Neural Networks

Bùi Tiến Lên

2023

KHOA CÔNG NGHỆ THÔNG TIN
TRƯỜNG ĐẠI HỌC KHOA HỌC TỰ NHIÊN

Contents

Notation

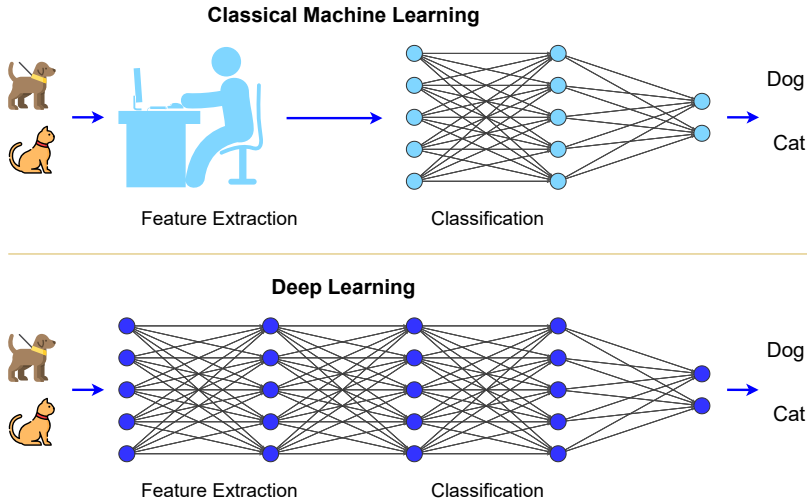
symbol	meaning
$a, b, c, N \ldots$	scalar number
$\boldsymbol{w}, \boldsymbol{v}, \boldsymbol{x}, \boldsymbol{y} \ldots$	column vector
$\boldsymbol{X}, \boldsymbol{Y} \ldots$	matrix
$\mathbb{R}$	set of real numbers
$\mathbb{Z}$	set of integer numbers
$\mathbb{N}$	set of natural numbers
$\mathbb{R}^D$	set of vectors
$\mathcal{X}, \mathcal{Y}, \ldots$	set
$\mathcal{A}$	algorithm

operator	meaning
$\boldsymbol{w}^\mathsf{T}$	transpose
$\boldsymbol{X}\boldsymbol{Y}$	matrix multiplication
$\boldsymbol{X}^{-1}$	inverse
$\boldsymbol{X} \odot \boldsymbol{Y}$	an element-wise matrix-vector multiplication

Previous Lessons

- Ensemble
- Uncertainty
- Feature representation
  - Feature transformation
  - Hidden variables

Deep Learning vs. Classical Machine Learning

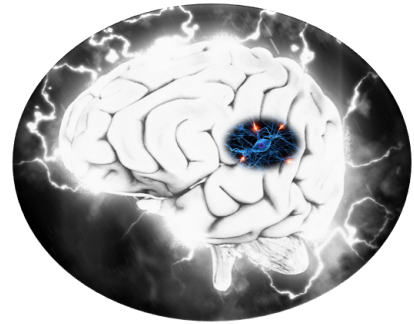


Artificial Neural Network

What is a neural network?

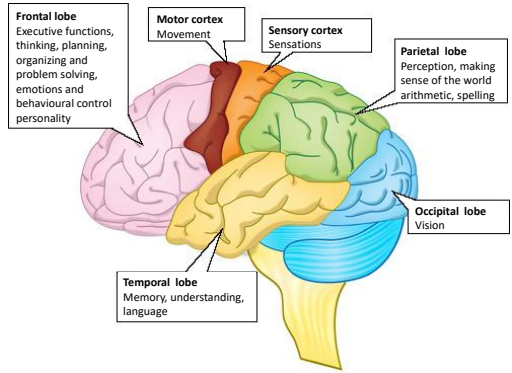
Neural network is a reasoning model based on the human brain, including billions of neurons and trillion connections between them.
The biological brain is considered as a highly complex, nonlinear and parallel information-processing system.

- Number of neurons 10^{10}
- Connections per neuron 10^{4-5}
- Neuron switching time $.001$ second
- Scene recognition time $.1$ second



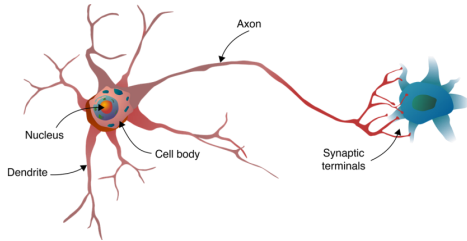
Artificial neural networks

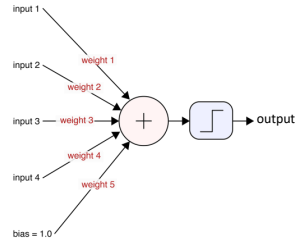
- **Artificial neural networks** (ANNs) resembles the human brain in **terms of learning mechanisms**.
- An ANN consists of a number of very simple and highly interconnected processors (or **neurons**), arranging in **a hierarchy of layers**.



Neuron

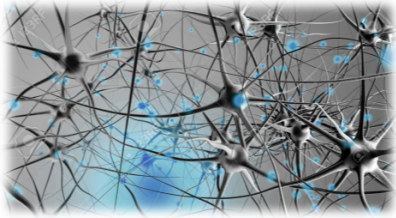
- Each **neuron** is an elementary information-processing unit.
- The neurons are connected by **link**s passing signals from one neuron to another.
- Each neuron receives a number of **input signals** through its connections and produces at most a single **output signal**.
- Each link associates with **a numerical weight** expressing the strength of the neuron input $\rightarrow$ basic means of long-term memory in ANNs
- ANNs "learn" through repeated adjustments of these weights

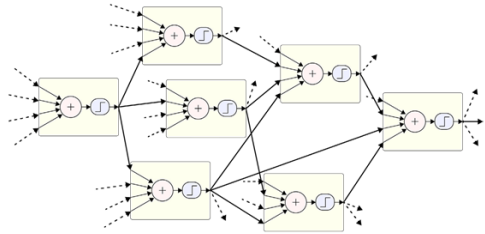




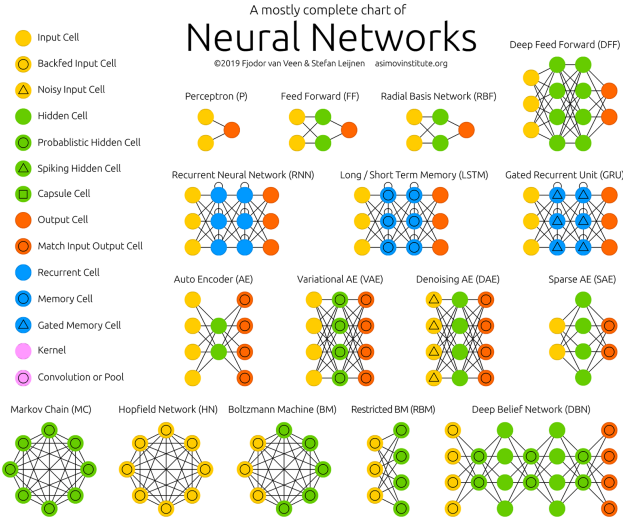
The network architecture

- The network architecture includes
  - How many neurons are to be used?
  - How the neurons are to be connected to form a network?

