Universität
Konstanz

Lecture 6 Deep Learning – Part 1 Neural Network

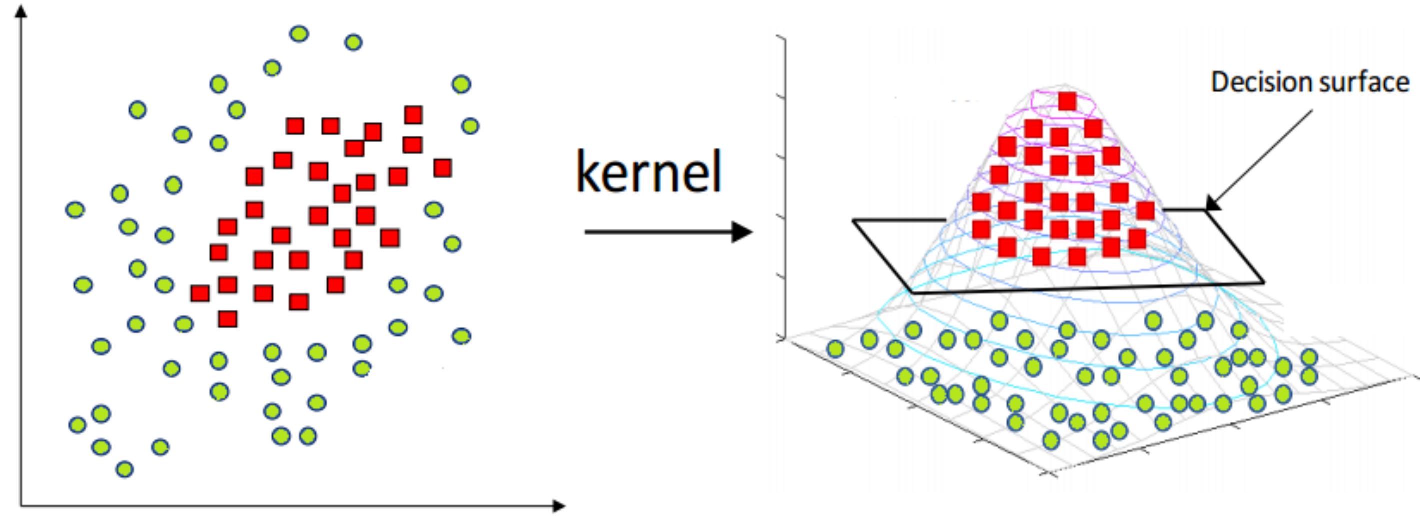
Dr. Hanhe Lin
Dept. of Computer and Information Science
University of Konstanz

Non-linear data representation

- Apply the linear model not to data x itself but to a transformed input $\Phi(x)$, where Φ is a non-linear transformation. In other words, mapping the linear data to a non-linear feature space
- Three options to choose the mapping Φ
  - Use a very generic Φ
  - Manually engineer Φ
  - Deep learning strategy

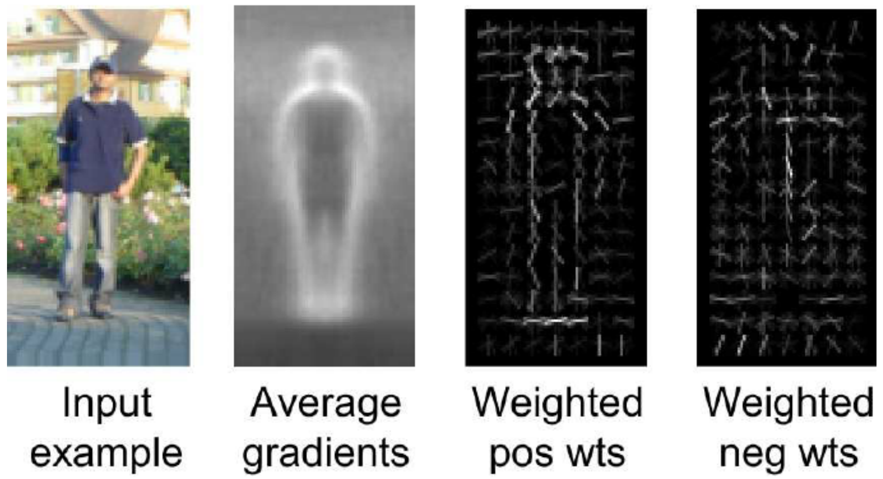
Option 1: Use a very generic Φ

- Mapping data from *n*-dimensional to *N*-dimensional space ($n << N$)
- Pros: Can always learn a linear classifier
- Cons: Poor generalization to the test set
- Example: Kernel trick in SVM



Option 2: Manually engineer Φ

- Requires human effort for each separate task with practitioners specializing in different domains, such as computer vision or speech recognition
- Pros: Good performance
- Cons: Time-consuming to design and difficult to transfer between domains
- Example: Histograms of oriented gradients (HOG)



Option 3: Deep learning strategy

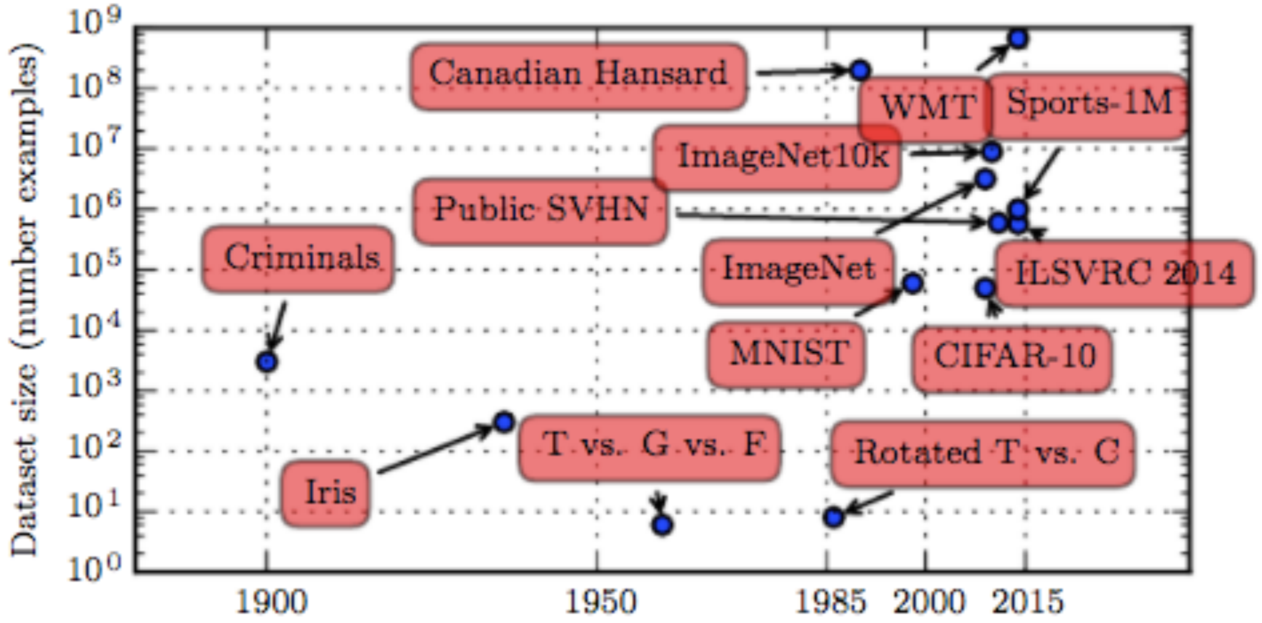
Given a model $y = f(x;\theta,w,b) = \Phi(x;\theta)w + b$, deep learning learns both θ, w, and b simultaneously

- Pros:
  - Outstanding performance
  - Easy to transfer between domains
- Cons:
  - Time-consuming
  - Non-convex optimization

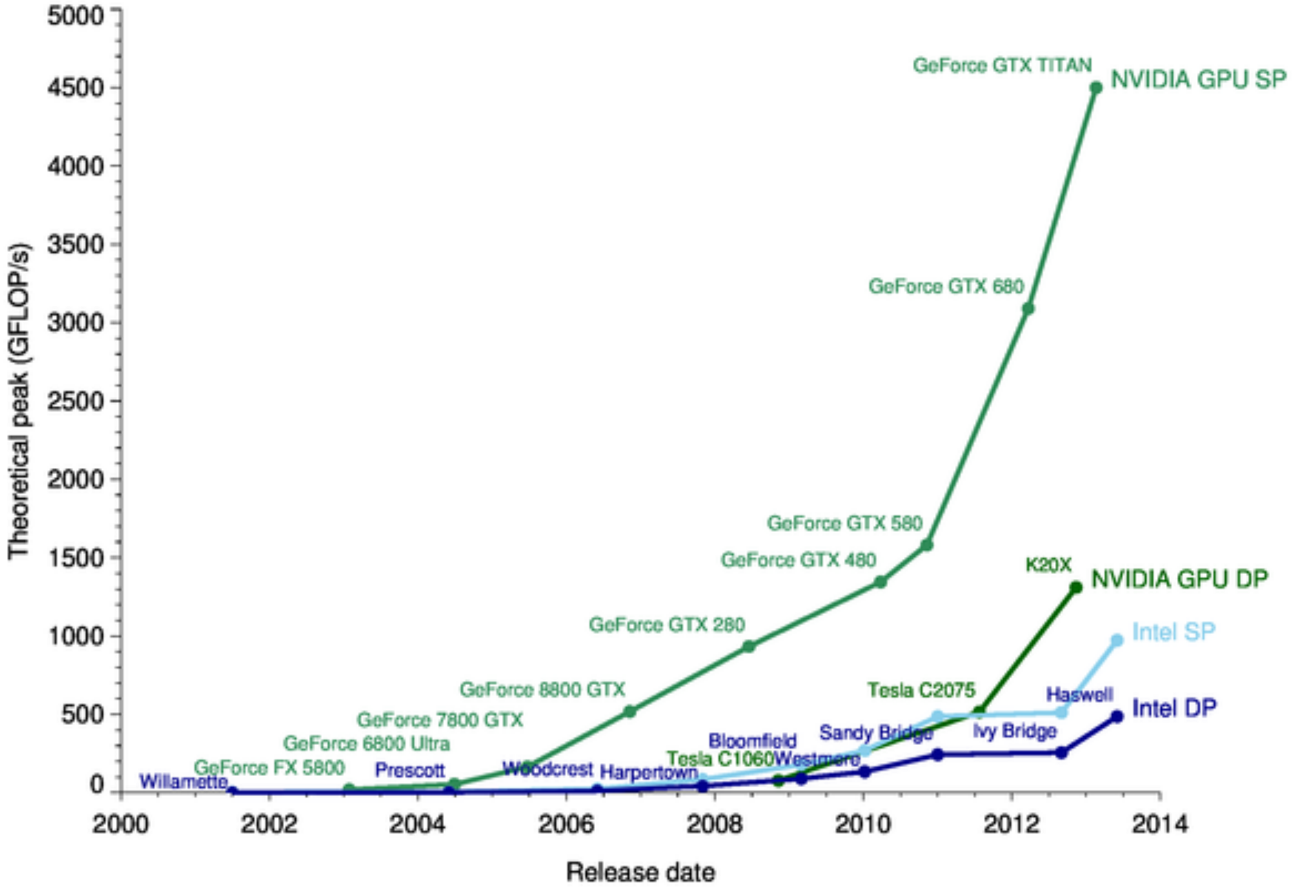
Why is deep learning taking off?

