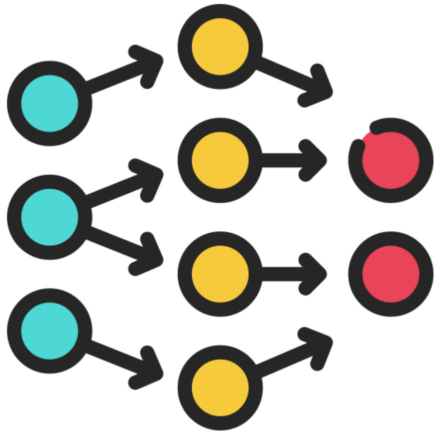
Artificial Neural Networks (ANN)

And maybe deep learning… and maybe a lot of stuff more



Artificial Neural Networks (ANN)

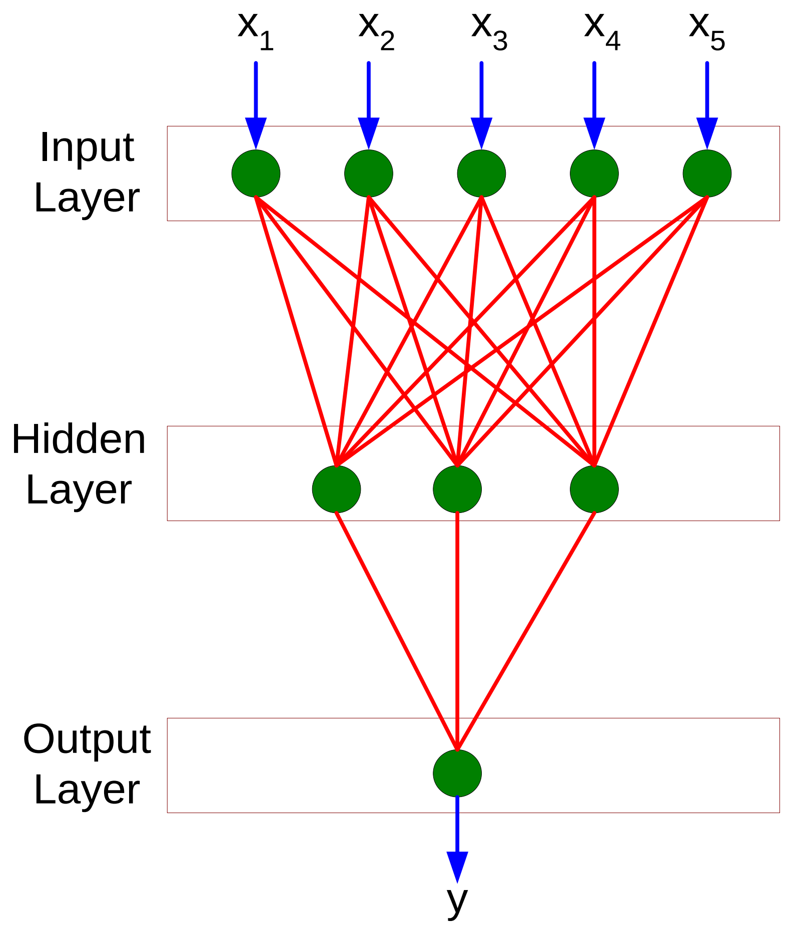
Artificial neural networks (ANN) are powerful classification models that are able to learn highly complex and nonlinear decision boundaries

Basic Idea: A complex non-linear function can be learned as a composition of simple processing units

ANN is a collection of simple processing units (**nodes**) that are connected by directed links (**edges**)

