CS722/822: Machine Learning

Instructor: Jiangwen Sun
Computer Science Department

Neural networks

- Introduction
- Different designs of NN
- Feed-forward Network (MLP)
- Network Training
- Error Back-propagation
- Regularization

Introduction

- Neuroscience studies how networks of neurons produce intellectual behavior, cognition, emotion and physiological responses
- Computer science studies how to simulate the functions that biological neural network has
  - Artificial neural networks simulate the connectivity in the neural system, the way it passes through signal, and mimic the massively parallel operations of the human brain

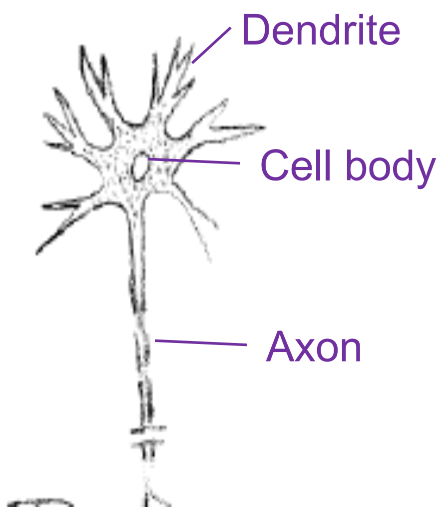
Common features

Common features

Biological Neural Networks

models used in

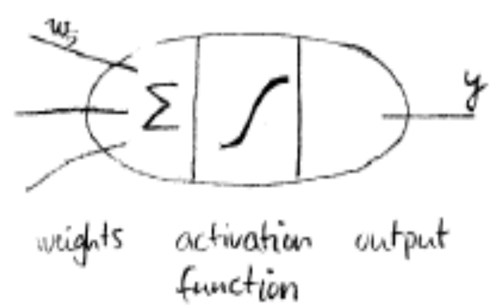
Theoretical Neuroscience



Artificial Neural Networks

models used in

Function Approximation
Classification
Data Processing



Common features

Biological Neural Networks

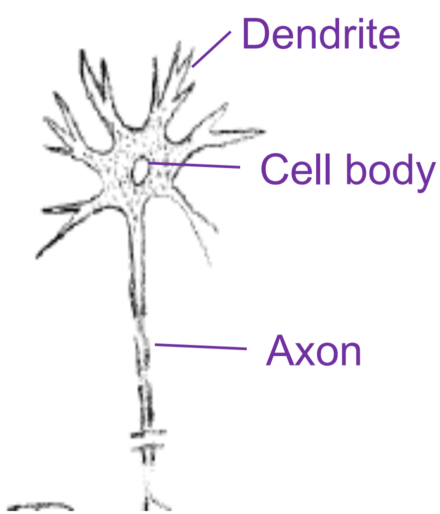
common features

Artificial Neural Networks

Parallel computing
Non-linear units
Local processing
Adaptation

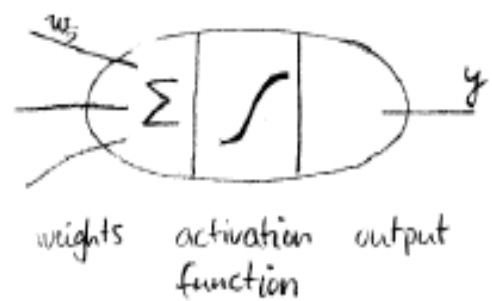
models used in

Theoretical Neuroscience



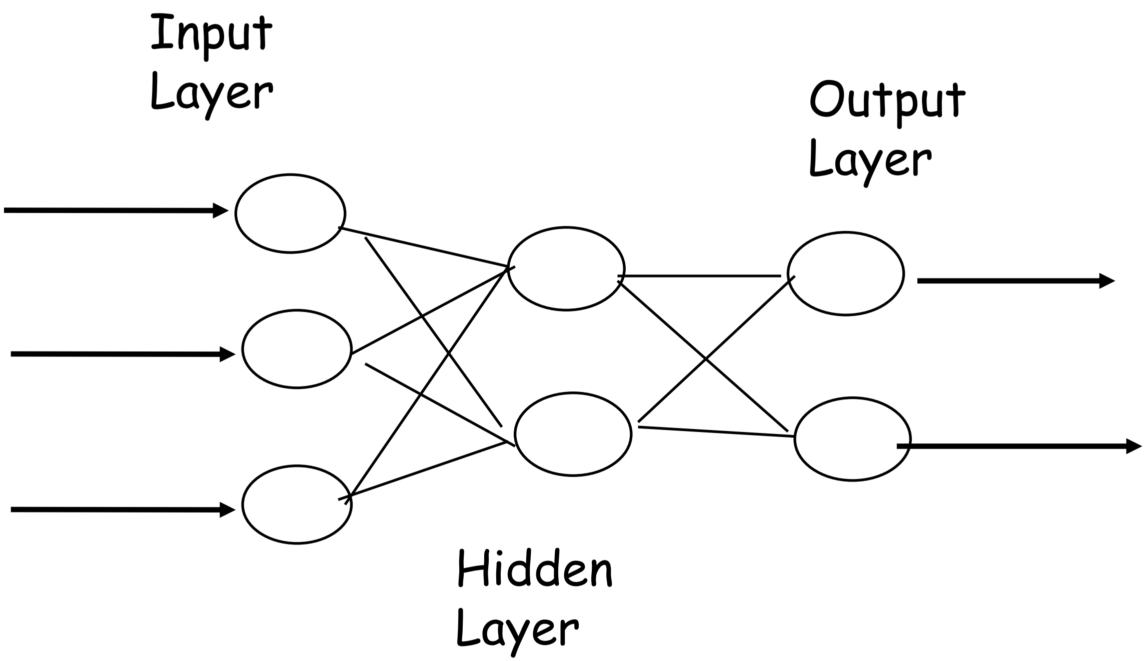
models used in

Function Approximation
Classification
Data Processing



Different types of NN

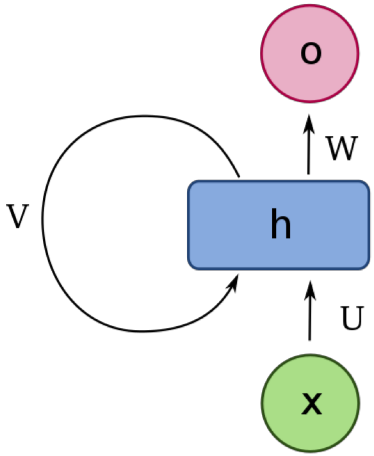
- Feed-forward NN



Variants: CNN, ResNet, etc.

