San José State University

Math 251: Statistical and Machine Learning Classification

Neural Networks

Dr. Guangliang Chen

Outline of the presentation:

- Overview
  - What is a neural network
  - What is a neuron
- Perceptrons
- Sigmoid neurons network: training and practical issues
- Summary

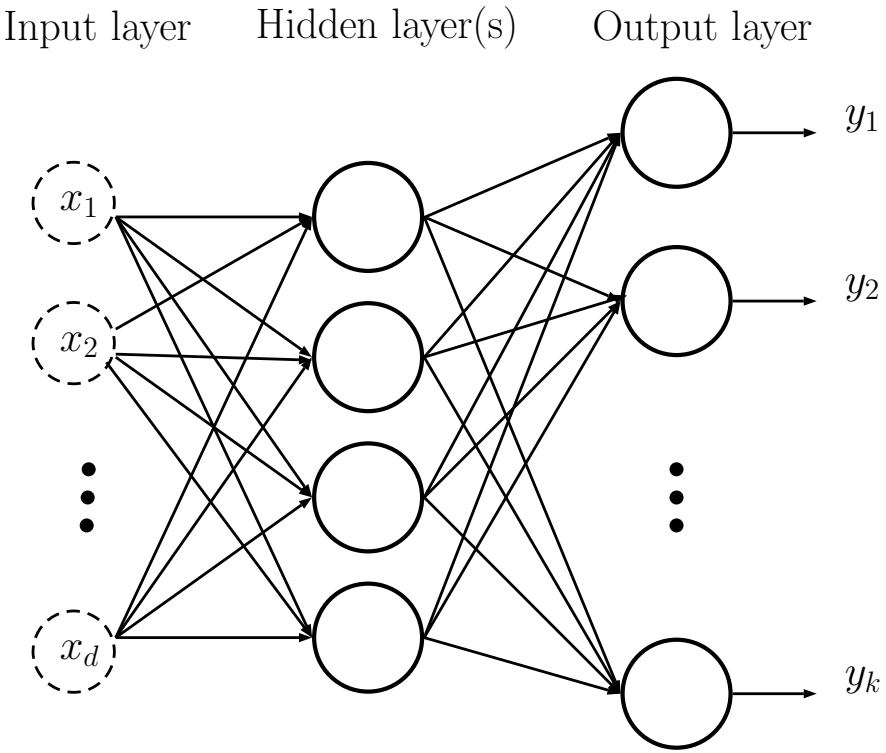
Acknowledgments

This presentation is based on the following references:

- **Nielsen's online book "Neural Networks and Deep Learning"**[1]
- **Olga Veksler's lecture on neural networks**[2]

[1] `http://neuralnetworksanddeeplearning.com`
[2] `http://www.csd.uwo.ca/courses/CS9840a/Lecture10_NeuralNets.pdf`

What is an artificial neural network?



