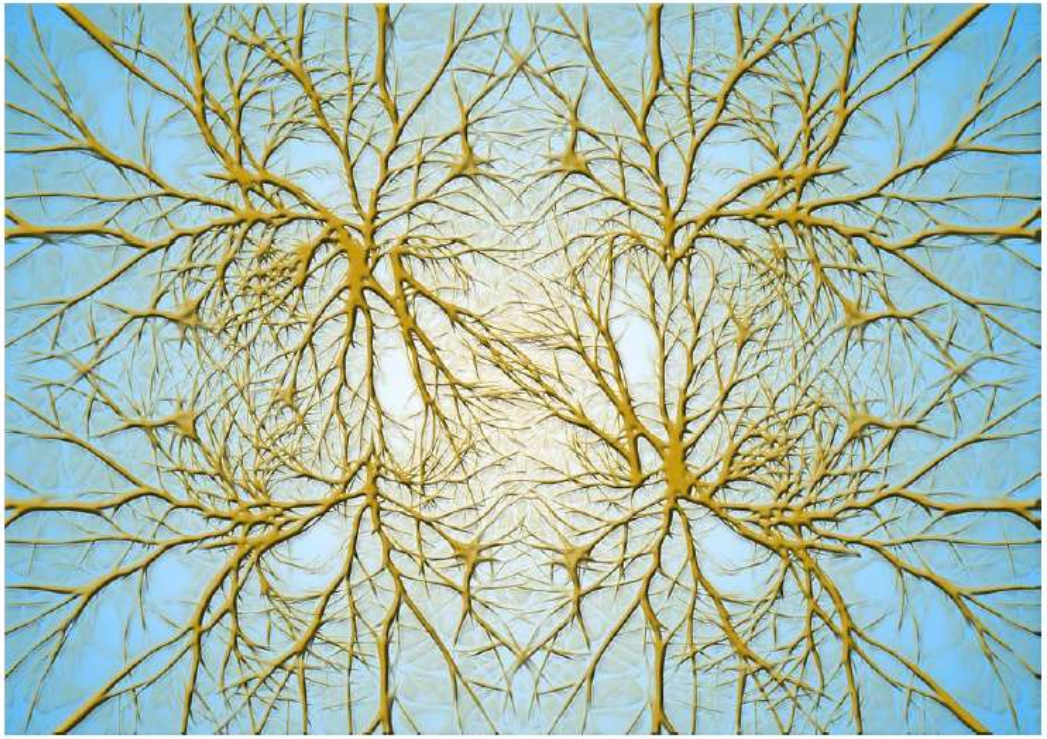
ANN : Multi Layered Perceptron (MLP)

Sourav Karmakar

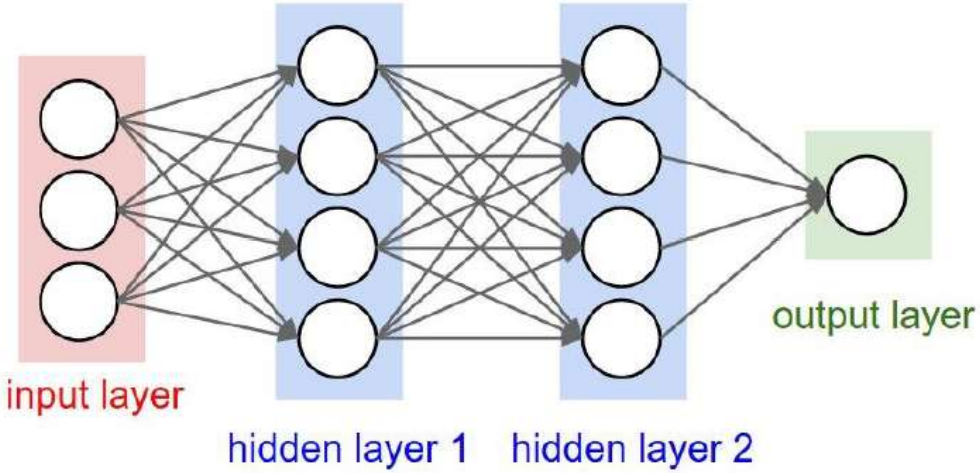
souravkarmakar29@gmail.com

Artificial Neural Networks

Biological Neurons:
Complex connectivity patterns

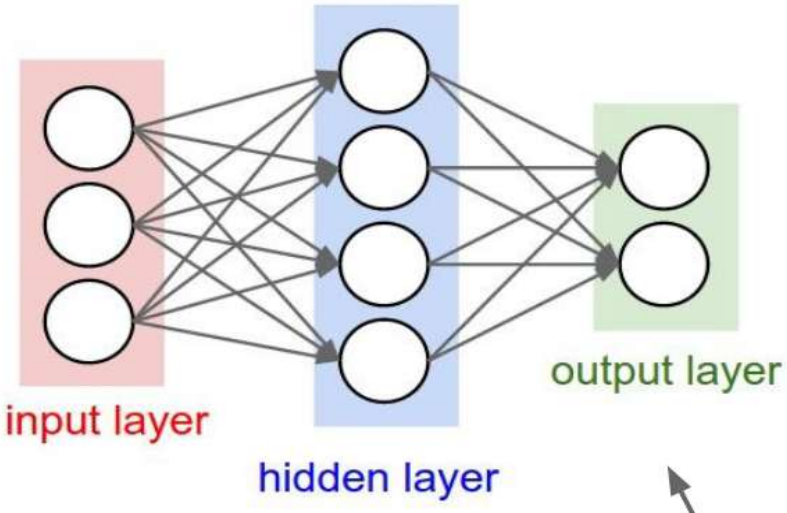


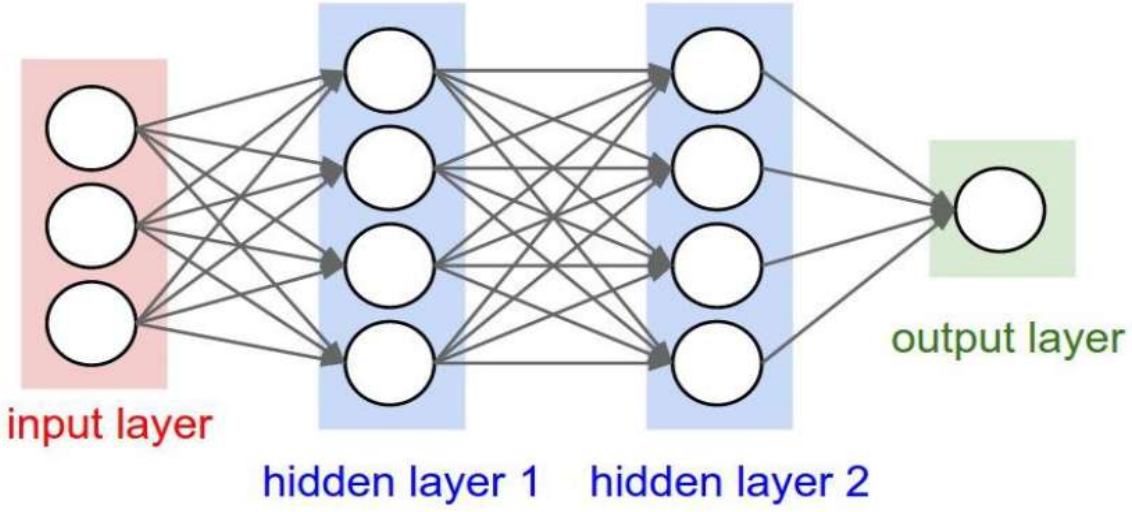
Neurons in a neural network:
Organized into regular layers for computational efficiency



This type of architectures are called **Multi-Layered Perceptron (MLP)**

Multi Layered Perceptron (MLP)





"3-layer Neural Net", or
"1-hidden-layer Neural Net"

"4-layer Neural Net", or
"2-hidden-layer Neural Net"

"Fully-connected" layers

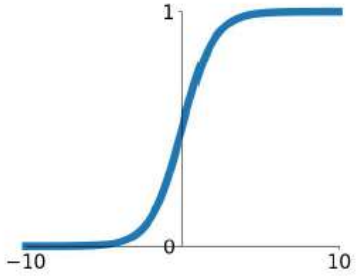
Fully-connected layers are also called **Dense Layers**

Sometimes, people don't consider input layer as a layer because it has no computation in it. In that case the first diagram will be called "2-layer Neural Net" and the second one "3-layer Neural Net".
But in our discussion in this deck, we will consider input layer as a separate non-computing layer.

Different Activation Functions

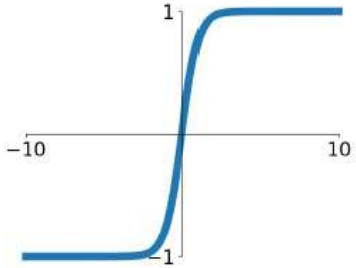
Sigmoid

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



tanh

$$\tanh(x)$$



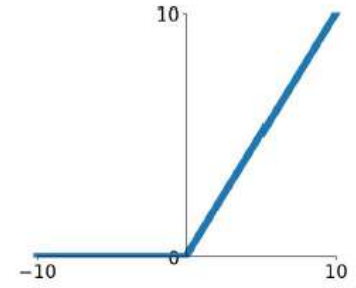
Rectified Linear Unit

ReLU

$$\max(0, x)$$

if $x > 0$, ReLU(x) = x

$x \leq 0$, ReLU(x) = 0



ReLU is a good practical choice for most problems

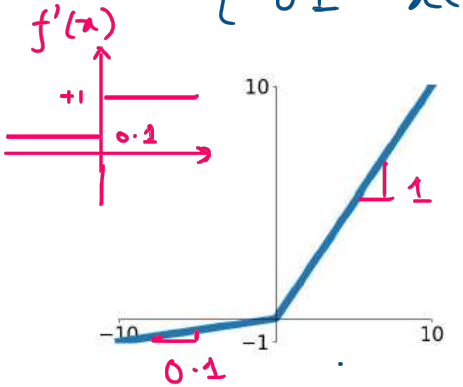
$$f'(x) = \begin{cases} 1 & x \geq 0 \\ 0.1 & x < 0 \end{cases}$$