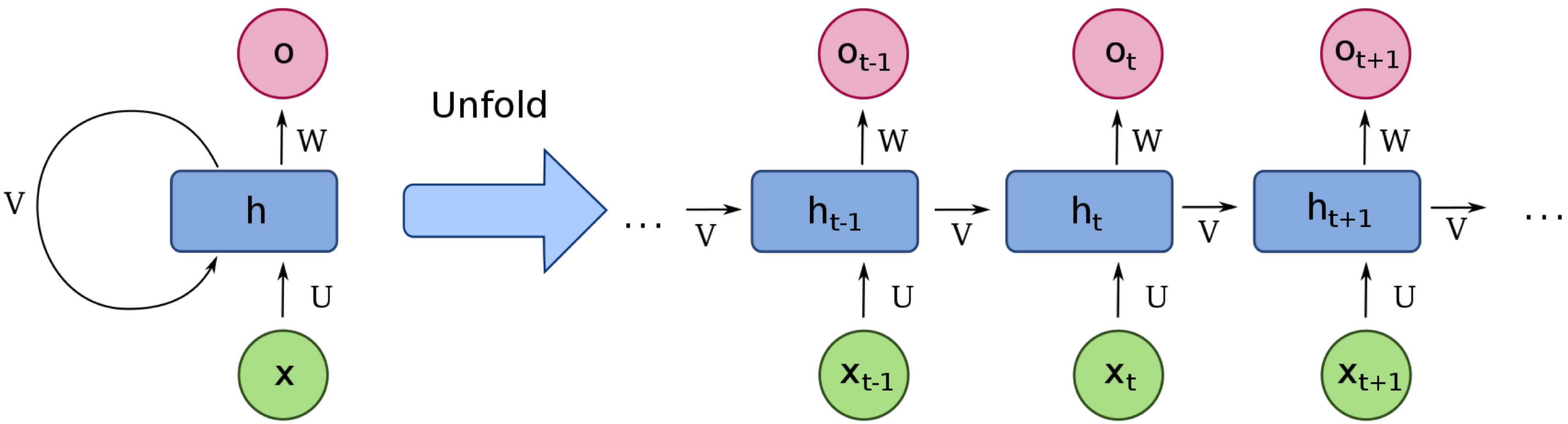
Different types of NN

- Recurrent NN



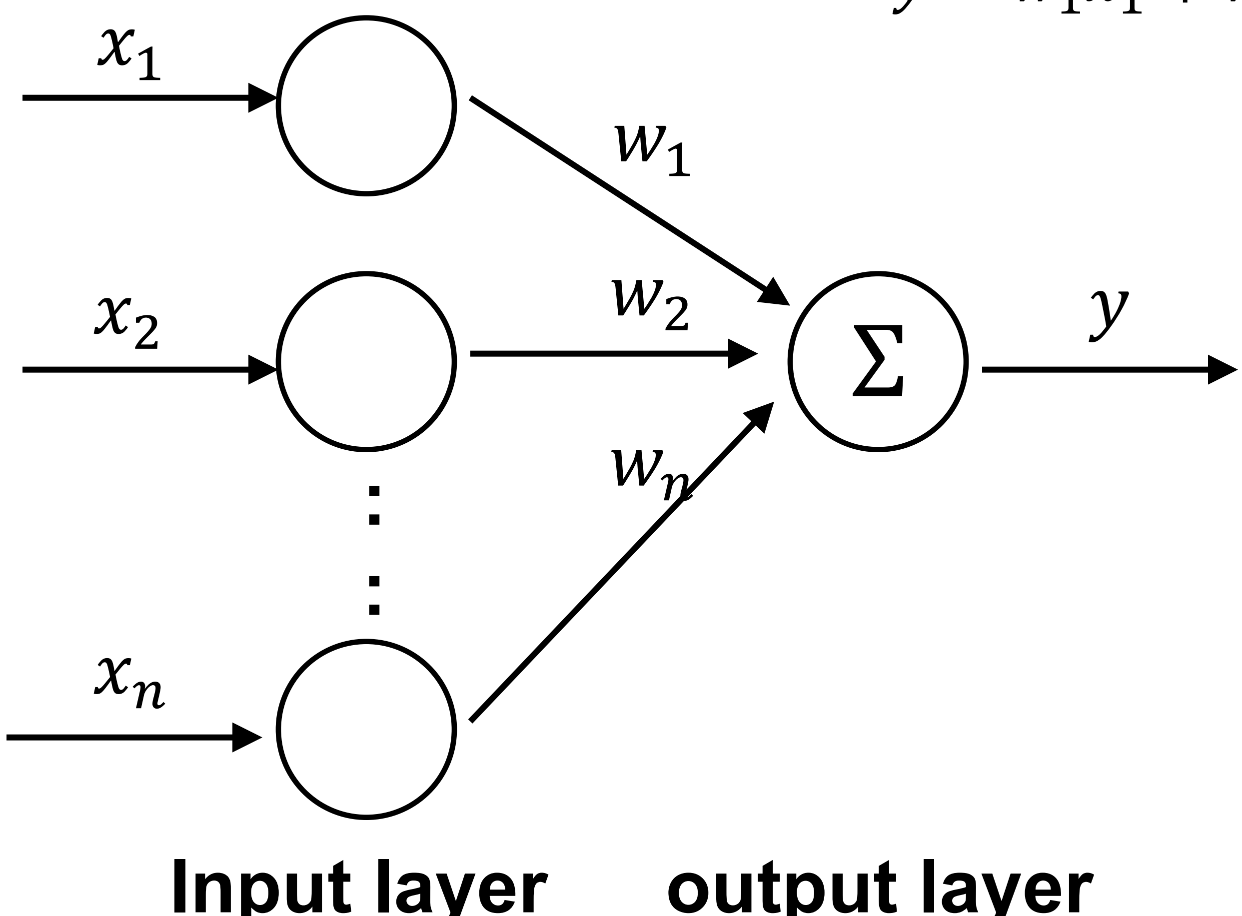
Different types of NN

- Recurrent NN



Linear perceptron

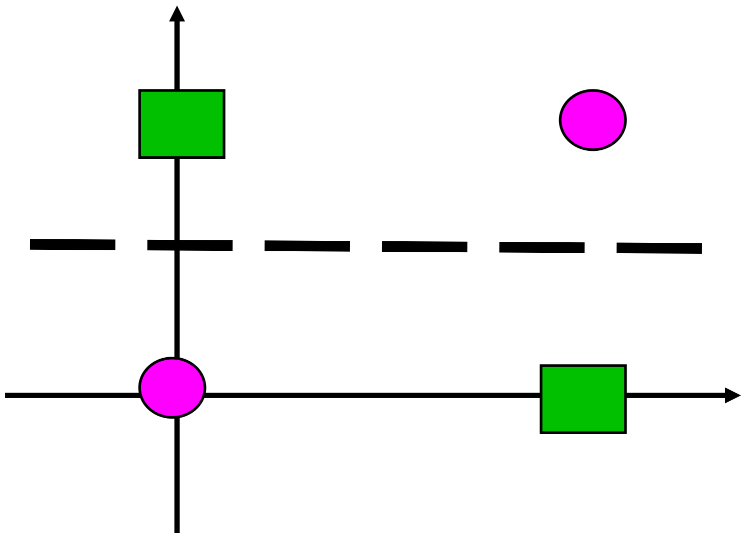
$$y = w_1x_1 + w_2x_2 + \cdots + w_nx_n$$



Many functions can not be approximated using linear perceptron

Linear Perceptron

- XOR (exclusive OR) problem
- 0+0=0
- 1+1=2=0 mod 2
- 1+0=1
- 0+1=1
- Perceptron does not work here!