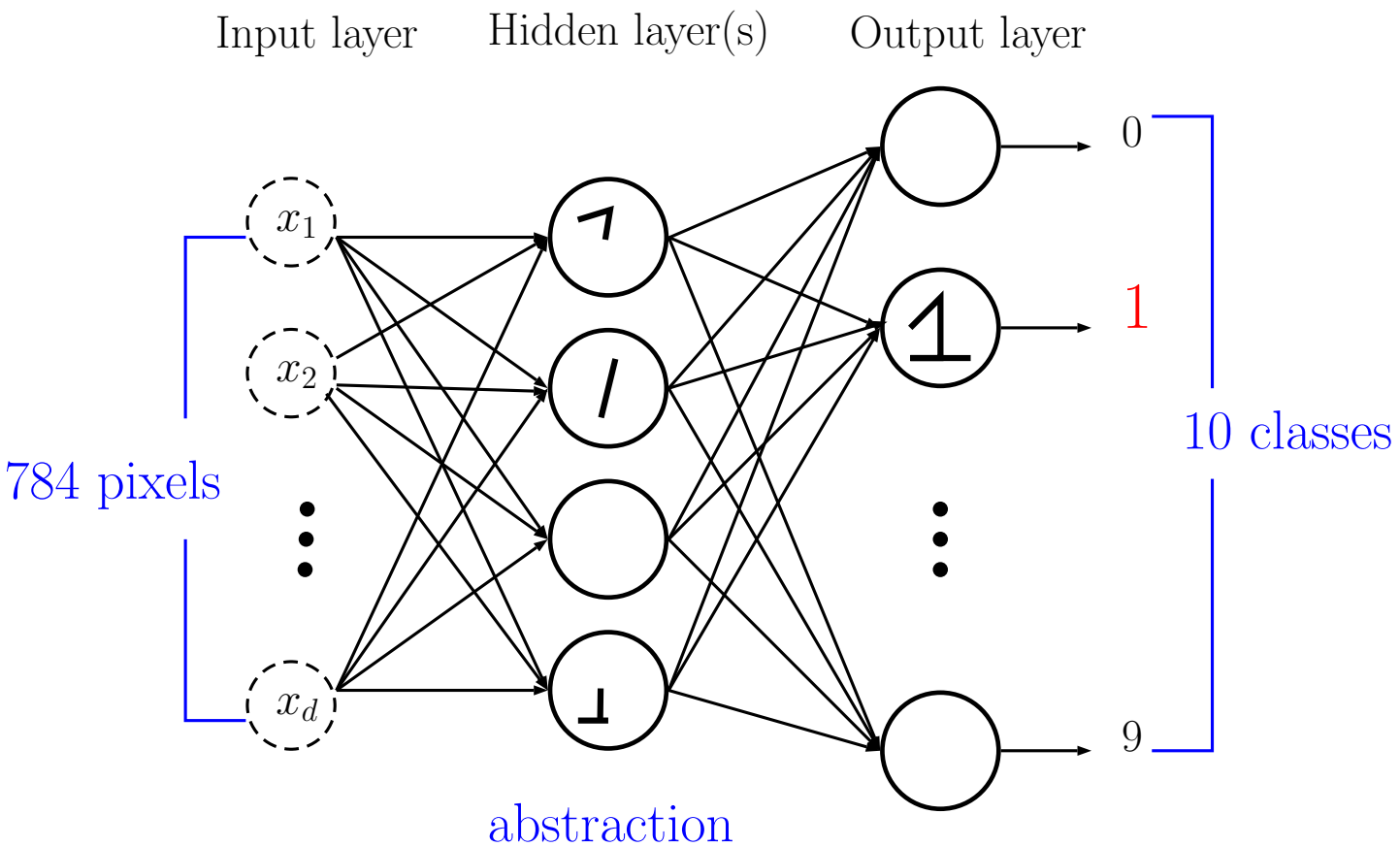
The leftmost layer inputs features x_i.

The rightmost layer outputs predictions y_i.

The solid circles represent **neurons**, which process inputs from preceding layer and output results for next layer.

The neural network is called a **deep** network if it has more than one layer (otherwise, it is said to be shallow).

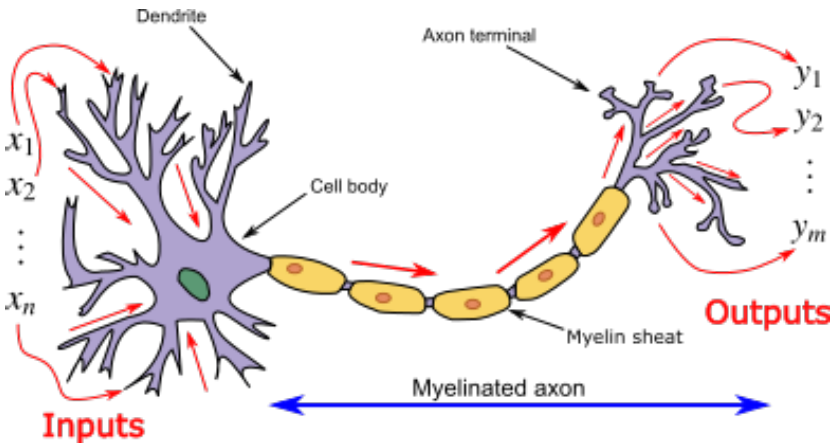
ANN for MNIST handwritten digits recognition



What is a biological neuron?

- Neurons (or nerve cells) are special cells that process and transmit information by electrical signaling (in brain and also spinal cord)
- Human brain has around 10^{11} neurons
- A neuron connects to other neurons to form a network
- Each neuron cell communicates to between 1000 and 10,000 other neurons

Components of a biological neuron



dendrites:

