6CS012 – Artificial Intelligence and Machine Learning.

Lecture – 04

Artificial Neural Network.

Multi – Layer Neural Network.

Siman Giri {Module Leader – 6CS012}

Learning Objective,

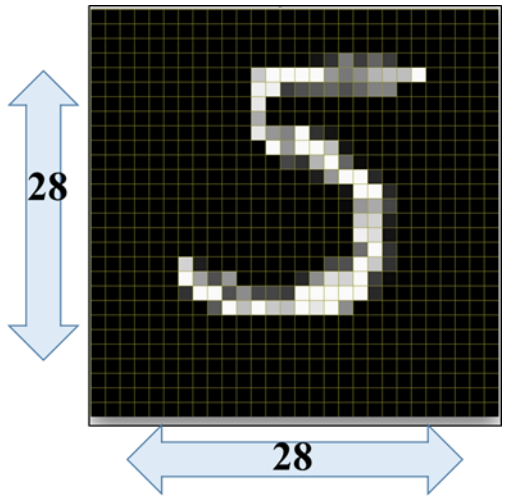
- We will first review some of the limitations about Perceptron discussed last week.
  - By the end of this week, we will try to overcome the limitations set by Perceptron and Perceptron Learning Algorithm.
  - To overcome the limitation we will discuss, Deeper Neural Network also called Multi Layer Perceptron or Multi layer Neural Network.
- Thus, By the end of this week we expect you to:
  - Understand the basic of Deep neural network
  - Explain the architecture of Deep neural network
  - Train the Deep Neural Network with Gradient Descent for Multi class classification problem.

On the Quest to Solve the Challenges.

{Recap the Challenge … }

Recap the Challenges:

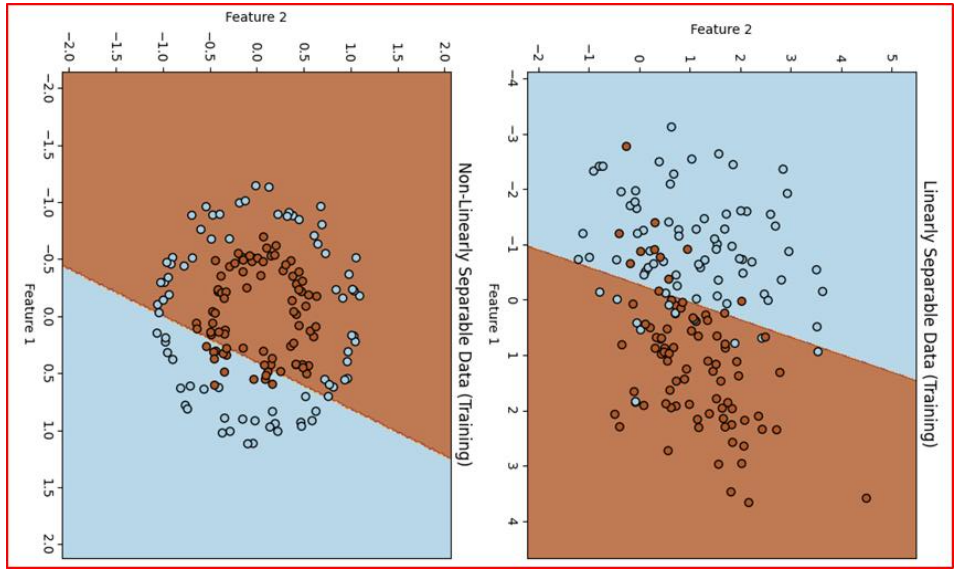
- **Challenge 1:**
  - **Extraction of Features:**
    - When we extracted only pixel values we got the csv file with 784 columns i.e. very high dimensions , and our dataset was only of the size of 28×28.
    - Imagine for larger images, how big our dataset may be.



- *"We want to automate the process of feature extraction."*

Recap the Challenges:

- **Challenge 2:**
  - **Non-Linear Decision Boundary:**
    - **Logistic Regression{Softmax Regression}** was better on separating a data in classes/label those were **linearly separable**.
    - **What for data where classes/label are not linearly separable?**
    - ***"Did The Perceptron Solved the Challenges …"***



1. Perceptron and Their Limitations.

1.1 The Perceptron.

- The perceptron is a fundamental component of many modern neural network architectures and often referred to as neurons.
- The perceptron are often a single hidden unit which is connected to some input x.

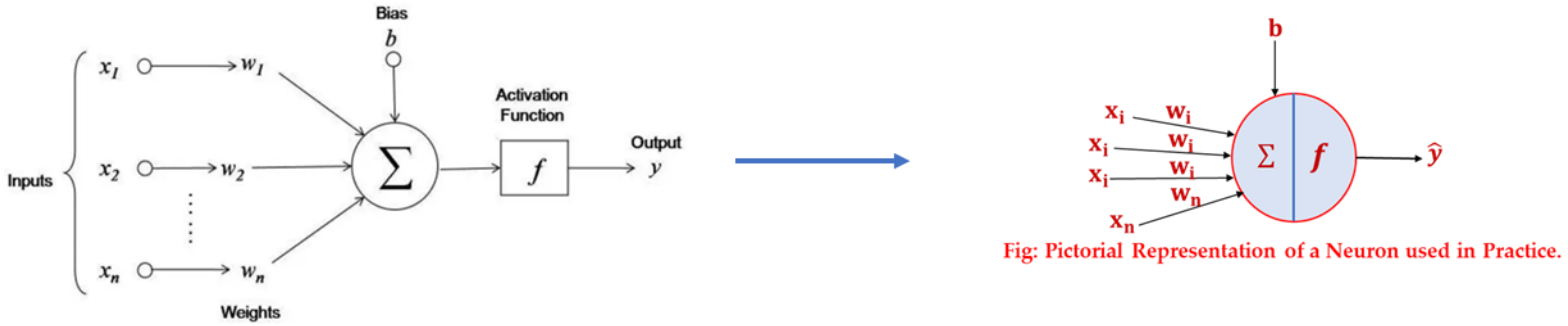
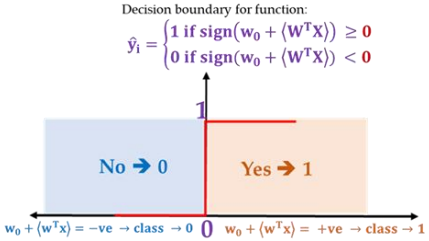


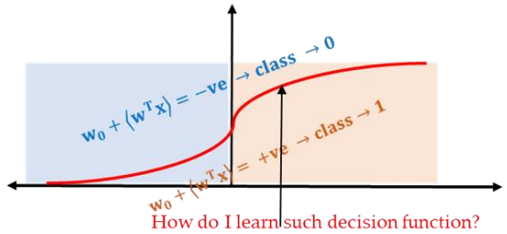
Fig: Pictorial Representation of a Neuron used in Practice.

- The perceptron computes the dot product of a weight vector w with input x, optionally adds a bias term then passes through **activation** function.
- The activation function determines whether the perceptron/neuron gets activated or not based on sum of weighted inputs.
  - The original Perceptron uses unit step function as an activation function, which is not ideal for most of the modern problem.

1.2 Why not a Unit Step Function?

- **Unit step function defined as:**
  - $\mathbf{f(z) = \begin{cases} 1 \text{ if } z \geq 0 \\ 0 \text{ if } z < 0 \end{cases}; \{z = w_0 + \langle w^T x \rangle\}}$
- **Does not capture the subtleties of the data:**
  - The **step function** may not capture the subtleties of the data;
    - it creates a sharp boundary at 0,
      - meaning that all positive sums are classified as class 1
      - and all negative sums are classified as class 0.
    - However, there may be cases where net negative values also belong to class 1.

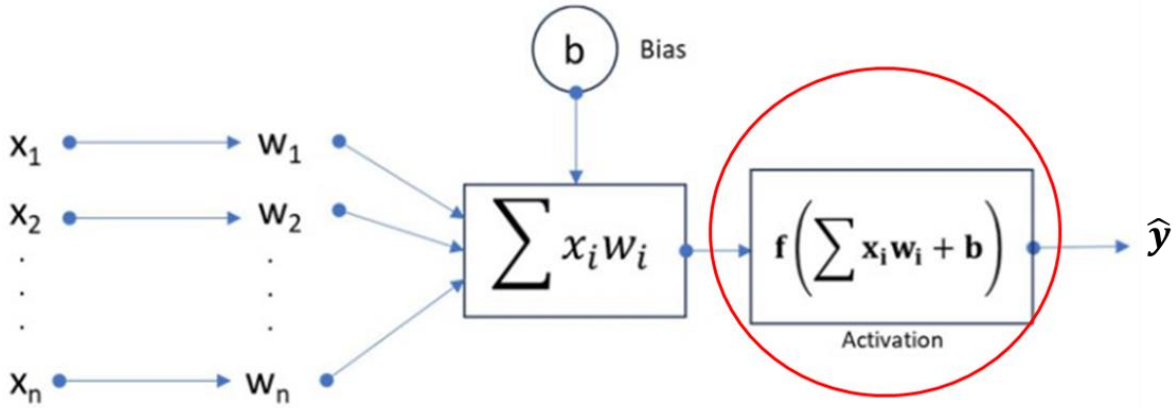




- **Binary Outputs Only:**
  - It only outputs 0 or 1, restricting the network from handling nuanced decision boundaries.
- **No Learning Capability:**
  - Small changes in input do not affect the output, limiting the model's ability to learn complex patterns.
- **Non-Differentiability:**
  - It lacks a derivative, making it incompatible with gradient-based optimization (e.g., backpropagation in neural networks).

1.3 The activation function.

- In General, Activation Functions are group of function that introduces **non-linearity** to the **output** of neurons.
- The Good activation function must have following key properties:
  - **Non – Linearity**: Allows the network to learn complex patterns and decision boundaries.
  - **Differentiability**: Must be differentiable to enable gradient based optimizations i.e. gradient descent.
  - **Computationally Efficiency**: Should be fast to compute for real – time applications.
  - **Gradient Behavior**: Should avoid vanishing or exploding gradients to stabilize deeper network.{*we will discuss this further after defining deep networks*.}



1.4 Sigmoid Neurons.

- **Sigmoid neurons** are a type of artificial neuron that uses the
  - **sigmoid "activation" function** to model the output of the neuron.
  - The sigmoid function is a smooth, S-shaped curve that maps any
    - real-valued input to a value between 0 and 1,
      - making it especially useful for problems that involve probabilities or binary classification.
- The **sigmoid activation function** is defined as:
  - $\sigma(z) = \frac{1}{1+e^{-z}}$
    - here: $z = w_0 + w_i x_i$ → the weighted sum of the inputs to the neuron.
- *There are more kind of activation function in use, we will discuss as required.*

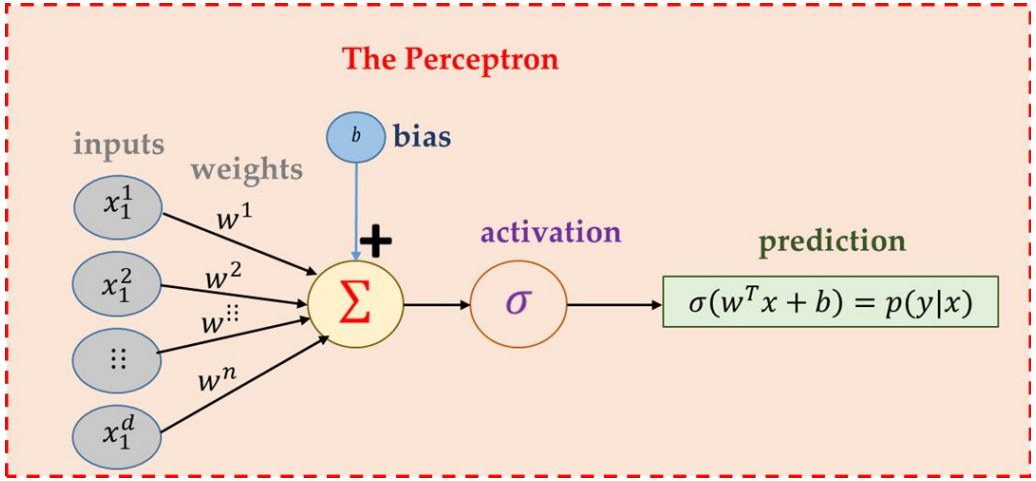


Fig: A single unit of Standard Perceptron with sigmoid.

1.5 Limitations of The Perceptron.

- Our Observations – 1:
  - The perceptrons are binary classifier and works well when data are linearly separable.

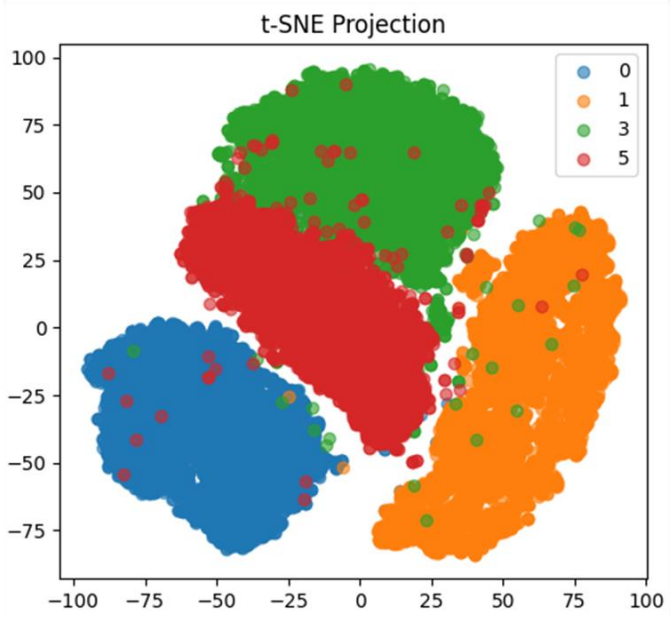
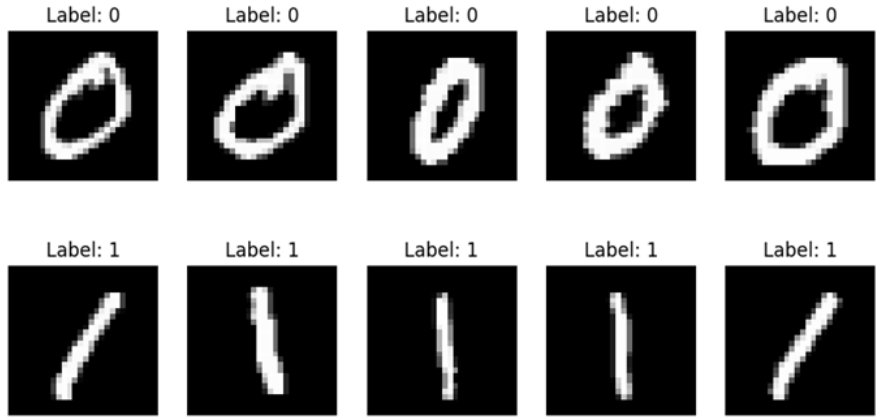


Fig: **0 vs. 1** are linearly separable but **3 vs. 5** are not

Performing 0 Vs. 1 Classification.



