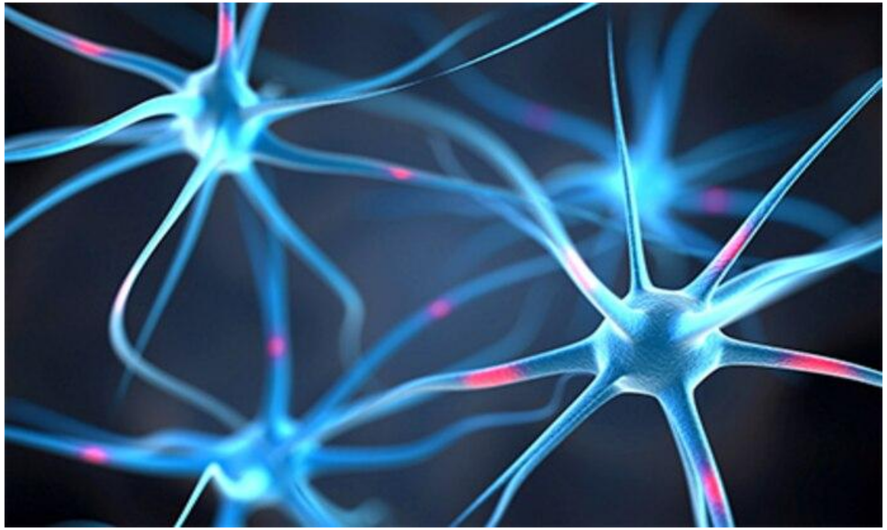
Introduction to Neural Networks

AI-NLP-ML Group
Department of Computer Science and Engineering
IIT Patna

Outline

Neural Networks

- Neural Networks are networks of interconnected neurons, for example in human brains.
- Artificial Neural Networks are highly connected to other neurons, and performs computations by combining signals from other neurons.

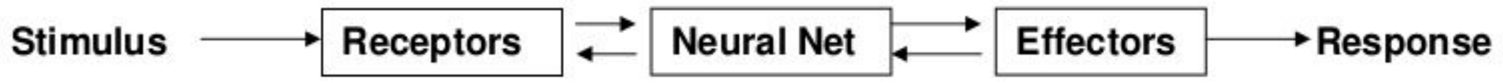


- Outputs of these computations may be transmitted to one or more other neurons.
- The neurons are connected together in a specific way to perform a particular task.

Artificial Neural Networks (High-Level Overview)

- A neural network is a function.
- It consists of basically:
  a. Neurons: which pass input values through functions and output the result.
  b. Weights: which carry values (real-number) between neurons.
- Neurons can be categorized into layers:
  a. Input Layer
  b. Hidden Layer
  c. Output Layer

Neurophysiology



(adapted from Arbib, 1987)

- The human nervous system can be divided into three stages:

a. Receptors:
   - Convert stimuli from the external environment into electrical impulses
   - Rods and Cones of eyes,
   - Pain, touch, hot and cold receptors of skin.

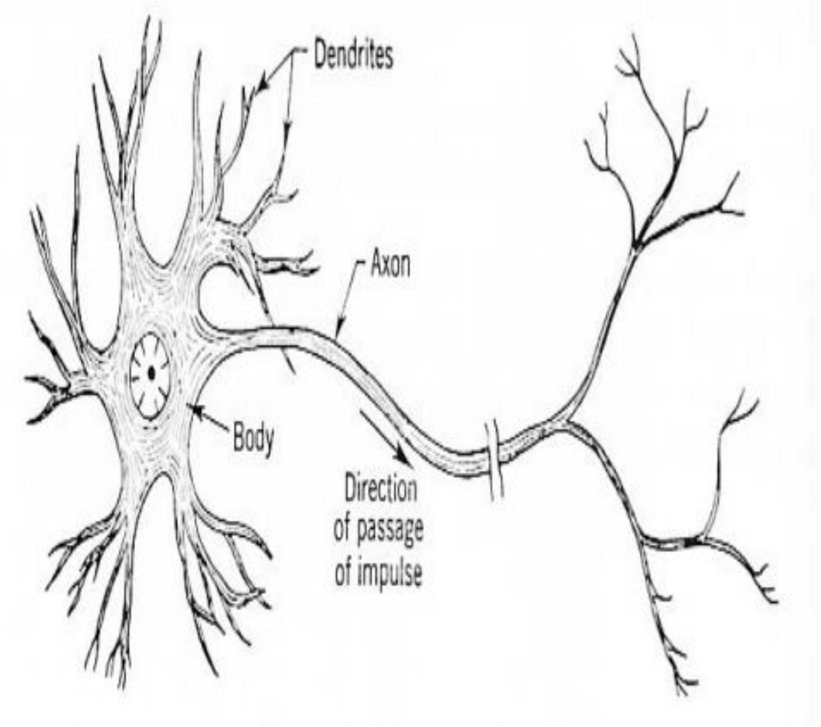
b. Neural Net:
   - Receive information, process it and make appropriate decisions.
   - Brain

c. Effectors:
   - Convert electrical impulses generated by the the neural net (brain) into responses to the external environment.
   - Muscles and glands, speech generators.

Basic Components of Biological Neurons

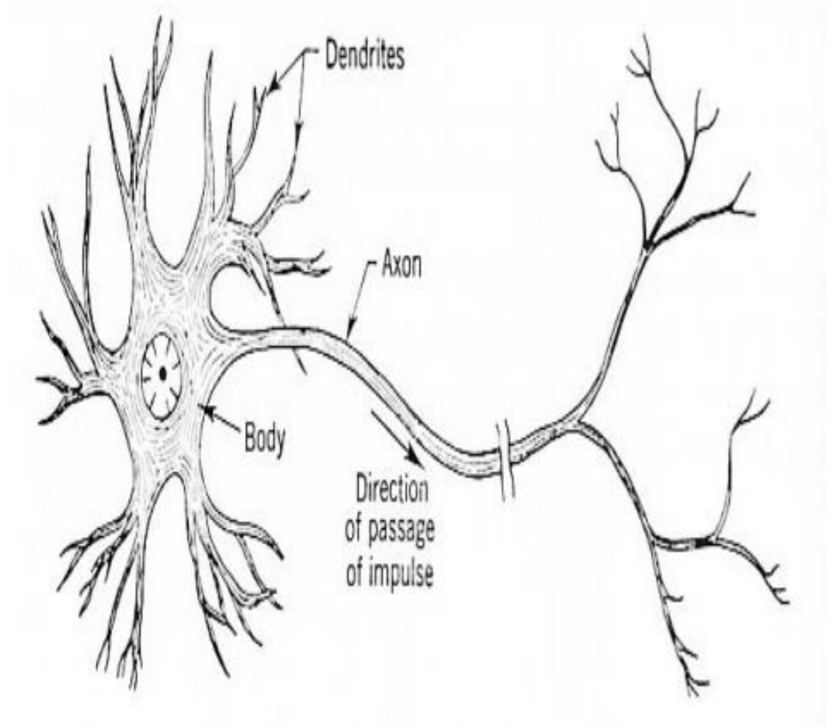
The basic components of a biological neuron are:

- Cell Body (Soma) processes the incoming activations and converts them into output activations.
- Neuron Nucleus contains the genetic material (DNA).
- Dendrites form a fine filamentary bush each fiber thinner than an axon.
- Axon: Long thin cylinder carrying impulses from soma to other cells
- Synapses: The junctions that allow signal transmission b/w the axons and dendrites.



Computation in Biological Neurons

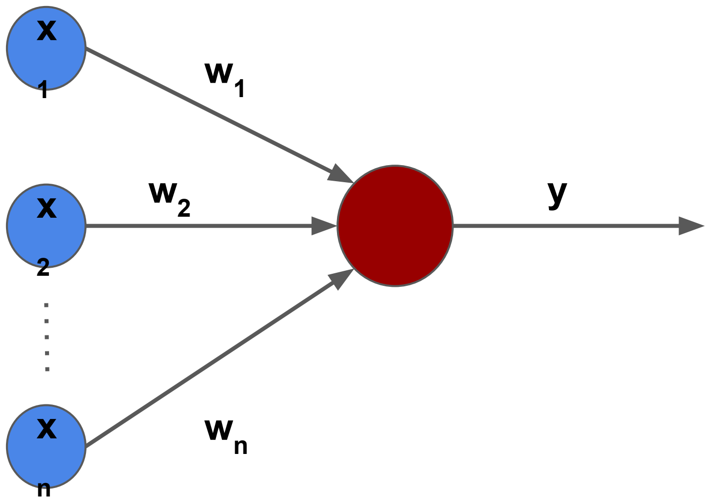
- Incoming signals from synapses are summed up at the soma.
- On crossing a threshold, the cell fires generating an action potential in the axon hillock region.