Leaky ReLU

$$\max(0.1x, x)$$

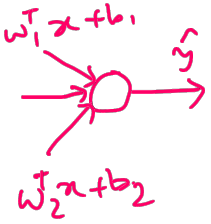
if $x \geq 0$; $f(x) = x$

$x < 0$; $f(x) = 0.1x$



Maxout (Rarely Used)

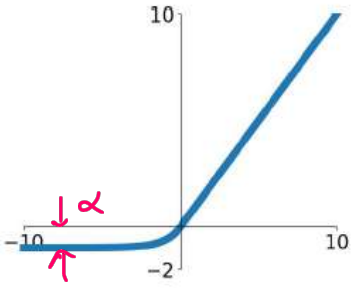
$$\max(w_1^T x + b_1, w_2^T x + b_2)$$



Exponential Linear Unit

ELU

$$\begin{cases} x & x \geq 0 \\ \alpha(e^x - 1) & x < 0 \end{cases}$$



↳ control parameter : saturation value

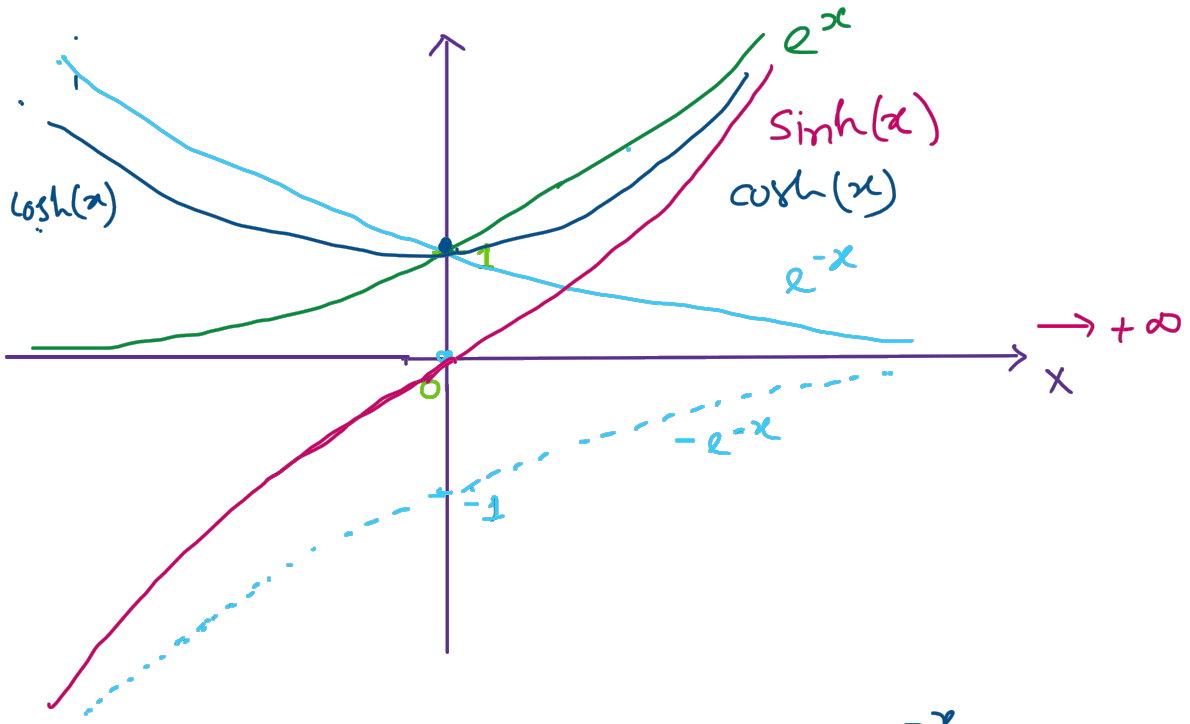
as $x \to -\infty$ ∴ ELU(x) = $-\alpha$

Hyperbolic Functions :—

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

$\sinh(0) = 0$

$-\infty \leftarrow$



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

$\cosh(0) = 1$

$$\tanh(x) = \frac{\sinh(x)}{\cosh(x)} = \frac{e^x - e^{-x}}{e^x + e^{-x}}$$

Range of $\tanh(x)$ is -1 to $+1$

$x \rightarrow \infty$

$\tanh(x) = +1$

$x \rightarrow -\infty$; $\tanh(x) = -1$

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

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

$$= \frac{4\cdot e^{2x}\cdot 1}{(e^{2x}+1)^2} \Leftarrow \frac{(e^{2x}+1)^2 - (e^{2x}-1)^2}{(e^{2x}+1)^2} = 1 - \left(\frac{e^{2x}-1}{e^{2x}+1}\right)^2$$

$$(a+b)^2 - (a-b)^2 = 4ab$$

$$\frac{d}{dx}(\tanh(x)) = 1 - (\tanh(x))^2$$

$$\frac{d}{dx}(\sigma(x)) = \sigma(x)\cdot(1-\sigma(x))$$

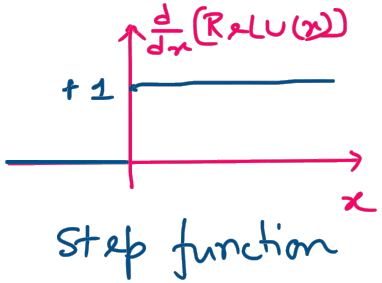
$$\sigma(x) = \frac{1}{1+e^{-x}}$$

$$\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}} \in [-1, +1)$$

$$\frac{d}{dx}\tanh(x) = 1 - \tanh^2(x)$$

$$\left.\frac{d}{dx}\tanh(x)\right|_{x=0} = 1$$

$$\text{ReLU}(x) = \begin{cases} 0 ; & x \leq 0 \\ x ; & x > 0 \end{cases} \Rightarrow \frac{d}{dx}[\text{ReLU}(x)] = \begin{cases} 0 ; & x \leq 0 \\ 1 ; & x > 0 \end{cases}$$



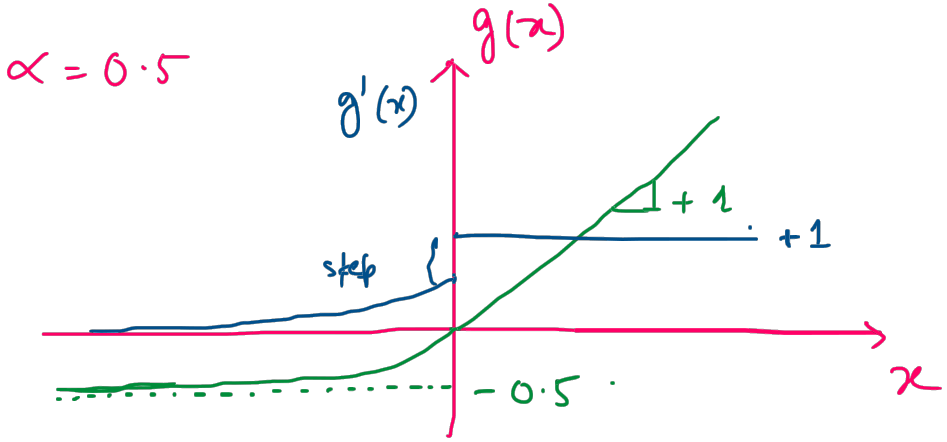
Step function

$$\text{ELU}(x) = g(x) = \begin{cases} x ; & x \geq 0 \\ \alpha(e^x - 1) ; & x < 0 \end{cases}$$

$$g'(x) = \begin{cases} 1 ; & x \geq 0 \\ \alpha e^x ; & x < 0 \end{cases}$$

$x \to 0-$

$g'(x) = \alpha$



GELU(x) → Gaussian Error Linear Unit

$\text{GELU}(x) = x \cdot \Phi(x)$

$\Phi(x)$ = CDF of standard normal distribution

The Role of Activation Functions

Without Activation Function:

- Suppose we only use **linear operations** in a network: $y = Wx + b$
- Even if we stack multiple layers, the output is still just a **linear transformation** of the input.

$$y = W_3(W_2(W_1x + b_1) + b_2) + b_3 = W'x + b'$$

- That means no mater how many layers you add, the whole network is just a linear model. Hence, without activation function the model can't capture the complex non-linear relationship in the data.

Purpose of Activation Function:

1. Introduce Non-linearity:
    - Helps capture complex patterns in data by introducing non-linearity.
    - With nonlinear activations, neural networks become **universal function approximators** (proved by the **Universal Approximation Theorem**)
2. Control Information Flow:
    - Decide which neurons "fire" (activate) and which don't.
    - Makes the network selective about what information passes forward.
3. Bounded Outputs / Normalization:
    - Some activations (like sigmoid, tanh) squash outputs into a small range, making training stable.
4. Gradient Flow: Differentiable activation functions allow gradient-based learning (backpropagation)

Softmax Regression

Softmax Regression is a generalization of Logistic Regression for multiclass classification problems.

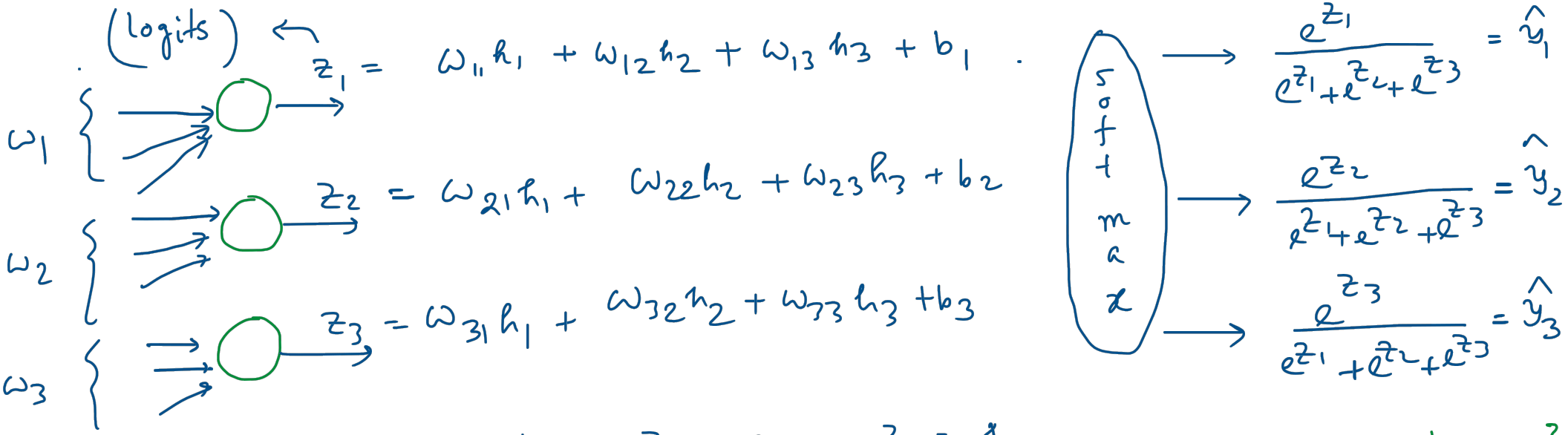
- **Training Set:** $\{\langle \boldsymbol{x}^{(1)}, y^{(1)} \rangle, \langle \boldsymbol{x}^{(2)}, y^{(2)} \rangle, \dots, \langle \boldsymbol{x}^{(m)}, y^{(m)} \rangle\}$,

  Where m = Number of training data, the feature vector $\boldsymbol{x}^{(i)} \in \mathbb{R}^{n+1}$ $(x_0 = 1)$, and the labels $y^{(i)} \in \{1, 2, 3, \dots, K\}$ $(K =$ Number of classes).

