Basics of Deep Learning

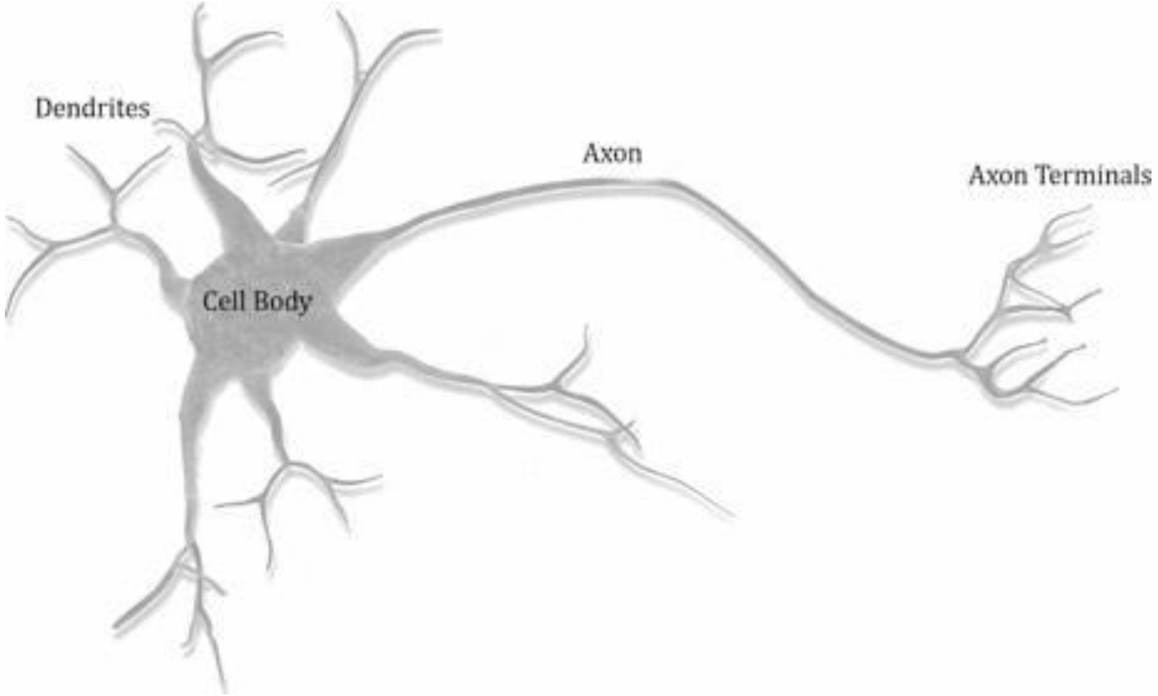
By

Professor M. Shahidur Rahman

DoCSE, SUST

Perceptron

- The perceptron is loosely inspired by biological neurons, connecting multiple inputs (signals to dendrites), combining and accumulating these inputs (in the cell body proper), and producing an output signal that resembles an axon.