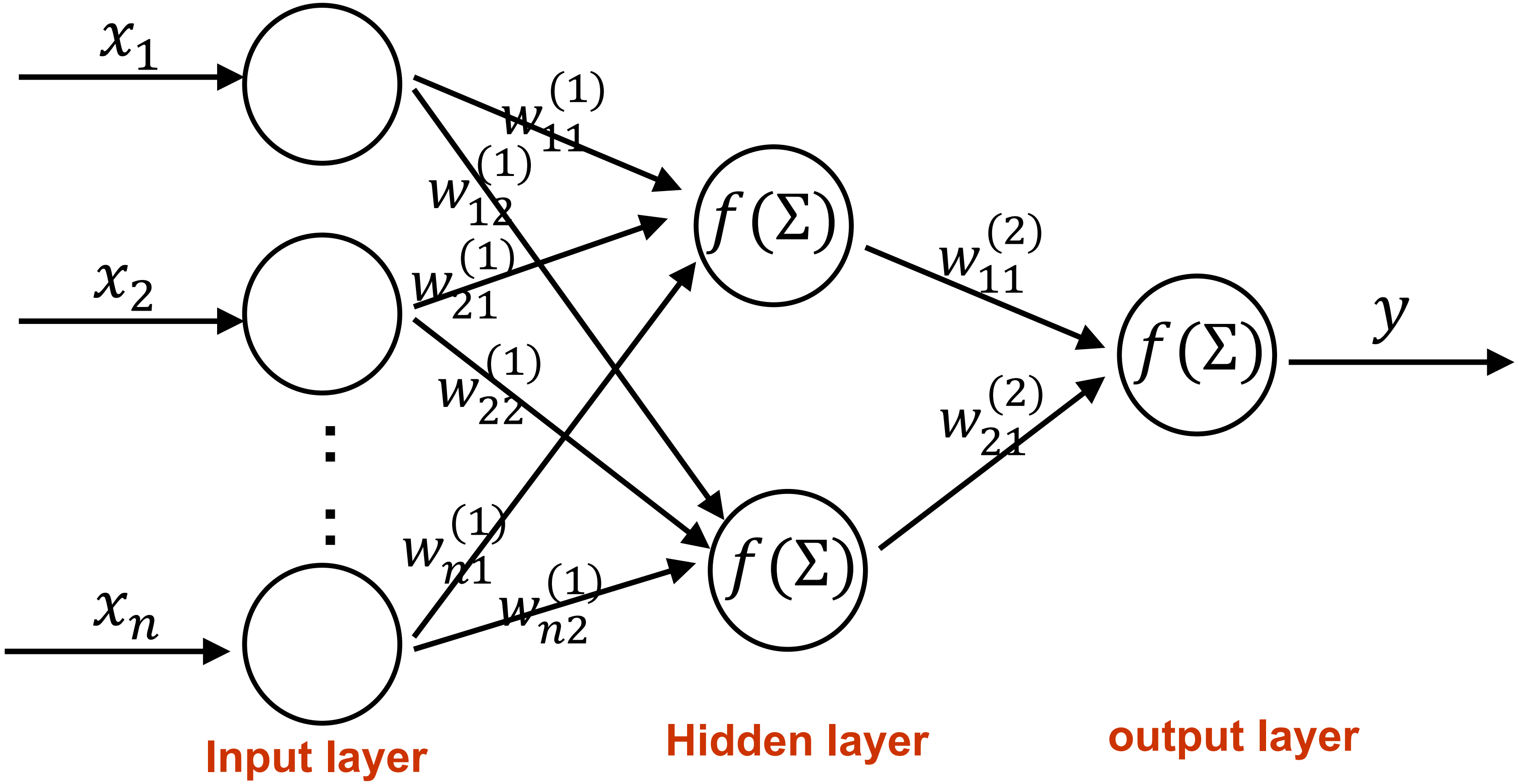


Single layer generates a linear decision boundary

Multi-Layer Perceptron

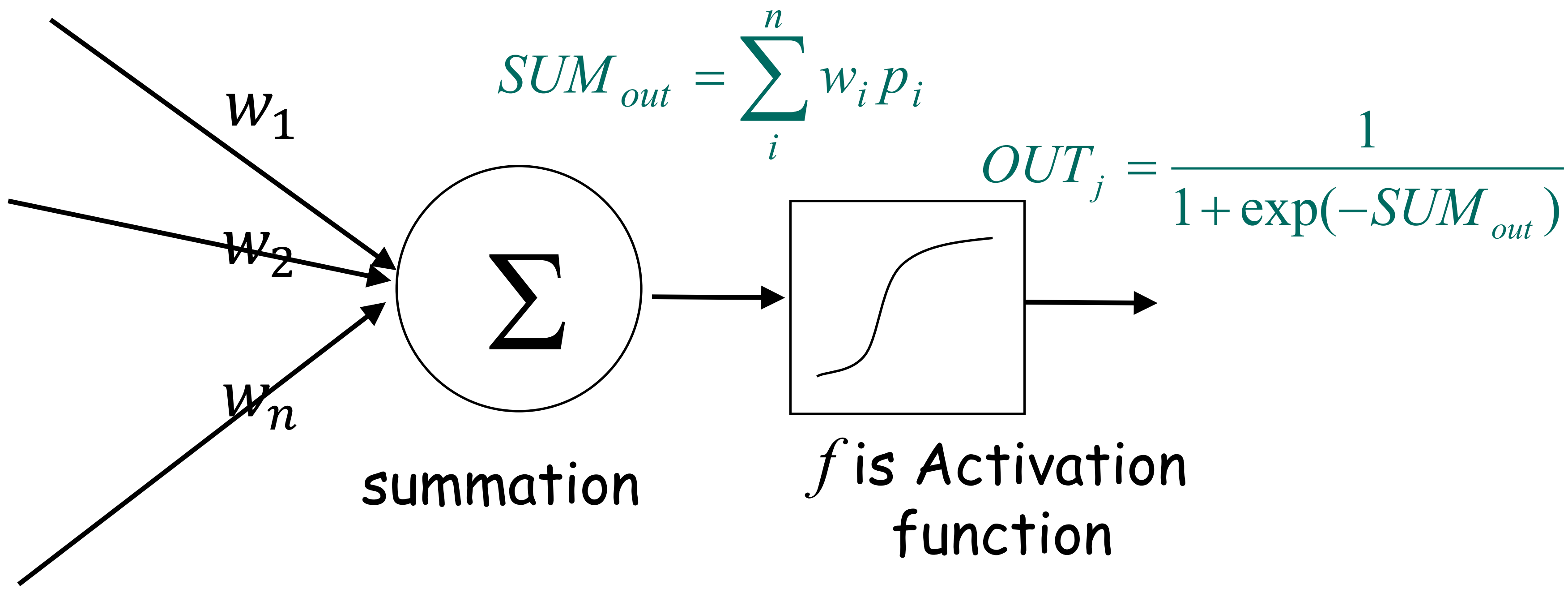
- Multi-layer perceptron (MLP) networks are a type of **feed-forward** NN
- They are a class of models that are formed from layered nodes with **activation function (*f*)** such as **sigmoidal**, which can be used for regression or classification purposes
- They can realize any logical function
- They are commonly trained using **gradient descent** on a **mean squared error performance function**, using a technique known as **error back propagation** in order to calculate the gradients
- Widely applied to many prediction and classification problems