- **Hypothesis:** $h_\theta(\boldsymbol{x}^{(i)}) = \frac{1}{\sum_{j=1}^{K} e^{\boldsymbol{\theta}_j^T \boldsymbol{x}^{(i)}}} \begin{bmatrix} e^{\boldsymbol{\theta}_1^T \boldsymbol{x}^{(i)}} \\ e^{\boldsymbol{\theta}_2^T \boldsymbol{x}^{(i)}} \\ \vdots \\ e^{\boldsymbol{\theta}_K^T \boldsymbol{x}^{(i)}} \end{bmatrix}$

  Where $\boldsymbol{\theta}_1, \boldsymbol{\theta}_2, \dots, \boldsymbol{\theta}_K \in \mathbb{R}^{n+1}$, are the parameters of our model. The denominator $\sum_{j=1}^{K} e^{\boldsymbol{\theta}_j^T \boldsymbol{x}^{(i)}}$ normalizes the distribution, so that it sums to one.

Usually Softmax regression is used at the output / classification layer of the ANN. Where it is also referred to as **Softmax Activation function**.

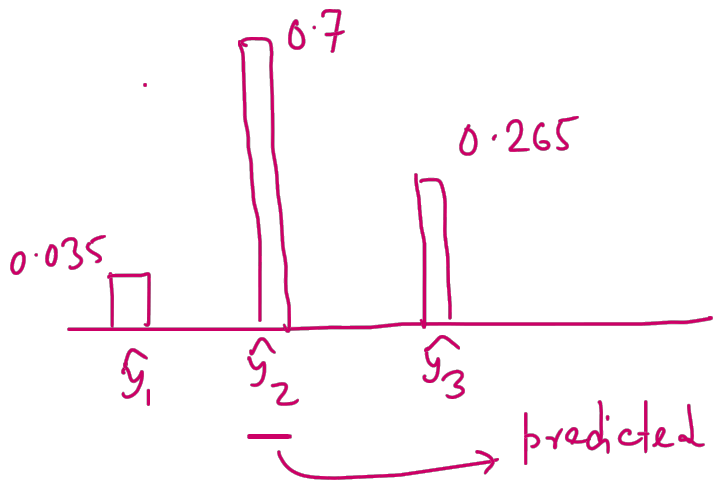


logits: $z_1 = -1, \quad z_2 = 2, \quad z_3 = 1$

$$\hat{y}_1 = \frac{e^{-1}}{e^{-1}+e^{2}+e^{1}} = 0.035$$

$$\hat{y}_2 = \frac{e^{2}}{e^{-1}+e^{2}+e^{1}} = 0.70$$

$$\hat{y}_3 = \frac{e^{1}}{e^{-1}+e^{2}+e^{-1}} = 0.265$$



0 1 2

[0.035, 0.7, 0.265]

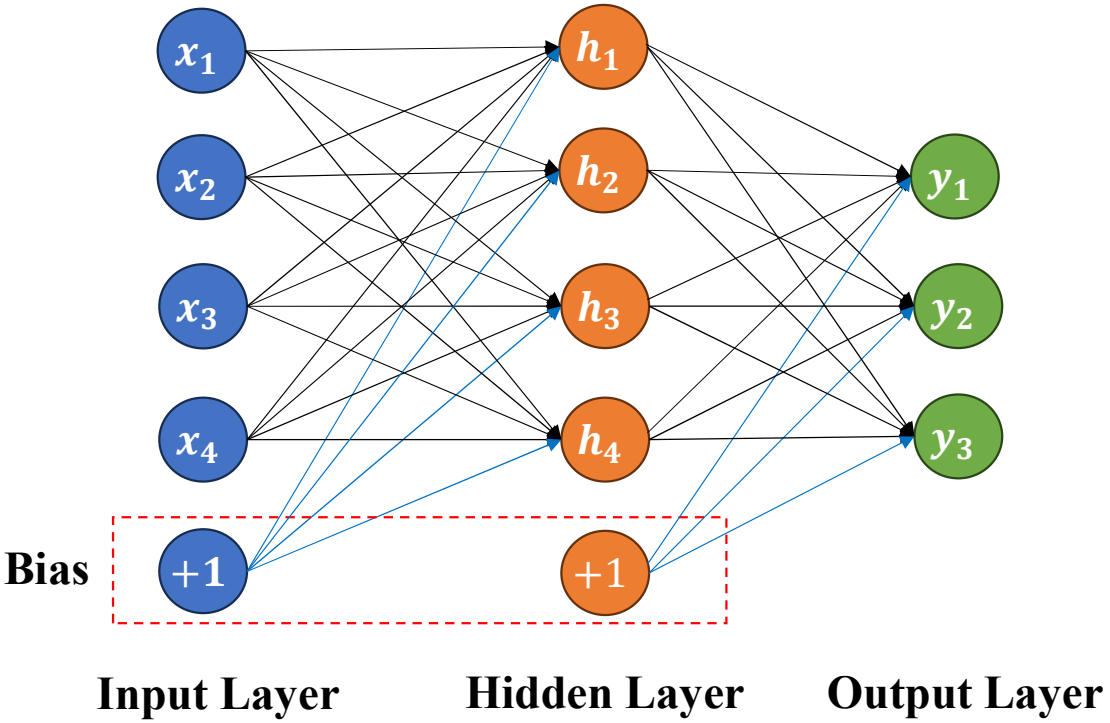
Multi Layered Perceptron (MLP): Architecture

- A layer of sensory nodes called input nodes and the layer is called input layer. No computation happens in input nodes.
- The number of nodes in the input layer = number of features or attributes of the data.
- Layers of computing nodes. These are called **hidden layers**. In successive layers, the nodes are all connected by learnable weights. This is called 'dense architecture'.
- Number of Hidden layers and Number of nodes in each hidden layer are user choice (hyper-parameters) and there is no general guideline for the choice of number of hidden layers or the number of nodes in each hidden layers. However, there should be at least 1 hidden layer.
- Layer of final computing nodes is called output layer.
- The number of nodes in the output layer = number of classes of the dataset. (for a classification problem). For multi-class classification it is usually called 'softmax' layer due to the softmax regression. For binary classification only one node with sigmoid activation is needed. For regression problem, there will be only one output node without any activation function.
- Usually we use **'ReLU'** activation for all nodes in hidden layer and **'softmax'** activation in output layer for multi-class classification problem or **'sigmoid'** activation in output layer for binary classification problem. However we can also use different activation functions in the hidden layer.

Multi Layered Perceptron (MLP): Architecture

For example: To classify the simple IRIS dataset using ANN

- The number of input nodes = 4 (as the dataset has four features / attributes)
- The number of output nodes = 3 (as there are three classes)
- We can build an ANN with one hidden layer containing 4 nodes (say)



- Number of connections from input layer to hidden layer:

$$(4 + 1) \times 4 = 20$$

Number of input nodes — Bias — Number of nodes in hidden layer

- Number of connections from hidden layer to output layer:

$$(4 + 1) \times 3 = 15$$

Number of nodes in hidden layer — Bias — Number of nodes in output layer

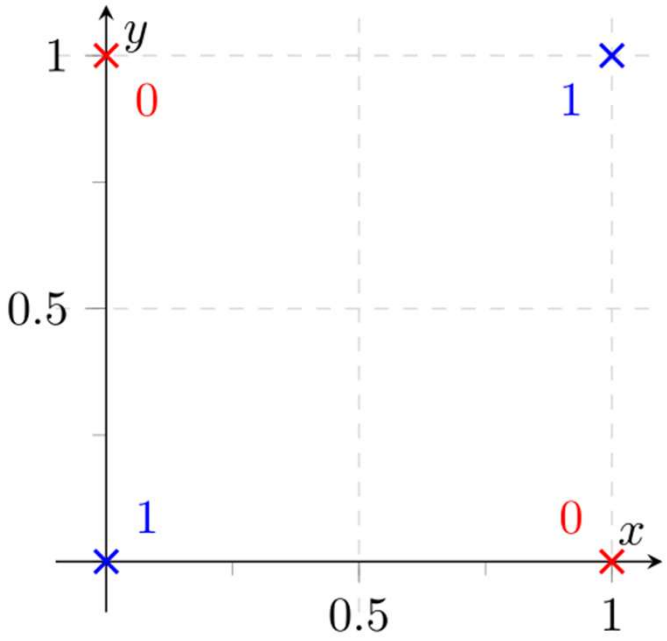
Multi Layered Perceptron (MLP): Advantages

- **Non-linearity:** MLP is very suitable to model a physical phenomenon which is in general non-linear.
- **Automatic Feature Extraction:** Each layer in MLP learns a representation of the data. The network learns features automatically during training by adjusting weights. The learned features are optimal for final task as MLP is an universal function approximator. The first few hidden layers learns low level patterns, where as the deeper layers learn high-level representation. This is called **representation learning**.
- **Adaptivity:** It can cope with the change in dataset distribution.
- **Parallelism:** In can be implemented in multi-core parallel architecture (like GPU) for faster computation.
- **Robustness:** It has inherent ability to handle confusing / noisy / missing datapoints
- **Fault tolerance:** It has the ability to work to some extent even in case of component (neuron) failure.
- **Complex decision boundary:** It has the ability to have a complex decision boundary for classification.