- Neural network has a very long history in machine learning, it has attracted more attentions recently due to:
  - Increasing dataset size
  - Increasing computational power
  - Increasing performance

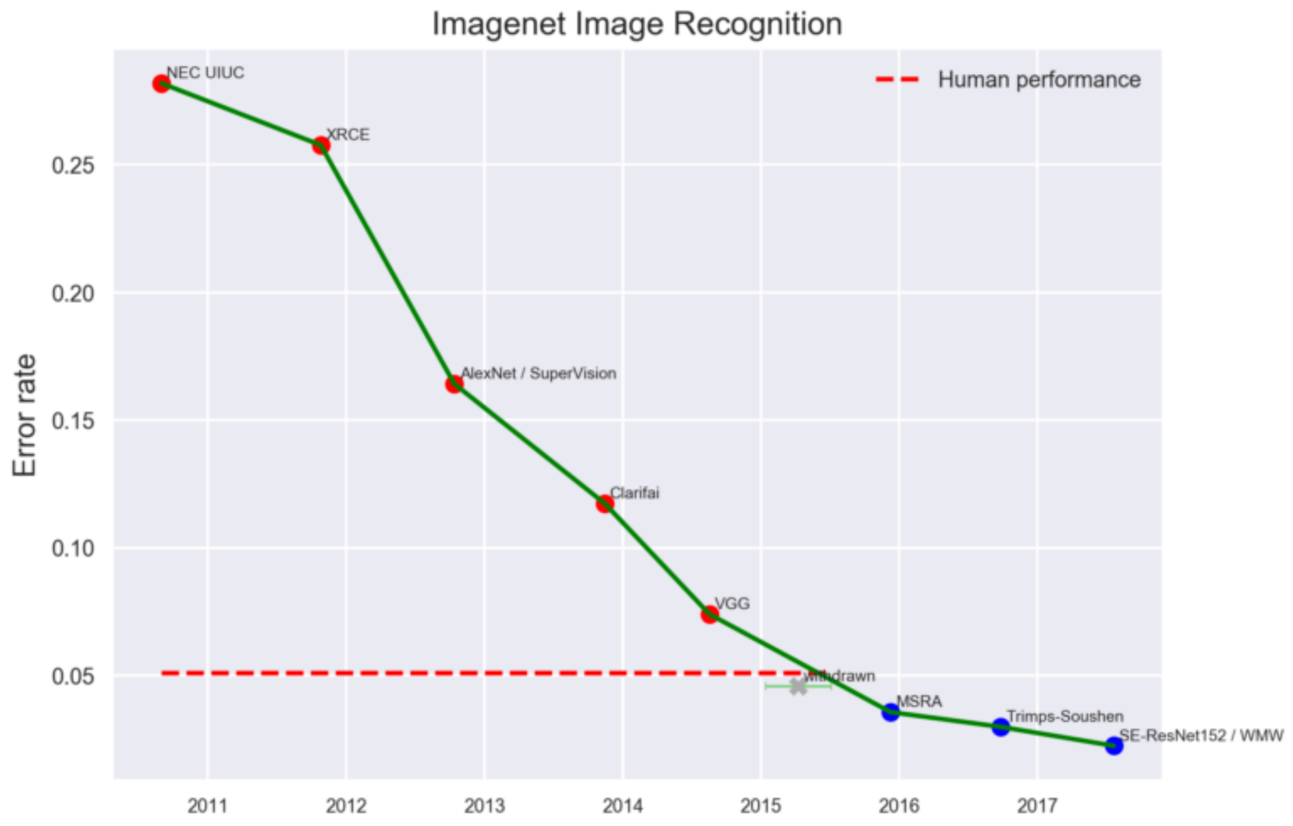
Increasing dataset size



Increasing computational power



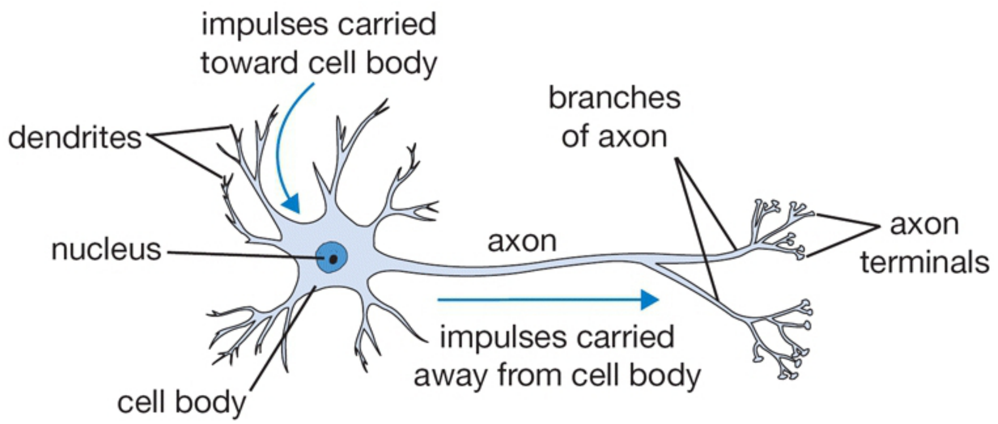
Increasing performance



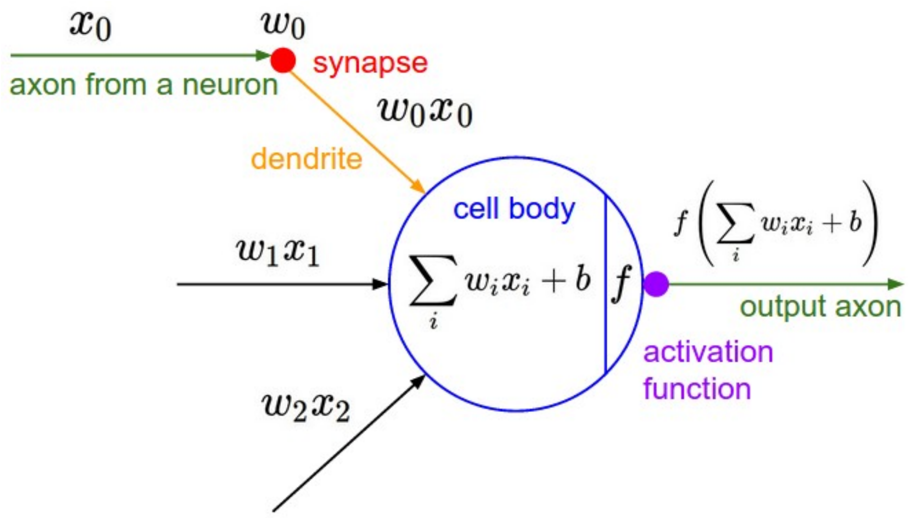
Definition and history

- Definition: "Artificial neural networks" are massively parallel interconnected networks of simple elements and their hierarchical organizations which are intended to interact with the objects of the real world in the same way as biological nervous systems do. (Kohonen, 1988)
- History:
  - Scientists proposed neural networks to mimic the brain
  - Neural networks are widely used in 80s and early 90s, but diminished in late 90s
  - With the increasing of computational power, neural networks become resurgent

Neuron


impulses carried toward cell body
dendrites
branches of axon
nucleus
axon
axon terminals
impulses carried away from cell body
cell body

A biological neuron



Mathematical model of a biological neuron

- A **neuron** is a basic computational unit of the brain
- All the neurons are connected by **synapses**
- Each neuron receives signals from its **dendrites** and outputs signals along its **axon**

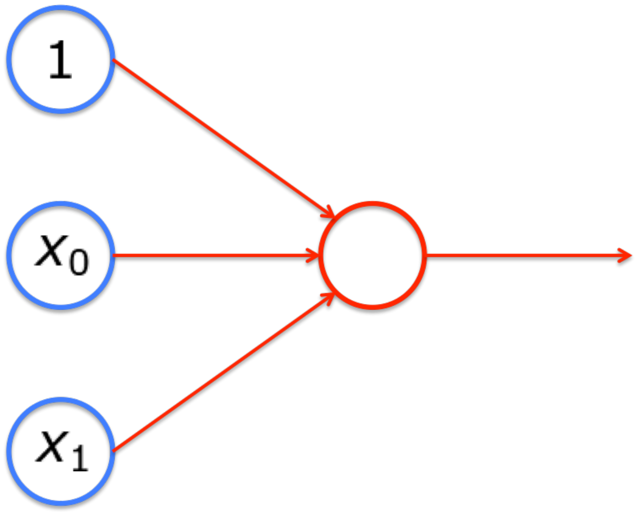
Example: implement logic gate by neurons