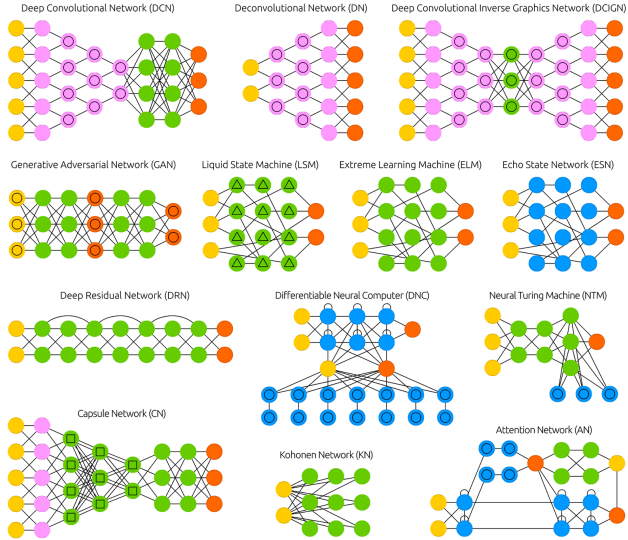




Some architectures



Some architectures (cont.)



Perceptron: The simplest ANN (Revisited)

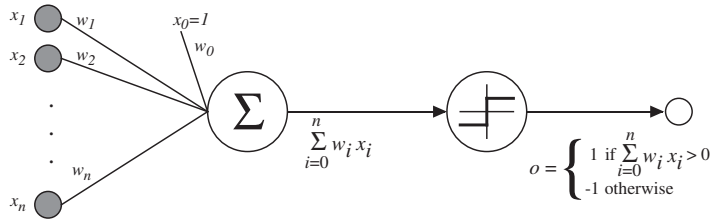
Perceptron Model

- A **perceptron** proposed by Minsky-Papert, more general than McCulloch-Pitts neuron, consists of n inputs $x_1, ..., x_n$ and 1 output o
- First, it computes a **weighted sum of its inputs**

$$s = \sum_{i=0}^{n} w_i x_i = w_0 + w_1 x_1 + \cdots + w_n x_n \tag{1}$$

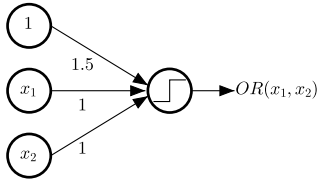
- Then it applies an **activation function** $g(s)$ (*sign*, *sigmoid* etc.)

$$o(x_1, \ldots, x_n) = g(s) = \begin{cases} 1 & \text{if } s \geq 0 \\ -1 & \text{otherwise.} \end{cases} \tag{2}$$

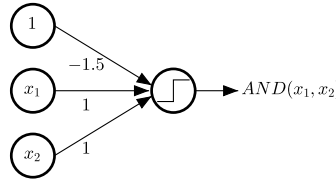


Representation

- Perceptron can represent some useful functions



x_1	x_2	$OR(x_1, x_2)$
-1	-1	-1
-1	1	1
1	-1	1
1	1	1



x_1	x_2	$AND(x_1, x_2)$
-1	-1	-1
-1	1	-1
1	-1	-1
1	1	1

- Perceptron cannot represent $XOR(x_1, x_2)$
  - Solution is to combine perceptrons $\rightarrow$ multilayer perceptrons

Multilayer Perceptrons (MLP)

- Learning Model
- Design Neural Network
- Neural Network Learning

Neural Network Playground

Epoch 000,000

Learning rate 0.03

Activation Tanh

Regularization None

Regularization rate 0

Problem type Classification

DATA

Which dataset do you want to use?

Ratio of training to test data: 50%

Noise: 0

Batch size: 10

REGENERATE

FEATURES

Which properties do you want to feed in?

X_1

X_2

X_1^2

X_2^2

X_1X_2

$\sin(X_1)$

$\sin(X_2)$

2 HIDDEN LAYERS

4 neurons

2 neurons

OUTPUT

Test loss 0.502

Training loss 0.536

Colors shows data, neuron and weight values.

Show train data

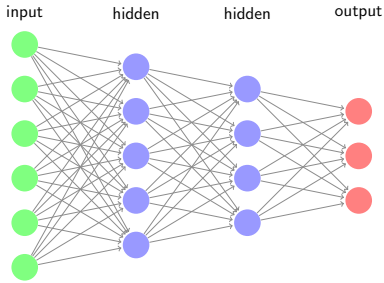
Show test data

Discretize output

Multilayer Perceptrons

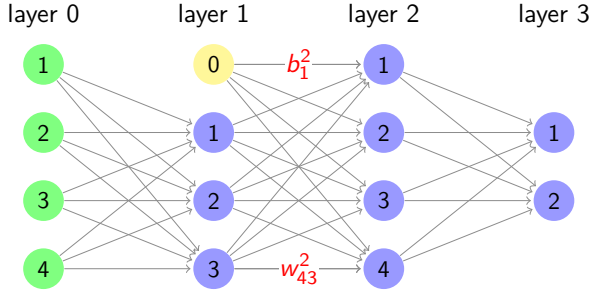
Multilayer perceptrons (neural networks) is a DAG graph that includes many layers.

- The first layer is called **input layer**
- The last layer is called **output layer**
- The other are called **hidden layers**



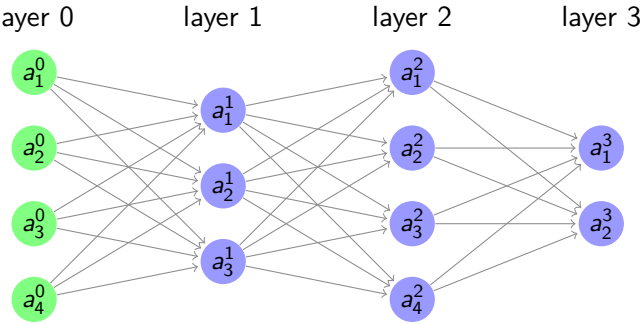
Multilayer Perceptrons (cont.)