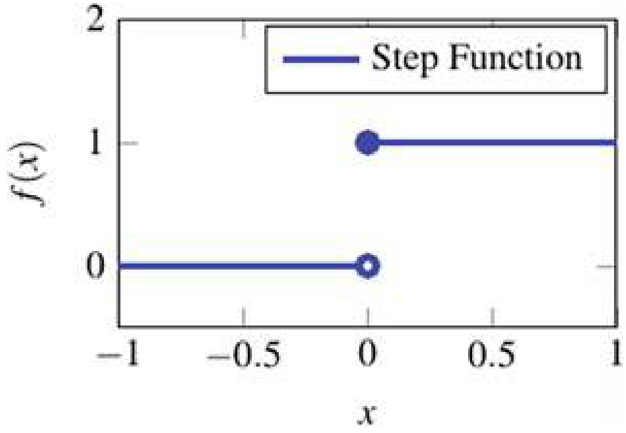


Perceptron Algorithm

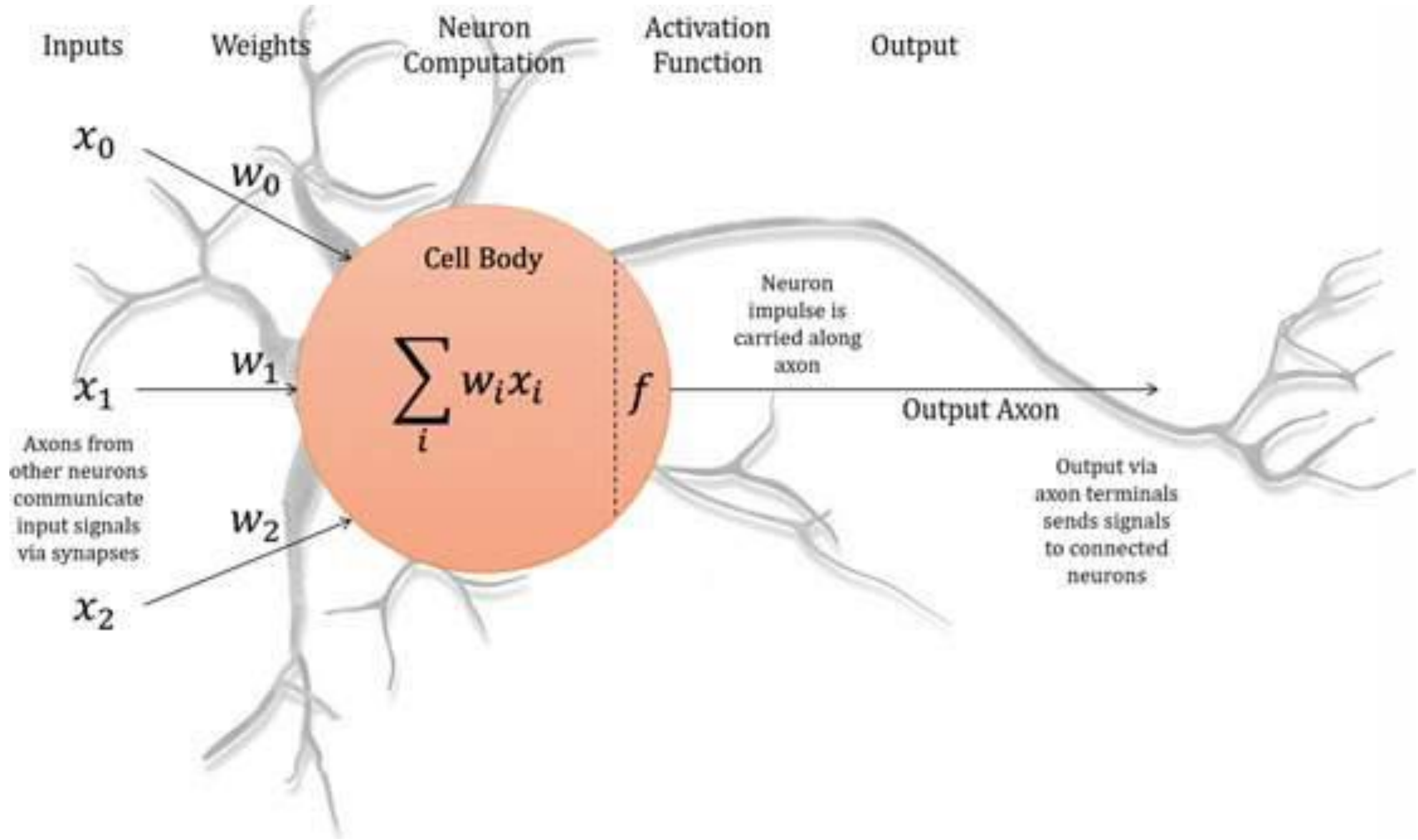
- Deep learning in its simplest form is an evolution of the perceptron algorithm, trained with a gradient-based optimizer.
- The perceptron algorithm is one of the earliest supervised learning algorithms, dating back to the 1950s.
- Much like a biological neuron, the perceptron algorithm acts as an artificial neuron, having multiple inputs, and weights associated with each input, each of which then yields an output.

$$y(x_1, \ldots, x_n) = f(w_1 x_1 + \ldots + w_n x_n)$$

$$f(v) = \begin{cases} 0 & \text{if } v < 0.5 \\ 1 & \text{if } v \geq 0.5 \end{cases}$$