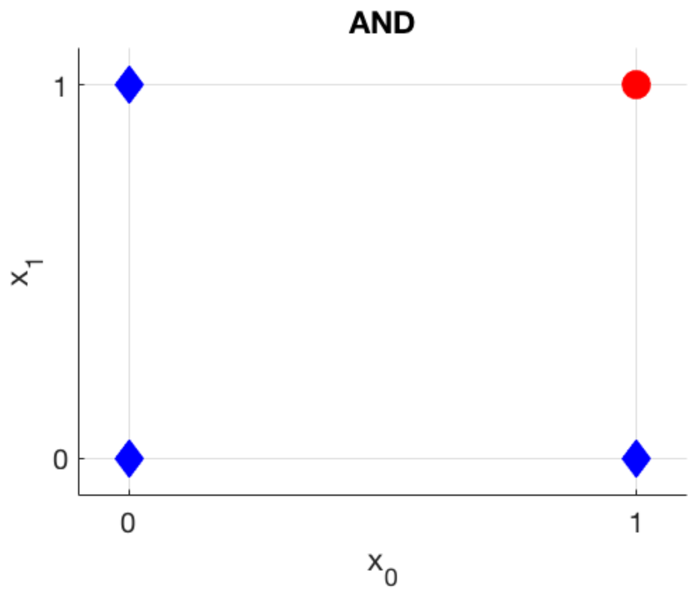
- Suppose we have two binary inputs x_0 and x_1, how can we use neurons to generate output of functions AND, OR, NOR, and XNOR? Sigmoid function is applied as activation function here.

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

Input		Output			
x_0	x_1	AND	OR	NOR	XNOR
0	0	0	0	1	1
0	1	0	1	0	0
1	0	0	1	0	0
1	1	1	1	0	1