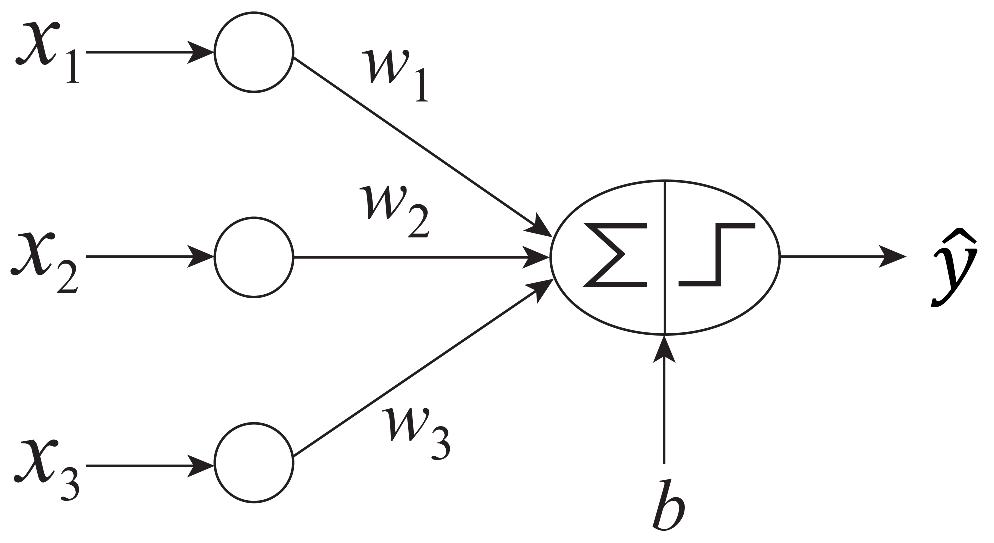
- Every **node** receives signals from incoming **edges**, performs computations, and transmits signals to its outgoing **edge**
- Weight of an **edge** determines the strength of connection between the nodes

ANN models provide a natural way of representing features at multiple levels of abstraction, where complex features are seen as compositions of simpler features



Basic Architecture of Perceptron

The fundamental element of an ANN id the perceptron



$$\boldsymbol{x} = \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} \qquad \boldsymbol{w} = \begin{bmatrix} w_1 \\ w_2 \\ w_3 \end{bmatrix}$$

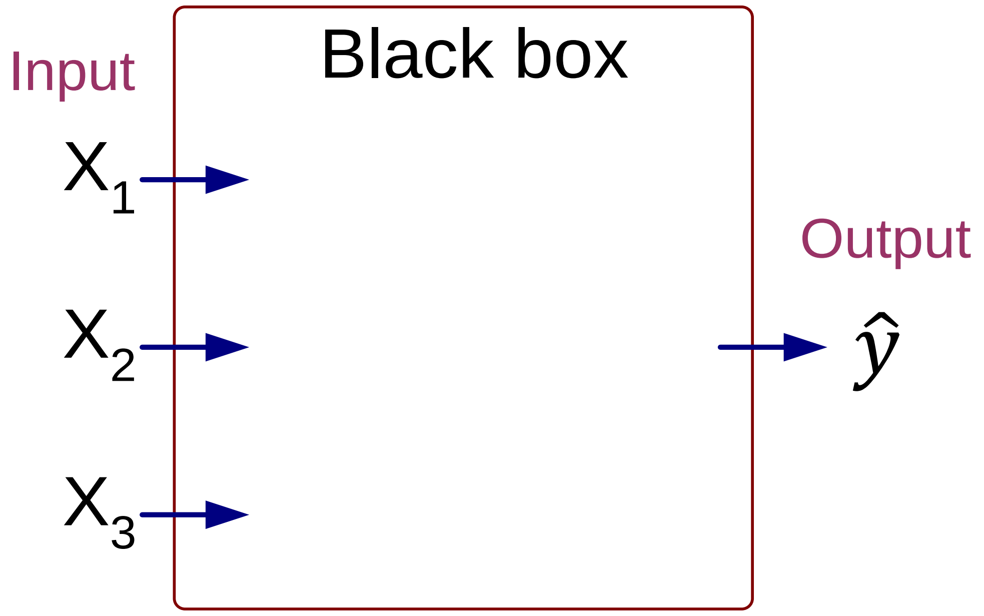
$$\boldsymbol{y} = \boldsymbol{w}^{\boldsymbol{T}} \boldsymbol{x} + b$$