The Perceptron Model

- Motivated by the biological neuron.
- A perceptron is a computing element where inputs are associated with the weights and the cell having a threshold value.

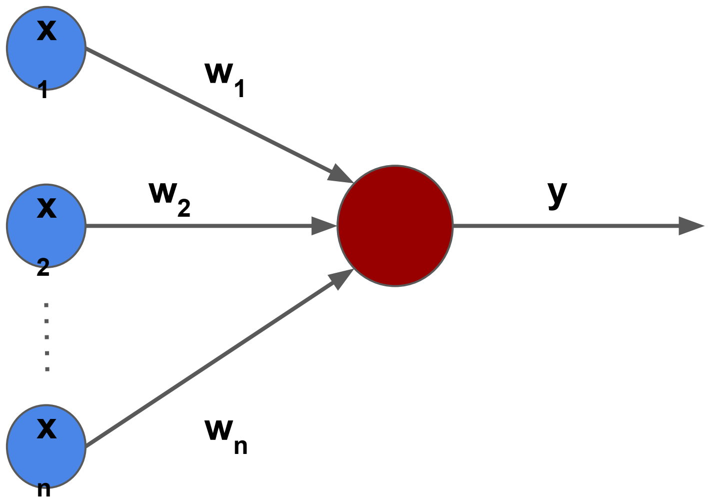
$$y = \begin{cases} 1, & \text{if } \sum w_i x_i > \text{threshold} \\ 0 & \text{otherwise} \end{cases}$$



The Perceptron Model

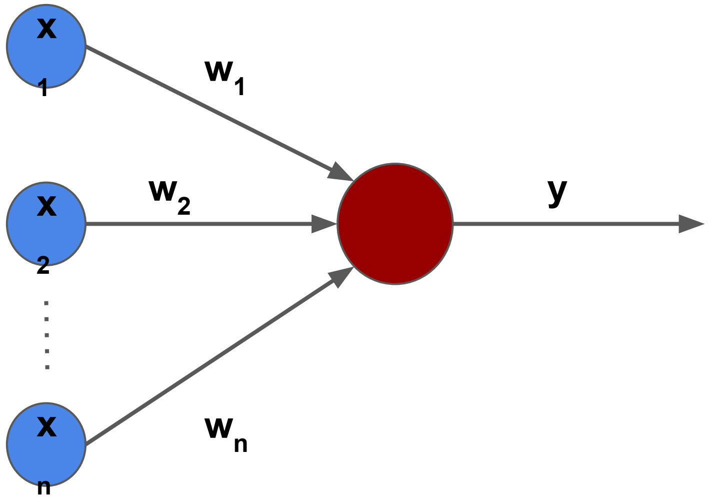
- Rewrite $\Sigma\ w_i x_i$ as $w.x$
- Replace threshold = -b
- **b:** Bias, a prior inclination towards some decision.

$$y = \begin{cases} 1, & \text{if } w.x + b > 0 \\ 0 & \text{otherwise} \end{cases}$$



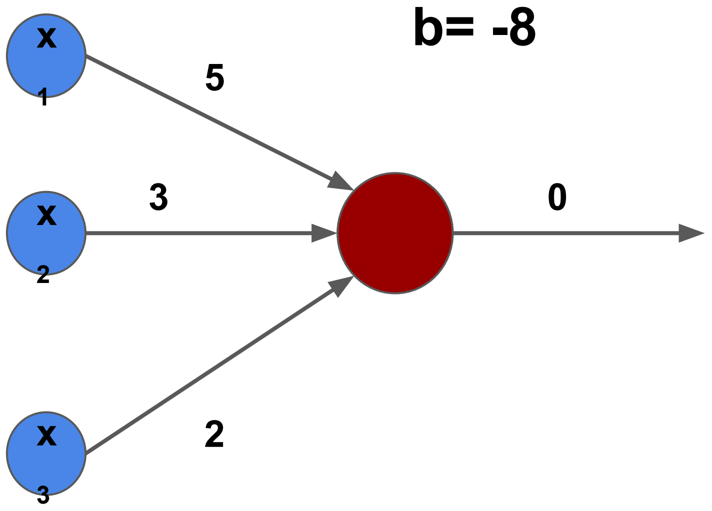
A simple decision via Perceptron

- *Whether should you go to watch movie this weekend?*
- The decision variables are:
  - Is there any extra lecture this weekend? (x_1)
  - Does your friend want to go with you? (x_2)
  - Do you have pending assignments due on the weekend? (x_3)



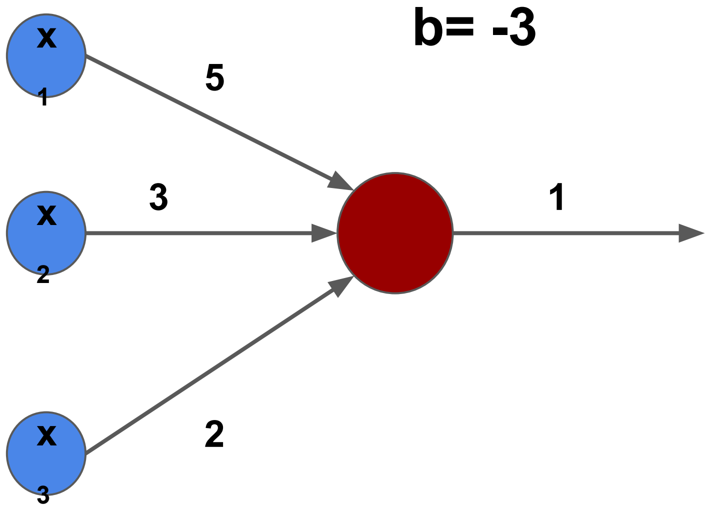
A simple decision via Perceptron

- *Whether should you go to watch movie this weekend?*
- The decision variables are:
  - Is there any extra lecture this weekend? ($x_1=1$)
  - Does your friend want to go with you? ($x_2=0$)
  - Do you have pending assignments due on the weekend? ($x_3=1$)



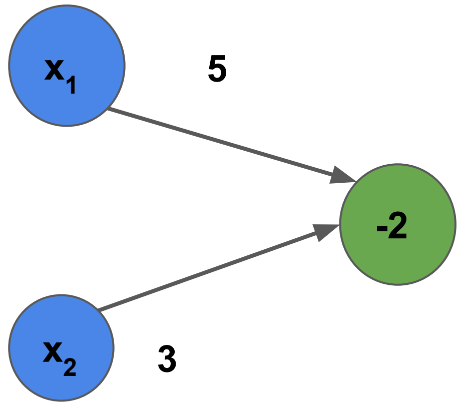
A simple decision via Perceptron

- *Whether should you go to watch movie this weekend?*
- The decision variables are:
    - Is there any extra lecture this weekend? ($x_1=1$)
    - Does your friend want to go with you? ($x_2=0$)
    - Do you have pending assignments due on the weekend? ($x_3=0$)