All images were correctly classified!

1.5.1 Limitations of The Perceptron.

- Our Observations – 2:
  - The perceptrons are binary classifier and works well when data are linearly separable.

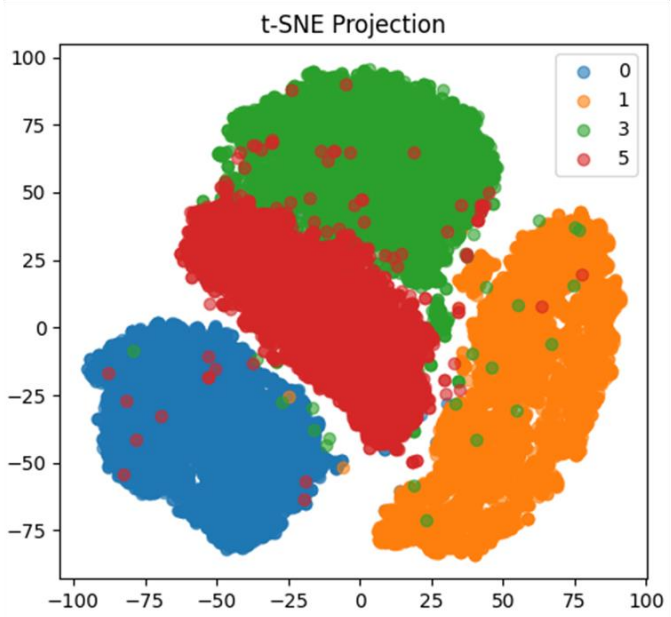
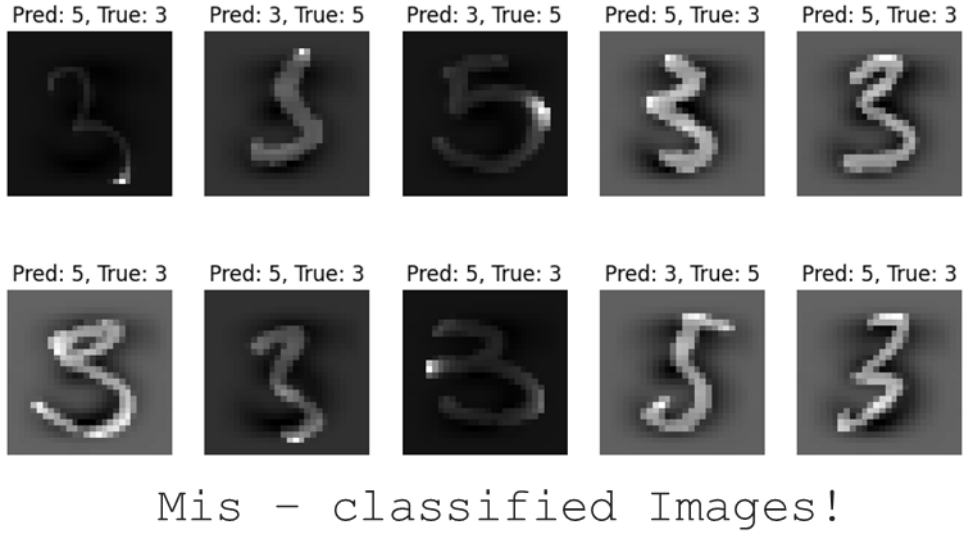


Fig: **0 vs. 1** are linearly separable but **3 vs. 5** are not

Performing 3 Vs. 5 Classification.



1.5.2 Limitations of The Perceptron.

- Our Observation – 3 :
  - In our Tutorial we also observed that:

x_1	x_2	x_1 xor x_2
0	0	0
0	1	1
1	0	1
1	1	0

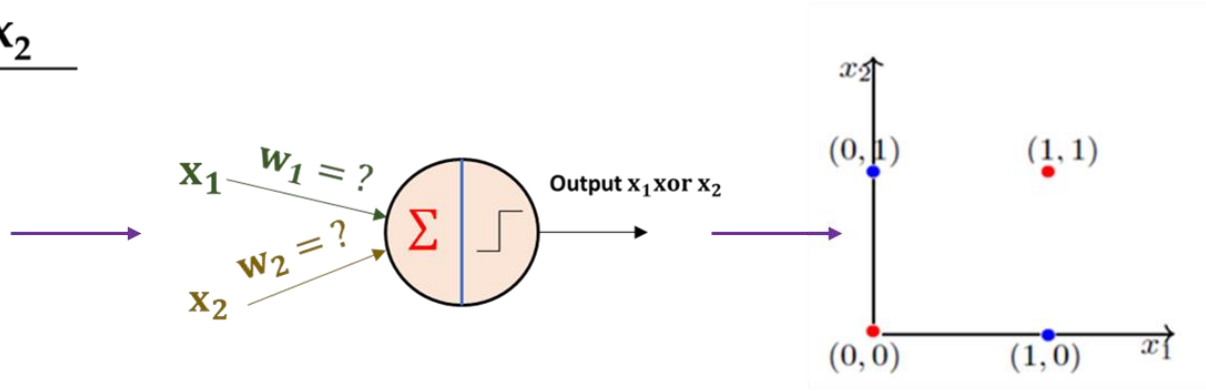
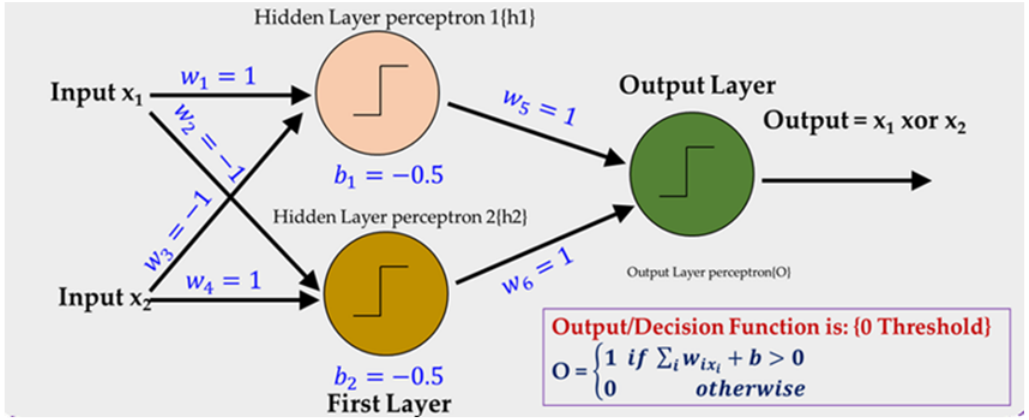


Fig: Not such value of w′s could be found, Thus No Decision Boundary Could be determined.

1.5.3 Limitations of The Perceptron.

- Our Observation – 4 :
  - In our Tutorial, we also observed that if we arrange multiple perceptrons/neurons in layers, it can be used to learn non-linearly separable functions, such as XOR,
    - This idea is backed by "*Theory of Universal Approximation*."

x_1	x_2	x_1 xor x_2
0	0	0
0	1	1
1	0	1
1	1	0



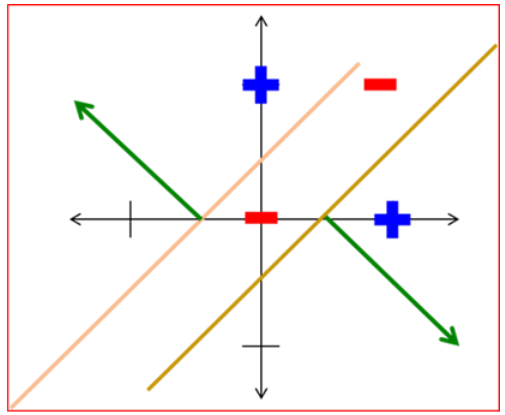


Fig: Arranging Perceptron in two layers allowed us to learn the decision boundary for non linearly separable dataset.

Q: Can Perceptron Learning Algorithm be used to learn such weights.

1.6 Theory of Universal Approximation.

- Idea: The **Universal Approximation Theory** as demonstrated by Cybenko (1989) and Hornik (1991) states that:
  - By arranging neurons in layers and using a nonlinear activation functions,
    - neural networks can **represent any nonlinear function**
      - making them powerful tools for function approximation, pattern recognition, and deep learning applications.
- **How many neurons?**
  - The result by Erol Gelenbe et al. (1999) formally proves that any **Boolean function** can be exactly implemented using a **one-hidden-layer perceptron network** with
    - **2^n {n → number of inputs} neurons in the hidden layer.**

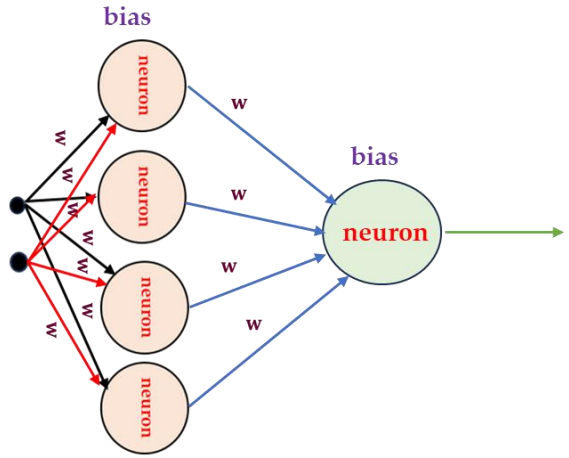
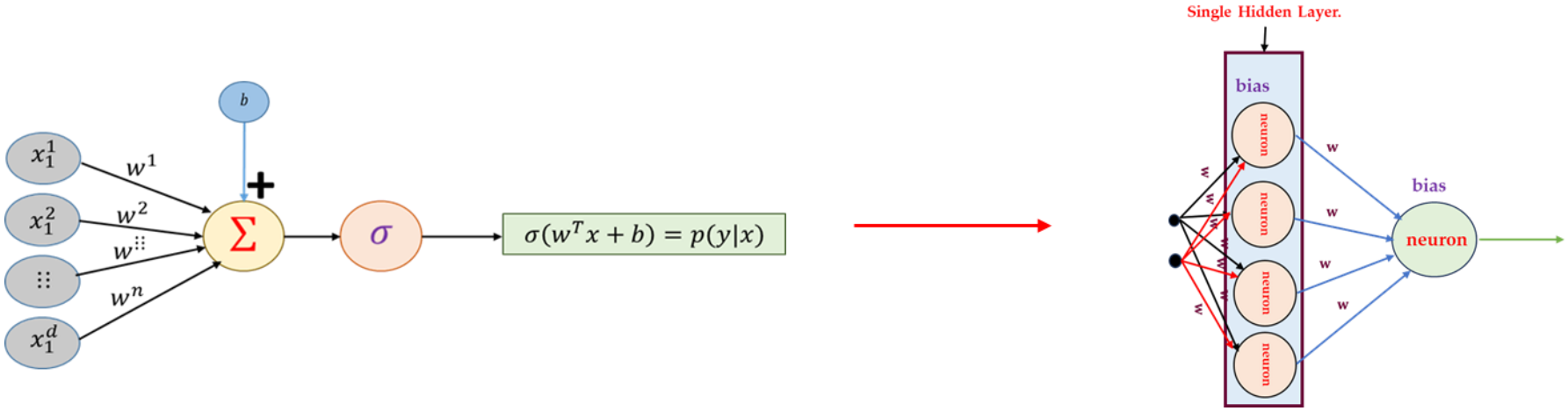


Fig: Claim Using this structure: I can learn the suitable value of weights which can learn a decision boundary of any non linear boolean function of 2 inputs: {2^n ⇒ 4 neurons in hidden layer}

"Warnings: The Interpretation of The UAT is limited to this class and module, and thus has a broader and more in general mathematical interpretation."

1.6.1 Challenges of Single Layer Neural Network.



- As we noted, for Boolean functions, **single {input} layer** with **2^n neurons and one output layer** can represent **any function**.{ These architecture are also referred as **Single Layer Neural Network**.}
- However, for real world functions, achieving a good approximation with a single hidden layer often requires an excessively large number of neurons.
  - Example: Suppose we need to approximate a complex function in image recognition i.e. dog vs cat classification.
    - A **single layer** would require **millions of neurons**, making **computing inefficient**.
    - Thus, in practice **we prefer a deeper multi – layer neural network aka deeper network**.