Activation Function

$$\hat{y} = \begin{cases} 1 & y > 0 \\ -1 & y < 0 \end{cases}$$

Perceptron Example

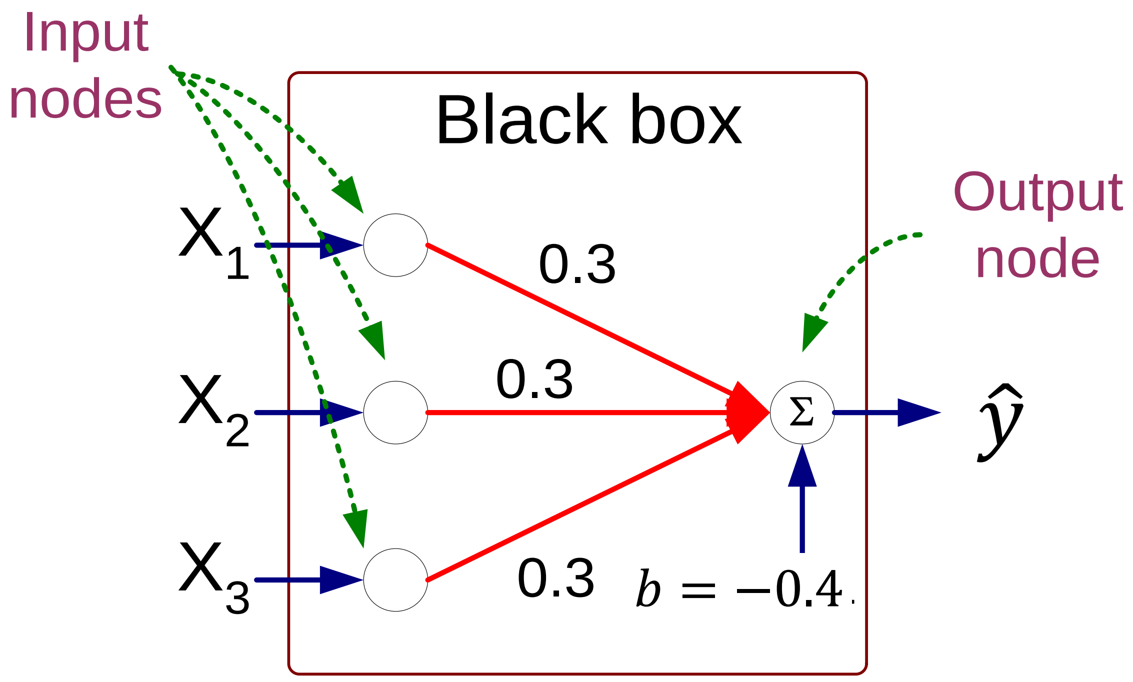
X_1	X_2	X_3	Y
1	0	0	-1
1	0	1	1
1	1	0	1
1	1	1	1
0	0	1	-1
0	1	0	-1
0	1	1	1
0	0	0	-1



Output y is 1 if at least two of the three inputs are equal to 1.