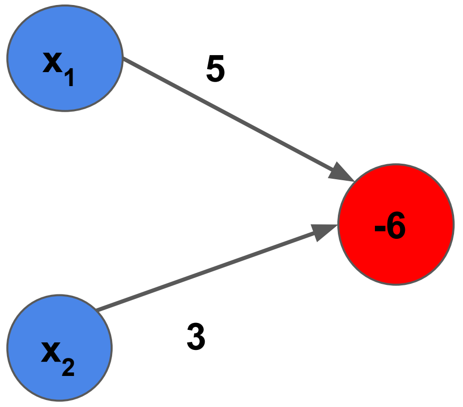


Emulating Logical Gates with Perceptron

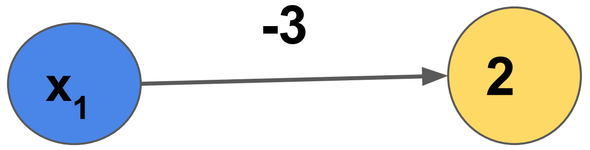
OR



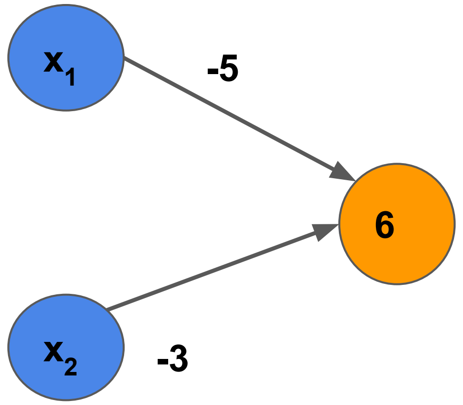
AND



NOT



NAND



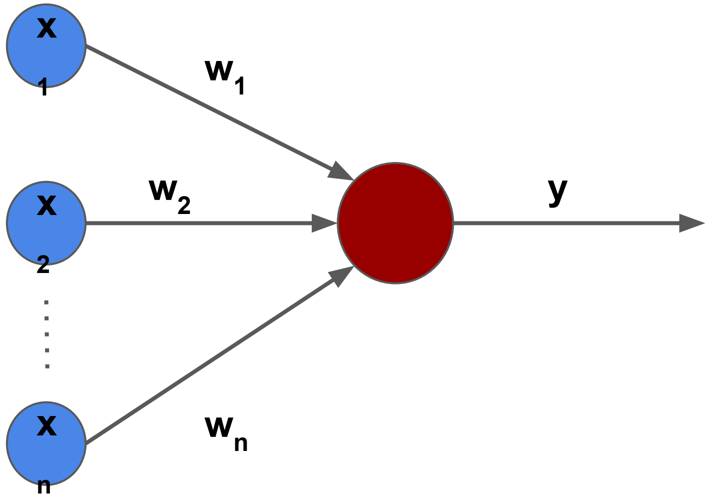
Perceptron Training

Step-1: Absorb bias ***b*** as weight.

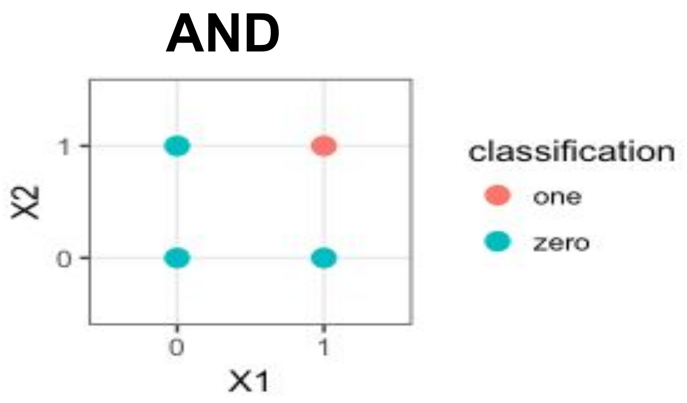
Step-2: Start with a random value of weight $\boldsymbol{w}_j$

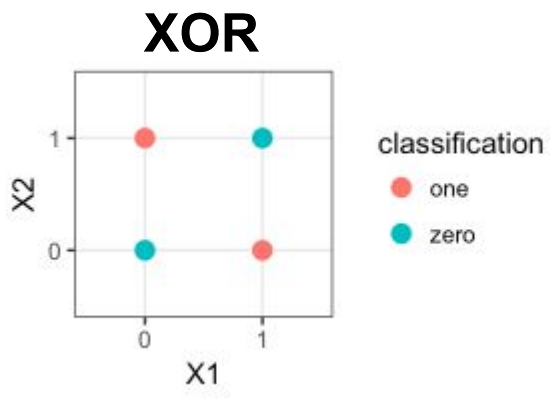
Step-3: Predict for each input $\boldsymbol{x}_i$ **:**
If the prediction is correct $\forall x$, *then*
Return ***w***
Step-4: On a mistake for given input ***x***
, update as follows:

- Mistake on positive (*y=1*), update $\boldsymbol{w}_{j+1} \leftarrow \boldsymbol{w}_j + \boldsymbol{x}$
- Mistake on negative (*y=0*), update $\boldsymbol{w}_{j+1} \leftarrow \boldsymbol{w}_j - \boldsymbol{x}$



Convergence of Perceptron Training





(source: https://jarvmiller.github.io/2017/10/14/neural-nets-pt1/)

- Whatever be the initial choice of weights and whatever be the input vector, PTA converges if the vectors are from a linearly separable function.
- If the weight repeats while training the perceptron, then the function is not linearly separable.

Activation Functions

- Activation function decide whether a neuron should be activated or not.
- It helps the network to use the useful information and suppress the irrelevant information.
- Usually a nonlinear function.
    - What if we choose a linear?
    - Linear classifier
    - Limited capacity to solve complex problems.

Activation Functions (cont'd)

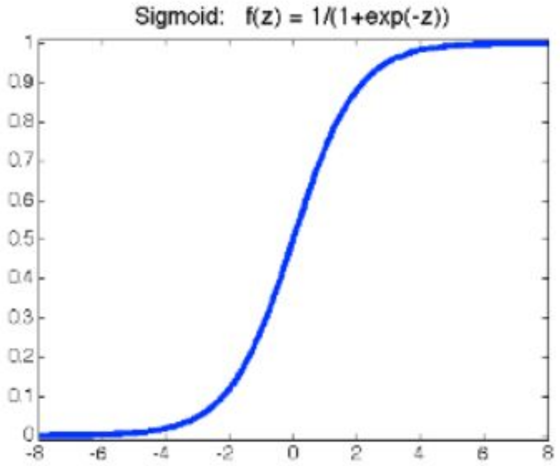
- **Sigmoid**

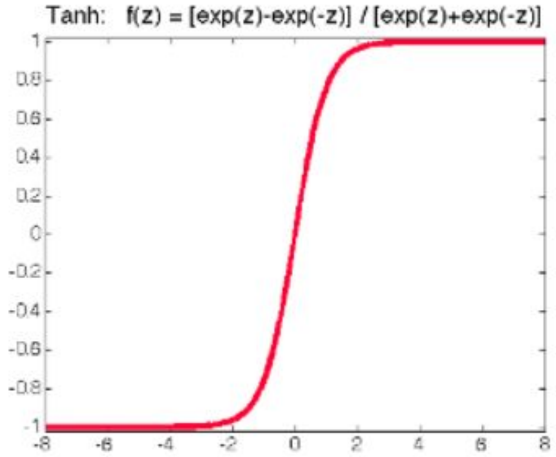
$$\sigma(z) = \frac{1}{1+\exp(-z)}$$

  - **continuously differentiable**
  - **ranges from 0-1**
  - **not symmetric around the origin**
- **Tanh**

$$\tanh(z) = \frac{\exp(z)-\exp(-z)}{\exp(z)+\exp(-z)}$$

  - **scaled version of the sigmoid**
  - **symmetric around the origin**
  - **vanishing gradient**





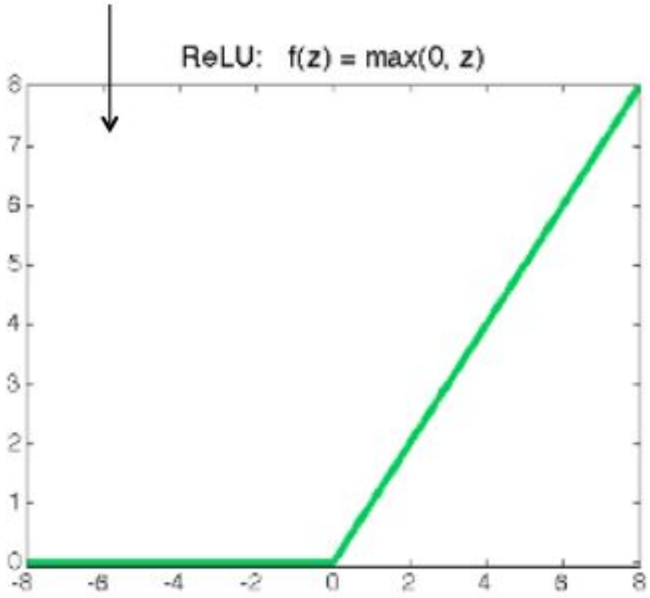
Activation Functions (cont'd)

- ReLU

$$\mathrm{ReLU}(z) = \max(0, z)$$

  - Also called piecewise linear function because rectified function is linear for half of the input domain and nonlinear for the other half.
  - trivial to implement
  - sparse representation
  - avoid the problem of vanishing gradients
  - dead neurons