1.7.1 Architecture of One Hidden Layer Neural Network.

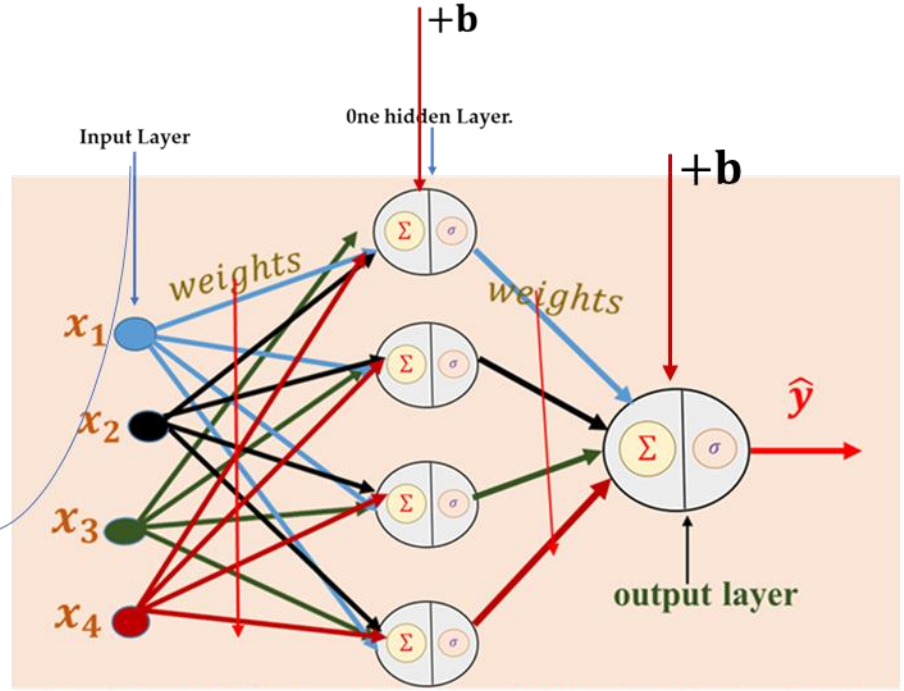


Fig: A Two {1 - hidden} layer Fully Connected Network aka Shallow Network.

Understanding the Architecture.

- **Input Layer – 1:**
  The input layer is Just a Place Holder for input values, no computation occurs here, Thus is not counted towards Final Layer.
  - $\mathbf{x} = [\mathbf{x_1}, \mathbf{x_2}, \mathbf{x_3}, \mathbf{x_4}]$

1.7.2 Architecture of One Hidden Layer Neural Network.

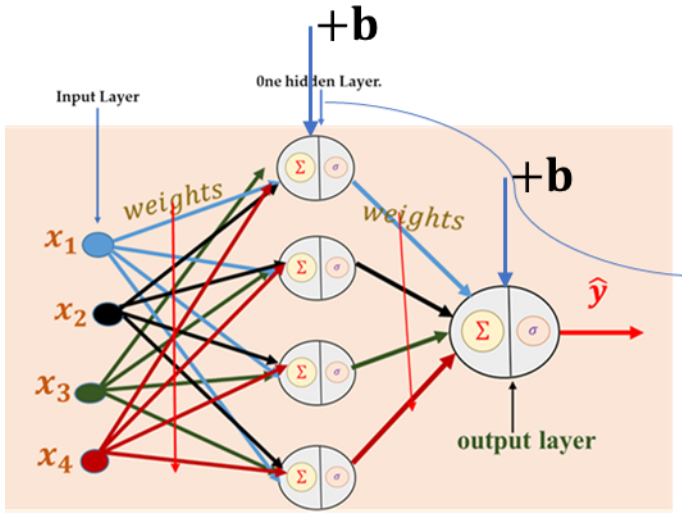


Fig: A Two {1 - hidden} layer Fully Connected Network aka Shallow Network.

- The computation starts at the first hidden layer and results are propagated forward to next layer. Thus, also called Feed Forward Network.
- The computation on Hidden layer is:
  - $\mathbf{z^1 = \langle W^1 x \rangle + b^1}$
    $\mathbf{a^1 = \left(f^1(z^1)\right)}$
- Here:
  - $*^1 \rightarrow$ **First Layer**
  - $\mathbf{w^1}$ **→ Weight vector at layer 1, there are 4 neurons at layer 1 , Thus there will be 4 weight vectors i. e. for neuron 1 to neuron 4:**
  - $\mathbf{w^1 = [w_1, w_2, w_3, w_4] \in}$ **for 4 neurons, each** $\mathbf{w_1 = \begin{bmatrix} w_1 \\ w_1 \\ w_1 \\ w_1 \end{bmatrix}}$; $\mathbf{w_2 = \begin{bmatrix} w_2 \\ w_2 \\ w_2 \\ w_2 \end{bmatrix}}$; $\mathbf{w_3 = \begin{bmatrix} w_3 \\ w_3 \\ w_3 \\ w_3 \end{bmatrix}}$; $\mathbf{w_4 = \begin{bmatrix} w_4 \\ w_4 \\ w_4 \\ w_4 \end{bmatrix}}$
  - $\mathbf{z^1}$ **→ weighted sum at first layer → vector →** $\mathbf{\begin{bmatrix} z_1 = (\langle w_1 \cdot x \rangle + b) \\ z_2 = (\langle w_2 \cdot x \rangle + b) \\ z_3 = (\langle w_3 \cdot x \rangle + b) \\ z_4 = (\langle w_4 \cdot x \rangle + b) \end{bmatrix}^1}$ **Example:** $\mathbf{\langle w_1 \cdot x \rangle \rightarrow \begin{bmatrix} w_1 \\ w_1 \\ w_1 \\ w_1 \end{bmatrix} \cdot [x_1, x_2, x_3, x_4]}$
  - $\mathbf{a^1}$ **→ Computed output after applying activiation function** $\mathbf{f^1}$ **at layer 1. The activation fucntion we use →** $\mathbf{f^1 \rightarrow}$ **sigmoid(**σ**).**
  - $\mathbf{a^1 = \sigma\left(\begin{bmatrix} z_1 \\ z_2 \\ z_3 \\ z_4 \end{bmatrix}^1\right) = \begin{bmatrix} a_1 = P(y = 1|z_1) \\ a_2 = P(y = 1|z_2) \\ a_3 = P(y = 1|z_3) \\ a_4 = P(y = 1|z_4) \end{bmatrix}}$

Hidden Layer.

1.7.3 Architecture of One Hidden Layer Neural Network.

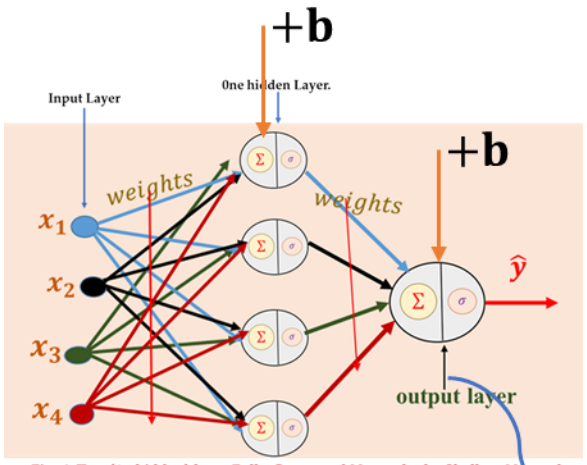


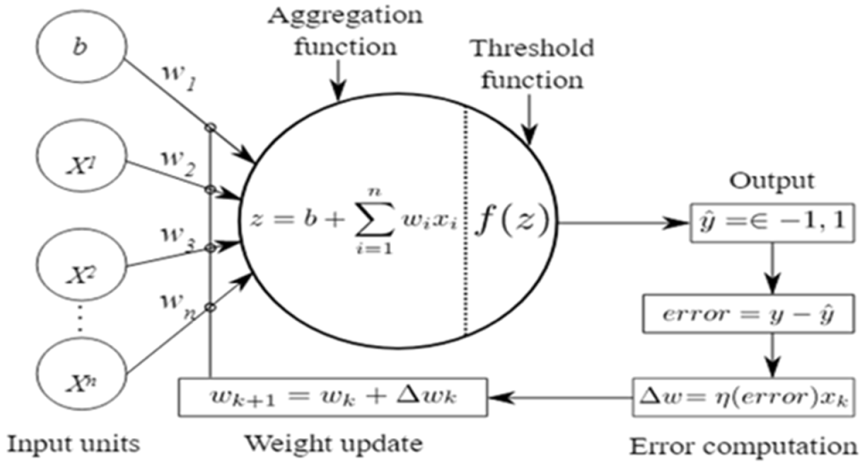
Fig: A Two (1 - hidden) layer Fully Connected Network aka Shallow Network.

- Neurons in the **output layer** may have a **different activation function** compared to those in **the hidden layers**, depending on the type of classification task being performed.
- For example,
  - in **binary classification**, we continue to use the **sigmoid activation function**,
  - whereas for **multi-class classification**, we use the **softmax activation function**.
- Additionally, **the number of neurons in the output layer** depends on whether:
- the task is binary or multi-class classification (**more details on this in the upcoming slide**).
- For this demonstration, we will use a **single neuron with a sigmoid activation function**.
- Computation on Output Layer:
  - $\mathbf{z^2 = w^2 a^1 + b^2}$
  - $\mathbf{a^2 = f^2(z^2)\{ f \rightarrow \sigma \text{ activation function}\}}$
- Here:
  - $*^2 \rightarrow$ Second layer.
  - $w^2 \rightarrow$ weight vector at layer 2. In this layer there are only one weight vector with 4 elements each coming from 4 neurons from first hidden layer. $\mathbf{w^2 = [w_1, w_2, w_3, w_4]^2}$
  - $z^2 \rightarrow$ weighted sum at second layer, which is given by:
    - $$\mathbf{z^2 = [\langle w^2 \cdot a^1 \rangle + b] = \left[\left([w_1 \quad w_2 \quad w_3 \quad w_4] \cdot \begin{bmatrix} a_1^1 \\ a_2^1 \\ a_3^1 \\ a_4^1 \end{bmatrix} \right) + b \right] = [w_1 a_1^1 + w_2 a_2^1 + w_3 a_3^1 + w_4 a_4^1 + b]}$$
  - $a^2 \rightarrow$ Computed output after applying activiation function f^1 at layer 1. The activation fucntion we use $\rightarrow f^2 \rightarrow$ sigmoid(σ).
    - $\mathbf{a^2 = \sigma[z^2] = [p(y = 1|z^2)]}$
- The final decision is made :
  - $$\mathbf{if: \begin{cases} a^2 \geq 0.5 \rightarrow \text{class 1} \\ \text{otherwise} \rightarrow \text{class 0} \end{cases}}$$

Output Layer.

1.8 Limitations of Perceptron Learning Algorithm.

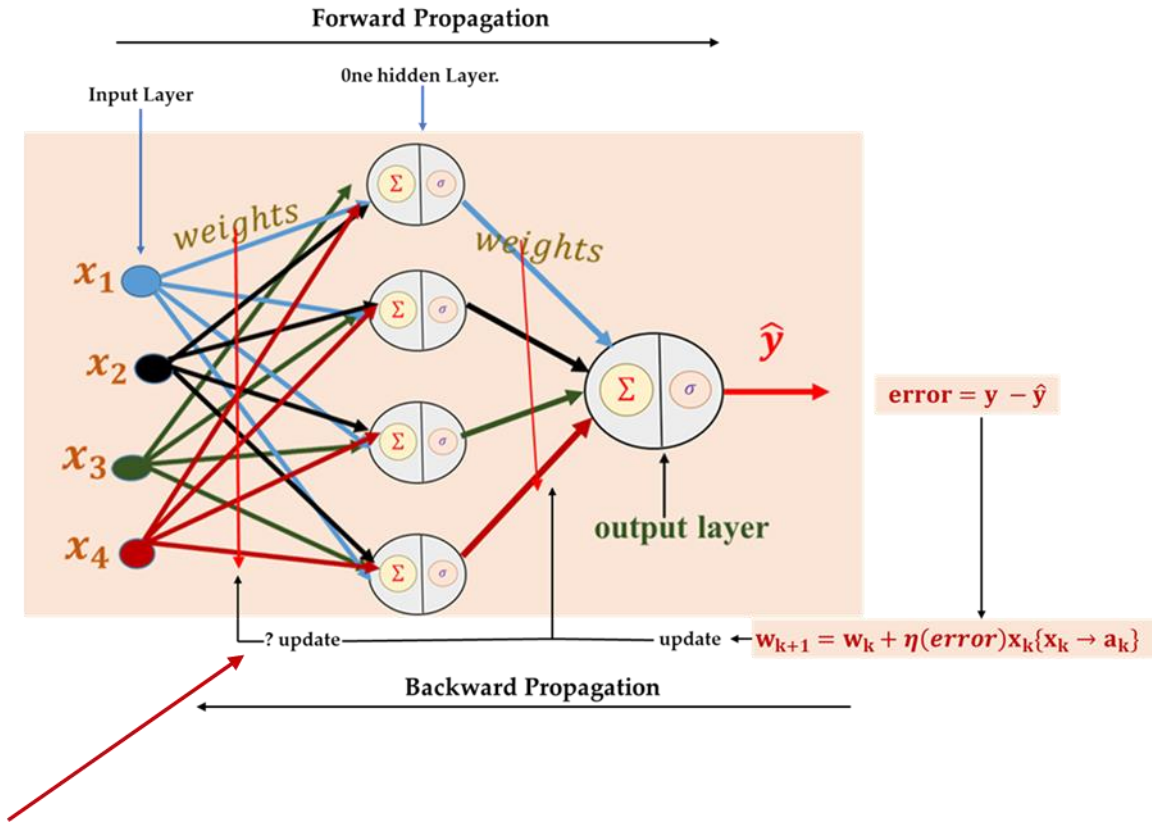
- The **Perceptron Learning Algorithm** is designed for single-layer perceptrons or one neurons
  - and is applicable only to problems that are **linearly separable**.
- The algorithm works by iteratively adjusting the weights based on the prediction error.
  - Specifically, it updates the weights when the model's output is incorrect, moving the weights in the direction that would reduce the error.



- However, the perceptron learning algorithm has a very important limitations if tried to use for multi – layer perceptron network:

1.8.1 Limitations of a Perceptron Learning Algorithm.

- **No Error Propagation to Middle (Hidden) Layers:**
  - In a single-layer perceptron, the algorithm updates the weights of the output layer based on the error.
    - However, it does not propagate this error to middle hidden layers (if any), as it does not have the mechanisms needed for **backpropagation**.
  - This is a key limitation if we need to build multi-layer neural networks, which can solve non-linear classification problems by learning complex decision boundaries.
    - Thus, we require an updated learning algorithm that can not only backpropagate the adjusted weights, via the middle or hidden layers, as per the error computation, to the **very first layer** but also perform **feedforward computations.**
    - **Can we use Gradient Descent?**



2. Towards Multi – Layer {Deep} Neural Network.

{ Shallow vs. Deep Neural Network}

2.1 From Single layer to Shallow Network.

Single Layer Neural Network.

