Neural Networks

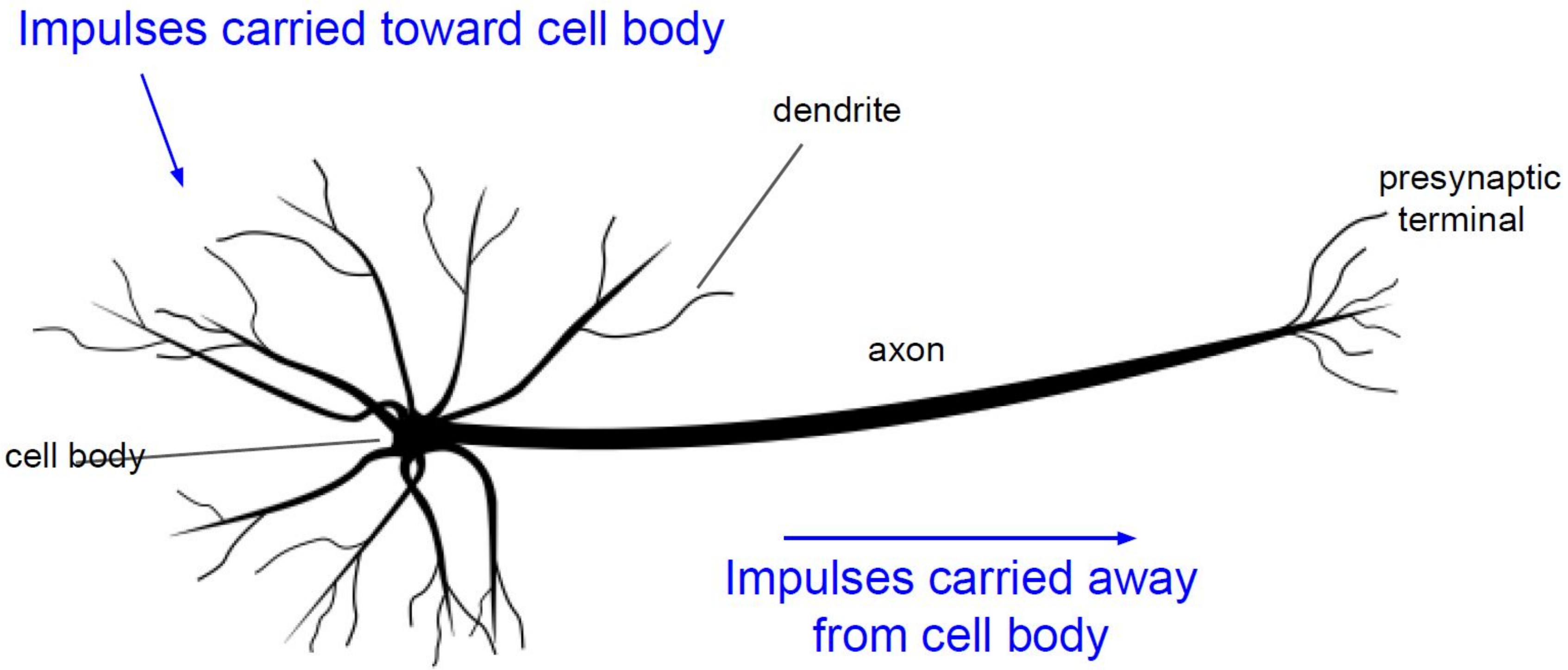
Machine Learning

Hamid R Rabiee – Zahra Dehghanian

Spring 2025

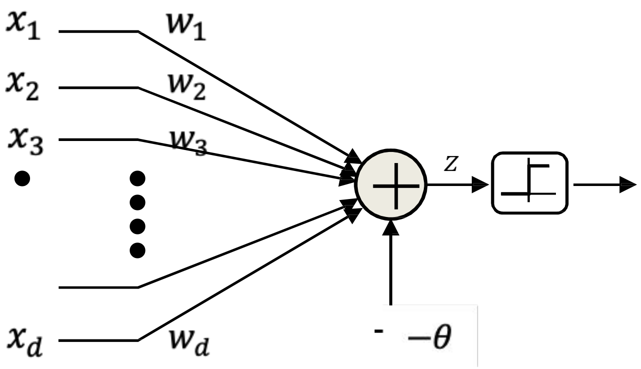
Sharif University of Technology

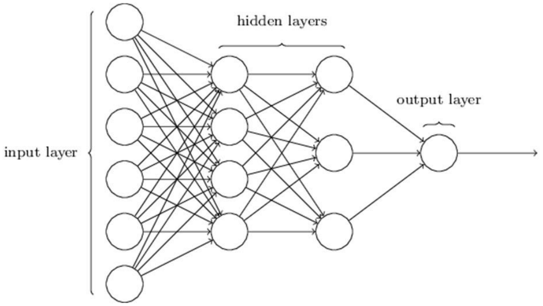
Neural Cell

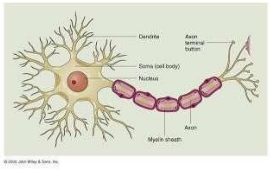

Impulses carried toward cell body
dendrite
presynaptic terminal
axon
cell body
Impulses carried away from cell body

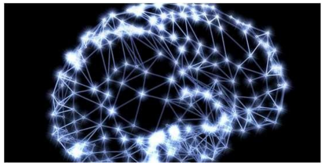
This image by Felipe Perucho is licensed under CC-BY 3.0

Neural Nets and the brain









- Neural nets are composed of networks of computational models of neurons called perceptrons

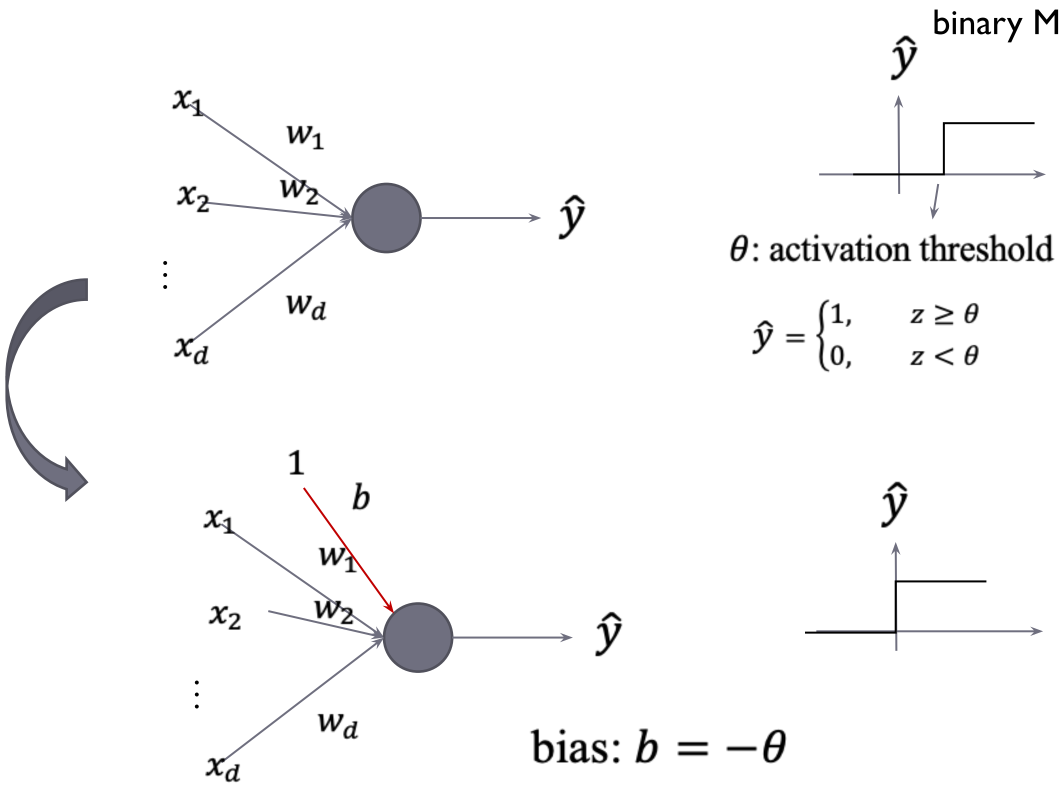
Neural Network

- Learn a non-linear function $f_w: X \rightarrow Y$:
  - X input or feature space: (vector of) continuous and/or discrete features
  - Y output space: (vector of) continuous and/or discrete variables
- Neural networks use basic units to provide a non-linear function

History

- 1940s–1960s:
  - development of theories of biological learning
  - implementations of the first models
    - perceptron (Rosenblatt, 1958) for training of a single neuron.
- 1980s-1990s: back-propagation algorithm to train a neural network with more than one hidden layer
  - too computationally costly to allow much experimentation with the hardware available at the time.

Binary threshold neurons) McCulloch-Pitts neuron(



binary McCulloch-Pitts neuron

θ: activation threshold

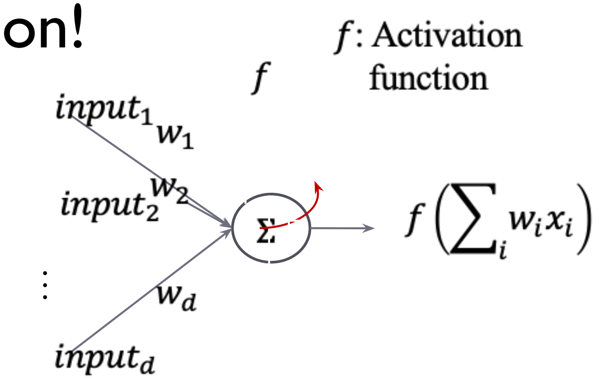
$$\hat{y} = \begin{cases} 1, & z \geq \theta \\ 0, & z < \theta \end{cases}$$

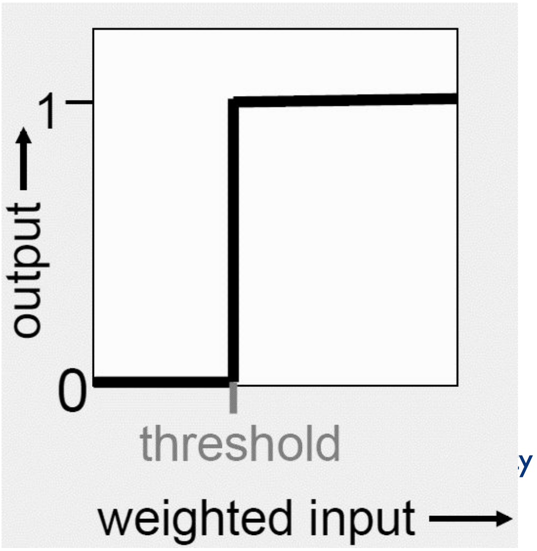
Equivalent to

bias: $b = -\theta$

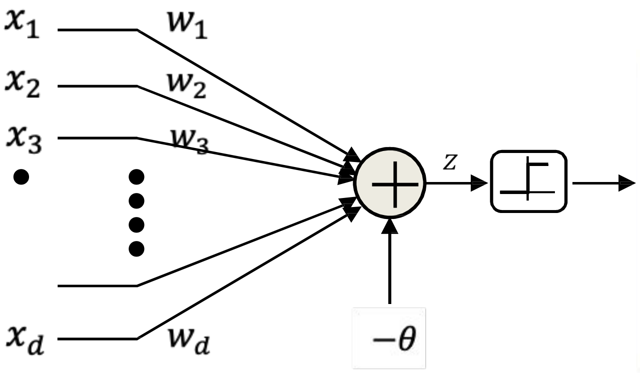
Binary Threshold Neurons

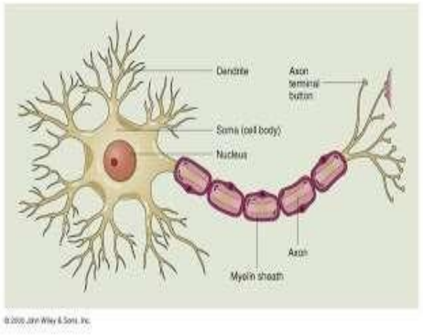
- McCulloch-Pitts (1943): influenced Von Neumann.
  - First compute a weighted sum of the inputs.
  - send out a spike of activity if the weighted sum exceeds a threshold.
  - McCulloch and Pitts thought that each spike is like the truth value of a proposition and each neuron combines truth values to compute the truth value of another proposition!





The Perceptron

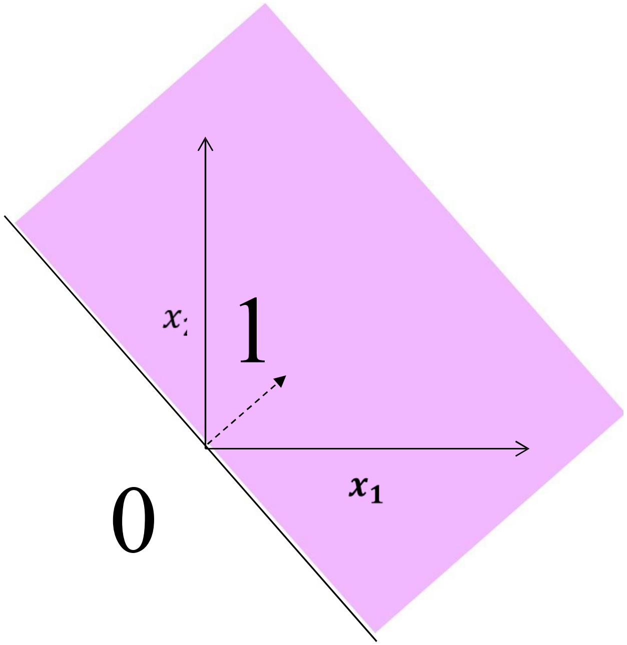


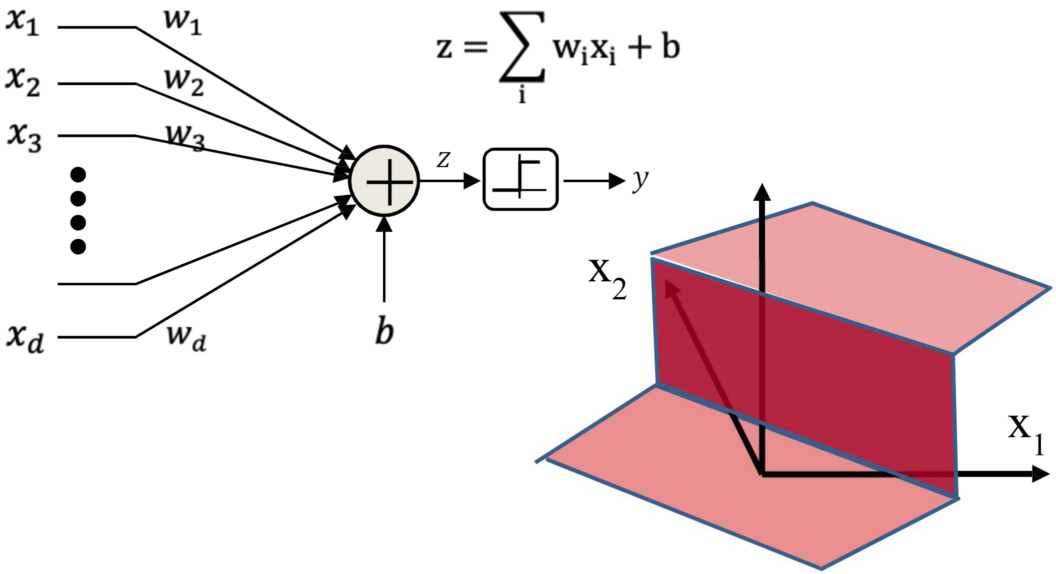


$$\hat{y} = \begin{cases} 1 & \text{if} \sum_i w_i x_i \geq \end{cases}$$

- A   threshold unit
  - "Fires" if the weighted sum of inputs exceeds a threshold
  - Electrical engineers will call this a threshold gate
    - A basic unit of Boolean circuits

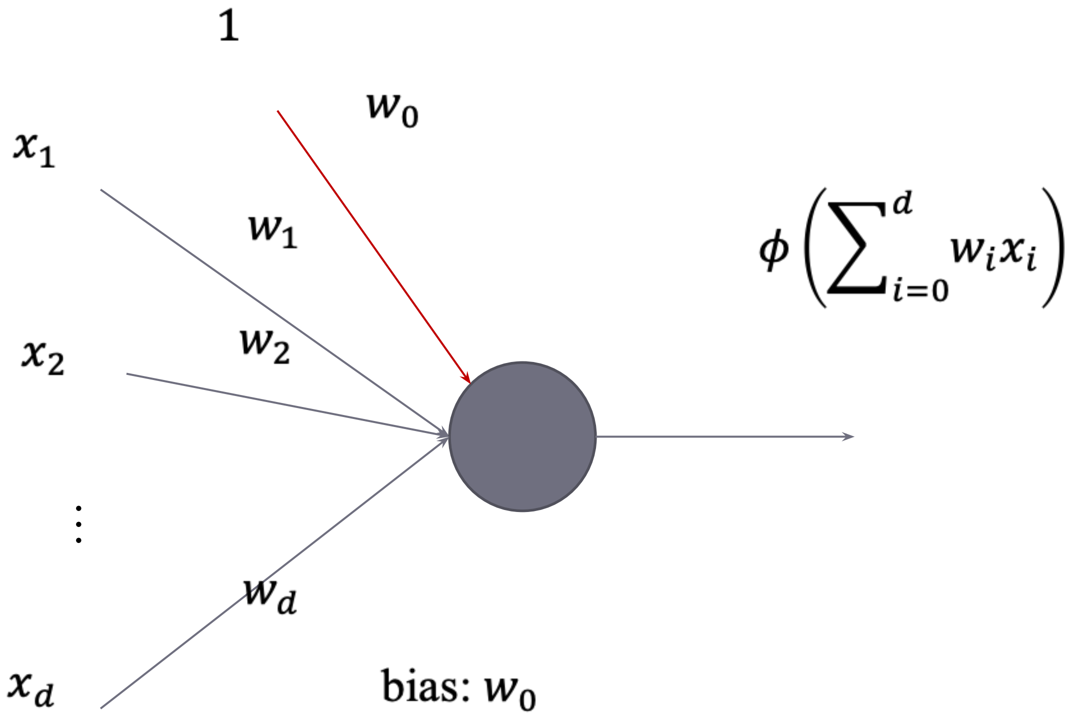
Perceptron





- Lean this function
  - A step function across a hyperplane

Single Layer



Single Layer

- Single layer network can be used as a linear decision boundary:
  - $\phi(\boldsymbol{w}^T \boldsymbol{x})$ shows the class of $\boldsymbol{x}$

- Types of single layer networks:
  - Perceptron (Rosenblatt, 1962)
  - ADALINE (Widrow and Hoff, 1960)

Perceptron Learning Algorithm

- Given N training instances $(\boldsymbol{x}^{(1)}, y^{(1)}), (\boldsymbol{x}^{(2)}, y^{(2)}), \dots, (\boldsymbol{x}^{(N)}, y^{(N)})$
  - $y^{(n)} = +1$ or -1

- Initialize $\boldsymbol{w}$
- Cycle through the training instance
- While more classification errors

```
• For $i = 1 \dots N_{train}$
      $\hat{y}^{(i)} = sign(\boldsymbol{w}^T \boldsymbol{x}^{(i)})$
    • If $\hat{y}^{(i)} \neq y^{(i)}$
          $\boldsymbol{w} = \boldsymbol{w} + y^{(i)} \boldsymbol{x}^{(i)}$
```

- If instance misclassified:
  - If instance is positive class

    $\boldsymbol{w} = \boldsymbol{w} + \boldsymbol{x}$
  - If instance is negative class

    $\boldsymbol{w} = \boldsymbol{w} - \boldsymbol{x}$

Training of Single Layer

- $$\boldsymbol{w}^{t+1} = \boldsymbol{w}^{t} - \eta \nabla E_n(\boldsymbol{w}^{t})$$

- Weight update for a training pair $(\boldsymbol{x}^{(n)}, y^{(n)})$:
  - Perceptron: If $\mathrm{sign}(\boldsymbol{w}^T \boldsymbol{x}^{(n)}) \neq y^{(n)}$ then

    $$\nabla E_n(\boldsymbol{w}^{t}) = -\eta \boldsymbol{x}^{(n)} y^{(n)}$$

    $E_n(\boldsymbol{w}) = -\boldsymbol{w}^T \boldsymbol{x}^{(n)} y^{(n)}$
    if misclassified