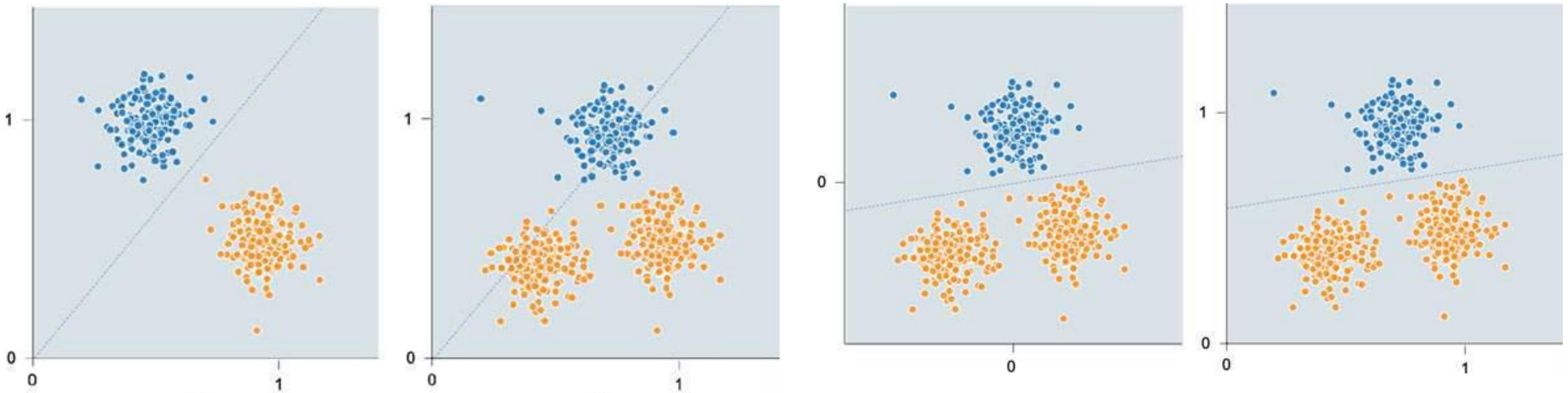


An Artificial Neuron



Bias

- The perceptron algorithm learns a hyperplane that separates two classes.
- At this point (left figure), the separating hyperplane cannot shift away from the origin. Restricting the hyperplane in this fashion causes issues.
- The bias allows the perceptron algorithm to relocate the separating plane (right figures), allowing it to correctly classify the data points.

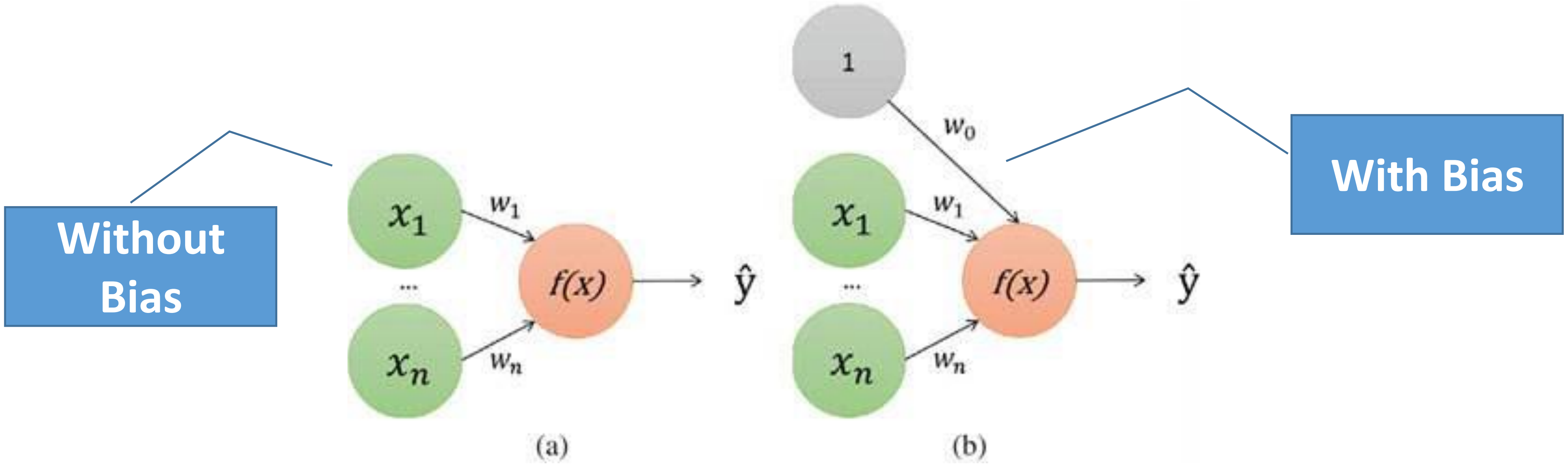


(a) (b)