- Each layers composed of **nodes** (neurons)
- Neurons are connected by directed **links** to propagate the **activation**. w_{jk}^{l} is the weight of the link from the k-th neuron in the $(l-1)$-th layer to the j-th neuron in the l-th layer
- Each layer has ***weight matrix*** $\boldsymbol{w}^{l}$ and a ***bias vector*** $\boldsymbol{b}^{l}$



Multilayer Perceptrons (cont.)

- The j-th neuron in the l-th layer has the **activation** a_j^l



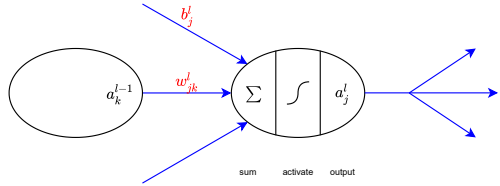
Multilayer Perceptrons (cont.)

- The j-th neuron in the l-th layer first computes a weighted sum of its inputs

$$in = \sum_k w_{jk}^l a_k^{l-1} + b_j^l \tag{3}$$

- Then it applies an **activation function** g to this sum to derive the output:

$$a_j^l = g(in) = g\left(\sum_k w_{jk}^l a_k^{l-1} + b_j^l\right) \tag{4}$$



Activation functions

- Sign function

$$a = \text{sign}(x) = \begin{cases} 1 & x \geq 0 \\ -1 & x < 0 \end{cases} \quad (5)$$

- Sigmoid function

$$a = \sigma(x) = \frac{1}{1 + \exp(-x)} \quad (6)$$

- Linear function

$$a = x \quad (7)$$

- Step function

$$a = \text{step}(x) = \begin{cases} 1 & x \geq 0 \\ 0 & x < 0 \end{cases} \quad (8)$$

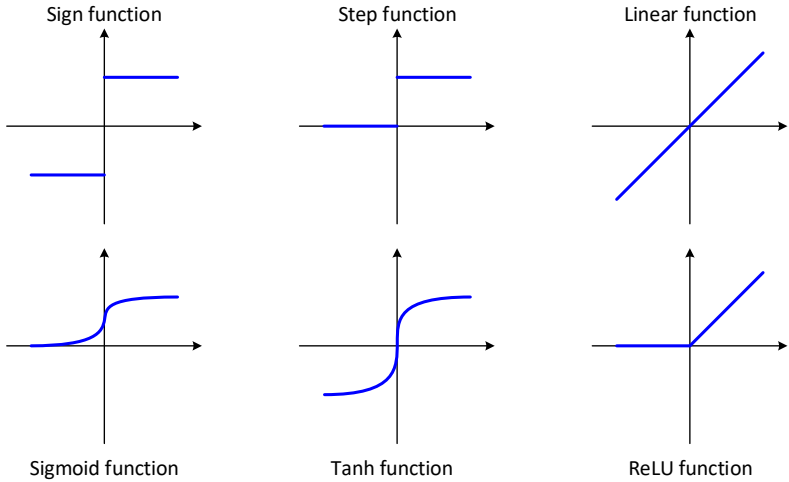
- Tanh function

$$a = \tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}} \quad (9)$$

- ReLU (rectified linear unit)

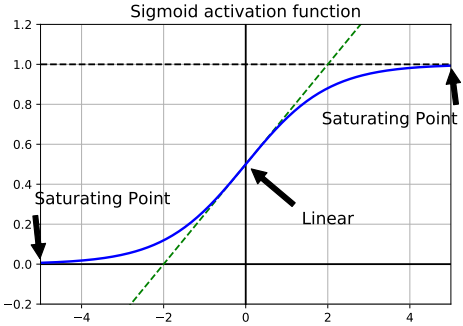
$$a = \text{ReLU}(x) = \begin{cases} x & x \geq 0 \\ 0 & x < 0 \end{cases} \quad (10)$$

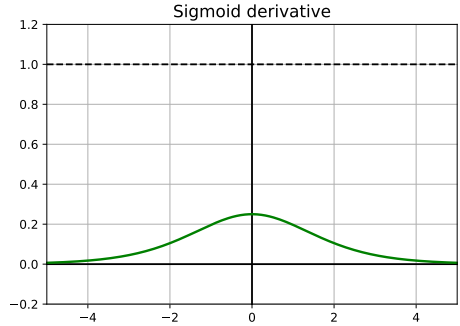
Activation functions (cont.)



Activation functions (cont.)

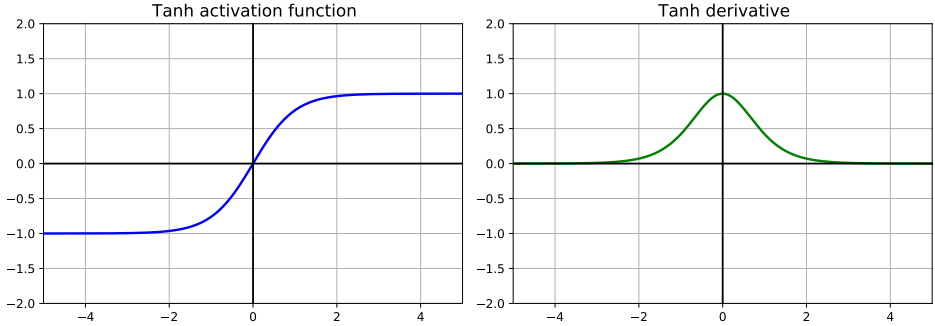
- Sigmoid function





Activation functions (cont.)

- Tanh function



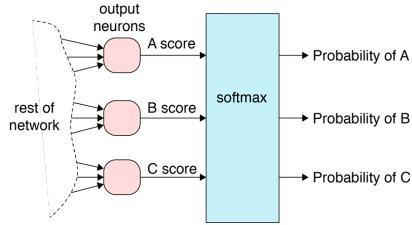
Softmax Gate

- The softmax function takes all the network's outputs and modifies them simultaneously. The result is that the scores are turned into probabilities (this technique is not a normal activation function)

$$f_t(\boldsymbol{x}) = f_t(x_1, x_2, ..., x_k) = \frac{e^{x_i}}{\sum_{j=1}^{k} e^{x_j}}, \text{ for } t = 1, ..., k \tag{11}$$

$$f_t(\boldsymbol{x}) = f_t(x_1, x_2, ..., x_k) = \frac{e^{x_i - M}}{\sum_{j=1}^{k} e^{x_j - M}}, \text{ stable version} \tag{12}$$

where $M = \max(x_1, x_2, ..., x_3)$



Universal Approximation Theorem

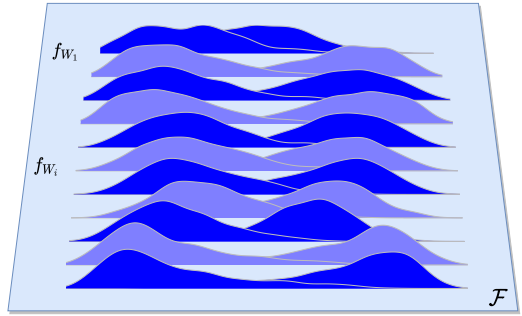
Theorem 1

Expressive Capabilities of MLP

- ***Boolean functions***
  - *Every boolean function can be represented by network with **single hidden layer** but might require exponential (in number of inputs) hidden units*
- ***Continuous functions***
  - *Every bounded continuous function can be approximated with arbitrarily small error, by network with **one hidden layer***
  - *Any function can be approximated to arbitrary accuracy by a network with **two hidden layers***