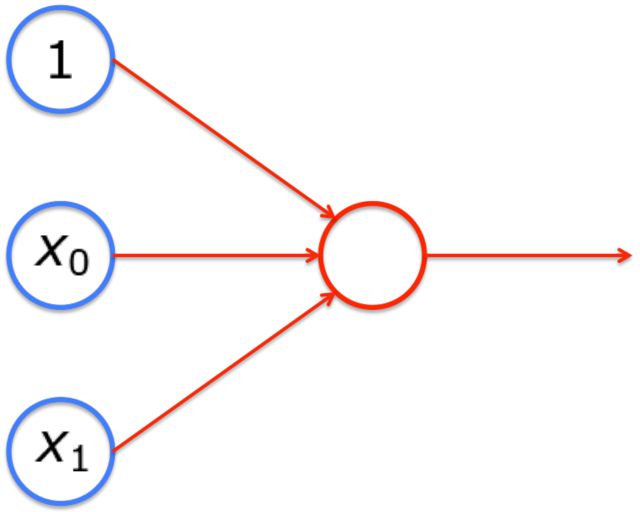
AND function



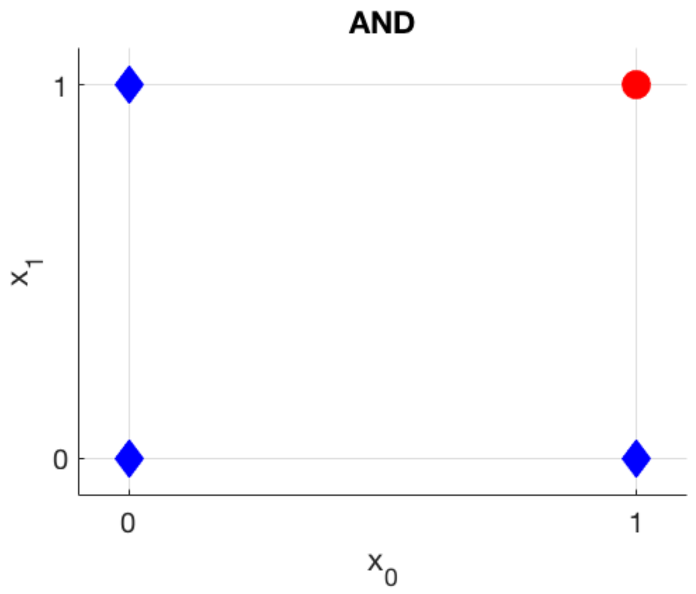


Input		Output
x_0	x_1	AND
0	0	0
0	1	0
1	0	0
1	1	1

AND function

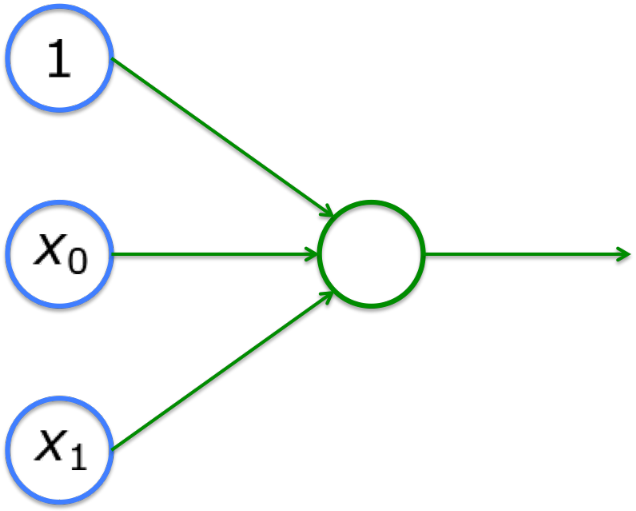


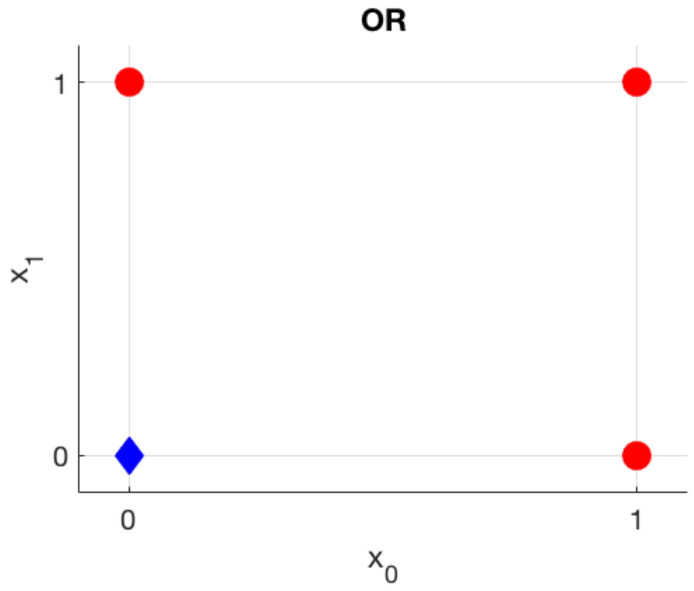
w= [20 20], b = -30



Input		Output
x_0	x_1	AND
0	0	0
0	1	0
1	0	0
1	1	1

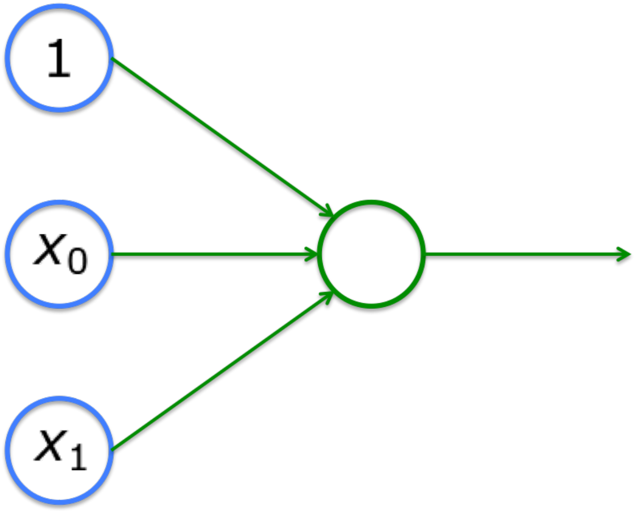
OR function



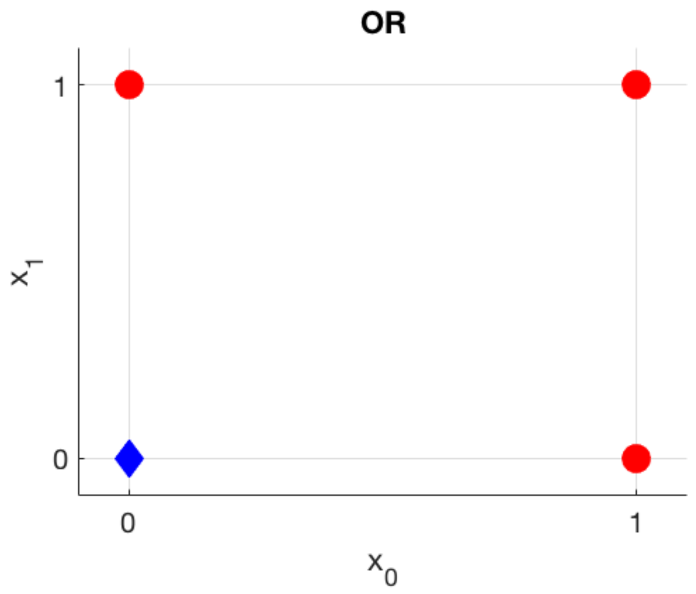


Input		Output
x_0	x_1	OR
0	0	0
0	1	1
1	0	1
1	1	1

OR function

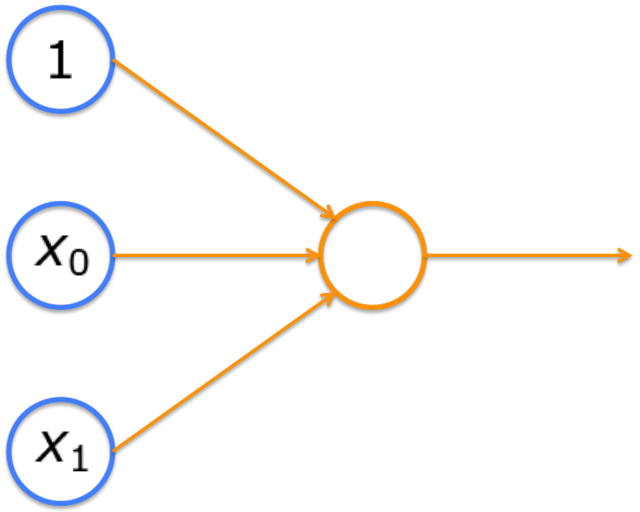


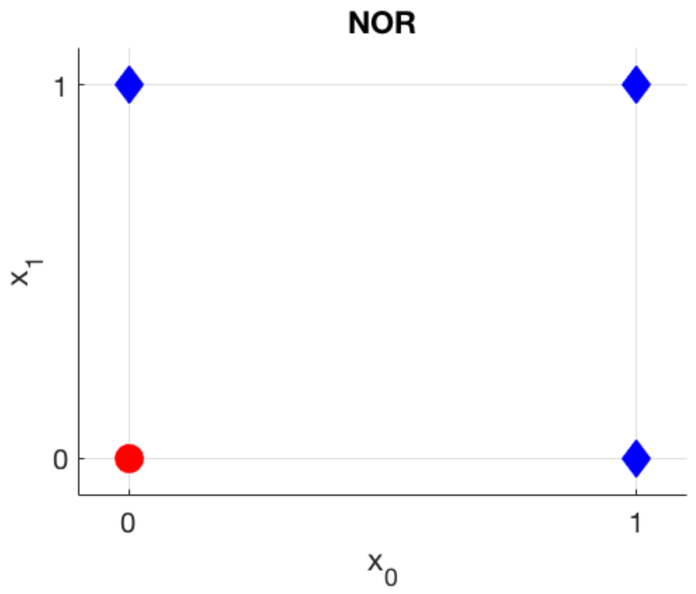
w= [20 20], b = -10



Input		Output
x_0	x_1	OR
0	0	0
0	1	1
1	0	1
1	1	1

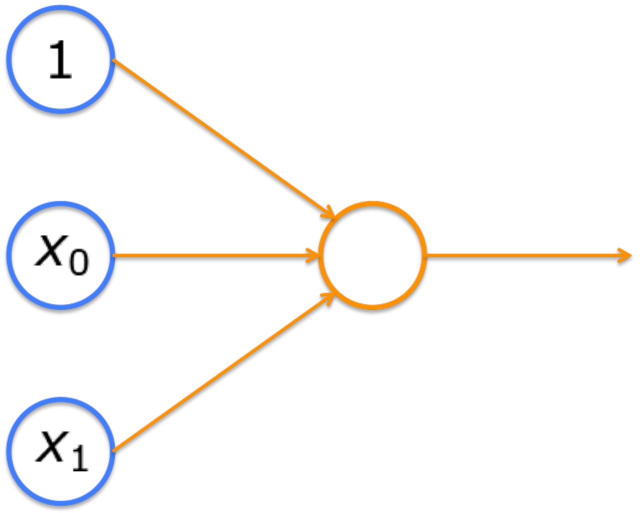
NOR function



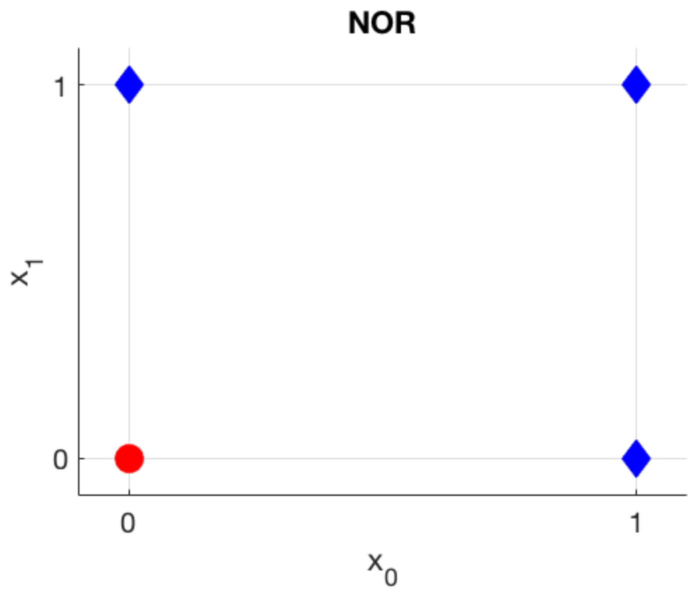


Input		Output
x_0	x_1	NOR
0	0	1
0	1	0
1	0	0
1	1	0

NOR function

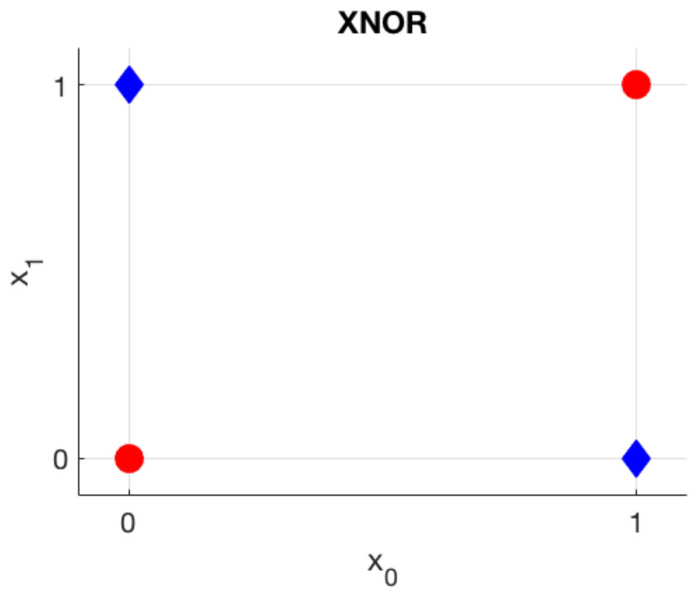


w= [-20 -20], b = 10



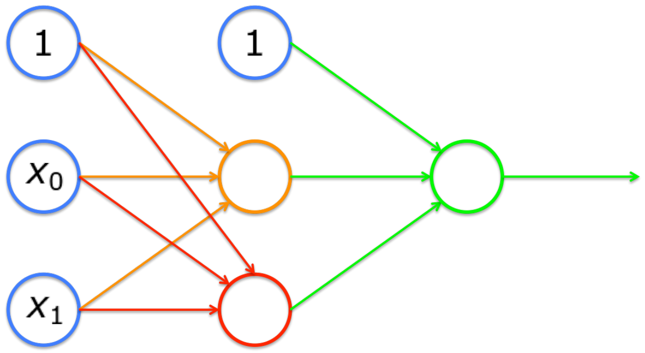
Input		Output
x_0	x_1	NOR
0	0	1
0	1	0
1	0	0
1	1	0