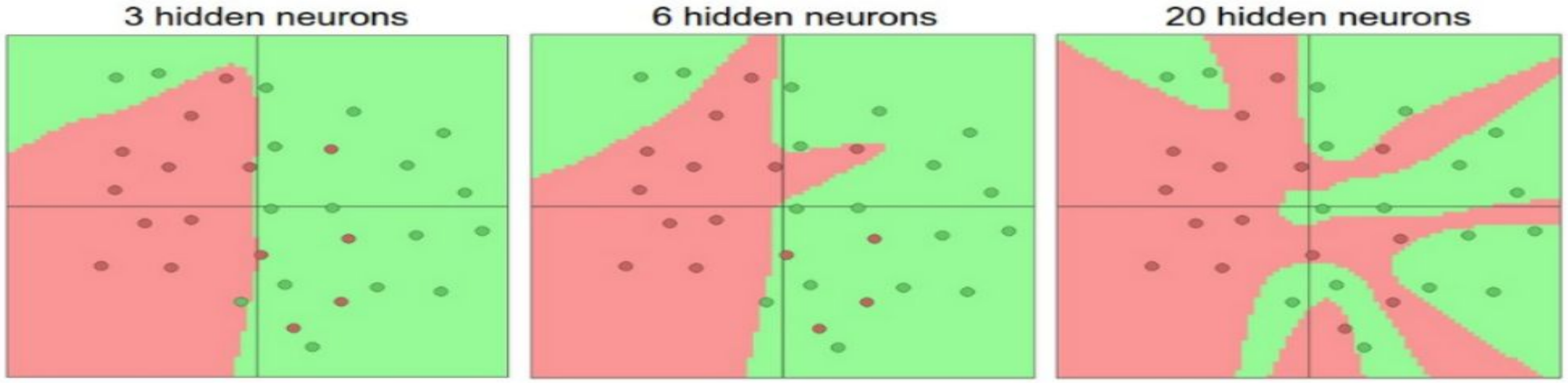
Most popular recently for deep learning



Representation Power

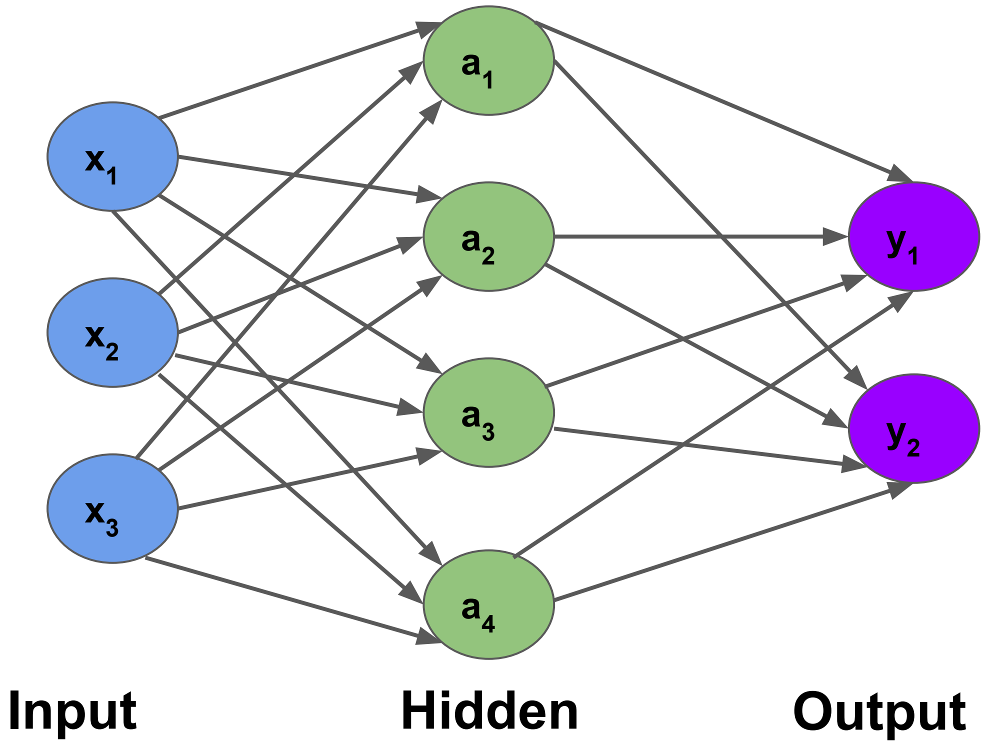
- A neural network with at least one hidden layer can approximate any function.[1]
- The representation power of network increase with more hidden units and more hidden layers.

But, "with great power comes great overfitting"



[1] Cybenko, George. "Approximation by superpositions of a sigmoidal function." *Mathematics of control, signals and systems* 2.4 (1989): 303-314.

Feed-forward Neural Network



$$a_1 = f(W_{11}x_1 + W_{12}x_2 + W_{13}x_3)$$

.

.

$$a_3 = f(W_{31}x_1 + W_{32}x_2 + W_{33}x_3)$$

$$y_1 = f(U_{11}a_1 + U_{12}a_2 + U_{13}a_3 + U_{14}a_4)$$

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

Matrix form:

$$z_1 = Wx$$

$$a = f(z_1)$$

$$z_2 = Ua$$

$$y = f(z_2)$$

where $x \in \mathbb{R}^{d_i}$, $W \in \mathbb{R}^{d_1 \times d_i}$, $a \in \mathbb{R}^{d_1}$, $U \in \mathbb{R}^{d_o \times d_1}$, $y \in \mathbb{R}^{d_o}$

Objective Function

- The function we want to minimize or maximize is called the objective function or criterion.
- When we are minimizing it, we may also call it the **cost function**, **loss function**, or error function.
- A loss function tells how good our current classifier is.
- Given a dataset:

$$\mathbf{x}^{(i)} = \{x_1^{(i)}, x_2^{(i)}, \ldots, x_m^{(i)}\} \in \mathbb{R}^m$$

y^i, The loss function can be written as:

$$\mathcal{L}(\theta) = \frac{1}{n}\sum_{i=1}^{n} L\big(y^{(i)}, f(\mathbf{x}^{(i)}\theta)\big)$$

Objective Function (cont'd)

- **Mean Squared Error:**
  - Mean Squared Error (MSE), or quadratic, loss function is widely used in linear regression as the performance measure.
  - It measures the average of the squares of the errors—that is, the average squared difference between the estimated values and the actual value.
  - It is always non-negative, and values closer to zero are better.

$$\boxed{\mathcal{L} = \frac{1}{n} \sum_{i=1}^{n} \left(y^{(i)} - \hat{y}^{(i)}\right)^2}$$

Objective Function (cont'd)

- **Mean Absolute Error:**
  - Mean Absolute Error (MAE) is a quantity used to measure how close forecasts or predictions are to the eventual outcomes.
  - Both MSE and MAE are used in predictive modeling.
  - MSE has nice mathematical properties which makes it easier to compute the gradient.

$$\boxed{\mathcal{L} = \frac{1}{n} \sum_{i=1}^{n} \left(\left| y^{(i)} - \hat{y}^{(i)} \right| \right)}$$

Objective Function (cont’d)

- **Cross-entropy:**
  - Coss-entropy comes from the field of information theory and has the unit of “bits.”
  - The cross-entropy between a “true” distribution ***p*** and an estimated distribution ***q*** is defined as:

$$\boxed{H(p,q) = -\sum_x p(x)\log q(x)}$$

  - Cross-entropy can be re-written in terms of the entropy and Kullback-Leibler divergence between the two distributions

$$\boxed{H(p,q) = H(p) + D_{KL}(p||q)}$$

Objective Function (cont'd)

- **Cross-entropy:**
  - Assuming a ground truth probability distribution that is **1** at the right class and **0** everywhere else ***p*** *= [0,...,0,1,0,...0]* and our computed probability is ***q***
  - Kullback-Leibler divergence can be written as:

$$\begin{aligned} D_{KL}(p||q) &= \sum_x p(x) \log \frac{p(x)}{q(x)} \\ &= \sum_x p(x) \log p(x) - \sum_x p(x) \log q(x) \\ &= -H(p) + H(p,q) \end{aligned}$$

or

$$H(p,q) = H(p) + D_{KL}(p||q)$$

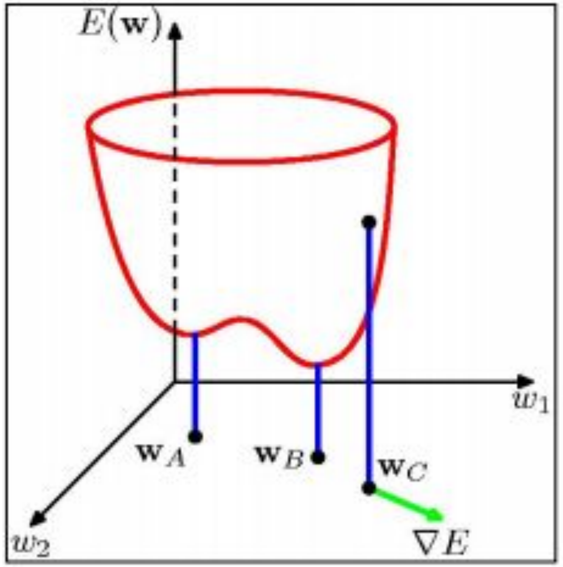
Optimization

- The goal of optimization is to find parameter (weights) that minimizes the loss function.
- How to find such weights?
  - Random Search
    - Very bad idea.
  - Random Local Search
    - Start with a random weight ***w*** and generate random perturbations **Δw** to it and if the loss at the perturbed **w+Δw** is lower, we will perform an update.
    - Computationally expensive
  - Follow the Gradient
    - No need to search for a good direction.
    - We can compute the best direction along which we should change our weight vector that is mathematically guaranteed to be the direction of the steepest descent.