Perceptron with Bias

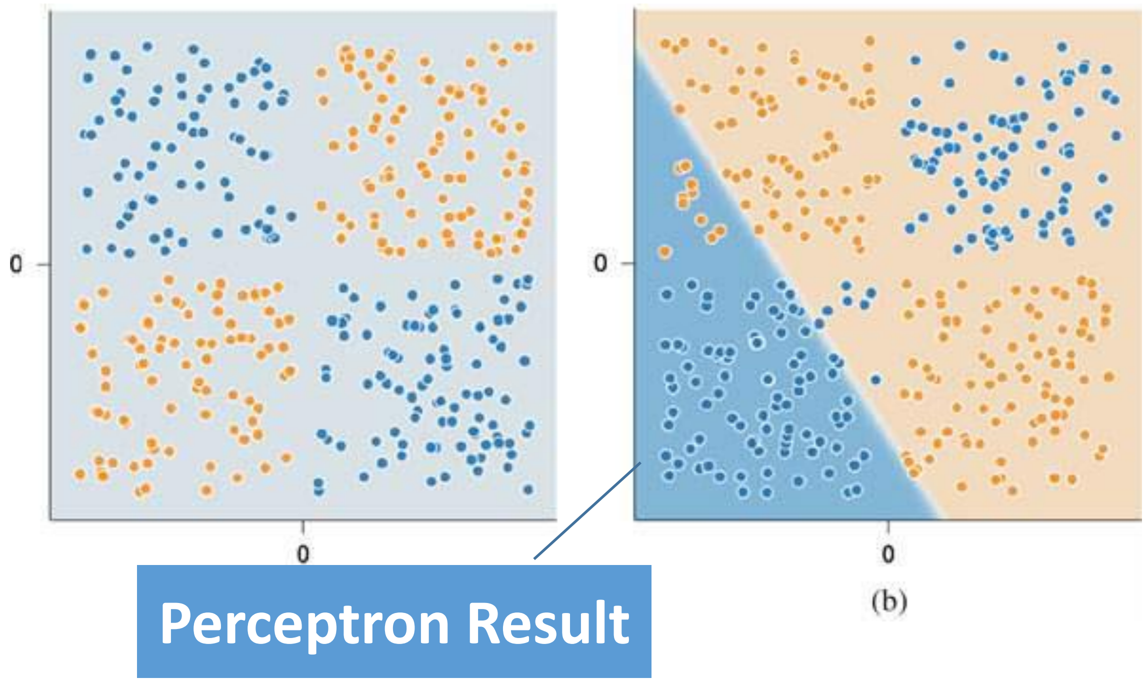
- perceptron with a bias term:

$$y(x_1, \ldots, x_n) = f(w_1 x_1 + \ldots + w_n x_n + b)$$



Linear and Non-linear Separability

- Two sets of data are linearly separable if a single decision boundary can separate them. $\sum_i w_i x_i \geq t$
- If we apply the perceptron to a non-linearly separable dataset, then we are unable to separate the data.
- Unfortunately, most data in NLP and speech is highly non-linear.



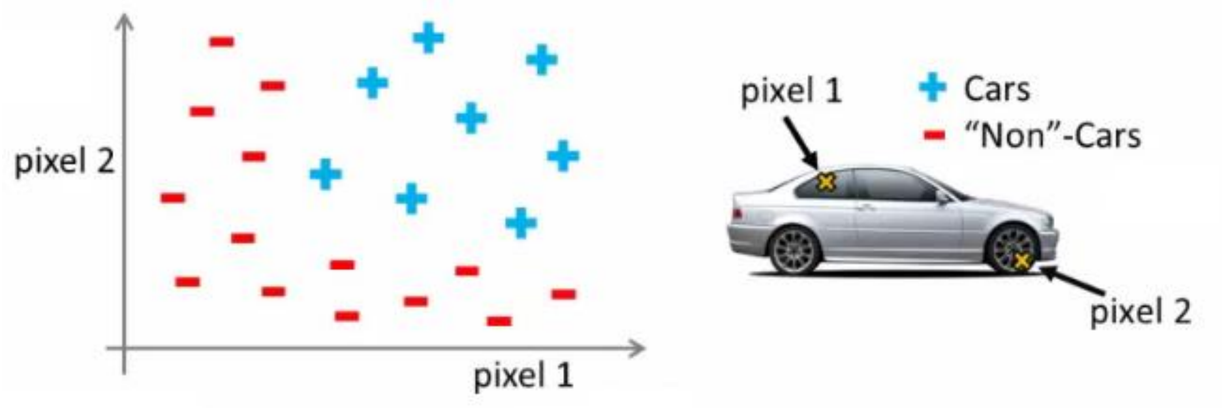
Linear and Non-linear Separability…

Say we have a complex supervised learning classification problem

- Can use logistic regression with many polynomial terms
- Works well when you have 1-2 features
- If you have 100 features and if you include all the quadratic terms (second order), there are lots of them (x_1^2 ,x_1x_2, x_1x_4 ..., x_1x_{100})
  - For the case of n = 100, you have about 5000 features
  - Number of features grows $O(n^2)$
- If you include the cubic terms, e.g. ($x_1^2x_2$, $x_1x_2x_3$, $x_1x_4x_{23}$ etc.)
  - There are even more features grows $O(n^3)$