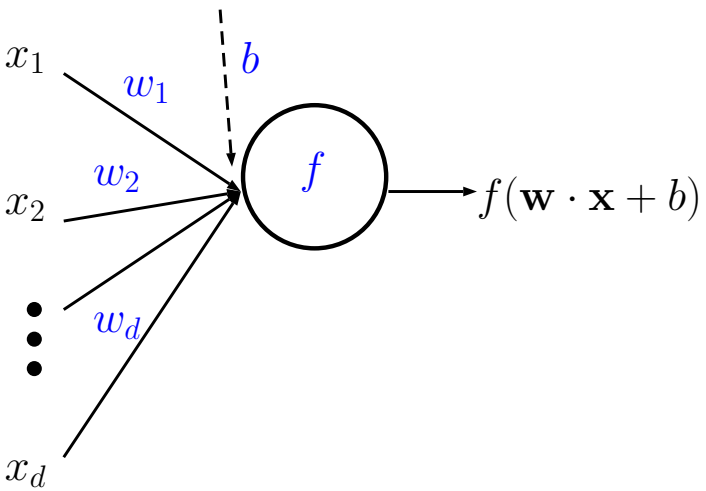
- "input wires", receive inputs from other neurons
- a neuron may have thousands of dendrites, usually short

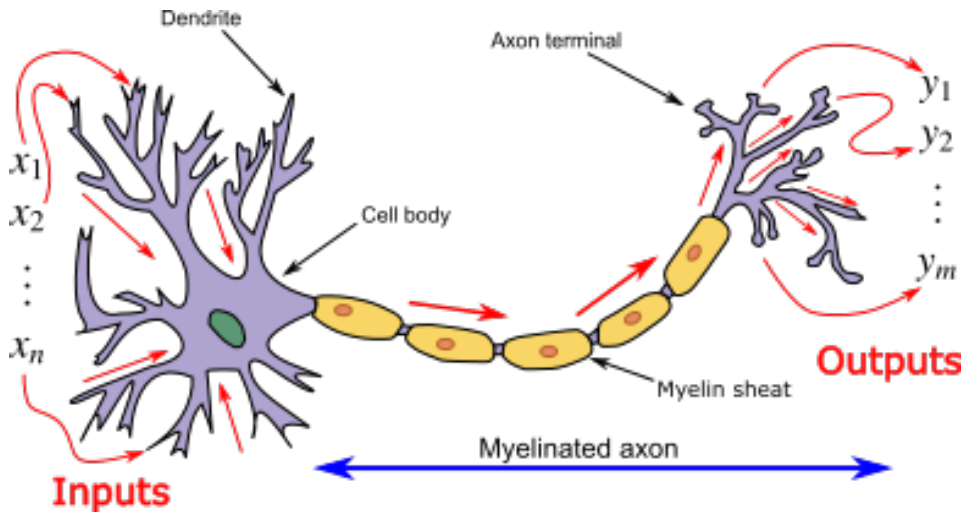
cell body: computational unit

axon:

- "output wire", sends signal to other neurons
- single long structure (up to 1 m)
- splits in possibly thousands of branches at the end

Artificial neurons are mathematical functions



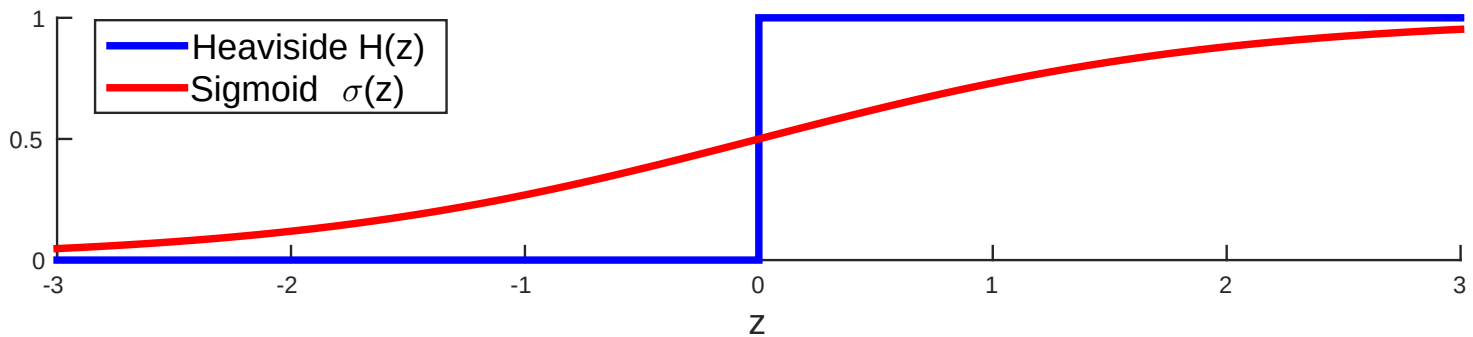


In the above,

- w_i: **weights**, b: **bias**, and f: **activation function**

Two simple activation functions

- Heaviside step function: $H(z) = 1_{z>0}$
- Sigmoid: $\sigma(z) = \frac{1}{1+e^{-z}}$