We shall use MLP with supervised learning for complex classification tasks.

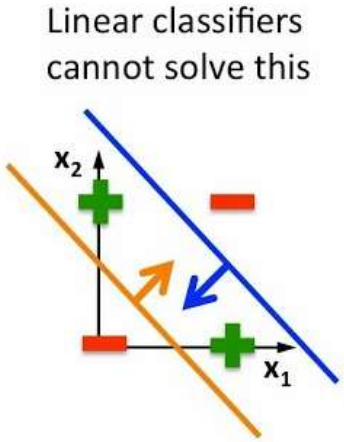
How MLP Solved The XOR Problem

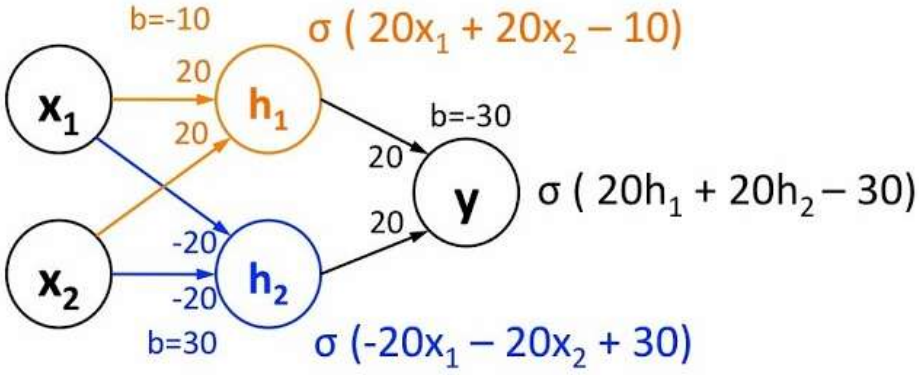
XOR Problem



Input 1	Input 2	Output
0	0	0
0	1	1
1	1	0
1	0	1

Multi Layered Perceptron

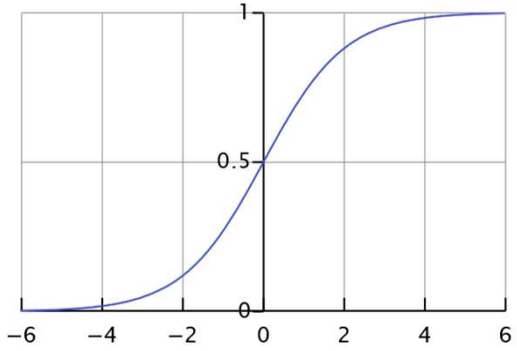




σ(20*0 + 20*0 – 10) ≈ 0
σ(20*1 + 20*1 – 10) ≈ 1
σ(20*0 + 20*1 – 10) ≈ 1
σ(20*1 + 20*0 – 10) ≈ 1

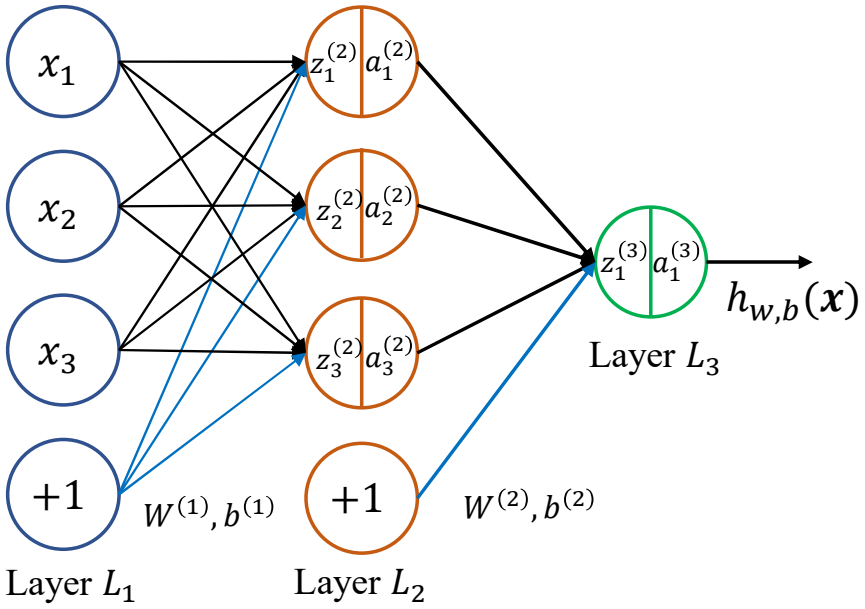
σ (-20*0 – 20*0 + 30) ≈ 1
σ (-20*1 – 20*1 + 30) ≈ 0
σ (-20*0 – 20*1 + 30) ≈ 1
σ (-20*1 – 20*0 + 30) ≈ 1

σ (20*0 + 20*1 – 30) ≈ 0
σ (20*1 + 20*0 – 30) ≈ 0
σ (20*1 + 20*1 – 30) ≈ 1
σ (20*1 + 20*1 – 30) ≈ 1



Multi Layered Perceptron (MLP)

Key Notations:



- n_l denotes the number of layers in our network; thus in the figure beside $n_l = 3$.
- s_l denotes the number of nodes in layer l (not counting the bias unit)
- L_1 is the input layer and L_{n_l} is the output layer.
- $b_i^{(l)}$ is the bias associated with unit i in layer $l + 1$

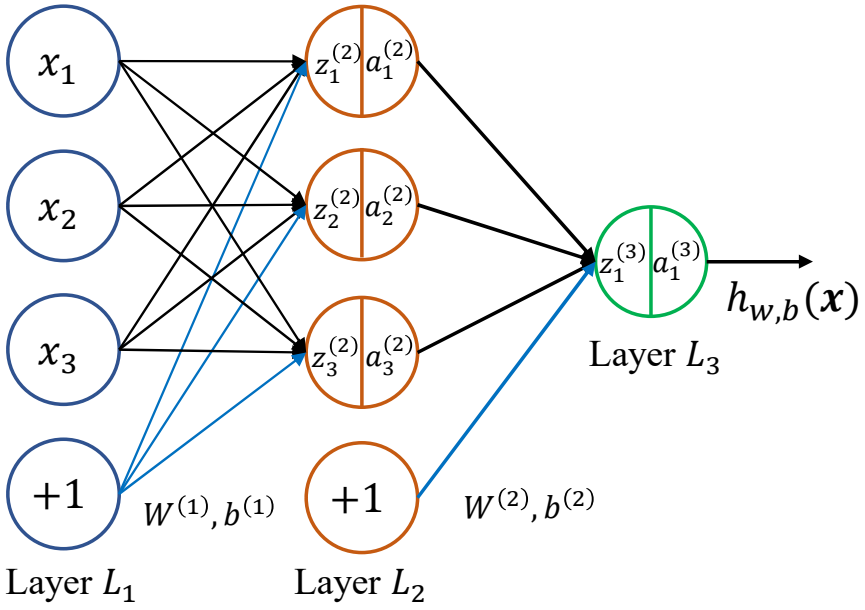
- The neural network as shown above has parameters $\langle W, b \rangle = \langle (W^{(1)}, b^{(1)}), (W^{(2)}, b^{(2)}) \rangle$,

Where $W_{ij}^{(l)}$ denotes the parameter (or weight) associated with the connection between unit j in layer l, and unit i in layer $l + 1$. Note that bias units don't have inputs or connections going into them, since they always output value $+1$.

- In this example we have $W^{(1)} \in \mathbb{R}^{3\times3}$, $b^{(1)} \in \mathbb{R}^{3\times1}$, $W^{(2)} \in \mathbb{R}^{1\times3}$, and $b^{(2)} \in \mathbb{R}$

Multi Layered Perceptron (MLP)

Forward Propagation:



- $z_i^{(l)}$ is the pre-activation of unit i in layer l. This is the weighted summation of inputs connected to that unit.
- $a_i^{(l)}$ to denote the post activation output of unit i in layer l. For $l = 1$, we also use $a_i^{(1)} = x_i$ to denote i^{th} input.

$$a_1^{(2)} = f\left(W_{11}^{(1)}x_1 + W_{12}^{(1)}x_2 + W_{13}^{(1)}x_3 + b_1^{(1)}\right) = f\left(z_1^{(2)}\right)$$

$$a_2^{(2)} = f\left(W_{21}^{(1)}x_1 + W_{22}^{(1)}x_2 + W_{13}^{(1)}x_3 + b_2^{(1)}\right) = f\left(z_2^{(2)}\right)$$

$$a_3^{(2)} = f\left(W_{31}^{(1)}x_1 + W_{32}^{(1)}x_2 + W_{33}^{(1)}x_3 + b_3^{(1)}\right) = f\left(z_3^{(2)}\right)$$

Here $f(.)$ denotes the activation function.