Perceptron Example

X_1	X_2	X_3	Y
1	0	0	-1
1	0	1	1
1	1	0	1
1	1	1	1
0	0	1	-1
0	1	0	-1
0	1	1	1
0	0	0	-1

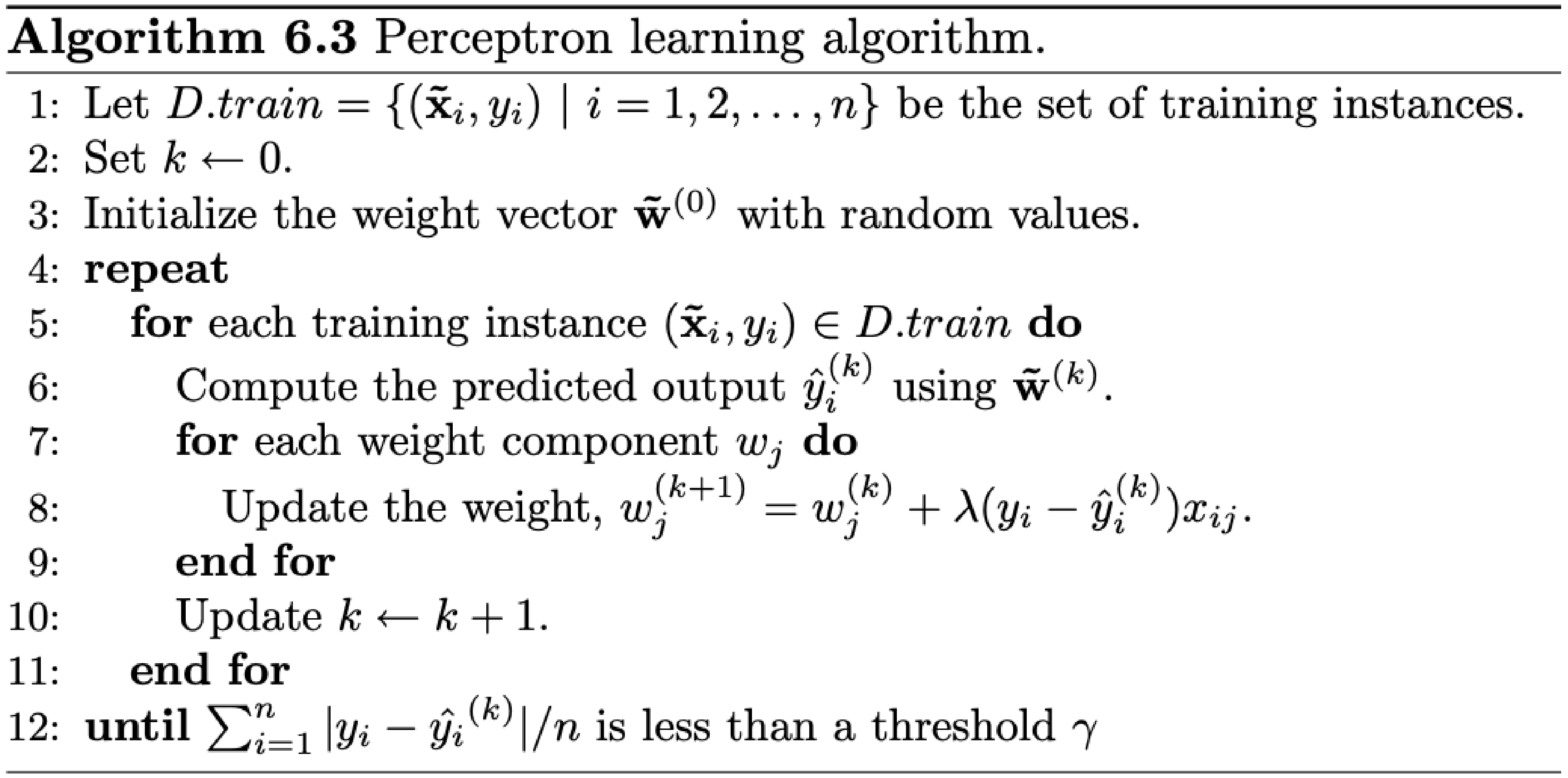


$$\boldsymbol{w} = \begin{bmatrix} w_1 \\ w_2 \\ w_3 \end{bmatrix} = \begin{bmatrix} 0.3 \\ 0.3 \\ 0.3 \end{bmatrix} \quad b = 0.4$$

$$y = \boldsymbol{w}^{\boldsymbol{T}}\boldsymbol{x} + b$$
$$y = 0.3\, x_1 + 0.3\, x_2 + 0.3\, x_3 - 0.4$$

$$\hat{y} = sign(y)$$

Perceptron Learning Rule



Algorithm 6.3 Perceptron learning algorithm.

```
1: Let $D.train = \{(\tilde{\mathbf{x}}_i, y_i) \mid i = 1, 2, \ldots, n\}$ be the set of training instances.
2: Set $k \leftarrow 0$.
3: Initialize the weight vector $\tilde{\mathbf{w}}^{(0)}$ with random values.
4: repeat
5:     for each training instance $(\tilde{\mathbf{x}}_i, y_i) \in D.train$ do
6:         Compute the predicted output $\hat{y}_i^{(k)}$ using $\tilde{\mathbf{w}}^{(k)}$.
7:         for each weight component $w_j$ do
8:             Update the weight, $w_j^{(k+1)} = w_j^{(k)} + \lambda(y_i - \hat{y}_i^{(k)})x_{ij}$.
9:         end for
10:        Update $k \leftarrow k + 1$.
11:    end for
12: until $\sum_{i=1}^{n} |y_i - \hat{y}_i{}^{(k)}|/n$ is less than a threshold $\gamma$
```