  - ADALINE: $\nabla E_n(\boldsymbol{w}^{t}) = -\eta (y^{(n)} - \boldsymbol{w}^T \boldsymbol{x}^{(n)}) \boldsymbol{x}^{(n)}$
    - Widrow-Hoff, LMS, or delta rule

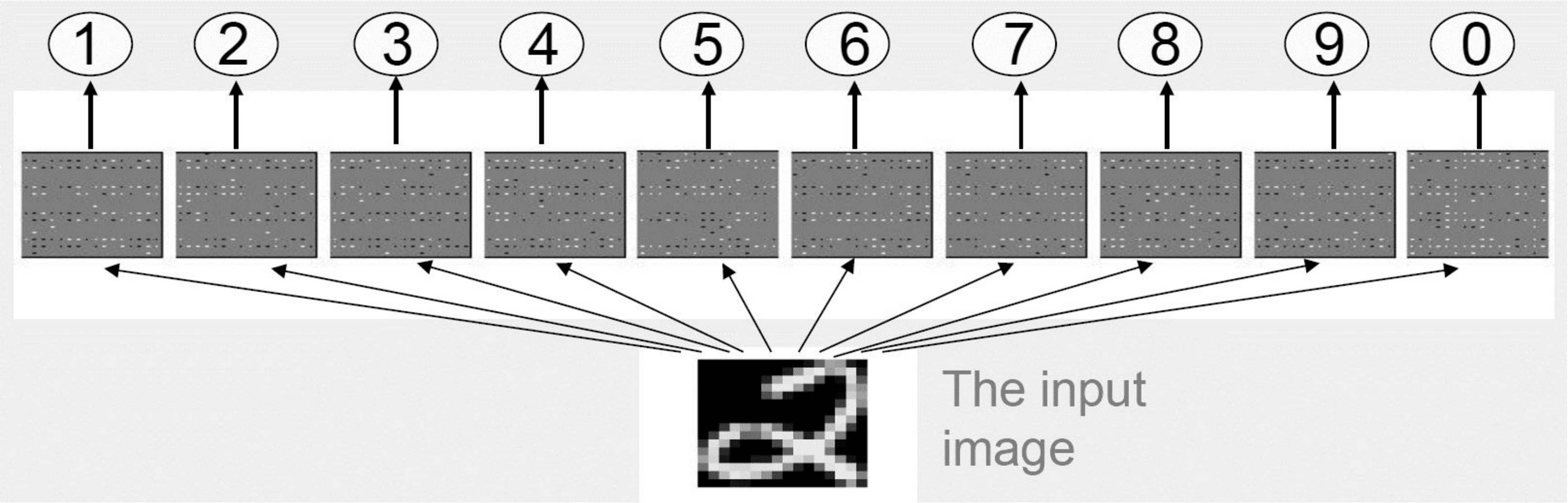
    $E_n(\boldsymbol{w}) = \left(y^{(n)} - \boldsymbol{w}^T \boldsymbol{x}^{(n)}\right)^2$

Perceptron vs. Delta Rule

- Perceptron learning rule:
  - guaranteed to succeed if training examples are linearly separable

- Delta rule:
  - guaranteed to converge to the hypothesis with the minimum squared error
  - succeed if sufficiently small learning rate
    - Even when training data contain noise or are not separable by a hyperplane
  - can also be used for regression problems

How to learn the weights: multi class example

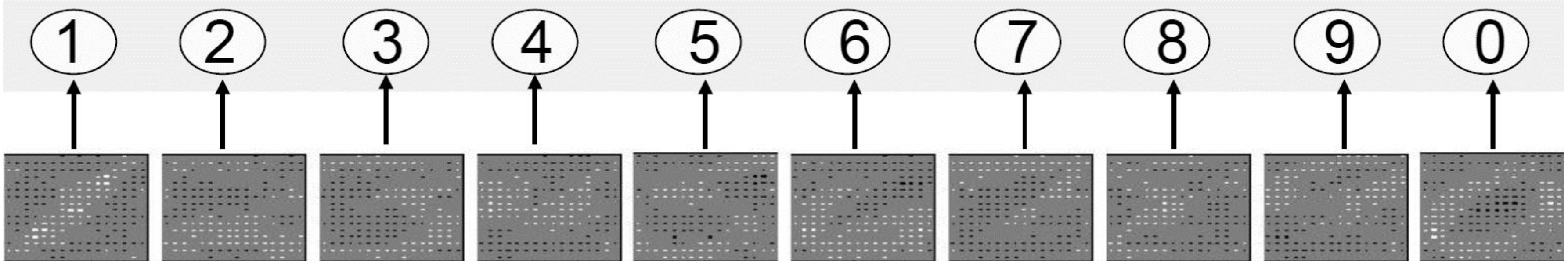


$$W = \begin{bmatrix} b_1 & w_{11} & \cdots & w_{1d} \\ \vdots & \ddots & \cdots & \vdots \\ b_k & w_{k1} & \cdots & w_{kd} \end{bmatrix} \quad \boldsymbol{x} = \begin{bmatrix} 1 \\ x_1 \\ x_2 \\ \vdots \\ x_d \end{bmatrix}$$

This slide has been adopted from Hinton’s lectures, “NN for Machine Learning” course, 2015.

How to learn the weights: multi class example

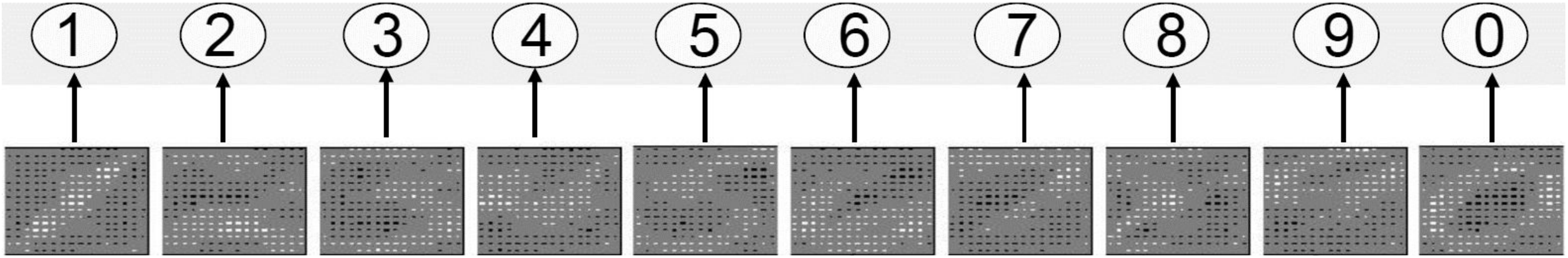
- If correct: no change
- If wrong:
  - lower score of the wrong answer (by removing the input from the weight vector of the wrong answer)
  - raise score of the target (by adding the input to the weight vector of the target class)



This slide has been adopted from Hinton's lectures, "NN for Machine Learning" course, 2015.

How to learn the weights: multi class example

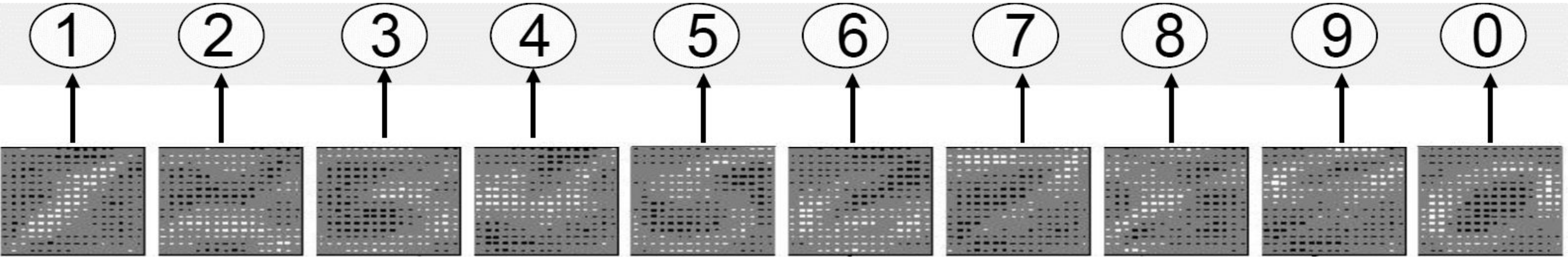
- If correct: no change
- If wrong:
  - lower score of the wrong answer (by removing the input from the weight vector of the wrong answer)
  - raise score of the target (by adding the input to the weight vector of the target class)



This slide has been adopted from Hinton’s lectures, “NN for Machine Learning” course, 2015.

How to learn the weights: multi class example

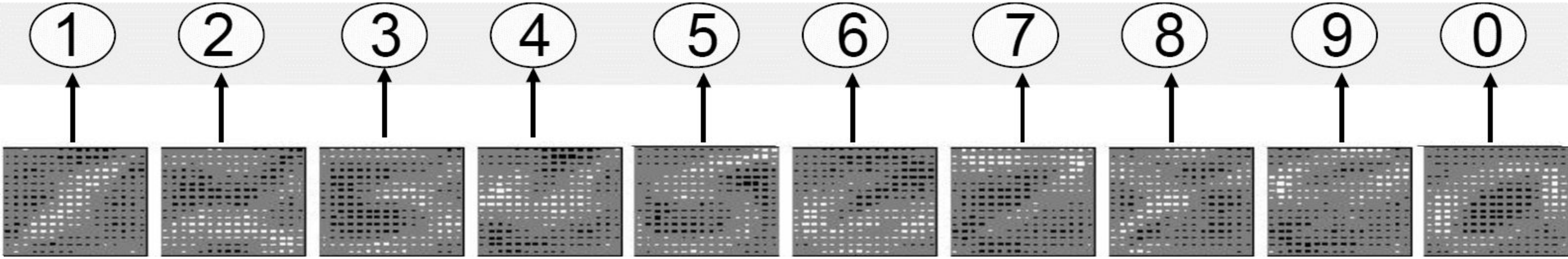
- If correct: no change
- If wrong:
  - lower score of the wrong answer (by removing the input from the weight vector of the wrong answer)
  - raise score of the target (by adding the input to the weight vector of the target class)



This slide has been adopted from Hinton’s lectures, “NN for Machine Learning” course, 2015.

How to learn the weights: multi class example

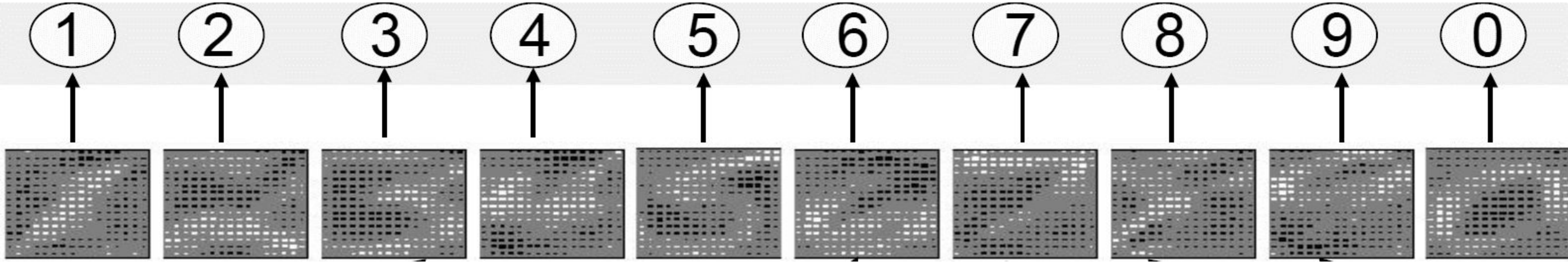
- If correct: no change
- If wrong:
  - lower score of the wrong answer (by removing the input from the weight vector of the wrong answer)
  - raise score of the target (by adding the input to the weight vector of the target class)



This slide has been adopted from Hinton’s lectures, “NN for Machine Learning” course, 2015.

How to learn the weights: multi class example

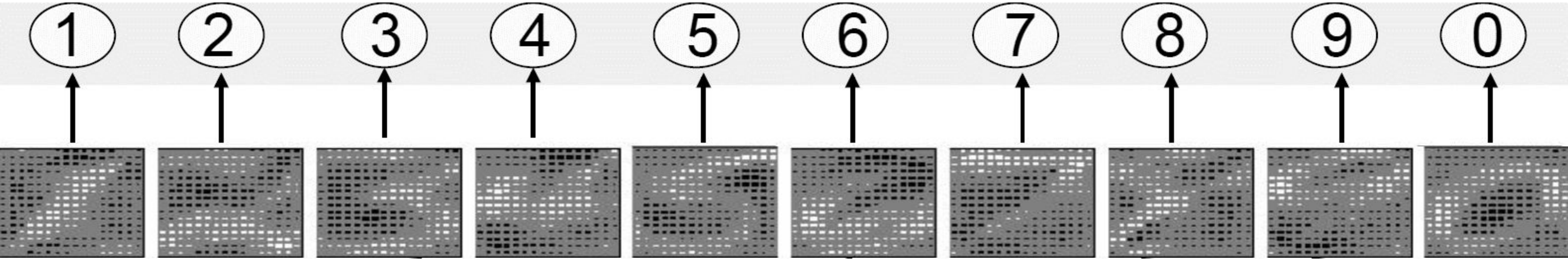
- If correct: no change
- If wrong:
  - lower score of the wrong answer (by removing the input from the weight vector of the wrong answer)
  - raise score of the target (by adding the input to the weight vector of the target class)



This slide has been adopted from Hinton’s lectures, “NN for Machine Learning” course, 2015.

How to learn the weights: multi class example

- If correct: no change
- If wrong:
  - lower score of the wrong answer (by removing the input from the weight vector of the wrong answer)
  - raise score of the target (by adding the input to the weight vector of the target class)



This slide has been adopted from Hinton's lectures, "NN for Machine Learning" course, 2015.

Limitation of single layer network

- Single layer networks is equivalent to template matching
  - Weights for each class as a template for that class.

- The ways in which a digit can be written are much too complicated to be captured by simple template

- Thus, networks without hidden units are very limited in the mappings that they can learn

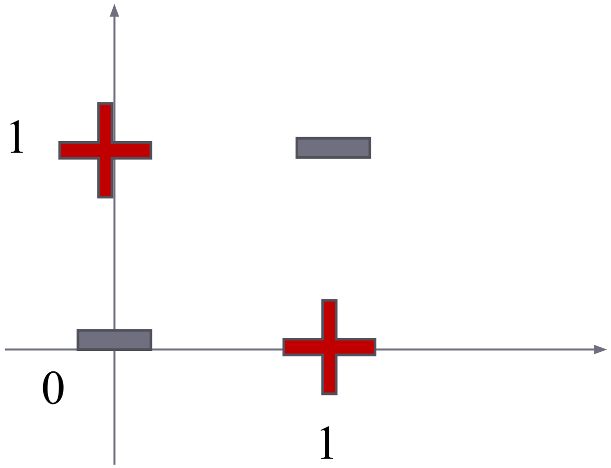
The history of Perceptron

- They were popularized by Frank Rosenblatt in the early 1960’s.
  - They appeared to have a very powerful learning algorithm.
  - Lots of grand claims were made for what they could learn to do.

- In 1969, Minsky and Papert published a book called “Perceptrons” that analyzed what they could do and showed their limitations.
  - Many people thought these limitations applied to all neural network models.

What binary threshold neurons cannot do

- A binary threshold output unit cannot even tell if two single bit features are the same!
- A geometric view of what binary threshold neurons cannot do
- The positive and negative cases cannot be separated by a plane

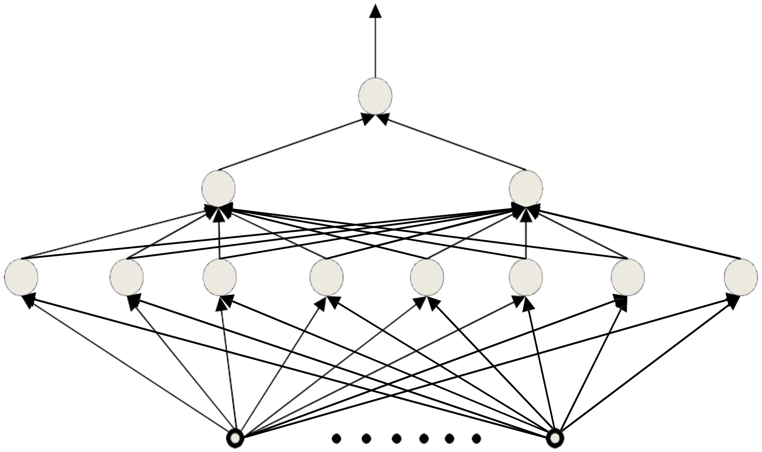


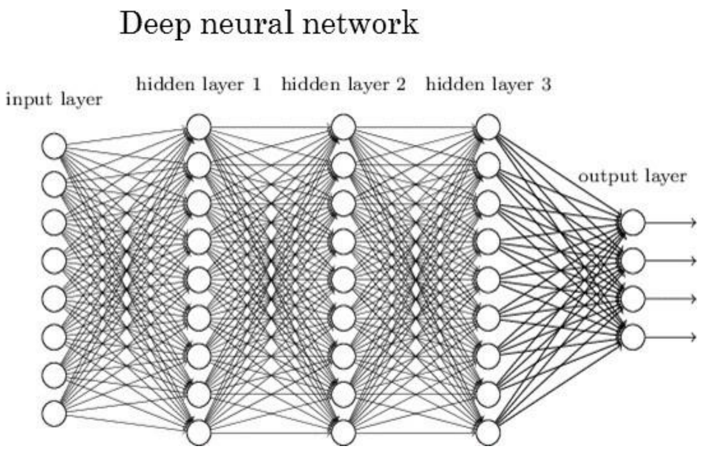
Networks with hidden units

- Networks without hidden units are very limited in the input-output mappings they can learn to model.
    - More layers of linear units do not help. Its still linear.
    - Fixed output non-linearities are not enough.

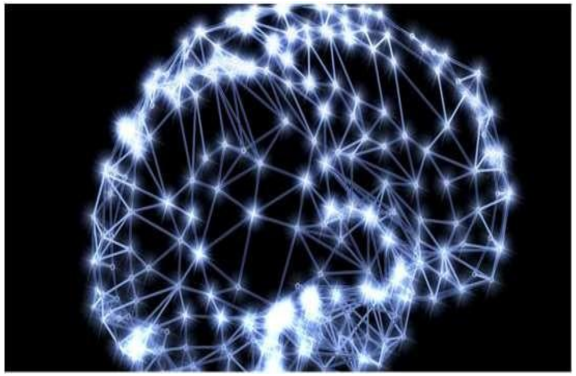
- We need multiple layers of adaptive, non-linear hidden units. But how can we train such nets?