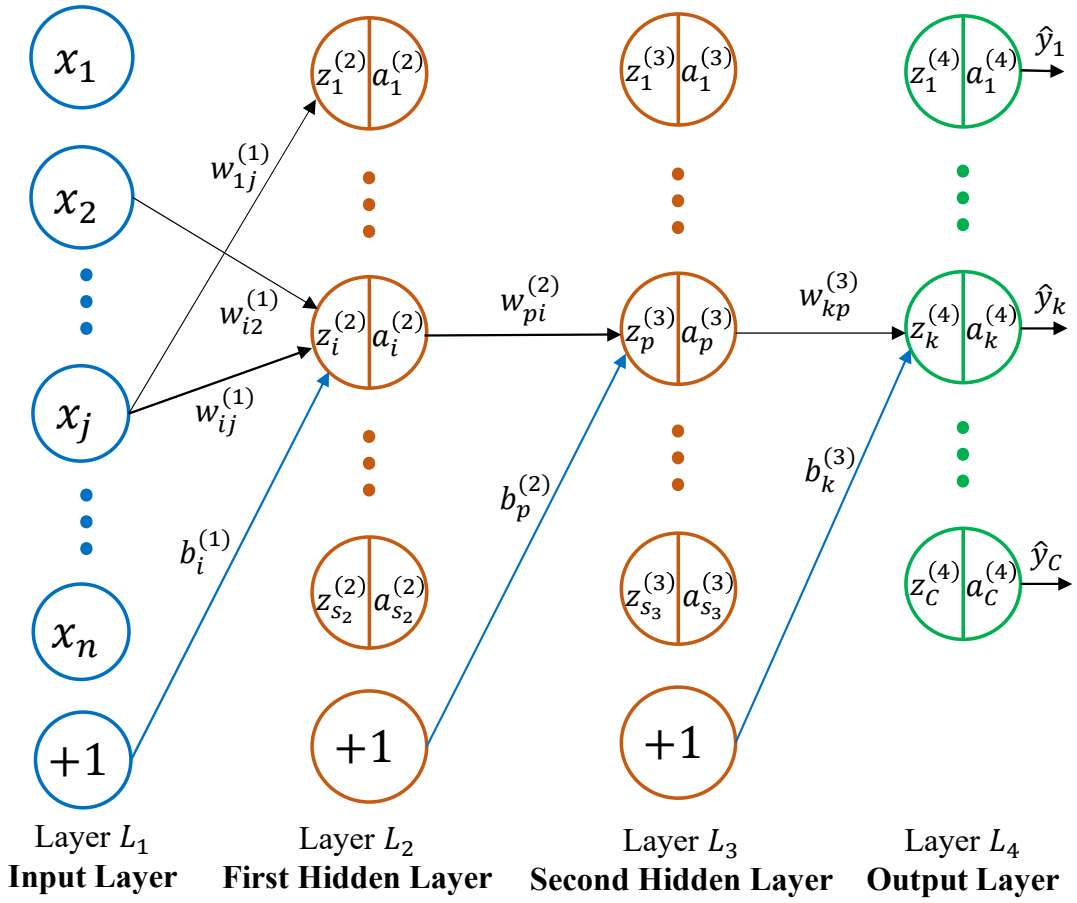
The final output $h_{w,b}(\boldsymbol{x}) = a_1^{(3)} = f\left(W_{11}^{(2)}\, a_1^{(2)} + W_{12}^{(2)}\, a_2^{(2)} + W_{13}^{(2)}\, a_3^{(2)} + b_1^{(2)}\right) = f\left(z_1^{(3)}\right)$

All the equations shown above can be written in simplified fashion in vector matrix notation:

$$\boldsymbol{a}^{(l+1)} = f\left(\boldsymbol{z}^{(l+1)}\right) = f\left(W^{(l)}\boldsymbol{a}^{(l)} + \boldsymbol{b}^{(l)}\right)$$

Multi Layered Perceptron (MLP)

- n_l denotes the number of layers in the network. s_l denotes the number of nodes in layer l.
- $W_{ij}^{(l)}$ denotes the parameter (or weight) associated with the connection between unit j in layer l, and unit i in layer $l+1$.
- $b_i^{(l)}$ is the bias associated with unit i in layer $l+1$.



$$\begin{bmatrix} z_1^{(2)} \\ \vdots \\ z_i^{(2)} \\ \vdots \\ z_{s_2}^{(2)} \end{bmatrix}_{s_2\times 1} = \begin{bmatrix} w_{11}^{(1)} & \dots & w_{1n}^{(1)} \\ \vdots & \vdots & \vdots \\ w_{i1}^{(1)} & \dots & w_{in}^{(1)} \\ \vdots & \dots & \dots \\ w_{s_2 1}^{(1)} & \dots & w_{s_2 n}^{(1)} \end{bmatrix}_{s_2\times n} \begin{bmatrix} x_1 \\ \vdots \\ x_j \\ \vdots \\ x_n \end{bmatrix}_{n\times 1} + \begin{bmatrix} b_1^{(1)} \\ \vdots \\ b_i^{(1)} \\ \vdots \\ b_{s_2}^{(1)} \end{bmatrix}_{s_2\times 1} \text{ and } \begin{bmatrix} a_1^{(2)} \\ \vdots \\ a_i^{(2)} \\ \vdots \\ a_{s_2}^{(2)} \end{bmatrix}_{s_2\times 1} = f\left(\begin{bmatrix} z_1^{(2)} \\ \vdots \\ z_i^{(2)} \\ \vdots \\ z_{s_2}^{(2)} \end{bmatrix}_{s_2\times 1} \right)$$

$$\boldsymbol{z}^{(2)} = W^{(1)}\boldsymbol{x} + \boldsymbol{b}^{(1)} \text{ and } \boldsymbol{a}^{(2)} = f\left(\boldsymbol{z}^{(2)}\right)$$

$$\begin{bmatrix} z_1^{(3)} \\ \vdots \\ z_p^{(3)} \\ \vdots \\ z_{s_3}^{(3)} \end{bmatrix}_{s_3\times 1} = \begin{bmatrix} w_{11}^{(2)} & \dots & w_{1s_2}^{(2)} \\ \vdots & \vdots & \vdots \\ w_{p1}^{(2)} & \dots & w_{ps_2}^{(2)} \\ \vdots & \dots & \dots \\ w_{s_3 1}^{(2)} & \dots & w_{s_3 s_2}^{(2)} \end{bmatrix}_{s_3\times s_2} \begin{bmatrix} a_1^{(2)} \\ \vdots \\ a_i^{(2)} \\ \vdots \\ a_{s_2}^{(2)} \end{bmatrix}_{s_2\times 1} + \begin{bmatrix} b_1^{(2)} \\ \vdots \\ b_p^{(2)} \\ \vdots \\ b_{s_3}^{(2)} \end{bmatrix}_{s_3\times 1} \text{ and } \begin{bmatrix} a_1^{(3)} \\ \vdots \\ a_p^{(3)} \\ \vdots \\ a_{s_3}^{(3)} \end{bmatrix}_{s_3\times 1} = f\left(\begin{bmatrix} z_1^{(3)} \\ \vdots \\ z_p^{(3)} \\ \vdots \\ z_{s_3}^{(3)} \end{bmatrix}_{s_3\times 1} \right)$$

$$\boldsymbol{z}^{(3)} = W^{(2)}\boldsymbol{a}^{(2)} + \boldsymbol{b}^{(2)} \text{ and } \boldsymbol{a}^{(3)} = f\left(\boldsymbol{z}^{(3)}\right)$$

$$\begin{bmatrix} z_1^{(4)} \\ \vdots \\ z_k^{(4)} \\ \vdots \\ z_C^{(4)} \end{bmatrix}_{C\times 1} = \begin{bmatrix} w_{11}^{(3)} & \dots & w_{1s_3}^{(3)} \\ \vdots & \vdots & \vdots \\ w_{k1}^{(3)} & \dots & w_{ks_3}^{(3)} \\ \vdots & \dots & \dots \\ w_{C1}^{(3)} & \dots & w_{Cs_3}^{(3)} \end{bmatrix}_{C\times s_3} \begin{bmatrix} a_1^{(3)} \\ \vdots \\ a_p^{(3)} \\ \vdots \\ a_{s_3}^{(3)} \end{bmatrix}_{s_3\times 1} + \begin{bmatrix} b_1^{(3)} \\ \vdots \\ b_k^{(3)} \\ \vdots \\ b_C^{(3)} \end{bmatrix}_{C\times 1} \text{ and } \begin{bmatrix} a_1^{(4)} \\ \vdots \\ a_k^{(4)} \\ \vdots \\ a_C^{(4)} \end{bmatrix}_{C\times 1} = f\left(\begin{bmatrix} z_1^{(4)} \\ \vdots \\ z_k^{(4)} \\ \vdots \\ z_C^{(4)} \end{bmatrix}_{C\times 1} \right)$$

$$\boldsymbol{z}^{(4)} = W^{(3)}\boldsymbol{a}^{(3)} + \boldsymbol{b}^{(3)} \text{ and } \boldsymbol{a}^{(4)} = f\left(\boldsymbol{z}^{(4)}\right)$$

The predicted output, $\hat{\boldsymbol{y}} = \boldsymbol{a}^{(4)}$

Multi Layered Perceptron (MLP)

Loss / Cost Function:

Given a training set of 'm' samples, we can define the overall cost functions in following way:

- **Mean Squared Error Loss:** Suitable for regression tasks.

$$J(W,b) = \frac{1}{m}\left[\sum_{i=1}^{m}\frac{1}{2}\left(h_{W,b}\left(x^{(i)}\right) - y^{(i)}\right)^2\right]$$

- **Binary Cross-entropy Loss:** Suitable for binary classification.

$$J(W,b) = -\frac{1}{m}\left[\sum_{i=1}^{m} y^{(i)}\log\left(h_{W,b}\left(x^{(i)}\right)\right) + \left(1-y^{(i)}\right)\log\left(1-h_{W,b}\left(x^{(i)}\right)\right)\right]$$