Multi-Layer Perceptron



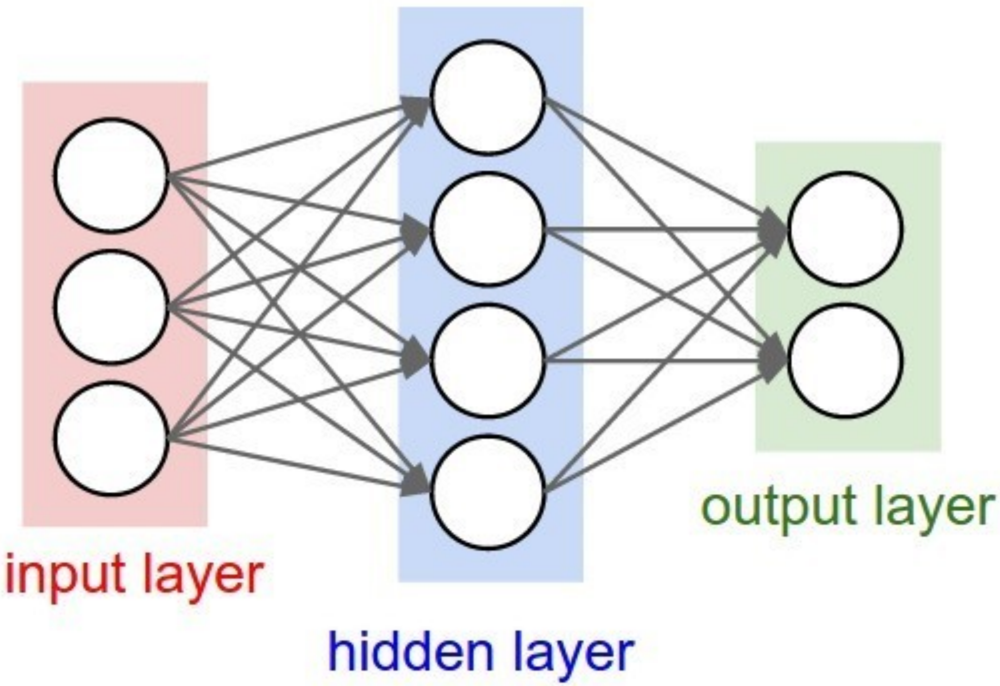
- Each link is associated with a weight, and these weights are the tuning parameters to be learned
- Each neuron except ones in the input layer receives inputs from the previous layer, and reports an output to next layer

Each neuron



- The activation function f can be
  - Identity function: $f(x) = x$
  - Sigmoid function: $f(x) = 1/(1 + e^{-x})$
  - Hyperbolic tangent: $f(x) = (e^{2x} - 1)/(e^{2x} + 1)$

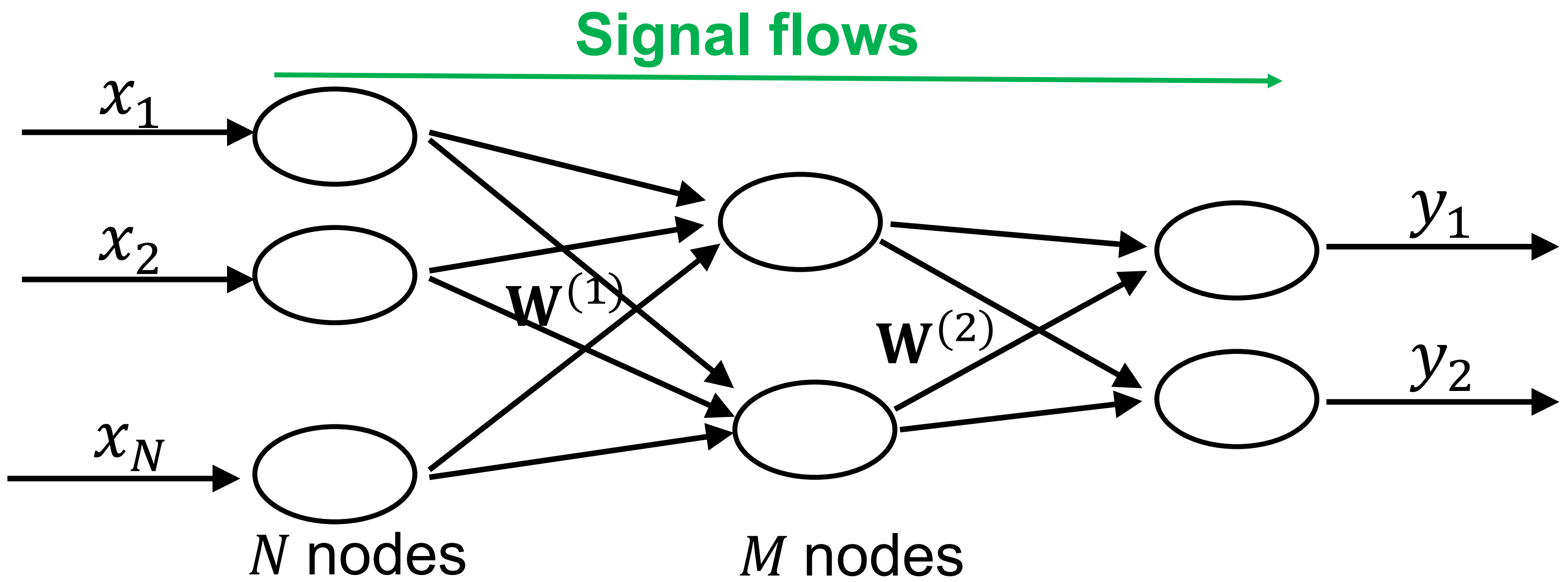
Universal Approximation of MLP



MLP with 1 hidden layer can represent any bounded continuous function to arbitrary ε

- Universal Approximation Theorem [Cybenko 1998]