The multi-layer perceptron

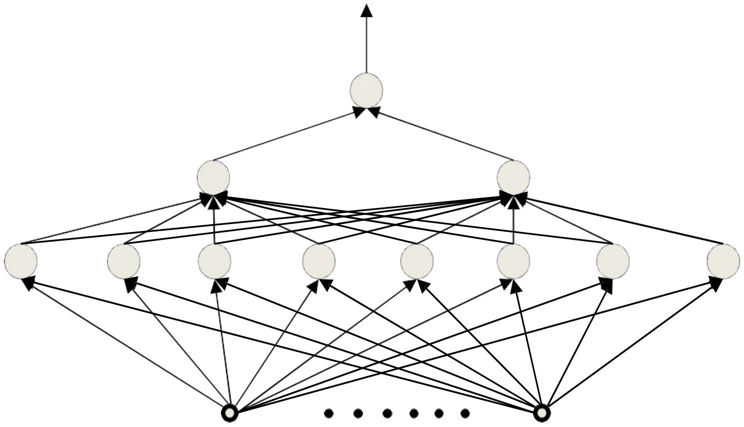


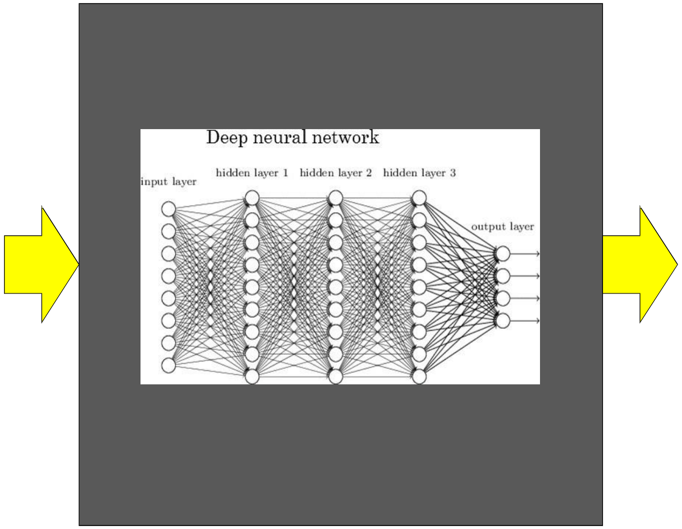


- A network of perceptrons
  - Generally “layered”



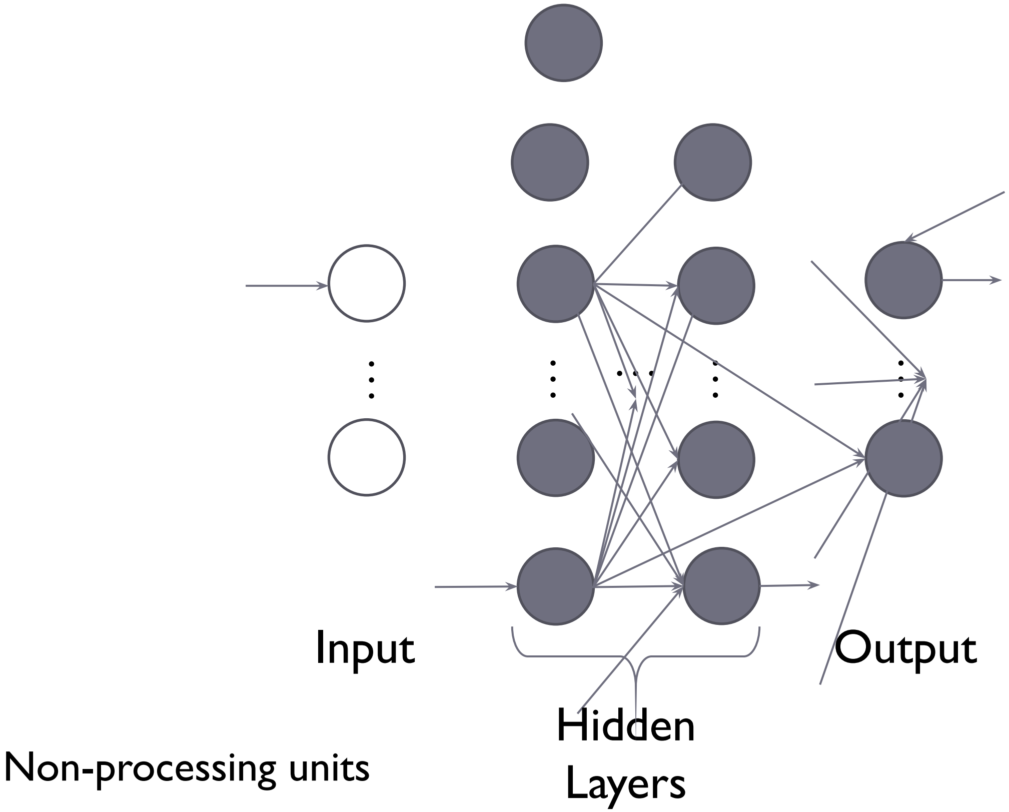
The multi-layer perceptron





- Inputs are real or Boolean stimuli
- Outputs are real or Boolean values
  - Can have multiple outputs for a single input
- What can this network compute?
  - What kinds of input/output relationships can it model?

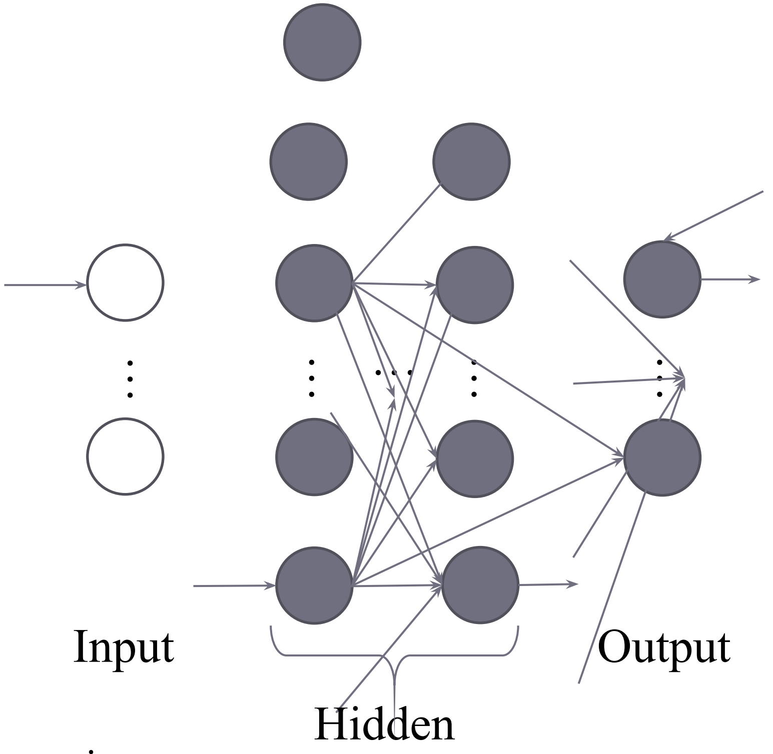
Feed-forward neural networks



Weights on links can be adapted using training data and a learning algorithm

Feed-forward Neural Networks

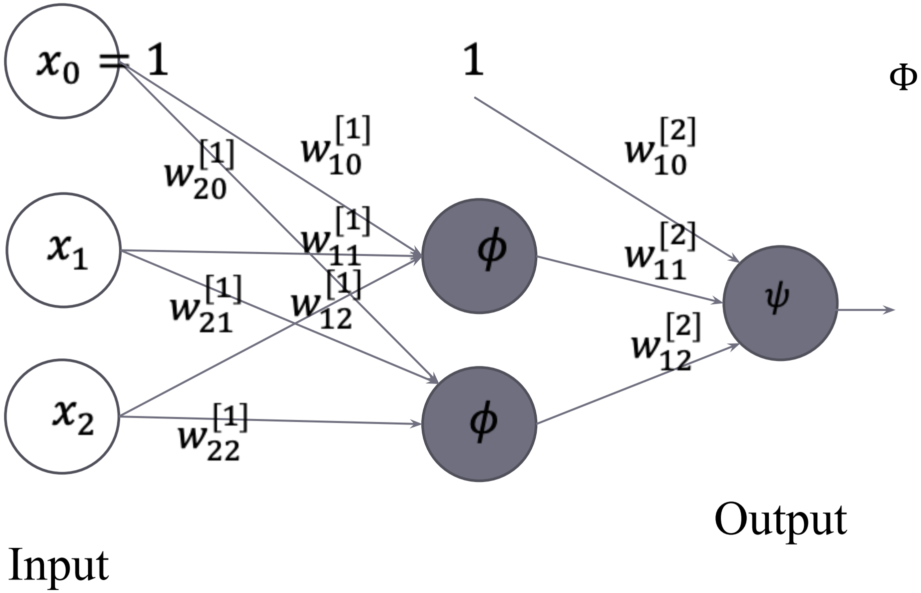
- We need multiple layers of adaptive, non-linear hidden units.
  - Also called **Multi-Layer Perceptron (MLP)**
- Each *unit* takes some inputs and produces one output.
  - Output of one unit can be the input of a next layer(s) unit.



Weights on links can be adapted using training data and a learning algorithm

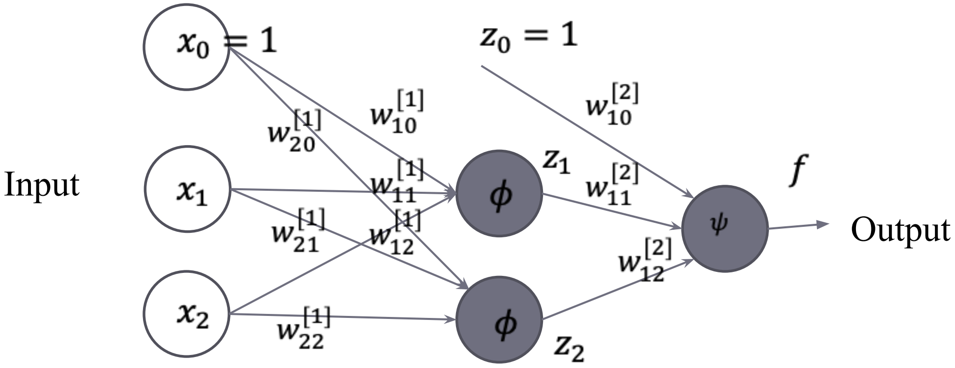
Non-processing units

Multi-layer Neural Network



Φ: activation function

Multi-layer Neural Network



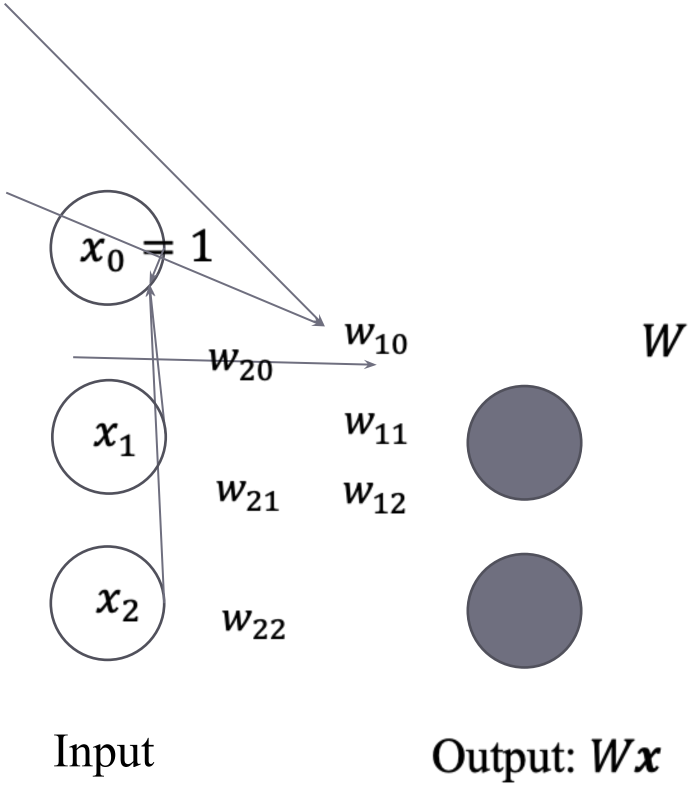
Examples:

$$\phi(z) = \max(0, z)$$

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

$$z_j = \phi\left(\sum_{i=0}^{d} w_{ji}^{[1]} x_i\right) \qquad f = \psi\left(\sum_{j=0}^{2} w_{kj}^{[2]} z_j\right) = \left(\sum_{j=0}^{2} w_{kj}^{[2]} \phi\left(\sum_{i=0}^{d} w_{ji}^{[1]} x_i\right)\right)$$

Linear Model

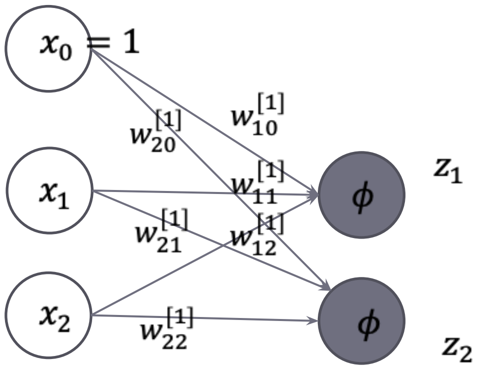


$$W = \begin{bmatrix} w_{10} & w_{11} & \cdots & w_{1d} \\ \vdots & \ddots & \cdots & \vdots \\ w_{k0} & w_{k1} & \cdots & w_{kd} \end{bmatrix} \quad \boldsymbol{x} = \begin{bmatrix} 1 \\ x_1 \\ x_2 \\ \vdots \\ x_d \end{bmatrix}$$

Multi-layer Neural Network

$$\boldsymbol{W}^{[1]} = \begin{bmatrix} w_{10}^{[1]} & w_{11}^{[1]} & w_{12}^{[1]} \\ w_{20}^{[1]} & w_{21}^{[1]} & w_{22}^{[1]} \end{bmatrix}$$

Input



$$z_j = \phi\left(\sum_{i=0}^{d} w_{ji}^{[1]} x_i\right)$$

Matrix form:

$$\boldsymbol{z} = \phi\left(W^{[1]}\boldsymbol{x}\right)$$

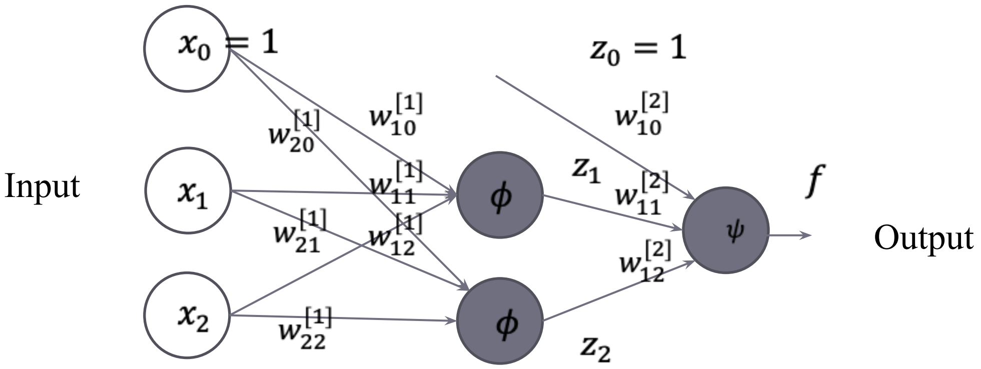
Examples:

$$\phi(z) = \max(0, z)$$

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

Multi-layer Neural Network

$$\boldsymbol{W}^{[1]} = \begin{bmatrix} w_{10}^{[1]} & w_{11}^{[1]} & w_{12}^{[1]} \\ w_{20}^{[1]} & w_{21}^{[1]} & w_{22}^{[1]} \end{bmatrix}$$



Examples:

$$\phi(z) = \max(0, z)$$

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

$$z_j = \phi\left(\sum_{i=0}^{d} w_{ji}^{[1]} x_i\right)$$

$$f = \psi\left(\sum_{j=0}^{2} w_{kj}^{[2]} z_j\right) = \left(\sum_{j=0}^{2} w_{kj}^{[2]} \phi\left(\sum_{i=0}^{d} w_{ji}^{[1]} x_i\right)\right)$$

Matrix form:

$$\boldsymbol{z} = \phi\left(W^{[1]}\boldsymbol{x}\right)$$

$f($

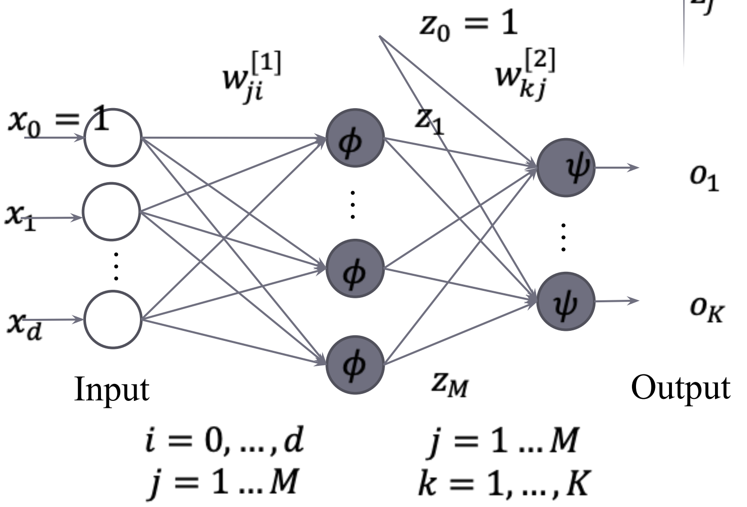
$=$

$=$

MLP With Single Hidden Layer

- Two-layer MLP (Number of layers of adaptive weights is counted)

$$o_k(\boldsymbol{x}) = \psi\left(\sum_{j=0}^{M} w_{kj}^{[2]} z_j\right) \Rightarrow o_k(\boldsymbol{x}) = \psi\left(\sum_{j=0}^{M} w_{kj}^{[2]} \underbrace{\phi\left(\sum_{i=0}^{d} w_{ji}^{[1]} x_i\right)}_{z_j}\right)$$



- Thus, we don't need expert knowledge or time consuming tuning of hand-crafted features
  - The form of the nonlinearity (basis functions f_j) is adapted from the training data

Expressiveness Of Neural Networks

- All Boolean functions can be represented by a network with a single hidden layer
  - But it might require exponential (in number of inputs) hidden units

- Continuous functions:
  - Any continuous function on a compact domain can be approximated to an arbitrary accuracy, by network with one hidden layer [Cybenko 1989]
  - Any function can be approximated to an arbitrarily accuracy by a network with two hidden layers [Cybenko 1988]

MLP Universal Approximator

- A feed-forward network with a single hidden layer and linear outputs can approximate any continuous function on a compact domain to an arbitrary accuracy
  - under mild assumptions on the activation function
    - e.g., sigmoid activation functions (Cybenko,1989)
  - when sufficiently large (but finite) number of hidden units is used

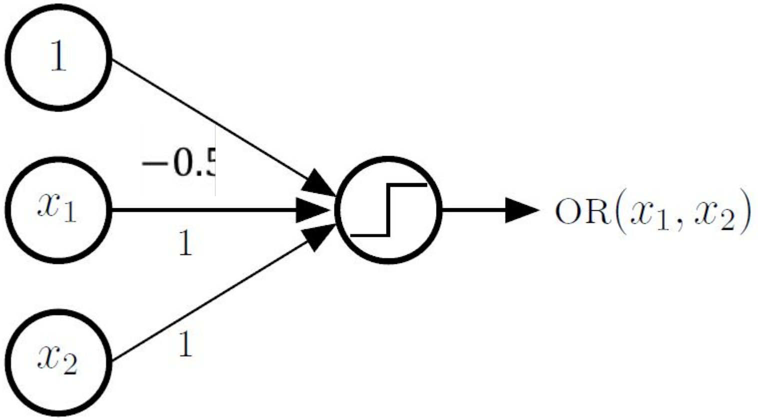
$$F_k(x) = \sum_{j=1}^{M} w_{kj}^{[2]} \, \phi\left(\sum_{i=0}^{d} w_{ji}^{[1]} x_i\right)$$

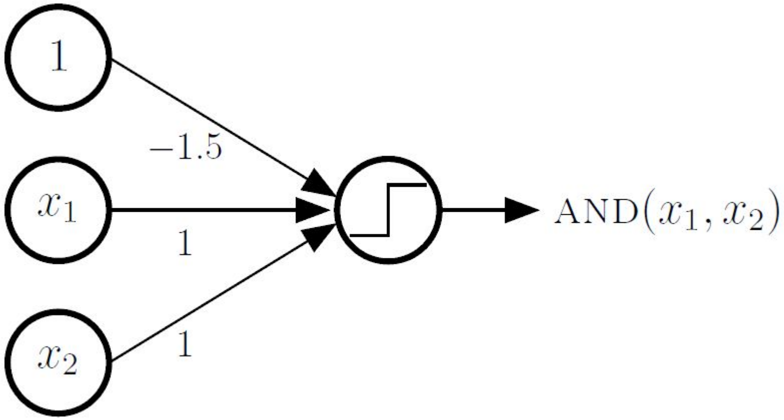
- It is of greater theoretical interest than practical
  - the construction of such a network requires the nonlinear activation functions and weight values which are unknown

MLPs approximate functions

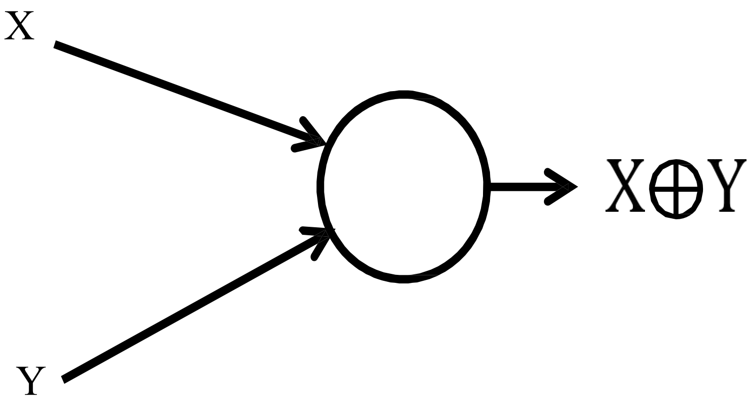
- MLP s can compose Boolean functions
- MLPs as universal classifiers
- MLPs as universal approximators (of real-valued functions)

AND & OR networks





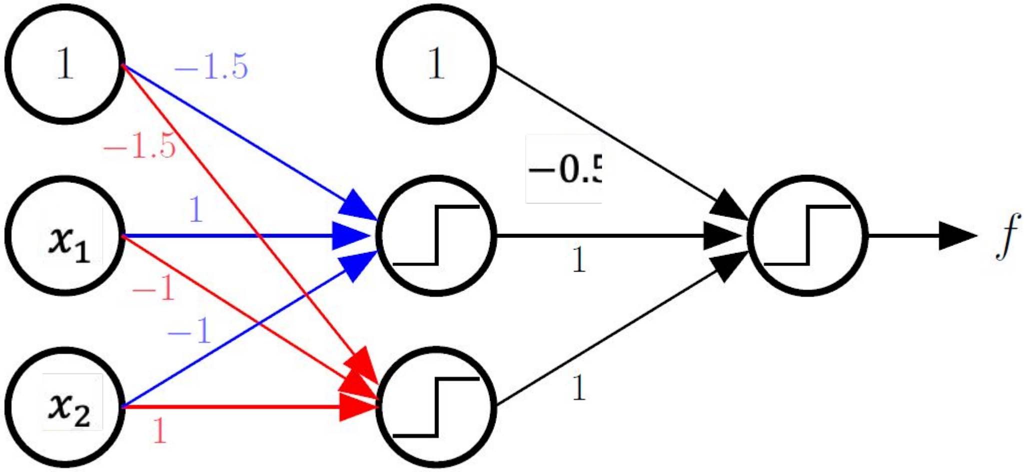
The perceptron is not enough



- Cannot compute an XOR

XOR example

- $$f = XOR(x_1, x_2) = OR\big(AND(x_1, \bar{x}_2), AND(\bar{x}_1, x_2)\big)$$



Input variables that are True are considered as 1 and False ones as -1

General Boolean functions

- Every Boolean function can be represented by a network with a single hidden layer

1. Consider the truth table of the Boolean function
2. Write Boolean function as OR of ANDs, with one AND for each positive entry in the truth table.
3. Construct a 2-layer network that is composed of OR of ANDs (first layer contains ANDs and second layer contains OR)

- It might need an exponential number of hidden units

How Many Layers For A Boolean MLP?