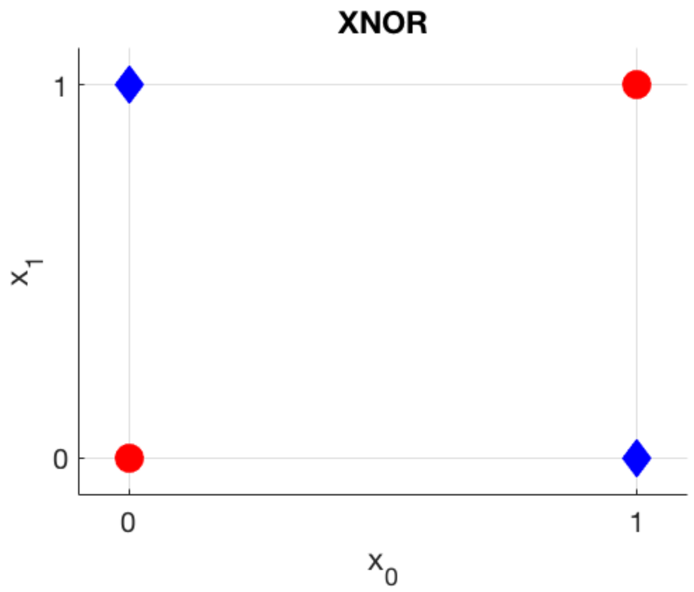
XNOR function



Input		Output
x_0	x_1	XNOR
0	0	1
0	1	0
1	0	0
1	1	1

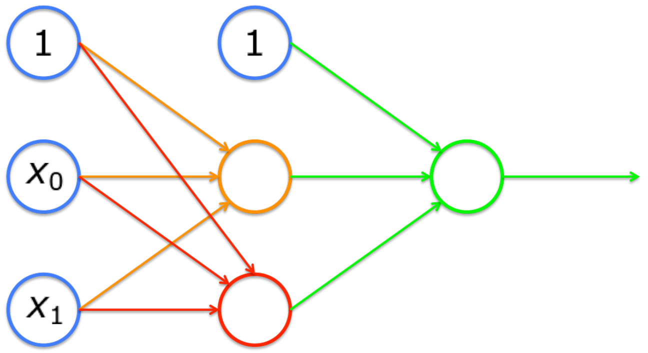
XNOR function

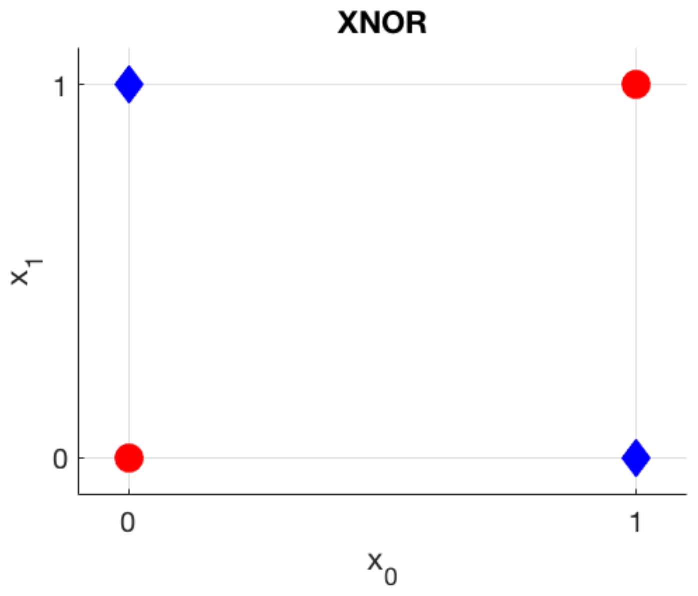




Input		Output
x_0	x_1	XNOR
0	0	1
0	1	0
1	0	0
1	1	1

XNOR function





Input		Output
x_0	x_1	XNOR
0	0	1
0	1	0
1	0	0
1	1	1

Intuitive understanding: while a single neuron can be regarded as a binary linear classifier, put multiple neurons together is able to apply *nonlinear* transformation.

Activation function

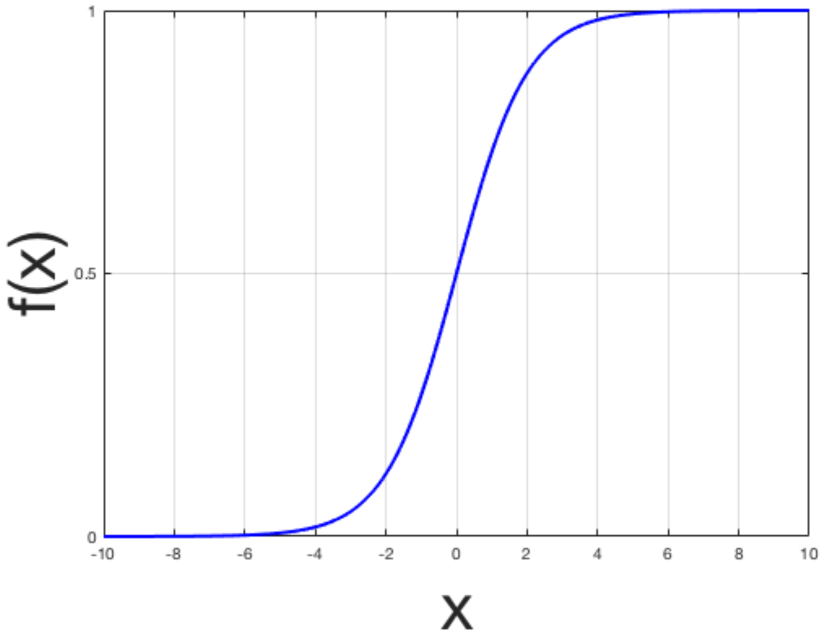
- Every activation function takes a single number and performs a certain fixed mathematical operation on it
- Commonly used activation functions:
  - Sigmoid
  - Hyperbolic tangent (tanh)
  - Rectified Linear Unit (ReLU)
  - Leaky ReLU
  - Maxout

Sigmoid

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

- It squashes the input real-valued number into range between 0 and 1
- Frequent use historically but recently out of favor because:
  - Sigmoid saturates and kills gradients
  - Sigmoid outputs are not zero-centered

$$\frac{\partial}{\partial x}\sigma(x) = \sigma(x)(1 - \sigma(x))$$



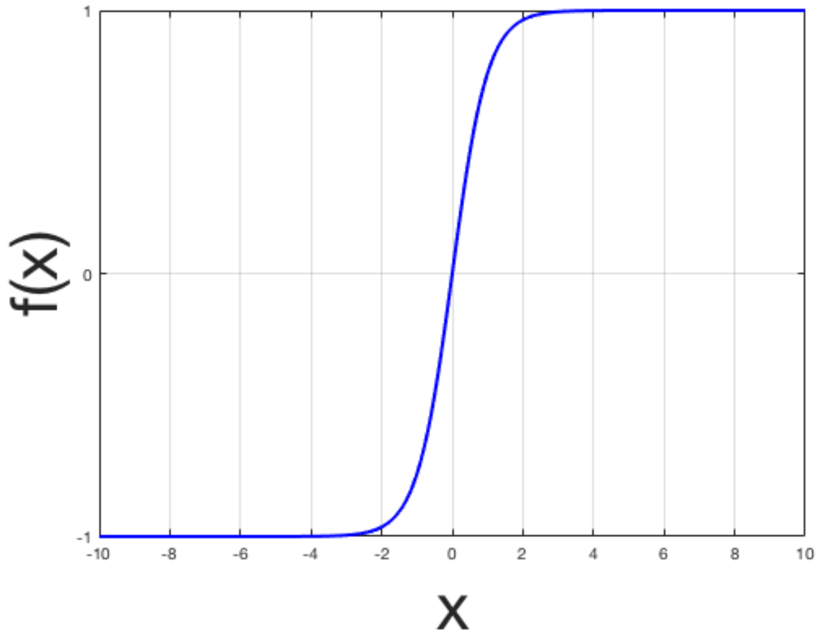
Tanh

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

- It squashes the input real-valued number into range between -1 and 1 (zero-centered)
- tanh is simply a scaled sigmoid, namely,

$$\tanh(x) = 2\sigma(2x) - 1$$

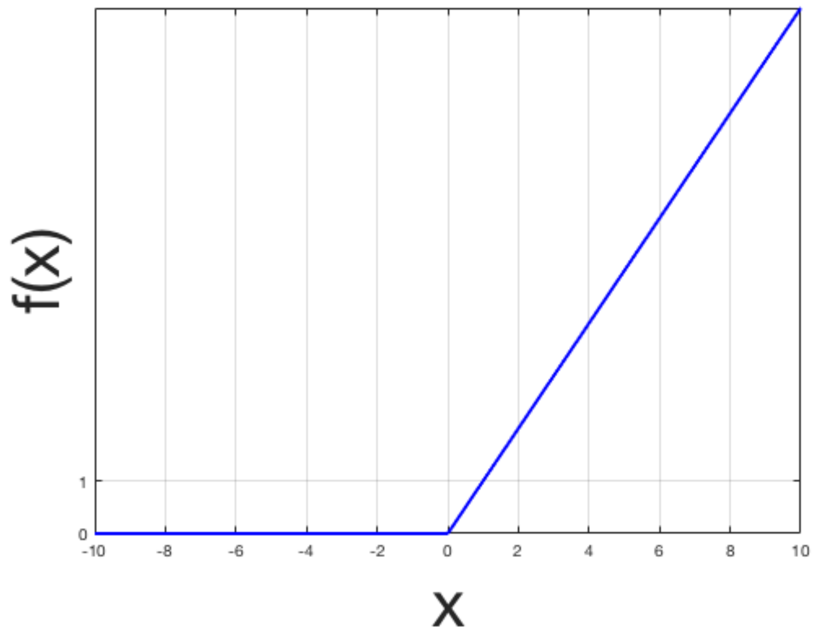
$$\frac{\partial}{\partial x}\tanh(x) = 1 - \tanh(x)^2$$



ReLU

$$f(x) = \max(0, x)$$

- Pros and cons:
  - greatly accelerate the convergence of stochastic gradient descent
  - simple operations with less computation
  - ReLU units can be fragile during training and can "die"



$$\frac{\partial}{\partial x} f(x) = \begin{cases} 0 \text{ if } f(x) \leq 0 \\ 1 \text{ if } f(x) > 0 \end{cases}$$

Others

Leaky ReLU

$$f(x) = 1\{x \le 0\}\alpha x + 1\{x > 0\}x$$

where α is a small constant

- An attempt to fix the "dying ReLU" problem
- The consistency of the benefit across tasks is unclear

Maxout

$$f(x) = \max(w_1 x + b_1, w_2 x + b_2)$$

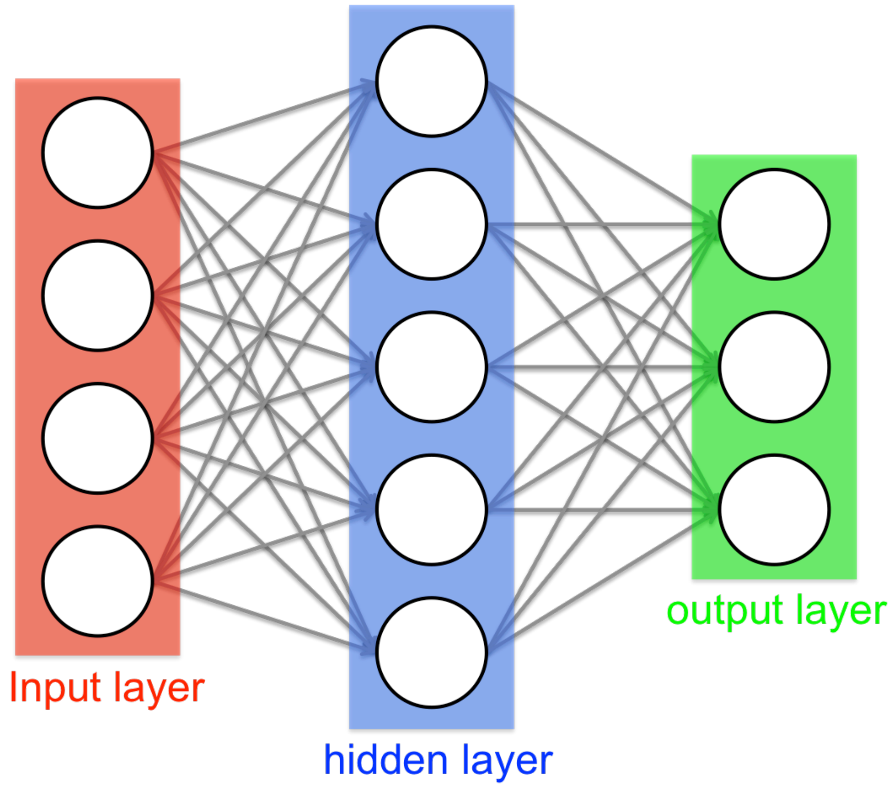
- Generalize the ReLU and its leaky version
- Have all the benefits of a ReLU unit
- Doubles the number of parameters for every single neuron

Which activation function?