$$D.train = \{\boldsymbol{x_1}, \boldsymbol{x_2}, \ldots, \boldsymbol{x_n}\}$$

$$\boldsymbol{x_i} = \begin{bmatrix} x_{i1} \\ x_{i2} \\ . \\ x_{ik} \end{bmatrix} \qquad \boldsymbol{w} = \begin{bmatrix} w_1 \\ w_2 \\ . \\ w_k \end{bmatrix}$$

$$k = epoch\ number$$

Perceptron Learning Rule

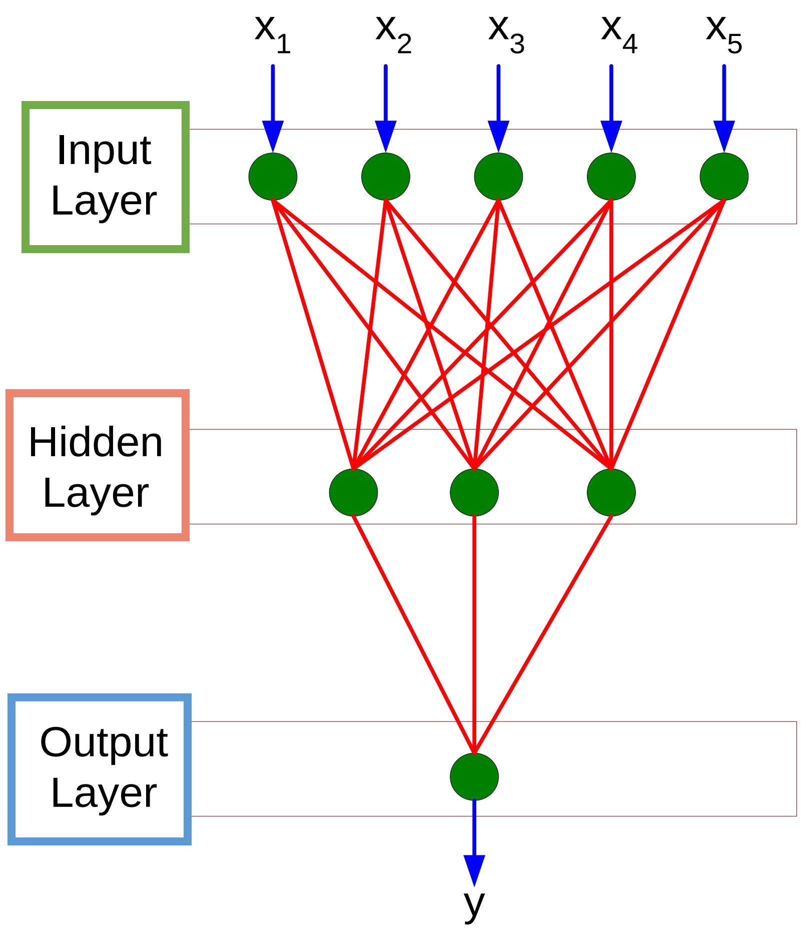
Weight update formula:

$$w_j^{(k+1)} = w_j^{(k)} + \lambda\big(y_i - \hat{y}_i^{(k)}\big)x_{ij}$$

Intuition:

- Update weight based on error: e = $(y_i - \hat{y}_i)$
  - If y = $\hat{y}$, e=0: no update needed
  - If y > $\hat{y}$, e=2: weight must be increased (assuming x_{ij} is positive) so that $\hat{y}$ will increase
  - If y < $\hat{y}$, e=-2: weight must be decreased (assuming x_{ij} is positive) so that $\hat{y}$ will decrease

Multi-layer Neural Network



The *first layer* of the network, called the **input layer**, is used for representing inputs from attributes

These inputs are fed into intermediary layers known as **hidden layers**, which are made up of processing units known as hidden nodes

- Every hidden node operates on signals received from the input nodes or hidden nodes at the preceding layer, and produces an activation value that is transmitted to the next layer

The **output layer** processes the activation values from its preceding layer to produce predictions of output variables.