Truth table shows all input combinations for which output is 1

Truth Table

X_1	X_2	X_3	X_4	X_5	Y
0	0	1	1	0	1
0	1	0	1	1	1
0	1	1	0	0	1
1	0	0	0	1	1
1	0	1	1	1	1
1	1	0	0	1	1

$$y = \bar{X}_1\bar{X}_2X_3X_4\bar{X}_5 + \bar{X}_1X_2\bar{X}_3X_4X_5 + \bar{X}_1X_2X_3\bar{X}_4\bar{X}_5 + X_1\bar{X}_2\bar{X}_3\bar{X}_4X_5 + X_1\bar{X}_2X_3X_4X_5 + X_1X_2\bar{X}_3\bar{X}_4X_5$$

- Expressed in disjunctive normal form

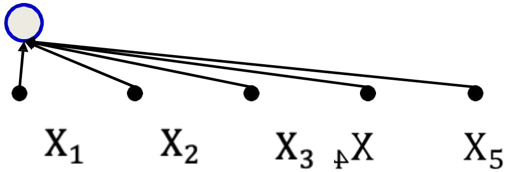
How Many Layers For A Boolean MLP?

Truth table shows all input combinations for which output is 1

Truth Table

X_1	X_2	X_3	X_4	X_5	Y
0	0	1	1	0	1
0	1	0	1	1	1
0	1	1	0	0	1
1	0	0	0	1	1
1	0	1	1	1	1
1	1	0	0	1	1

$$y = \boxed{\bar{X}_1\bar{X}_2X_3X_4\bar{X}_5} + \bar{X}_1X_2\bar{X}_3X_4X_5 + \bar{X}_1X_2X_3\bar{X}_4\bar{X}_5 + X_1\bar{X}_2\bar{X}_3\bar{X}_4X_5 + X_1\bar{X}_2X_3X_4X_5 + X_1X_2\bar{X}_3\bar{X}_4X_5$$



- Expressed in disjunctive normal form

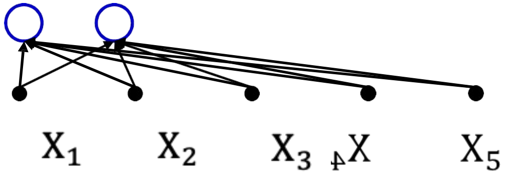
How Many Layers For A Boolean MLP?

Truth table shows all input combinations
for which output is 1

Truth Table

X_1	X_2	X_3	X_4	X_5	Y
0	0	1	1	0	1
0	1	0	1	1	1
0	1	1	0	0	1
1	0	0	0	1	1
1	0	1	1	1	1
1	1	0	0	1	1

$$y = \bar{X}_1\bar{X}_2X_3X_4\bar{X}_5 + \boxed{\bar{X}_1X_2\bar{X}_3X_4X_5} + \bar{X}_1X_2X_3\bar{X}_4\bar{X}_5 + X_1\bar{X}_2\bar{X}_3\bar{X}_4X_5 + X_1\bar{X}_2X_3X_4X_5 + X_1X_2\bar{X}_3\bar{X}_4X_5$$



- Expressed in disjunctive normal form

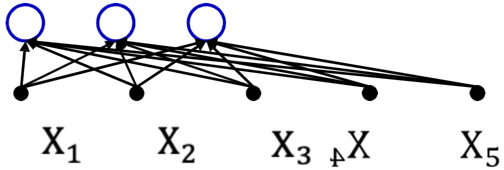
How Many Layers For A Boolean MLP?

Truth table shows all input combinations for which output is 1

Truth Table

X_1	X_2	X_3	X_4	X_5	Y
0	0	1	1	0	1
0	1	0	1	1	1
0	1	1	0	0	1
1	0	0	0	1	1
1	0	1	1	1	1
1	1	0	0	1	1

$$y = \bar{X}_1\bar{X}_2X_3X_4\bar{X}_5 + \bar{X}_1X_2\bar{X}_3X_4X_5 \boxed{+\bar{X}_1X_2X_3\bar{X}_4\bar{X}_5+} X_1\bar{X}_2\bar{X}_3\bar{X}_4X_5 + X_1\bar{X}_2X_3X_4X_5 + X_1X_2\bar{X}_3\bar{X}_4X_5$$



- Expressed in disjunctive normal form

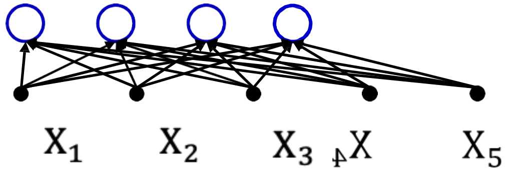
How Many Layers For A Boolean MLP?

Truth table shows all input combinations for which output is 1

Truth Table

X_1	X_2	X_3	X_4	X_5	Y
0	0	1	1	0	1
0	1	0	1	1	1
0	1	1	0	0	1
1	0	0	0	1	1
1	0	1	1	1	1
1	1	0	0	1	1

$$y = \bar{X}_1\bar{X}_2X_3X_4\bar{X}_5 + \bar{X}_1X_2\bar{X}_3X_4X_5 + \bar{X}_1X_2X_3\bar{X}_4\bar{X}_5 + \boxed{X_1\bar{X}_2\bar{X}_3\bar{X}_4X_5} + X_1\bar{X}_2X_3X_4X_5 + X_1X_2\bar{X}_3\bar{X}_4X_5$$



- Expressed in disjunctive normal form

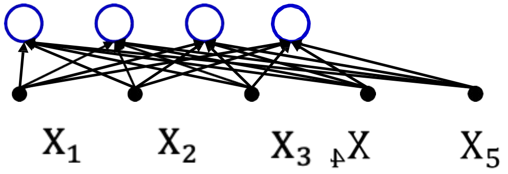
How Many Layers For A Boolean MLP?

Truth Table

X_1	X_2	X_3	X_4	X_5	Y
0	0	1	1	0	1
0	1	0	1	1	1
0	1	1	0	0	1
1	0	0	0	1	1
1	0	1	1	1	1
1	1	0	0	1	1

Truth table shows all input combinations for which output is 1

$$y = \bar{X}_1\bar{X}_2X_3X_4\bar{X}_5 + \bar{X}_1X_2\bar{X}_3X_4X_5 + \bar{X}_1X_2X_3\bar{X}_4\bar{X}_5 + \\ X_1\bar{X}_2\bar{X}_3\bar{X}_4X_5 + \boxed{X_1\bar{X}_2X_3X_4X_5} + X_1X_2\bar{X}_3\bar{X}_4X_5$$



- Expressed in disjunctive normal form

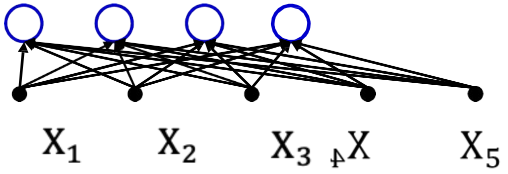
How many layers for a Boolean MLP?

Truth table shows all input combinations for which output is 1

Truth Table

X_1	X_2	X_3	X_4	X_5	Y
0	0	1	1	0	1
0	1	0	1	1	1
0	1	1	0	0	1
1	0	0	0	1	1
1	0	1	1	1	1
1	1	0	0	1	1

$$y = \bar{X}_1\bar{X}_2X_3X_4\bar{X}_5 + \bar{X}_1X_2\bar{X}_3X_4X_5 + \bar{X}_1X_2X_3\bar{X}_4\bar{X}_5 + X_1\bar{X}_2\bar{X}_3\bar{X}_4X_5 + X_1\bar{X}_2X_3X_4X_5 + \boxed{X_1X_2\bar{X}_3\bar{X}_4X_5}$$



- Expressed in disjunctive normal form

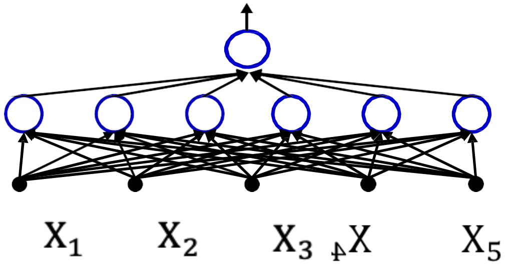
How many layers for a Boolean MLP?

Truth Table

X_1	X_2	X_3	X_4	X_5	Y
0	0	1	1	0	1
0	1	0	1	1	1
0	1	1	0	0	1
1	0	0	0	1	1
1	0	1	1	1	1
1	1	0	0	1	1

Truth table shows all input combinations for which output is 1

$$y = \bar{X}_1\bar{X}_2X_3X_4\bar{X}_5 + \bar{X}_1X_2\bar{X}_3X_4X_5 + \bar{X}_1X_2X_3\bar{X}_4\bar{X}_5 + X_1\bar{X}_2\bar{X}_3\bar{X}_4X_5 + X_1\bar{X}_2X_3X_4X_5 + X_1X_2\bar{X}_3\bar{X}_4X_5$$



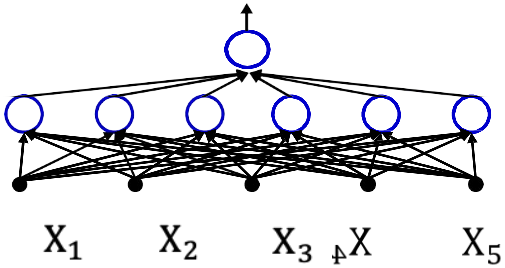
- Expressed in disjunctive normal form

How Many Layers For A Boolean MLP?

Truth Table

X_1	X_2	X_3	X_4	X_5	Y
0	0	1	1	0	1
0	1	0	1	1	1
0	1	1	0	0	1
1	0	0	0	1	1
1	0	1	1	1	1
1	1	0	0	1	1

$$y = \bar{X}_1\bar{X}_2X_3X_4\bar{X}_5 + \bar{X}_1X_2\bar{X}_3X_4X_5 + \bar{X}_1X_2X_3\bar{X}_4\bar{X}_5 + X_1\bar{X}_2\bar{X}_3\bar{X}_4X_5 + X_1\bar{X}_2X_3X_4X_5 + X_1X_2\bar{X}_3\bar{X}_4X_5$$

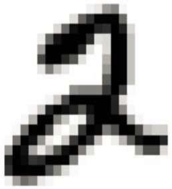


- Any truth table can be expressed in this manner!
- A one-hidden-layer MLP is a Universal Boolean Function
- But what is the largest number of perceptrons required in the single hidden layer for an N-input-variable function?

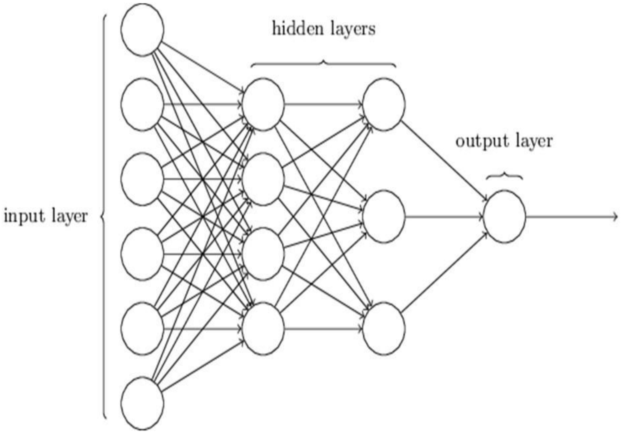
Mlps Approximate Functions

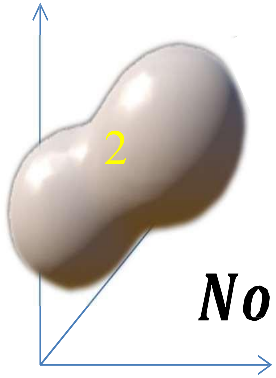
- MLP s can compose Boolean functions
- MLPs as universal classifiers
- MLPs as universal approximators (of real-valued functions)

The MLP As A Classifier



784 dimensions
(MNIST)

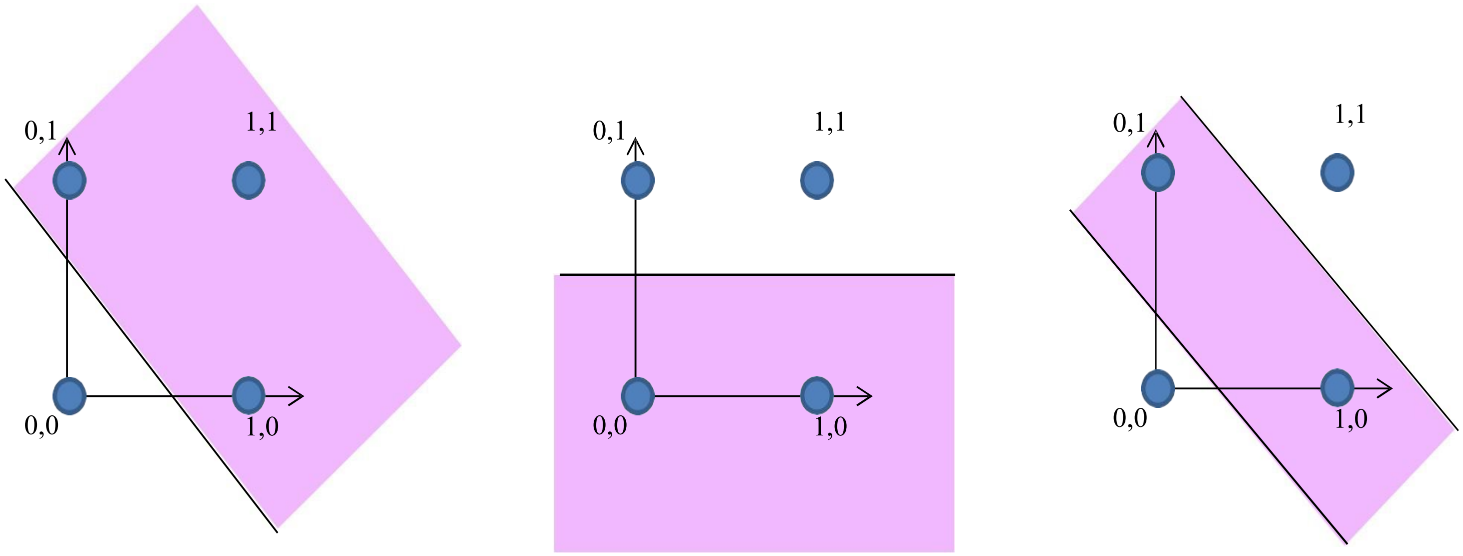




784 dimensions

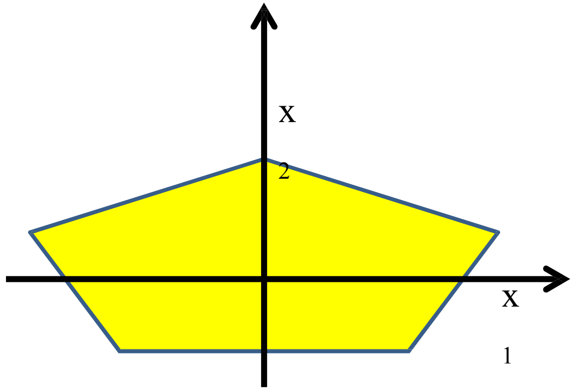
- MLP as a function over real inputs
- MLP as a function that finds a complex "decision boundary" over a space of reals

Boolean Functions With A Real Perceptron



- Boolean perceptrons are also linear classifiers
  - Purple regions are 1

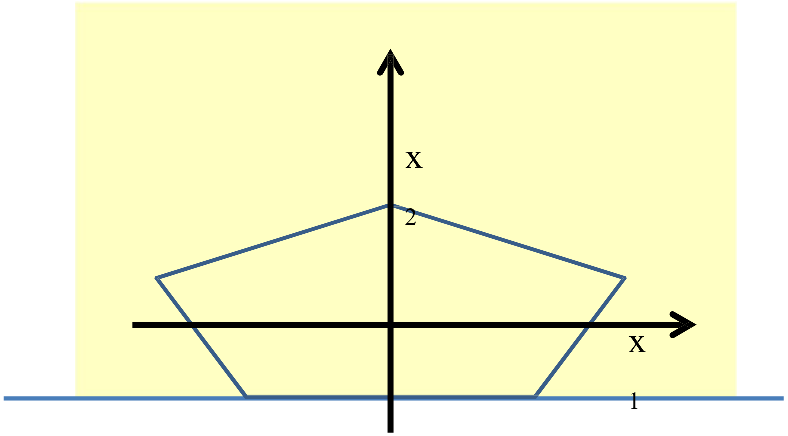
Composing complicated "decision" boundaries

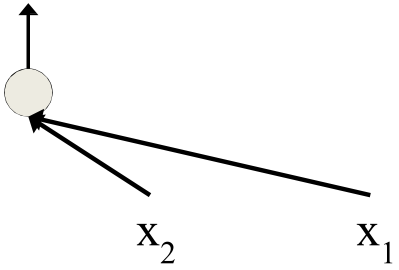


Can now be composed into "networks" to compute arbitrary classification "boundaries"

- Build a network of units with a single output that fires if the input is in the coloured area