Feed-forward network function



- The output from each hidden node

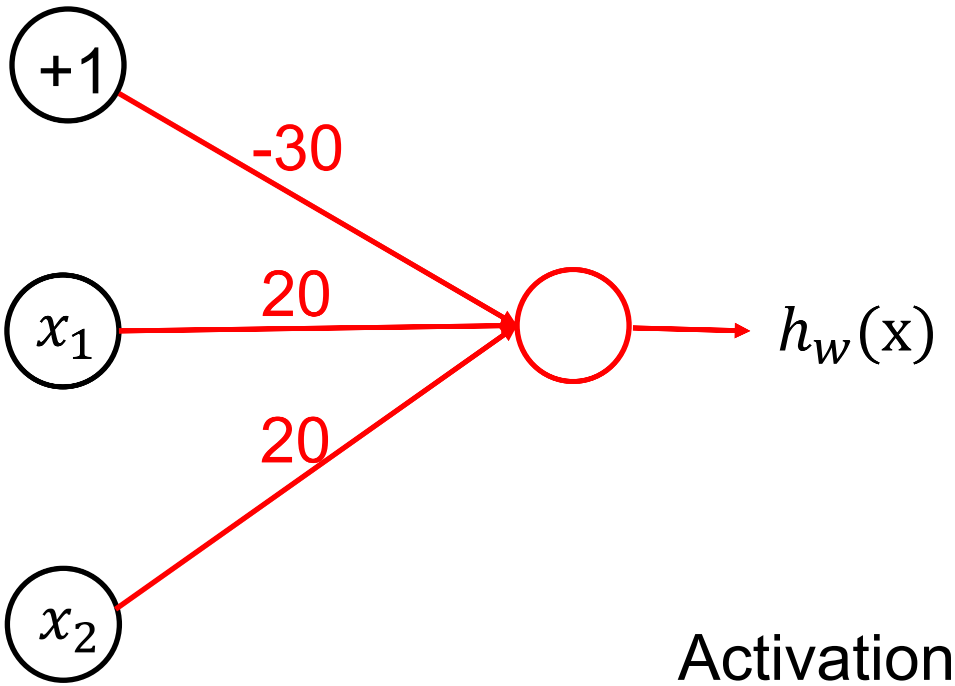
$$o_j^{(1)} = f\left(\sum_{i=1}^{N} w_{ij}^{(1)} x_i\right)$$