- Are the networks with single hidden layer with 2^n neurons {**n → number of neurons which is equal to number of inputs.** }.
  - Theoretically, it **can represent any function**:
    - however, in practice, achieving a good approximation often requires an excessively large number of neurons, making it computationally inefficient.
  - Thus, Idea of Shallow Network.

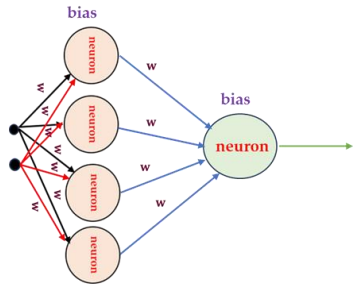
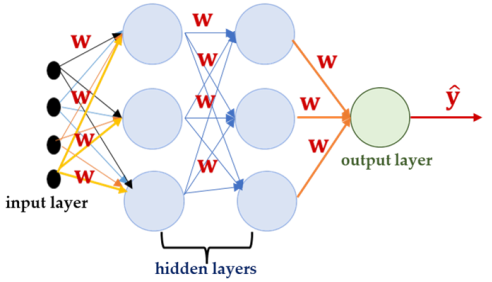


Fig: Claim Using this structure: I can learn the suitable value of weights which can learn a decision boundary of any non linear boolean function of 2 inputs: {2^n ⇒ 4 neurons in hidden layer}

Shallow Neural Network.

- The concept of a Shallow Network:
  - The idea behind a **shallow network** is to avoid placing an exponentially large number of neurons in a single hidden layer.
    - Instead, we distribute computation across **multiple hidden layers** with **fewer neurons** per layer.
  - Thus, a shallow neural network typically consists of **one or a few hidden layers**, reducing the need for 2^n neurons in a single layer while still capturing complex patterns.



every edges has different associated weights value.

Fig: A shallow network with 2 hidden layers.

2.2 From Shallow to Deep Neural Network.

- **Limitations of Shallow Neural Network:**
  - **Universality:**
    - A shallow MLP with one hidden layer can **approximate** any continuous function given enough neurons. However, the required number of neurons can **be impractically large**.
  - **Expressivity:**
    - A small number of neurons may not capture complex, non – linear decision boundaries needed for real – world classification problems.
  - **Learnability:**
    - There's no guarantee that the chosen learning method **will find the optimal solution**, as the network may get stuck in **local minima or fail to converge effectively.**
      - {If we pick gradient decent, more on this in upcoming slide.}

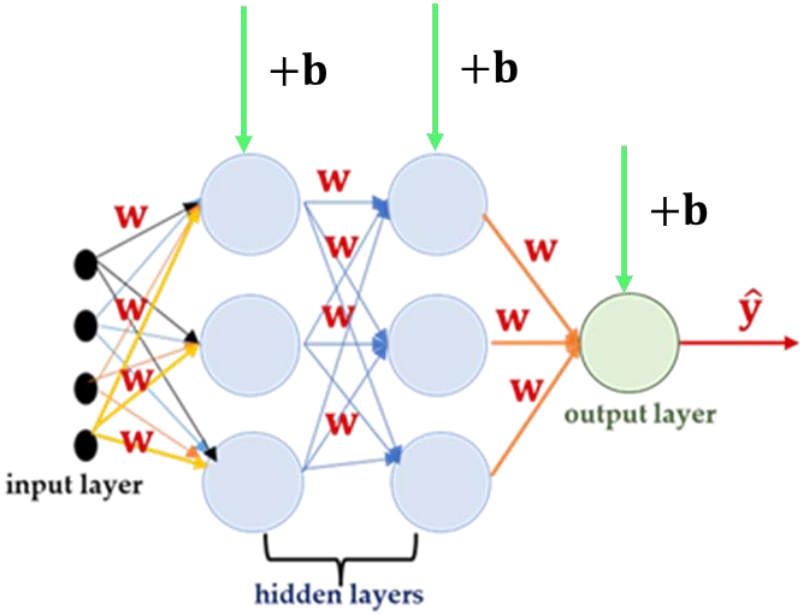


Fig: A shallow network with 2 hidden layers.

From this point forward I will not show +b in my architectural drawing, please imagine there is a bias.

2.3 Deep Neural Network.

- Deep Neural Network: Any network with more than 2 hidden layer.
- Why More layer?
- Two motivations for creating deeper network (as per Deep learning by Goodfellow et. Al)

1. Statistical:

- Deep networks are ***inherently compositional***, making them well-suited for **representing hierarchical structures**, where **simple patterns are recursively combined** to form more complex ones.
- This aligns with the structure of many real-world datasets.

2. Computational and Expressiveness:

- Under ***certain conditions***, deep architectures can be more expressive than shallow ones, enabling them to learn **more complex patterns** with the **same total parameters.**

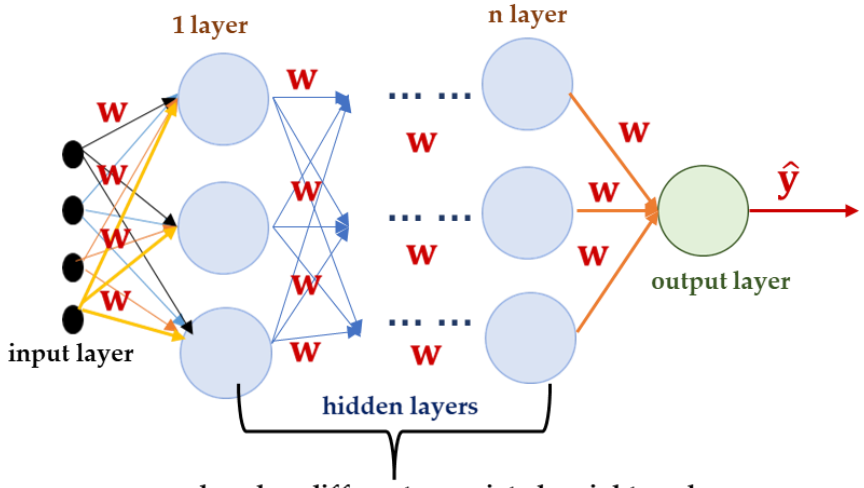
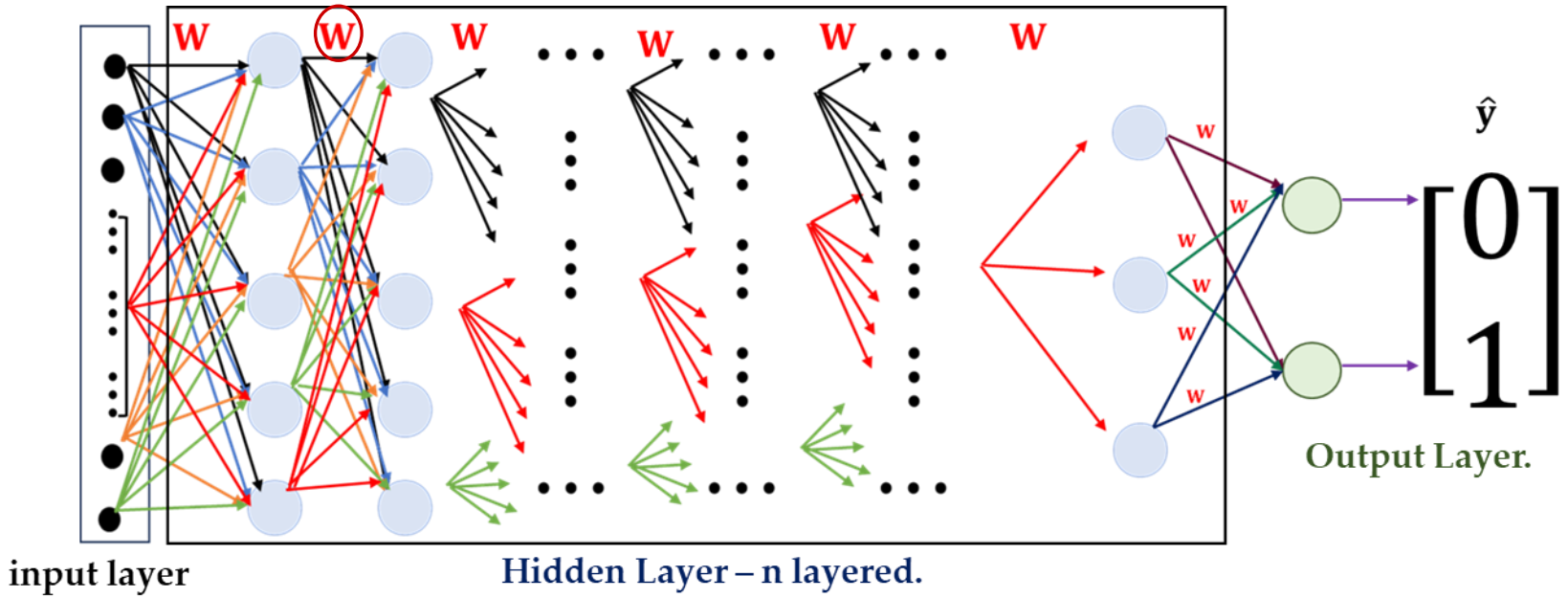


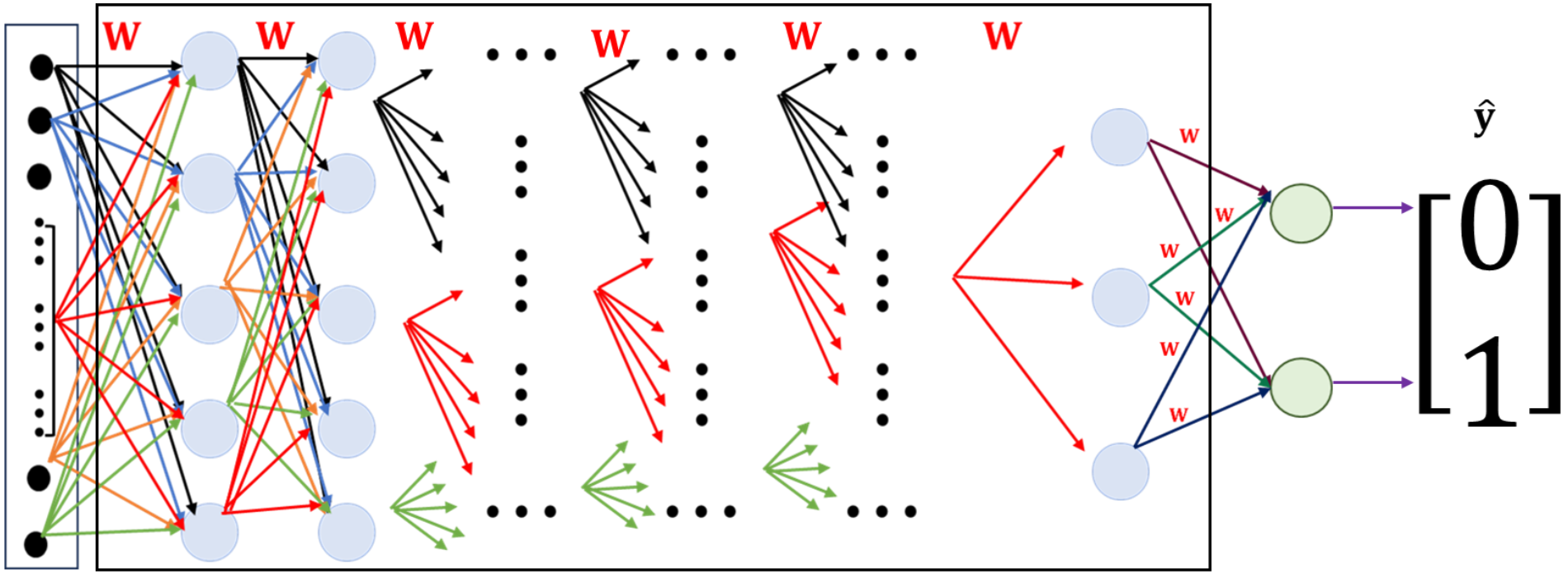
Fig: A deep network with n hidden layers.

2.4 Understanding the Architecture of DNN.

- **Architecture :**
  - Multi – layer Fully Connected Neural Network, with unique weight value at every edges.
    - It is called Fully Connected Neural Network; as in this design all the neurons are connected with each other.
      - Each individual neuron performs two task:
        - **Compute a weighted sum and**
        - **Applies an activation function.**



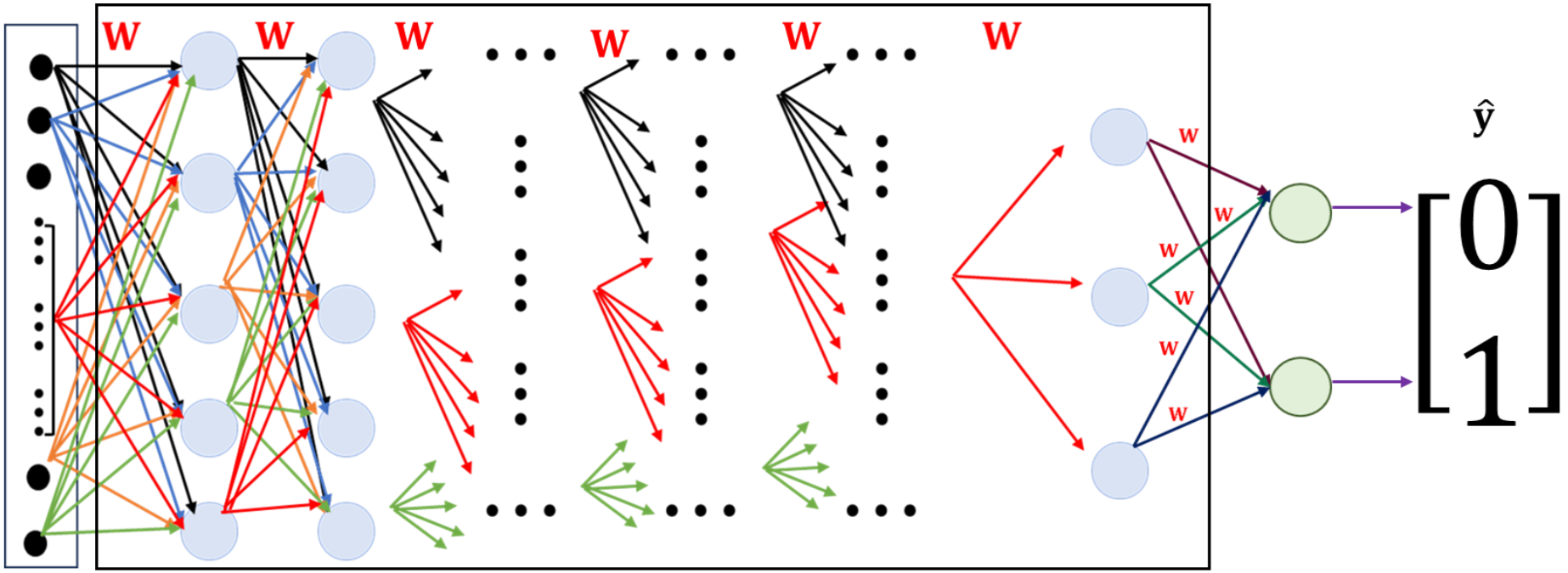
2.4.1 Understanding the Architecture: Number of Neurons.



