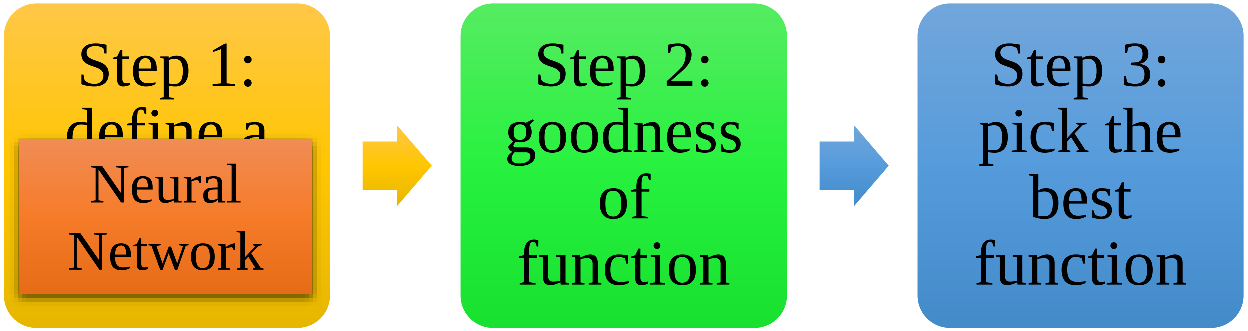
Lecture 2 Deep Feedforward Networks

课程：机器学习与深度学习

Three Steps for Deep Learning



Step 1: define a

Neural Network

Step 2: goodness of function

Step 3: pick the best function

Deep Learning is as simple as linear model……

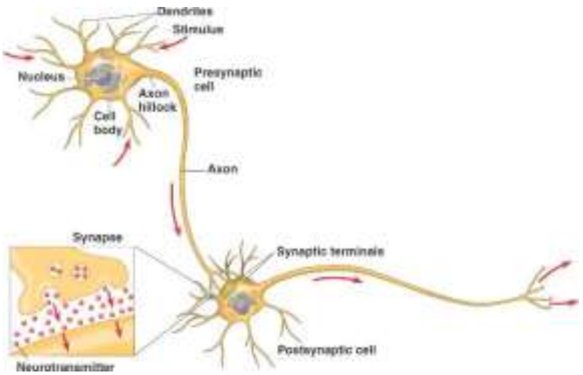
Overview

- Model Architectures
    - Artificial neurons
    - Activation function and saturation
    - Feedforward neural nets
- How to train a neural net
    - Loss Function Design
    - Optimization
        - Gradient Descent and Stochastic Gradient Descent
        - Back-propagation
    - Advanced Training tips

The Perceptron

- Invented in 1954 by Frank Rosenblatt
- Inspired by neurobiology

Artificial Neuron

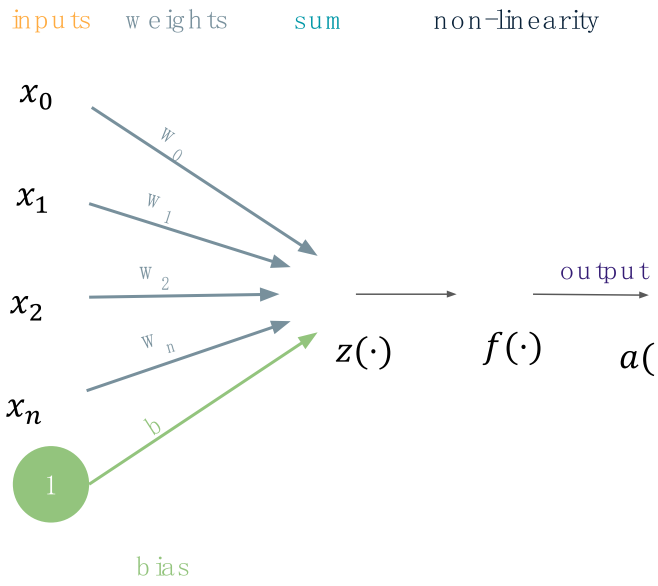


- Each neuron is a very simple function
- Pre-activation: $z(x) = \sum_i w_i x_i + b = w^T x + b$
- Output activation: $a(x) = f\big(z(x)\big) = f(w^T x + b)$

$f(\cdot)$: nonlinear activation function

w: weight

b: bias term

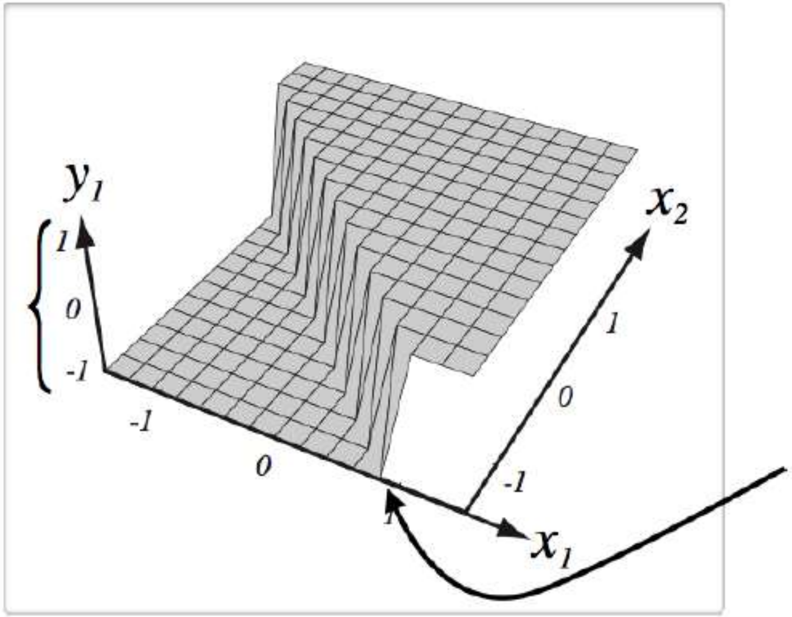


Artificial Neuron

- Output activation

$$a(x) = f\big(z(x)\big) = f(w^T x + b)$$

Range is determined by $g(\cdot)$



Bias only changes the position of the riff

(from Pascal Vincent's slides)

Activation Function

- Linear activation function
- Sigmoid activation function
- Hyperbolic tangent activation function
- Rectified linear (ReLU) activation function
- Softmax activation function

Non-linear activation function, frequently used in deep neural networks.

Linear Activation Function

- No input squashing
- No nonlinear transformation

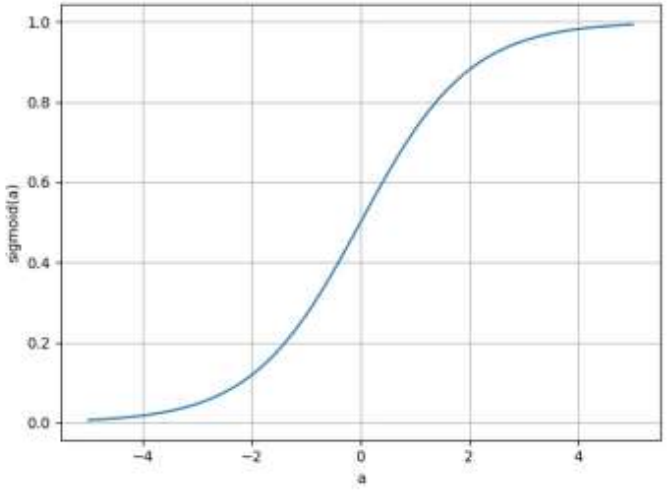
$$f(z) = z$$

- Why non-linearity?
  - **Without non-linearity**, deep neural networks work the same as linear transform
    - $W_1(W_2.x) = (W_1W_2)x = Wx$
  - **With non-linearity**, networks with more layers can approximate more complex function

Sigmoid Activation Function

- Squashes the neuron's output to (0,1)
- Bounded
- Strictly increasing

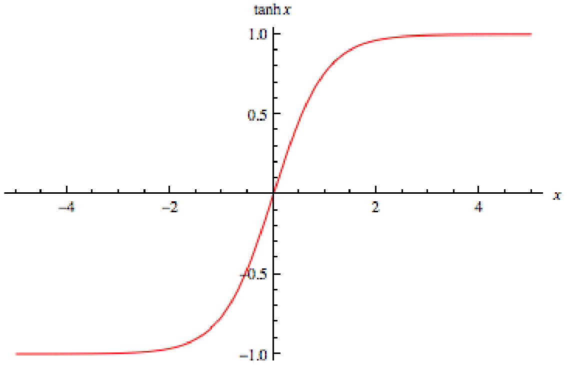
$$f(z) = \sigma(z) \overset{\text{def}}{=} \frac{1}{1 + \exp(-z)}$$



Hyperbolic Tangent (“tanh”) Function

- Squashes the neuron’s output to (-1,1)
- Can be positive or negative
- Bounded
- Strictly increasing

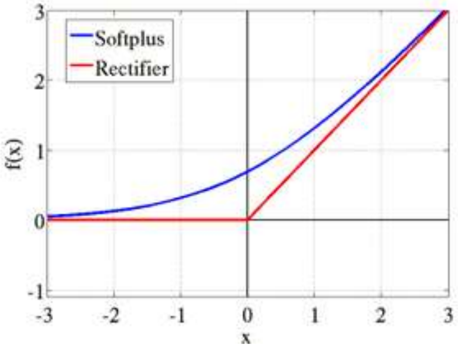
$$f(z) = \tanh(z) \overset{\text{def}}{=} \frac{\exp(2z)\ - 1}{\exp(2z) + 1}$$



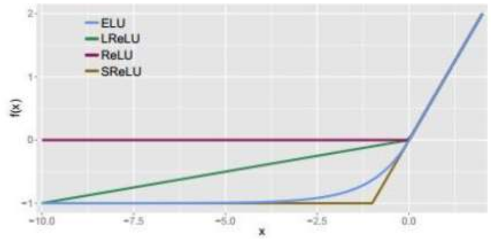
Rectified Linear Activation Function (ReLU)

- Tends to produce units with sparse activities
- No upper-bound
- Increasing

$$f(z) = \mathrm{reclin}(z) \overset{\text{def}}{=} \max(z, 0)$$



- ReLU variants:
  - Shift ReLU: $\max(-1, z)$
  - Leaky ReLU: $\max(0.1z, z)$
  - Parameter ReLU: $\max(\mu z, z)$
  - Exponential Linear Units: $\max\big(z, \mu(\exp(z) - 1)\big)$
  - Maxout: $\max(w_1^T z + b_1, w_2^T z + b_2)$



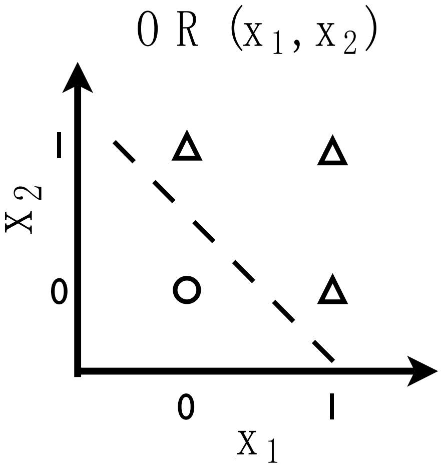
Softmax Activation Function

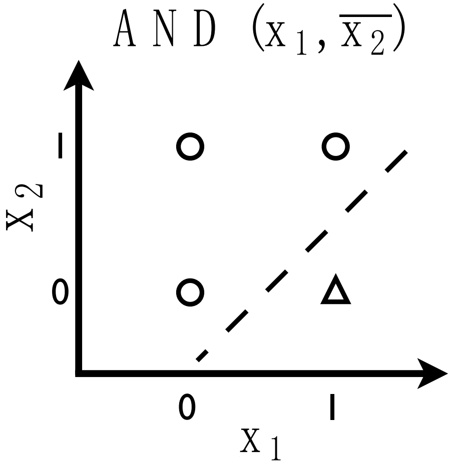
- In multi-class classification (C classes), we need to
  - generate multiple output: $\mathbf{z} \in \mathbb{R}^C$ (in vector form)
  - estimate the conditional probability of each class: $p(y = i|\mathbf{z}) = \frac{\exp z_i}{\sum_c \exp z_c}$
- Strictly positive
- Sum up to one

$$\mathbf{f}(\mathbf{z}) = \mathrm{softmax}(\mathbf{z}) = \left[\frac{\exp z_1}{\sum_c \exp z_c} \ldots \frac{\exp z_C}{\sum_c \exp z_c}\right]^T$$

Capacities of a Single Neuron

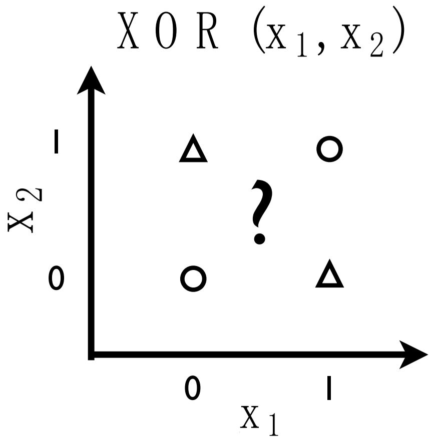
- Solve linearly separable problems


OR (x_1, x_2)

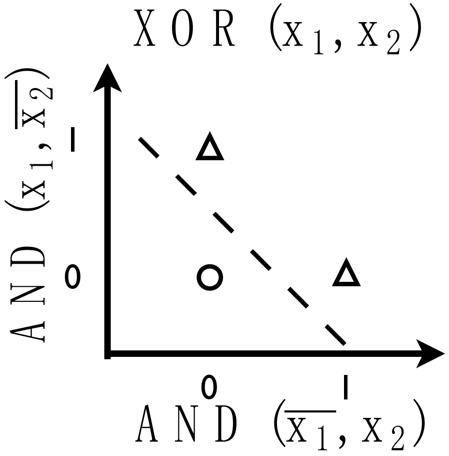

AND $(x_1, \overline{x_2})$

Capacities of a Single Neuron

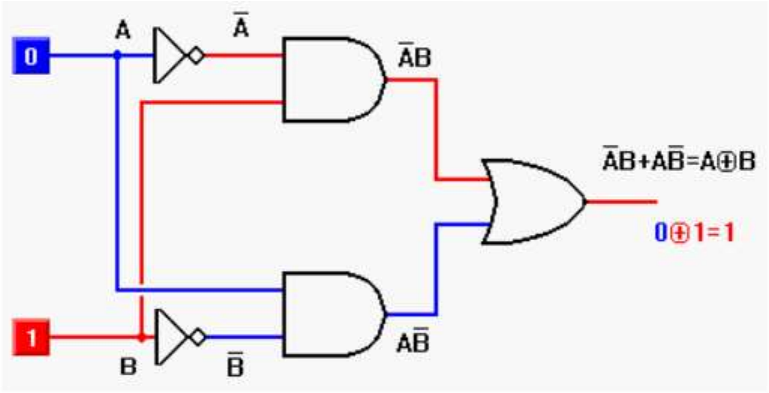
- Can’t solve linearly inseparable problems



How to implement XOR?