- The final output

$$y_k = f\left(\sum_{j=1}^{M} w_{jk}^{(2)} o_j^{(1)}\right)$$

Network for Computing logic functions

x_1 **AND** x_2

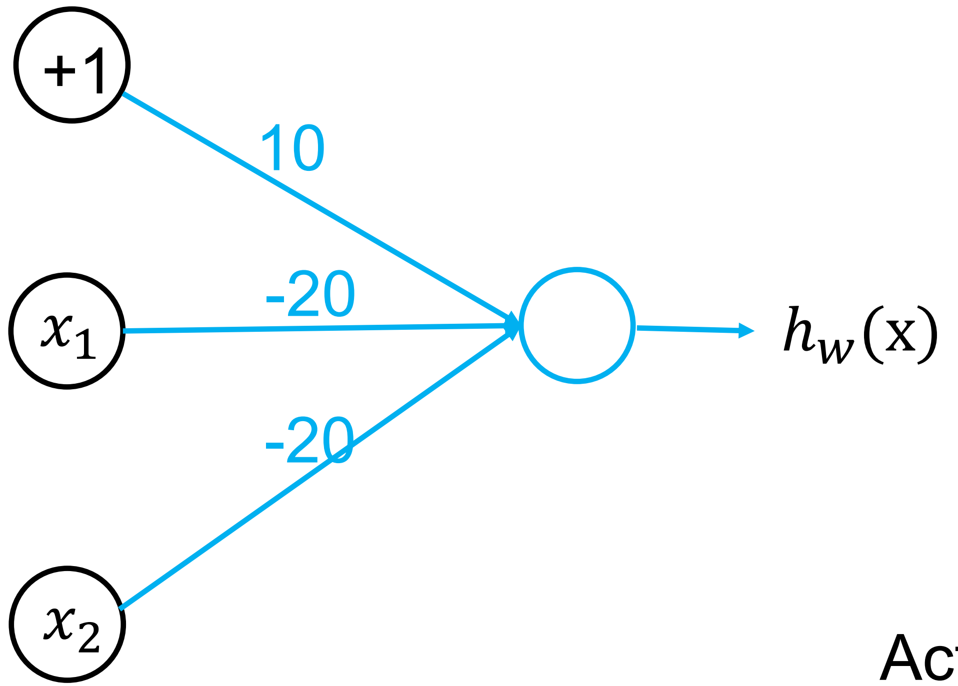


Activation function

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

Network for Computing logic functions

$(\mathbf{NOT}\ x_1)\ \mathbf{AND}\ (\mathbf{NOT}\ x_2)$

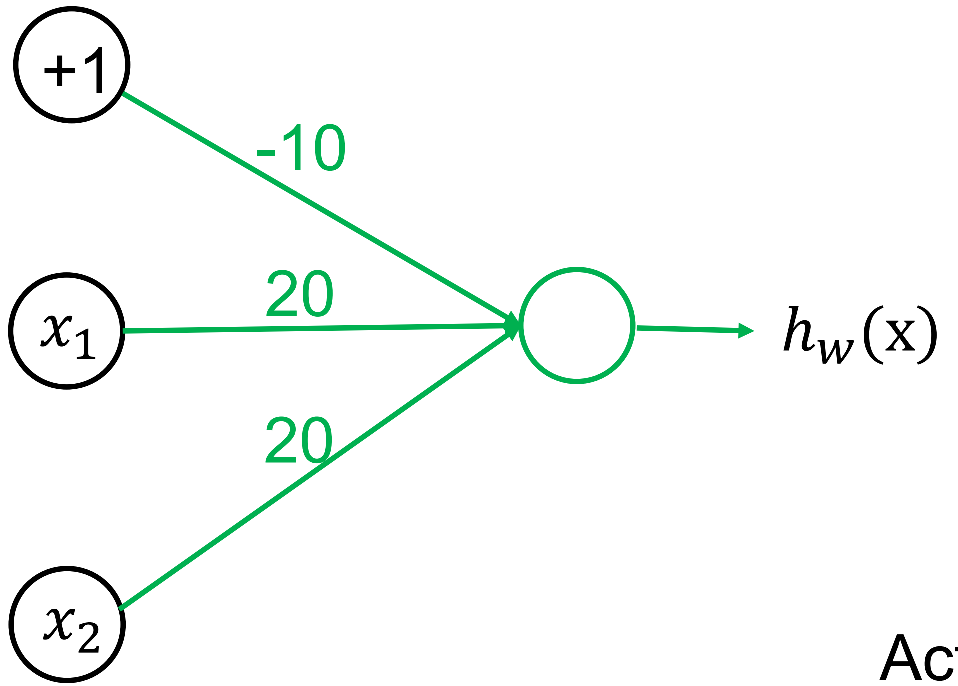


Activation function

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

Network for Computing logic functions

x_1 **OR** x_2

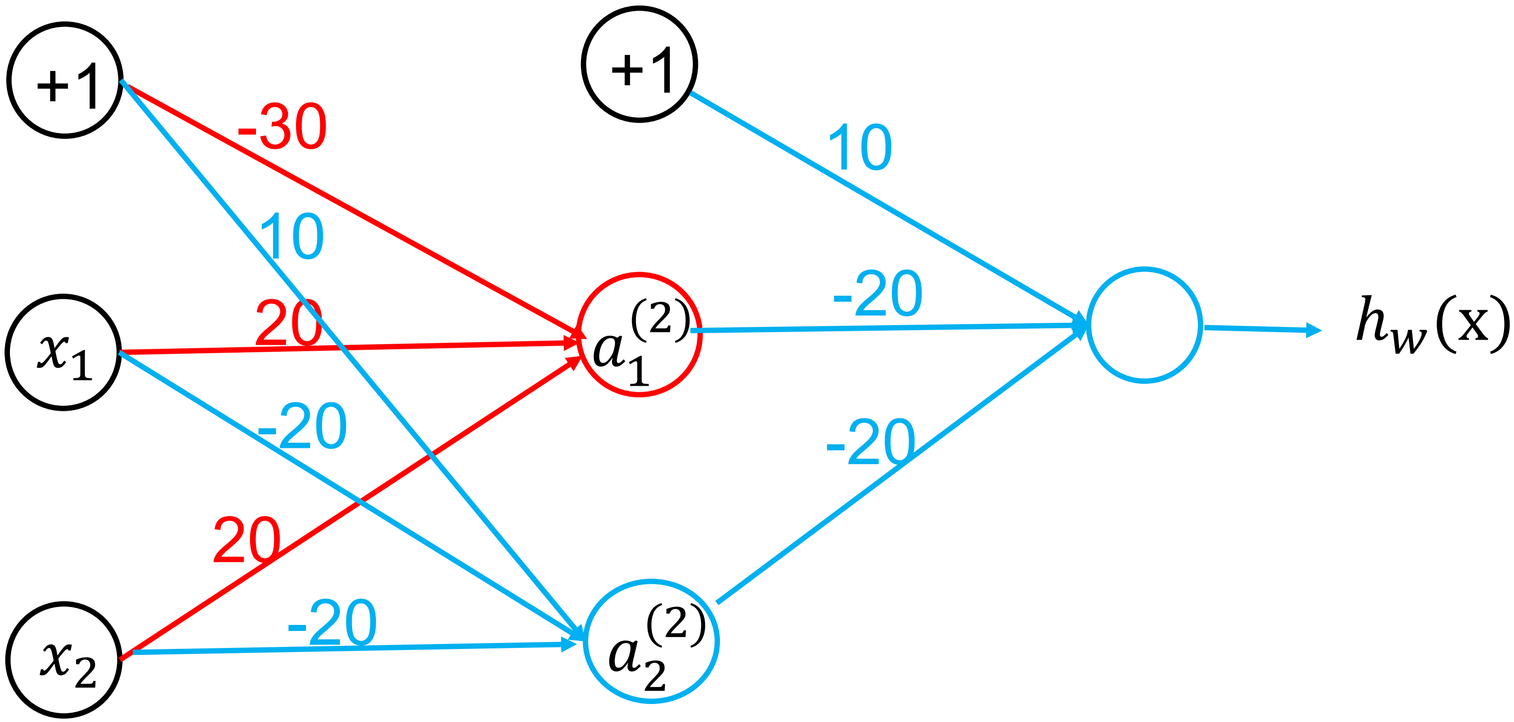


Activation function

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

Network for Computing logic functions

x_1 **XOR** x_2



Network Training

- A supervised neural network is a function $h_{\mathbf{w}}(\mathbf{x})$ that maps from inputs $\mathbf{x}$ to target y
- Usually training a NN does not involve the change of NN structures (such as how many hidden layers or how many hidden nodes)
- Training NN refers to adjusting the values of connection weights so that $h_{\mathbf{w}}(\mathbf{x})$ adapts to the problem
- Use sum of squares as the error metric

$$E(\mathbf{w}) = \sum_{l=1}^{L} \left(y_l - h_{\mathbf{w}}(\mathbf{x}_l)\right)^2$$

Use gradient descent

$$-\frac{\partial E(\mathbf{w})}{\partial w_{ij}^{(k)}}$$

Gradient descent

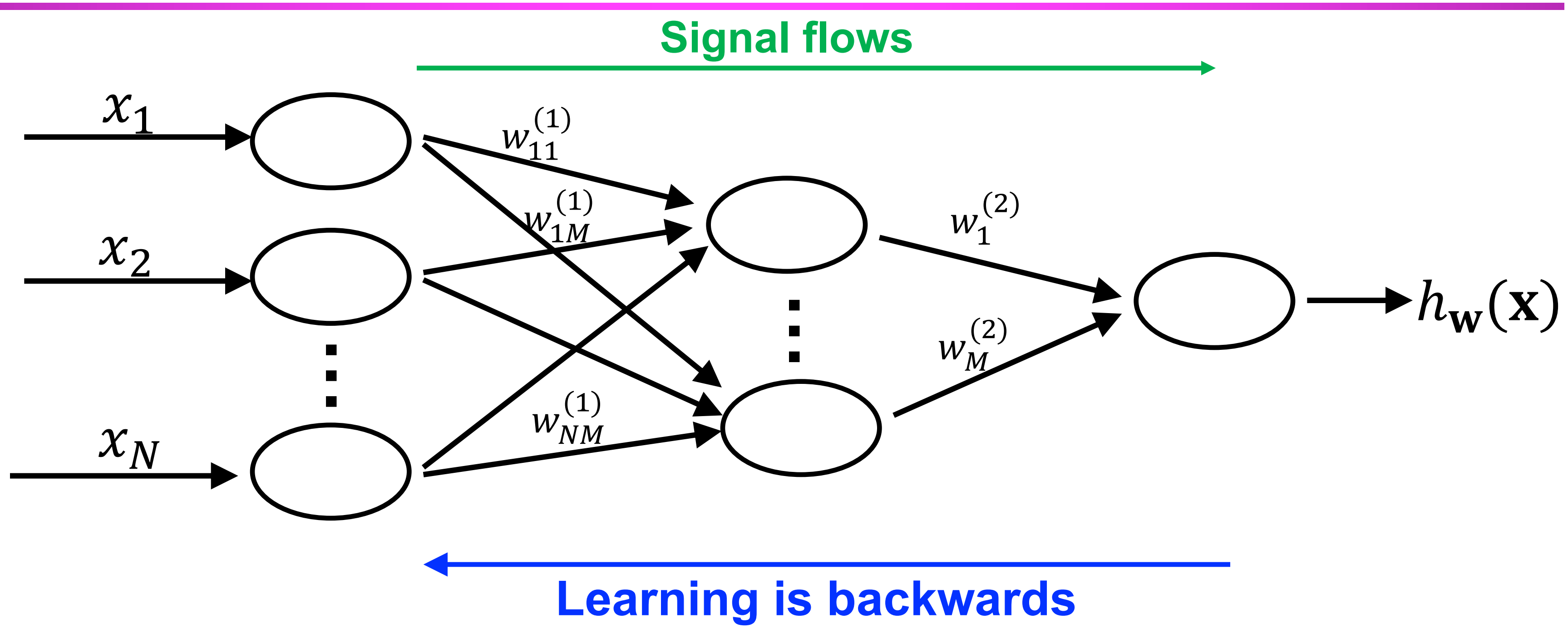
- Review of gradient descent
- Iterative algorithm containing many iterations
- Each iteration t, the weights $\mathbf{w}$ receive a small update

$$w_{ij}^{t} = w_{ij}^{t-1} + \alpha\left(-\frac{\partial E}{\partial w_{ij}}\right)$$