Problems where *n* is large - Computer Vision

- To build a car detector
  - Build a training set of Cars/ Not cars
  - Then test against a car
- How can we do this
  - Plot two pixels (two pixel locations)
  - Plot car or not car on the graph
- Feature space
  - If we used 50 x 50 pixels --> 2500 pixels, so n = 2500
  - If RGB then 7500
  - If 100 x 100 pixels--> even more features
- Too big - Simple logistic regression here is not appropriate for large complex systems
- Neural networks are much better for a complex non-linear hypothesis even when feature space is huge

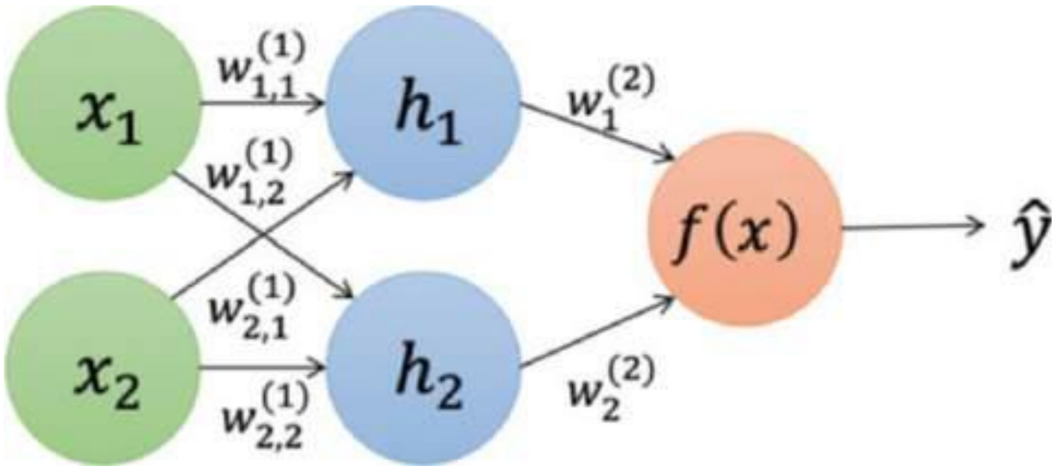


Multilayer Perceptron (Neural Networks)

- The multilayer perceptron (MLP) links multiple perceptrons together into a network.
- Neurons that take the same input are grouped into a layer of perceptrons.
- Instead of using the step function, we substitute a differentiable, non-linear function.
- Applying this **non-linear function** (activation function), allows the output value to be a non-linear, weighted combination of its inputs, thereby creating non-linear features
- The MLP is composed of interconnected neurons and thus is **a neural network**.
- It is a **feed-forward** neural network, since there is one direction to the flow of data through the network.

Multilayer Perceptron…

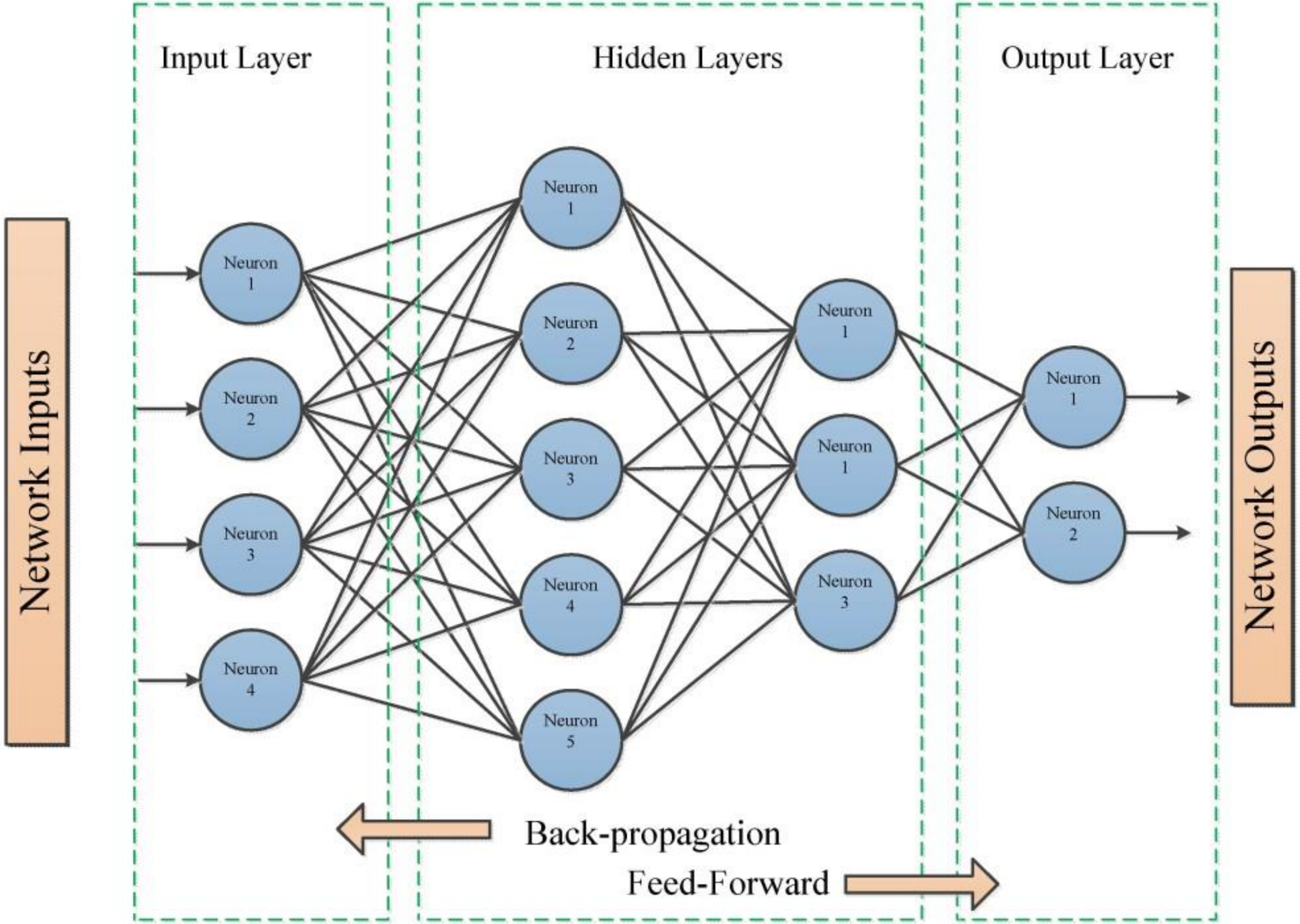
- An MLP must contain an input and output layer and at least one hidden layer.
- The layers are also "fully connected," meaning that the output of each layer is connected to each neuron of the next layer.
- $g(x)$ is the activation function and $f(x)$ is the output function, such as the step or sigmoid function.