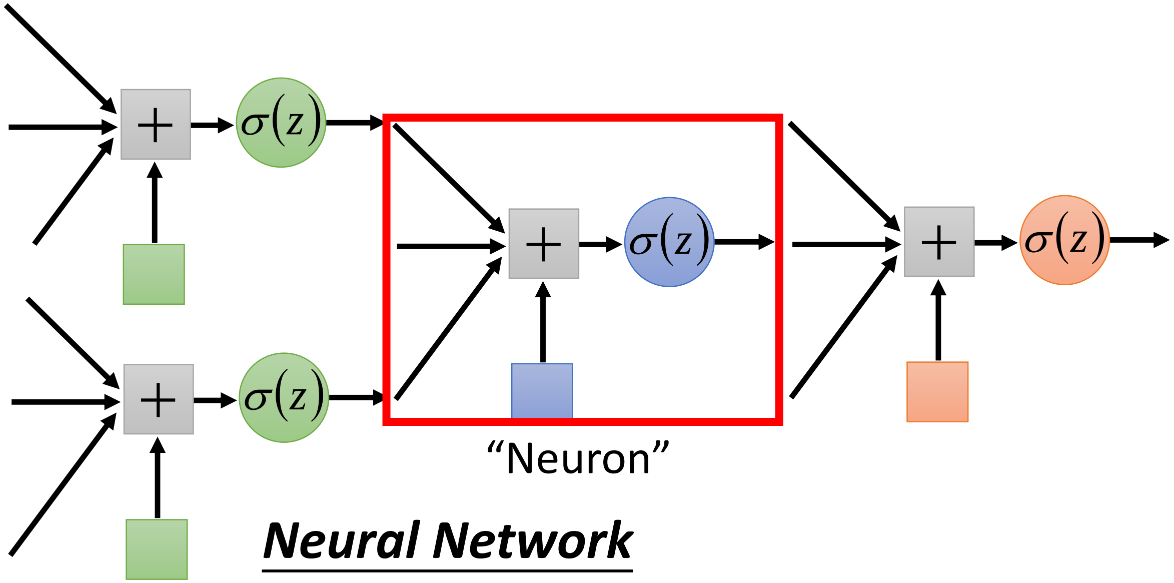
Input		Output
A	B	
0	0	0
0	1	1
1	0	1
1	1	0



$$A\ XOR\ B = AB' + A'B$$

Multiple operations can produce more complicate output

Neural Network



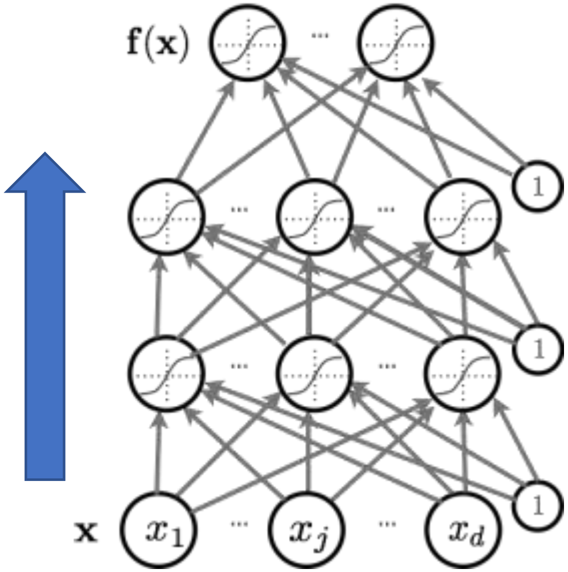
Neural Network

Different connection leads to different network structures

Network parameter θ: all the weights and biases in the "neurons"

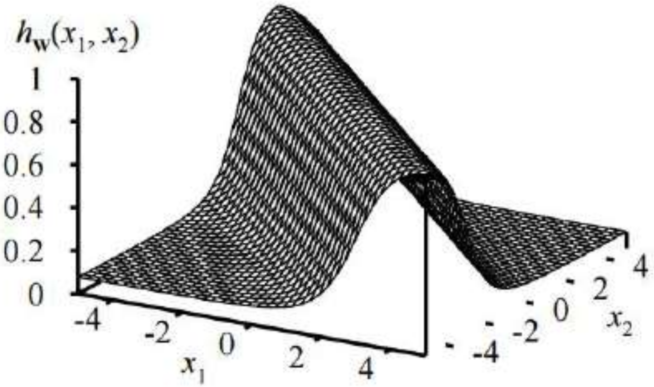
Multilayer Feedforward Neural Networks

- Each neuron in one layer has directed connections to the neurons of the subsequent layer
- Information propagates from input x to output $f(x)$, through many hidden layers

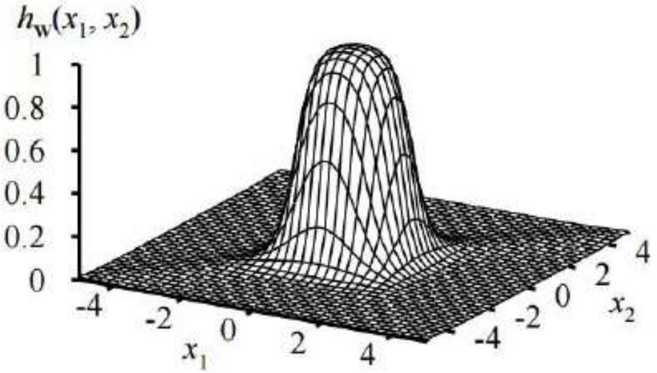


Expressions of Multi-Layer Neural Network

Continuous function w/ 2 layers



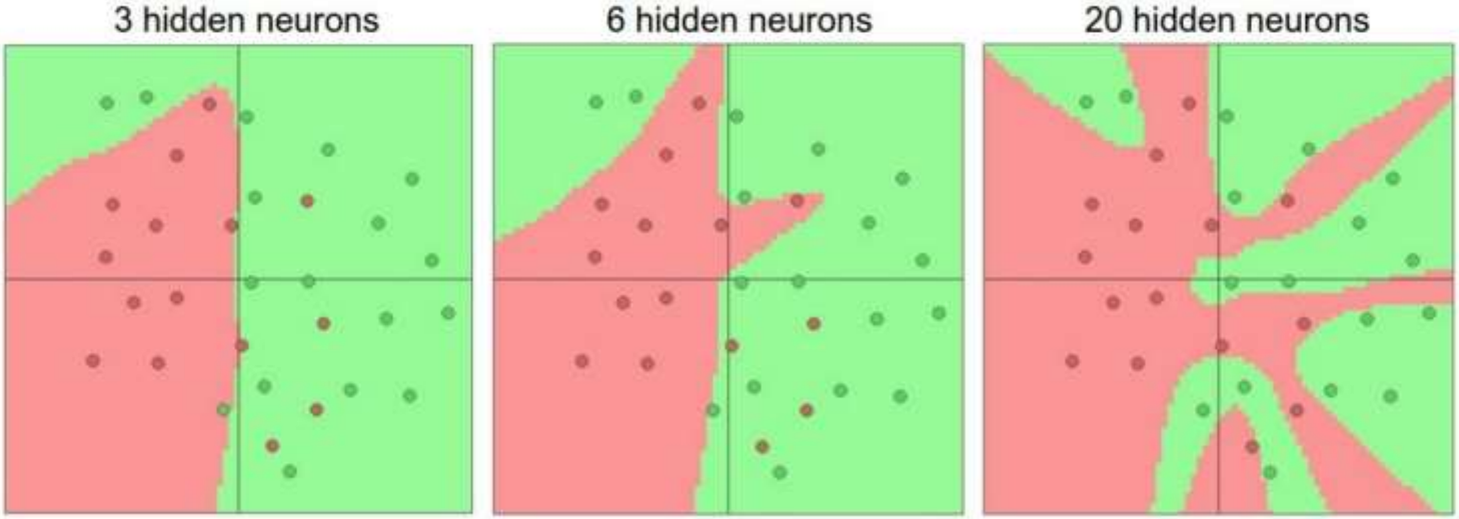
Continuous function w/ 3 layers



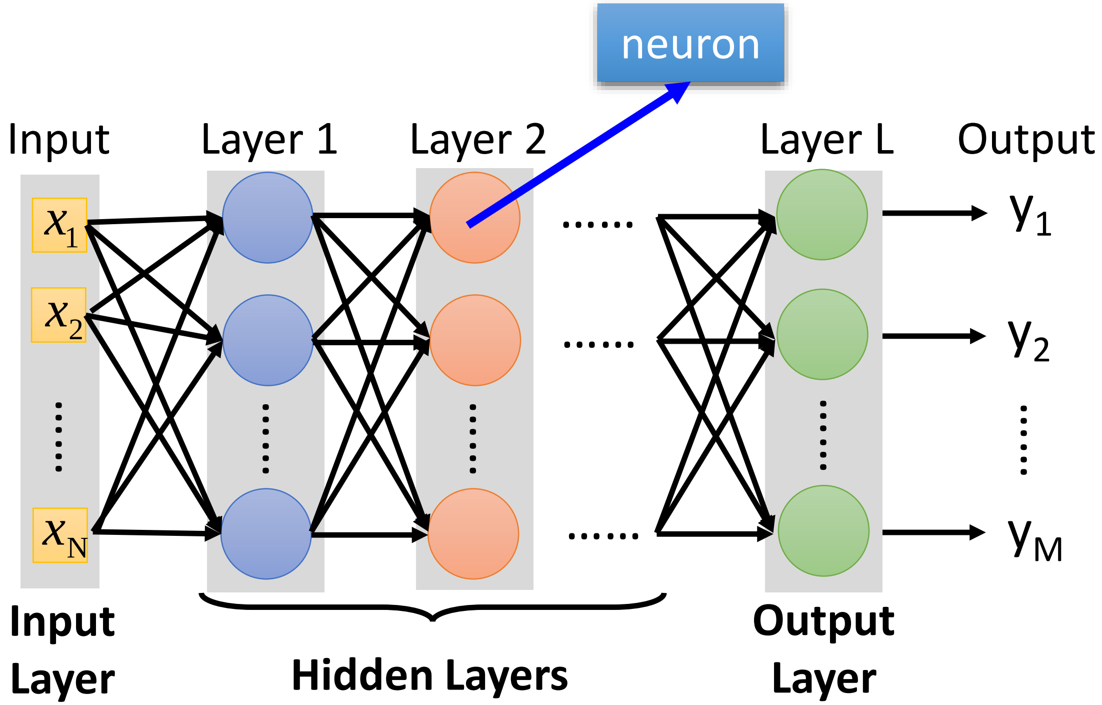
Multiple layers enhance the model expression -> the model can approximate more complex functions

Setting the Number of Neuros and Layers

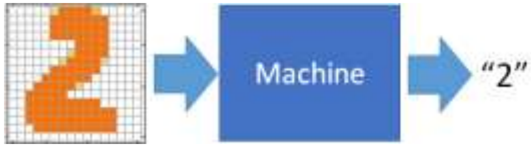
- More neuros = more capacity
- More layers = more capacity



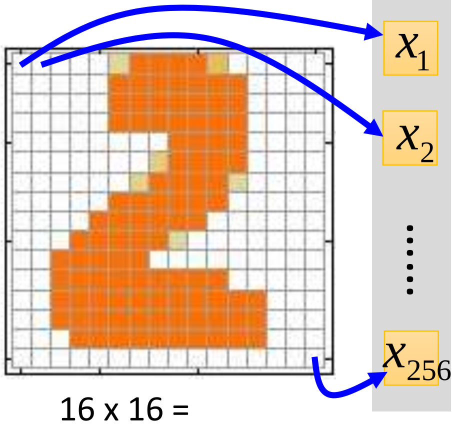
Fully Connect Feedforward Network



Example Application



Input

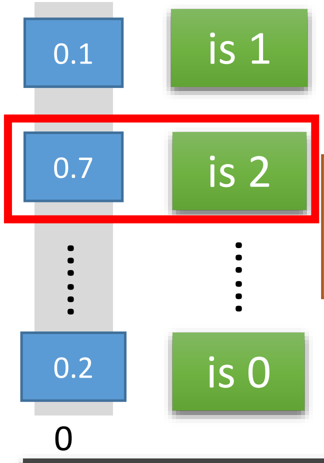


16 x 16 = 256

Ink → 1

No ink → 0

Output

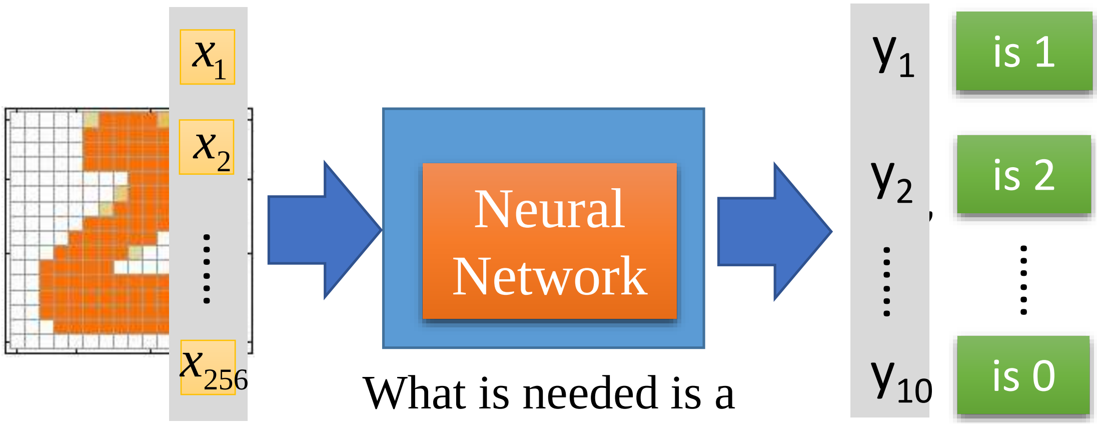


The image is "2"

Each dimension represents the confidence of a digit.

Example Application

- Handwriting Digit Recognition

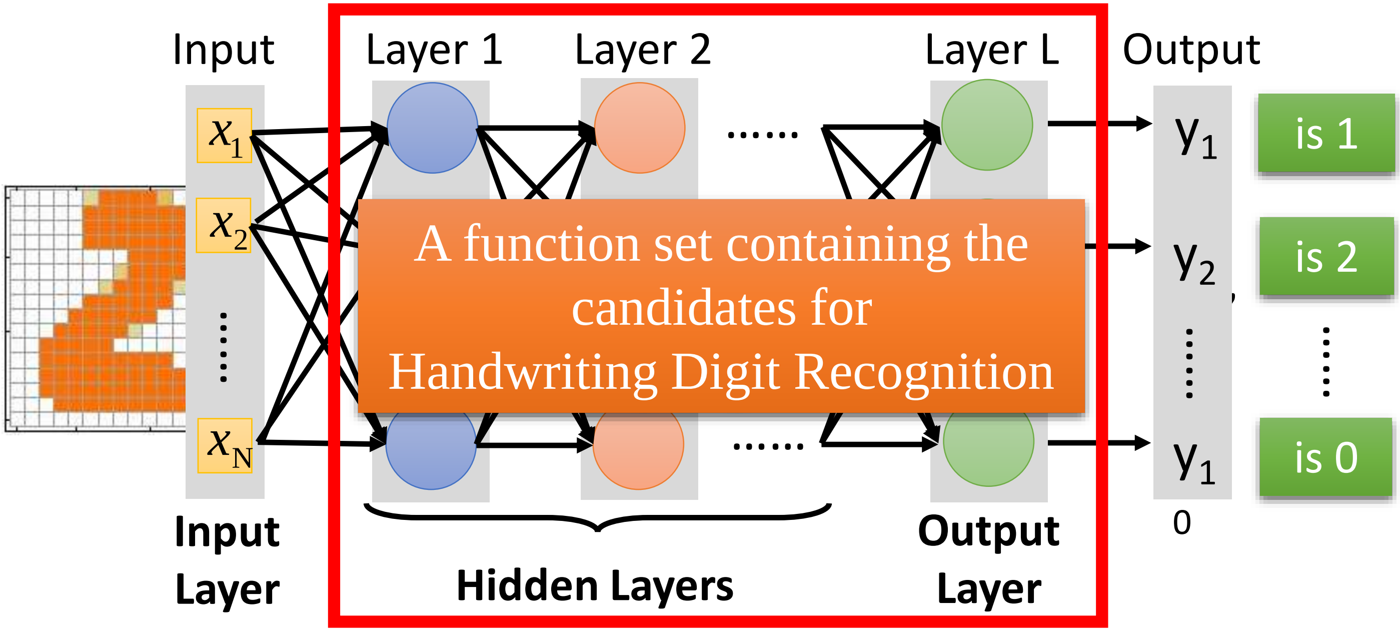


What is needed is a function ……

Input:
256-dim
vector

output:
10-dim vector

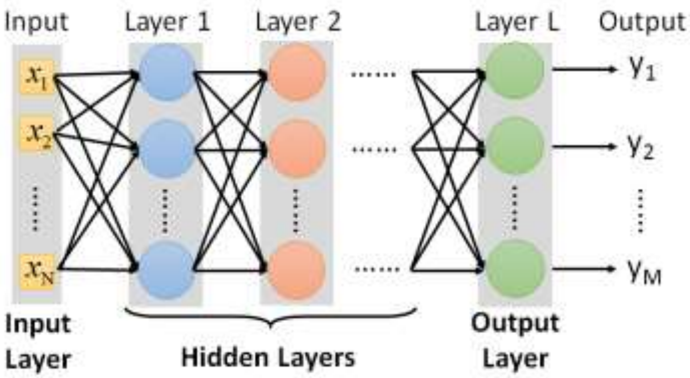
Example Application



A function set containing the candidates for Handwriting Digit Recognition

You need to decide the network structure to let a good function in your function set.

FAQ



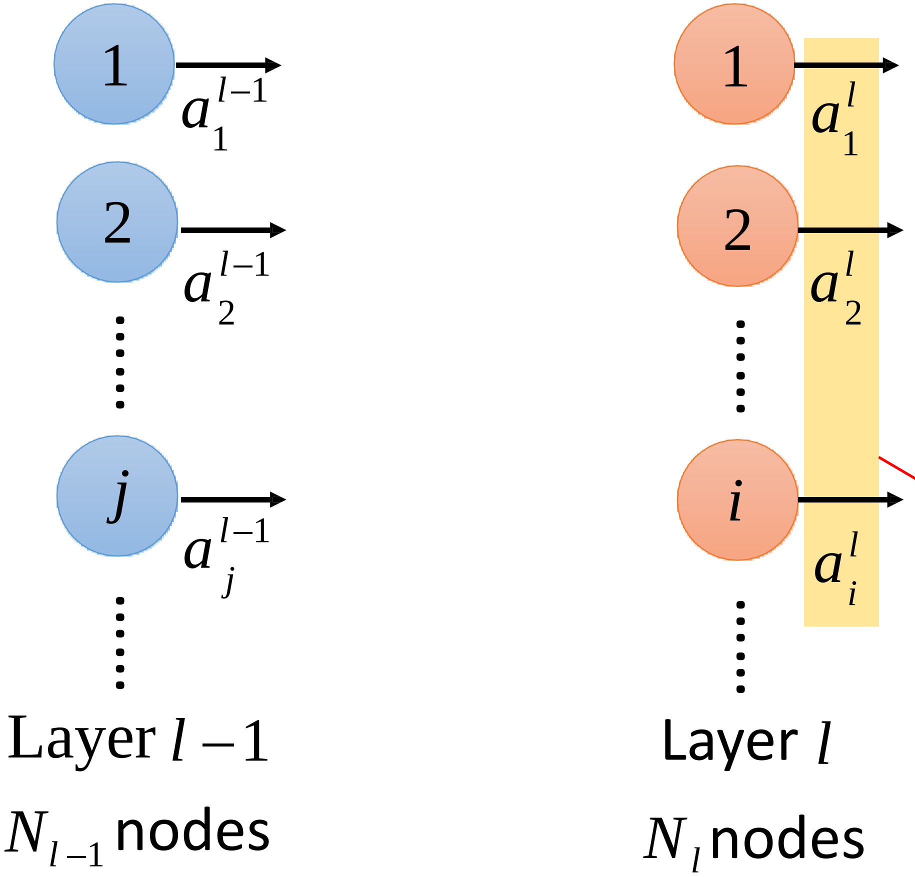
- Q: How many layers? How many neurons for each layer?