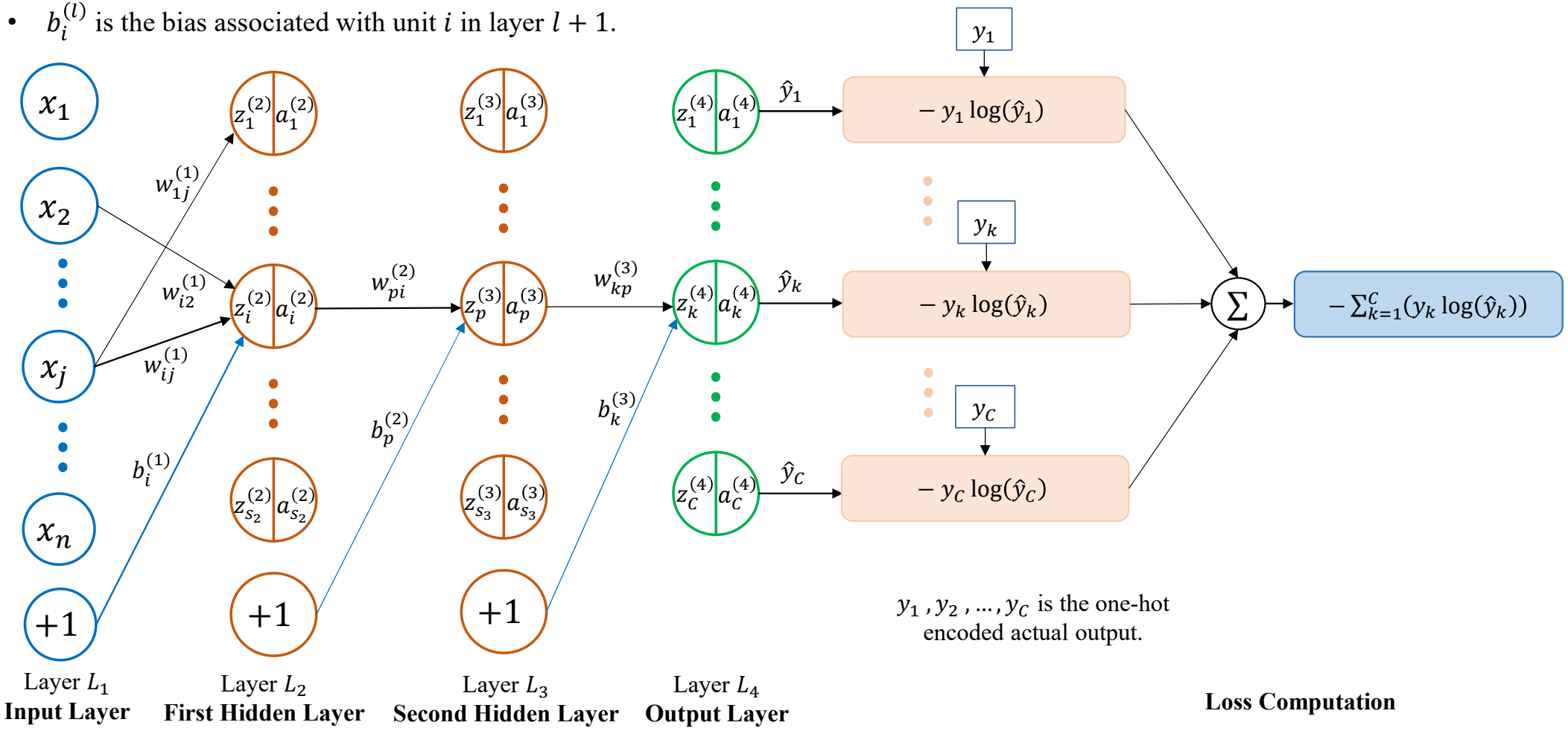
- **Categorical Cross-entropy Loss:** Suitable for multi-class classification (number of classes = C).

$$J(W,b) = -\frac{1}{m}\left[\sum_{i=1}^{m}\sum_{k=1}^{C} y_k^{(i)}\log\left(h_{W,b}\left(x^{(i)}\right)_k\right)\right]$$

$h_{W,b}\left(x^{(i)}\right)$ is the model's prediction for input $x^{(i)}$. So, $h_{W,b}\left(x^{(i)}\right) = \hat{y}^{(i)}$ and the actual label is $y^{(i)}$.

Multi Layered Perceptron (MLP)

- n_l denotes the number of layers in the network. s_l denotes the number of nodes in layer l.
- $W_{ij}^{(l)}$ denotes the parameter (or weight) associated with the connection between unit j in layer l, and unit i in layer $l+1$.
- $b_i^{(l)}$ is the bias associated with unit i in layer $l+1$.



$y_1, y_2, \ldots, y_C$ is the one-hot encoded actual output.

Multi Layered Perceptron (MLP)

Back Propagation Learning Algorithm:

The weights and biases in the neural network are updated using Backpropagation Algorithm, where each weights are updated using Gradient Descent update rule (or some variant of it). For that we have to calculate the partial derivatives of the loss function with respect to the weights and biases.

User has to specify:

- Number of **layers**: n_l , or people often specify number of hidden layers which is $n_l - 2$.
- Number of **nodes** in each hidden layer: $s_l \ for \ l = 2, 3, \dots, n_l - 1$
- Number of **epochs** for which the network to be trained. An epoch = one complete pass of the entire dataset through the neural network. During an epoch, the model sees all training samples once and update weights depending on the **batch size**.
- The **loss function** to be used by the neural network.
- The optimization technique (also called **optimizer**) to be used to optimize the loss. The most basic optimizer is gradient descent.
- The **batch size**: The number of training samples processed before the model updates its weights.

Gradient Descent Optimization Technique

General idea of Gradient Descent:

We want to minimize a loss function $J(W, b)$ with respect to model parameters $W \; and \; b$.

Update rule: (showing only for w for simplification)

$$w \leftarrow w - \eta \nabla_w J \; ; \; \eta \Rightarrow learning \; rate \; , \nabla J \Rightarrow gradient \; of \; loss \; J \; w.r.t. \; parameter \; w.$$

(a) Batch Gradient Descent:

- In each epoch the error in the final layer is average of errors due to all the training samples in the dataset.
- Update the parameters / weights using backpropagation after all the training samples are fed to the network.
- This is stable and achieves smooth convergence. However, this is very slow for large dataset.

(b) Stochastic Gradient Descent (SGD):

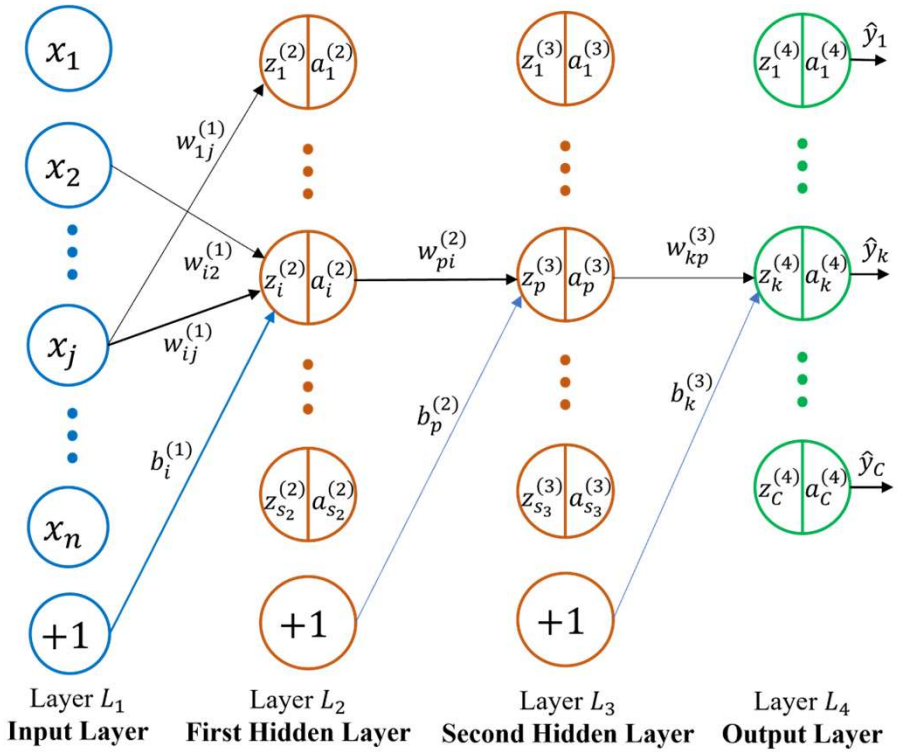
- Updates the weight after each training sample.
- Faster convergence but noisy updates due to randomness.
- It can escape local minima due to randomness.

(c) Mini-batch Gradient Descent:

- Uses small random subsets (batches) of data to compute gradients. Compromise between batch and SGD.
- For example: if dataset size = 5000 and batch size = 100, then one epoch = 50 batches. Each batch produces one weight update, so 50 iterations per epoch.
- It is most widely used in practice. The batch size depends on dataset size, GPU/RAM memory, and training stability. Most commonly used batch sizes are 32, 64, 128, 256 etc. [i.e. between 32 to 256]

Back Propagation Learning Algorithm

To better understand the flow of computations, let us draw a computation graph

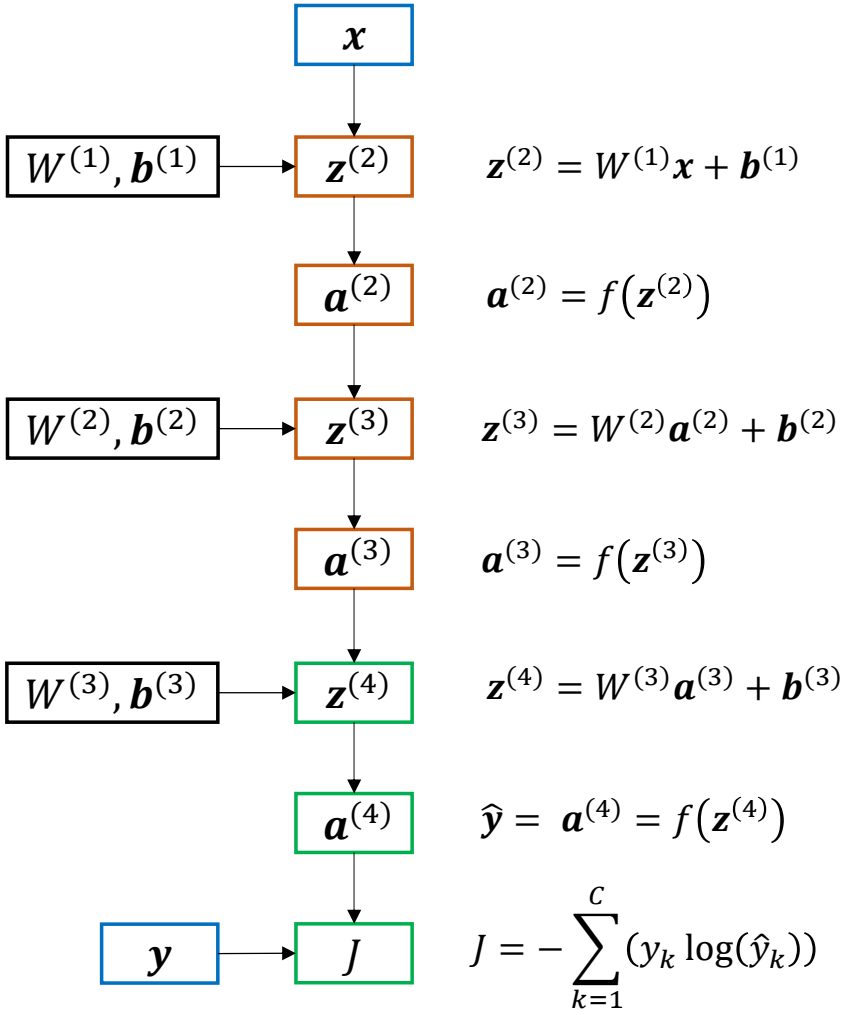


$$\boldsymbol{z}^{(2)} = W^{(1)}\boldsymbol{x} + \boldsymbol{b}^{(1)} \text{ and } \boldsymbol{a}^{(2)} = f\left(\boldsymbol{z}^{(2)}\right)$$