- Terminate
  - until the network is stable (in other words, the training error cannot be reduced further)

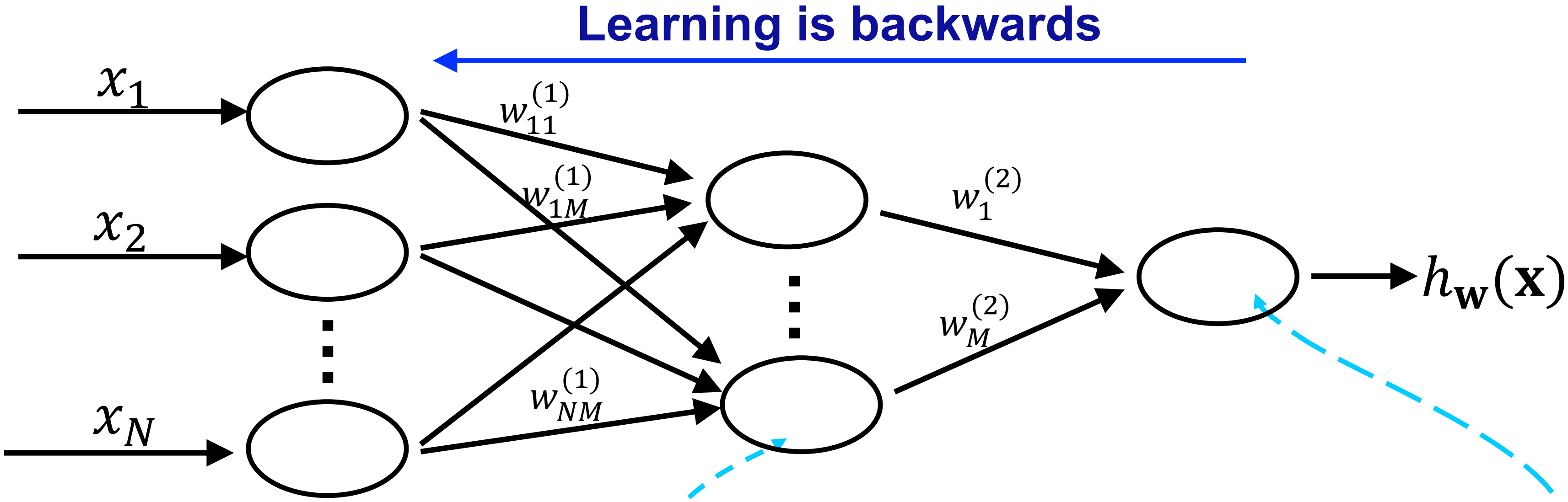
    $E(\mathbf{w}^{t}) < E(\mathbf{w}^{t-1})$ not hold
  - until the error on a validation set starts to climb up (early stopping)

Error Back-propagation



- The update of the weights goes backwards because we have to use the chain rule to evaluate the gradient of $E(w)$

Error Back-propagation



- Calculate the gradient associated with the weights in the output layer first
- Propagate from the high layer to low layer
- Recall

$$o_j^{(2)} = f\left(\sum_{i=1}^{N} w_{ij}^{(1)} x_i\right) \qquad h_{\mathbf{w}}(\mathbf{x}) = f\left(\sum_{j=1}^{M} w_j^{(2)} o_j^{(2)}\right)$$

Evaluate gradient

$$E(\mathbf{w}) = \sum_{l=1}^{L} (\hat{y}_l - y_l)^2 = \sum_{l=1}^{L} E_l, \qquad \text{where } \hat{y}_l = h_{\mathbf{w}}(\mathrm{x}_l)$$

- First compute the partial derivatives for weights in the output layer

$$\frac{\partial E_l}{\partial w_j^{(2)}} = \frac{\partial E_l}{\partial \hat{y}_l} \frac{\partial \hat{y}_l}{\partial w_j^{(2)}} = \underbrace{\boxed{2(\hat{y}_l - y_l)}}_{\frac{\partial E_l}{\partial \hat{y}_l}} \underbrace{\boxed{f'\left(\sum_{j=1}^{M} w_j^{(2)} o_j^{(2)}\right) o_j^{(2)}}}_{\frac{\partial \hat{y}_l}{\partial w_j^{(2)}}}$$

- Second compute the partial derivatives for weights in the hidden layer

$$\frac{\partial E_l}{\partial w_{ij}^{(1)}} = \frac{\partial E_l}{\partial \hat{y}_l} \frac{\partial \hat{y}_l}{\partial o_j^{(2)}} \frac{\partial o_j^{(2)}}{\partial w_{ij}^{(1)}} = \underbrace{\boxed{2(\hat{y}_l - y_l)}}_{\frac{\partial E_l}{\partial \hat{y}_l}} \underbrace{\boxed{f'\left(\sum_{j=1}^{M} w_j^{(2)} o_j^{(2)}\right) w_j^{(2)}}}_{\frac{\partial \hat{y}_l}{\partial o_j^{(2)}}} \underbrace{\boxed{f'\left(\sum_{i=1}^{N} w_{ij}^{(1)} x_{l,i}\right) x_{l,i}}}_{\frac{\partial o_j^{(2)}}{\partial w_{ij}^{(1)}}}$$

Evaluate gradient

$$\frac{\partial E_l}{\partial w_j^{(2)}} = 2(\hat{y}_l - y_l) f'\left(\sum_{j=1}^{M} w_j^{(2)}\, o_j^{(2)}\right) o_j^{(2)} \qquad \text{Layer 3}$$

$$\frac{\partial E_l}{\partial w_{ij}^{(1)}} = 2(\hat{y}_l - y_l) f'\left(\sum_{j=1}^{M} w_j^{(2)}\, o_j^{(2)}\right) w_j^{(2)} f'\left(\sum_{i=1}^{N} w_{ij}^{(1)}\, x_i\right) x_i \qquad \text{Layer 2}$$

Back-propagation

Back-propagation algorithm

- Design the structure of NN
- Initialize all connection weights
- For $t = 1$, to T
  - Present training examples, propagate forwards from input layer to output layer, compute y, and evaluate the errors
  - Pass errors backwards through the network to recursively compute derivatives, and use them to update weights $$w_{ij}{}^{t} = w_{ij}{}^{t-1} + \alpha\left(-\frac{\partial E}{\partial w_{ij}}\right)$$
  - If termination rule is met, stop; or continue
- end