Trial and Error + Intuition

- Q: Can the structure be automatically determined?
  - E.g. Evolutionary Artificial Neural Networks
- Q: Can we design the network structure?

Convolutional Neural Network (CNN)

Notation Definition



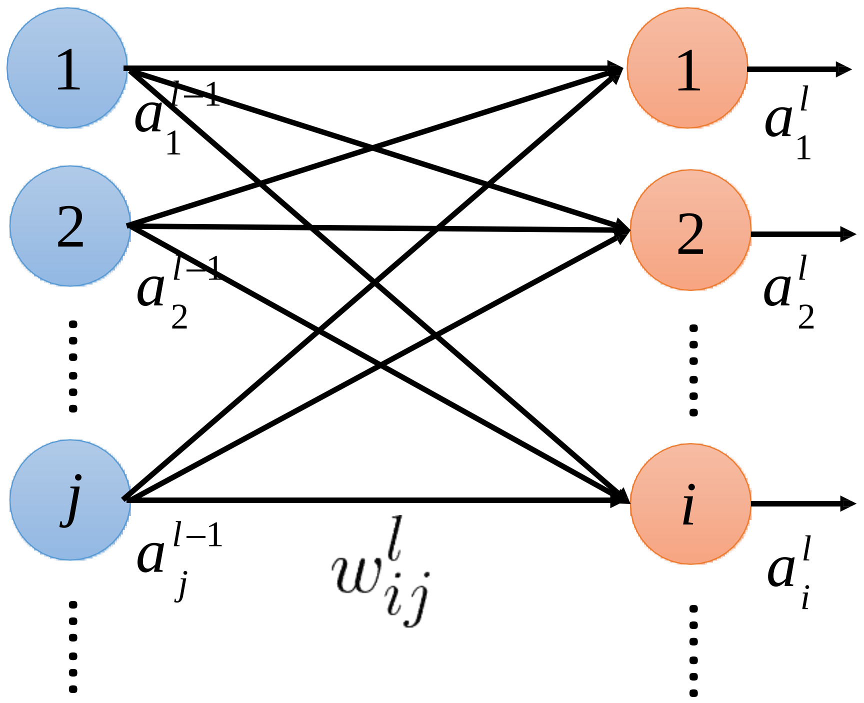
Output of a neuron:

a_i^l → layer l, neuron i

$$a^l = \begin{bmatrix} \vdots \\ a_i^l \\ \vdots \end{bmatrix}$$

output of one layer→ a vector

Notation Definition



Layer $l-1$

N_{l-1} nodes

Layer l

N_l nodes

w_{ij}^l

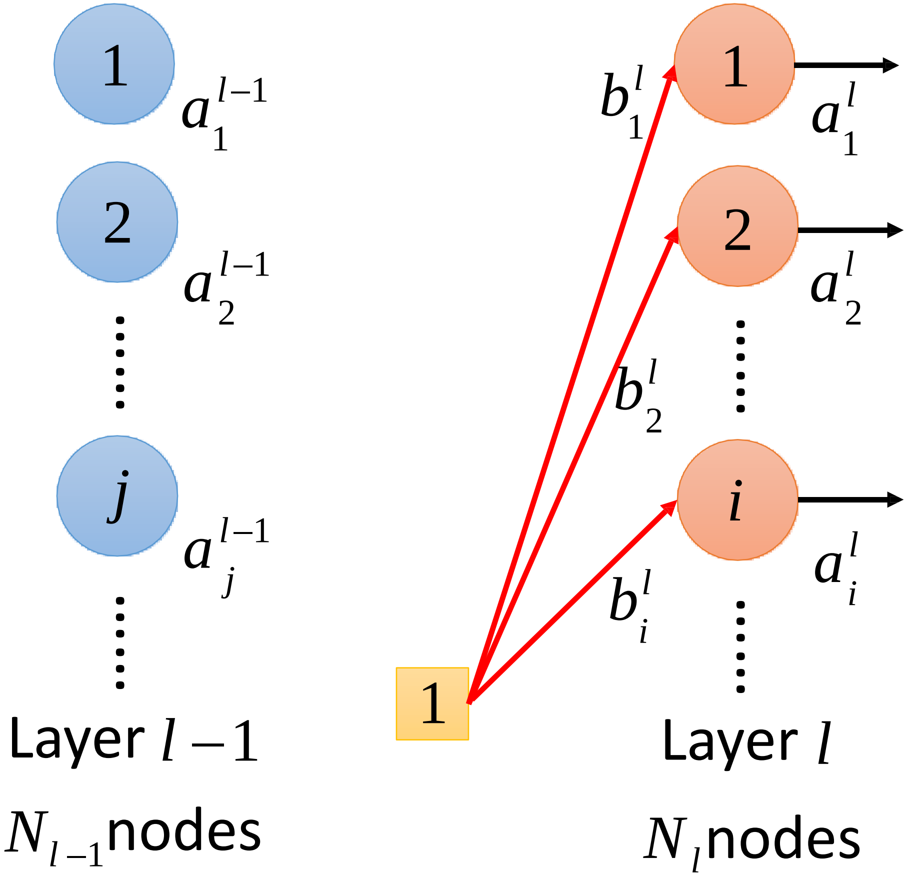
layer l -1
to layer l

from neuron *j* (layer *l*-*1*)
to neuron *i* (layer *l*)

$$W^l = \begin{array}{cc} \overleftrightarrow{\quad N_{l-1} \quad} & \\ \begin{bmatrix} w_{11}^l & w_{12}^l & \cdots \\ w_{21}^l & w_{22}^l & \\ \vdots & & \ddots \end{bmatrix} & \updownarrow N_l \end{array}$$

weights between two layers
→ a matrix

Notation Definition

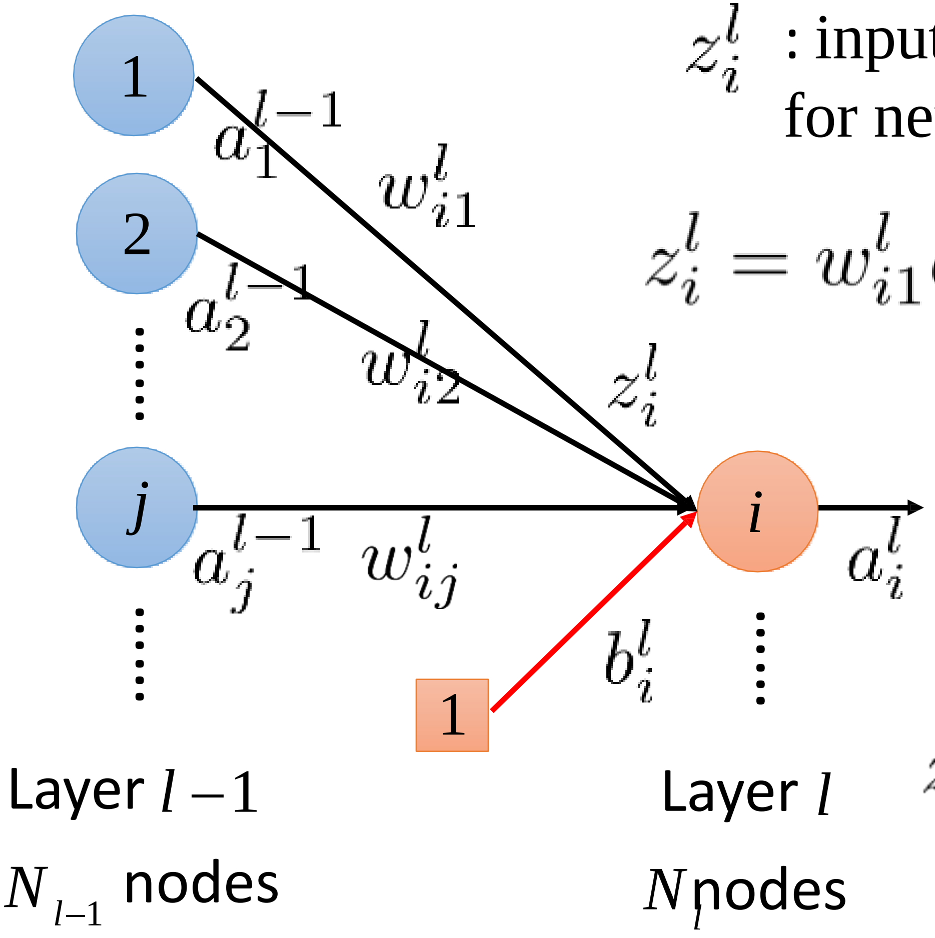


b_i^l : bias for neuron i at layer l

$$b^l = \begin{bmatrix} \vdots \\ b_i^l \\ \vdots \end{bmatrix}$$

bias of all neurons at each layer→ a vector

Notation Definition



z_i^l : input of the activation function for neuron *i* at layer *l*

$$z_i^l = w_{i1}^l a_1^{l-1} + w_{i2}^l a_2^{l-1} + \ldots + b_i^l$$

$$z_i^l = \sum_{j=1}^{N_{l-1}} w_{ij}^l a_j^{l-1} + b_i^l$$

$$z^l = \begin{bmatrix} \vdots \\ z_i^l \\ \vdots \end{bmatrix}$$

activation function input at each layer→ a vector

Notation Summary

a_i^l : output of a neuron

a^l : output vector of a layer

z_i^l : input of activation function

z^l : input vector of activation function for a layer

w_{ij}^l : a weight

W^l : a weight matrix

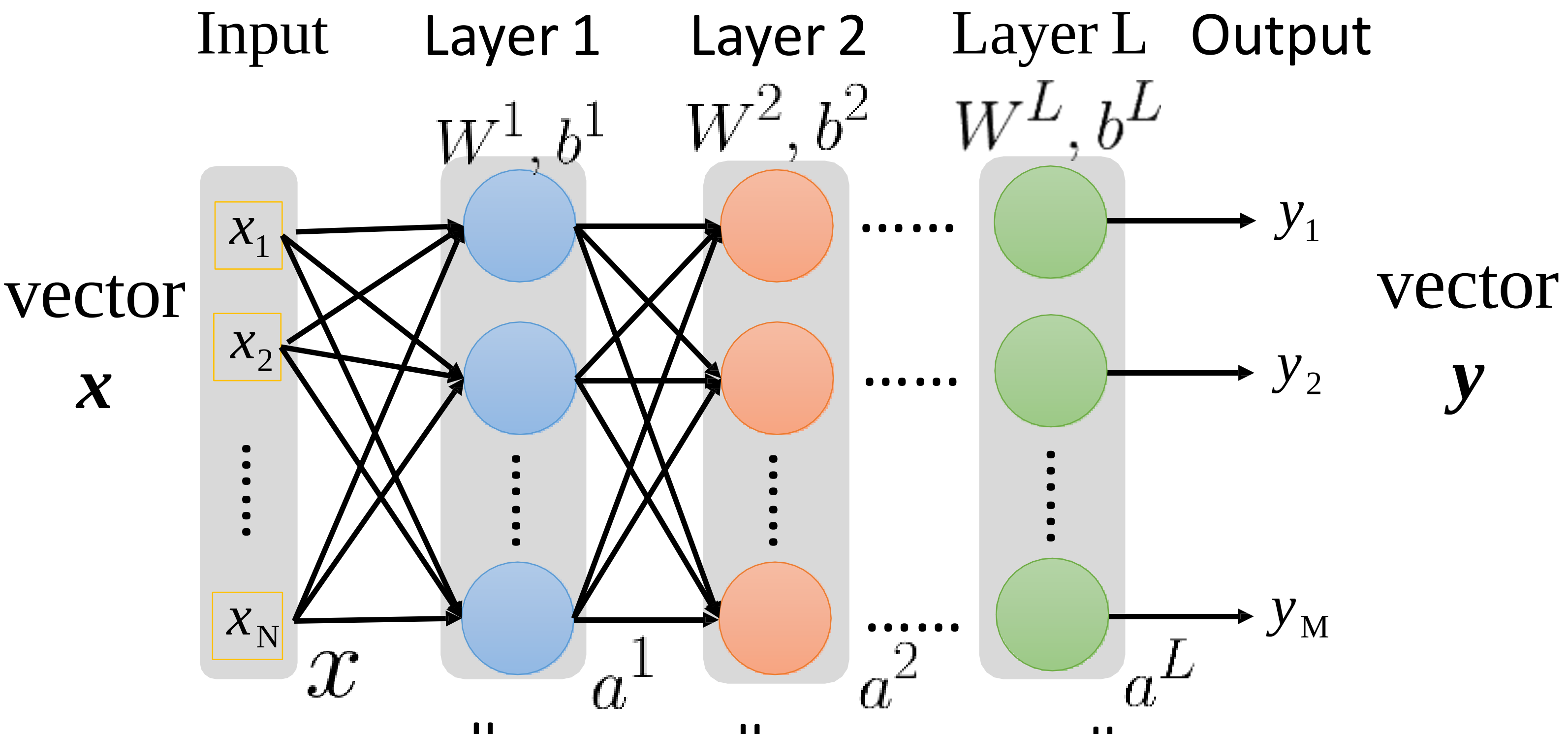
b_i^l : a bias

b^l : a bias vector

Neural Network Formulation

$$f : R^N \rightarrow R^M$$

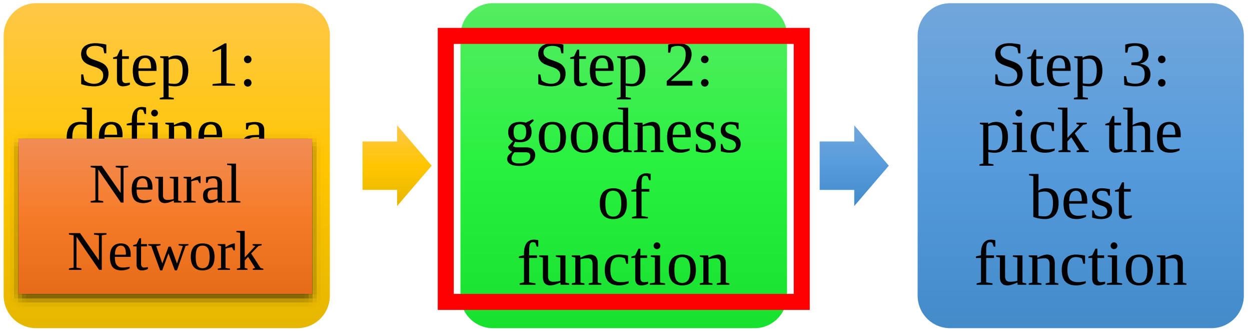
Fully connected feedforward network



$$\sigma(W^1 x + b^1) \quad \sigma(W^2 a^1 + b^2) \quad \sigma(W^L a^{L-1} + b^L) = y$$

$$y = f(x) = \sigma(W^L \cdots \sigma(W^2 \sigma(W^1 x + b^1) + b^2) \ldots + b^L$$

Three Steps for Deep Learning



Step 1: define a

Neural Network

Step 2: goodness of function

Step 3: pick the best function

Function = model parameters

Forward propagation

$$y = f(x) = \sigma(W^L \cdots \sigma(W^2\sigma(W^1x + b^1) + b^2)... + b^L)$$

Different parameters W and b → different functions

Formal definition:

$$f(x;\theta);\ \theta = \{W^1, b^1, W^2, b^2, \cdots, W^L, b^L\}$$

Pick a function f $=$ pick a set of model parameters θ

Training

- Empirical risk minimization $J(\theta)$

$$\arg\min_{\boldsymbol{\theta}} \frac{1}{T}\sum_{t} \underbrace{l(f(\mathbf{x}^{(t)};\boldsymbol{\theta}), y^{(t)})}_{\text{Loss function}} + \underbrace{\lambda\Omega(\boldsymbol{\theta})}_{\text{Regularizer}}$$

- Learning is cast as optimization
  - Find a model parameter set that minimize $J(\theta)$
  - Loss function can sometimes be viewed as a surrogate for what we want to optimize

Loss Function

- In discriminative model (判别模型), model $y|x$.
- Learning the maximum likelihood, equivalently the cross entropy between training data and model distribution:

$$l(\theta) = -E_{x,y\sim\text{Data}} \log p(y|x)$$

- The specific form of loss function depends on the model distribution $p(\cdot)$

Loss Functions

- Loss function evaluates the performance of our model, it is chosen according to the output units
  - Normal: $\hat{y} = \boldsymbol{w}^T \boldsymbol{x} + b$
  - Bernoulli: $\hat{y} = \sigma(\boldsymbol{w}^T \boldsymbol{x} + b)$
  - Multinomial: $\hat{\boldsymbol{y}} = \text{softmax}(\boldsymbol{W}^T \boldsymbol{x} + \boldsymbol{b})$
- Consider regularization $\Omega(\theta)$
- Equivalent form of Loss function: $J = l(y, \hat{y}) + \lambda\Omega(\theta)$

Frequently Used Loss Functions

- Square loss

$$l(y,\hat{y}) = (y-\hat{y})^2$$

- Hinge loss

$$l(y,\hat{y}) = \max(0, 1-\hat{y}\,\mathrm{y})$$

- Logistic loss

$$l(y,\hat{y}) = \log(1+\exp(-\hat{y}y))$$

- Cross entropy loss

$$l(y,\hat{y}) = -ylog\hat{y} + (1-y)\log(1-\hat{y})$$

How to Train Multilayer Neural Nets?