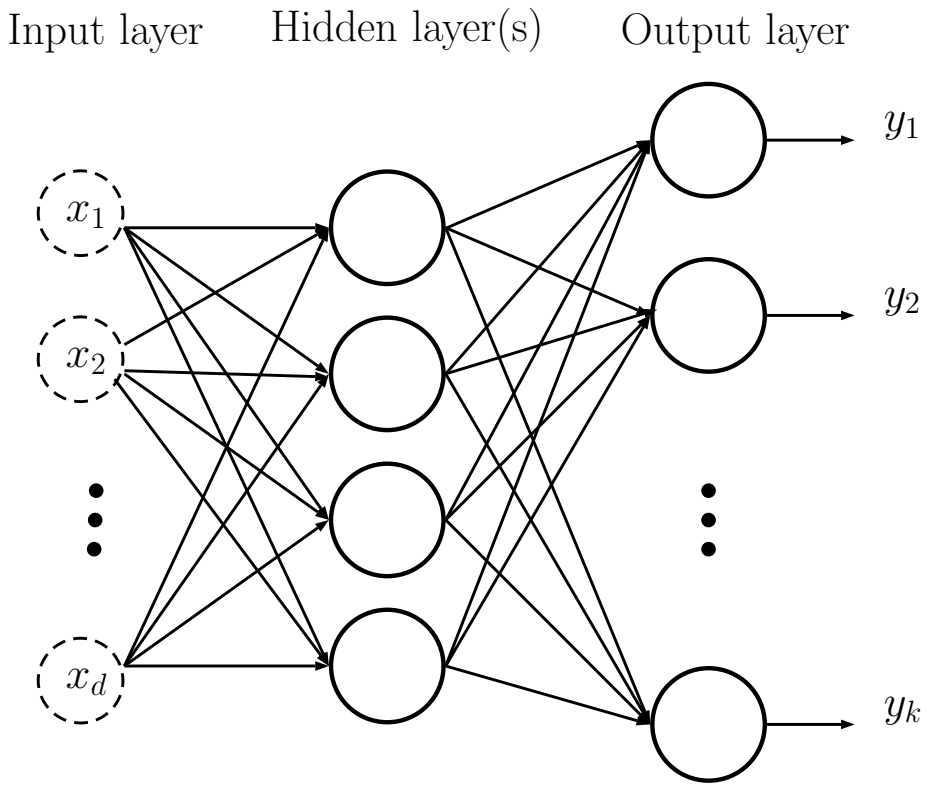


The corresponding neurons are called perceptrons and sigmoid neurons.

We will mention several other activation functions in the end.

ANN is a composition of functions!

- Each neuron is a function
- It accepts inputs from previous layer and outputs for next layer



It can be proved that every continuous function can be implemented with 1 hidden layer (containing enough hidden units) and proper nonlinear activation functions.

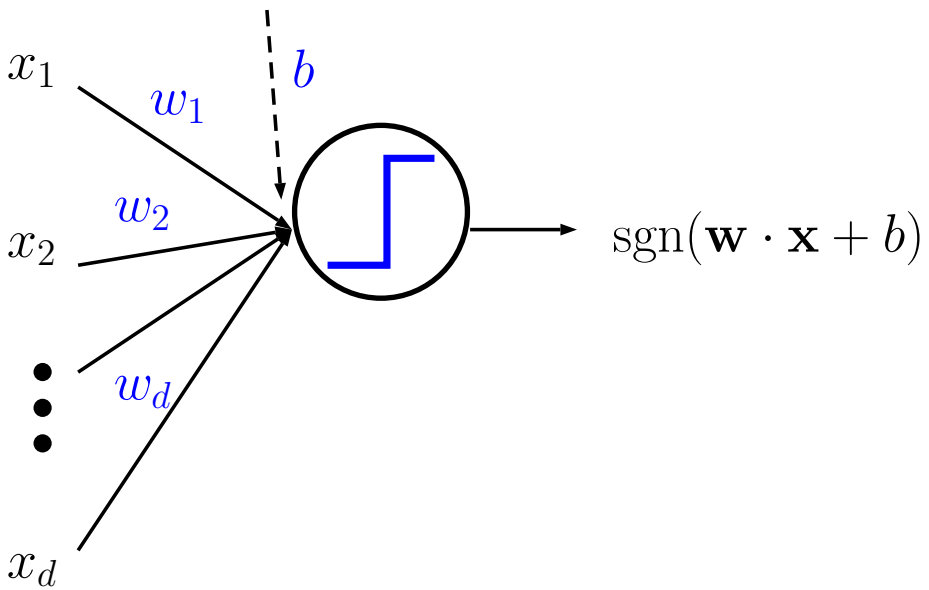
This is of theoretical importance.