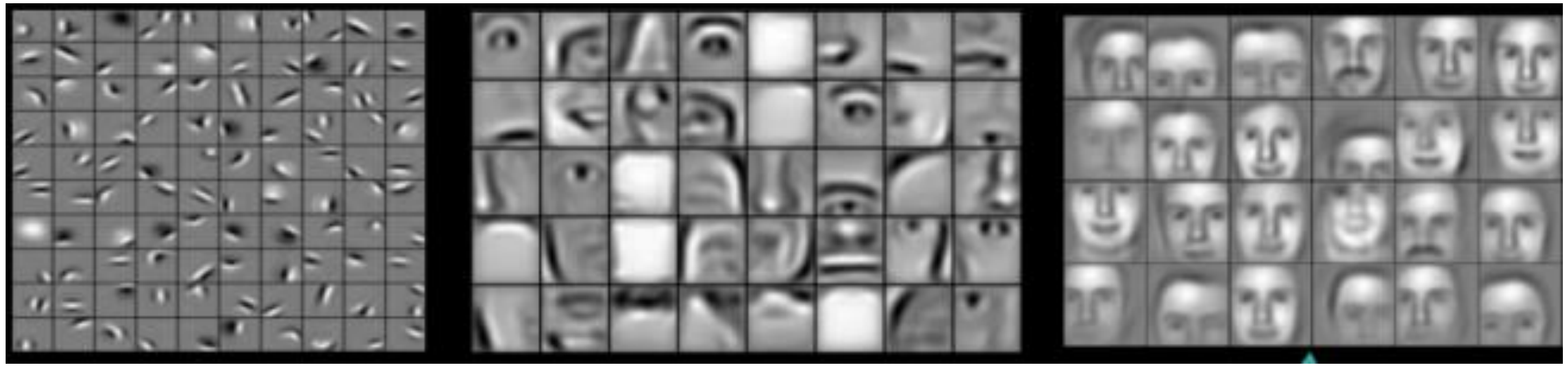
- For binary classification, the output layer contains a single node representing the binary class label

Why Multiple Hidden Layers?

Activations at hidden layers can be viewed as features extracted as functions of inputs

Every hidden layer represents a level of abstraction

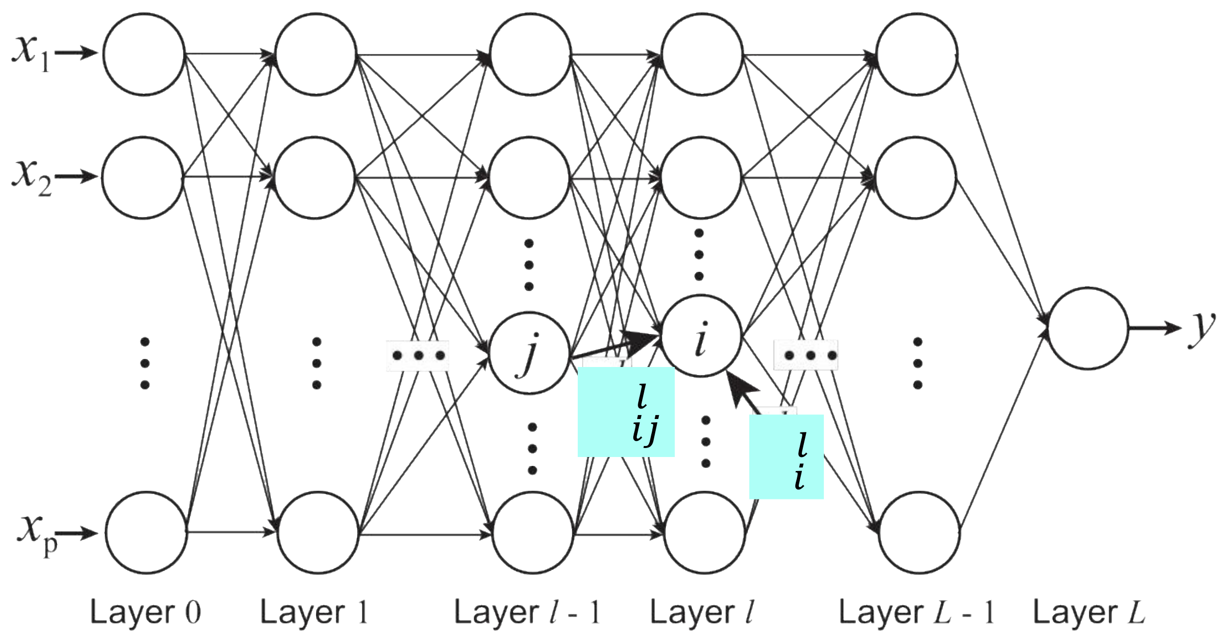
- *Complex features are compositions of simpler features*