- Learning is reduced to optimization.
  - Given a loss function $J(\theta)$ and several parameter sets
  - Find a model parameter set that minimize $J(\theta)$

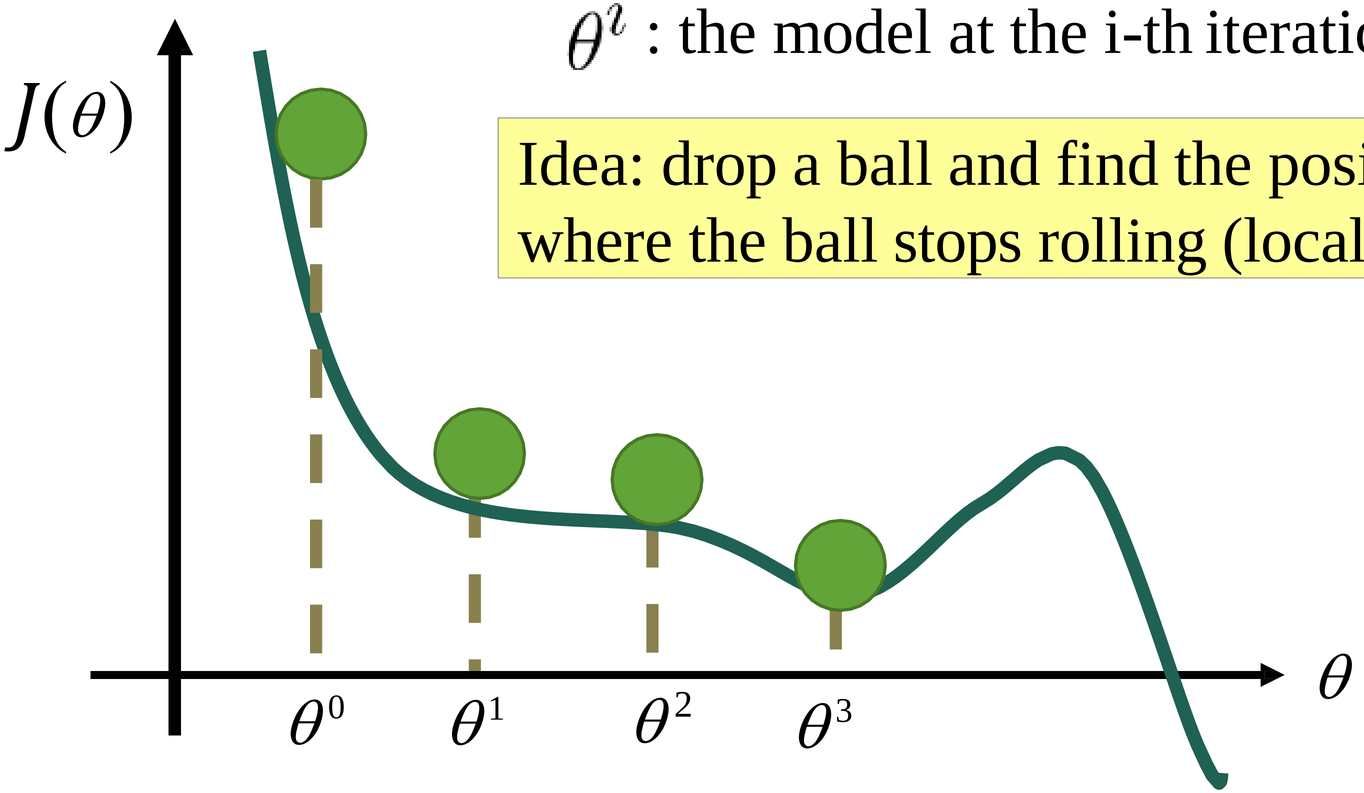
Overview

- Model Architectures
  - Artificial neurons
  - Activation function and saturation
  - Feedforward neural nets
- How to train a neural net
  - Loss Function Design
  - Optimization
    - Gradient Descent and Stochastic Gradient Descent
    - Backward propagation

Gradient Descent for Optimization

Assume that θ has only one variable

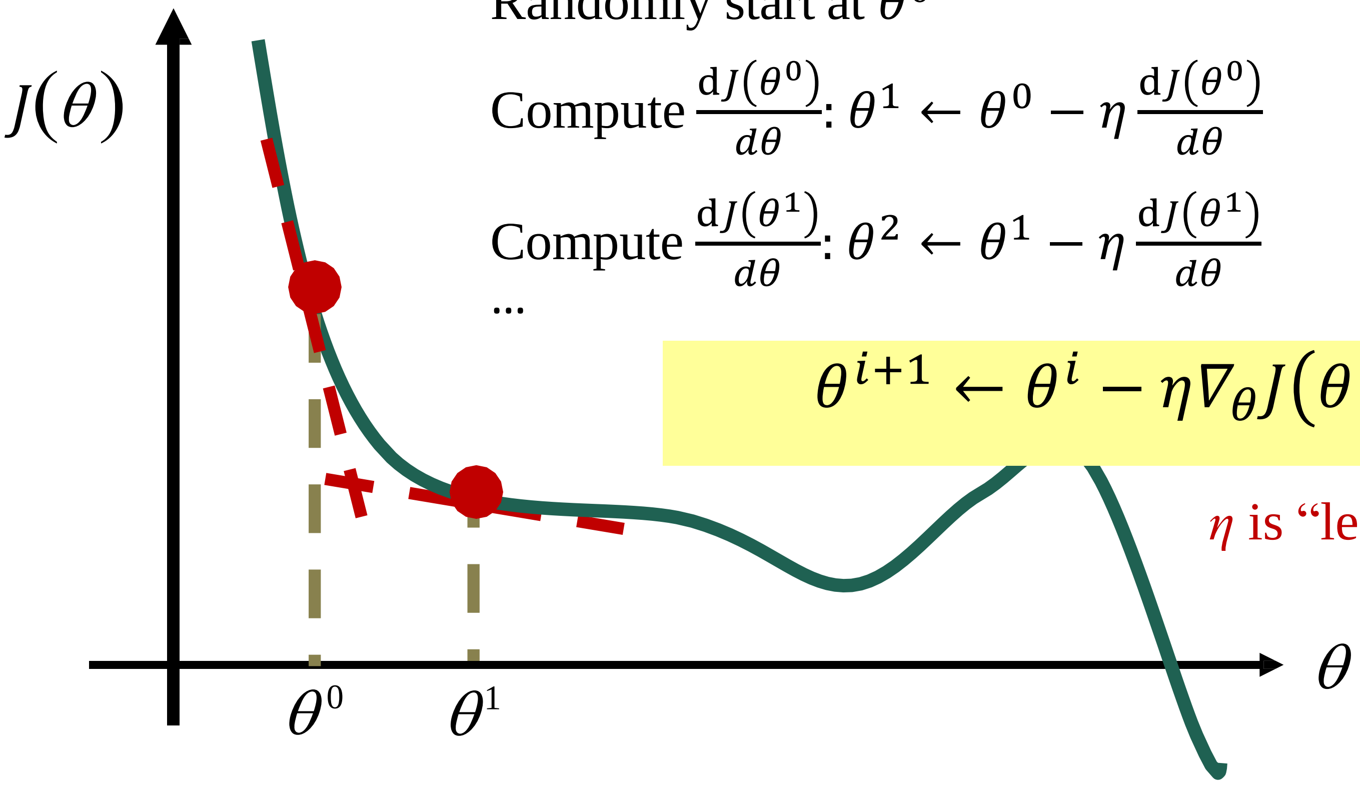
θ^i : the model at the i-th iteration



Idea: drop a ball and find the position where the ball stops rolling (local minima)

Gradient Descent for Optimization

Assume that θ has only one variable



Randomly start at θ^0

Compute $\frac{\mathrm{d}J(\theta^0)}{d\theta}$: $\theta^1 \leftarrow \theta^0 - \eta \frac{\mathrm{d}J(\theta^0)}{d\theta}$

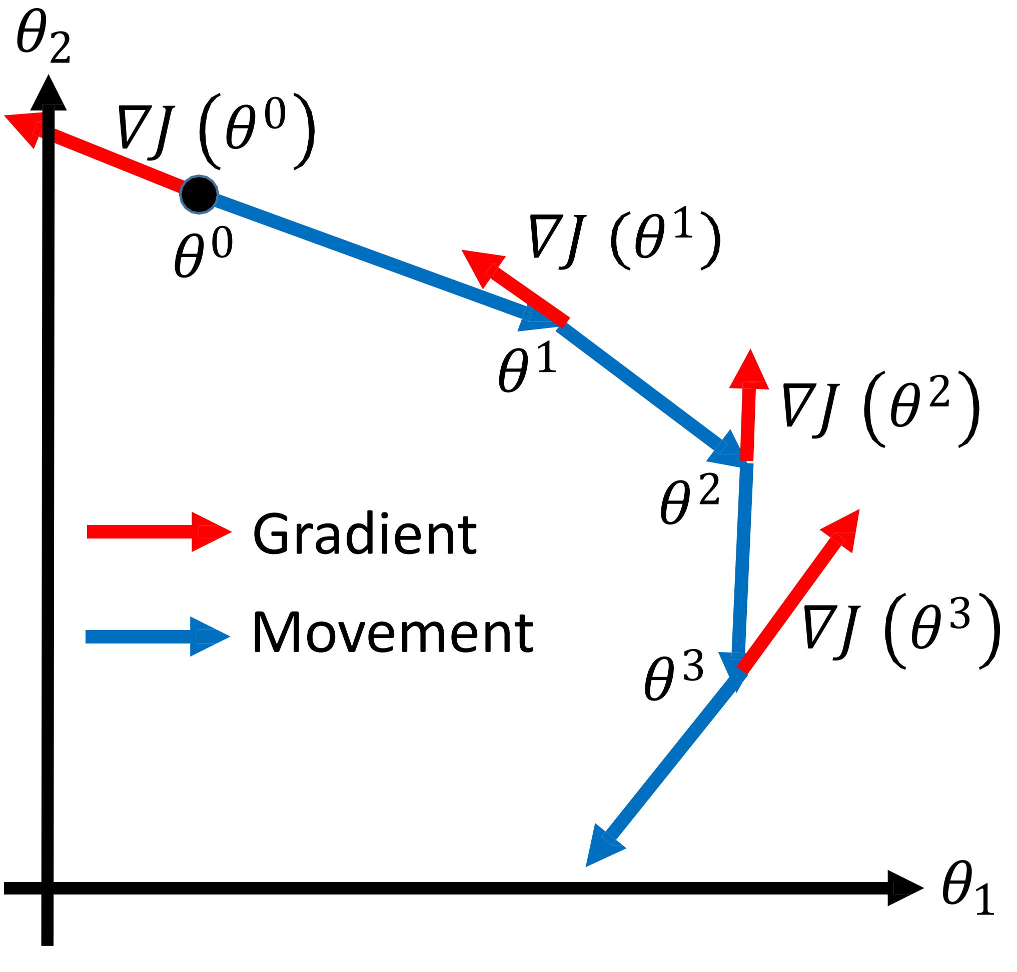
Compute $\frac{\mathrm{d}J(\theta^1)}{d\theta}$: $\theta^2 \leftarrow \theta^1 - \eta \frac{\mathrm{d}J(\theta^1)}{d\theta}$

…

$$\theta^{i+1} \leftarrow \theta^i - \eta \nabla_\theta J\left(\theta^i\right)$$

η is “learning rate”

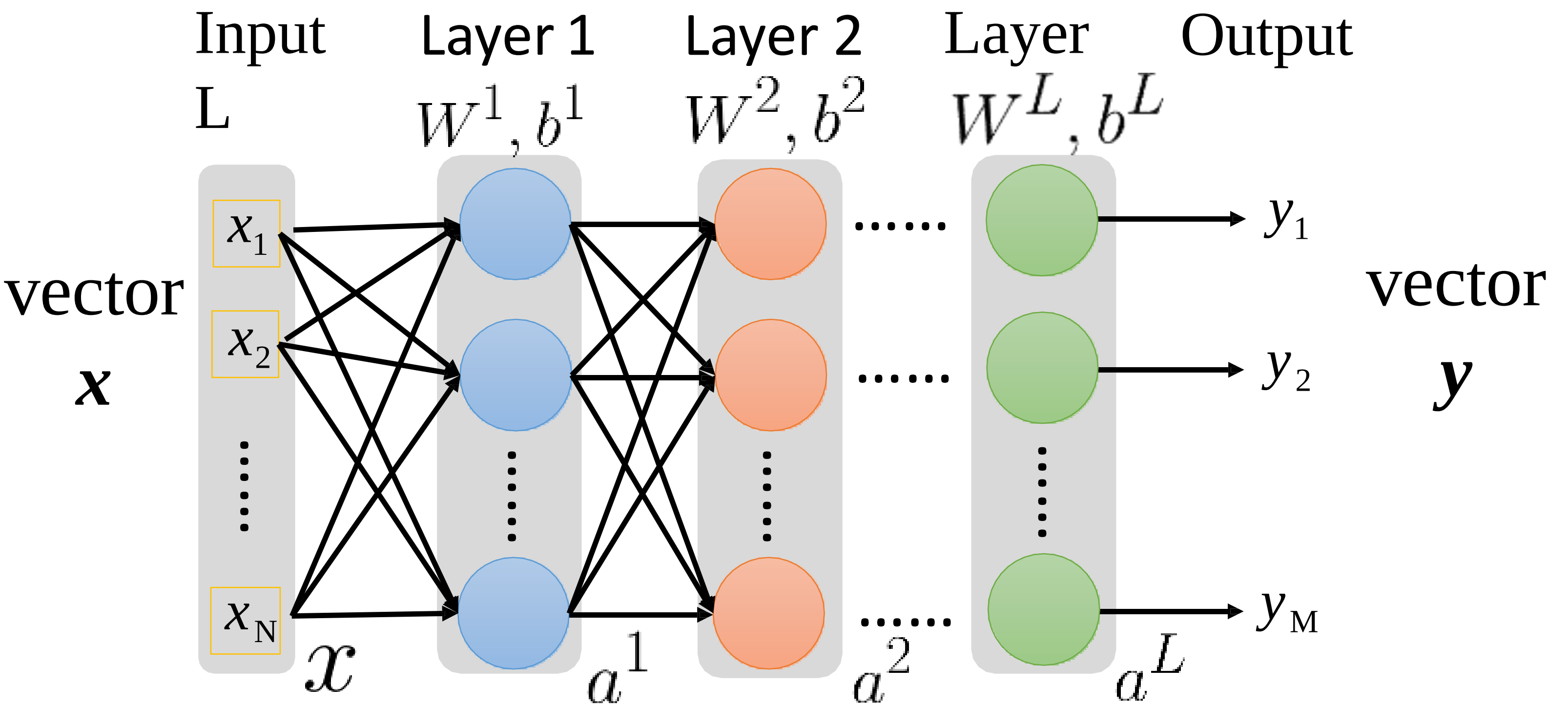
Gradient Descent for Optimization



θ_2
$\nabla J\left(\theta^0\right)$
θ^0
$\nabla J\left(\theta^1\right)$
θ^1
$\nabla J\left(\theta^2\right)$
θ^2
Gradient
Movement
$\nabla J\left(\theta^3\right)$
θ^3
θ_1

```
Algorithm
Initialization: start at $\theta^0$
while($\theta^{(i+1)} \neq \theta^i$)
{
    compute gradient at $\theta^i$
    update parameters
        $\theta^{i+1} \leftarrow \theta^i - \eta \nabla_\theta J(\theta^i)$
}
```

Revisit Neural Network Formulation



$$y = f(x) = \sigma(W^L \cdots \sigma(W^2 \sigma(W^1 x + b^1) + b^2) ... + b^L)$$

Gradient Descent for Neural Network

$$y = f(x) = \sigma(W^L \cdots \sigma(W^2\sigma(W^1x + b^1) + b^2)... + b^L)$$

$$\theta = \left\{W^1, b^1, W^2, b^2, \cdots W^L, b^L\right\}$$

$$W^l = \begin{bmatrix} w^l_{11} & w^l_{12} & \cdots \\ w^l_{21} & w^l_{22} & \\ \vdots & & \ddots \end{bmatrix} \quad b^l = \begin{bmatrix} \vdots \\ b^l_i \\ \vdots \end{bmatrix}$$

$$\nabla J(\theta) = \begin{bmatrix} \vdots \\ \dfrac{\partial J(\theta)}{\partial w^l_{ij}} \\ \vdots \\ \dfrac{\partial J(\theta)}{\partial b^l_i} \\ \vdots \end{bmatrix}$$

Algorithm

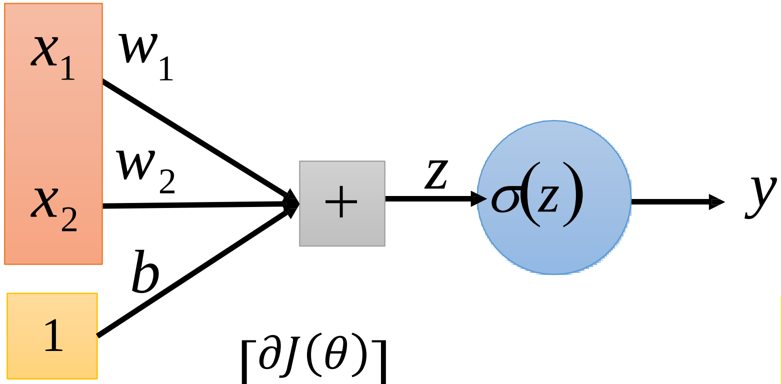
```
Initialization: start at $\theta^0$
while($\theta^{(i+1)} \neq \theta^i$)
{
    compute gradient at $\theta^i$
    update parameters
        $\theta^{i+1} \leftarrow \theta^i - \eta \nabla_\theta J(\theta^i)$
}
```