Optimization (cont'd)

Find $\boldsymbol{w}$ which minimizes the chosen error function $\boldsymbol{E(w)}$

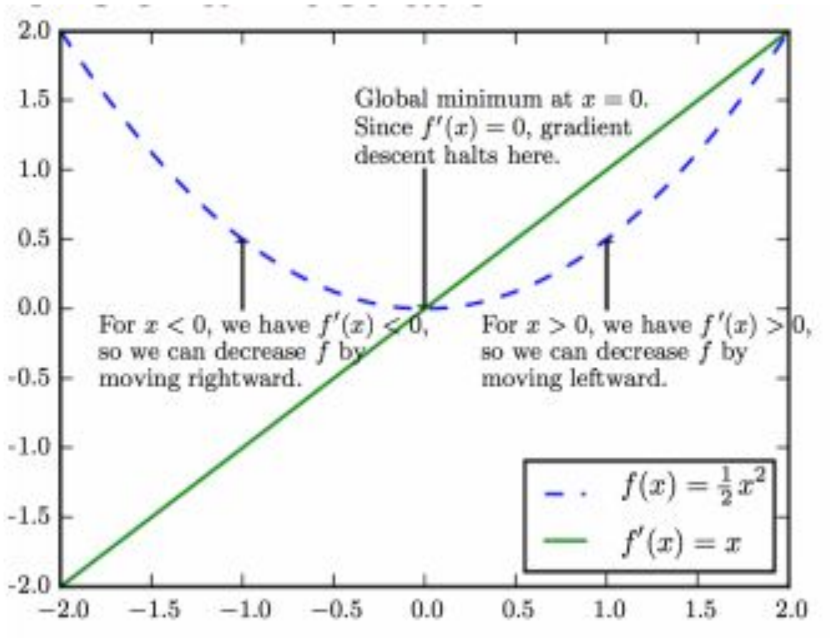
- $\boldsymbol{w_A}$: a local minimum
- $\boldsymbol{w_B}$: a global minimum
- At point $\boldsymbol{w_C}$ local gradient is given by vector $\boldsymbol{\Delta E(w)}$
- It points in direction of greatest rate of increase of $\boldsymbol{E(w)}$
- Negative gradient points to rate of greatest decrease



Optimization (cont'd)

Find $\boldsymbol{w}$ which minimizes the chosen error function $\boldsymbol{E(w)}$

- $\boldsymbol{w_A}$: a local minimum
- $\boldsymbol{w_B}$: a global minimum
- At point $\boldsymbol{w_C}$ local gradient is given by vector $\boldsymbol{\Delta E(w)}$
- It points in direction of greatest rate of increase of $\boldsymbol{E(w)}$
- Negative gradient points to rate of greatest decrease



Gradient and Hessian

- First derivative of a scalar function ***E(w)*** with respect to a vector $\mathbf{w}=[w_1,w_2]^T$ is a vector called the Gradient of ***E(w)***

$$\nabla E(\boldsymbol{w}) = \frac{d}{d\boldsymbol{w}} E(\boldsymbol{w}) = \begin{bmatrix} \frac{\partial E}{\partial w_1} \\ \frac{\partial E}{\partial w_2} \end{bmatrix}$$

If there are M elements in the vector then Gradient is a M **x** 1 vector

- Second derivative of a scalar function ***E(w)*** with respect to a vector $\mathbf{w}=[w_1,w_2]^T$ is a matrix called the Hessian of ***E(w)***

$$H = \nabla\nabla E(\boldsymbol{w}) = \frac{d^2}{d\boldsymbol{w}^2} E(\boldsymbol{w}) = \begin{bmatrix} \frac{\partial^2 E}{\partial w_1^2} & \frac{\partial^2 E}{\partial w_1 \partial w_2} \\ \frac{\partial^2 E}{\partial w_2 \partial w_1} & \frac{\partial^2 E}{\partial w_2^2} \end{bmatrix}$$

Gradient Descent Optimization

- Determine weights $\boldsymbol{w}$ from labeled set of training samples.
- Take a small step in the direction of the negative gradient

$$\boldsymbol{w}_{new} = \boldsymbol{w}_{old} - \eta \, \Delta E(\boldsymbol{w}_{old})$$

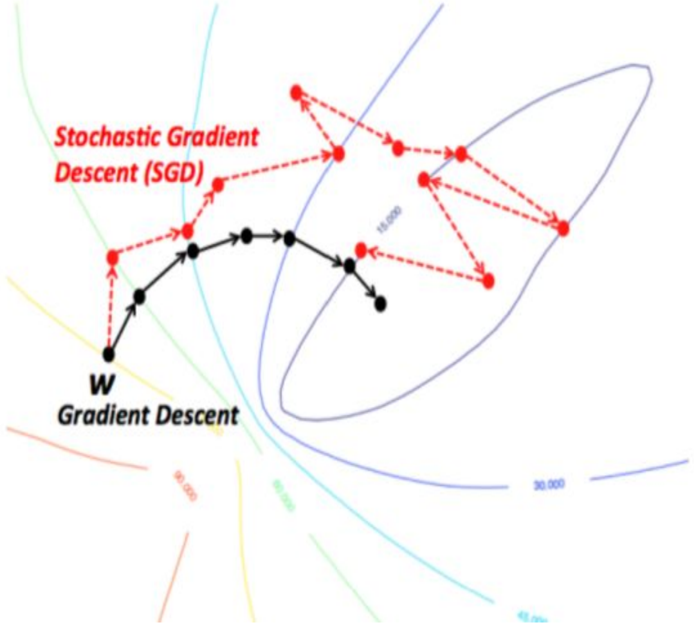
- After each update, the gradient is re-evaluated for the new weight vector and the process is repeated
- This size of steps η taken to reach the minimum or bottom is called Learning Rate.

Gradient Descent Variants

- **Batch gradient descent:**
  - Vanilla gradient descent, aka batch gradient descent, computes the gradient of the cost function w.r.t. to the parameters $\boldsymbol{w}$ for the entire training dataset.
  - $\boldsymbol{w_{new} = w_{old} - \eta\ \Delta E(w_{old})}$
  - Guaranteed to converge to global minimum for convex error surfaces and to a local minimum for non-convex surfaces.
  - Need to calculate the gradients for the whole dataset to perform just one update.
  - Very slow and is intractable for datasets that don't fit in memory.

Gradient Descent Variants

- **Stochastic gradient descent:**
  - Stochastic gradient descent (SGD) in contrast performs a parameter update for each training example, say (x_i, y_i)
  - $w_{new} = w_{old} - \eta\ \Delta E(w_{old}; x_i; y_i)$
  - Much faster (avoid redundancy as exist in Batch gradient descent)
  - While slowly decreasing the learning rate, SGD shows the same convergence behaviour as batch gradient descent.
  - It performs frequent updates with a high variance that cause the objective function to fluctuate heavily.



Gradient Descent Variants

- **Mini-batch gradient descent:**
    - Performs update for every mini-batch of n examples.
    - $w_{new} = w_{old} - \eta \, \Delta E(w_{old:}; x_{i:i+n}; y_{i:i+n})$
    - Reduces variance of updates.
    - Algorithm of choice
    - Mini-batch size is a hyperparameter. Common sizes are 50-256.

Backpropagation Algorithm

- Backpropagation algorithm is used to train artificial neural networks, it can update the weights very efficiently.
- It is a computationally efficient approach to compute the derivatives of a complex cost function.
- Goal is to use those derivatives to learn the weight coefficients for parameterizing a multi-layer artificial neural network.
- It compute the gradient of a cost function with respect to all the weights in the network, so that the gradient is fed to the gradient descent method which in turn uses it to update the weights in order to minimize the cost function.