- “Use the ReLU non-linearity, be careful with your learning rates and possibly monitor the fraction of ‘dead’ units in a network. If this concerns you, give Leaky ReLU or Maxout a try. Never use sigmoid. Try Tanh, but expect it to work worse than ReLU/Maxout.”

Feedforward neural network – model representation

- Called "feedforward" because there is no feedback connection
- Three types of layers： input layer, hidden layer, output layer



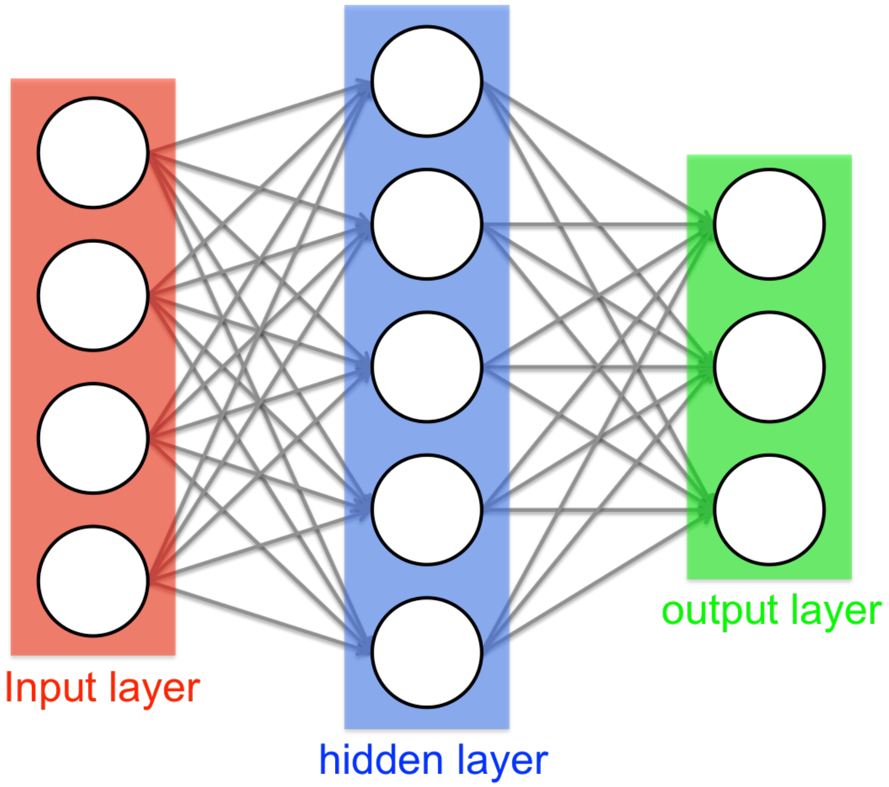
- A 3-layer neural networks, all the layers are **fully-connected**

Notation

- $x^{(i)}$: *i*-th input data
- $W^{(l)}$: matrix of weights controlling function mapping from layer *l* to *l+1*
- $b^{(l)}$: vector of biases controlling function mapping from layer *l* to *l+1*
- $A^{(l)}$: vector of activation output in layer *l*

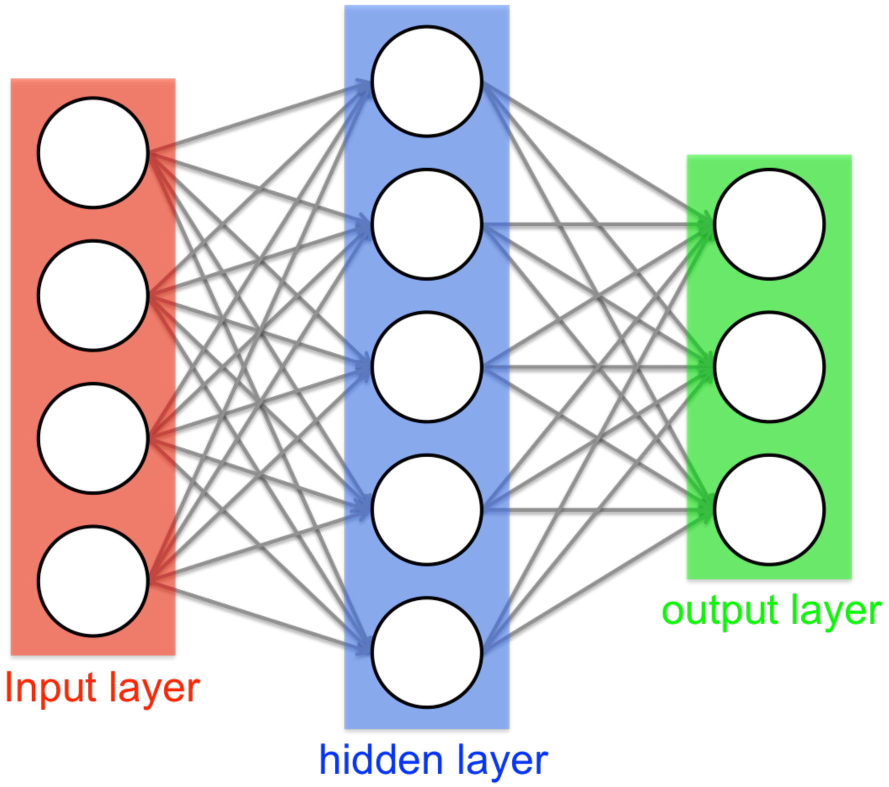
Sizing neural networks

- Two ways to measure the size of neural networks:
  - Number of layers
  - Number of parameters
- Example on the right:
  - 3 layers
  - $5 \times 4 + 3 \times 5 = 35$ weights and $5 + 3 = 8$ biases, 43 parameters in total
- Modern Convolutional Neural Networks contains of 100 million parameters and made up of a few hundred layers, hence called "deep learning"



A concrete example

- Suppose we have a 3-layer neural networks illustrated on the right, given the training examples $x^{(i)}$, our objective is to learn weight W and bias b so that
$$h_{W,b}(x^{(i)}) \approx y^{(i)}$$
- Similar to linear models, now the learning problem is simplified to 3 sub-problems:
  - how to compute hypothesis?
  - how to compute loss?
  - how to compute gradient?