Input Layer:

- The **number of neurons** in the **input layer** is determined by the **size of the input image**.
  - For example, in the case of an image **with shape** $(28 \times 28 \times 1)$ (like in the MNIST dataset),
    - the input layer will have **784 neurons** $\{(28 \times 28 \times 1) = 784\}$.
  - This layer serves as a placeholder, as no computations occur here.
  - Therefore,
    - *the number of neurons in the input layer is equal to the size of the input.*

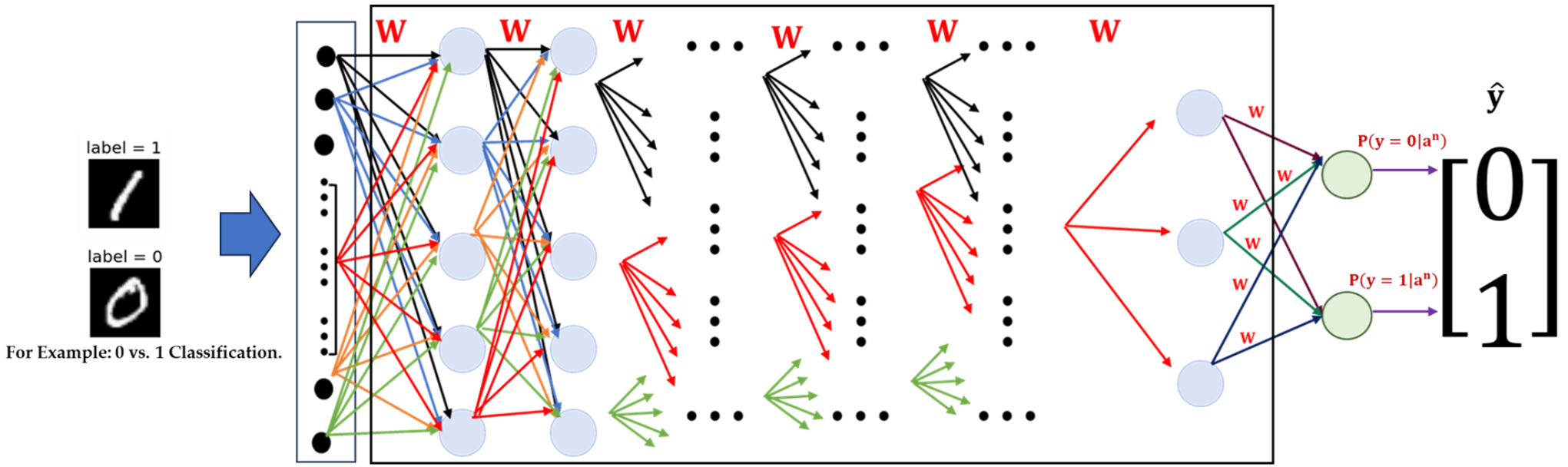
2.4.2 Understanding the Architecture: Number of Neurons.



Hidden Layer:

- **Number of Hidden Layer $\{L^k\}$ and Neurons per layer $\{k_n: k^{th}$ layer n neurons$\}$:**
  - The **number of hidden layers and neurons per layer** are **hyperparameters** and must be determined by the **designing engineer** based on **specific task.**
- **Cautions:**
  - Selection of L^k and k_n impacts the network's capacity and computational efficiency.
    - **Increasing neurons** improves capacity but may lead to overfitting.
    - **Decreasing neurons** may cause underfitting.
- **General Heuristics on selection:**
  - Start simple i.e. start with reasonable size e.g. **powers of 2 → 32, 64, 128, → increase the complexity as per requirement**

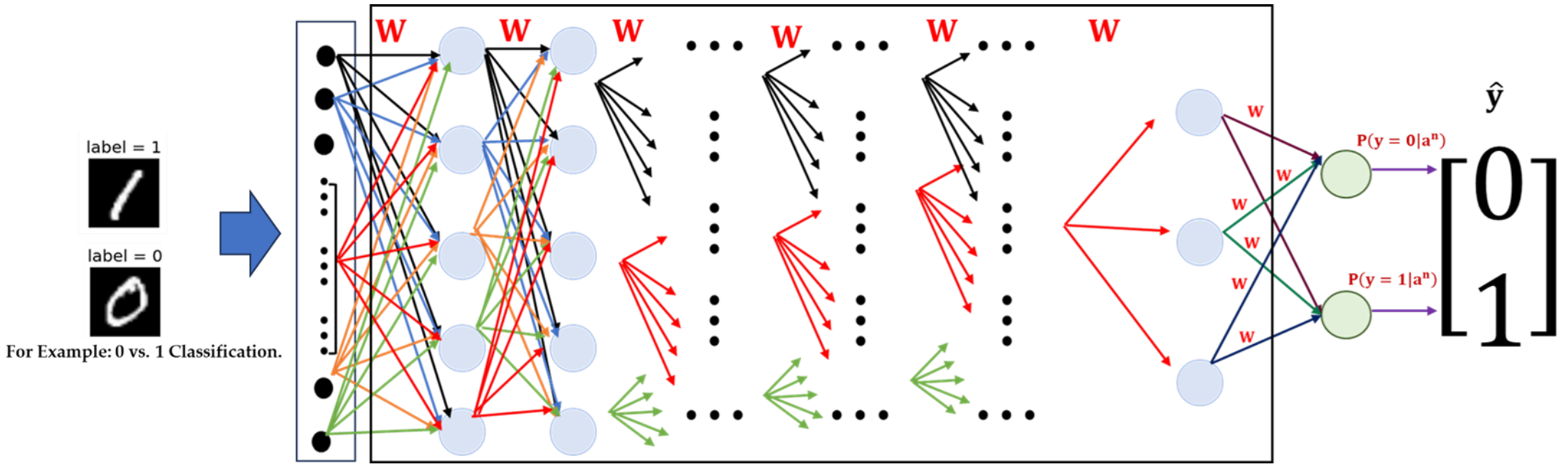
2.4.3 Understanding the Architecture: Number of Neurons.



Output Layer:

- The number of neurons in the output layer depends on the classification task and the activation function used:
  - **For Binary Classification:**
    - 1 neuron with sigmoid activation function,
      - where the output is interpreted as a probability and a threshold (0.5) is used for decision making.
    - 2 neurons with a softmax activation function can also be used as demonstrated above (**but not a common practice**),
      - where each neuron represents a class (e.g. 0 vs. 1) and probability of input belonging particular class is computed.
  - **For Multiclass Classification:**
    - The number of neurons equals to the number of classes.
    - **Each neuron** computes the probability of the input belonging to its corresponding class, **mostly** using a softmax activation function.

2.4.3.1 Understanding the Architecture: Number of Neurons.



Cautions:

- In the above example, **two neurons were used for binary classification solely for demonstration purposes**, even though this is not necessary.
- I wanted to **demonstrate the general practice with DNN and multiclass classification**,
  - as the **majority of tasks we perform with DNN in this module will be multiclass in nature**,
    - where the **number of output neurons** typically **corresponds** to the **number of classes**, with a **softmax activation** function.

2.5 Training the DNN or Multi – Layer Neural Network.

- How can we learn the weights and biases or parameters in Deep Neural Network or Multi Layer Neural Network?

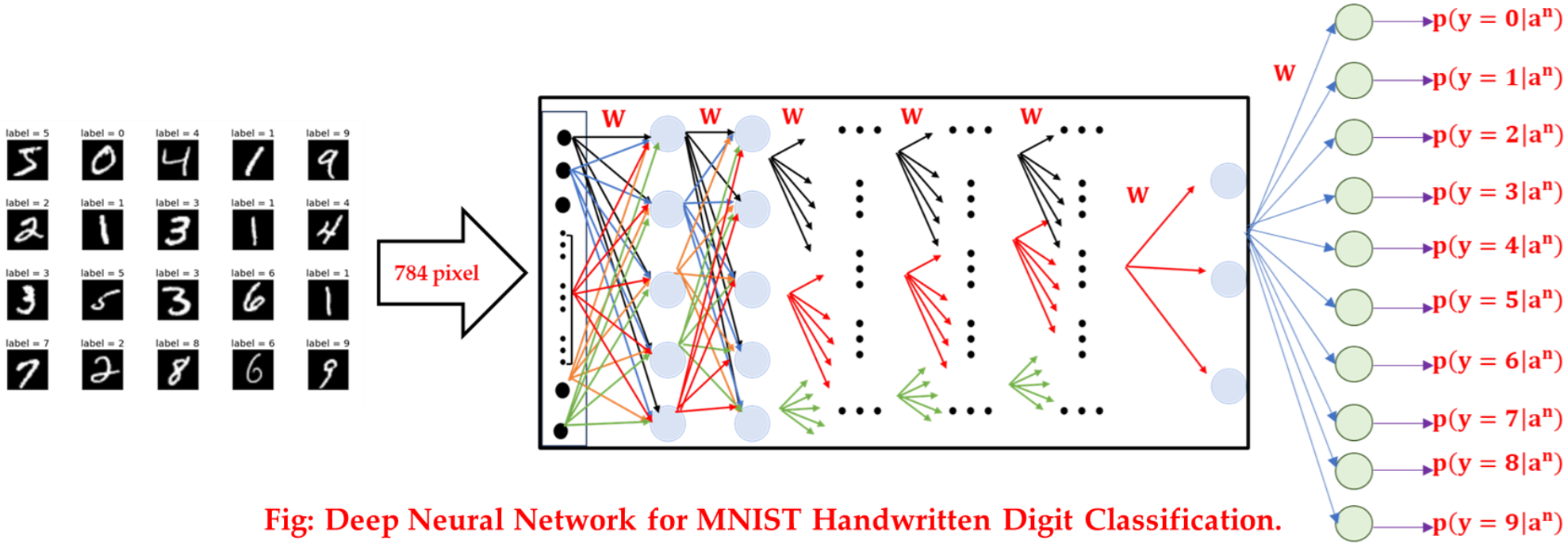


Fig: Deep Neural Network for MNIST Handwritten Digit Classification.

How can we train such Model? Can we fit into ERM Framework.

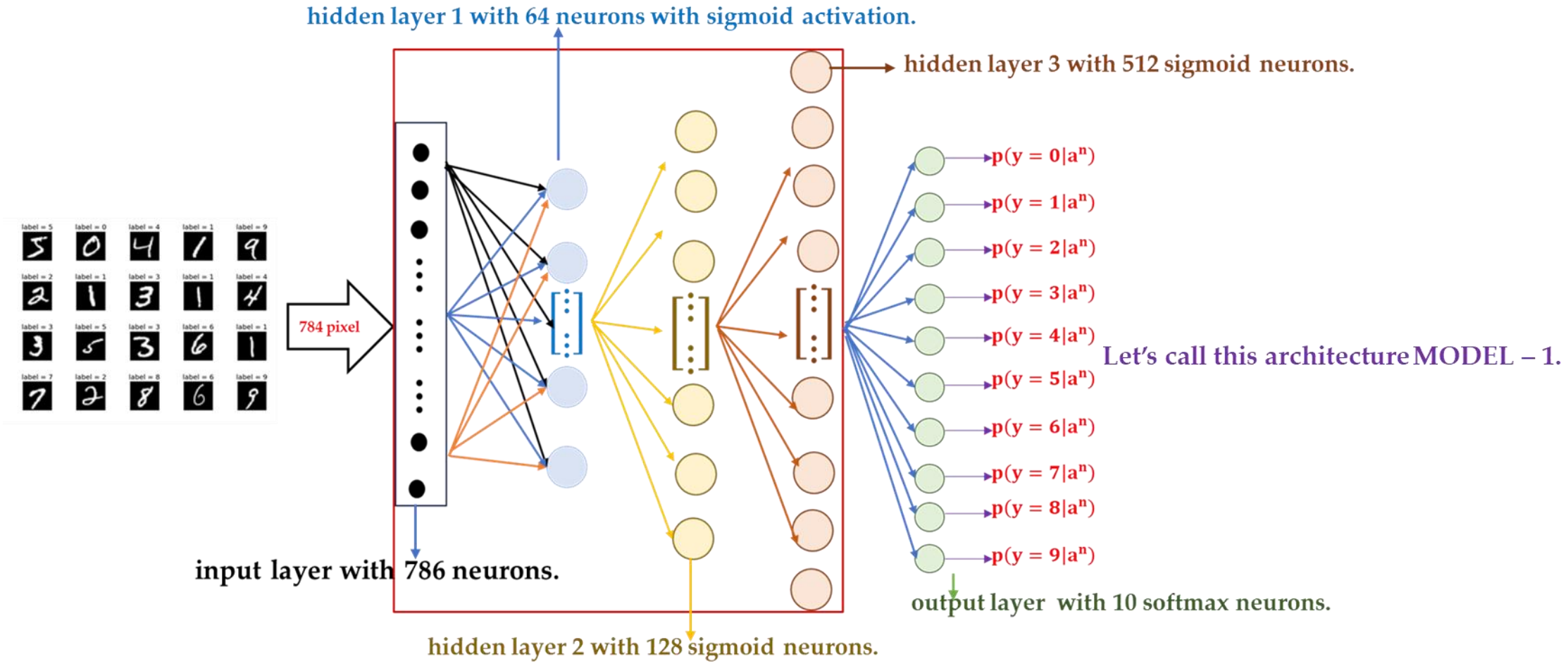
3. Training DNN within ERM Framework.