Learning Model

1. Hypothesis set or function space $\mathcal{F} = \{f \mid y = f(x; W)\}$

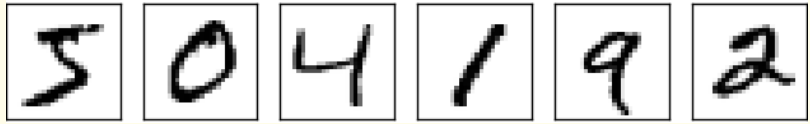


2. Learning algorithm:
   - **Gradient-based algorithms**
   - Delta learning rule
   - ...

Design Neural Network

Example 1

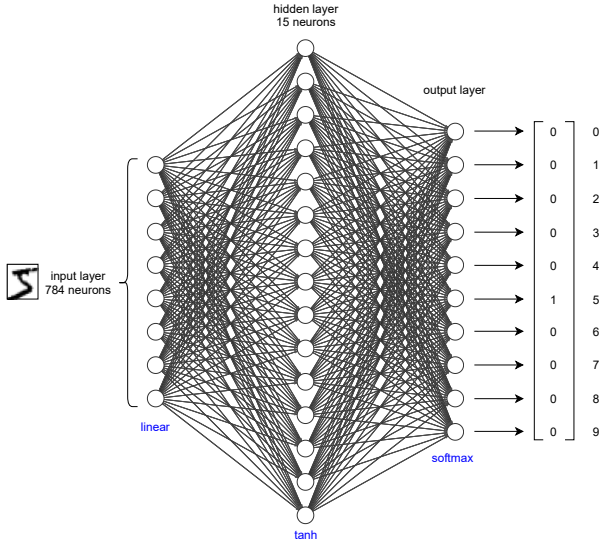
A simple network to classify handwritten digits



Design Neural Network (cont.)

- The input layer of the network contains neurons encoding the values of the input pixels. Our training data for the network will consist of many 28 by 28 pixel images of scanned handwritten digits, and so the input layer contains $784 = 28 \times 28$ neurons
- The second layer of the network is a hidden layer
- The output layer of the network contains 10 neurons. If the first neuron fires, i.e., has an output ≈ 1, then that will indicate that the network thinks the digit is a 0. If the second neuron fires then that will indicate that the network thinks the digit is a 1. And so on

Design Neural Network (cont.)



Design Neural Network (cont.)

Example 2

A simple object classification

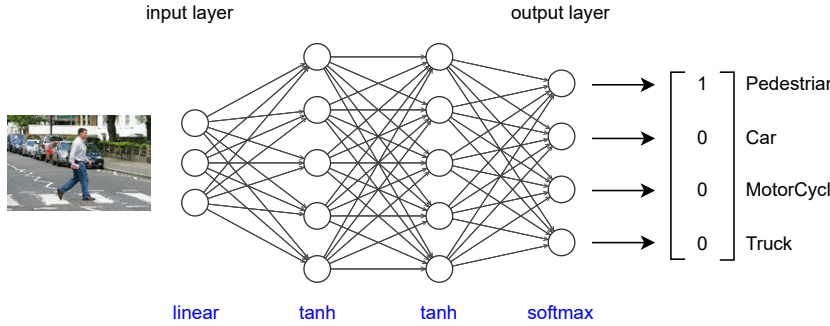
Pedestrian

Car

Motorcycle

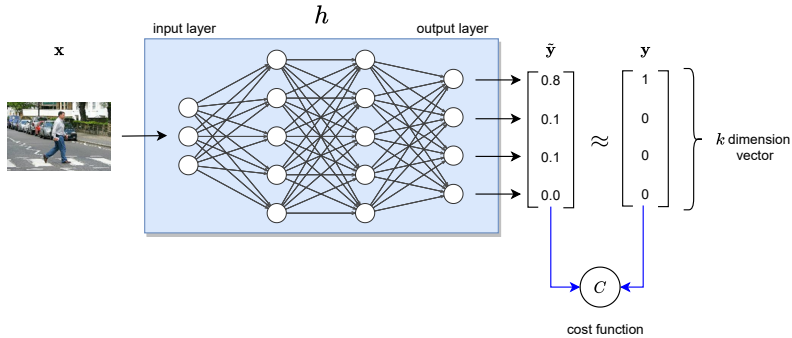
Truck

Design Neural Network (cont.)



Neural Network Learning

- In training step, we **fit** the network to the data based on a cost function



Cost Function

- In the neural network learning, the selection of **output layer** depends on the learning problems.
  - Classification: sigmoid, softmax .etc
  - Regression: linear etc.
- **Cost function** (lost function) can be derived from many methods, the most common of them
  - Mean Square Error
  - Cross Entropy

Cost Function (cont.)

Concept 1 (Mean Square Error)

Let $\boldsymbol{y}$ denotes the desired label of data $\boldsymbol{x}$, and $\tilde{\boldsymbol{y}} = h(\boldsymbol{x})$ as the prediction. The cost function C_{MSE} is defined by

$$C_{MSE} = \frac{1}{k}\sum_{i=1}^{k}(y_i - \tilde{y}_i)^2 \tag{13}$$

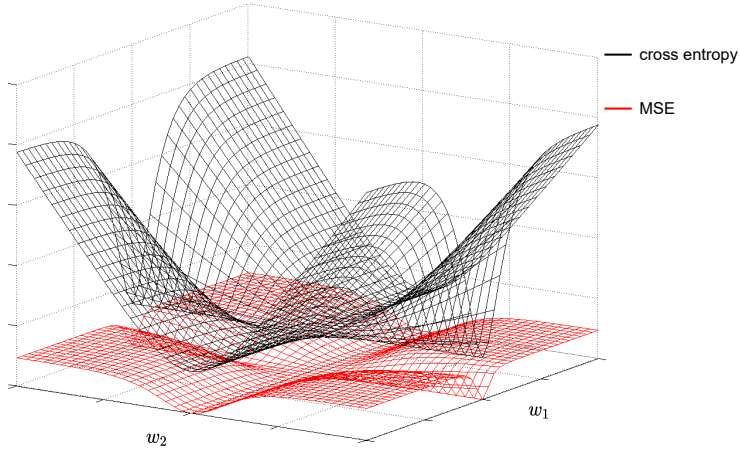
Cost Function (cont.)