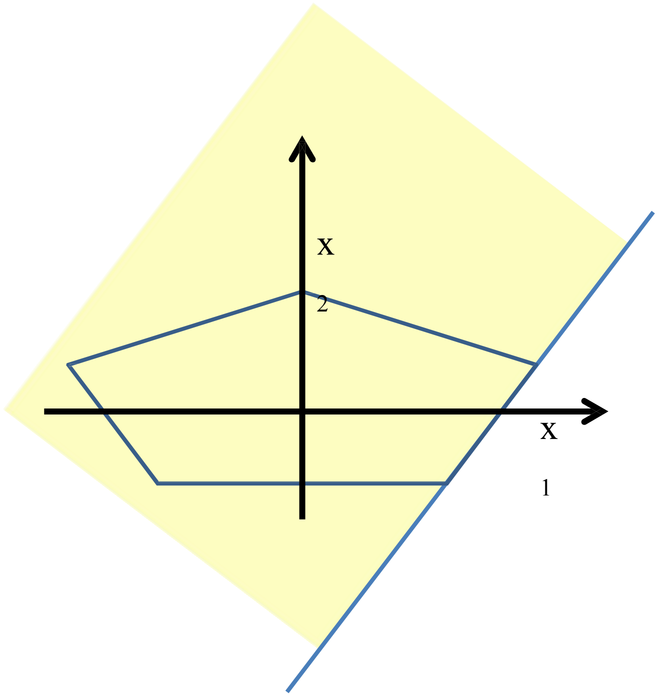
Booleans Over The Reals

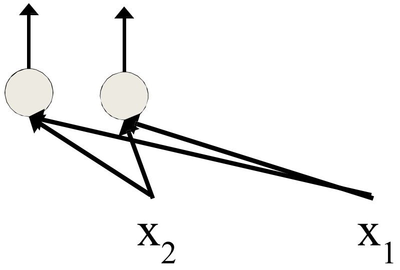




- The network must fire if the input is in the coloured area

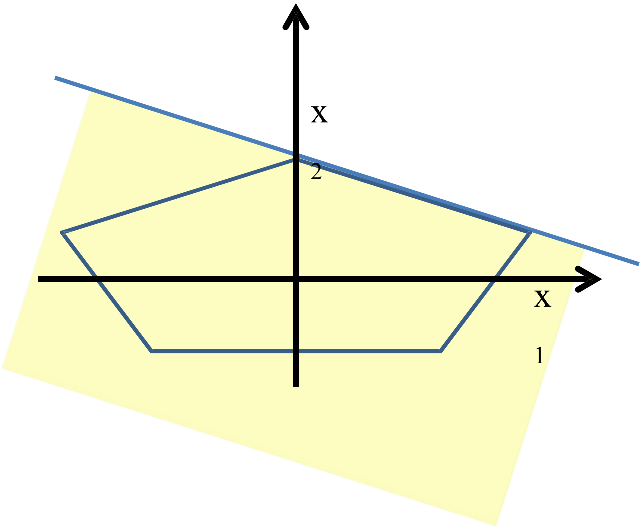
Booleans Over The Reals

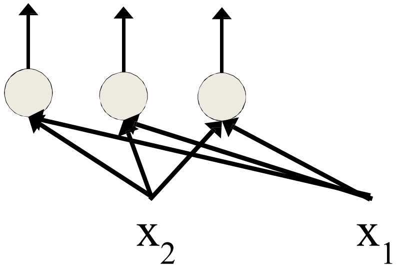




- The network must fire if the input is in the coloured area

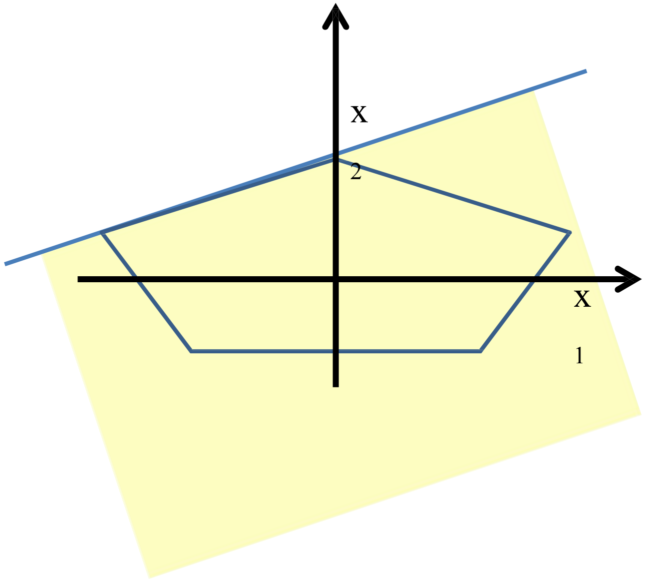
Booleans Over The Reals

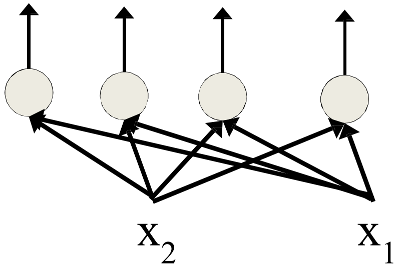




- The network must fire if the input is in the coloured area

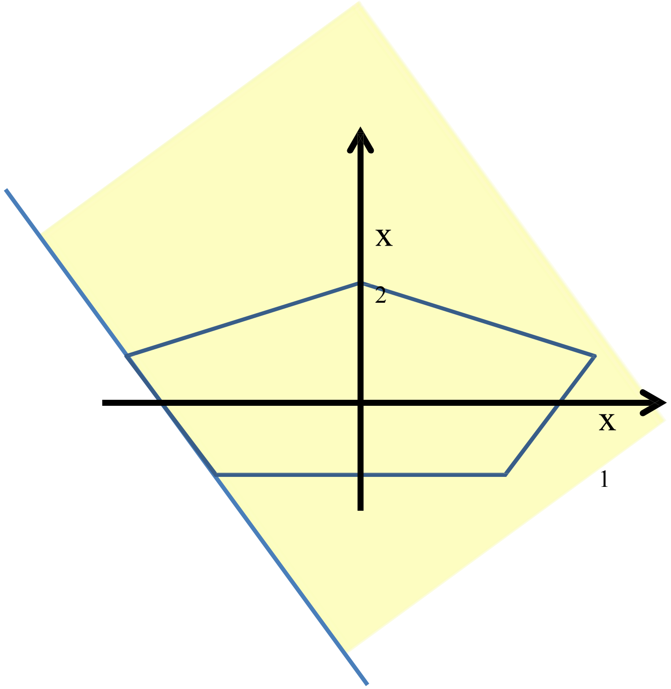
Booleans Over The Reals

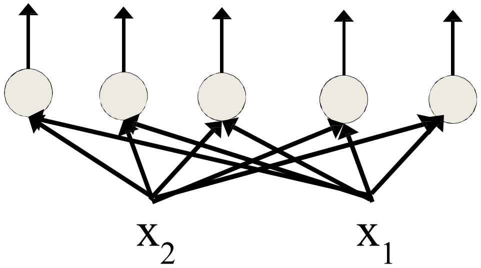




- The network must fire if the input is in the coloured area

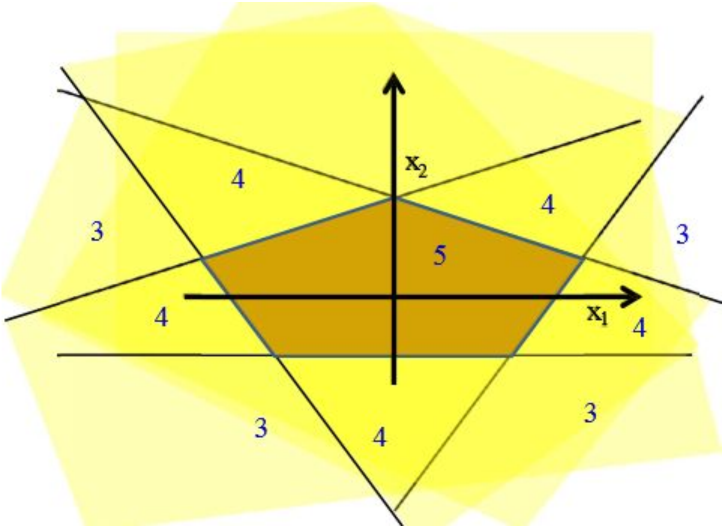
Booleans Over The Reals

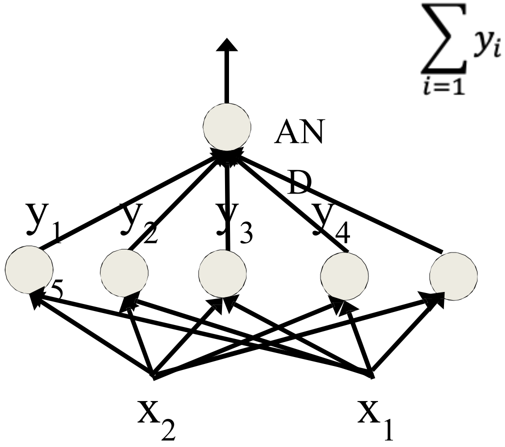




- The network must fire if the input is in the coloured area

Booleans over the reals

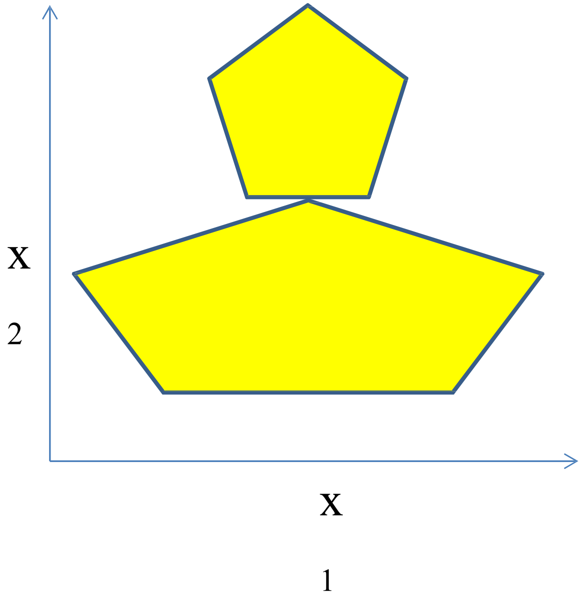


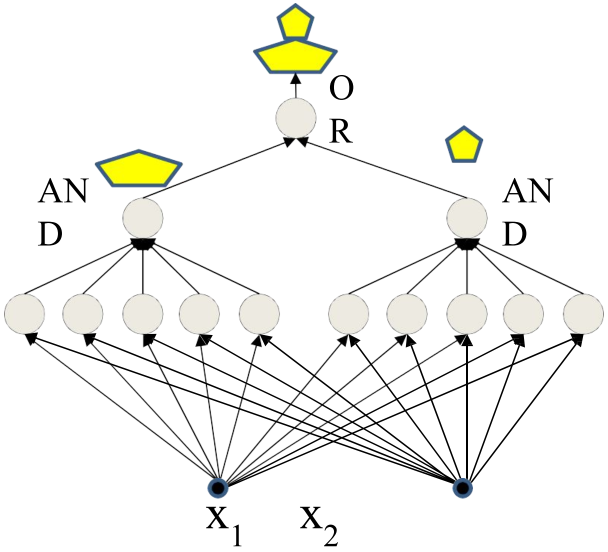


$$\sum_{i=1}^{N} y_i \geq 4.5$$

- The network must fire if the input is in the coloured area

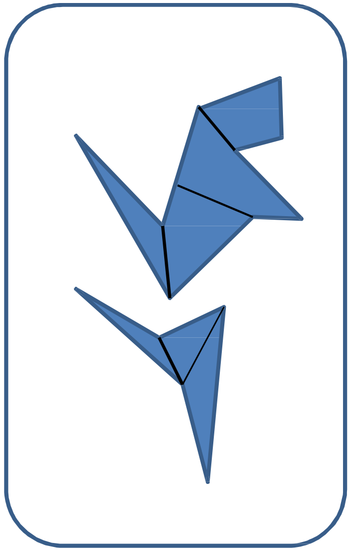
More complex decision boundaries

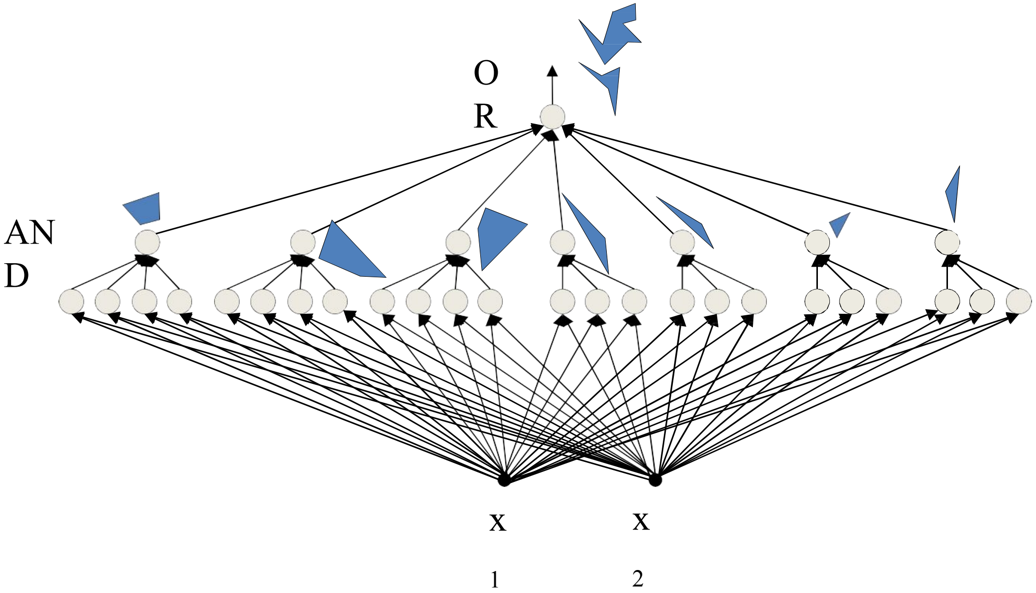




- Network to fire if the input is in the yellow area
  - "OR" two polygons
  - A third layer is required

Complex decision boundaries



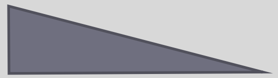
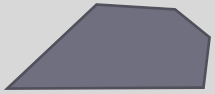
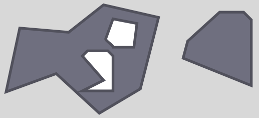


- Can compose arbitrarily complex decision boundaries
  - With only one hidden layer!
  - How?

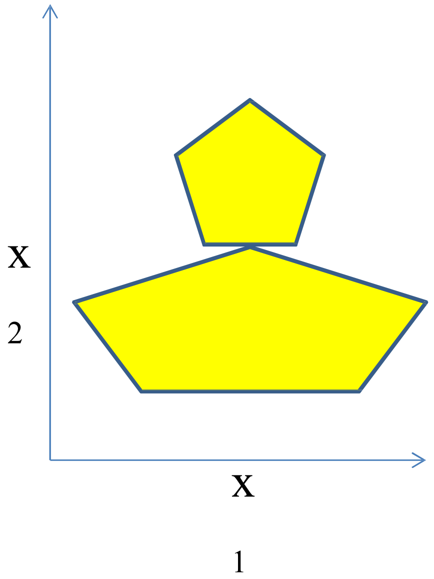
MLP With Different Number Of Layers

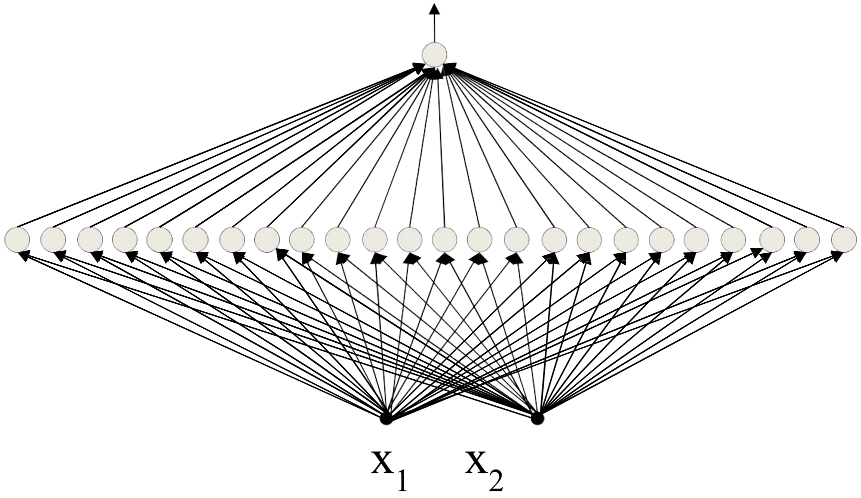
MLP with unit step activation function

Decision region found by an output unit.

Structure	Type of Decision Regions	Interpretation	Example of region
Single Layer (no hidden layer)	Half space	Region found by a hyper-plane	
Two Layer (one hidden layer)	Polyhedral (open or closed) region	Intersection of half spaces	
Three Layer (two hidden layers)	Arbitrary regions	Union of polyhedrals	

Exercise: Compose This With One Hidden Layer





- How would you compose the decision boundary to the left with only one hidden layer?

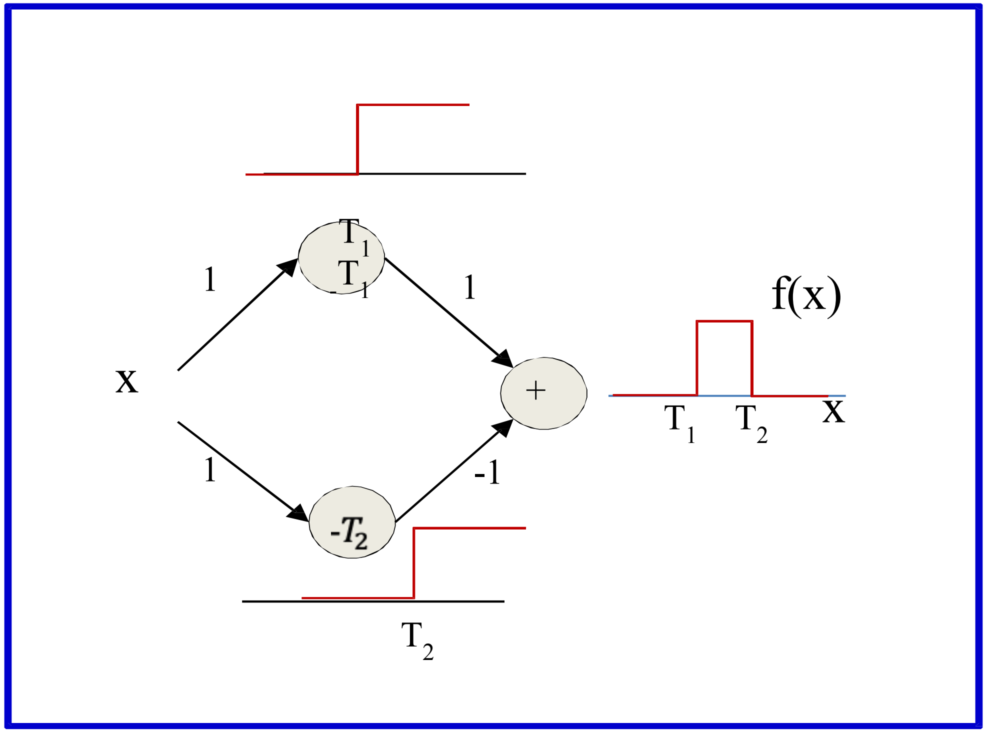
Mlps Approximate Functions

- MLP s can compose Boolean functions

- MLPs as universal classifiers

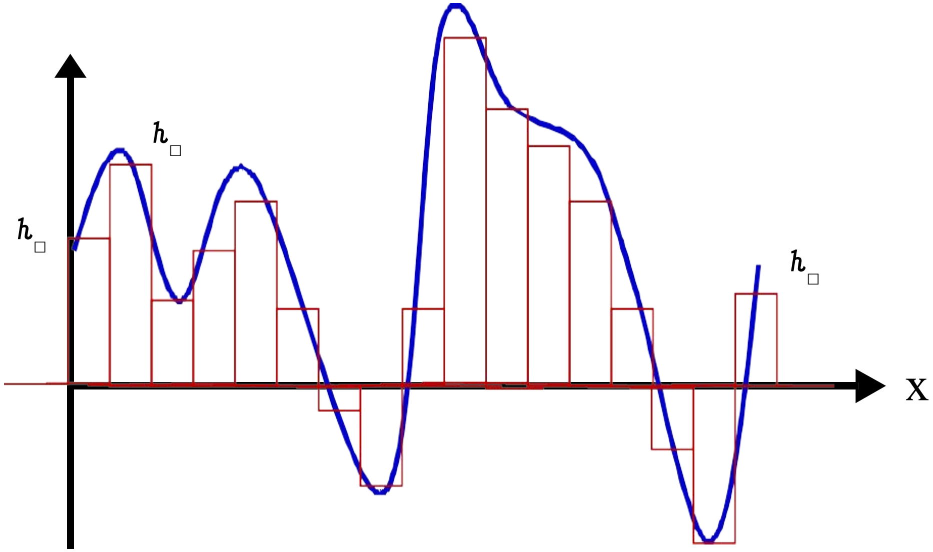
- MLPs as universal approximators (of real-valued functions)

MLP As A Continuous-valued Regression



- A simple 3-unit MLP with a "summing" output unit can generate a "square pulse" over an input
  - Output is 1 only if the input lies between T_1 and T_2
  - T_1 and T_2 can be arbitrarily specified

MLP As A Continuous-valued Regression



- A simple 3-unit MLP can generate a "square pulse" over an input
- An MLP with many units can model an arbitrary function over an input
  - To arbitrary precision
    - Simply make the individual pulses narrower
- A one-layer MLP can model an arbitrary function of a single input

Summary

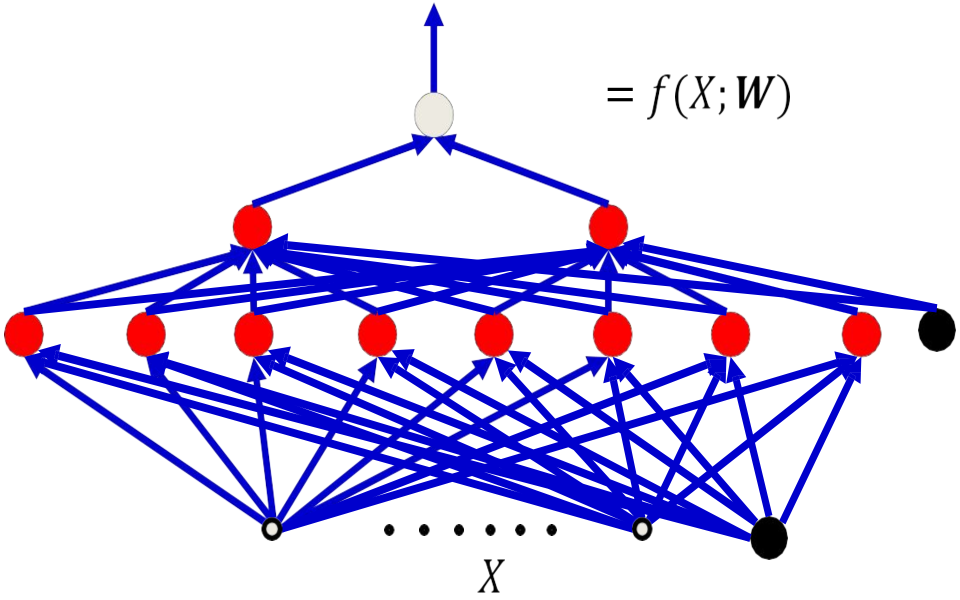
- MLPs are universal Boolean function
- MLPs are universal classifiers
- MLPs are universal function approximators

- An MLP with two (or even one) hidden layers can approximate anything to arbitrary precision
  - But could be exponentially or even infinitely wide in its inputs size

How to adjust weights for multi layer networks?