{ERM Framework – Decision Process, Loss Function, Empirical Risk, Optimization}

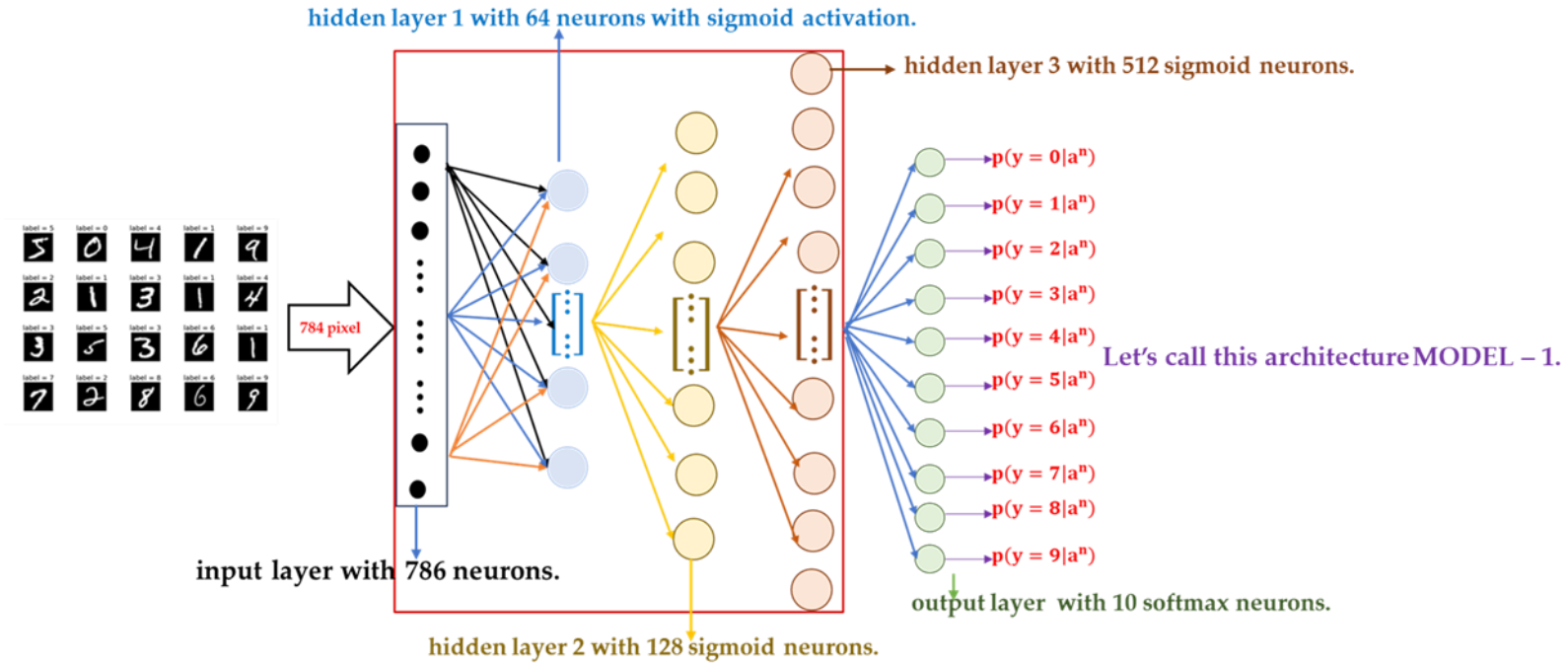
3.1 Defining a DNN as a Model.

- Unlike machine learning models, deep learning models are mathematical representations of deep neural networks consisting of multiple layers.
  - Thus, the models vary in the number of layers and the number of neurons (or architecture).
- The architecture depends on the number of inputs and the number of classes in the dataset.
- Thus, the architecture mainly depends on the nature of the task.
- For example, in MNIST digit classification,
  - the first layer needs to have 784 neurons as input placeholders (since each image is 28x28 pixels, flattened into a 784-dimensional vector).
  - We can have three hidden layers with 64, 128, and 512 neurons, each using the sigmoid activation function. { Design choice made by developing engineers}
  - The output layer would have 10 neurons (one for each digit) with the softmax activation function.
    {Detailed drawing in Next Slide}

3.1.1 Defining a DNN for MNIST Digit Classification Task.



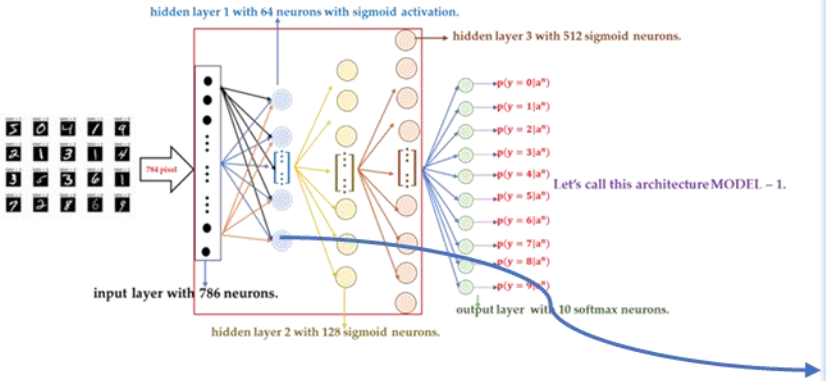
3.1.2 Mathematical Representation of Model – 1.



1. Input Layer:

Input $\mathbf{x} \in \mathbb{R}^{784}$, i.e. $\mathbf{x} \rightarrow \begin{bmatrix} \mathbf{pixel}_0 \\ \mathbf{p}_{\mathbf{ixel1}} \\ \vdots \\ \vdots \\ \mathbf{pixel}_{784} \end{bmatrix}$ {flattened $28 \times 28 \times 1$ image}

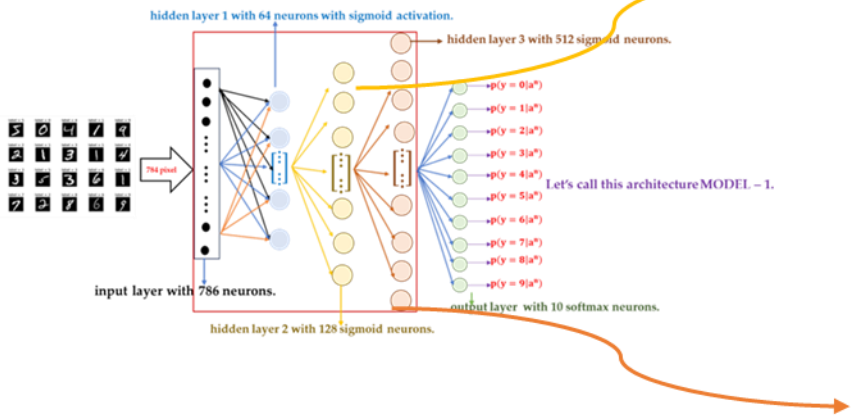
3.1.3 Mathematical Representation of Model – 1.



2. Hidden Layer:

- For each hidden layer, the output is a transformation(weighted sum and activation) of the previous layer's output using the weight matrix and bias vector.
  - First Hidden Layer - 64 neurons.
  - The output $h^1 \in \mathbb{R}^{64}$ is computes as:
  - $\mathbf{h^1 = \sigma(z^1) \rightarrow \{z^1 = W^1 x + b^1\}}$
- here:
  - $W^1 \in \mathbb{R}^{64\times784}$ is a weight matrix for first hidden layer.
  - $W^1 = \begin{bmatrix} w^1_{1,1} & w^1_{1,2} & \cdots & \cdots & \cdots & w^1_{1,784} \\ w^1_{1,2} & \ddots & \vdots & \vdots & \vdots & \vdots \\ \vdots & \vdots & \ddots & \vdots & \vdots & \vdots \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ w^1_{64,1} & w_{64,2} & \cdots & \cdots & \cdots & w_{64,784} \end{bmatrix}; x = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ \vdots \\ x_{784} \end{bmatrix}; b^1 = \begin{bmatrix} b^1_1 \\ b^1_2 \\ \vdots \\ \vdots \\ b^1_{64} \end{bmatrix}$
  - $z^1 = \begin{bmatrix} z^1_1 \\ \vdots \\ \vdots \\ z^1_{64} \end{bmatrix}$; each element computes as: $z^1_i = \sum_{j=1}^{784} w_{ij} + b_i$,
  - **In Matrix form:**
    - $z^1 = \begin{bmatrix} w^1_{1,1}x_1 + w^1_{1,2}x_2 + \cdots + w^1_{1,784}x_{784} + b^1_1 \\ w^1_{2,1}x_1 + w^1_{2,2}x_2 + \cdots + w^1_{2,784}x_{784} + b^1_2 \\ \vdots \\ w^1_{64,1}x_1 + w^1_{64,2}x_2 + \cdots + w^1_{64,784}x_{784} + b^1_{64} \end{bmatrix}$
  - $h^1 = \sigma\left(\begin{bmatrix} z^1_1 \\ \vdots \\ \vdots \\ z^1_{64} \end{bmatrix}\right) \Rightarrow \begin{bmatrix} \sigma(z^1_1) \\ \vdots \\ \vdots \\ \sigma(z^1_{64}) \end{bmatrix}$ {element wise sigmoid activation function. }

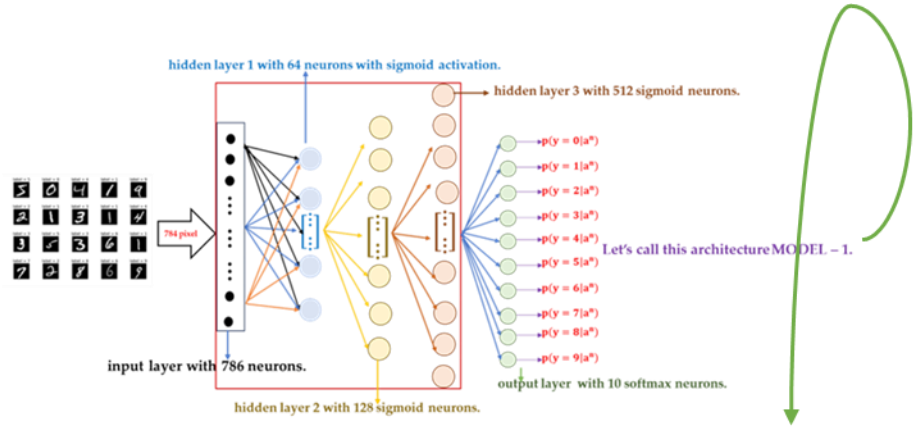
3.1.4 Mathematical Representation of Model – 1.



2. **Hidden Layer:**
   - **Second Hidden Layer – 128 neurons:**
     - The output: $\mathbf{h^2 = \sigma(z^2)\{z^2 = W^2 h^1 + b^2\}}$
       - Here:
         - $\mathbf{W^2 \in \mathbb{R}^{128\times 64}}$ is the **weight matrix** for the second hidden layer.
         - $\mathbf{b^2 \in \mathbb{R}^{128}}$ is the **bias vector** for the second hidden layer.
         - $\mathbf{h^1 \in [0\ to\ 1]^{64}}$ is the **vector of activated output** from **hidden layer 1**.

2. **Hidden Layer:**
   - **Third Hidden Layer – 512 neurons:**
     - The output: $\mathbf{h^3 = \sigma(z^3)\{z^3 = W^3 h^2 + b^3\}}$
       - Here:
         - $\mathbf{W^3 \in \mathbb{R}^{512\times 128}}$ is the **weight matrix** for the second hidden layer.
         - $\mathbf{b^3 \in \mathbb{R}^{512}}$ is the **bias vector** for the second hidden layer.
         - $\mathbf{h^2 \in [0\ to\ 1]^{128}}$ is the **vector of activated output** from **hidden layer 2**.

3.1.5 Mathematical Representation of Model – 1.



3. Output Layer – 10 Neurons:

- The Output $\hat{y} \in \mathbb{R}^{10}$ (the predicted class probabilities) is computed as:
  - $\hat{y} = \mathbf{Softmax}(z^4)\{z^4 = W^4 h^3 + b^4\}$
- Here:
  - $W^4 \in \mathbb{R}^{10\times512}$ is the **weight matrix** for the **output layer**. 10 because there are 10 classes.
  - $b^4 \in \mathbb{R}^{10}$ is the **bias vector** of the **output layer**.
  - The softmax function is applied element wise to the vector:
    - $\hat{y}_j = \frac{e^{(W^4 h^3 + b^4)_j}}{\sum_{k=1}^{10} e^{(W^4 h^3 + b^4)_k}} \; \forall j \in \{1, 2, \dots, 10\}$
  - The **predicted class** $\hat{y}$ is the index of **the maximum value in** $\hat{y}$:
    - $\hat{y} = \mathbf{argmax}_j \, \hat{y}_j$.