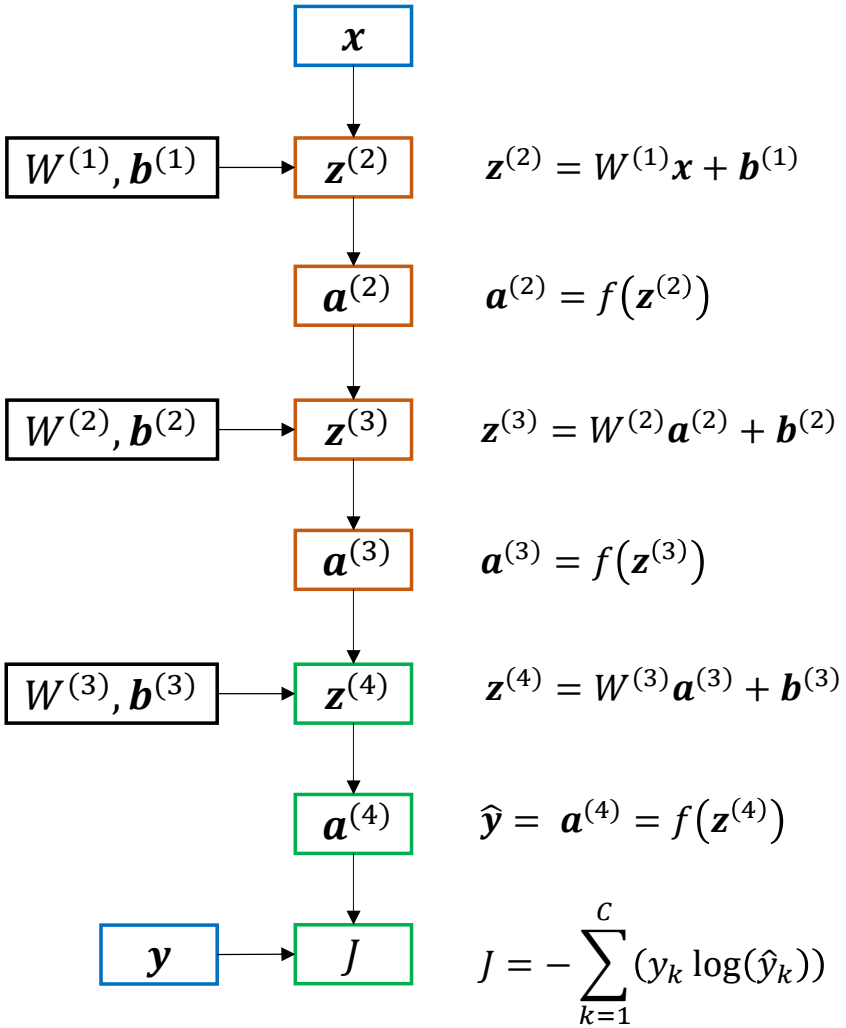
$$\boldsymbol{z}^{(3)} = W^{(2)}\boldsymbol{a}^{(2)} + \boldsymbol{b}^{(2)} \text{ and } \boldsymbol{a}^{(3)} = f\left(\boldsymbol{z}^{(3)}\right)$$

$$\boldsymbol{z}^{(4)} = W^{(3)}\boldsymbol{a}^{(3)} + \boldsymbol{b}^{(3)} \text{ and } \boldsymbol{a}^{(4)} = f\left(\boldsymbol{z}^{(4)}\right)$$

The predicted output, $\hat{\boldsymbol{y}} = \boldsymbol{a}^{(4)}$



Back Propagation Learning Algorithm



Cost function J is a function of $a^{(4)}$, $a^{(4)}$ is in turn a function of $z^{(4)}$. $z^{(4)}$ is function of $W^{(3)}, \boldsymbol{b}^{(3)}$ *and* $\boldsymbol{a}^{(3)}$.

So, to calculate the partial derivatives of the cost function J w.r.t. $W^{(3)}, \boldsymbol{b}^{(3)}$ we will have to use the **chain rule**.

$$\frac{\partial J}{\partial W^{(3)}} = \frac{\partial J}{\partial \boldsymbol{a}^{(4)}} \cdot \frac{\partial \boldsymbol{a}^{(4)}}{\partial \boldsymbol{z}^{(4)}} \cdot \frac{\partial \boldsymbol{z}^{(4)}}{\partial W^{(3)}} \quad \text{and} \quad \frac{\partial J}{\partial \boldsymbol{b}^{(3)}} = \frac{\partial J}{\partial \boldsymbol{a}^{(4)}} \cdot \frac{\partial \boldsymbol{a}^{(4)}}{\partial \boldsymbol{z}^{(4)}} \cdot \frac{\partial \boldsymbol{z}^{(4)}}{\partial \boldsymbol{b}^{(3)}}$$

Similarly, to calculate the partial derivatives of the cost function J w.r.t. $W^{(2)}, \boldsymbol{b}^{(2)}$ we will use the following formula:

$$\frac{\partial J}{\partial W^{(2)}} = \frac{\partial J}{\partial \boldsymbol{a}^{(4)}} \cdot \frac{\partial \boldsymbol{a}^{(4)}}{\partial \boldsymbol{z}^{(4)}} \cdot \frac{\partial \boldsymbol{z}^{(4)}}{\partial \boldsymbol{a}^{(3)}} \cdot \frac{\partial \boldsymbol{a}^{(3)}}{\partial \boldsymbol{z}^{(3)}} \cdot \frac{\partial \boldsymbol{z}^{(3)}}{\partial W^{(2)}}$$

$$\text{and} \quad \frac{\partial J}{\partial \boldsymbol{b}^{(2)}} = \frac{\partial J}{\partial \boldsymbol{a}^{(4)}} \cdot \frac{\partial \boldsymbol{a}^{(4)}}{\partial \boldsymbol{z}^{(4)}} \cdot \frac{\partial \boldsymbol{z}^{(4)}}{\partial \boldsymbol{a}^{(3)}} \cdot \frac{\partial \boldsymbol{a}^{(3)}}{\partial \boldsymbol{z}^{(3)}} \cdot \frac{\partial \boldsymbol{z}^{(3)}}{\partial \boldsymbol{b}^{(2)}}$$

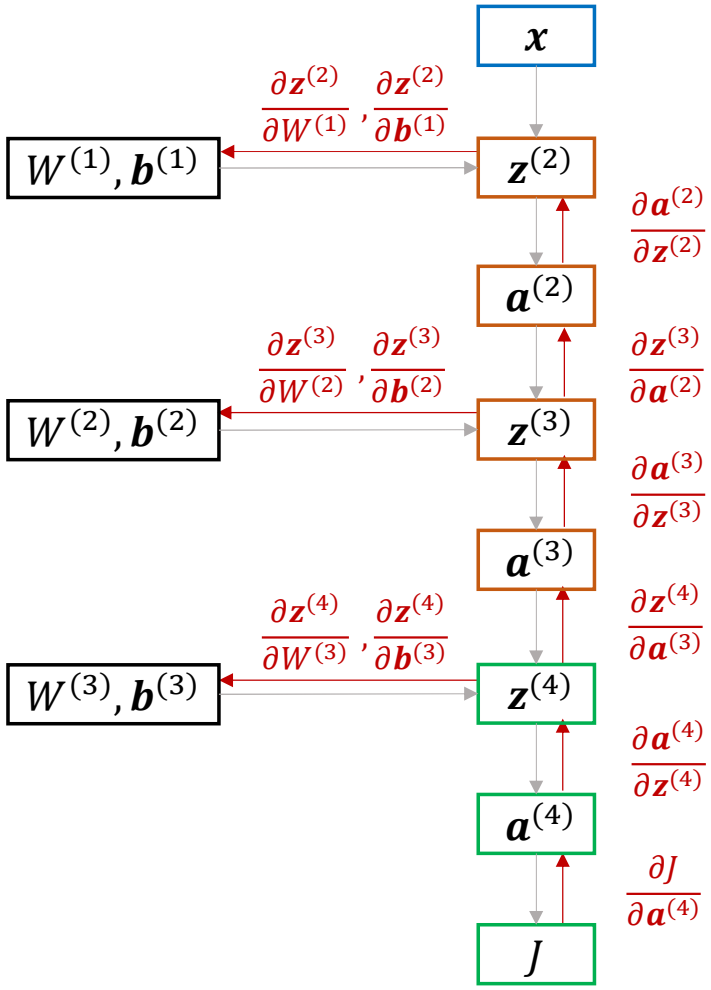
Likewise, for calculating the partial derivatives of the cost function J w.r.t. $W^{(1)}, \boldsymbol{b}^{(1)}$ we will use:

$$\frac{\partial J}{\partial W^{(1)}} = \frac{\partial J}{\partial \boldsymbol{a}^{(4)}} \cdot \frac{\partial \boldsymbol{a}^{(4)}}{\partial \boldsymbol{z}^{(4)}} \cdot \frac{\partial \boldsymbol{z}^{(4)}}{\partial \boldsymbol{a}^{(3)}} \cdot \frac{\partial \boldsymbol{a}^{(3)}}{\partial \boldsymbol{z}^{(3)}} \cdot \frac{\partial \boldsymbol{z}^{(3)}}{\partial \boldsymbol{a}^{(2)}} \cdot \frac{\partial \boldsymbol{a}^{(2)}}{\partial \boldsymbol{z}^{(2)}} \cdot \frac{\partial \boldsymbol{z}^{(2)}}{\partial W^{(1)}}$$

$$\text{and} \quad \frac{\partial J}{\partial \boldsymbol{b}^{(1)}} = \frac{\partial J}{\partial \boldsymbol{a}^{(4)}} \cdot \frac{\partial \boldsymbol{a}^{(4)}}{\partial \boldsymbol{z}^{(4)}} \cdot \frac{\partial \boldsymbol{z}^{(4)}}{\partial \boldsymbol{a}^{(3)}} \cdot \frac{\partial \boldsymbol{a}^{(3)}}{\partial \boldsymbol{z}^{(3)}} \cdot \frac{\partial \boldsymbol{z}^{(3)}}{\partial \boldsymbol{a}^{(2)}} \cdot \frac{\partial \boldsymbol{a}^{(2)}}{\partial \boldsymbol{z}^{(2)}} \cdot \frac{\partial \boldsymbol{z}^{(2)}}{\partial b^{(1)}}$$

Back Propagation Learning Algorithm

This can be better understood from the following figure:



$$z^{(2)} = W^{(1)}x + b^{(1)}$$

$$a^{(2)} = f(z^{(2)})$$

$$z^{(3)} = W^{(2)}a^{(2)} + b^{(2)}$$

$$a^{(3)} = f(z^{(3)})$$

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

$$\hat{y} = a^{(4)} = f(z^{(4)})$$

$$J = -\sum_{k=1}^{C} (y_k \log(\hat{y}_k))$$

Forward propagation

Back propagation

- It's called **backpropagation** because the error (loss gradient) is **propagated backward** from the output layer to the input layer.
- This backward flow of gradients updates weights layer by layer using the chain rule.