Hypothesis computing

Input layer $\Rightarrow$ hidden layer

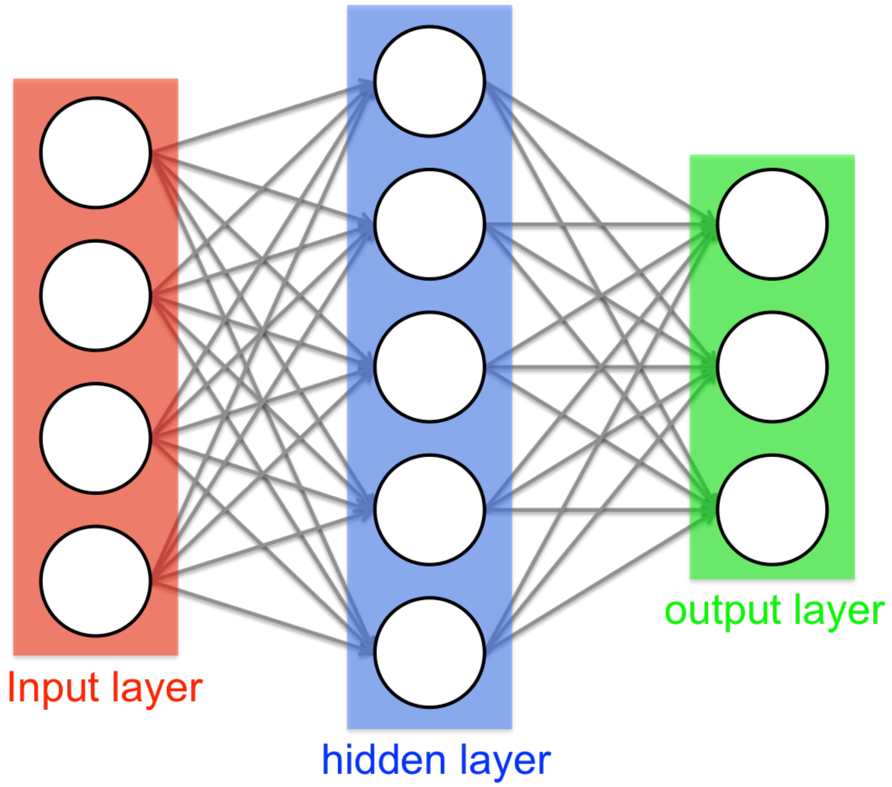
$$Z^{(2)} = W^{(1)}x + b^{(1)}$$

$$A^{(2)} = \max(0, Z^{(2)}) \quad \text{ReLU}$$

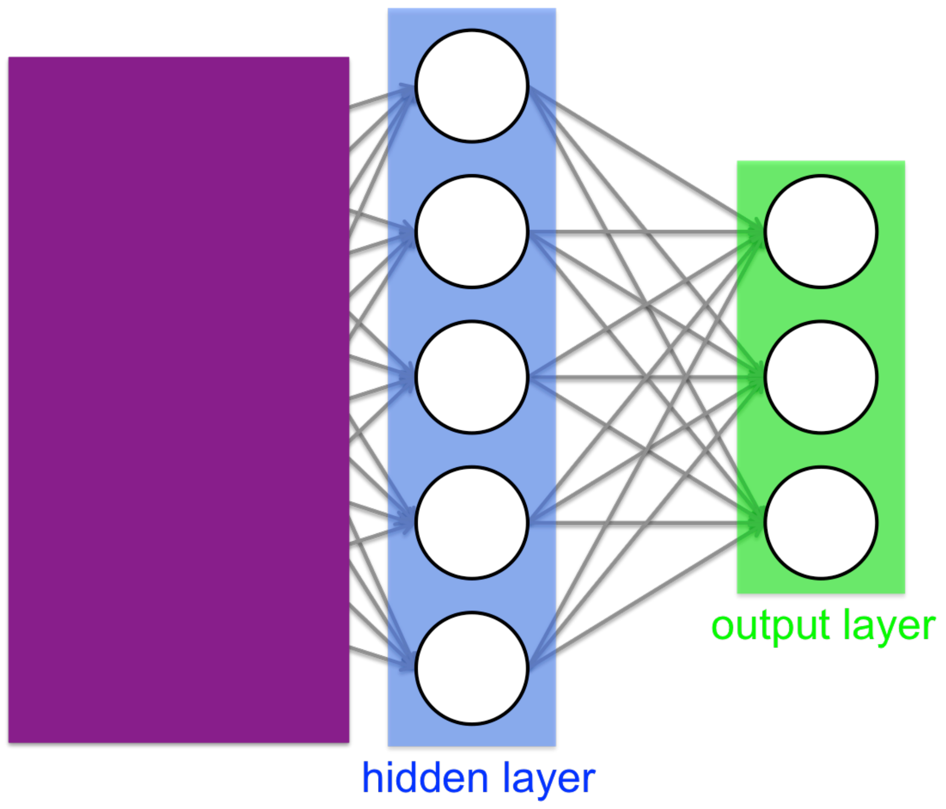
Hidden layer $\Rightarrow$ output layer

$$Z^{(3)} = W^{(2)}A^{(2)} + b^{(2)}$$

$$h_{W,b} = \frac{1}{\sum_{i=1}^{3} e^{Z_i^{(3)}}} \begin{bmatrix} e^{Z_1^{(3)}} \\ e^{Z_2^{(3)}} \\ e^{Z_3^{(3)}} \end{bmatrix}$$



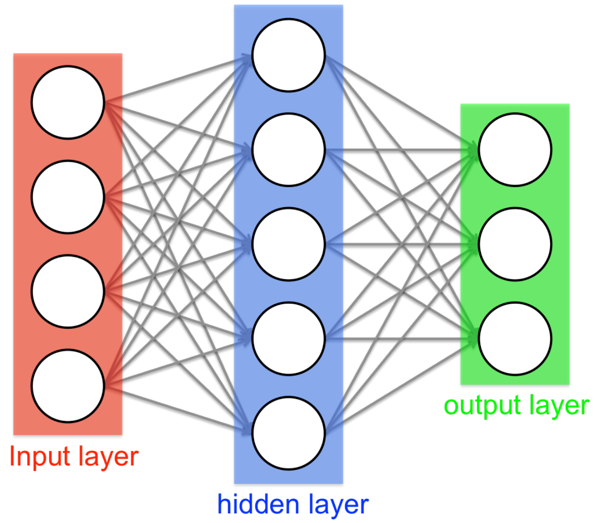
Relation to softmax regression



- The output layer implements the function of softmax
- Neural Network learns its own features and train classification model together

Loss function

$$L(W,b) = -\frac{1}{m}\sum_{i=1}^{m}\sum_{k=1}^{3}(1\left\{y^{(i)} = k\right\}\log h_{W,b}(x^{(i)})) + \frac{\lambda}{2}\sum_{l=1}^{2}\left\|W^{(l)}\right\|^2$$



$$\text{Objective: } \min_{W,b} L(W,b) \Rightarrow \begin{cases} \dfrac{\partial L}{\partial W^{(1)}} \\ \dfrac{\partial L}{\partial b^{(1)}} \\ \dfrac{\partial L}{\partial W^{(2)}} \\ \dfrac{\partial L}{\partial b^{(2)}} \end{cases}$$

Chain rule

- We have solved the first two problems, now how can we compute the gradient?

$$y = f(u)$$
$$u = g(v)$$
$$v = h(x)$$

$$\frac{\partial y}{\partial v} = \frac{\partial y}{\partial u} \cdot \frac{\partial u}{\partial v}$$
$$\frac{\partial y}{\partial x} = \frac{\partial y}{\partial u} \cdot \frac{\partial u}{\partial v} \cdot \frac{\partial v}{\partial x}$$

Chain rule: example

- Consider the function:

$$y = e^{\sin x^2}$$

- It can be decomposed as the composite of three functions:

$$y = f(u) = e^u \quad u = g(v) = \sin v = \sin x^2 \quad v = h(x) = x^2$$

- Their derivatives are:

$$\frac{\partial y}{\partial u} = e^u \quad \frac{\partial u}{\partial v} = \cos v \quad \frac{\partial v}{\partial x} = 2x$$

$$\frac{\partial y}{\partial v} = e^{\sin v} \cdot \cos v$$

$$\frac{\partial y}{\partial x} = e^{\sin x^2} \cdot \cos x^2 \cdot 2x$$

Partial derivative

- Let's consider the partial derivative as an extreme example, namely only one example *x*, we also omit superscript for simplicity.

$$\Delta^{(3)} = -\left(1\left\{ y^{(i)} = k \right\} - \frac{e^{Z^{(3)}}}{\sum_{i=1}^{3} e^{Z_i^{(3)}}} \right)$$

$$\frac{\partial L}{\partial W^{(2)}} = \Delta^{(3)} A^{(2)} + \lambda W^{(2)}$$

$$\frac{\partial L}{\partial b^{(2)}} = \Delta^{(3)}$$

$$\Delta^{(2)} = \Delta^{(3)} W^{(2)} f'(Z^{(2)})$$

$$f'(Z^{(2)}) = \begin{cases} 1 \text{ if } \mathrm{Z}_i^{(2)} > 0 \\ 0 \text{ else} \end{cases}$$

$$\frac{\partial L}{\partial W^{(1)}} = \Delta^{(2)} x + \lambda W^{(1)}$$

$$\frac{\partial L}{\partial b^{(1)}} = \Delta^{(2)}$$

A 4-layer NN – Hypothesis computing

Let $A^{(1)} = x$

$$Z^{(2)} = W^{(1)}A^{(1)} + b^{(1)}$$

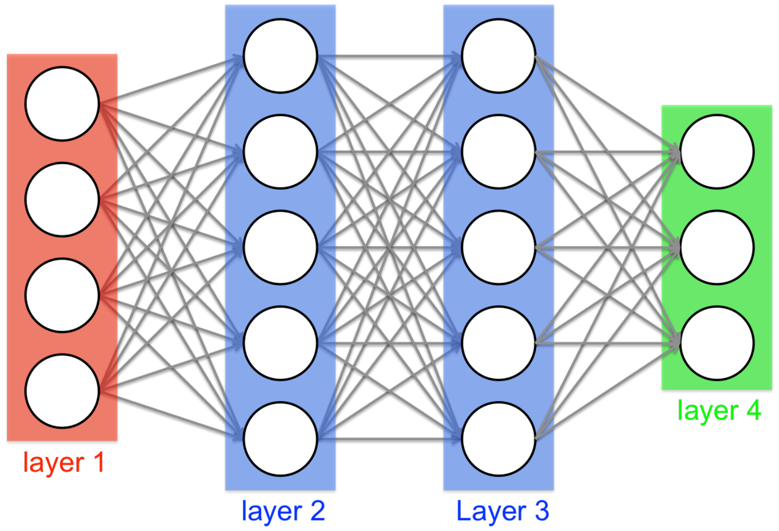
$$A^{(2)} = f(Z^{(2)})$$

$$Z^{(3)} = W^{(2)}A^{(2)} + b^{(2)}$$

$$A^{(3)} = f(Z^{(3)})$$

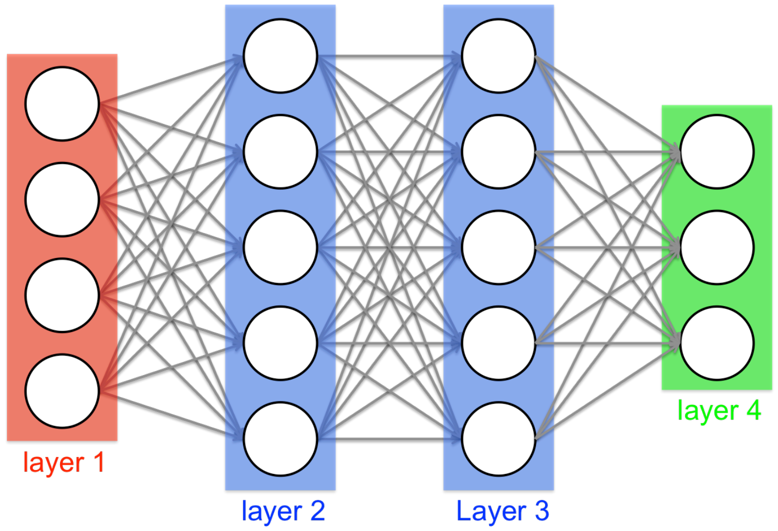
$$Z^{(4)} = W^{(3)}A^{(3)} + b^{(3)}$$

$$\underbrace{h_{W,b}(x)}_{A^{(4)}} = \underbrace{\frac{1}{\sum_{i=1}^{3} e^{Z_i^{(4)}}} \begin{bmatrix} e^{Z_1^{(4)}} \\ e^{Z_2^{(4)}} \\ e^{Z_3^{(4)}} \end{bmatrix}}_{f(Z^{(4)})}$$