3.2 Loss Function: Categorical Cross Entropy.

- **Loss function – Categorical Cross Entropy (aka CE loss):**
  - **Categorical Cross-Entropy Loss** measures how well the predicted probability distribution matches the true class labels in a classification task.
  - It is widely used in multiclass classification problems.
  - The goal of the loss is to maximize the probability of the correct class.
- **Formula**: For a single sample $\mathbf{x_i}$ with true label $\mathbf{y_i}$ (one hot encoded) and predicted probabilities $\hat{\mathbf{y}}_\mathbf{i}$, the loss is:
  - $\ell(\mathbf{y_i}, \hat{\mathbf{y}}_\mathbf{i}) = -\sum_{\mathbf{k=1}}^{\mathbf{C}} \mathbf{y_{ik}} \mathbf{log}(\hat{\mathbf{y}}_{\mathbf{ik}})$
  - Where:
    - $\mathbf{C} \rightarrow$ **is the number of classes.**
    - $\mathbf{y_{ik}} = \mathbf{1}: \rightarrow$ if the **k-th class** is the true class for sample i, otherwise $\mathbf{y_{ik}} = \mathbf{0}$ {one hot encoding}.
    - $\hat{\mathbf{y}}_{\boldsymbol{ik}} \rightarrow$ is the predicted probability **for class k**.
  - For a dataset of n samples, the average loss (**empirical risk**) given by:
    - $\widehat{\mathbf{R}} = -\frac{\mathbf{1}}{\mathbf{n}} \sum_{\mathbf{i=1}}^{\mathbf{n}} \sum_{\mathbf{k=1}}^{\mathbf{C}} \mathbf{y_{ik}} \, \mathbf{log}(\hat{\mathbf{y}}_{\mathbf{ik}})$ **{also called cost function}**

3.2.1 Do CE Loss measures the Difference?

- For Example:
  - We built a model that can classify following three classes present in our data:

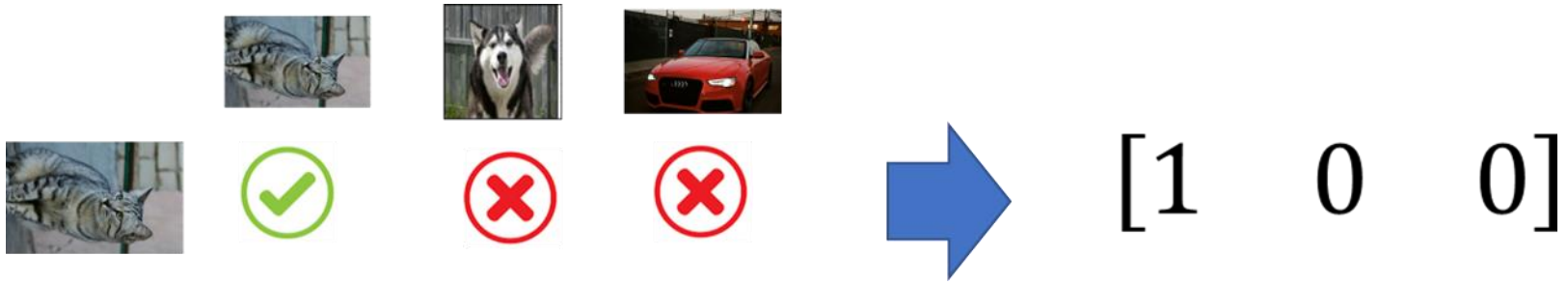
Cat

Dog

Car

3.2.2 Should Know: Target Representation.

- We represent **our target variable** with its **one-hot-encoded version** i.e.
  - Our First Data point $\{y_1\}$ is image of cat then one hot encoded representation will be:



3.2.2 Should Know: Target Representation.

- We represent our target variable with its one-hot-encoded version i.e.
  - Our First Data point $\{y_1\}$ is image of cat then one hot encoded representation will be:

$$[1 \quad 0 \quad 0]$$

  - Our Second Data point $\{y_2\}$ is image of dog then one hot encoded representation will be:

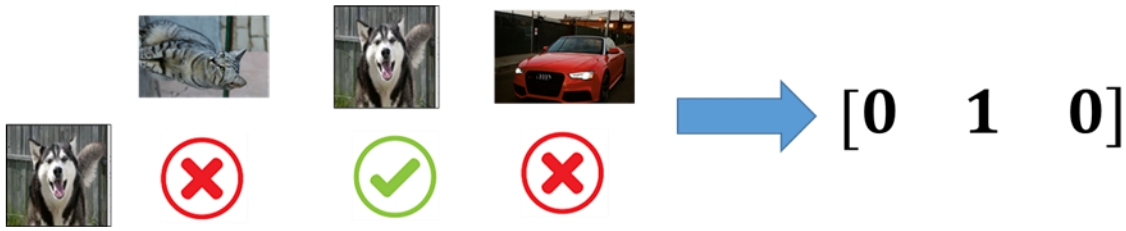
$$[0 \quad 1 \quad 0]$$

3.2.2 Should Know: Target Representation.

- We represent our target variable with its one-hot-encoded version i.e.
  - Our First Data point $\{y_1\}$ is image of cat then one hot encoded representation will be:

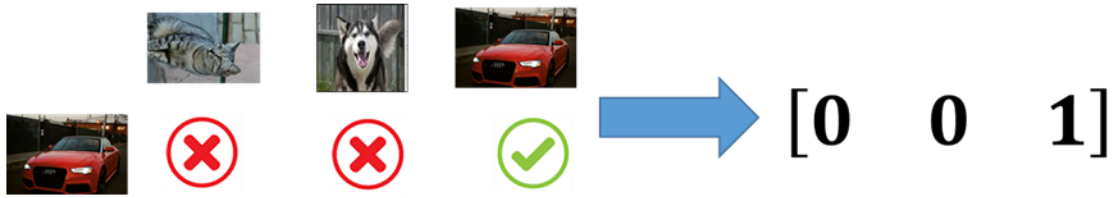
$$[1 \quad 0 \quad 0]$$

  - Our Second Data point$\{y_2\}$ is image of dog then one hot encoded representation will be:



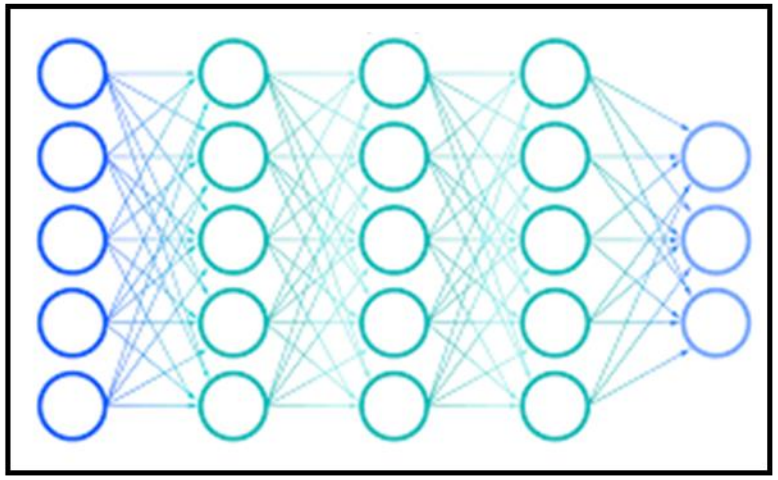
$$[0 \quad 1 \quad 0]$$

  - Our Second Data point $\{y_3\}$ is image of car then one hot encoded representation will be:



$$[0 \quad 0 \quad 1]$$

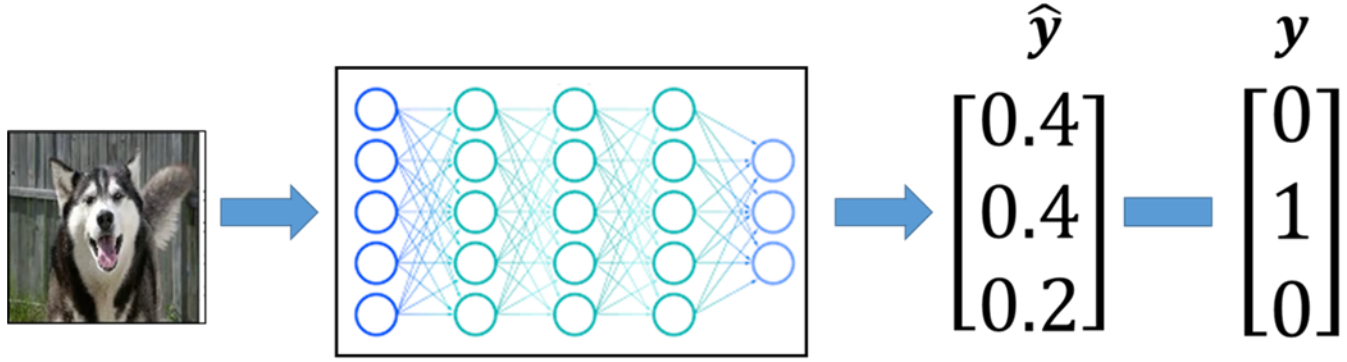
We build a Network as shown below:



$$\hat{y} = \begin{bmatrix} 0.4 \\ 0.4 \\ 0.2 \end{bmatrix} \quad y = \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix}$$

Loss Function must be high or low?

Compute a Loss:



Loss must be high or low?

Loss Calculation:

$$\mathbb{L}(y, \hat{y}) = -\sum_{i}^{C} y_i \log \hat{y}_i$$

$$= -0 \times \log(0.4) - 1 \times \log(0.4) - 0 \times \log(0.2)$$

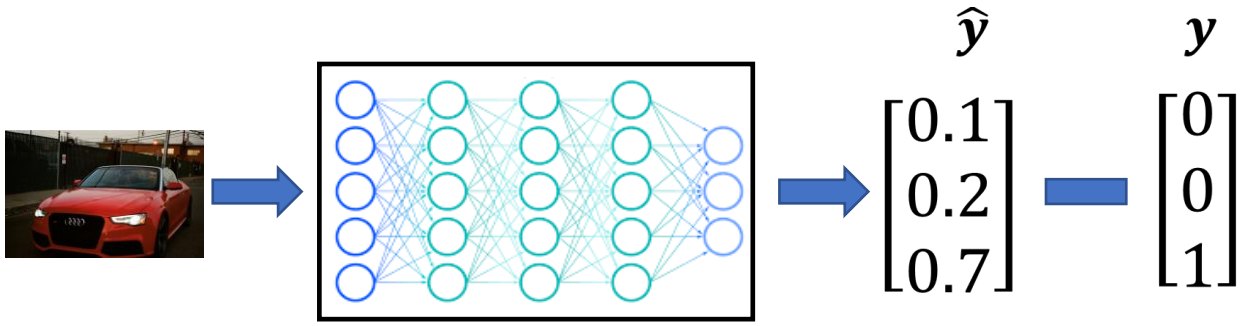
$$= 0.92$$

Why the loss function is high?

- **Model is confused between whether the image is cat or dog i.e.**
  - **Model is 40% image is of cat and 40% image is of dog.**

For Example: Do it Yourself.

- **Loss must be high or low?**



3.3 Training the DNN within ERM Framework.

- **Multi layer Neural Network or DNN as Model Fitting Problem:**
  - **ERM Objective {Explicitly for DNN with Softmax in Output Layer}:**
    - The objective is to minimize the average loss (empirical risk) over the training dataset:
      - $\mathcal{L}(\mathbf{W}, \mathbf{b}) = -\frac{1}{n}\sum_{i=1}^{n}\sum_{k=1}^{C} y_{ik} \log(\hat{y}_{ik})$;
    - Substituting $\hat{y}_{ik} = \frac{\exp(z_i)}{\sum_{k=1}^{C}\exp(z_{ik})}$, the loss can be written as:
      - $\mathcal{L}(\mathbf{W}, \mathbf{b}) = -\frac{1}{n}\sum_{i=1}^{n} \log\left(\frac{\exp(z_i)}{\sum_{k=1}^{C}\exp(z_{ik})}\right)$;
    - here $\mathbf{z_i} = \mathbf{W}^T\mathbf{x_i} + \mathbf{b}$; $\mathbf{W} \in \mathbb{R}^{d\times C}$ → is the weight matrix and $\mathbf{b} \in \mathbb{R}^{C}$ → is the bias vector.
  - **Formulating as an Optimization problem:**
    - For any parameter(s) → $\boldsymbol{\theta}^* = [\mathbf{w}, \mathbf{b}: \mathbf{w} \in \mathbb{R}^{d}, \mathbf{b} \in \mathbb{R}] \in \boldsymbol{\Theta}$:
      - $\boldsymbol{\theta}^* = \min_{\boldsymbol{\theta}^*} \mathcal{L}(\mathbf{w}, \mathbf{b})$
    - This means **finding** the *weight vector* $\mathbf{w} \in \mathbb{R}^{d\times C}$ and *bias term* $\mathbf{b} \in \mathbb{R}^{C}$ that *minimize the average log loss* over the *training data*.

Towards Gradient Descent!!