$$\mathbf{h} = g(\mathbf{W}^{(1)}\mathbf{x})$$
$$\hat{y} = f(\mathbf{W}^{(2)}\mathbf{h})$$

where, $g(x)$ is the activation function and $f(x)$ is the output function, such as the step or sigmoid function.

Neural Network with Forward and Backward Propagation



Training an MLP/ Neural Network

Steps to train a neural network:

- Forward propagation: Compute the network output for an input example.
- Error computation: Compute the prediction error between the network prediction and the target.
- Backpropagation: Compute the gradients in reverse order with respect to the input and the weights.
- Parameter update: Use stochastic gradient descent (SGD) to update the weights of the network to reduce the error for that example.

Forward Propagation

- The first step in training an MLP is to compute the output of the network for an example from the dataset. We use the sigmoid function $\sigma(x)$ as the activation function.

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

$$f(v) = \sigma(v)$$

- The prediction $\hat{y}$ for an example *x*, where ***h*1 and *h*2 represent the respective layer outputs,** becomes:

$$\begin{aligned} \mathbf{h_1} &= f(\mathbf{W}_1\mathbf{x} + \mathbf{b}_1) \\ h_2 &= f(\mathbf{W}_2\mathbf{h}_1 + b_2) \\ \hat{y} &= h_2. \end{aligned}$$

- The bias $\mathbf{b}_1$ is a vector because there is a bias value associated with each neuron in the layer. There is only one neuron in the output layer, so the bias b_2 is a scalar.
- Once network is trained, a new example is evaluated thru forward propagation.