Concept 2 (Cross Entropy)

The cost function defined by Cross Entropy is

$$C_{CE} = \sum_{i=1}^{k} y_i \ln \tilde{y}_i \tag{14}$$

Cross Entropy vs. MSE

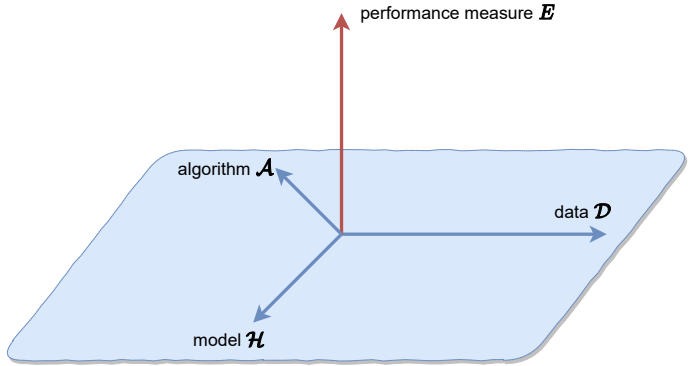


Output Types

Output Type	Output Distribution	Output Layer	Cost Function
Binary	Bernoulli	Sigmoid	Binary cross-entropy
Discrete	Multinoulli	Softmax	Discrete cross-entropy
Continuous	Gaussian	Linear	Gaussian cross-entropy (MSE)
Continuous	Mixture of Gaussian	Mixture Density	Cross-entropy

Regularization

"Regularization is any modification we make to a learning model that is intended to reduce its generalization error but not its training error."



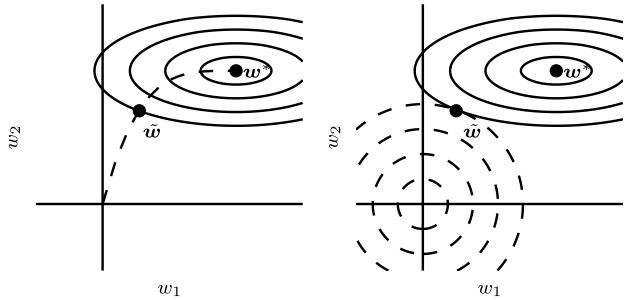
Regularization (cont.)

- To avoid overfitting, we can use the regularized cost function (weight decay)

$$C_{reg}(\boldsymbol{w}) = C(\boldsymbol{w}) + \lambda regularizer(\boldsymbol{w}) \tag{15}$$

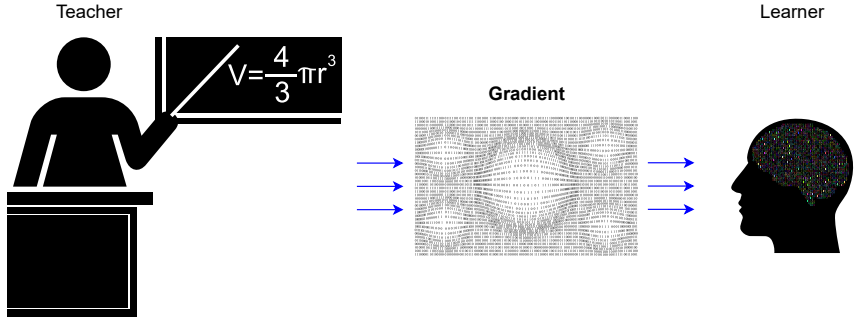
where λ is the regularization coefficient (*hyper-parameter*) that controls the relative importance of the data-dependent error $C(\boldsymbol{w})$ and the regularization term

$$regularizer(\boldsymbol{w}) = \boldsymbol{w}^{\mathsf{T}}\boldsymbol{w} \tag{16}$$



Learning

- **Learning goal**: find h_W minimize the cost function C.
- **Solution**: use gradient-based techniques



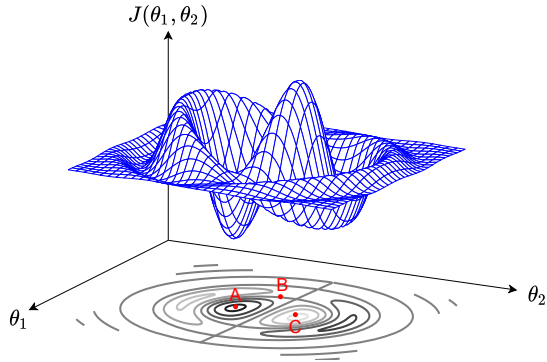
Gradient-Based Learning

- Gradient Descent
- Gradient Ascent
- Min-max optimization

Gradient Descent

- **Objective**: minimize a objective function (cost function) $J(\boldsymbol{\theta}) = J(\theta_1, \theta_2, ..., \theta_D)$

$$\hat{\boldsymbol{\theta}} = \arg\min_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) \tag{17}$$



Gradient Descent (cont.)

- **Gradient descent method**: an iterative procedure where, at each iteration i we modify the parameters $\boldsymbol{\theta}$

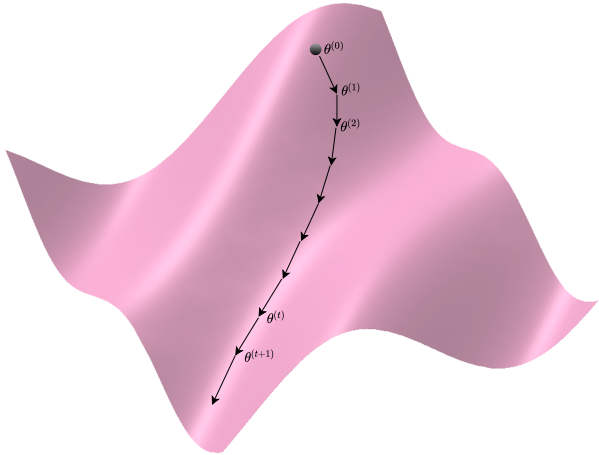
1. **Init** $\boldsymbol{\theta}^{(0)}$
2. **Do until satisfied**
   update $\boldsymbol{\theta}$ to reduce $J(\boldsymbol{\theta})$

$$\boldsymbol{\theta}^{(t+1)} \leftarrow \boldsymbol{\theta}^{(t)} - \eta \nabla_{\boldsymbol{\theta}} J \tag{18}$$

where η is learning rate $(0 < \eta < 1)$ and

$$\nabla_{\boldsymbol{\theta}} J = \left[\frac{\partial J}{\partial \theta_1}, \frac{\partial J}{\partial \theta_2}, \cdots, \frac{\partial J}{\partial \theta_D} \right]^{\mathsf{T}} \tag{19}$$

Gradient Descent (cont.)