4. Computing Gradients.

{ Forward and Backward Propagation with Gradient Descent.}

4.1 Computing Gradients: Forward and Backward Propagations.

- The weights in Multi layer networks are learned with the **combinations** of **forward and backward propagations.**
  - a network forward propagates activation to produce an output and it backward propagates error to determine weight changes

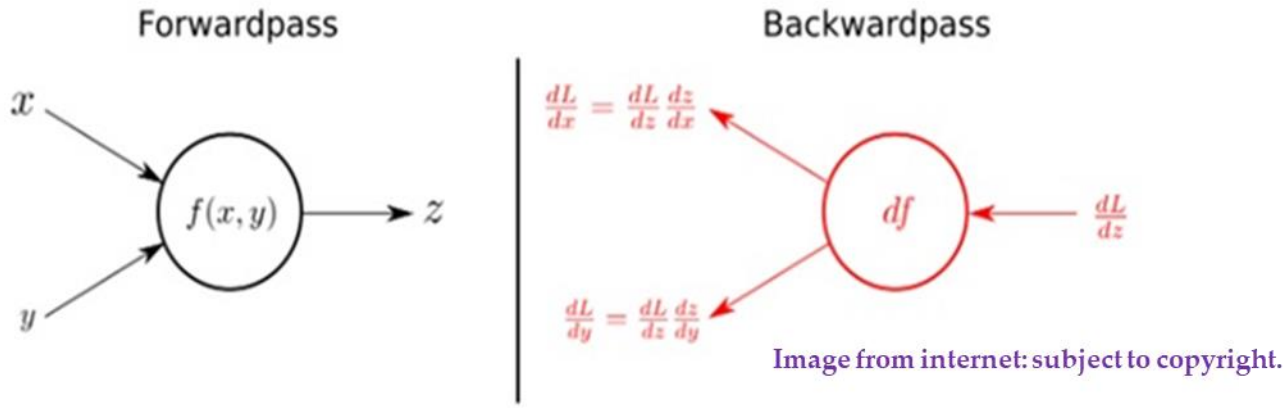
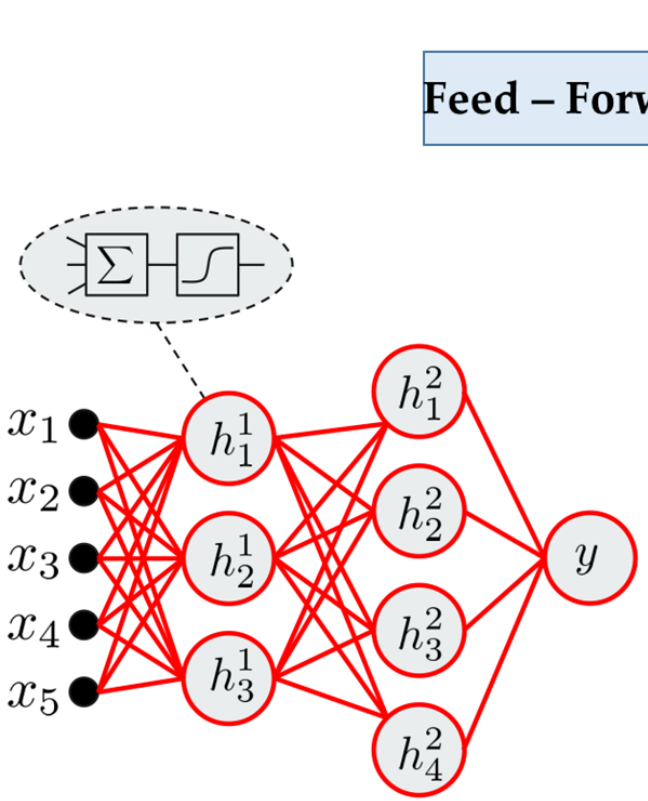


Image from internet: subject to copyright.

4.2 Forward Propagation.



- $\mathbf{h_i^1 = act(W_i^1 \cdot x + b_i^1)}$
- $\mathbf{h_i^2 = act(W_i^2 \cdot h_i^1 + b_i^2)}$
- $\mathbf{\hat{y}_i = act(W_i^3 \cdot h_i^2 + b_i^3)}$

4.3 Backward Propagation.

- **Backpropagation** is a technique used by deep layer networks to find the error of the network.
  - The error is calculated by comparing an expected output with a predicted output, this algorithm then propagated these errors backward to update weights and biases.

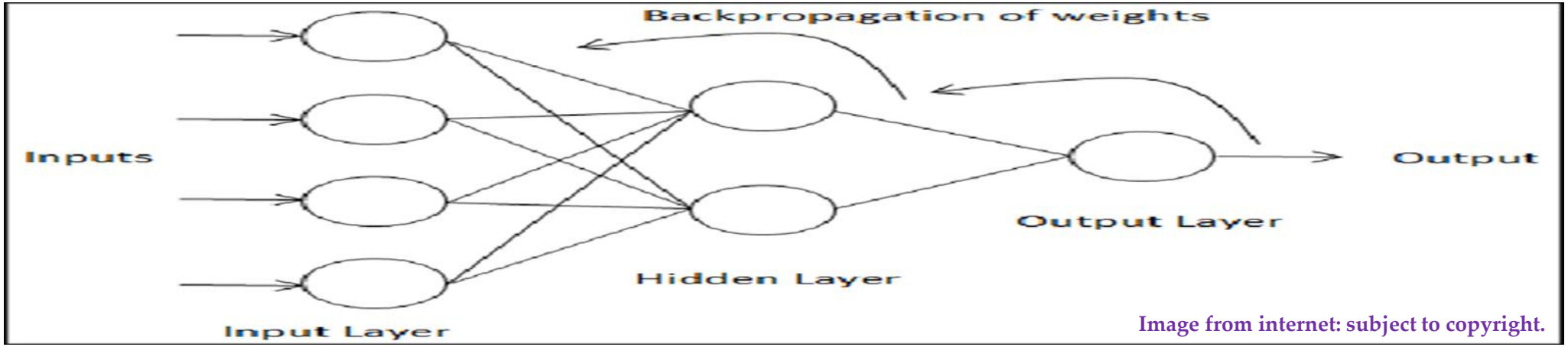
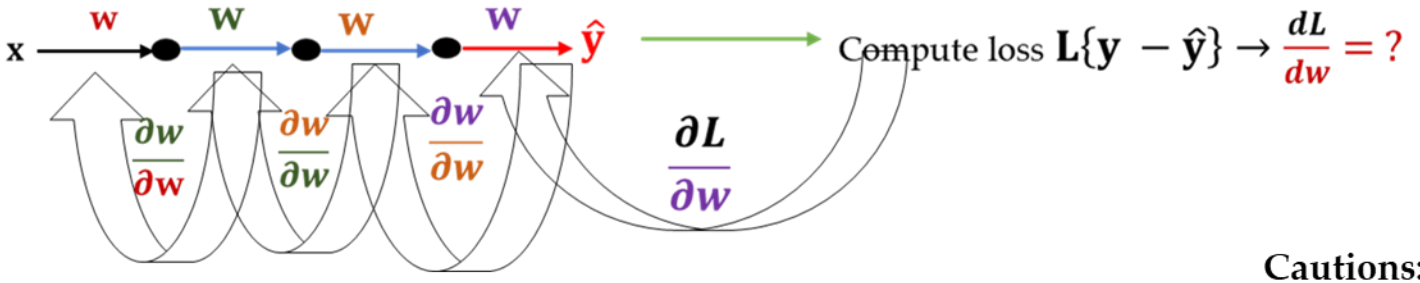


Image from internet: subject to copyright.

4.3.1 Idea Behind Backpropagation.

- Let's look into following network architecture:

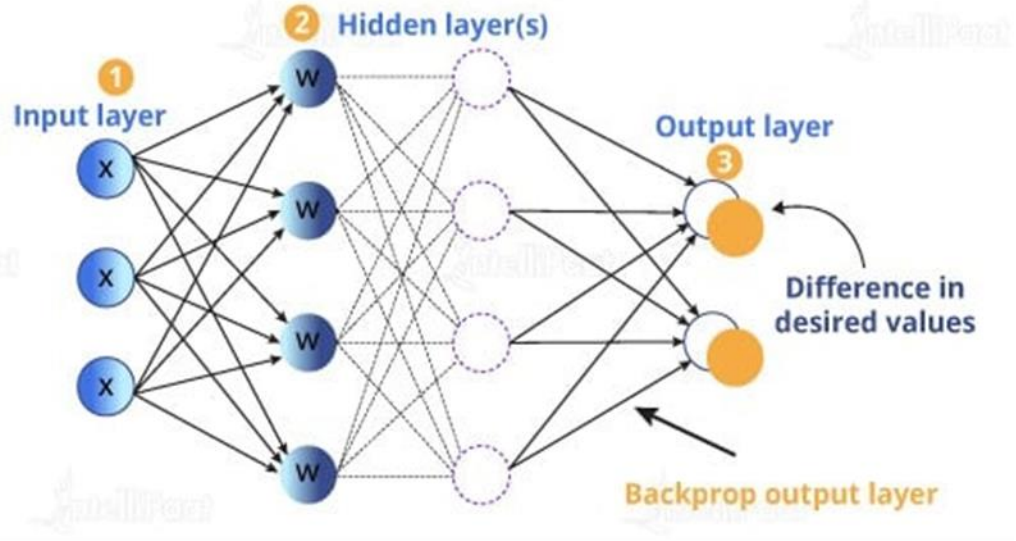
  - Now we want to update our weight at first layer **w** using gradient descent for which we need to compute:
    - $\frac{dL}{dw} = ?$
  - We can compute such using chain rule of derivative as:
    - $\frac{dL}{dw} = \frac{\partial L}{\partial w} \cdot \frac{\partial w}{\partial w} \cdot \frac{\partial w}{\partial w} \cdot \frac{\partial w}{\partial \mathbf{w}}$



Cautions: Only for demonstration purposes.

4.4 Backpropagation Requirement.

- Backprop Requires following three things:
  1. Dataset: in the pairs of input-output i.e. $(\mathbf{x_i}, \mathbf{y_i})$.
  2. A feed-forward neural network;
  3. An Error Function, E which defines the error between the desired output and calculated output.



4.5 Backpropagation Algorithm.

- Backpropagation in Practice are implemented through Computational Graph; we will discuss more on this in Tutorial session.
- Following is the General Framework for implementation of Backpropagation with Gradient Descent.

Algorithm 2 Backpropagation algorithm for feedforward networks.

Require: a set of training examples D, learning rate α.

1. Create a feed-forward network with n_{in} inputs, n_h hidden units, and n_o output units.
2. Initialize all network weights to *small* random numbers.
3. **Repeat** until the termination condition is met:

   For each training example $(\mathbf{x}, \mathbf{t}) \in D$ **do:**

   Propagate the input $\mathbf{x}$ forward through the network, i.e.:

   1. Input $\mathbf{x}$ to the network and compute the output a_k of units in the output layer.

   Backpropagate the errors through the network:

   1. **For** each network output unit k, calculate its error term δ_k:

$$\delta_k \leftarrow -a_k(1 - a_k)(t_k - a_k) \tag{3}$$

   2. **For** each hidden unit h, calculate its error term δ_h:

$$\delta_h \leftarrow a_h(1 - a_h) \sum_k \delta_k w_{kh} \tag{4}$$

   3. Update each network weight w_{ji}

$$w_{ji} \leftarrow w_{ji} + \Delta w_{ji}$$

   where

$$\Delta w_{ji} = -\alpha \delta_j x_{ji}$$

Notations:

- The **subscript k denotes** the **output layer**.
- The **subscript j denotes** the **hidden layer**.
- The **subscript i denotes** the **input layer**

5.Variants of Gradient Decent.

{Making it Faster and Efficient.}

5.1 Gradient descent variants.

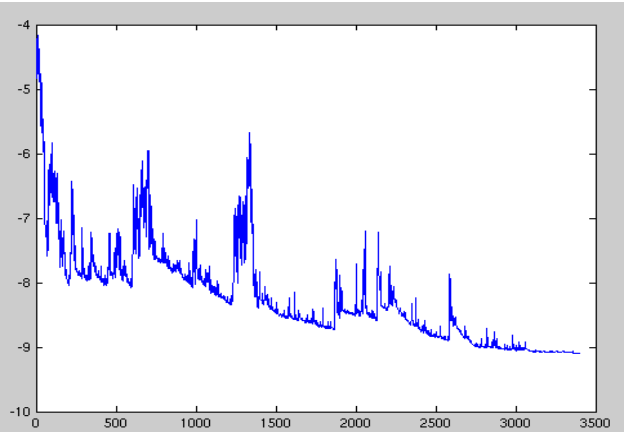
- Based **on how much data we use to compute the gradient** of the **objective function**, there are **three main variants of gradient descent**:
  - **Batch Gradient Descent.**
  - **Stochastic Gradient Descent.**
  - **Mini-Batch Gradient Descent.**
- **More data** may **mean more accuracy of the parameter update**, but it also means more **time needed to reach convergence**, thus the **trade off** is necessity.

5.2 Batch Gradient Descents.

- Also known as vanilla gradient descent, **computes the gradient** of the **cost function** w.r.t to the **parameters w** for the **entire training set**:
  - **Update rule:**
    - $w = w - \alpha \nabla_w J(\theta)$.
- It can be **very slow** **and is intractable** for datasets **that do not fit the memory**.

5.3 Stochastic Gradient Descent.

- In contrast SGD performs a parameter update **for each training example {loss function}** i.e. x^i **and label** y^i.
  - **Update Rule:**
  - $w = w - \alpha \nabla_w J(w; x^i; y^i)$.
- It is usually **much faster compared to BGD**, and also **can be used to learn online**.
- SGD performs frequent updates with a high variance that cause the objective function to fluctuate heavily as in image:
  - It can enable it to jump to new and potential better local minima.
  - It can also ultimately complicate the convergence to the exact minimum.



5.4 Mini Batch Gradient Descent.

- It is the mixture of BGD and SGD i.e. it updates the parameter for every mini batch of n training examples:
  - **Update Rule:**
  - $\mathbf{w = w - \alpha \nabla_w J(w; x^{(i:i+n)}; y^{(i:i+n)}}$.
- Merits:
  - Reduces the variance of the parameter updates, which can lead to more stable convergence;
  - Efficient computation for large models.
- Common mini-batch size range between 50 and 256.

5.5 Challenges : Gradient Descent.

- Same learning rate applies to all parameter updates.
  - Getting trapped into suboptimal local minima or saddle points.
  - Choosing a Proper Learning Rate.
    - **Annealing:**
      - Learning rate schedules or reducing the learning rate according to a pre-defined schedule or it becomes smaller than a pre-set threshold.
      - Schedules and threshold values have to be pre-defined, thus may not be able to adapt to a dataset's characteristics.

5.6 Basic Algo for: Mini-Batch Gradient Descent.

- Guess an **initial learning rate**.
  - If the **error** keeps getting worse or **oscillates** wildly, **reduce** the **learning rate**.
  - If the **error** is falling **fairly** consistently but slowly, increase the **learning rate**.
- Write a simple program to automate this way of **adjusting** the **learning rate**.
- Towards the end of mini-batch learning it nearly always helps to **turn down** the **learning rate**.
  - This removes **fluctuations** in the **final weights** caused by the **variations between mini-batches**.
- Turn down the **learning rate** when the **error stops decreasing**.
  - Use the error on a separate validation set

Thank You