- How can we train such multi-layer networks?
  - We need to adapt all the weights, not just the last layer.
  - adapting the weights entering hidden units is equivalent to learning features.
    - seems difficult to learn them since the target output of hidden units is not specified (only we access the target output of the whole network).

What we learn : The parameter of the network



- Given: the architecture of the network
- The parameters of the network: The weights and biases
  - The weights associated with the blue arrows in the picture
- Learning the network: Determining the values of these parameters such that the network computes the desired function

History

- 1940s–1960s:
  - development of theories of biological learning
  - implementations of the first models
    - perceptron (Rosenblatt, 1958) for training of a single neuron.

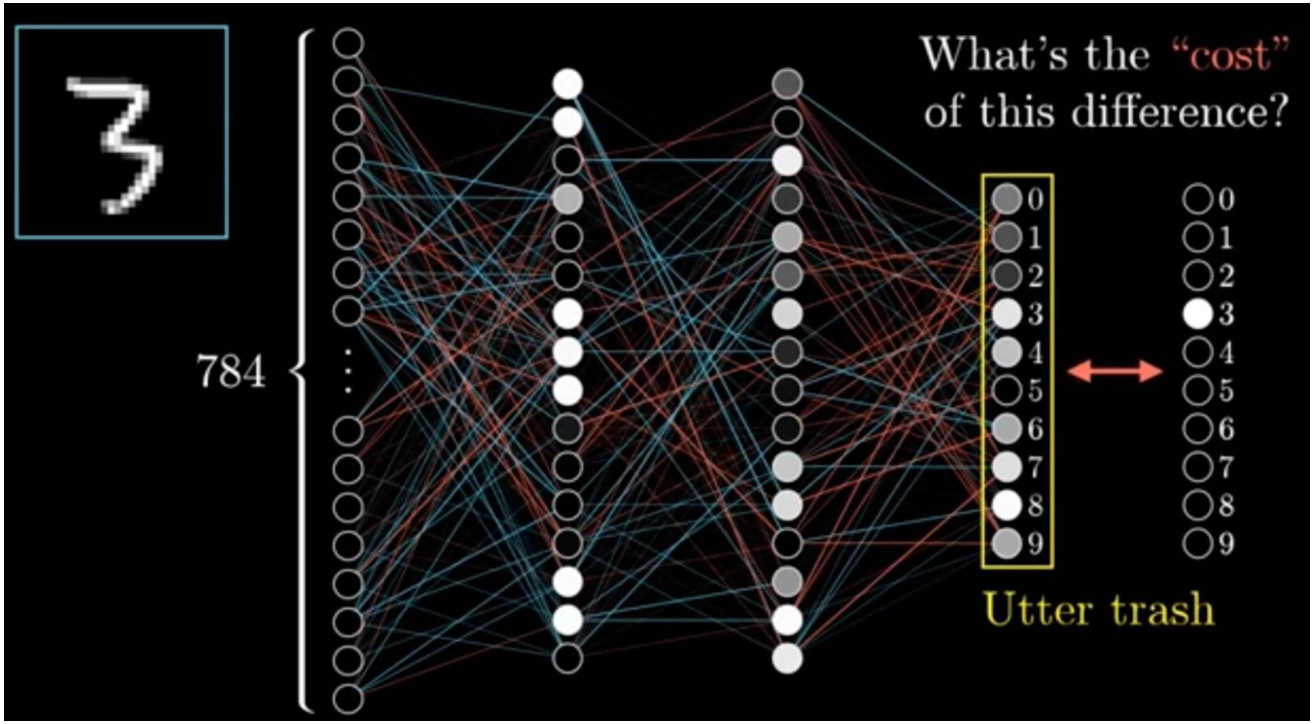
- 1980s-1990s: back-propagation algorithm to train a neural network with more than one hidden layer
  - too computationally costly to allow much experimentation with the hardware available at the time.

Training multi-layer networks

- Back-propagation
  - Training algorithm that is used to adjust weights in multi-layer networks
    - The backpropagation algorithm is based on gradient descent
      - The direction of the most rapid decrease in the cost function
  - Use chain rule to efficiently compute gradients

Find the weights by optimizing the cost

- Start from random weights and then adjust them iteratively to get lower cost.
- Update the weights according to the gradient of the cost function



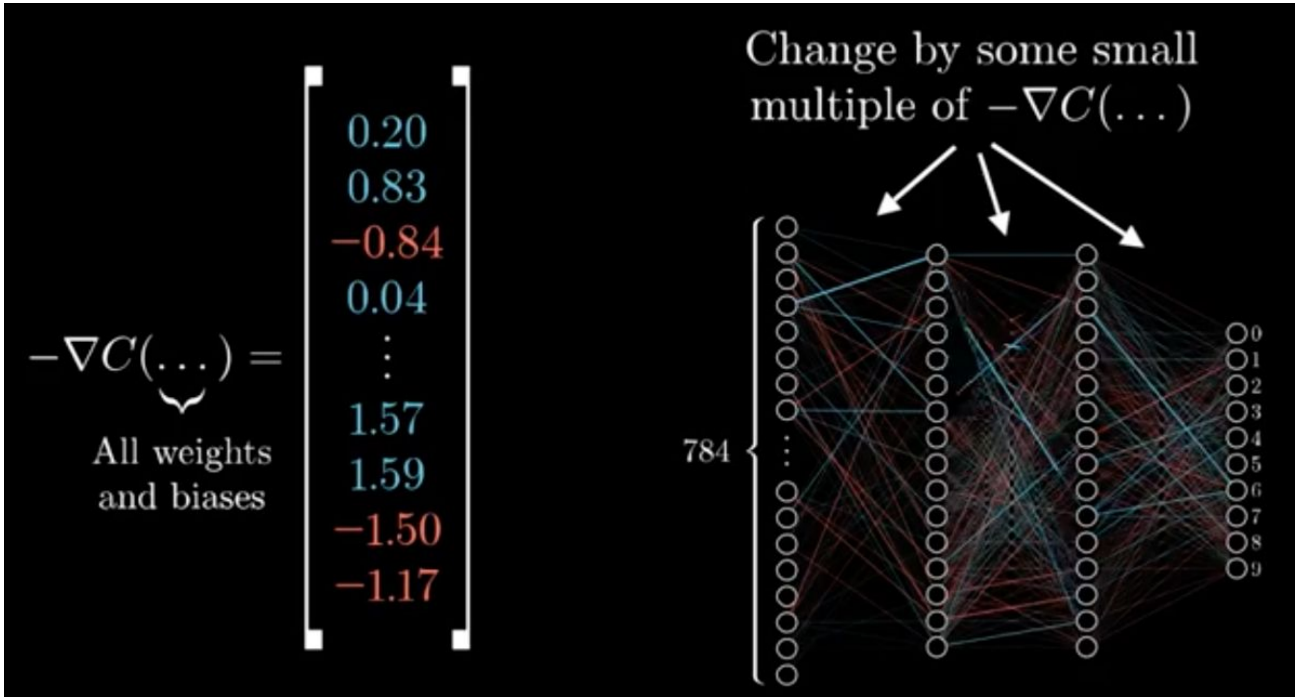
Source:
http://3b1b.co

Choosing cost function: Examples

- Regression problem
  - SSE
- Classification problem
  - Cross-entropy
  - SVM

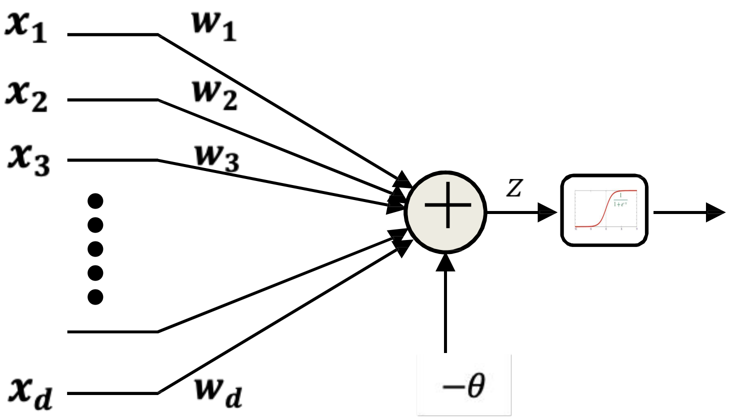
How does the network learn?

- Which changes to the weights do improve the most?
- The magnitude of each element shows how sensitive the cost is to that weight or bias.


$$\underbrace{-\nabla C(\ldots)}_{\text{All weights and biases}} = \begin{bmatrix} 0.20 \\ 0.83 \\ -0.84 \\ 0.04 \\ \vdots \\ 1.57 \\ 1.59 \\ -1.50 \\ -1.17 \end{bmatrix}$$
Change by some small multiple of $-\nabla C(\ldots)$
784
0
1
2
3
4
5
6
7
8
9

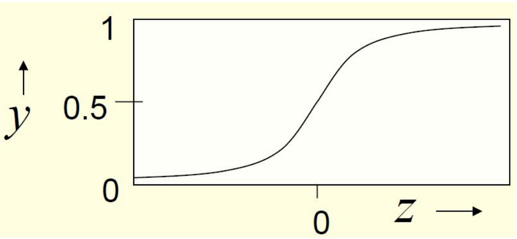
Source: http://3b1b.co

Sigmoid



$$z = \sum_i w_i x_i - \theta$$

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



- A “squashing” function instead of a threshold
  - The sigmoid “activation” replaces the threshold
    - These give a real-valued output that is a smooth and bounded function of their input.
    - They have nice derivatives.

Training Neural Nets through Gradient Descent

Total training error:

$$E = \sum_{n=1}^{N} loss\left(\boldsymbol{o}^{(n)}, \boldsymbol{y}^{(n)}\right)$$

- Gradient descent algorithm
- Initialize all weights and biases $\left\{w_{ji}^{[k]}\right\}$
  - Using the extended notation : the bias is also weight

Assuming the bias is also represented as a weight

- Do :
  - For every layer k for all i, j update:
    - $w_{ji}^{[k]} = w_{ji}^{[k]} - \eta \frac{dE}{dw_{ji}^{[k]}}$
- Until E has converged

The derivative

Total training error:

$$E = \sum_{n=1}^{N} loss\left(\boldsymbol{o}^{(n)}, \boldsymbol{y}^{(n)}\right)$$

Total derivative:

$$\frac{dE}{dw_{ji}^{[k]}} = \sum_{n=1}^{N} \frac{loss\left(\boldsymbol{o}^{(n)}, \boldsymbol{y}^{(n)}\right)}{dw_{ji}^{[k]}}$$

Training by gradient descent

- Initialize all weights $\left\{w_{ji}^{[k]}\right\}$
- Do :
  - For all i , j , k, initialize $\frac{dE}{dw_{ji}^{[k]}} = 0$
  - For all $n = 1 : N$
    - For every layer k for all i, j:
      - Compute $\frac{d\ loss\left(o^{(n)}, y^{(n)}\right)}{dw_{ji}^{[k]}}$
      - $\frac{dE}{dw_{ji}^{[k]}} += \frac{d\ loss\left(o^{(n)}, y^{(n)}\right)}{dw_{ji}^{[k]}}$
  - For every layer k for all i, j:

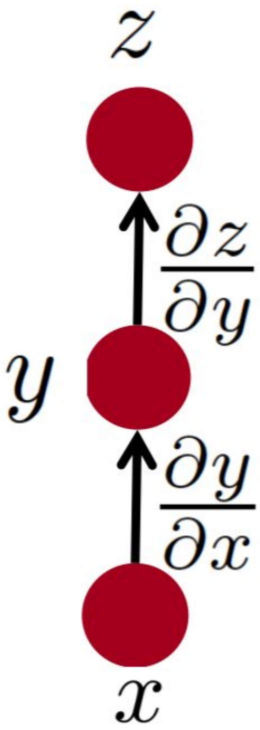
$$w_{ji}^{[k]} = w_{ji}^{[k]} - \frac{\eta}{N} \frac{dE}{dw_{ji}^{[k]}}$$

Training multi-layer networks

- Back-propagation
  - Training algorithm that is used to adjust weights in multi-layer networks
    - The backpropagation algorithm is based on gradient descent
      - The direction of the most rapid decrease in the cost function
  - Use chain rule to efficiently compute gradients

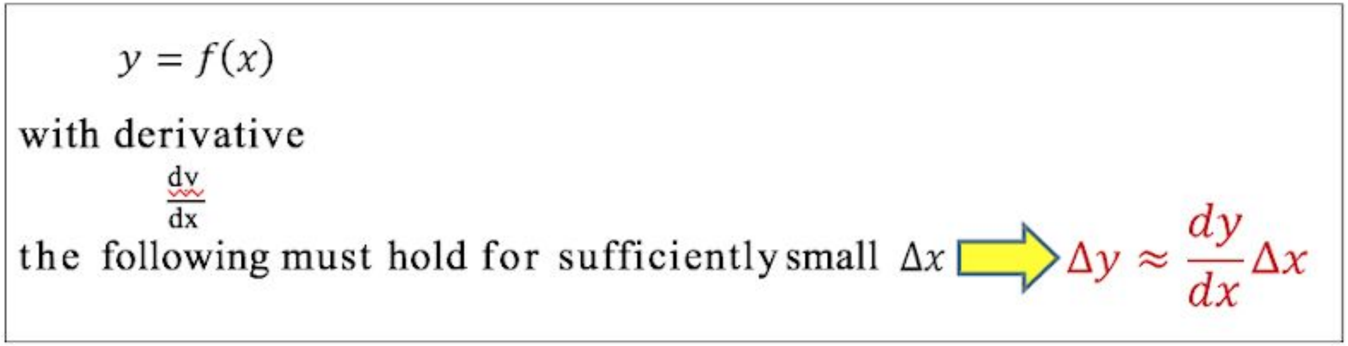
Simple chain rule

- •$z = f\left(g(x)\right)$
- $y = g(x)$


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

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

Calculus Refresher: Basic rules of calculus



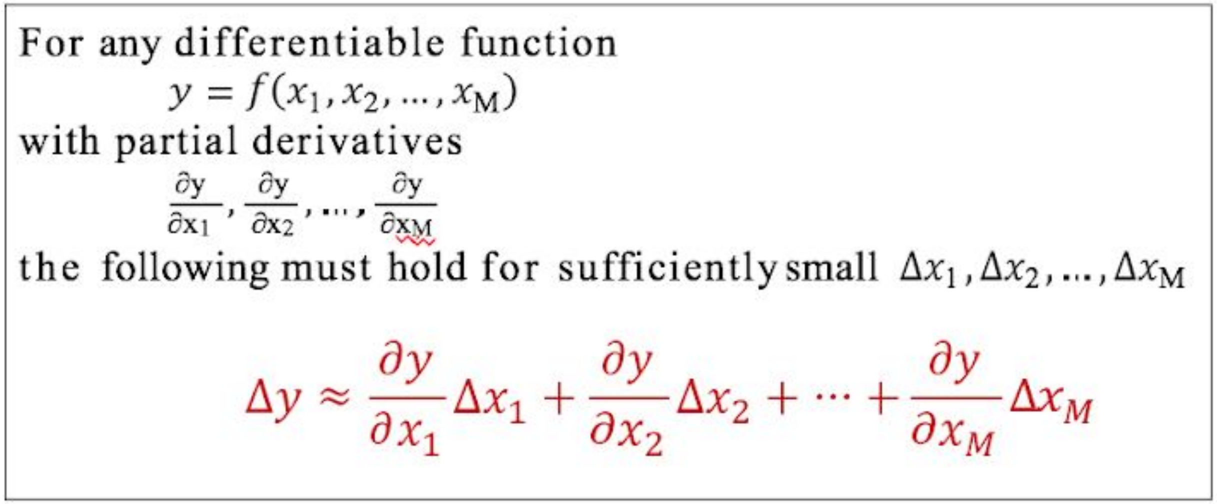
$$y = f(x)$$

with derivative

$$\frac{dy}{dx}$$

the following must hold for sufficiently small Δx ⇨ $\Delta y \approx \frac{dy}{dx}\Delta x$

$$\frac{dy}{dx} \approx \frac{\Delta y}{\Delta x}$$



For any differentiable function

$$y = f(x_1, x_2, \ldots, x_M)$$

with partial derivatives

$$\frac{\partial y}{\partial x_1}, \frac{\partial y}{\partial x_2}, \ldots, \frac{\partial y}{\partial x_M}$$

the following must hold for sufficiently small $\Delta x_1, \Delta x_2, \ldots, \Delta x_M$

$$\Delta y \approx \frac{\partial y}{\partial x_1}\Delta x_1 + \frac{\partial y}{\partial x_2}\Delta x_2 + \cdots + \frac{\partial y}{\partial x_M}\Delta x_M$$