Gradient Descent for Optimization

Simple Case

$$y = f(x;\theta) = \sigma(Wx + b)$$

$$\theta = \{W, b\} = \{w_1, w_2, b\}$$

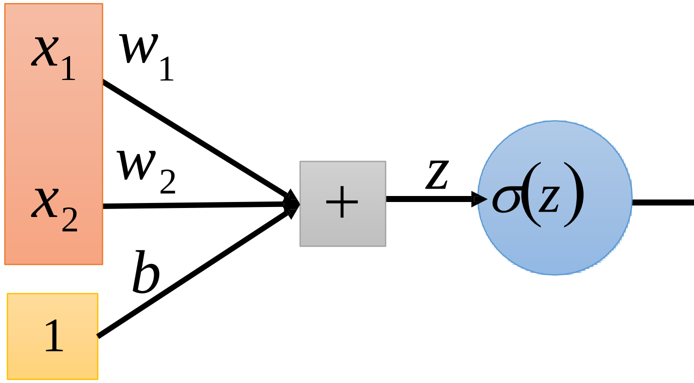


```
Algorithm
Initialization: start at $\theta^0$
while($\theta^{(i+1)} \neq \theta^i$)
{
    compute gradient at $\theta^i$
    update parameters
        $\theta^{i+1} \leftarrow \theta^i - \eta \nabla_\theta J(\theta^i)$
}
```

$$\nabla_\theta J(\theta) = \begin{bmatrix} \dfrac{\partial J(\theta)}{\partial w_1} \\ \dfrac{\partial J(\theta)}{\partial w_2} \\ \dfrac{\partial J(\theta)}{\partial b} \end{bmatrix}$$

$$\begin{bmatrix} w_1^{i+1} \\ w_2^{i+1} \\ b^{i+1} \end{bmatrix} = \begin{bmatrix} w_1^i \\ w_2^i \\ b^i \end{bmatrix} - \eta \begin{bmatrix} \dfrac{\partial J(\theta^i)}{\partial w_1} \\ \dfrac{\partial J(\theta^i)}{\partial w_2} \\ \dfrac{\partial J(\theta^i)}{\partial b} \end{bmatrix}$$

To compute the Gradients



- If square loss
  - $\hat{y} = \sigma(Wx + b) = \sigma(w_1x_1 + w_2x_2 + \mathrm{b})$
  - $J(\theta) = (\sigma(Wx + b) - y)^2$
  - $\frac{\partial J(\theta)}{\partial w_1} = 2(\sigma(Wx + b) - y)\boxed{\big(1 - \sigma(Wx + b)\big)\sigma(Wx + b)x_1}$
  - $\frac{\partial J(\theta)}{\partial w_2} = 2(\sigma(Wx + b) - y)\big(1 - \sigma(Wx + b)\big)\sigma(Wx + b)x_2$
  - $\frac{\partial J(\theta)}{\partial b} = 2(\sigma(Wx + b) - y)\big(1 - \sigma(Wx + b)\big)\sigma(Wx + b)$

```
Algorithm
Initialization: start at $\theta^0$
while($\theta^{(i+1)} \neq \theta^i$)
{compute gradient at $\theta^i$
    update parameters
    $\theta^{i+1} \leftarrow \theta^i - \eta\nabla_\theta J(\theta^i)$ }
```

Gradient Descent for Neural Network

$$y = f(x) = \sigma(W^L \cdots \sigma(W^2 \sigma(W^1 x + b^1) + b^2) \ldots + b^L)$$

$$\theta = \left\{ W^1, b^1, W^2, b^2, \cdots W^L, b^L \right\}$$

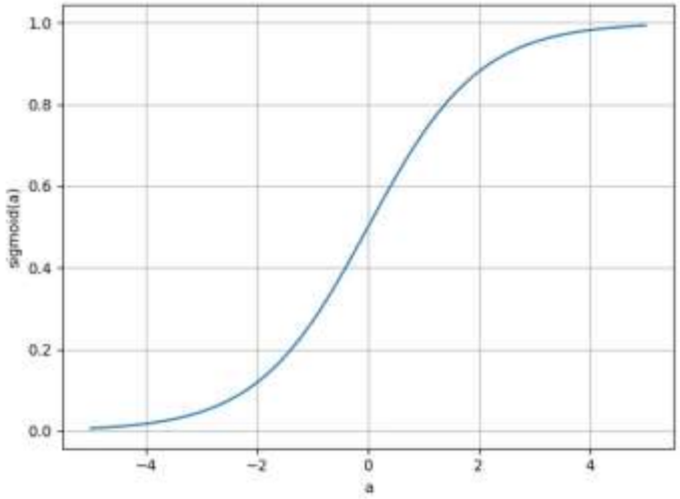
$$W^l = \begin{bmatrix} w_{11}^l & w_{12}^l & \cdots \\ w_{21}^l & w_{22}^l & \\ \vdots & & \ddots \end{bmatrix} \quad b^l = \begin{bmatrix} \vdots \\ b_i^l \\ \vdots \end{bmatrix}$$

$$\nabla J(\theta) = \begin{bmatrix} \vdots \\ \frac{\partial J(\theta)}{\partial w_{ij}^l} \\ \vdots \\ \frac{\partial J(\theta)}{\partial b_i^l} \\ \vdots \end{bmatrix}$$

```
Algorithm
Initialization: start at $\theta^0$
while($\theta^{(i+1)} \neq \theta^i$)
{
    compute gradient at $\theta^i$
    update parameters
        $\theta^{i+1} \leftarrow \theta^i - \eta \nabla_\theta J(\theta^i)$
}
```

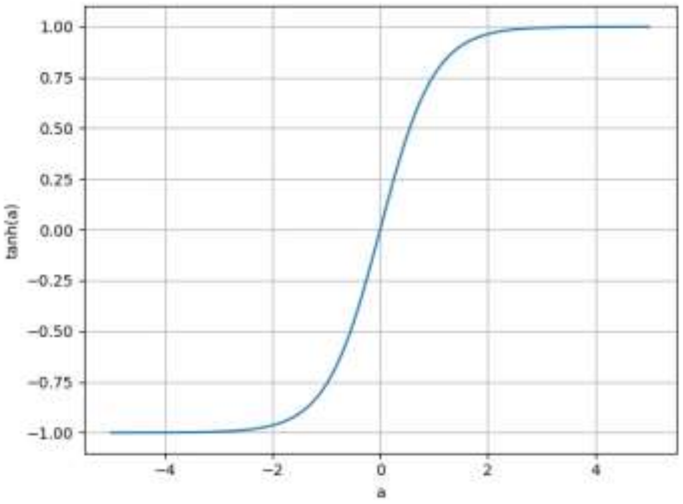
Gradient Computation: Sigmoid Unit

$f(z) = \sigma(z),\ f'(z) = \sigma(z)\big(1$



Gradient Computation:Tanh Unit

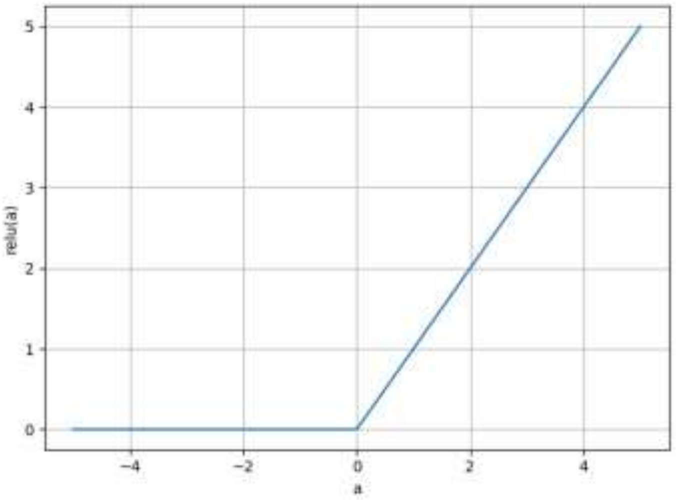
- $f(z) = \tanh(z) = 2\sigma(2z) - 1$ and $\tanh'(z) = 1 - f(z)^2$
- $\tanh(z)$ approximates linear function when z is small
- Often it is preferable to sigmoid in feedforward neural nets (zero-centered)
- Problem: still kill gradient when saturated ☹



[LeCun et al., 1991]

Gradient Computation: ReLU

- $f(z) = \max(z, 0)$
  $f'(z) = \begin{cases} 1, & z \geq 0 \\ 0, & \text{o.w.} \end{cases}$
- Models are easier to optimize if their behavior is close to linear
- Converge much faster than sigmoid and tanh in practice (6x faster)
- Not differentiable at z = 0, but it is not a problem in practice
- Not zero-centered
- Fragile during training and can “die”

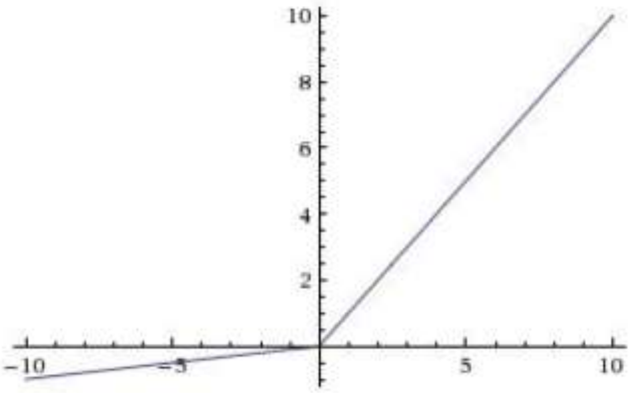


[Krizhevsky et al., 2012]

Gradient Computation: Leaky ReLU

- $f(a) = \max(z, 0.01z)$ `

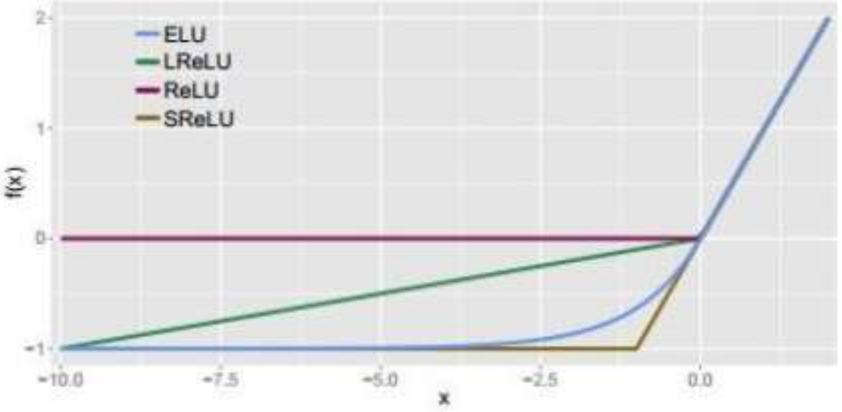
  $$f'(a) = \begin{cases} 1, & z \geq 0 \\ 0.01, & \text{o.w.} \end{cases}$$
- Will not die
- Parametric ReLU
  - $f(z) = \max(z, \mu z)$
  - $f'(z) = \begin{cases} 1, z \geq 0 \\ \mu, o.w. \end{cases}$
  - Update μ through backpropagation



[Mass et al., 2013]
[He et al., 2015]

Gradient Computation: Exponential Linear Unit (ELU)

- $f(z) = \begin{cases} z, z \geq 0 \\ \mu(e^z - 1) \end{cases}$
- $f'(z) = \begin{cases} 1, z \geq 0 \\ \mu e^z, o.w. \end{cases}$
- All benefit of ReLU
- Almost zero-centered
- Compute exp() ☹



[Clevert et al., 2015]

Gradient Computation: Maxout

- $f(z) = \max(w_1 z + b_1, w_2 z + b_2)$
- Generalization of Leaky ReLU and ReLU
- Double the number of parameters

Softmax

- Cross-entropy:

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

- In multiclass classification, y-th

$$\mathbf{y} = [0,0,0,\ldots,0,1,0,0,\ldots 0]^T \in \mathbb{R}^C,$$

Then

$$J = H(\mathbf{y},\mathbf{h}) = -\log h_y$$

where $h_{\mathrm{i}} = f_i(\mathbf{z}) = \frac{\exp(z_i)}{\sum_c^C \exp(z_c)} = \mathrm{P}(y = i|\mathbf{z}),\ 1 \leq i \leq C.$

Softmax

- $\nabla_{\mathbf{h}} J = \left[0, \dots, 0, -h_y, 0, \dots, 0\right]^T$
- $\nabla_{\mathbf{z}} J = \left(\frac{\partial \mathbf{h}}{\partial \mathbf{z}}\right)^T \nabla_{\mathbf{h}} J = -\frac{1}{h_y} \nabla h_y(\mathbf{z}) = -\left(\mathbf{e}_y - \mathbf{h}\right)$

In Practice

- For forward nn
  - Use ReLU, be careful with the learning rate
  - Tryout Leaky ReLU, ELU and Maxout
  - Tryout Tanh with low expectation
  - Never use sigmoid

Gradient Descent for Neural Network

$$y = f(x) = \sigma(W^L \cdots \sigma(W^2\sigma(W^1 x + b^1) + b^2) \ldots + b^L)$$

$$\theta = \left\{W^1, b^1, W^2, b^2, \cdots W^L, b^L\right\}$$

$$W^l = \begin{bmatrix} w_{11}^l & w_{12}^l & \cdots \\ w_{21}^l & w_{22}^l & \\ \vdots & & \ddots \end{bmatrix} \quad b^l = \begin{bmatrix} \vdots \\ b_i^l \\ \vdots \end{bmatrix}$$

$$\nabla J(\theta) = \left[\cdots, \frac{\partial J(\theta)}{\partial w_{ij}^l}, \cdots, \frac{\partial J(\theta)}{\partial b_i^l}, \cdots\right]^T$$

```
Algorithm
Initialization: start at $\theta^0$
while($\theta^{(i+1)} \neq \theta^i$)
{
    compute gradient at $\theta^i$
    update parameters
        $\theta^{i+1} \leftarrow \theta^i - \eta \nabla_\theta J(\theta^i)$
}
```

To compute the gradients of millions of parameters efficiently, we use **backpropagation**.

Gradient Descent Issue

- After see all training samples, the model can be updated slowly.
- It is too expensive to compute the full gradient

Thus, we have stochastic gradient descent (SGD)

Stochastic Gradient Descent

For $t = 1,2,3, \ldots$ (epoch means one pass over the full training set)

sample $i \in \{1,2, \ldots, n\}$

$$\mathbf{s} = \nabla l\left(f\left(\mathrm{x}^{(i)}, \boldsymbol{\theta}^{(t)}\right), y^{(i)}\right) + \lambda \nabla \Omega\left(\boldsymbol{\theta}^{(t)}\right)$$

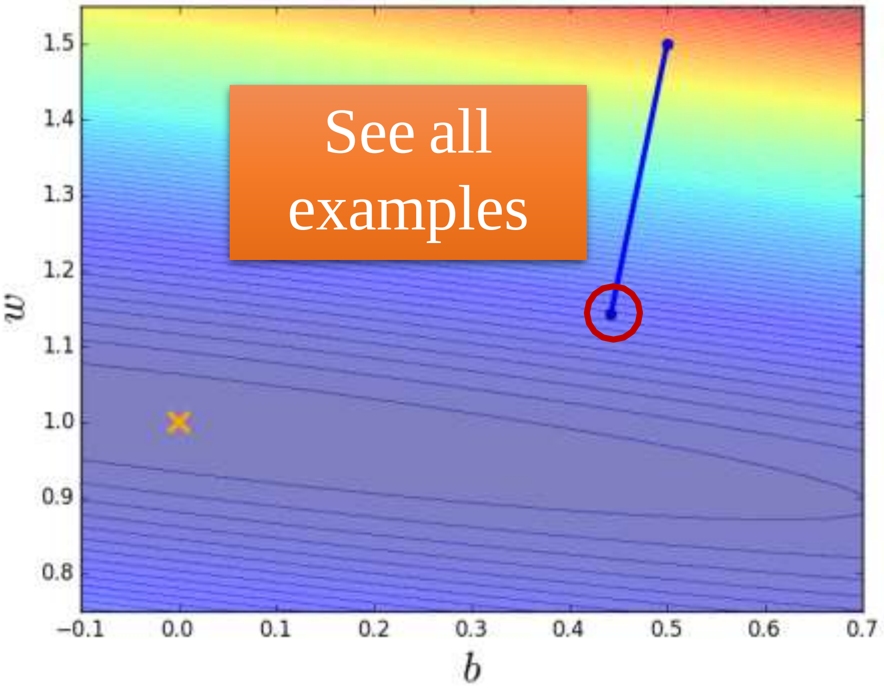
$$\boldsymbol{\theta}^{(t+1)} = \boldsymbol{\theta}^{(t)} - \eta \cdot \mathbf{s}$$