Backpropagation Algorithm (cont'd)

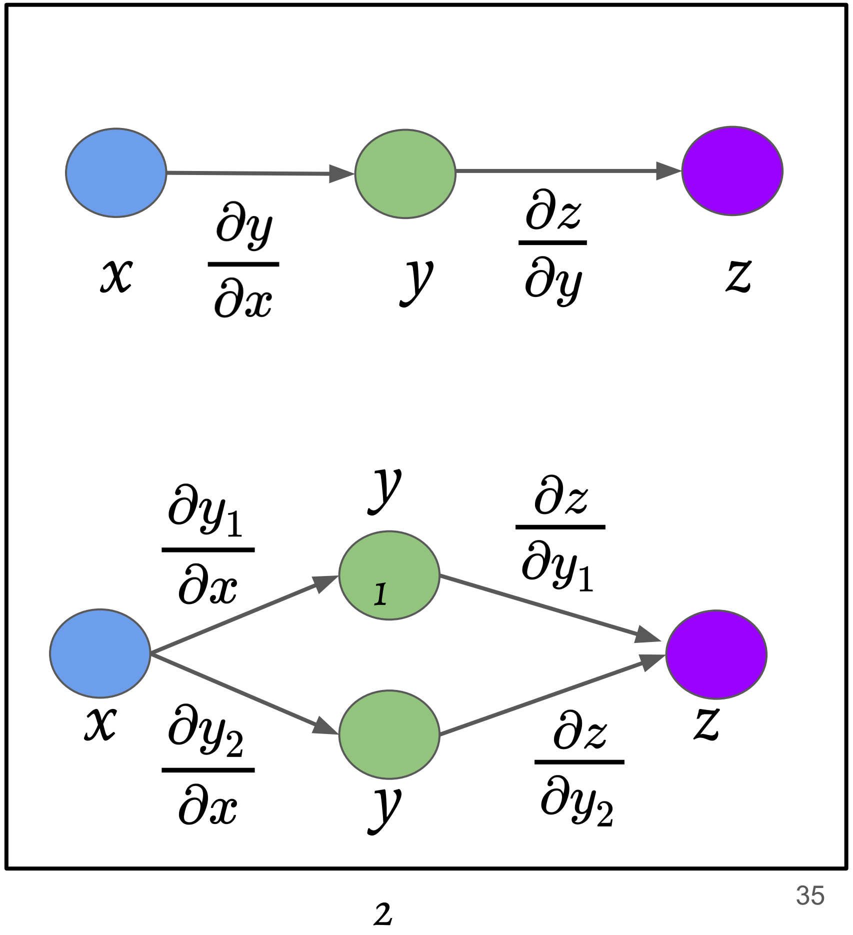
- **Chain Rule:**
  - **Single Path**

$$\frac{\partial z}{\partial x} = \frac{\partial z}{\partial y}\frac{\partial y}{\partial x}$$

  - **Multiple Path**

$$\frac{\partial z}{\partial x} = \frac{\partial z}{\partial y_1}\frac{\partial y_1}{\partial x} + \frac{\partial z}{\partial y_2}\frac{\partial y_2}{\partial x}$$

$$\frac{\partial z}{\partial x} = \sum_{t=1}^{T} \frac{\partial z}{\partial y_t}\frac{\partial y_t}{\partial x}$$



x
$\frac{\partial y}{\partial x}$
y
$\frac{\partial z}{\partial y}$
z
$\frac{\partial y_1}{\partial x}$
y
1
$\frac{\partial z}{\partial y_1}$
x
$\frac{\partial y_2}{\partial x}$
y
$\frac{\partial z}{\partial y_2}$
z
2

Backpropagation Algorithm (cont'd)

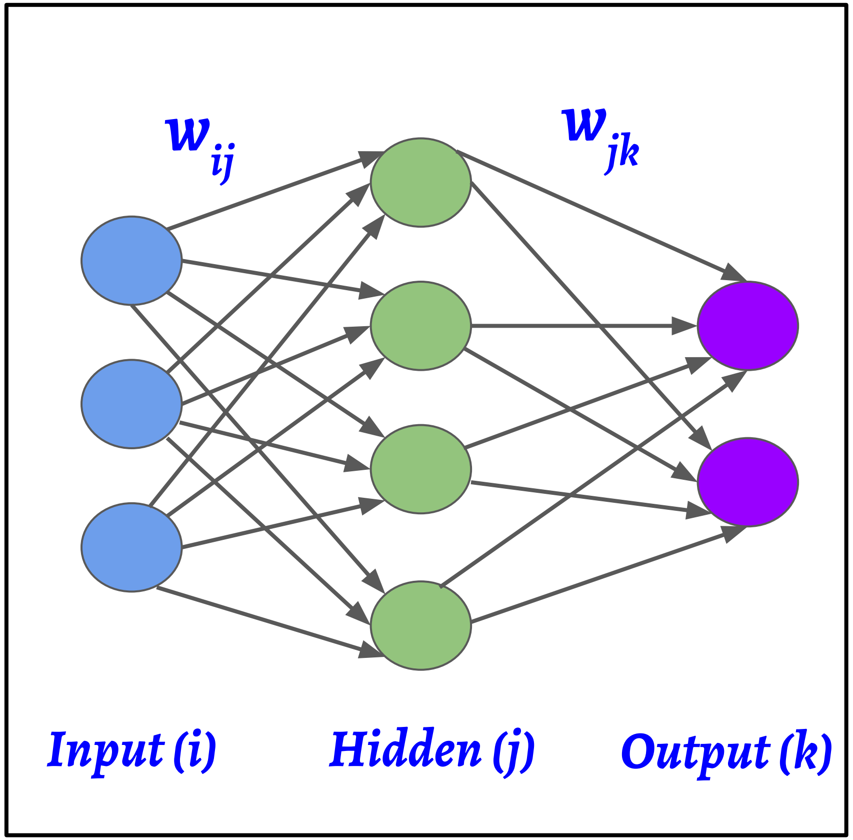
- The total error in the network for a single input is given by the following equation

$$E = \frac{1}{2}\sum_{k=1}^{K}(a_k - t_k)^2$$

$where$

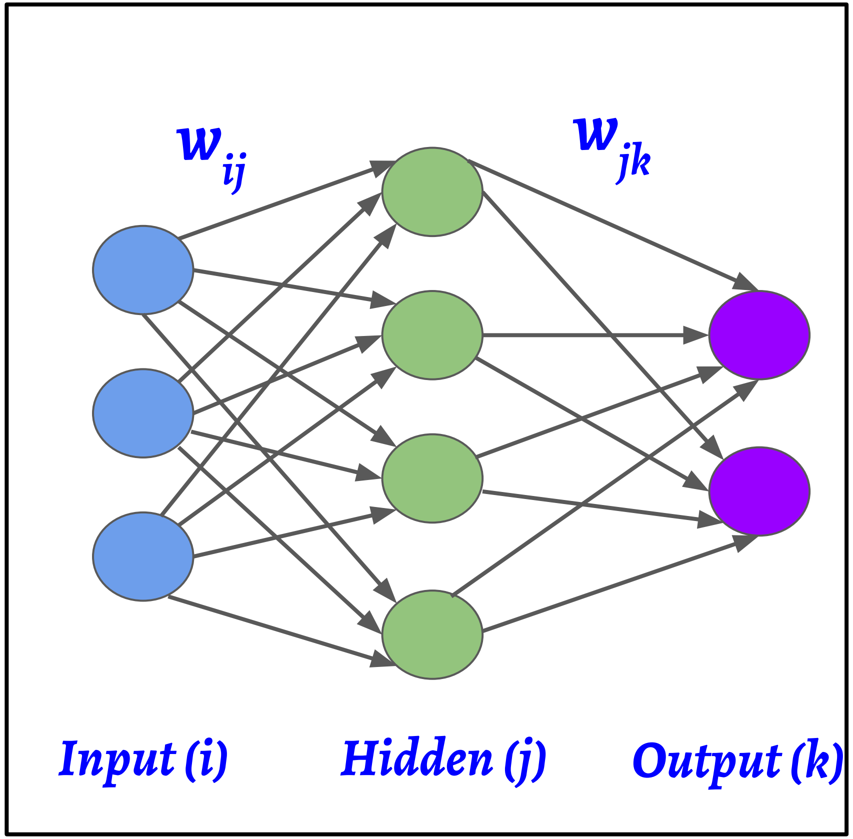
a_k : predicted output/activation of a node k

t_k : actual output of a node k



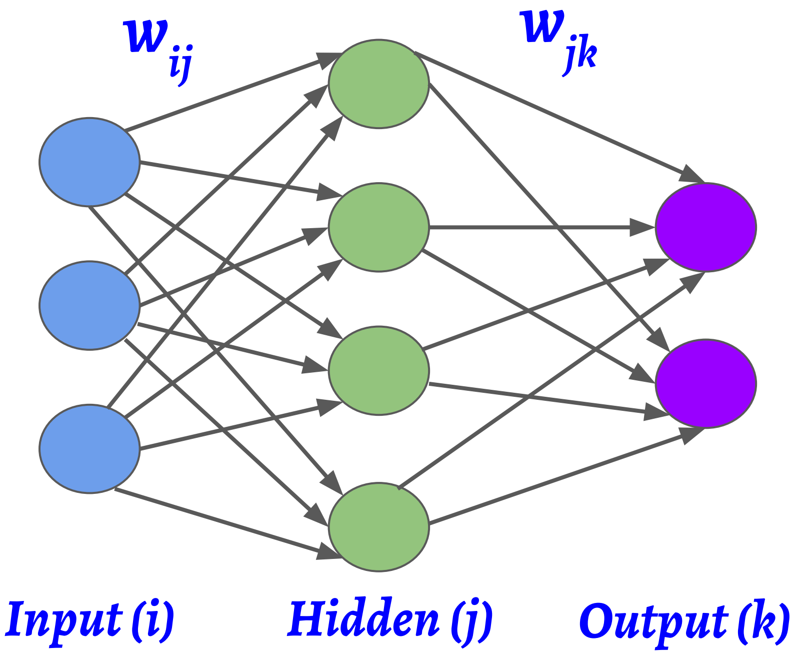
Backpropagation Algorithm (cont'd)