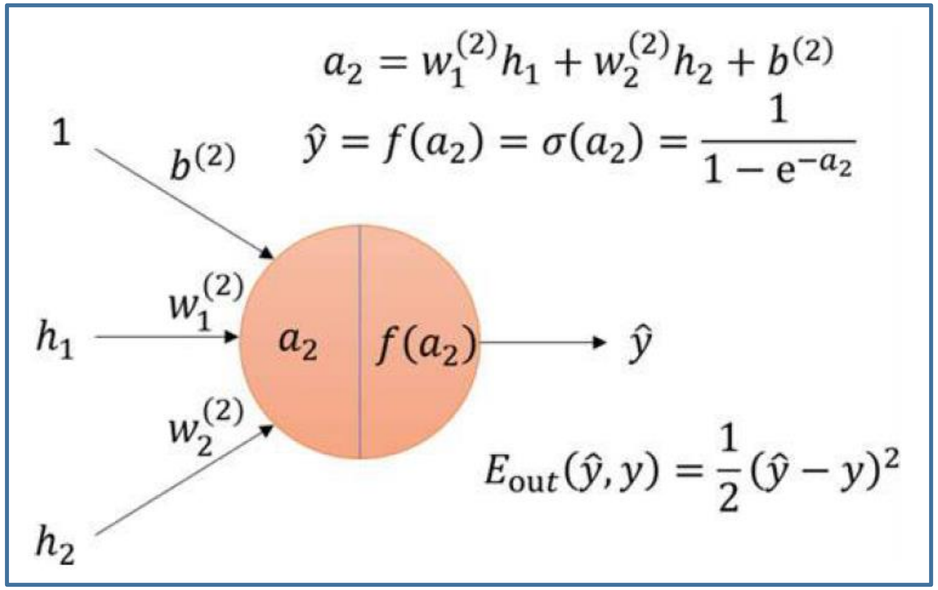
Error Computation

- The error computation step verifies how well our network performed on the example given. We use MSE as the loss function (for regression problem). The ½ simplifies backpropagation.

$$E(\hat{\mathbf{y}}, \mathbf{y}) = \frac{1}{2n} \sum_{i=1}^{n} (\hat{y}_i - y_i)^2$$

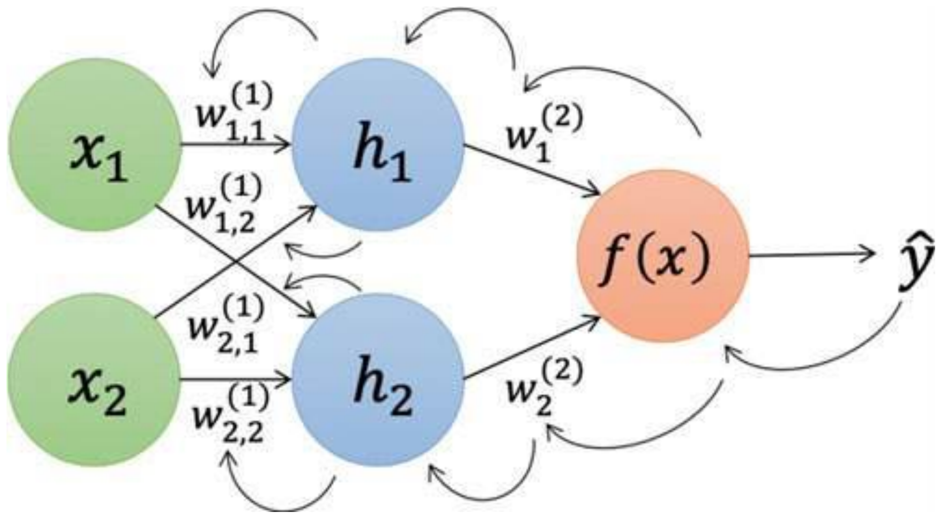
$$E(\hat{y}, y) = \frac{1}{2}(\hat{y} - y)^2$$

Error for single output



Backpropagation

- During forward propagation, an output prediction $\hat{y}$ is computed for the input x and the network parameters
- We can use SDG to decrease the error of the whole network via and the chain rule of calculus to compute the derivatives of each layer in the reverse order of forward propagation



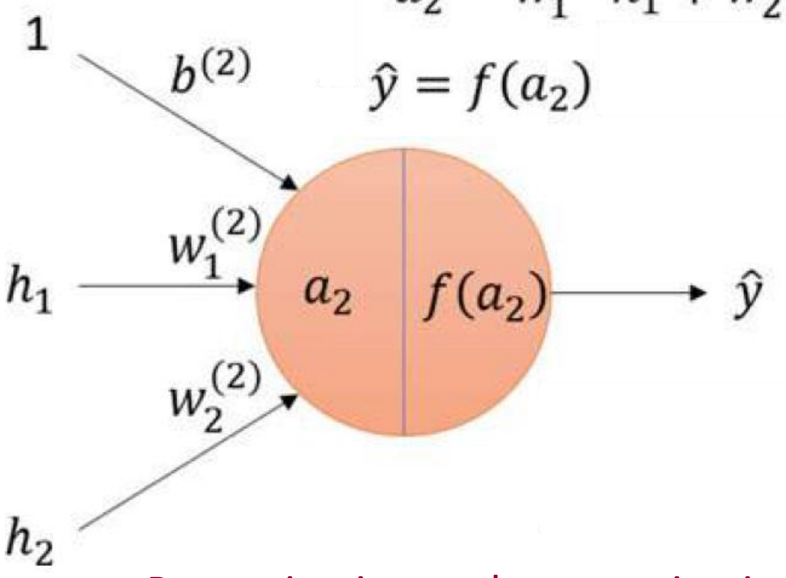
Backpropagation: Chain rule

- We begin by computing the gradient on the output layer with respect to the prediction.