Calculus Refresher: Chain rule

$$y = f(g(x))$$

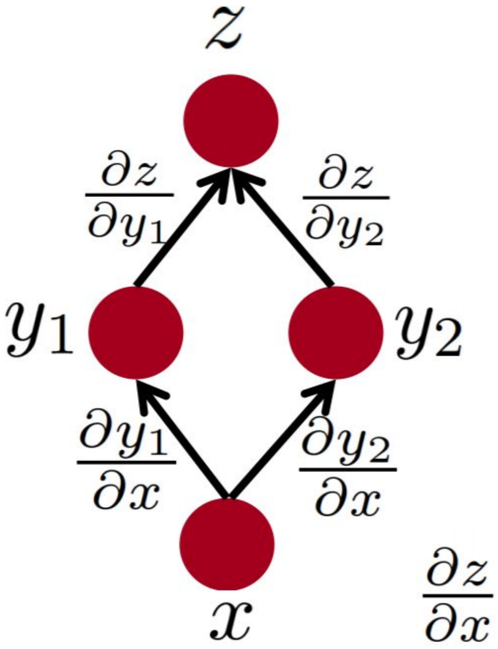
$$\boxed{\frac{dy}{dx} = \frac{\partial f}{\partial g(x)} \frac{dg(x)}{dx}}$$

$$\Delta y = \frac{dy}{dx} \Delta x$$

$$z = g(x) \Rightarrow \Delta z = \frac{dg(x)}{dx} \Delta x$$

$$y = f(z) \Rightarrow \Delta y = \frac{df}{dz} \Delta z = \frac{df}{dz} \frac{dg(x)}{dx} \Delta x$$ ✓

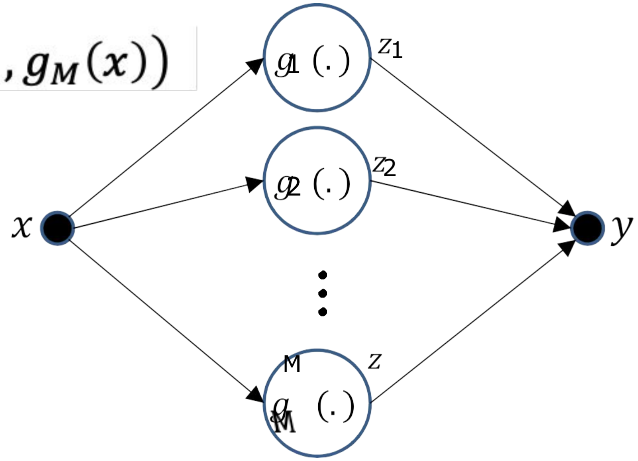
Multiple paths chain rule


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

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

Distributed Chain Rule: Influence Diagram

$$y = f(g_1(x), g_2(x), \ldots, g_M(x))$$



- x affects y through each $g_1, \ldots, g_M$

Calculus Refresher: Distributed Chain rule

$$y = f\big(g_1(x), g_1(x), \dots, g_M(x)\big)$$

$$\boxed{\frac{dy}{dx} = \frac{\partial f}{\partial g_1(x)}\frac{dg_1(x)}{dx} + \frac{\partial f}{\partial g_2(x)}\frac{dg_2(x)}{dx} + \cdots + \frac{\partial f}{\partial g_M(x)}\frac{dg_M(x)}{dx}}$$

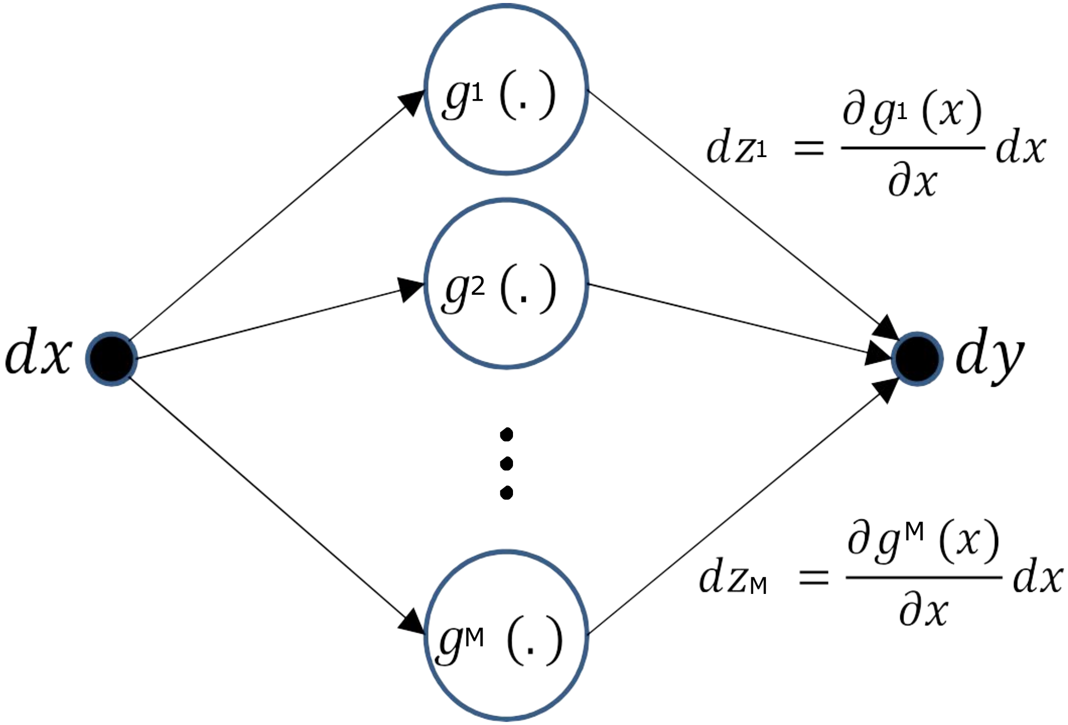
$$\Delta y = \frac{dy}{dx}\Delta x$$

$$\Delta y = \frac{\partial f}{\partial g_1(x)}\Delta g_1(x) + \frac{\partial f}{\partial g_2(x)}\Delta g_2(x) + \cdots + \frac{\partial f}{\partial g_M(x)}\Delta g_M(x)$$

$$\boxed{\Delta y = \frac{\partial f}{\partial g\ (x)}\frac{dg\ (x)}{dx}\Delta x + \frac{\partial f}{\partial g\ (x)}\frac{dg\ (x)}{dx}\Delta x + \cdots + \frac{\partial f}{\partial g\ \ (x)}\frac{dg\ \ (x)}{dx}\Delta x}$$

$$\boxed{\Delta y = \left(\frac{\partial f}{\partial g\ (x)}\frac{dg\ (x)}{dx} + \frac{\partial f}{\partial g\ (x)}\frac{dg\ (x)}{dx} + \cdots + \frac{\partial f}{\partial g\ \ (x)}\frac{dg\ \ (x)}{dx}\right)\Delta x}$$ ✓

Distributed Chain Rule: Influence Diagram


$g_1(.)$
$g_2(.)$
$g_M(.)$
dx
dy
$dz_1 = \frac{\partial g_1(x)}{\partial x} dx$
$dz_M = \frac{\partial g_M(x)}{\partial x} dx$

- Small perturbations in x cause small perturbations in each of $g_1 \ldots g_M, y$

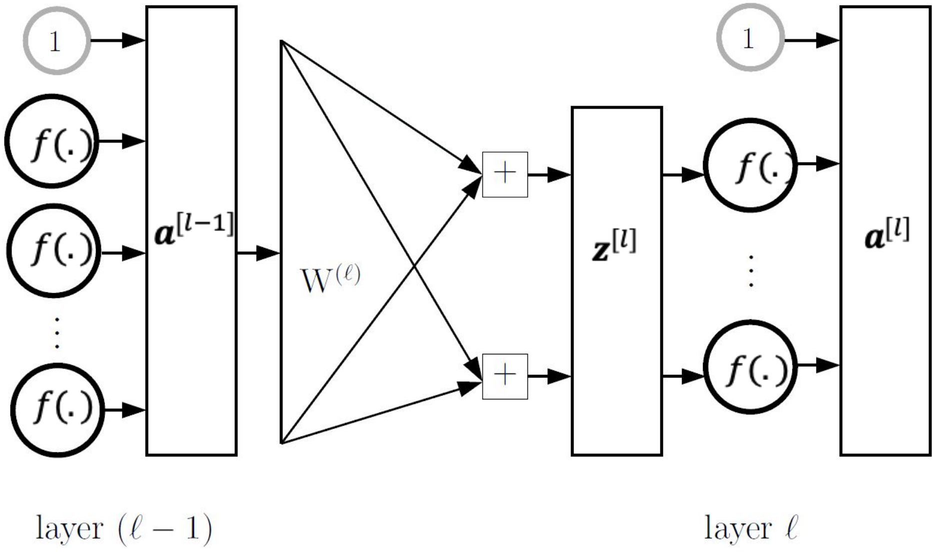
which individually additively perturbs

Returning to our problem

- How to compute $\frac{d\ loss(o,y)}{dw_{ij}^{[k]}}$

Backpropagation: Notation

- $\boldsymbol{a}^{[0]} \leftarrow Input$
- $output \leftarrow \boldsymbol{a}^{[L]}$



Backpropagation: Last layer gradient

For squared error loss:

$$loss = \frac{1}{2}\sum_j (o_j - y_j\ )^2$$

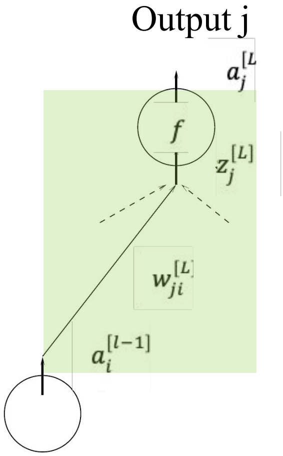
$$o_j = a_j^{[L]}$$

$$\frac{\partial loss}{\partial a_j^{[L]}} = (a_j^{[L]} - y_j)$$

$$\frac{\partial loss}{\partial w_{ji}^{[L]}} = ?$$

$$a_i^{[L]} = f\left(z_i^{[L]}\right)$$

$$z_j^{[L]} = \sum_{i=0}^{M} w_{ji}^{[L]} a_i^{[L-1]}$$



$$\frac{\partial loss}{\partial a_j^{[L]}}$$

$$\frac{\partial loss}{\partial w_{ji}^{[L]}}$$

Backpropagation: Last layer gradient

For squared error loss:

$$loss = \frac{1}{2}\sum_j (o_j - y_j)^2$$

$$o_j = a_j^{[L]}$$

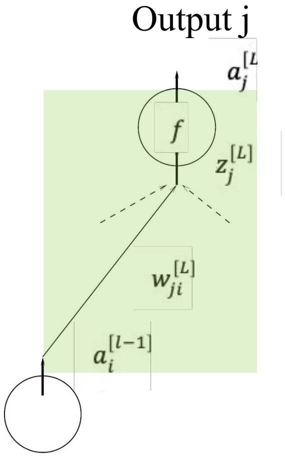
$$\frac{\partial loss}{\partial a_j^{[L]}} = (a_j^{[L]} - y_j)$$

$$a_j^{[L]} = f\left(z_j^{[L]}\right)$$

$$z_j^{[L]} = \sum_{i=0}^{M} w_{ji}^{[L]} a_i^{[L-1]}$$

$$\frac{\partial loss}{\partial w_{ji}^{[L]}} = \frac{\partial loss}{\partial a_j^{[L]}} \frac{\partial a_j^{[L]}}{\partial w_{ji}^{[L]}}$$

$$\frac{\partial a^{[L]}}{\partial w_{ji}^{[L]}} = f'\left(z_j^{[L]}\right)\frac{\partial z_j^{[L]}}{\partial w_{ji}^{[L]}} = f'\left(z_j^{[L]}\right) a_i^{[L-1]}$$

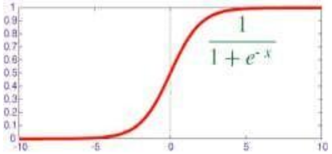


$\frac{\partial l}{\partial}$

$\frac{\partial loss}{\partial w_{ji}^{[L]}}$

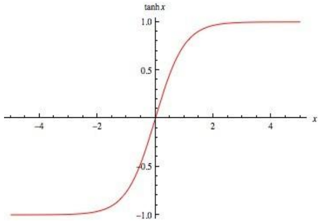
$$\frac{\partial loss}{\partial w_{ji}^{[L]}} = \frac{\partial loss}{\partial a_j^{[L]}} f'\left(z_j^{[L]}\right) a_i^{[L-1]}$$

Activations and their derivatives



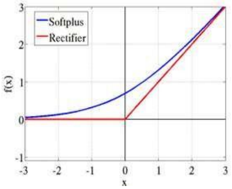
$$f(z) = \frac{1}{1 + \exp(-z)}$$

$$f'(z) = f(z)(1 - f(z))$$



$$f(z) = \tanh(z)$$

f $\quad f'(z) = 1 - f^2(z)$



$$f(z) = \begin{cases} 0, & z < 0 \\ z, & z \geq 0 \end{cases}$$

$$f'(z) = \begin{cases} 1, z \geq 0 \\ 0, z < 0 \end{cases}$$

$$f(z) = \log(1 + \exp(z))$$

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

- Some popular activation functions and their derivatives

Previous layers gradients

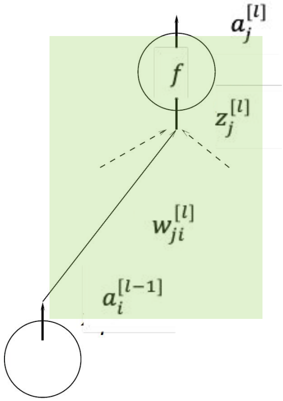
$$\frac{\partial\, loss}{\partial w_{ji}^{[l]}} = \frac{\partial\, loss}{\partial z_j^{[l]}} \frac{\partial z_j^{[l]}}{\partial w_{ji}^{[l]}}$$

$$\frac{\partial\, loss}{\partial w_{ji}^{[l]}} = \frac{\partial\, loss}{\partial z_j^{[l]}} a_i^{[l-1]}$$

$$\frac{\partial\, loss}{\partial z_j^{[l]}} = ?$$

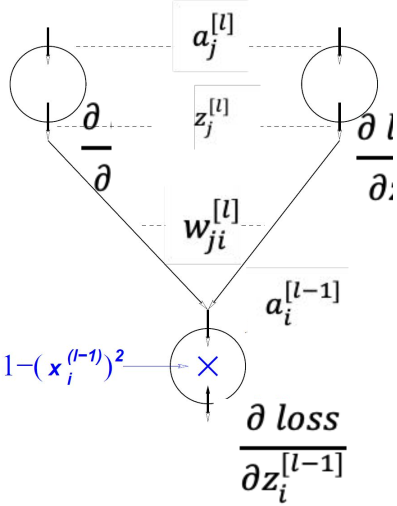
$$a_j^{[l]} = f\left(z_j^{[l]}\right)$$

$$z_j^{[l]} = \sum_{i=0}^{M} w_{ji}^{[l]} a_i^{[l-1]}$$



$$\frac{\partial\, loss}{\partial z_j^{[l]}}$$

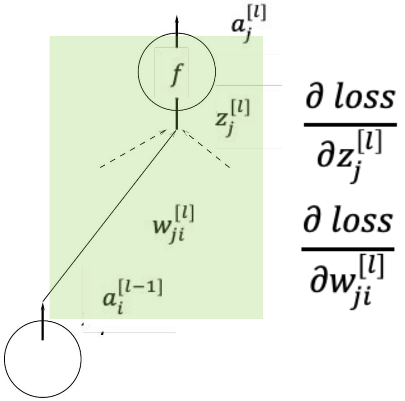
$$\frac{\partial\, loss}{\partial w_{ji}^{[l]}}$$



Previous layers gradients

$$a_j^{[l]} = f\left(z_j^{[l]}\right)$$

$$z_j^{[l]} = \sum_{i=0}^{M} w_{ji}^{[l]} a_i^{[l-1]}$$

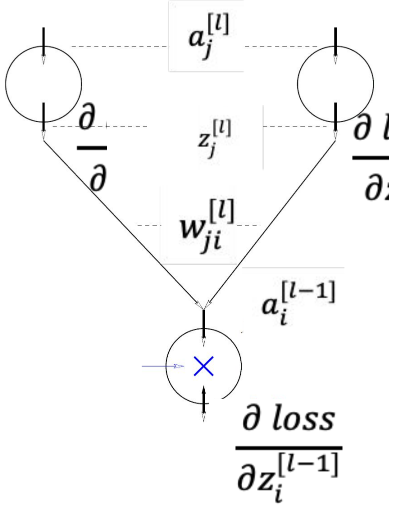


$a_j^{[l]}$
f
$z_j^{[l]}$
$w_{ji}^{[l]}$
$a_i^{[l-1]}$
$\frac{\partial loss}{\partial z_j^{[l]}}$
$\frac{\partial loss}{\partial w_{ji}^{[l]}}$

$$\frac{\partial loss}{\partial w_{ji}^{[l]}} = \frac{\partial loss}{\partial z_j^{[l]}} \frac{\partial z_j^{[l]}}{\partial w_{ji}^{[l]}} = \frac{\partial loss}{\partial z_j^{[l]}} a_i^{[l-1]}$$

$$\frac{\partial loss}{\partial z_i^{[l-1]}} = \frac{\partial a_i^{[l-1]}}{\partial z_i^{[l-1]}} \sum_{j=1}^{d^{[l]}} \frac{\partial loss}{\partial z_j^{[l]}} \times \frac{\partial z_j^{[l]}}{\partial a_i^{[l-1]}}$$

$$= f'\left(z_i^{[l-1]}\right) \sum_{j=1}^{d^{[l]}} \frac{\partial loss}{\partial z_j^{[l]}} \times w_{ji}^{[l]}$$



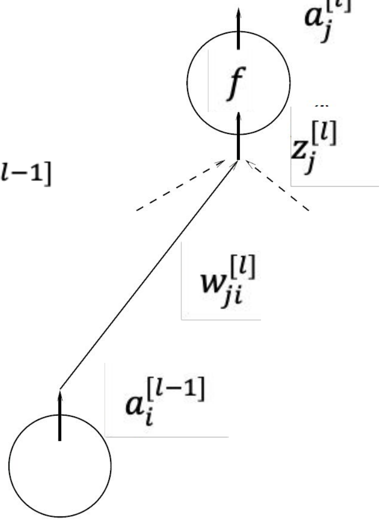
Backpropagation:

$$\frac{\partial\, loss}{\partial w_{ji}^{[l]}} = \boxed{\frac{\partial\, loss}{\partial z_j^{[l]}}} \times \frac{\partial z_j^{[l]}}{\partial w_{ji}^{[l]}}$$

$$= \boxed{\delta_j^{[l]}} \times a_i^{[l-1]}$$

$$a_j^{[l]} = f\left(z_j^{[l]}\right)$$

$$z_j^{[l]} = \sum_{i=0}^{M} w_{ji}^{[l]} a_i^{[l-1]}$$



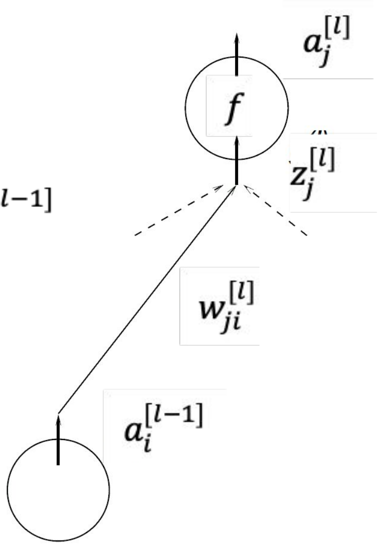
Backpropagation:

$$\frac{\partial\, loss}{\partial w_{ji}^{[l]}} = \boxed{\frac{\partial\, loss}{\partial z_j^{[l]}}} \times \frac{\partial z_j^{[l]}}{\partial w_{ji}^{[l]}}$$

$$= \boxed{\delta_j^{[l]}} \times a_i^{[l-1]}$$

$$a_j^{[l]} = f\left(z_j^{[l]}\right)$$

$$z_j^{[l]} = \sum_{i=0}^{M} w_{ji}^{[l]} a_i^{[l-1]}$$



- $\delta_j^{[l]} = \frac{\partial\, loss}{\partial z_j^{[l]}}$ is the sensitivity of the loss to $z_j^{[l]}$
- Sensitivity vectors can be obtained by running a backward process in the network architecture (hence the name backpropagation.)