- Computing stochastic gradient is much cheaper than full gradient

Gradient Descent v.s. SGD

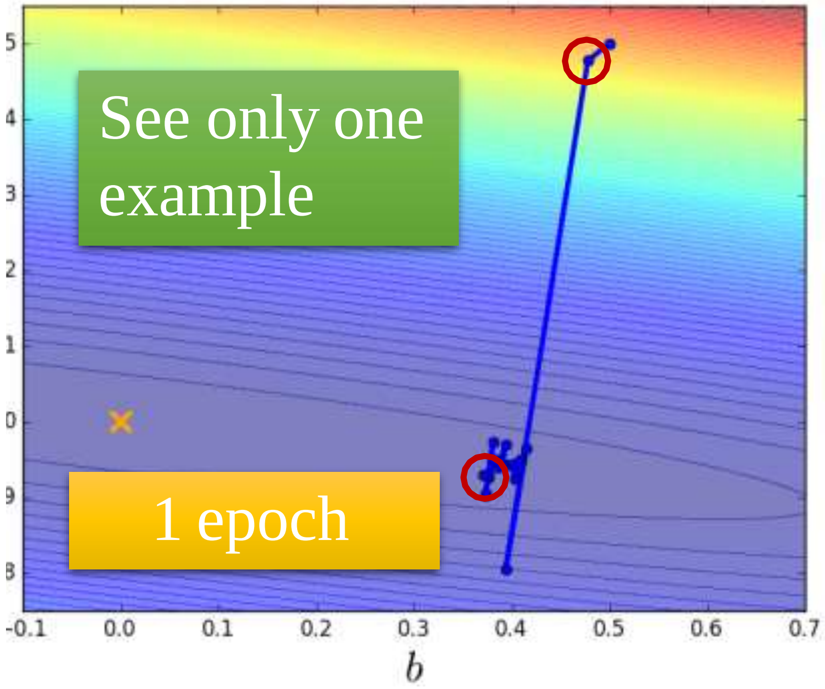
Gradient Descent

Update after seeing all examples


See all examples

Stochastic Gradient Descent

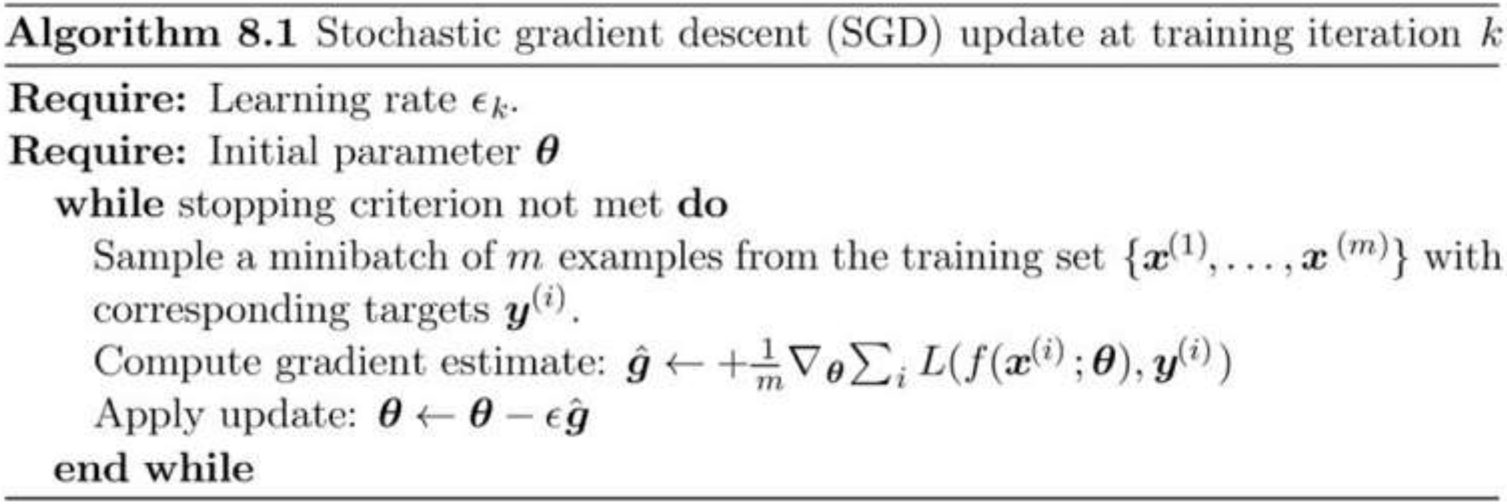
If there are 20 examples, update 20 times in one epoch.


See only one example

1 epoch

SGD approaches to the target point faster than gradient descent

Mini-Batch SGD

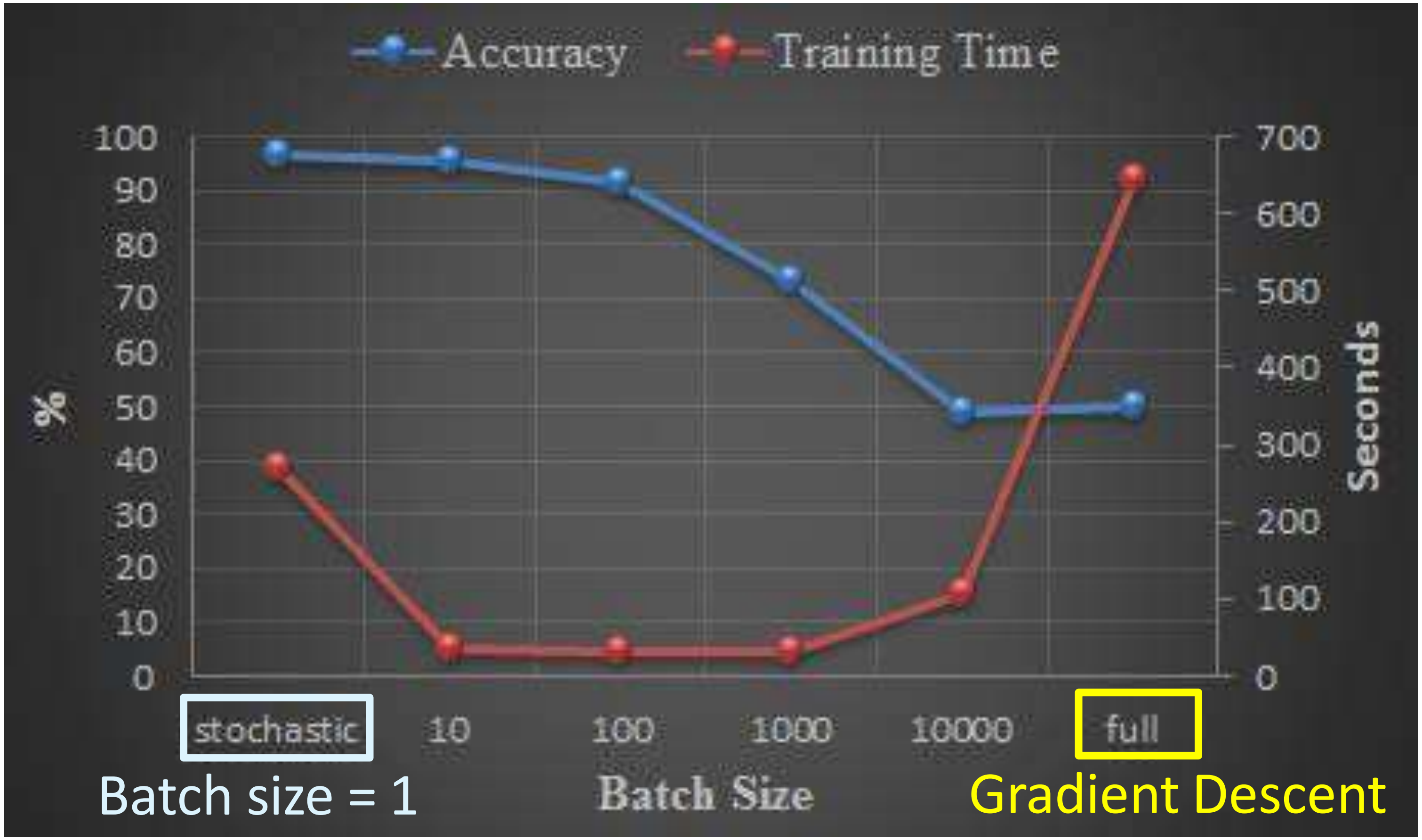


Algorithm 8.1 Stochastic gradient descent (SGD) update at training iteration k

```
Require: Learning rate $\epsilon_k$.
Require: Initial parameter $\boldsymbol{\theta}$
  while stopping criterion not met do
    Sample a minibatch of $m$ examples from the training set $\{\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(m)}\}$ with corresponding targets $\boldsymbol{y}^{(i)}$.
    Compute gradient estimate: $\hat{\boldsymbol{g}} \leftarrow +\frac{1}{m}\nabla_{\boldsymbol{\theta}}\sum_i L(f(\boldsymbol{x}^{(i)};\boldsymbol{\theta}), \boldsymbol{y}^{(i)})$
    Apply update: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \epsilon\hat{\boldsymbol{g}}$
  end while
```

Handwriting Digit Classification

Training speed: mini-batch > SGD > Gradient Descent

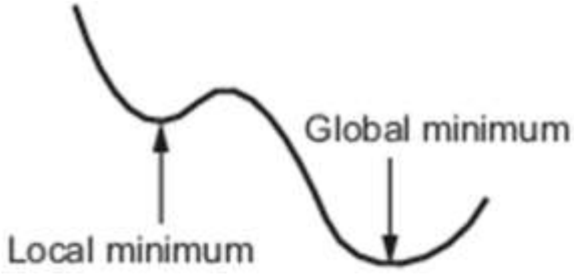


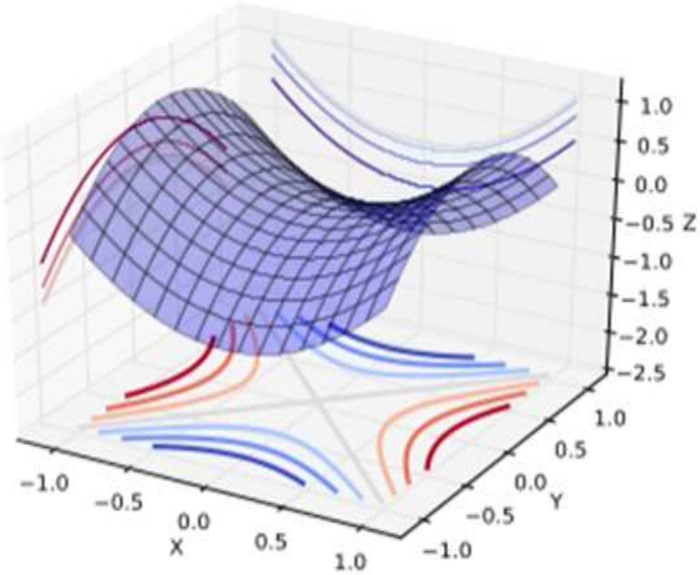
Batch size = 1

Gradient Descent

Big Issue: Local Optima

- Neural networks has no guarantee for obtaining global optimal solution
- Saddle points



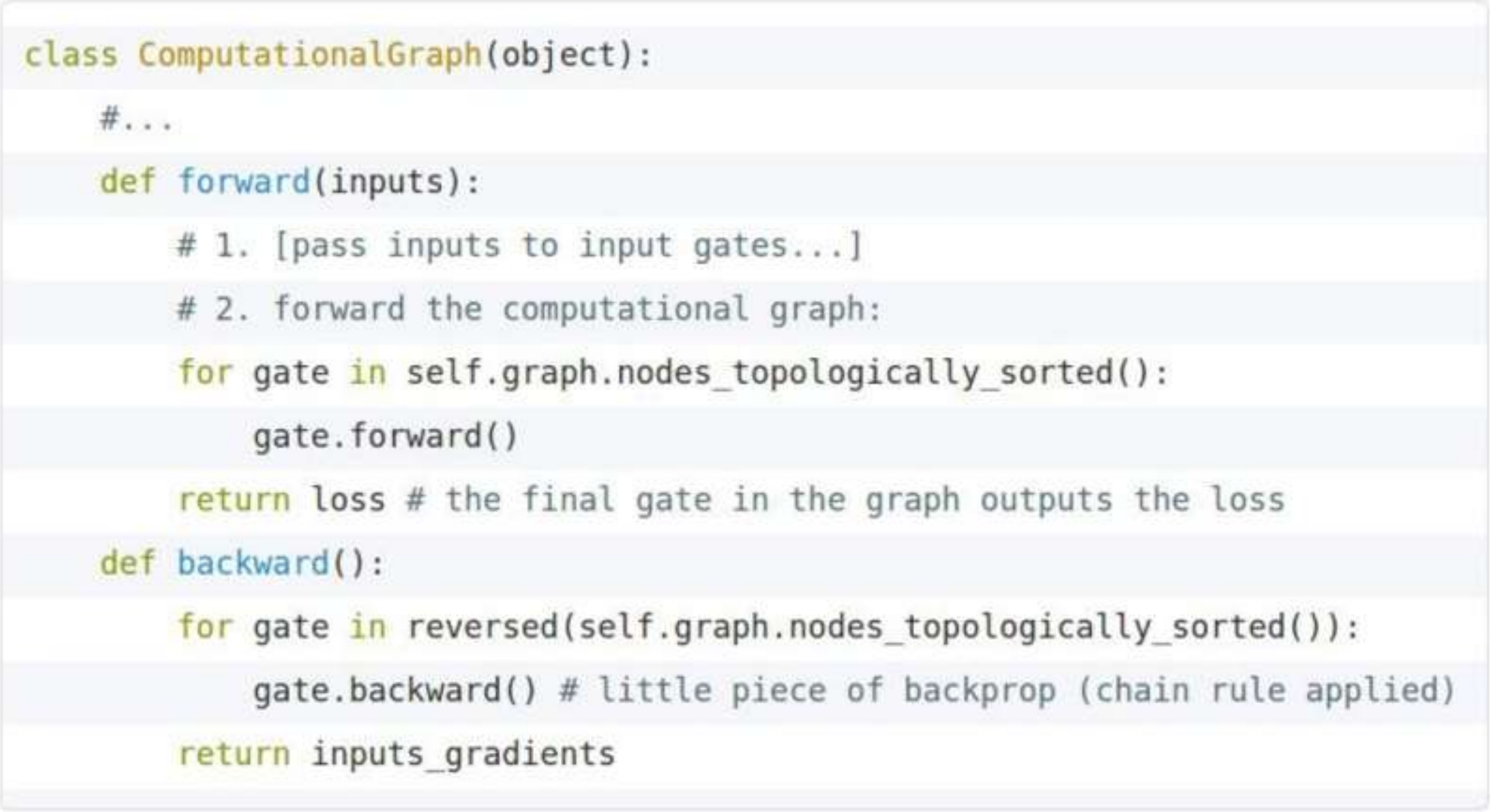


Advanced practical tips (to be presented in the last lecture)

Summary: How to Train Multilayer Neural Nets?

- Define the loss function $l(,)$ properly
- A procedure to compute loss $l(,)$ (***forward propagation***)
- A procedure to compute gradient $\nabla l(,)$ (***back propagation***)
- Regularizer and its gradient $\Omega(,)$ and $\nabla\Omega(,)$
- Perform gradient based optimization method

Forward/Backward Propagation



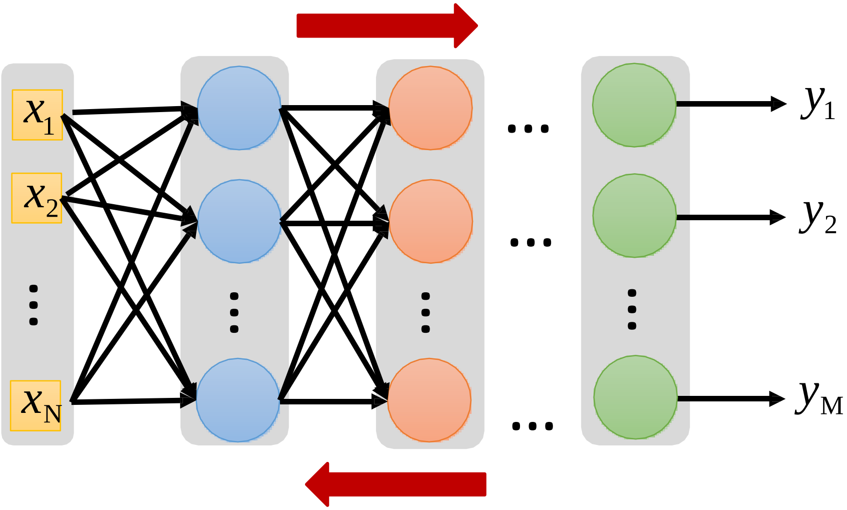
```
class ComputationalGraph(object):
    #...
    def forward(inputs):
        # 1. [pass inputs to input gates...]
        # 2. forward the computational graph:
        for gate in self.graph.nodes_topologically_sorted():
            gate.forward()
        return loss # the final gate in the graph outputs the loss
    def backward():
        for gate in reversed(self.graph.nodes_topologically_sorted()):
            gate.backward() # little piece of backprop (chain rule applied)
        return inputs_gradients
```

Overview

- Model Architectures
  - Artificial neurons
  - Activation function and saturation
  - Feedforward neural nets
- How to train a neural net
  - Loss Function Design
  - Optimization
    - Gradient Descent and Stochastic Gradient Descent
    - Back-propagation

Forward v.s. Back Propagation

- In a feedforward neural network
  - forward propagation
    - from input x to output y information flows forward through the network
    - during training, forward propagation can continue onward until it produces a scalar cost $J(\theta)$
  - back-propagation
    - allows the information from the cost to then flow backwards through the network, in order to compute the **gradient**
    - can be applied to any function



Chain Rule

$$\Delta w \to \Delta x \to \Delta y \to \Delta z$$

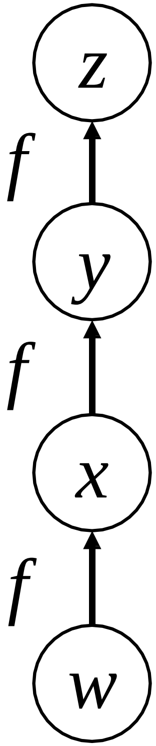
$$\frac{\partial z}{\partial w} = \frac{\partial z}{\partial y}\frac{\partial y}{\partial x}\frac{\partial x}{\partial w}$$

$$= f'(y)f'(x)f'(w)$$

forward propagation for loss (cost)

$$= \boxed{f'(f(f(w)))}\boxed{f'(f(w))}\boxed{f'(w)}$$

back-propagation for gradient



Gradient Descent for Optimization

$$y = f(x) = \sigma(W^L \cdots \sigma(W^2\sigma(W^1 x + b^1) + b^2) \cdots + b^L)$$