$$\nabla_{\hat{y}} E(\hat{y}, y) = \frac{\partial E}{\partial \hat{y}} = (\hat{y} - y)$$

$$a_2 = w_1^{(2)} h_1 + w_2^{(2)} h_2 + b^{(2)}$$

$$\hat{y} = f(a_2)$$



Pre-activation and post-activation output

Backpropagation: Chain rule…

- We can then compute error with respect to the layer 2 parameters.

$$\nabla_{\mathbf{a}_2} E = \frac{\partial E}{\partial \mathbf{a_2}} = \frac{\partial E}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial \mathbf{a}_2}$$

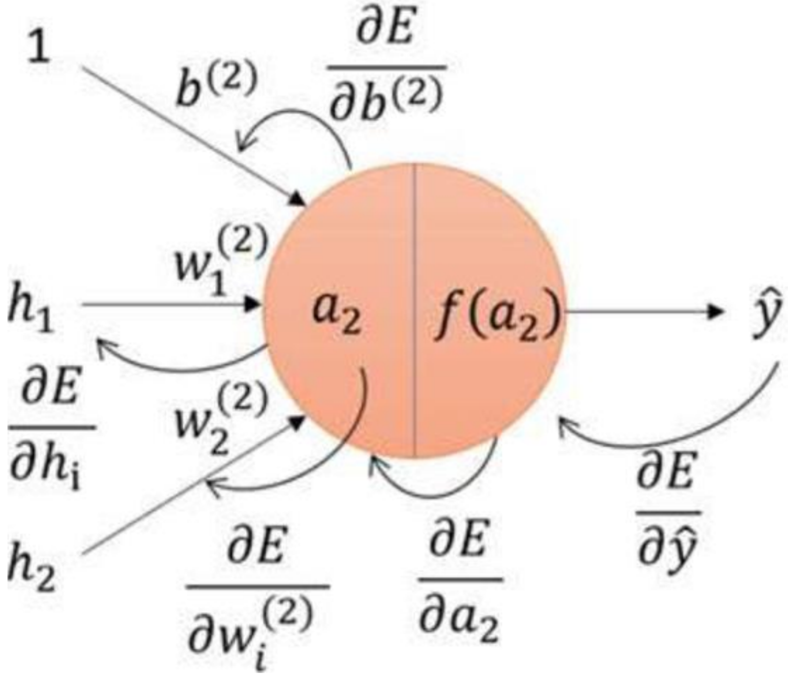
$$\nabla_{\mathbf{W}_2} E = \frac{\partial E}{\partial \mathbf{W_2}} = \frac{\partial E}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial \mathbf{a}_2} \cdot \frac{\partial \mathbf{a}_2}{\partial \mathbf{W}_2}$$

$$\nabla_{\mathbf{b}_2} E = \frac{\partial E}{\partial \mathbf{b_2}} = \frac{\partial E}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial \mathbf{a}_2} \cdot \frac{\partial \mathbf{a}_2}{\partial \mathbf{b}_2}$$

- We can also compute the error for the input to layer 2 (the post-activation output of layer 1).

$$\nabla_{\mathbf{h}_1} E = \frac{\partial E}{\partial \mathbf{h_1}} = \frac{\partial E}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial \mathbf{a}_2} \cdot \frac{\partial \mathbf{a}_2}{\partial \mathbf{h}_1}$$

Backpropagation through the Output Neuron



$$\nabla_{\hat{y}} E = \frac{\partial E}{\partial \hat{y}} = (\hat{y} - y)$$

$$\frac{\partial E}{\partial a_2} = \frac{\partial E}{\partial \hat{y}} f'(a_2)$$

$$\frac{\partial E}{\partial w_i^{(2)}} = \frac{\partial E}{\partial a_2} h_i$$

$$\frac{\partial E}{\partial b^{(2)}} = \frac{\partial E}{\partial a_2}$$

$$\frac{\partial E}{\partial h_i} = \frac{\partial E}{\partial a_2} w_i^{(2)}$$

Parameter Update

- After obtaining the gradients with respect to all learnable parameters in the network, we can complete a single SGD step, updating the parameters for each layer according to the learning rate α.

$$\theta = \theta - \alpha \nabla_{\theta} E$$

- The value of is particularly vital in SGD and affects the speed of convergence, the quality of convergence, and even the ability for the network to converge at all.
- Too small of a learning rate and the network converges very slowly and can potentially get stuck in local minima near the random weight initialization.
- If the learning rate is too large, the weights may grow too quickly, becoming unstable and failing to converge at all.
- The selection of the learning rate depends on a combination of factors such as network depth and normalization method.