- There are two sets of weights in our network:
  - w_{ij} : from the input to the hidden layer.
  - w_{jk} : from the hidden to the output layer.
- We want to adjust the network's weights to reduce this overall error.
  - $\triangle W \propto -\frac{\partial E}{\partial W}$



Backpropagation Algorithm (cont'd)

- **Backpropagation – for outermost layer**
  - outermost layer parameters directly affect the value of the error function.
  - only one term of the E summation will have a non-zero derivative: the one associated with the particular weight we are considering.



$$a_k = f_k\left(\overbrace{\sum_j \underbrace{f_j\big(\underbrace{\sum_i a_i w_{ij}}_{z_j}\big)}_{a_j} w_{jk}}^{z_k}\right)$$

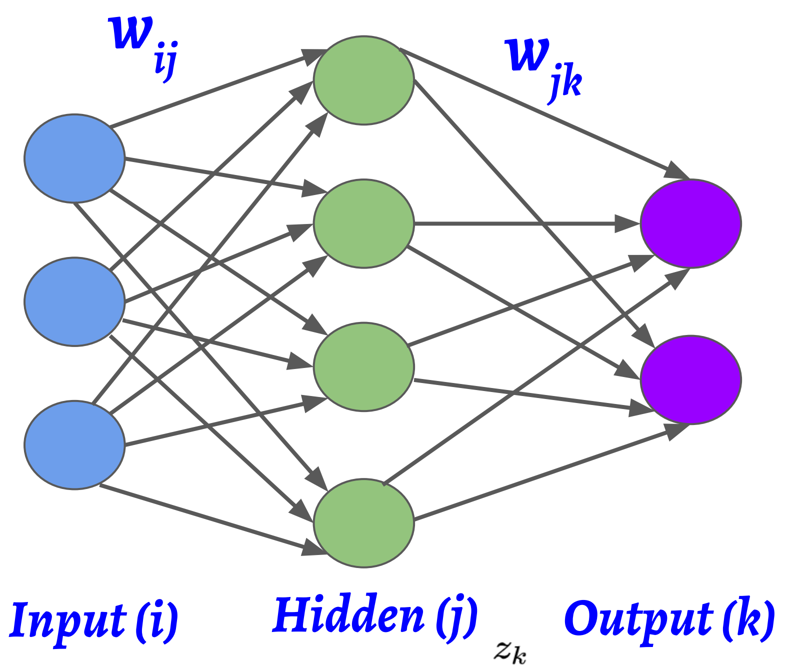
Backpropagation Algorithm (cont’d)

- **Backpropagation – for outermost layer**

$$\boxed{\frac{\partial E}{\partial w_{jk}} = \frac{\partial E}{\partial a_k}\frac{\partial a_k}{\partial z_k}\frac{\partial z_k}{\partial w_{jk}}}$$

$$\frac{\partial E}{\partial a_k} = \frac{\partial}{\partial a_k}\left(\frac{1}{2}\sum_{k\in K}(a_k - t_k)^2\right) = (a_k - t_k)$$

$$\frac{\partial a_k}{\partial z_k} = \frac{\partial}{\partial z_k}(f_k(z_k)) = f'_k(z_k)$$



$$a_k = f_k\Big(\overbrace{\textstyle\sum_j \underbrace{f_j(\underbrace{\textstyle\sum_i a_i w_{ij}}_{z_j})}_{a_j} w_{jk}}^{z_k}\Big)$$

Backpropagation Algorithm (cont'd)

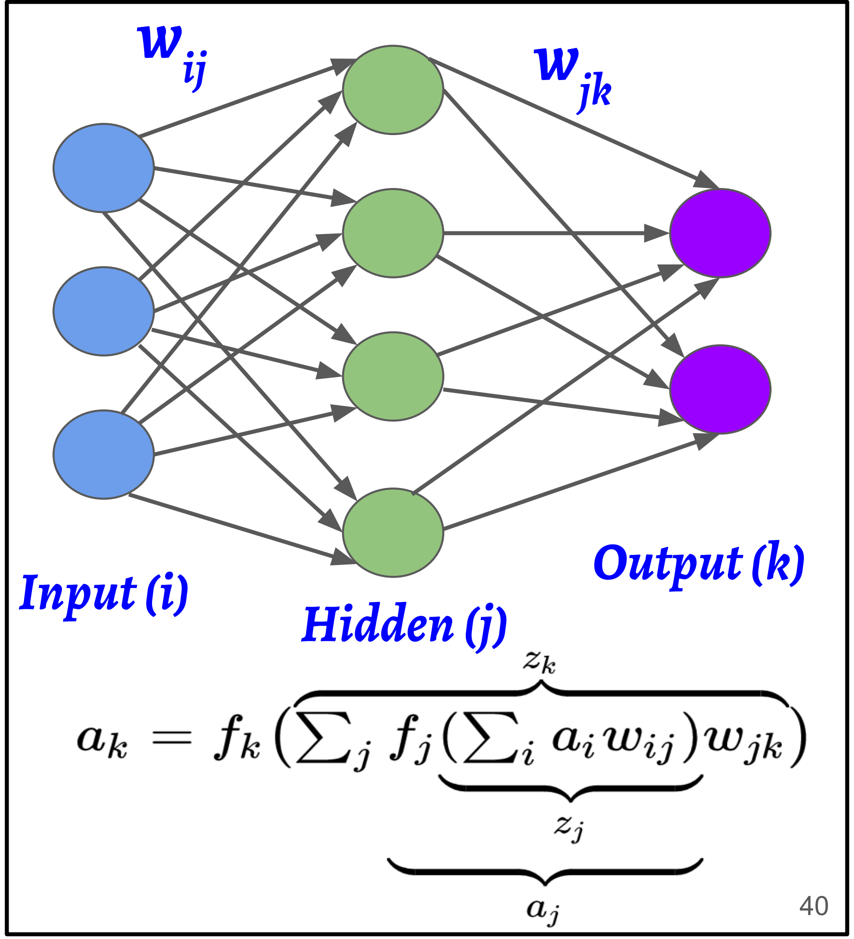
- **Backpropagation – for outermost layer**

$$\boxed{\begin{aligned}\frac{\partial z_k}{\partial w_{jk}} &= \frac{\partial}{\partial w_{jk}}\left(\sum_j a_j w_{jk}\right)\\ &= a_j\end{aligned}}$$

$$\boxed{\frac{\partial E}{\partial w_{jk}} = \underbrace{(a_k - t_k) f_k'(z_k)}_{\delta_k} a_j}$$

For sigmoid activation function

$$\boxed{\frac{\partial E}{\partial w_{jk}} = (a_k - t_k) a_k (1 - a_k) a_j}$$



$$a_k = f_k\Big(\overbrace{\textstyle\sum_j \underbrace{f_j(\underbrace{\textstyle\sum_i a_i w_{ij}}_{z_j})}_{a_j} w_{jk}}^{z_k}\Big)$$

Backpropagation Algorithm (cont'd)