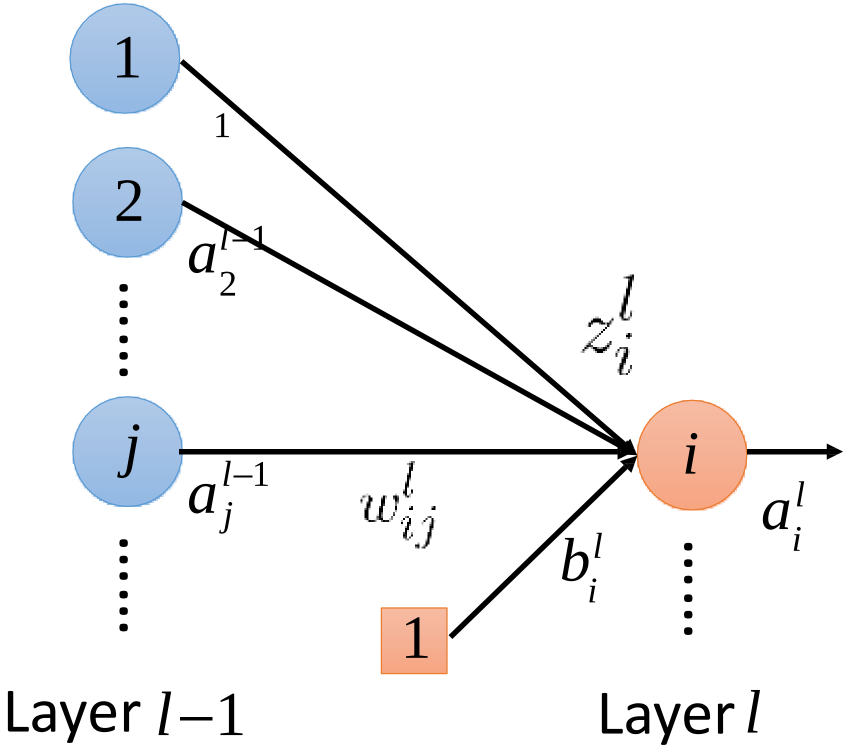
$$\theta = \left\{W^1, b^1, W^2, b^2, \cdots W^L, b^L\right\}$$

$$W^l = \begin{bmatrix} w^l_{11} & w^l_{12} & \cdots \\ w^l_{21} & w^l_{22} & \\ \vdots & & \ddots \end{bmatrix} \quad b^l = \begin{bmatrix} \vdots \\ b^l_i \\ \vdots \end{bmatrix}$$

$$\nabla J(\theta) = \begin{bmatrix} \vdots \\ \boxed{\dfrac{\partial J(\theta)}{\partial w^l_{ij}}} \\ \vdots \\ \dfrac{\partial J(\theta)}{\partial b^l_i} \\ \vdots \end{bmatrix}$$

```
Algorithm
Initialization: start at $\theta^0$
while($\theta^{(i+1)} \neq \theta^i$)
{
    compute gradient at $\theta^i$
    update parameters
        $\theta^{i+1} \leftarrow \theta^i - \eta \nabla_\theta J(\theta^i)$
}
```

$$\frac{\partial J(\theta)}{\partial w_{ij}^l}$$

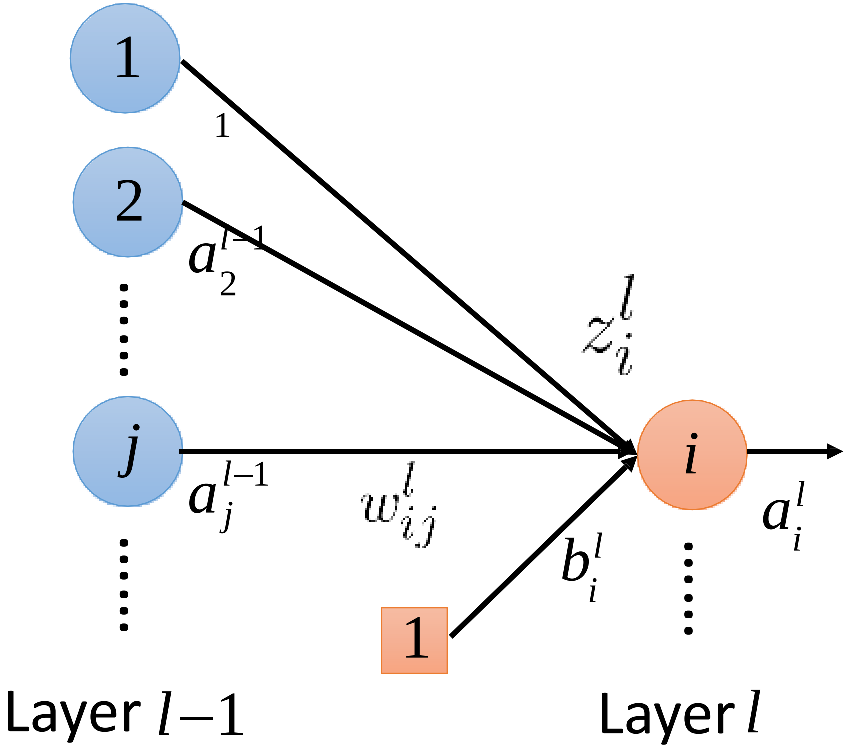


$$\frac{\partial J(\theta)}{\partial w_{ij}^l} = \frac{\partial J(\theta)}{\partial z_i^l} \frac{\partial z_i^l}{\partial w_{ij}^l}$$

$$\frac{\partial z_i^l}{\partial w_{ij}^l} \quad (l > 1)$$

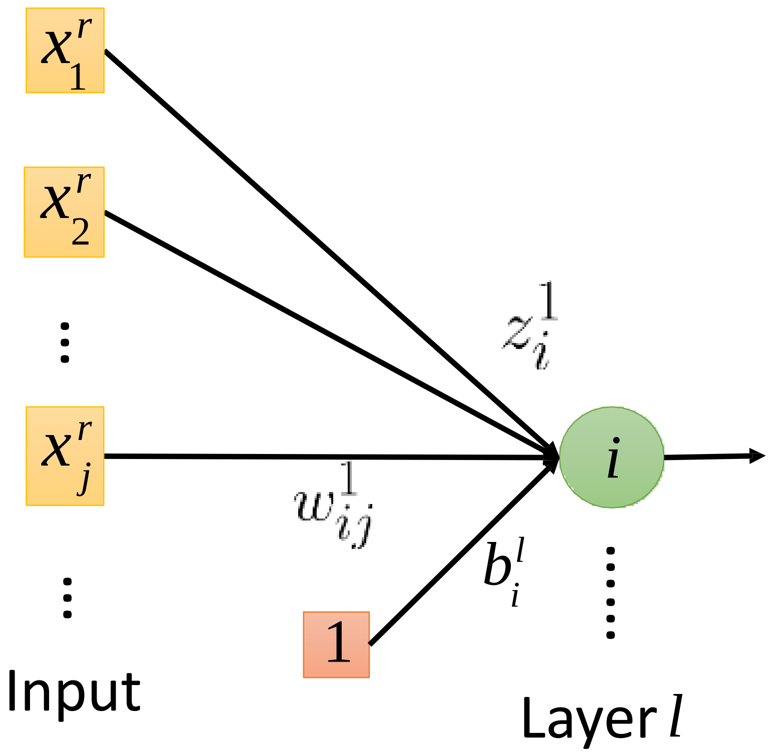
$$z^l = W^l a^{l-1} + b^l$$

$$z_i^l = \sum_j w_{ij}^l a_j^{l-1} + b_i^l$$



$$\frac{\partial z_i^l}{\partial w_{ij}^l} = a_j^{l-1}$$

$$\frac{\partial z_i^l}{\partial w_{ij}^l} \quad (l = 1)$$

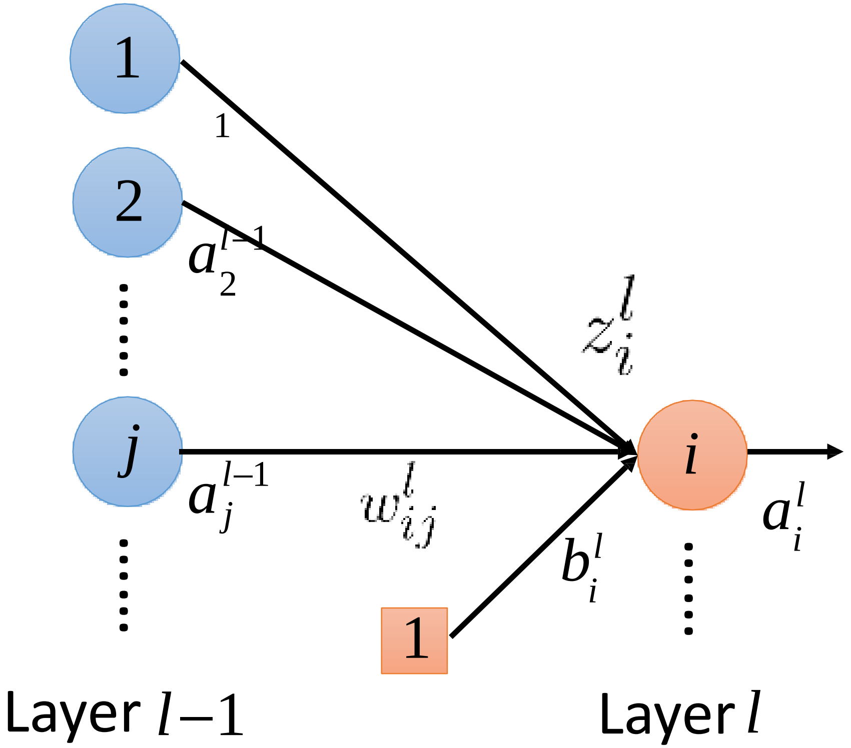


$$z^1 = W^1 x + b^1$$

$$z_i^1 = \sum_j w_{ij}^1 x_j + b_i^1$$

$$\frac{\partial z_i^1}{\partial w_{ij}^1} = x_j$$

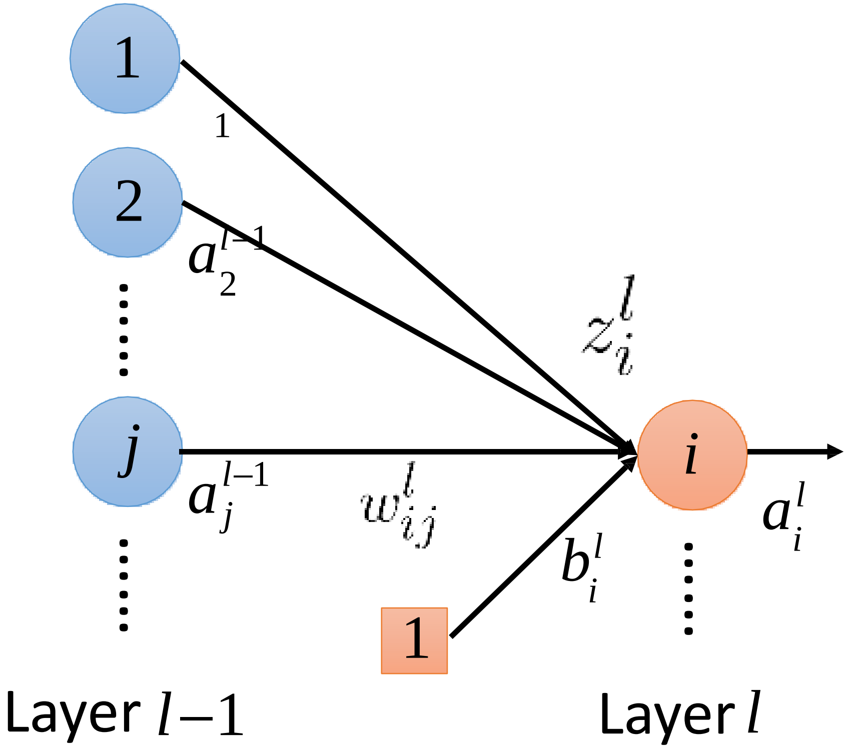
$$\frac{\partial J(\theta)}{\partial w_{ij}^{l}}$$



$$\frac{\partial J(\theta)}{\partial w_{ij}^{l}}=\frac{\partial J(\theta)}{\partial z_i^{l}}\frac{\partial z_i^{l}}{\partial w_{ij}^{l}}$$

$$\frac{\partial z_i^l}{\partial w_{ij}^l}=\begin{cases} a_j^{l-1} & ,l>1 \\ x_j & ,l=1 \end{cases}$$

$$\frac{\partial J(\theta)}{\partial w_{ij}^l}$$

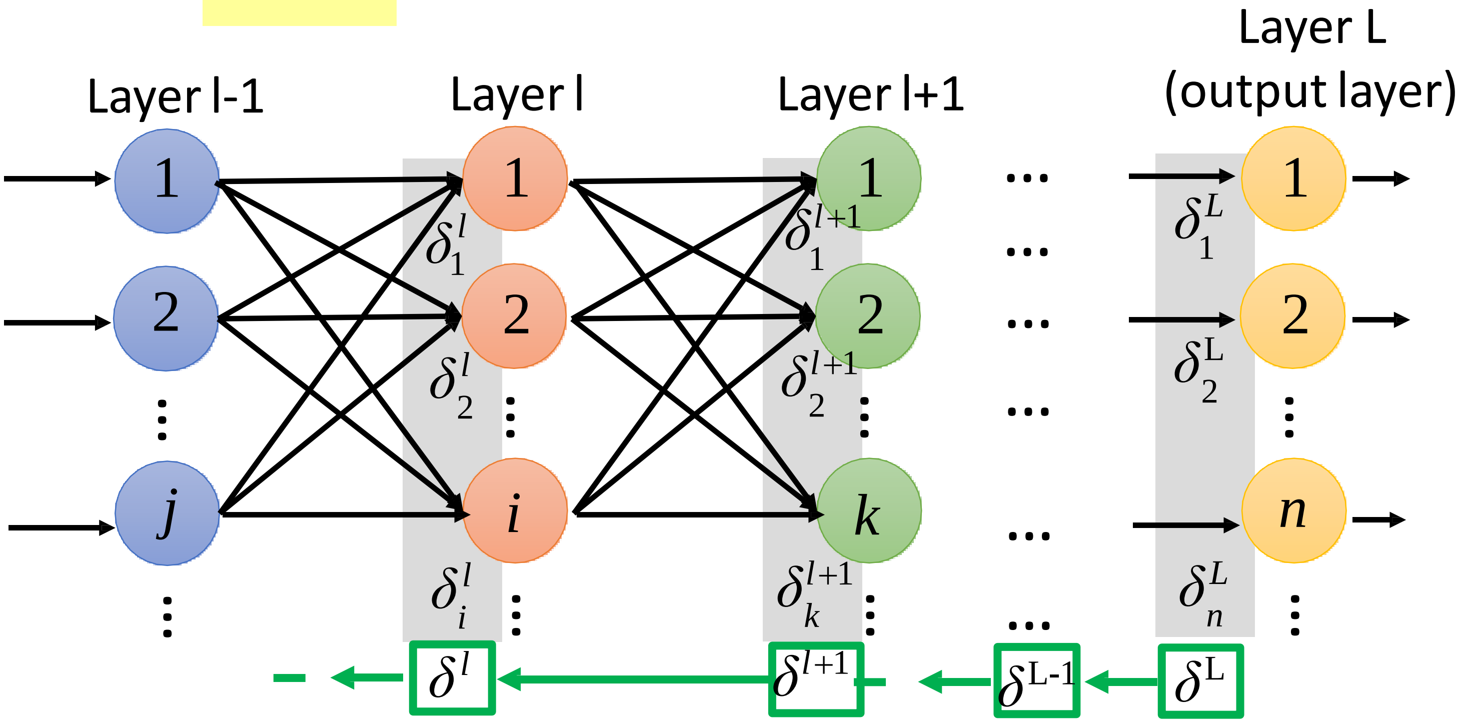


$$\frac{\partial J(\theta)}{\partial w_{ij}^l} = \frac{\partial J(\theta)}{\partial z_i^l} \frac{\partial z_i^l}{\partial w_{ij}^l}$$

$$\frac{\partial J(\theta)}{\partial z_i^l}$$

$$\frac{\partial J(\theta)}{\partial w_{ij}^l} = \frac{\partial J(\theta)}{\partial z_i^l} \frac{\partial z_i^l}{\partial w_{ij}^l}$$

δ_i^l : the propagated gradient corresponding to the *l*-th layer



Idea: computing δ^l layer by layer (from δ^L to δ^1) is more efficient

$$\frac{\partial J(\theta)}{\partial z_i^l} = \delta_i^l$$

Idea: from L to 1

(1) Initialization: compute δ^L

(2) Compute δ^l based on δ^{l+1}

$$\frac{\partial J(\theta)}{\partial z_i^l} = \delta_i^l$$

Idea: from L to 1

(1) Initialization: compute $\boldsymbol{\delta^L}$

(2) Compute δ^l based on δ^{l+1}

$$\delta_i^L = \frac{\partial J}{\partial z_i^L} = \boxed{\frac{\partial J}{\partial y_i}} \frac{\partial y_i}{\partial z_i^L}$$