Gradient descent variants

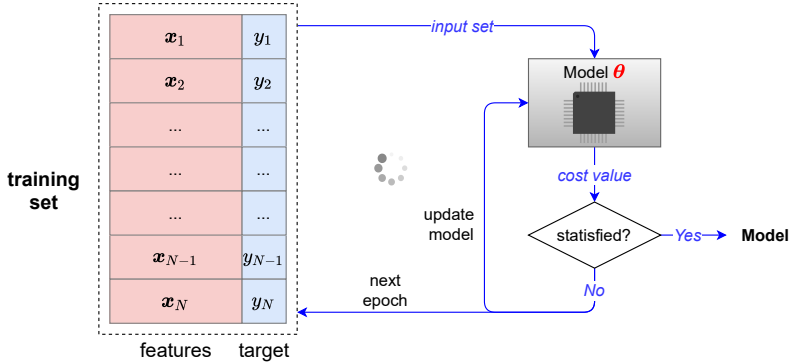
There are three variants of gradient descent. Depending on the amount of data, we make a trade-off between the accuracy of the parameter update and the time it takes to perform an update.

- Batch gradient descent
- Stochastic gradient descent
- Mini-batch gradient descent

Batch gradient descent

- Batch gradient descent (vanilla gradient descent) computes the gradient of the cost function w.r.t. to the parameters $\boldsymbol{\theta}$ for the entire training dataset $\mathcal{D} = \{(\boldsymbol{x}_1, y_1), (\boldsymbol{x}_2, y_2)...(\boldsymbol{x}_N, y_N)\}$

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t - \eta \cdot \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}_t \mid \mathcal{D}) \tag{20}$$



Batch gradient descent (cont.)

Cons

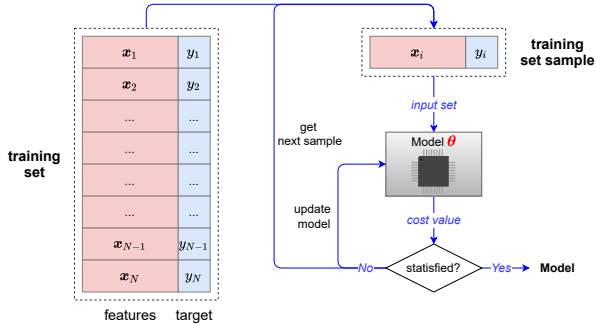
- We need to calculate the gradients for the whole dataset to perform just *one* update
- It can be very slow and is intractable for datasets that do not fit in memory.
- It also does not allow us to update our model *online*, i.e. with new examples on-the-fly.

Stochastic gradient descent

- Stochastic gradient descent (SGD) in contrast performs a parameter update for *each* training example $\boldsymbol{x}_i$ and label y_i of the training dataset $\mathcal{D} = \{(\boldsymbol{x}_1, y_1), (\boldsymbol{x}_2, y_2) ... (\boldsymbol{x}_N, y_N)\}$

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t - \eta \cdot \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}_t \mid \boldsymbol{x}_i, y_i) \tag{21}$$

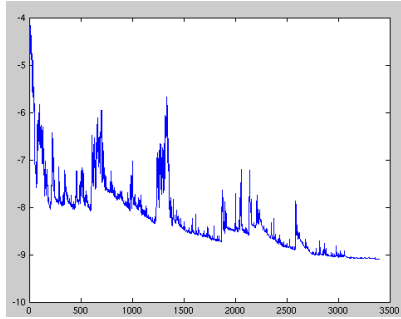
- Note that we should **shuffle** the training data at every epoch



Stochastic gradient descent (cont.)

Cons

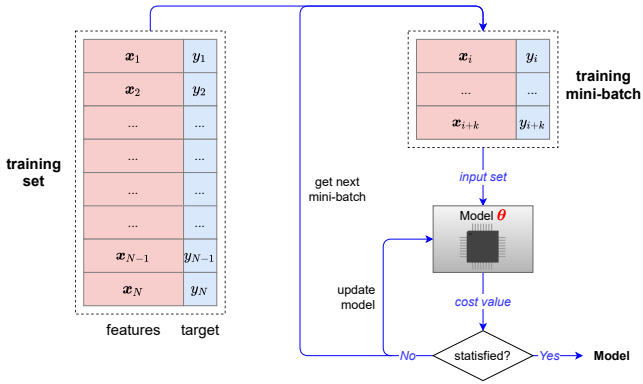
- SGD performs frequent updates with a high variance that cause the objective function to fluctuate heavily



Mini-batch gradient descent

- Mini-batch gradient descent finally takes the best of both worlds and performs an update for every mini-batch of k training examples of the training dataset $\mathcal{D} = \{(\boldsymbol{x}_1, y_1), (\boldsymbol{x}_2, y_2) ... (\boldsymbol{x}_N, y_N)\}$

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t - \eta \cdot \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}_t \mid \boldsymbol{x}_{i:i+k}, y_{i:i+k}) \tag{22}$$



Mini-batch gradient descent (cont.)

Pros

- It reduces the variance of the parameter updates, which can lead to more stable convergence
- It can make use of highly optimized matrix optimizations common to state-of-the-art deep learning libraries that make computing the gradient w.r.t. a mini-batch very efficient.

Gradient Ascent

- **Objective**: maximize a objective function (cost function) $J(\boldsymbol{\theta}) = J(\theta_1, \theta_2, ..., \theta_D)$

$$\hat{\boldsymbol{\theta}} = \arg\max_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) \tag{23}$$

1. **Init** $\boldsymbol{\theta}^{(0)}$
2. **Do until satisfied**
   update $\boldsymbol{\theta}$ to increase $J(\boldsymbol{\theta})$

$$\boldsymbol{\theta}^{(t+1)} \leftarrow \boldsymbol{\theta}^{(t)} + \eta \nabla_{\boldsymbol{\theta}} J \tag{24}$$

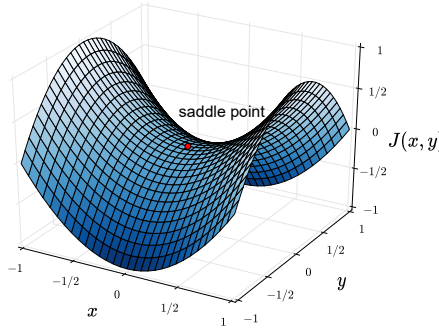
where η is learning rate $(0 < \eta < 1)$ and

$$\nabla_{\boldsymbol{\theta}} J = \left[\frac{\partial J}{\partial \theta_1}, \frac{\partial J}{\partial \theta_2}, \cdots, \frac{\partial J}{\partial \theta_D} \right]^{\mathsf{T}} \tag{25}$$

Min-max

- **Objective**: Given a objective function (cost function) $J(x, y)$

$$\hat{x}, \hat{y} = \arg\min_{x} \max_{y} J(x, y) \tag{26}$$



Min-max (cont.)