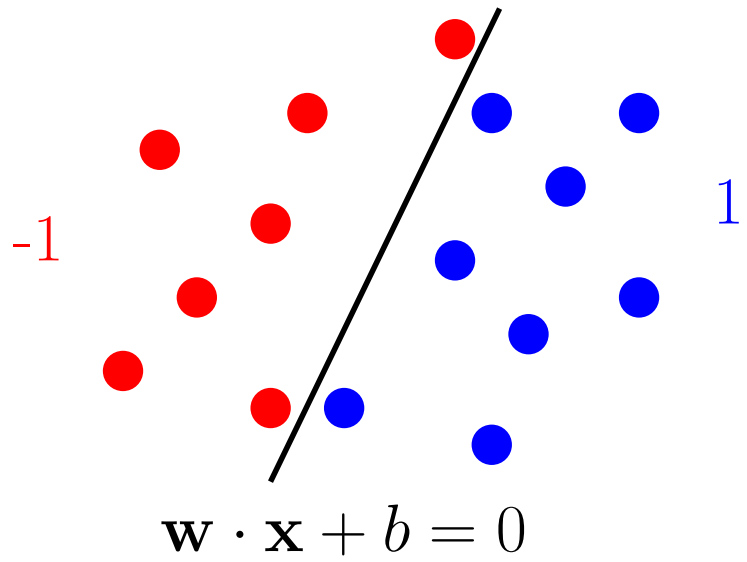
How to train ANNs in principle

1. Select an activation function for all neurons.
2. Tune weights and biases at all neurons to match prediction and truth "as closely as possible":
    - formulate an objective or loss function L
    - optimize it with gradient descent
        - the technique is called backpropagation
        - lots of notation due to complex from of gradient
        - lots of tricks to get gradient descent work reasonably well

Perceptrons

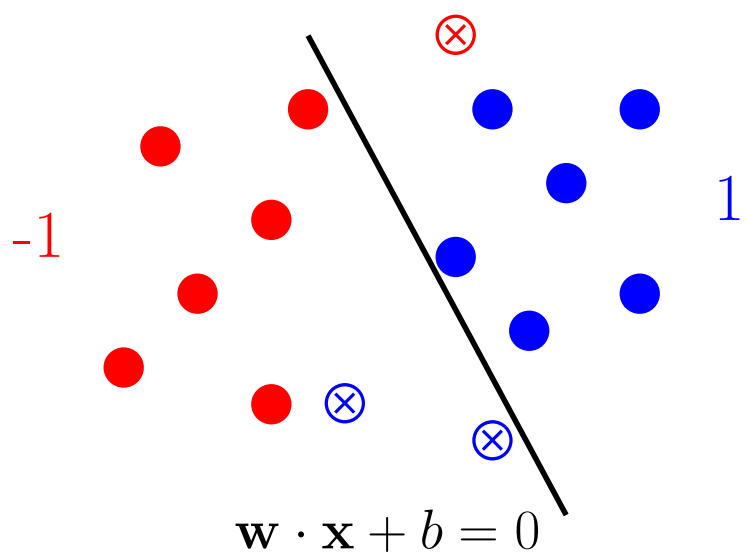
A perceptron is a neuron whose activation function is the Heaviside step function. It defines a linear, binary classifier (not necessarily optimal).





Derivation of the Perceptron loss function

- If a point $\mathbf{x}_i$ is misclassified, then $y_i(\mathbf{w}\cdot\mathbf{x}_i+b)<0$ (implying that $-y_i(\mathbf{w}\cdot\mathbf{x}_i+b)>0$, which can be regarded as penalty/loss).



- Denote the set of misclassified points by $\mathcal{M}$.
- Minimize the total loss

$$\ell(\mathbf{w},b)=-\sum_{i\in\mathcal{M}} y_i(\mathbf{w}\cdot\mathbf{x}_i+b)$$

over **unit vectors** $\mathbf{w}$, scalars b.

- If ℓ gets to zero, we know we have the best possible solution ($\mathcal{M}$ empty $\rightarrow$ no training error)

How to minimize the perceptron loss

The perceptron loss contains a discrete object (i.e. $\mathcal{M}$) that depends on the variables $\mathbf{w}, b$, making it hard to solve analytically.

To obtain an approximate solution, use gradient descent in an alternating fashion:

Initialize with random choices of $\mathbf{w}$ and b:

- Given weights $\mathbf{w}$ and bias b: determine $\mathcal{M}$;
- Given $\mathcal{M}$: update $\mathbf{w}$ and b slightly using gradient descent

Iterate until stopping criterion is met.