$\frac{\partial J}{\partial y_i}$ depends on the loss function

$$\delta^L = \nabla J(y) \odot \nabla \mathrm{a}\left(\mathrm{z}^{\mathrm{L}}\right)$$

$$\frac{\partial J(\theta)}{\partial z_i^l} = \delta_i^l$$

(1) Initialization: compute δ^L

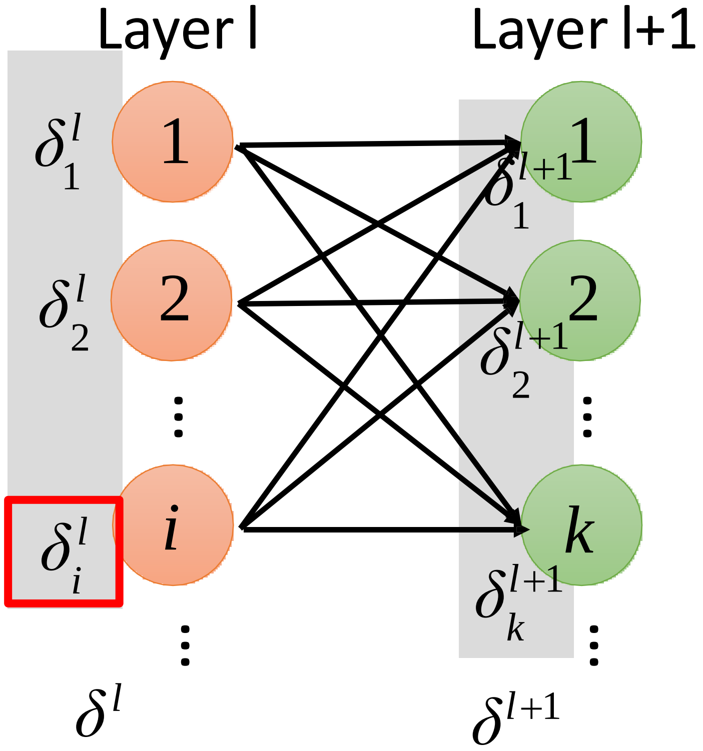
(2) Compute δ^l based on δ^{l+1}

$$\begin{tikzcd}
 & & \Delta z_1^{l+1} \arrow[dddr, dotted] & \\
 & & \Delta z_2^{l+1} \arrow[ddr, dotted] & \\
 & & \vdots & \\
\Delta z_i^l \arrow[r] & \Delta a^l_i \arrow[uuur] \arrow[uur] \arrow[r] & \Delta z_k^{l+1} \arrow[r, dotted] & \Delta J \\
 & & \vdots &
\end{tikzcd}$$

$$\delta_i^l = \frac{\partial J}{\partial z_i^l} = \sum_k \left(\frac{\partial J}{\partial z_k^{l+1}} \frac{\partial z_k^{l+1}}{\partial a_i^l} \frac{\partial a_i^l}{\partial z_i^l}\right)$$

$$= \frac{\partial a_i^l}{\partial z_i^l} \sum_k \left(\frac{\partial J}{\partial z_k^{l+1}} \frac{\partial z_k^{l+1}}{\partial a_i^l}\right)$$

$$\delta_i^{l+1}$$



$$\frac{\partial J(\theta)}{\partial z_i^l} = \delta_i^l$$

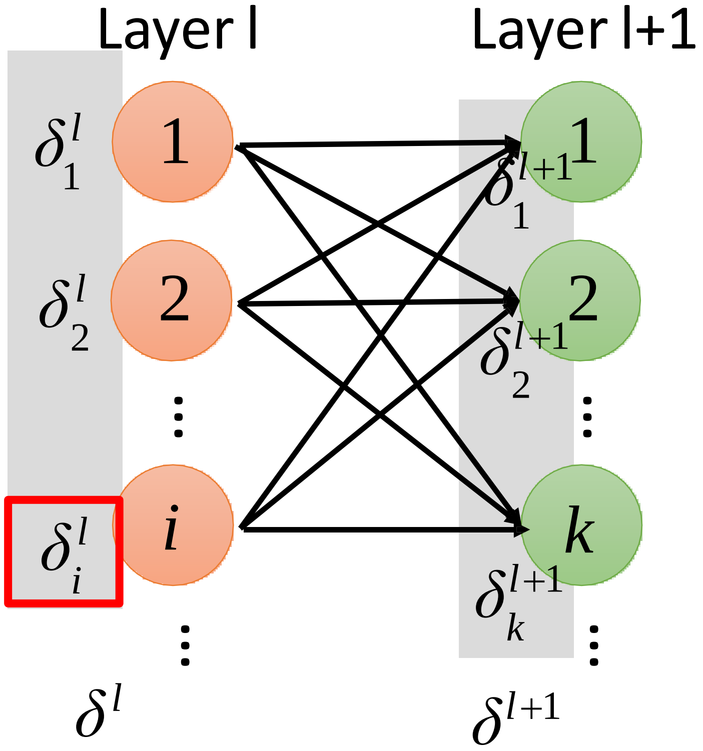
(1) Initialization: compute δ^L

(2) Compute δ^l based on δ^{l+1}

$$\delta_i^l = \frac{\partial a_i^l}{\partial z_i^l} \sum_k \left(\frac{\partial J}{\partial z_k^{l+1}} \frac{\partial z_k^{l+1}}{\partial a_i^l}\right)$$

$$= \frac{\partial a_i^l}{\partial z_i^l} \sum_k \delta_k^{l+1} w_{ki}^{l+1}$$

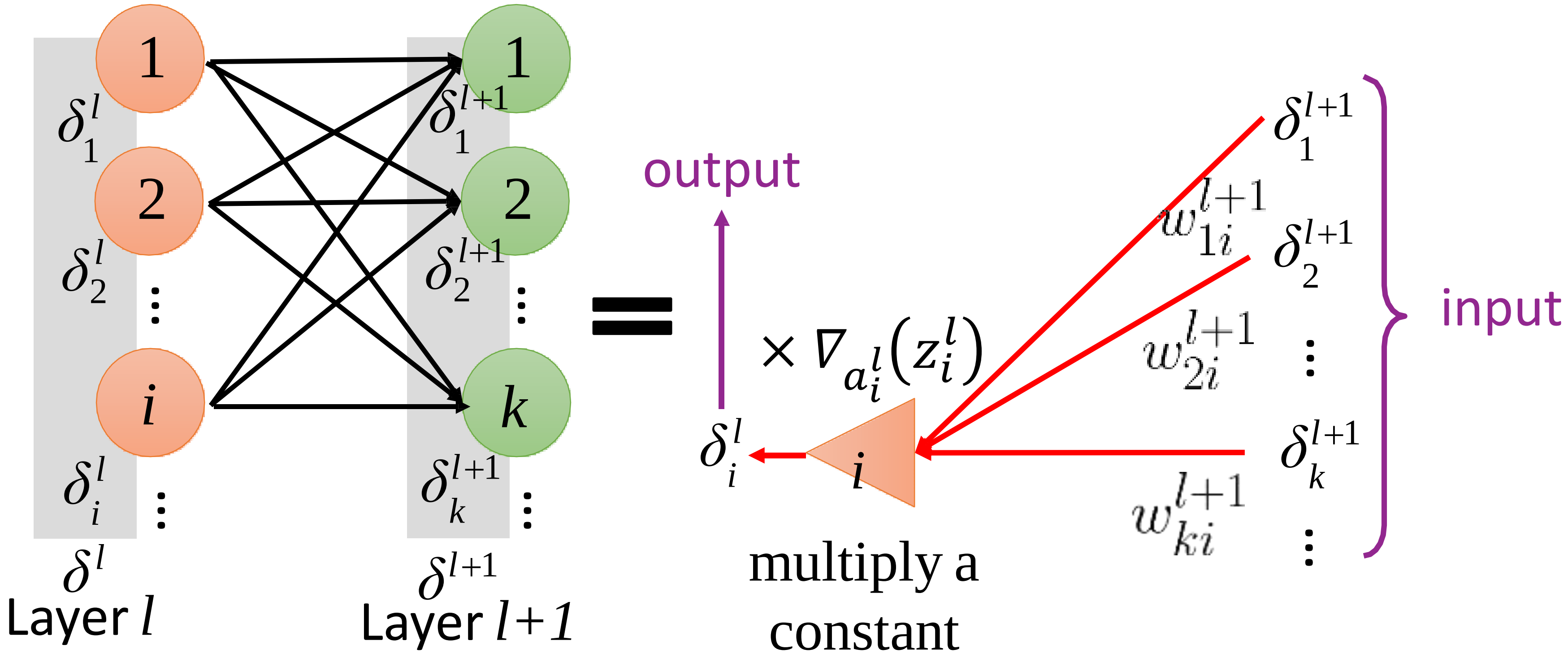
$$= \nabla a_i^l(z_i^l) \sum_k \delta_k^{l+1} w_{ki}^{l+1}$$



$$\frac{\partial J(\theta)}{\partial z_i^l} = \delta_i^l$$

Rethink the propagation

$$\delta_i^l = \nabla a_i^l(z_i^l) \sum_k \delta_k^{l+1} w_{ki}^{l+1}$$



multiply a constant

$$\delta^l = \nabla a(z^l) \odot (W^{l+1})^T \delta^{l+1}$$

$$\frac{\partial J(\theta)}{\partial z_i^l} = \delta_i^l$$

$$\frac{\partial J(\theta)}{\partial w_{ij}^l} = \frac{\partial J(\theta)}{\partial z_i^l} \frac{\partial z_i^l}{\partial w_{ij}^l}$$

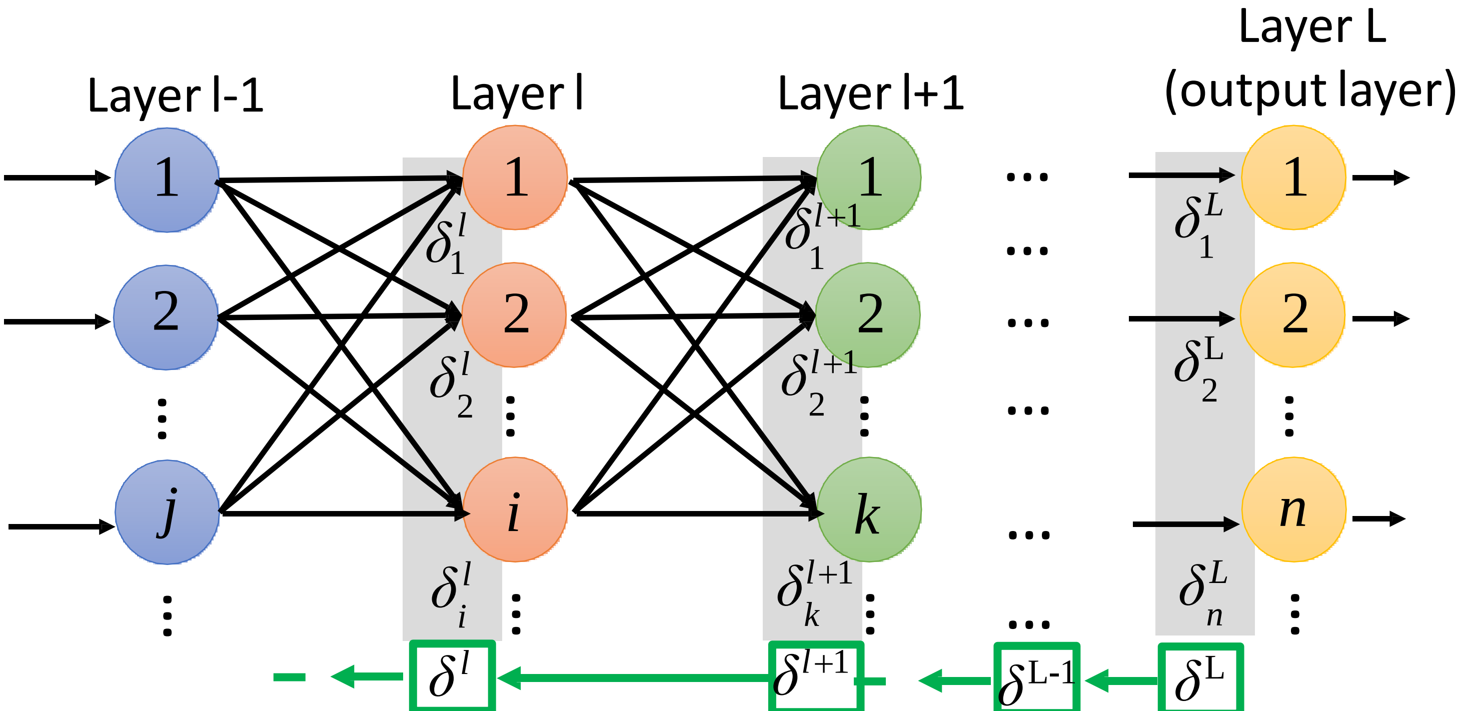
Idea: from L to 1

(1) Initialization: compute δ^L

(2) Compute δ^l based on δ^{l+1}

$$\delta^L = \nabla J(y) \odot \nabla \mathrm{a}(\mathrm{z}^\mathrm{L})$$

$$\delta^l = \nabla a(z^l) \odot (W^{l+1})^T \delta^{l+1}$$



Backpropagation

$$\frac{\partial J(\theta)}{\partial w_{ij}^{l}} = \frac{\partial J(\theta)}{\partial z_i^{l}} \frac{\partial z_i^{l}}{\partial w_{ij}^{l}}$$

$$\frac{\partial z_i^l}{\partial w_{ij}^l} = \begin{cases} a_j^{l-1} & , l > 1 \\ x_j & , l = 1 \end{cases}$$

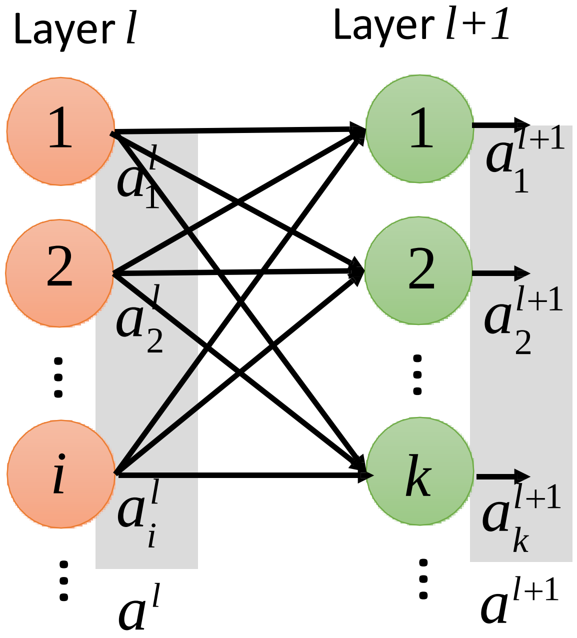
Forward Pass

$$z^1 = W^1 x + b^1 \qquad a^1 = \sigma(z^1)$$

$$\vdots$$

$$z^l = W^l a^{l-1} + b^l \quad a^l = \sigma(z^l)$$

$$\vdots$$



$$\frac{\partial J(\theta)}{\partial w_{ij}^{l}} = \frac{\partial J(\theta)}{\partial z_i^l}\frac{\partial z_i^l}{\partial w_{ij}^{l}}$$

Backpropagation

$$\frac{\partial J(\theta)}{\partial z_i^l} = \delta_i^l$$

Backward Pass

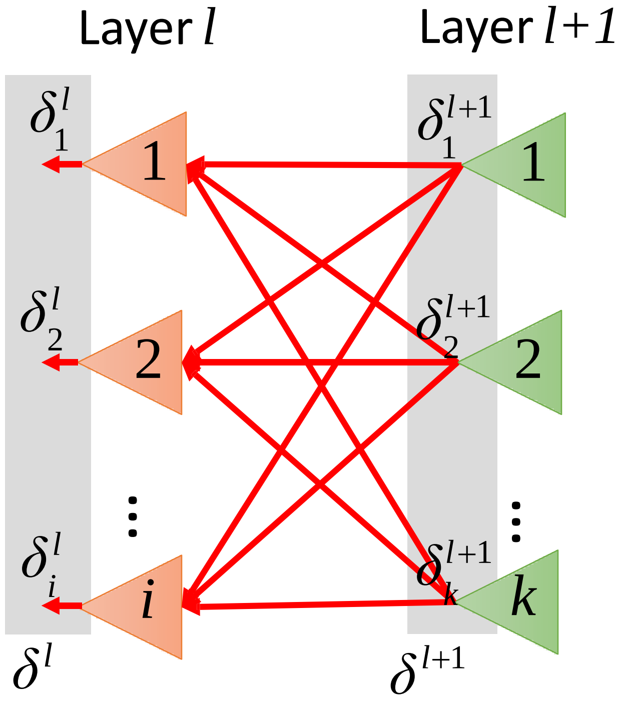
$$\delta^L = \nabla J(y) \odot \nabla \mathrm{a}(\mathrm{z}^{\mathrm{L}})$$

$$\delta^{L-1} = \nabla a(z^l) \odot \left(W^{l+1}\right)^T \delta^{l+1}$$

⋮

$$\delta^l = \nabla a(z^l) \odot \left(W^{l+1}\right)^T \delta^{l+1}$$

⋮



Reading Materials

- Automatic Differentiation in Machine Learning: a Survey (2015)

Summary: How to Train Multilayer Neural Nets?

- Define the loss function $l(,)$ properly
- A procedure to compute loss $l(,)$ (***forward propagation***)
- A procedure to compute gradient $\nabla l(,)$ (***back propagation***)
- Regularizer and its gradient $\Omega(,)$ and $\nabla\Omega(,)$
- Perform gradient based optimization method

Summary

- Model Architectures
  - Artificial neurons
  - Activation function and saturation
  - Feedforward neural nets
- How to train a neural net
  - Loss Function Design
  - Optimization
    - Gradient Descent and Stochastic Gradient Descent
    - Back-propagation