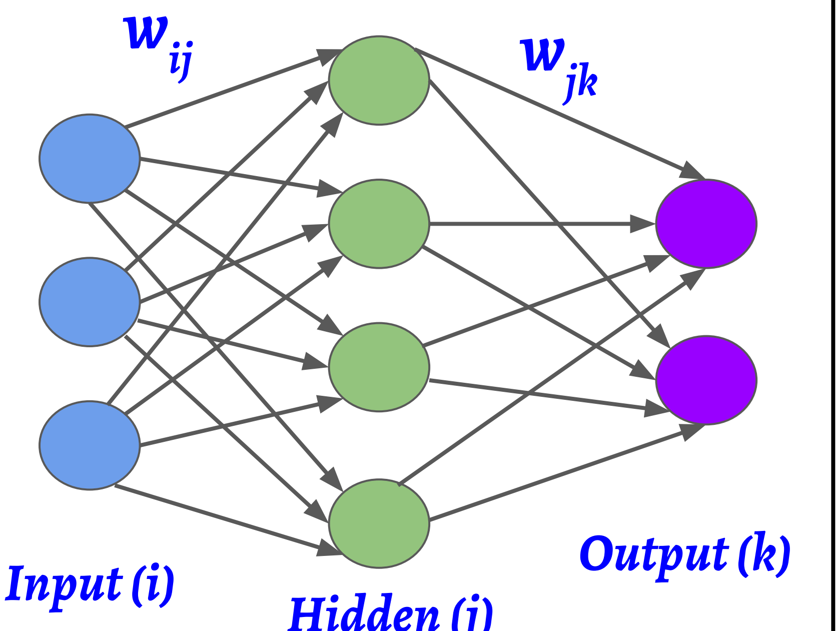
- **Backpropagation – for hidden layer**

$$\frac{\partial E}{\partial w_{ij}} = \frac{\partial}{\partial w_{ij}}\left(\frac{1}{2}\sum_{k\in K}(a_k - t_k)^2\right)$$

$$= \sum_{k\in K}(a_k - t_k)\frac{\partial}{\partial w_{ij}}a_k$$

$$= \sum_{k\in K}(a_k - t_k)\frac{\partial}{\partial w_{ij}}(f_k(z_k))$$

$$= \sum_{k\in K}(a_k - t_k)f_k'(z_k)\frac{\partial}{\partial w_{ij}}z_k$$

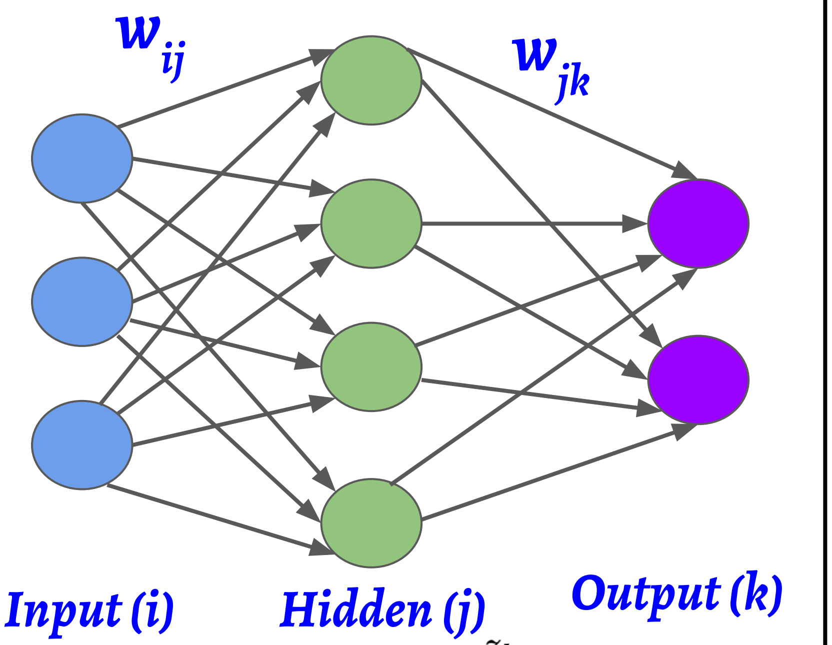


$$a_k = f_k\Big(\overbrace{\textstyle\sum_j \underbrace{f_j(\underbrace{\textstyle\sum_i a_i w_{ij}}_{z_j})}_{a_j} w_{jk}}^{z_k}\Big)$$

Backpropagation Algorithm (cont'd)

- **Backpropagation – for hidden layer**

$$\begin{aligned}
\frac{\partial z_k}{\partial w_{ij}} &= \frac{\partial z_k}{\partial a_j}\frac{\partial a_j}{\partial w_{ij}} \\
&= \frac{\partial}{\partial a_j} a_j w_{jk} \frac{\partial a_j}{\partial w_{ij}} \\
&= w_{jk} \frac{\partial a_j}{\partial w_{ij}} \\
&= w_{jk} \frac{\partial f_j(z_j)}{\partial w_{ij}} \\
&= w_{jk} f_j'(z_j) \frac{\partial z_j}{\partial w_{ij}} \\
&= w_{jk} f_j'(z_j) \frac{\partial}{\partial w_{ij}} \left(\textstyle\sum_i a_i w_{ij}\right) \\
&= w_{jk} f_j'(z_j) a_i
\end{aligned}$$

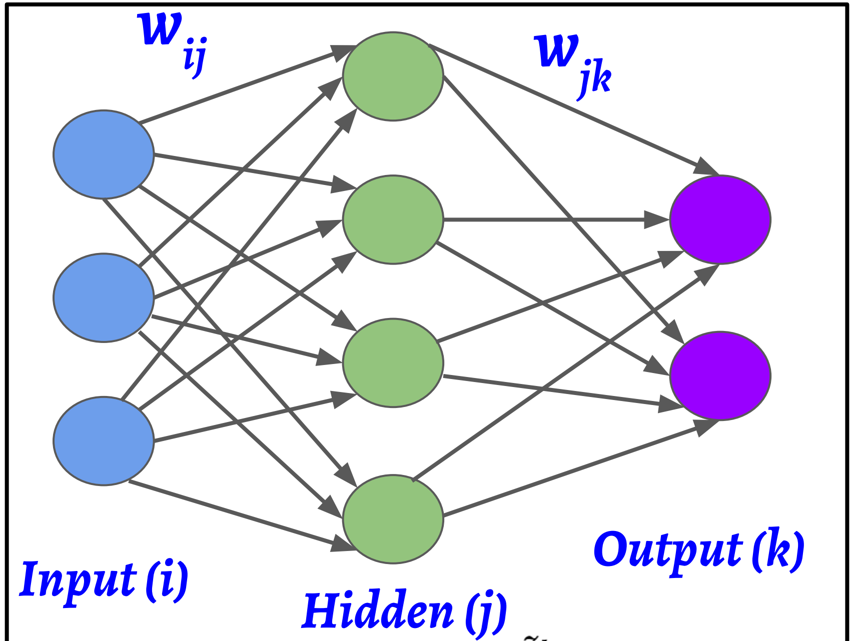


$$a_k = f_k\Big(\overbrace{\textstyle\sum_j \underbrace{f_j(\underbrace{\textstyle\sum_i a_i w_{ij}}_{z_j})}_{a_j} w_{jk}}^{z_k}\Big)$$

Backpropagation Algorithm (cont'd)

- **Backpropagation – for hidden layer**

$$
\begin{aligned}
\frac{\partial E}{\partial w_{ij}} &= \sum_{k\in K}(a_k - t_k)f_k'(z_k)\underbrace{w_{jk}f_j'(z_j)a_i}_{\partial z_k/\partial w_{ij}} \\
&= f_j'(z_j)a_i\sum_{k\in K}\underbrace{(a_k - t_k)f_k'(z_k)w_{jk}}_{\delta_k} \\
&= \underbrace{a_i f_j'(z_j)\sum_{k\in K}\delta_k w_{jk}}_{\delta_j} \\
&= \delta_j a_i
\end{aligned}
$$



$$a_k = f_k\Big(\overbrace{\sum_j \underbrace{f_j(\underbrace{\sum_i a_i w_{ij}}_{z_j})}_{a_j} w_{jk}}^{z_k}\Big)$$

References

- https://www.cs.swarthmore.edu/~meeden/cs81/s10/BackPropDeriv.pdf
- https://cs224d.stanford.edu/lecture_notes/notes3.pdf
- http://www.cs.cmu.edu/~ninamf/courses/315sp19/lectures/3_29-NNs.pdf
- https://cedar.buffalo.edu/~srihari/CSE574/Chap5/Chap5.1-FeedFor.pdf
- http://ruder.io/optimizing-gradient-descent/
- http://www.cs.cmu.edu/~ninamf/courses/315sp19/lectures/Perceptron-01-25-2019.pdf
- http://www.cs.cornell.edu/courses/cs5740/2016sp/resources/backprop.pdf

Thank You!

AI-NLP-ML Group, Department of CSE, IIT Patna (http://www.iitp.ac.in/~ai-nlp-ml/)