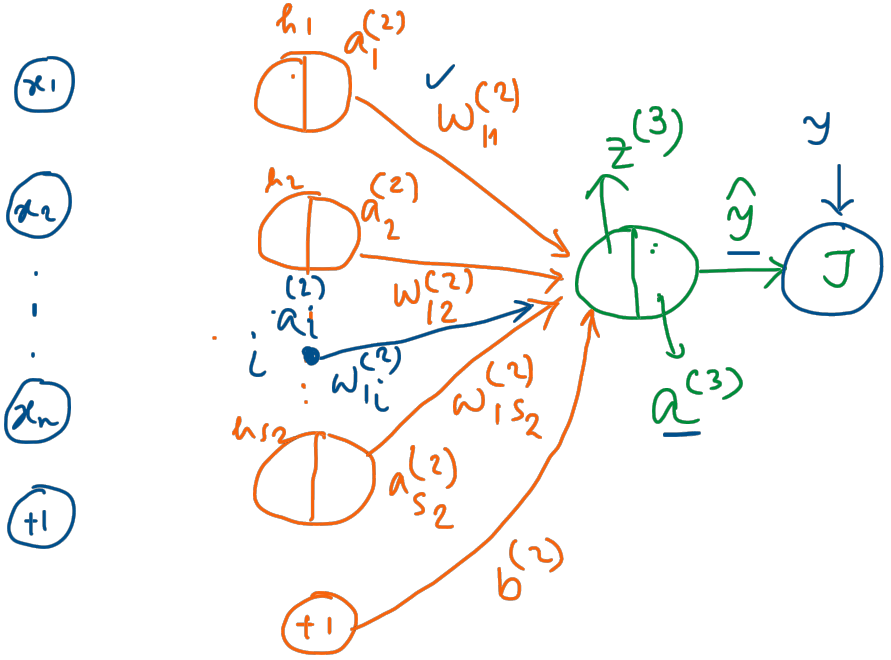
Back Propagation Learning Algorithm

Input to the algorithm: Labelled dataset: $\langle \boldsymbol{X}, y \rangle$; **Output of the algorithm:** Trained weights and biases

1. For each epoch : (Number of epochs should be specified by user)
2. Perform a feedforward pass, computing the activations for layers L_2, L_3, and so on up to the output layer L_{n_l}.
3. For each output unit i in layer n_l (the final/output layer), set
$$\delta_i^{(n_l)} = \frac{\partial J}{\partial a_i^{(n_l)}} . f'\left(z_i^{(n_l)}\right)$$
4. For $l = (n_l - 1), (n_l - 2), (n_l - 3), \ldots, 2, 1$

   For each node i in layer l, set
$$\delta_i^{(l)} = \left(\sum_{j=1}^{s_{l+1}} W_{ji}^{(l)} \delta_j^{(l+1)} \right) . f'\left(z_i^{(l)}\right)$$
5. Compute the desired partial derivatives which are given as:
$$\frac{\partial}{\partial W_{ij}^{(l)}} J(W,b) = a_j^{(l)} . \delta_i^{(l+1)} \quad and \quad \frac{\partial}{\partial b_i^{(l)}} J(W,b) = \delta_i^{(l+1)}$$
6. Finally update the parameters:
$$W_{ij}^{(l)} \leftarrow W_{ij}^{(l)} - \eta \frac{\partial}{\partial W_{ij}^{(l)}} J(W,b) \quad and \quad b_i^{(l)} \leftarrow b_i^{(l)} - \eta \frac{\partial}{\partial b_i^{(l)}} J(W,b); \ \eta \text{ is the learning rate}$$

Back - Propagation :



$$z^{(3)} = w_{11}^{(2)} a_1^{(2)} + \cdots + w_{1s_2}^{(2)} a_{s_2}^{(2)} + b^{(2)} = \sum_{i=1}^{s_2} w_{1i}^{(2)} a_i^{(2)} + b^{(2)}$$

$$a^{(3)} = \hat{y} = f(z^{(3)}) \qquad [f(\cdot) \rightarrow \text{activation function}]$$

$$J = -\left(y \log \hat{y} + (1-y) \log(1-\hat{y})\right)$$

$$\frac{\partial J}{\partial w_{1i}^{(2)}} = \frac{\partial J}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial z^{(3)}} \cdot \frac{\partial (z^{(3)})}{\partial w_{1i}^{(2)}}$$

$$= -\left(\frac{y}{\hat{y}} - \frac{1-y}{1-\hat{y}}\right) \cdot \frac{\partial f(z^{(3)})}{\partial z^{(3)}} \cdot \boxed{\frac{\partial z^{(3)}}{\partial w_{1i}^{(2)}}} \rightarrow a_i^{(2)}$$

$\sigma(x)$

$\sigma(1-\sigma)$

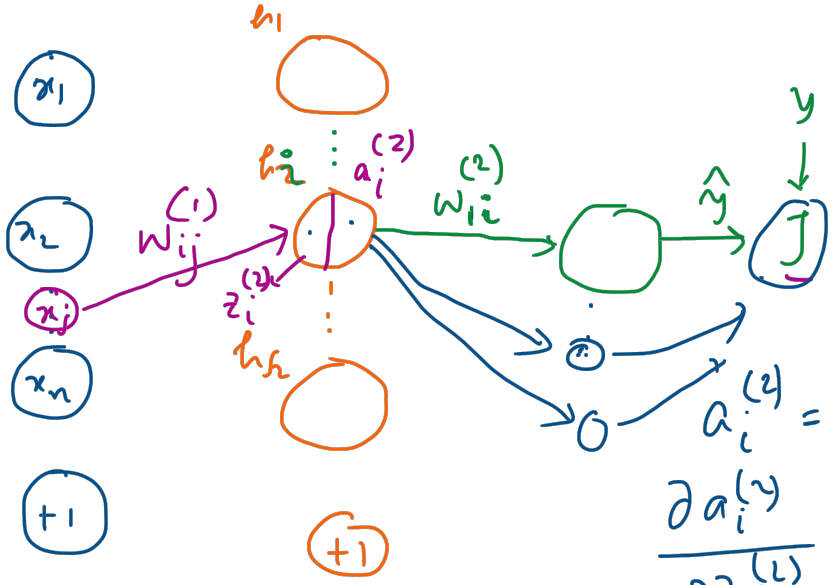
$$= -\frac{y(1-\hat{y}) - \hat{y}(1-y)}{\hat{y}(1-\hat{y})} \cdot f'(z^{(3)}) \cdot a_i^{(2)}$$

$$\frac{\partial J}{\partial b^{(2)}} = -\delta_i^{(3)}$$

(Step-5)

$$= -\frac{(y-\hat{y})}{\hat{y}(1-\hat{y})} \cdot f'(z^{(3)}) \cdot a_i^{(2)} = \boxed{-(y-a^{(3)}) \cdot f'(z^{(3)})} \; a_i^{(2)}$$

$$\frac{\partial J}{\partial w_{1i}^{(2)}} = -\delta_i^{(3)} \cdot a_i^{(2)} \rightarrow \text{(Step-5)}$$



$$\frac{\partial J}{\partial w_{ij}^{(1)}} = \boxed{\frac{\partial J}{\partial \hat{y}} \cdot \frac{\partial a^{(3)}}{\partial z^{(3)}}} \cdot \frac{\partial z^{(3)}}{\partial a_i^{(2)}} \cdot \frac{\partial a_i^{(2)}}{\partial z_i^{(2)}} \cdot \frac{\partial z_i^{(2)}}{\partial w_{ij}^{(1)}}$$

$$\delta_i^{(3)} \quad w_{1i}^{(2)} \quad f'(z_i^{(2)}) \cdot a_j^{(1)}$$

$$a_i^{(2)} = f(z_i^{(2)})$$

$$\frac{\partial a_i^{(2)}}{\partial z_i^{(2)}} = f'(z_i^{(2)})$$

$$\sum \delta_i^{(3)} \; w_{1i}^{(2)} \cdot f'(z_i^{(3)}) \cdot a_j^{(1)}$$

Step-4

$$z^{(3)} = w_{11}^{(2)} a_1^{(2)} + w_{12}^{(2)} a_2^{(2)} + \cdots + w_{1i}^{(2)} a_i^{(2)} + \cdots + b^{(2)}$$

$$\frac{\partial z^{(3)}}{\partial a_i^{(2)}} = w_{1i}^{(2)}$$

$$z_i^{(2)} = w_{11}^{(1)} x_1 + w_{12}^{(1)} x_2 + \cdots + w_{1j}^{(1)} \cdot x_j + \cdots + b_i^{(1)}$$

$$\frac{\partial z_i^{(2)}}{\partial w_{ij}^{(1)}} = x_j = a_j^{(1)}$$

Back-Prop has a sequential dependency layerwise, but within a layer the weight updates can be performed parallelly.

Thank You