The details are shown on next slide.

- *Given* $\mathcal{M}$: The gradient may be computed as follows

$$\frac{\partial \ell}{\partial \mathbf{w}} = -\sum_{i \in \mathcal{M}} y_i \mathbf{x}_i, \quad \frac{\partial \ell}{\partial b} = -\sum_{i \in \mathcal{M}} y_i$$

We then update $\mathbf{w}, b$ as follows:

$$\mathbf{w} \longleftarrow \mathbf{w} + \rho \sum_{i \in \mathcal{M}} y_i \mathbf{x}_i, \quad b \longleftarrow b + \rho \sum_{i \in \mathcal{M}} y_i$$

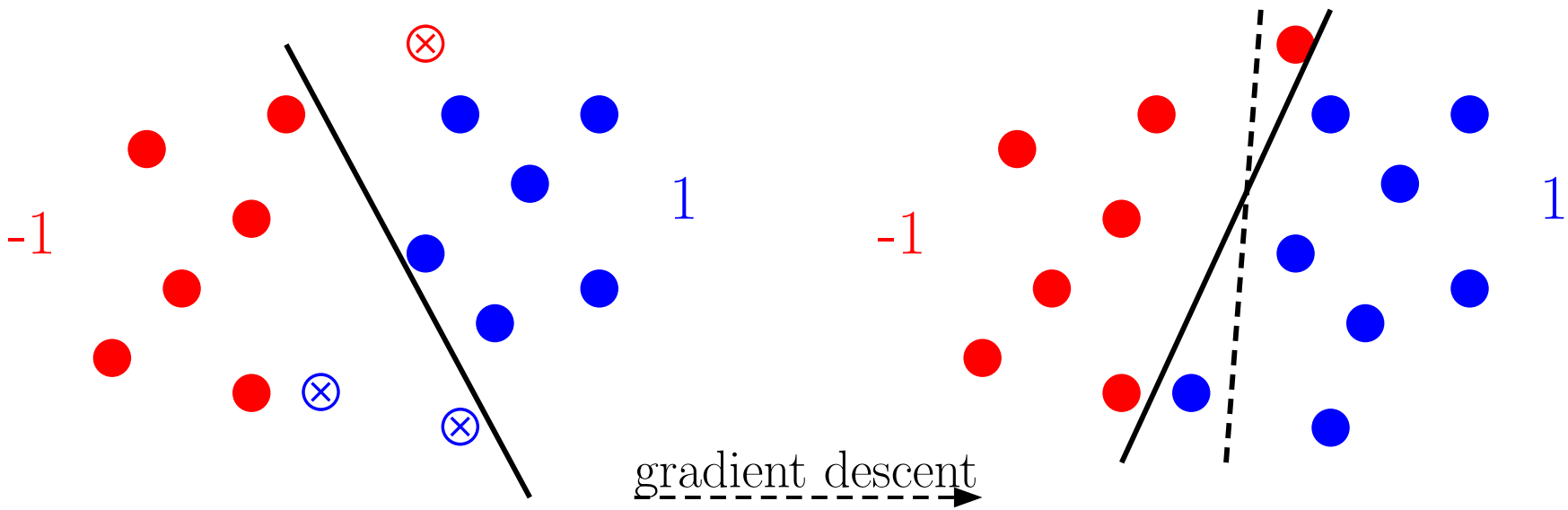
where $\rho > 0$ is a parameter, called **learning rate**.

Renormalize $\mathbf{w}$ to be a unit vector and scale b accordingly:

$$b \longleftarrow \frac{b}{\|\mathbf{w}\|}, \quad \mathbf{w} \longleftarrow \frac{\mathbf{w}}{\|\mathbf{w}\|}$$

- *Given* $\mathbf{w}, b$: update $\mathcal{M}$ as the set of new errors:

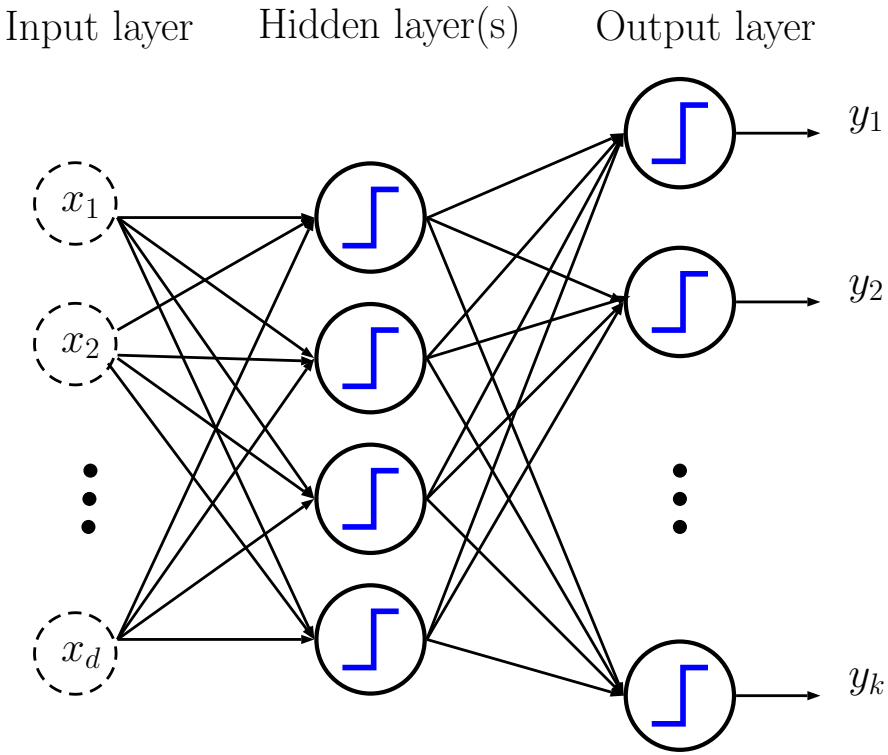
$$\mathcal{M} = \{1 \leq i \leq n \mid y_i(\mathbf{w} \cdot \mathbf{x}_i + b) < 0\}$$



Some remarks about the Perceptron algorithm

- If the classes are linearly separable, the algorithm converges to a separating hyperplane in a finite number of steps, but not necessarily optimal.
- The number of steps can be very large. The smaller the margin (between the classes), the longer it takes to find it.
- When the data are not separable, the algorithm will not converge, and cycles develop (which can be long and therefore hard to detect).
- It is thus not a good classifier, but it is conceptually very important (neuron, loss function, gradient descent).

Multilayer perceptrons (MLP)



MLP is a network of perceptrons.

However, each perceptron has a discrete behavior, making its effect on latter layers hard to predict.

Next we will look at the network of sigmoid neurons.

Sigmoid neurons

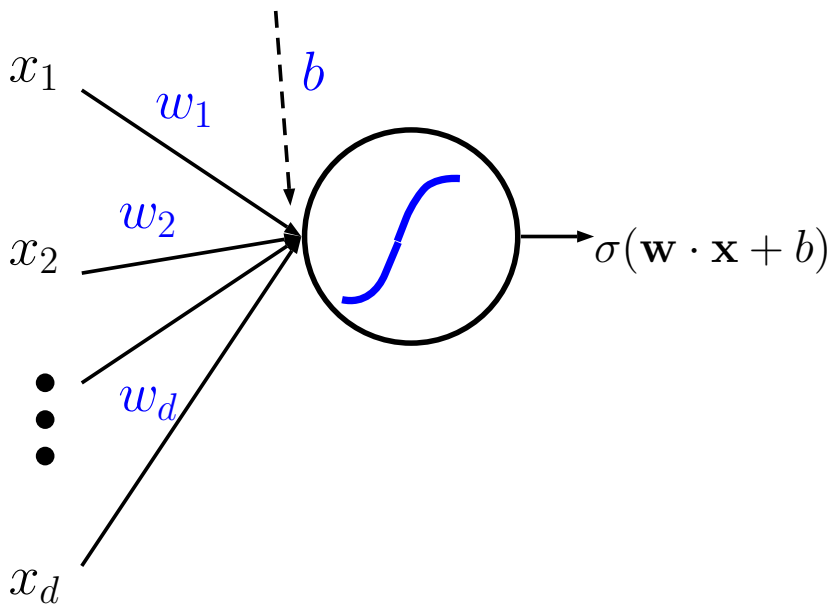
Sigmoid neurons are smoothed-out (or soft) versions of the perceptrons:

$$\sigma(\mathbf{w}\cdot\mathbf{x}+b)=\frac{1}{1+e^{-(\mathbf{w}\cdot\mathbf{x}+b)}}$$

A small change in any weight or bias causes only a small change in output.