- GAN optimization (original version), does not always converge

1. **Init** $x^{(0)}, y^{(0)}$
2. **Do until satisfied**
   update x, y

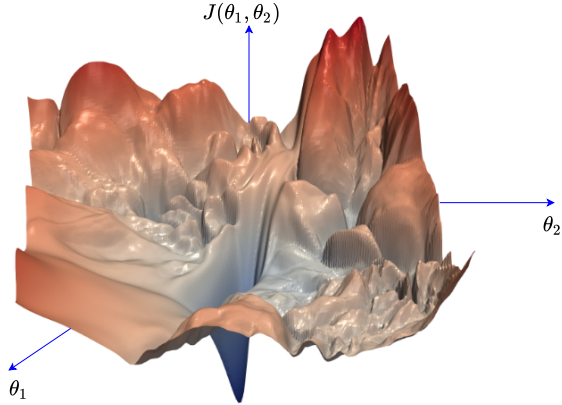
$$x^{(t+1)} \leftarrow x^{(t)} - \rho \nabla_x J \tag{27}$$

$$y^{(t+1)} \leftarrow y^{(t)} + \eta \nabla_y J \tag{28}$$

Advanced Gradient-Based Learning

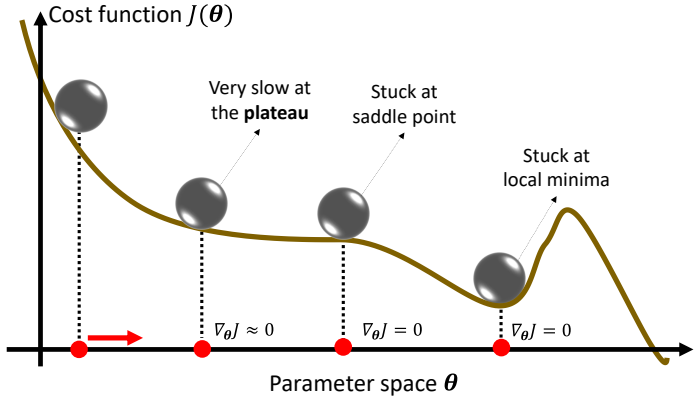
Gradient descent: Local minima

- Gradient descent never guarantee **global minima**
  - Different initial point $\boldsymbol{\theta}^{(0)} \rightarrow$ reach different minima, so different results



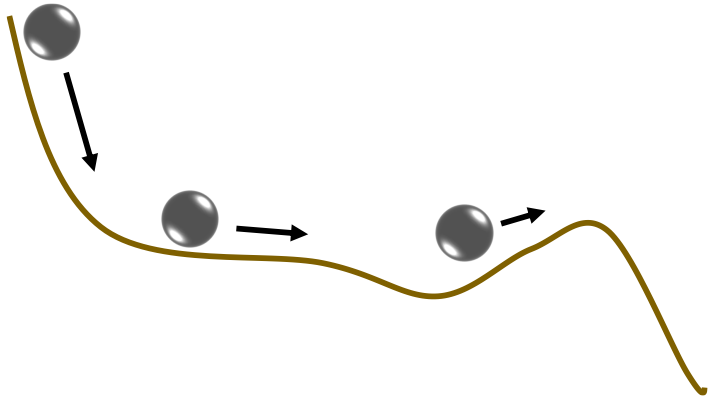
More issues of gradient descent

- It also has issues at **plateau** and **saddle point**



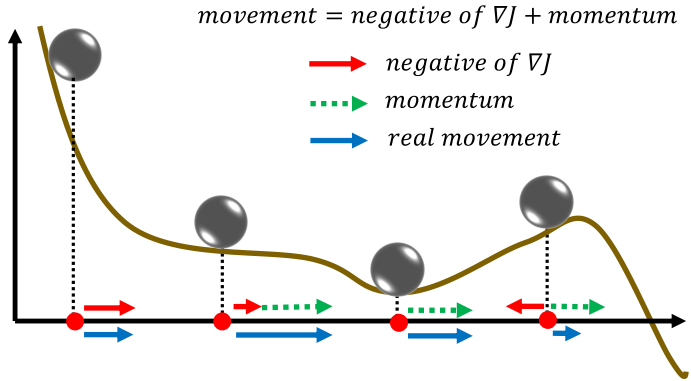
Momentum in physical world

- How about put this phenomenon in gradient descent?



Momentum

- Still not guarantee reaching global minima, but give some hope ...



Momentum

- **Momentum** is a method that helps accelerate SGD in the relevant direction and dampens oscillations
- It does this by adding a fraction γ of the update vector of the past time step to the current update vector

$$\begin{aligned} \boldsymbol{v}_0 &= \mathbf{0} \\ \boldsymbol{v}_t &= \gamma \boldsymbol{v}_{t-1} + \eta \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}_{t-1}) \\ \boldsymbol{\theta}_t &= \boldsymbol{\theta}_{t-1} - \boldsymbol{v}_t \end{aligned} \tag{29}$$

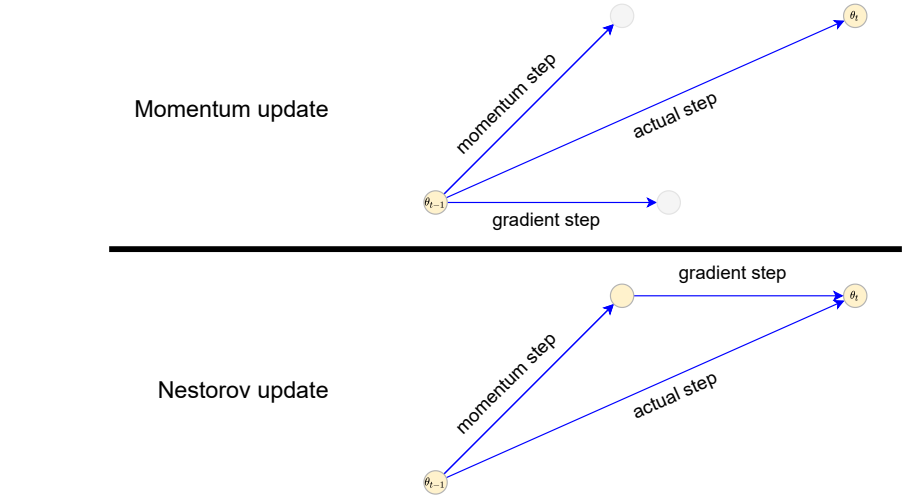
- The momentum term γ is usually set to 0.9 or a similar value.