Notes on back-propagation

- Note that these rules apply to different kinds of feed-forward networks. It is possible for connections to skip layers, or to have mixtures. However, errors always start at the highest layer and propagate backwards

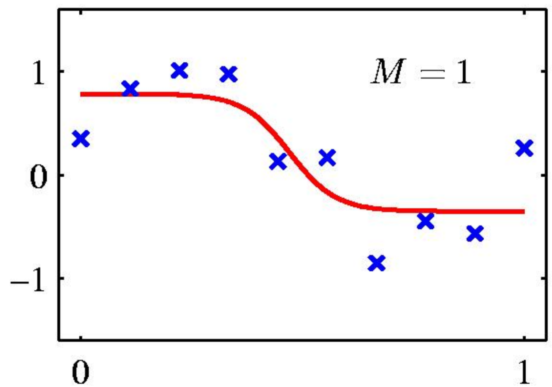
Two schemes of training

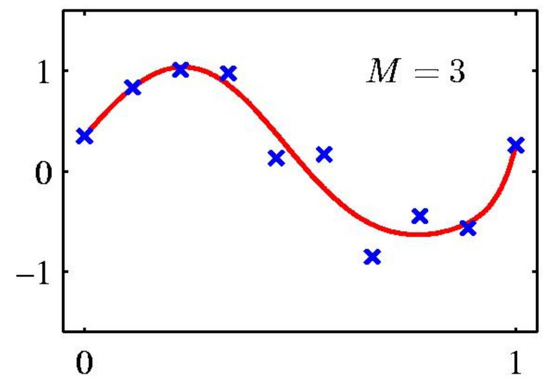
- There are two schemes of updating weights
  - Batch: Update weights after all examples have been presented (epoch).
  - Online: Update weights after each example is presented.
- Although the batch update scheme implements the true gradient descent, the second scheme is often preferred since
  - it requires less storage,
  - it has more noise, hence is less likely to get stuck in a local minima (which is a problem with nonlinear activation functions). In the online update scheme, order of presentation matters!

Problems of back-propagation

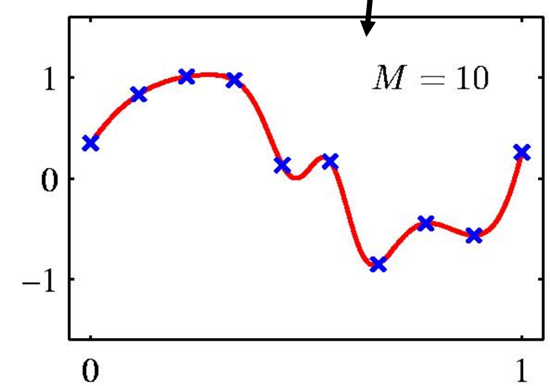
- It is extremely slow, if it does converge.
- It may get stuck in a local minima.
- It is sensitive to initial conditions.
- It may start oscillating.

Overfitting – number of hidden units





Over-fitting



Sinusoidal data set used in polynomial curve fitting example

Regularization (1)

- How to adjust the number of hidden units to get the best performance while avoiding over-fitting

- Add a penalty term to the error function

$$\tilde{E}(\mathbf{W}) = E(\mathbf{W}) + \lambda R(\mathbf{W})$$

- The simplest regularizer is the *weight decay*:

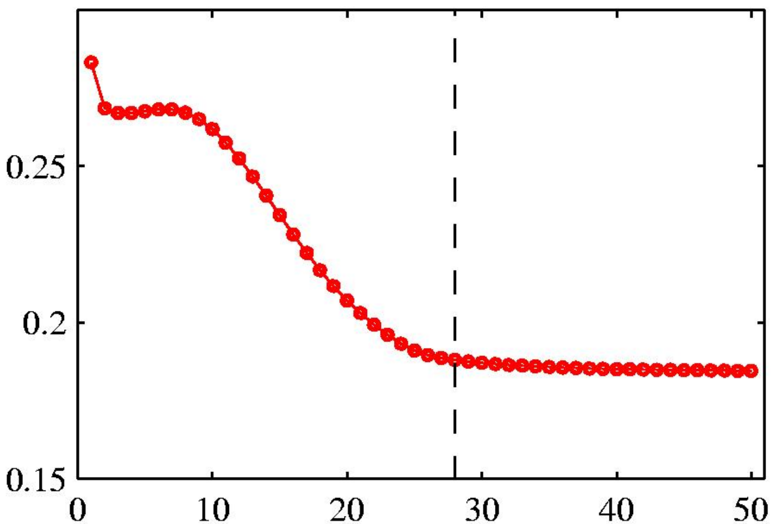
$$w_{ij}^{t} = (1 - \alpha\lambda) w_{ij}^{t-1} + \alpha\left(-\frac{\partial E}{\partial w_{ij}}\right)$$

Regularization (2)

- A method to Early Stopping
  - obtain good generalization performance and
  - control the effective complexity of the network
- Instead of iteratively reducing the error until a minimum error on the training data set has been reached
- We have a validation set of data available
- Stop when the NN achieves the smallest error w.r.t. the validation data set

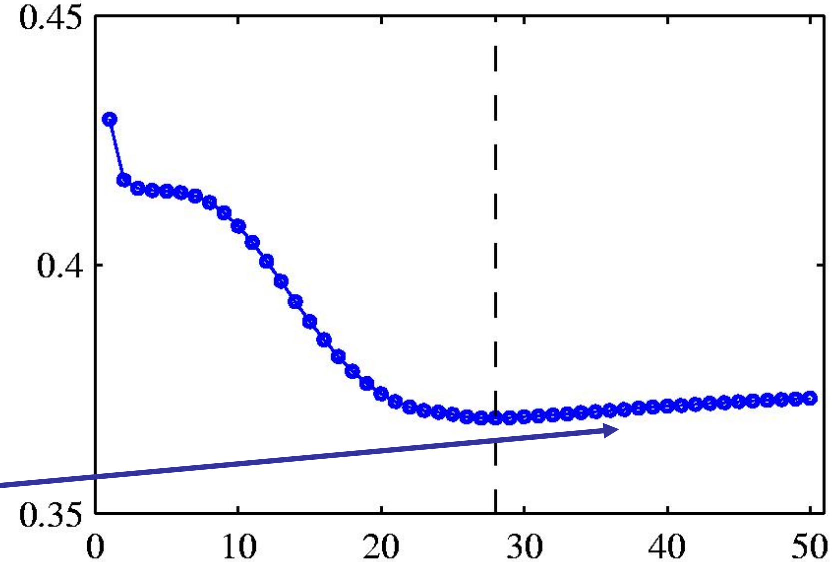
Effect of early stopping

Training Set



Error vs. Number of iterations

Validation Set



A slight increase in the validation set error