Number of layers is known as **depth** of ANN

- *Deeper networks express complex hierarchy of features*

Multi-Layer Network Architecture



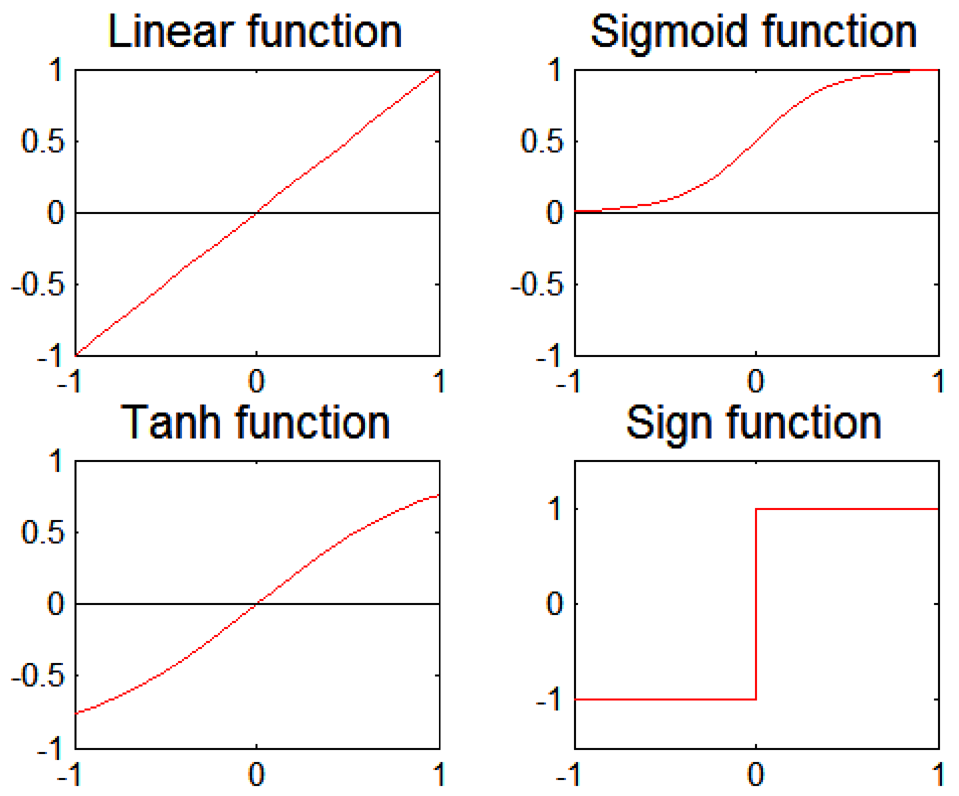
$$a_i^l = f(z_i^l) = f\Big(\underbrace{\sum_j w_{ij}^l a_j^{l-1} + b_i^l}_{\text{Linear Predictor}}\Big)$$

Activation value
at node i at layer l

Activation Function

Linear Predictor

Activation Functions



$$a_i^l = f(z_i^l) = f\Big(\sum_j w_{ij}^l a_j^{l-1} + b_i^l\Big)$$

$$a_i^l = \sigma(z_i^l) = \frac{1}{1+e^{-z_i^l}}.$$

$$\frac{\partial a_i^l}{\partial z_i^l} = \frac{\partial\ \sigma(z_i^l)}{\partial z_i^l} = a_i^l(1-a_i^l)$$

Learning Multi-layer Neural Network

Can we apply perceptron learning rule to each node, including hidden nodes?