A 4-layer NN – Loss function

$$L(W,b) = -\frac{1}{m}\sum_{i=1}^{m}\sum_{k=1}^{3}(1\left\{y^{(i)} = k\right\}\log h_{W,b}(x^{(i)}))$$
$$+\frac{\lambda}{2}\sum_{l=1}^{3}\left\|W^{(l)}\right\|^{2}$$



A 4-layer NN – Partial derivative

$$\Delta^{(4)} = -\left(1\{y^{(i)} = k\} - \frac{e^{Z^{(4)}}}{\sum_{i=1}^{3} e^{Z_i^{(4)}}}\right)$$

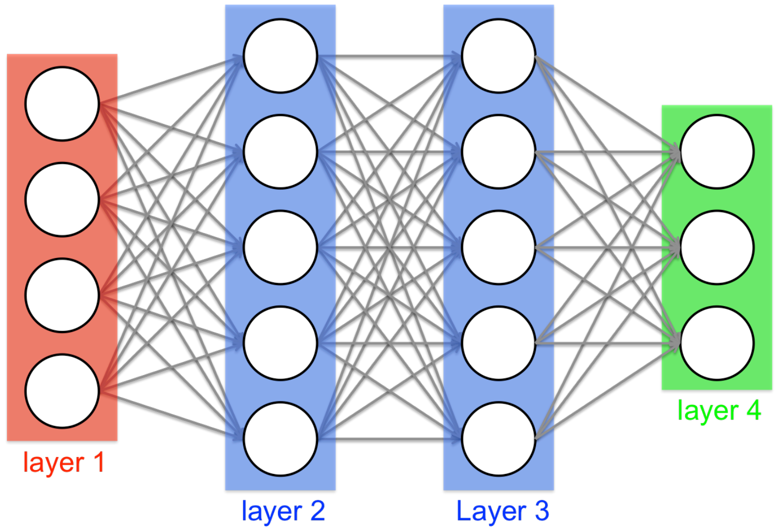
$$\frac{\partial L}{\partial W^{(3)}} = \Delta^{(4)} A^{(3)} + \lambda W^{(3)} \qquad \frac{\partial L}{\partial b^{(3)}} = \Delta^{(4)}$$

$$\Delta^{(3)} = \Delta^{(4)} W^{(3)} f'(Z^{(3)})$$

$$\frac{\partial L}{\partial W^{(2)}} = \Delta^{(3)} A^{(2)} + \lambda W^{(2)} \qquad \frac{\partial L}{\partial b^{(2)}} = \Delta^{(3)}$$

$$\Delta^{(2)} = \Delta^{(3)} W^{(2)} f'(Z^{(2)})$$

$$\frac{\partial L}{\partial W^{(1)}} = \Delta^{(2)} \underbrace{A^{(1)}}_{x} + \lambda W^{(1)} \qquad \frac{\partial L}{\partial b^{(1)}} = \Delta^{(2)}$$



Forward propagation

From layer l to layer $l+1$, the activation is computed as:

$$Z^{(l+1)} = W^{(l)}A^{(l)} + b^{(l)}$$

$$A^{(l+1)} = f(Z^{(l+1)})$$

When $l=1$, $A^{(1)} = x$.

When l is equal to the number of layers N, $f(x)$ is the loss function of linear model, e.g., softmax in previous example.

Backward propagation

From layer $l+1$ to layer l, the partial derivative is computed as:

$$\frac{\partial L}{\partial W^{(l)}} = \Delta^{(l+1)} A^{(l)} + \lambda W^{(l)}$$

$$\frac{\partial L}{\partial b^{(l)}} = \Delta^{(l+1)}$$

$$\Delta^{(l)} = \Delta^{(l+1)} W^{(l)} f'(Z^{(l)})$$

When $l = 1$, $A^{(1)} = x$.

When $l = N$, $\Delta^{(L)}$ is the partial derivative of linear model.

Weight and bias initialization

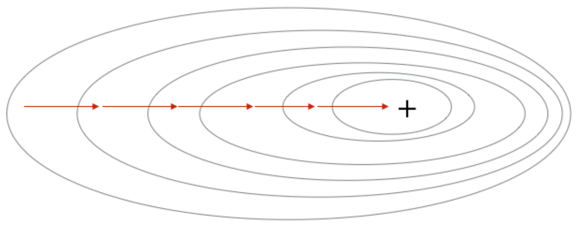
- Set all initial weights to zero ➡ same output ➡ same gradient ➡ same parameters update
- Initialize weights randomly to serve the purpose of **symmetry breaking**
- Solutions:
  - Random initialization with small numbers
  - Sparse initialization
  - Bias initialization: zero initialization
  - …

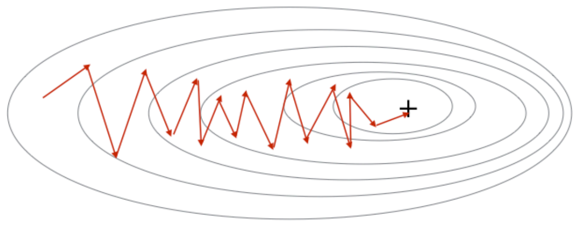
Batch and Mini-batch algorithms

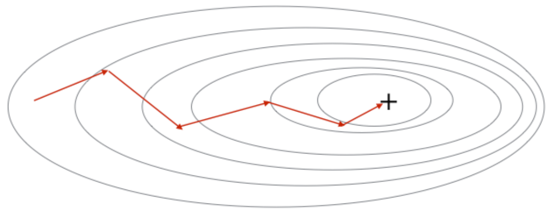
- Motivation: computational cost and redundancy in the training set
- Batch/deterministic method:
  - process all the training example simultaneously in a large batch
  - fast to converge
  - computation is very expensive when your training set is very large
- Stochastic/online method:
  - only a single example at a time
  - suitable when examples are drawn from a stream of continually created examples
  - hard to converge
- Mini-batch/mini-batch stochastic:
  - use more than one but fewer than all the training examples

Why mini-batch?

- Larger batches provide a more accurate estimate of the gradient
- Multicore architectures are usually underutilized by extremely small batches
- Some kinds of hardware achieve better runtime with specific sizes of arrays, e.g., GPU
- Small batches can offer a regularizing effect, perhaps due to the noise they add to the learning process
  - but may require small learning rate
  - may increase number of steps for convergence


Gradient Descent


Stochastic Gradient Descent


Mini-Batch Gradient Descent

Mini-batch

- It is crucial that the mini-batches be selected randomly) shuffle your data before selecting mini-batches
- In deep learning, one **epoch** means one pass of the full training set
- Make several epochs through the training set
  - the first epoch follows the unbiased gradient of the generalization error
  - additional epochs decrease gap between training error and test error
- In GPU, it is common for power of 2 batch sizes to offer better runtime

Gradient check - motivation

- Too many weights, not sure if the analytic gradient is correct or not?!
  - Solution: compare relative error between analytic gradient and numerical gradient

Example: given a function $f(x) = x^2$, we can compute its gradient when $x = 2$ in both analytic and numerical forms:

$$f_a'(x) = 2x\Big|_{x=2} = 4$$

$$f_n'(x) = \frac{f(x+\varepsilon) - f(x-\varepsilon)}{2\varepsilon}\Bigg|_{x=2} = \frac{2.0001^2 - 1.9999^2}{0.0002} \approx 4$$

$$\left|f_a'(x) - f_n'(x)\right| < \text{threshold}$$

Gradient check on loss function

Suppose we have a loss function $L(\theta)$ with a set of parameters $\theta = \left[\ \theta_1, \quad \theta_2, \quad \ldots, \quad \theta_n\ \right]$, let ε is a very small number, say 10^{-4}, a correct analytic gradient should meet:

$$\frac{\partial L}{\partial \theta_1} \approx \frac{L(\ \theta_1 + \varepsilon, \quad \theta_2, \quad \ldots, \quad \theta_n\ ) - L(\ \theta_1 - \varepsilon, \quad \theta_2, \quad \ldots, \quad \theta_n\ )}{2\varepsilon}$$

$$\frac{\partial L}{\partial \theta_2} \approx \frac{L(\ \theta_1, \quad \theta_2 + \varepsilon, \quad \ldots, \quad \theta_n\ ) - L(\ \theta_1, \quad \theta_2 - \varepsilon, \quad \ldots, \quad \theta_n\ )}{2\varepsilon}$$

$$\vdots$$

$$\frac{\partial L}{\partial \theta_n} \approx \frac{L(\ \theta_1, \quad \theta_2, \quad \ldots, \quad \theta_n + \varepsilon\ ) - L(\ \theta_1, \quad \theta_2, \quad \ldots, \quad \theta_n - \varepsilon\ )}{2\varepsilon}$$

Gradient check - overview

Compute analytic gradient f'_a

Estimate numerical gradient f'_n

Make sure they have a small relative error, say $1e-7$

$$\text{relative error } = \frac{|f'_a - f'_n|}{|f'_a| + |f'_n|}$$

Q: why not just use up term?

Gradient check can be generalized to check gradient of any loss function

Summary

- Forward and backward propagation allows us to design a very “deep” neural networks
- Use ReLU as activation function
- Mini-batch gradient descent is applied to neural network instead of gradient descent or stochastic gradient descent
- Random initialize weights for the purpose of symmetry breaking
- Gradient check is very useful when you implement your own loss function/layers