We will compute $\boldsymbol{\delta}^{[l-1]}$ from $\boldsymbol{\delta}^{[l]}$:

$$\delta_i^{[l-1]} = f'\left(z_i^{[l-1]}\right) \sum_{j=1}^{d^{[l]}} \delta_j^{[l]} \times w_{ji}^{[l]}$$

Backward process on sensitivity vectors

-
  - For the final layer $l = L$:

$$\delta_j^{[L]} = \frac{\partial\, loss}{\partial z_j^{[L]}}$$

  - Compute $\boldsymbol{\delta}^{[l-1]}$ from $\boldsymbol{\delta}^{[l]}$: by running a backward process in the network architecture:

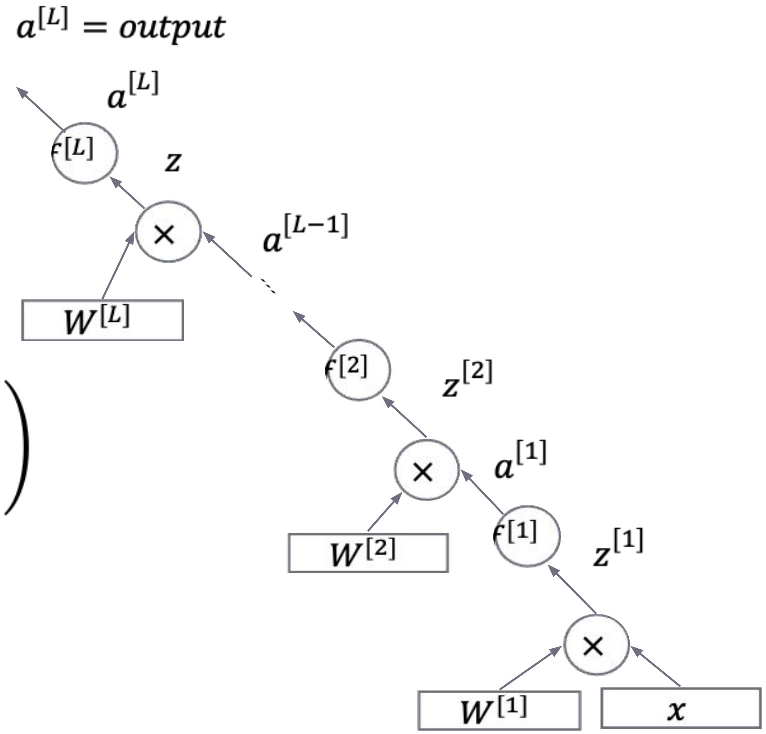
$$\delta_i^{[l-1]} = f'\left(z_i^{[l-1]}\right) \sum_{j=1}^{d^{[l]}} \delta_j^{[l]} \times w_{ji}^{[l]}$$

Backpropagation Algorithm

- Initialize all weights to small random numbers.
- **While not satisfied**
- **For** each training example **do**:
  1. Feed forward the training example to the network and compute the outputs of all units in forward step (z and a) and the loss
  2. For each unit find its δ in the backward step
  3. Update each network weight $w_{ji}^{[l]}$ as $w_{ji}^{[l]} \leftarrow w_{ji}^{[l]} - \eta \frac{\partial\, loss}{\partial w_{ji}^{[l]}}$ where $\frac{\partial\, loss}{\partial w_{ji}^{[l]}}$ $= \delta_j^{[l]} \times a_i^{[l-1]}$

Multi-layer network: Matrix notation

$$\begin{aligned} Output &= a^{[L]} \\ &= f\left(z^{[L]}\right) \\ &= f\left(W^{[L]}a^{[L-1]}\right) \\ &= f\left(W^{[L]}f(W^{[L-1]}a^{[L-2]}\right) \\ &= f\left(W^{[L]}f\left(W^{[L-1]}\ldots f\left(W^{[2]}f(W^{[1]}x)\right)\right)\right) \end{aligned}$$

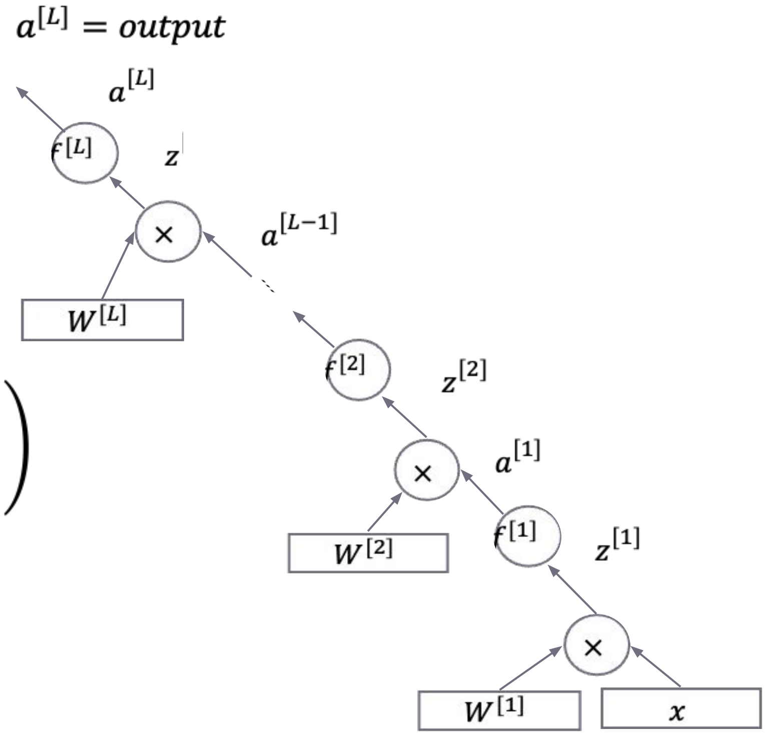


Multi-layer network: Matrix notation

$$
\begin{aligned}
Output &= a^{[L]} \\
&= f\left(z^{[L]}\right) \\
&= f\left(W^{[L]} a^{[L-1]}\right) \\
&= f\left(W^{[L]} f(W^{[L-1]} a^{[L-2]}\right) \\
&= f\left(W^{[L]} f\left(W^{[L-1]} \ldots f\left(W^{[2]} f(W^{[1]} x)\right)\right)\right)
\end{aligned}
$$

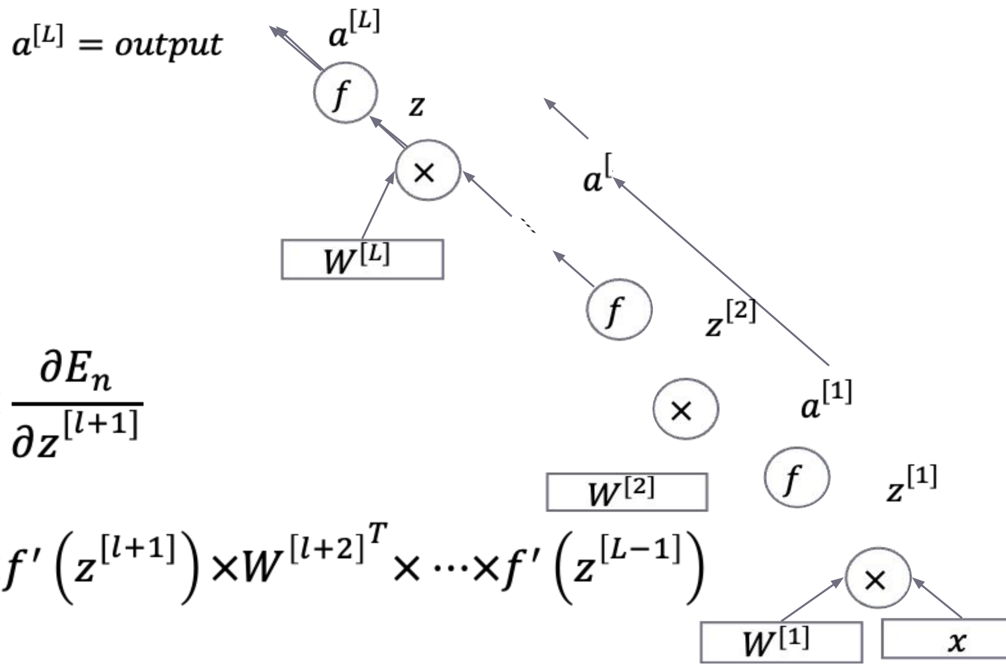
$$\frac{\partial\, loss}{\partial W^{[l]}} = \frac{\partial\, loss}{\partial z^{[l]}} a^{[l-1]^T}$$

$$\frac{\partial\, loss}{\partial z^{[l]}} = f'(z^{[l]}) W^{[l+1]^T} \frac{\partial\, loss}{\partial z^{[l+1]}}$$



Vanishing/exploding gradients

-



$$\frac{\partial E_n}{\partial z^{[l-1]}} = f'\left(z^{[l-1]}\right) \times {W^{[l]}}^T \times \frac{\partial E_n}{\partial z^{[l]}}$$

$$= f'\left(z^{[l-1]}\right) \times {W^{[l]}}^T \times f'\left(z^{[l]}\right) \times {W^{[l+1]}}^T \times \frac{\partial E_n}{\partial z^{[l+1]}}$$

$$= \cdots$$

$$= f'\left(z^{[l-1]}\right) \times {W^{[l]}}^T \times f'\left(z^{[l]}\right) \times {W^{[l+1]}}^T \times f'\left(z^{[l+1]}\right) \times {W^{[l+2]}}^T \times \cdots \times f'\left(z^{[L-1]}\right)$$
$$\times {W^{[L]}}^T \times f'\left(z^{[L]}\right) \times \frac{\partial E_n}{\partial a^{[L]}}$$

$$\frac{\partial E_n}{\partial W^{[l]}} = \frac{\partial E_n}{\partial z^{[l]}} \times {a^{[l-1]}}^T$$

Vanishing/exploding gradients

$$\delta^{[l-1]} =$$
$$= f'\left(z^{[l-1]}\right) \times W^{[l]^T} \times f'\left(z^{[l]}\right) \times W^{[l+1]^T} \times f'\left(z^{[l+1]}\right) \times W^{[l+2]^T} \times \cdots$$
$$\times f'\left(z^{[L-1]}\right) \times W^{[L]^T} \times \delta^{[L]}$$

For deep networks:

Large weights can cause exploding gradients

Small weights can cause vanishing gradients

Example:

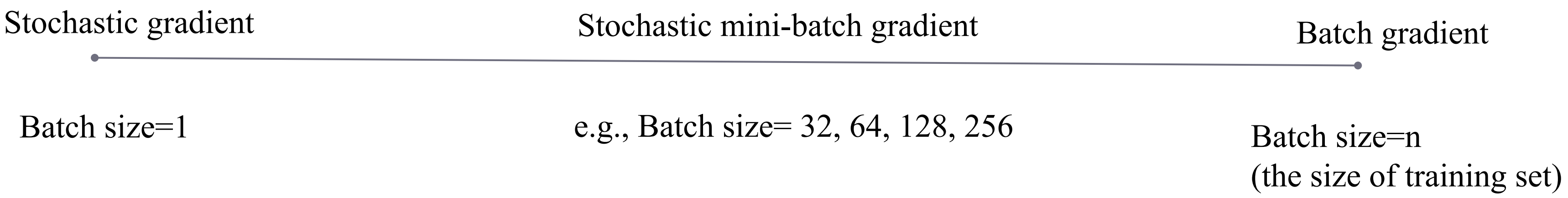
$W^{[1]} = \cdots = W^{[L]} = \omega I, f(z) = z \Rightarrow$

$\delta^{[1]} = (\omega I)^{L-1}\delta^{[L]} = (\omega)^{L-1}\delta^{[L]}$

Mini-batch gradient descent

- Large datasets
  - Divide dataset into smaller batches containing one subset of the main training set
  - Weights are updated after seeing training data in each of these batches

Gradient descent methods



Stochastic gradient

Batch size=1

Stochastic mini-batch gradient

e.g., Batch size= 32, 64, 128, 256

Batch gradient

Batch size=n
(the size of training set)

n: whole no of training data
bs: the size of batches
$m = \left\lceil \frac{n}{bs} \right\rceil$: the number of batches

Mini-batch gradient descent

```
● For epoch=1,…,k
  1. For t=1,…,m
       1. Forward propagation on $X^{\{t\}}$
       2. $J^{\{t\}} = \frac{1}{m}\sum_{n\in Batch_t} L\left(\hat{Y}_n^{\{t\}}, Y_n^{\{t\}}\right) + \lambda R(W)$
       3. Backpropagation on $J^{\{t\}}$ to compute gradients $dW$
       4. For $l = 1, \dots, L$
            1. $W^{[l]} = W^{[l]} - \alpha dW^{[l]}$
```

1 epoch: Single pass over all training samples

```
$A^{[0]} = X^{\{t\}}$
1. For $l = 1, \dots, L$
     1. $Z^{[l]} = W^{[l]}A^{[l-1]}$
     2. $A^{[l]} = f^{[l]}\left(Z^{[l]}\right)$
$\hat{Y}_n^{\{t\}} = A_n^{[L]}$
```

Vectorized computaion

Gradient descent methods

Stochastic gradient descent

Batch size=1

- Does not use vectorized form and thus not computationally efficient

Stochastic mini-batch gradient

e.g., Batch size= 32, 64, 128, 256

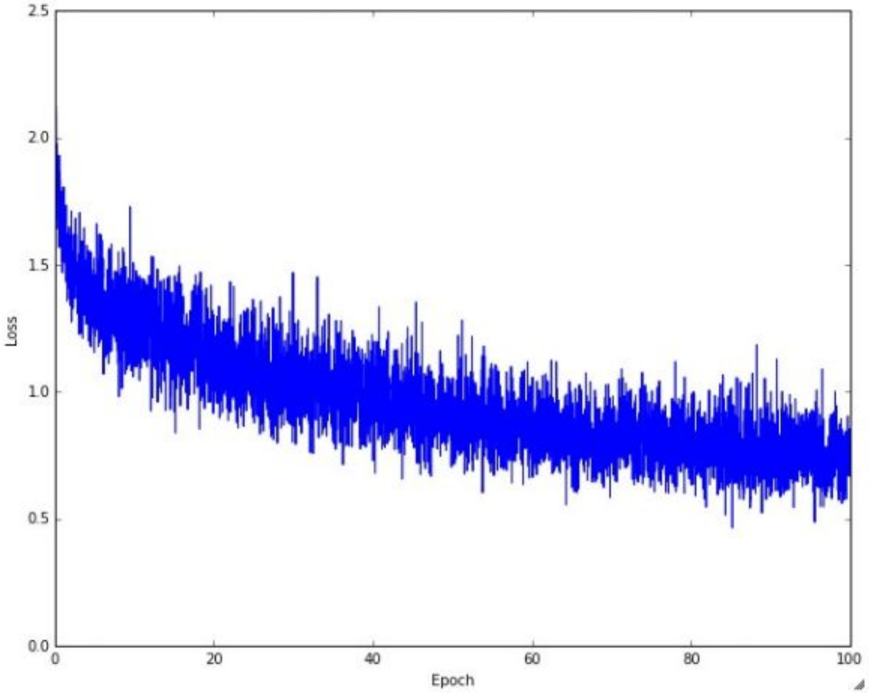
- Vectorization
- Fastest learning
  (for proper batch size)

Batch gradient descent

Batch size=n
(the size of training set)

- Need to process whole training set for weight update

Mini-batch gradient descent: loss-#epoch curve



Choosing mini-batch size

- For small training sets (e.g., n<2000) use full-batch gradient descent

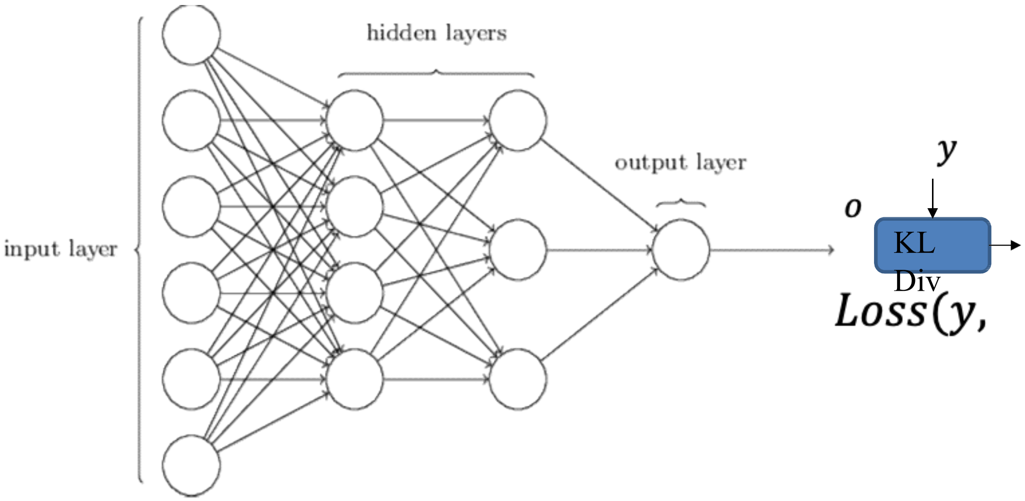
- Typical mini-batch sizes for larger training sets:
  - 64, 128, 256, 512

- Make sure one batch of training data and the corresponding forward, backward required to be cached can fit in GPU memory

Designing a network

- Input
- Output
  - Output activation
  - Loss function
- Hidden layers
  - Activation function of hidden layers
  - Number of hidden layers and number of hidden units in each layer

Binary classifier example: Logistic regression



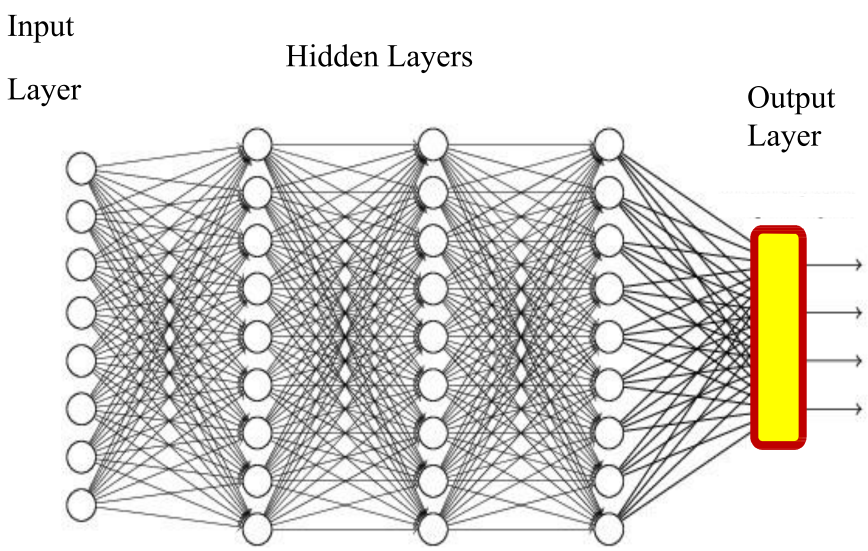
$$o = \sigma(z)$$

- For binary classifier with scalar output $o \in (0,1)$, y is 0/1, the cross entropy between the probability distribution $[o, 1-o]$ and the ideal output probability $[y, 1-y]$ is popular $Loss(y,o) = -ylogo \; - (1-y)\log(1-o)$

- Derivative

$$\frac{dL(y,o)}{do} = \begin{cases} -\frac{1}{o} \; if \; y = 1 \\ \frac{1}{1-o} \; if \; y = 0 \end{cases}$$

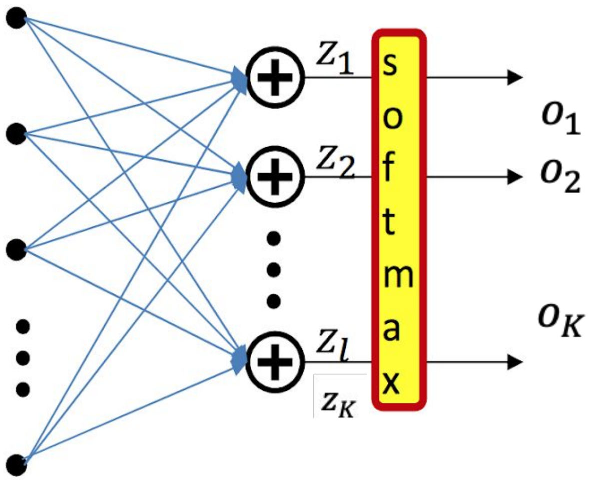
Multi-class networks



- For input of any class, we will have a K-dimensional target vector with K-1 zeros and a single 1 at the position of the class (1-hot)

- The neural network's output will be a probability vector
  - K probability values that sum to 1.

Softmax activation function



$$o_i = \frac{exp(z_i)}{\sum_j exp(z_j)}$$

- Softmax vector activation is often used at the output of multi-class classifier nets

$$z_i = \sum_j w_{ji}^{(l)} a_j^{(n-1)}$$

$$o_i = \frac{\exp(z_i)}{\sum_j \exp(z_j)}$$

- This can be viewed as the probability $o_i = P(class = i|\boldsymbol{x})$