- Perceptron learning rule computes error term e = y - $\hat{y}$ and updates weights accordingly
    - Problem: how to determine the true value of y for hidden nodes?
- Approximate error in hidden nodes by error in the output nodes
    - Problem:
        - Not clear how adjustment in the hidden nodes affect overall error
        - No guarantee of convergence to optimal solution

Gradient Descent

Loss Function to measure errors across all training points

$$E(\mathbf{w}, \mathbf{b}) = \sum_{k=1}^{n} \mathrm{Loss}\ (y_k,\ \hat{y}_k)$$

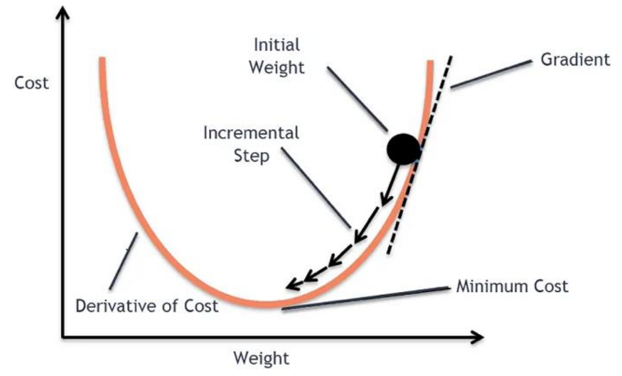
Example : Squared Loss

$$\mathrm{Loss}\ (y_k,\ \hat{y}_k) = (y_k - \hat{y}_k)^2.$$

Gradient descent: Update parameters in the direction of "maximum descent" in the loss function across all points

$$w_{ij}^l \longleftarrow w_{ij}^l - \lambda \frac{\partial E}{\partial w_{ij}^l},$$

$$b_i^l \longleftarrow b_i^l - \lambda \frac{\partial E}{\partial b_i^l},$$



- Stochastic gradient descent (SGD): update the weight for every instance (minibatch SGD: update over min-batches of instances)

λ: learning rate

Backpropagation Algorithm

The backpropagation equations provide us with a way of computing the gradient of the cost function. Let's explicitly write this out in the form of an algorithm:

- **Input** x: Set the corresponding activation a1 for the input layer.
- **Feedforward**: For each $l = 2,3, \ldots, L$ compute $z_l = w_l\, a_{l-1} + b\_l$ and $a_l = \sigma(z_l)$.
- **Output error** δ^L
- **Backpropagate the error**: For each $l = 2,3, \ldots, L$ compute δ^l
- **Output**: The gradient of the cost function is given by $\frac{\partial C}{w_{ljk}} = a_{l-1}\delta_j^l$ $\frac{\partial C}{b_{ljk}} = \delta_j^l$

Design Issues in ANN

Number of nodes in input layer

- One input node per **binary/continuous** attribute
- k or $\log_2$ k nodes for each **categorical** attribute with k values

Number of nodes in output layer

- One output for binary class problem
- k or $\log_2$ k nodes for k-class problem

Number of hidden layers and nodes per layer

Initial weights and biases

Learning rate, max. number of epochs, mini-batch size for mini-batch SGD, …

Characteristics of ANN

Multilayer ANN are universal approximators but could suffer from overfitting if the network is too large

- Naturally represents a hierarchy of features at multiple levels of abstractions

Model building is compute intensive, but testing is fast

Can handle redundant and irrelevant attributes because weights are automatically learned for all attributes

Sensitive to noise in training data

- This issue can be addressed by incorporating model complexity in the loss function

Difficult to handle missing attributes