Nesterov accelerated gradient (NAG)

- A ball that rolls down a hill, blindly following the slope, is highly unsatisfactory. We would like to have a smarter ball, a ball that has a notion of where it is going so that it knows to slow down before the hill slopes up again.
- We can now effectively look ahead by calculating the gradient not w.r.t. to our current parameters $\boldsymbol{\theta}$ but w.r.t. the approximate future position of our parameters:

$$\begin{aligned} \boldsymbol{v}_0 &= \mathbf{0} \\ \boldsymbol{v}_t &= \gamma \boldsymbol{v}_{t-1} + \eta \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}_{t-1} - \gamma \boldsymbol{v}_{t-1}) \\ \boldsymbol{\theta}_t &= \boldsymbol{\theta}_{t-1} - \boldsymbol{v}_t \end{aligned} \tag{30}$$

Nesterov accelerated gradient (NAG) (cont.)



Adagrad

- Previously, we performed an update for all parameters $\boldsymbol{\theta}$ at once as every parameter θ_i used the **same learning rate** η. As Adagrad (Duchi, 2011) uses a **different learning rate** for every parameter θ_i at every time step t

$$\theta_{t+1,i} = \theta_{t,i} - \eta_{t,i} \cdot g_{t,i} \tag{31}$$

where

$$g_{t,i} = \frac{\partial J}{\partial \theta_{t,i}} \tag{32}$$

and

$$\eta_{t,i} = \frac{\eta}{\sqrt{\sum_{k=0}^{t} g_{k,i}^2}}$$

Adagrad (cont.)

Pros

- One of Adagrad's main benefits is that it eliminates the need to manually tune the learning rate η. Most implementations use a default value of 0.01 and leave it at that.

Cons

- Since every added term is positive, the accumulated sum keeps growing during training. This in turn causes the learning rate to shrink and eventually become infinitesimally small, at which point the algorithm is no longer able to acquire additional knowledge.

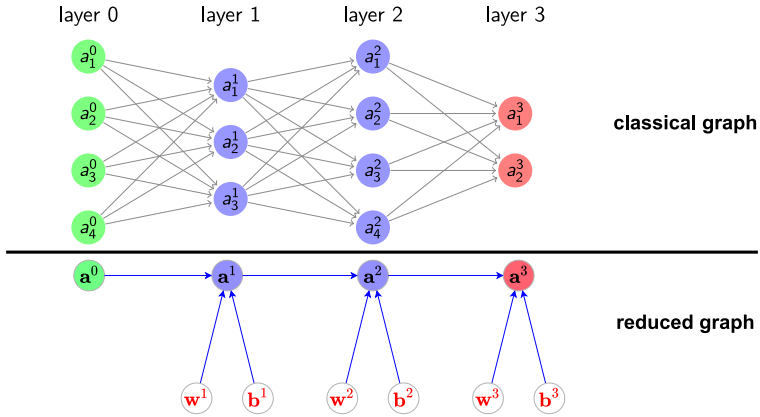
Other Learning Methods rather than Adagrad

- **Adadelta** (Zeiler, 2012) is an extension of Adagrad that seeks to reduce its aggressive, monotonically decreasing learning rate. Instead of accumulating all past squared gradients, Adadelta restricts the window of accumulated past gradients to some fixed size w.
- **RMSprop** is an adaptive learning rate method proposed by Geoff Hinton.
- **Adadelta**
- **Adam**
- **AdaMax**
- **Nadam**

Back-propagation Learning

Computational Graph

- The **classical graph** of neural network is cumbersome at first, and it does take some work to master
- We use the another reperesentaion, **reduced graph** (**computational graph**)



The back-propagation algorithm

- Proposed by Bryson and Ho, 1969 $\rightarrow$ **most popular** among over a hundred different learning algorithms available.
- The algorithm provide us with a way of computing the gradient of the cost function $C(\boldsymbol{w}, \boldsymbol{b})$.
- **Input** $(\boldsymbol{x}, \boldsymbol{y})$: Set the corresponding activation $\boldsymbol{a}^0$ for the input layer.

Feedforward:

1. Init $\boldsymbol{a}^0 = \boldsymbol{x}$
2. For each $l = 1, 2, \ldots, L$ compute

$$\boldsymbol{z}^l = \boldsymbol{w}^l \boldsymbol{a}^{l-1} + \boldsymbol{b}^l \qquad \text{and} \qquad \boldsymbol{a}^l = \sigma(\boldsymbol{z}^l) \tag{33}$$

   where σ is an activation function
3. Compute the cost value C

The back-propagation algorithm (cont.)

Backpropagate:

1. Compute **the error vector** δ^L

$$\boldsymbol{\delta}^L = \sigma'(\boldsymbol{z}^L) \odot \nabla C \tag{34}$$

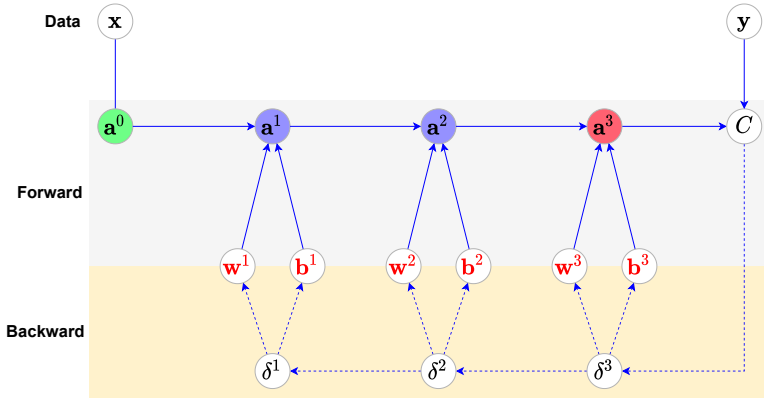
2. For each $l = L-1, L-2, \ldots, 1$, compute **the error vector** δ^l

$$\boldsymbol{\delta}^l = ((\boldsymbol{w}^{l+1})^\intercal \boldsymbol{\delta}^{l+1}) \odot \sigma'(\boldsymbol{z}^l) \tag{35}$$

3. Compute the partial gradient of the cost function with respect to parameters

$$\frac{\partial C}{\partial w^l_{jk}} = a^{l-1}_k \delta^l_j \qquad \text{and} \qquad \frac{\partial C}{\partial b^l_j} = \delta^l_j \tag{36}$$

The back-propagation algorithm (cont.)



References

Goodfellow, I., Bengio, Y., and Courville, A. (2016).
Deep learning.
MIT press.

Lê, B. and Tô, V. (2014).
Cở sở trí tuệ nhân tạo.
Nhà xuất bản Khoa học và Kỹ thuật.

Russell, S. and Norvig, P. (2021).
Artificial intelligence: a modern approach.
Pearson Education Limited.