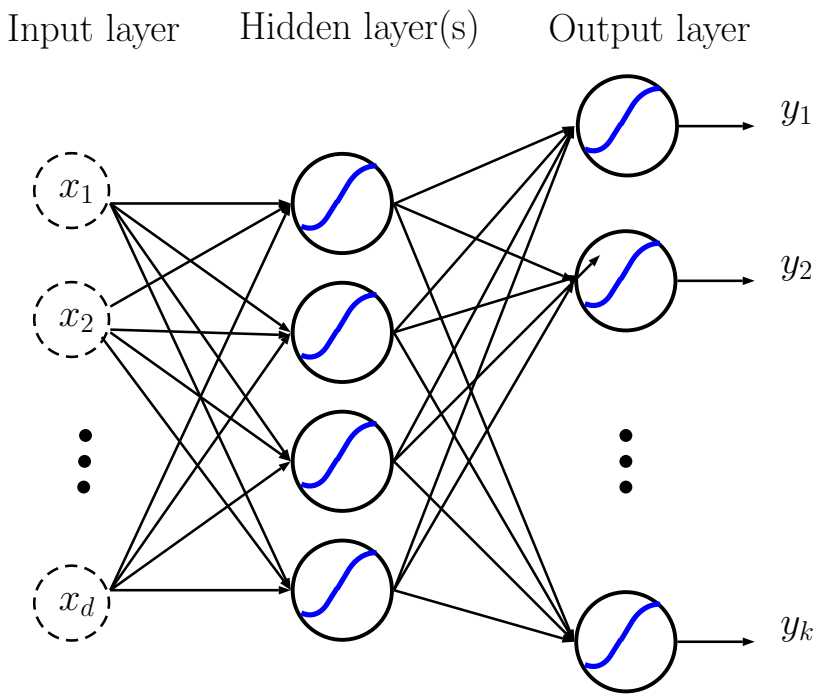
We say that the neuron is in low (high) activation if the output is near 0 (1). In both cases, we say that the neuron has *saturated*.

When the neuron is in high activation, we say that it fires.



The sigmoid neurons network

The output of such a network continuously depends on its weights and biases (so everything is more predictable comparing to the MLP).



How do we train a neural network?

- Notation, notation, notation
- Backpropagation
- Practical issues and solutions

Notation

Let ℓ be the layer index: input layer (0); output layer (L).

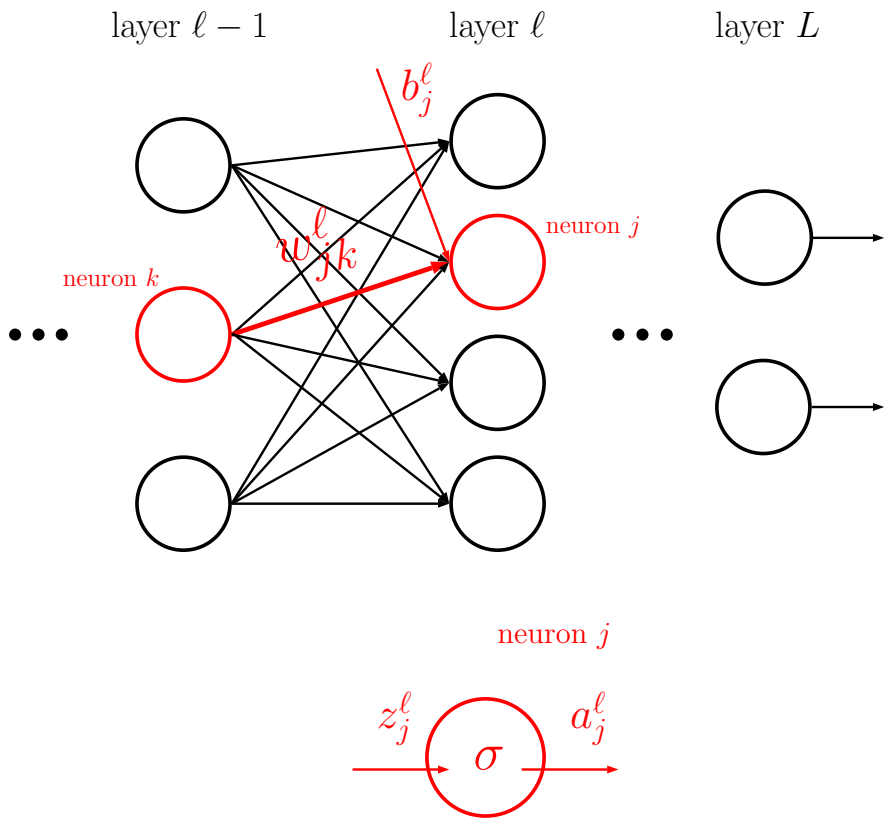
For each $\ell = 1, \ldots, L$:

- w_{jk}^{ℓ}: layer ℓ, "j back to k" weight;

- b_j^{ℓ}: layer ℓ, neuron j bias

- z_j^{ℓ}: total input to neuron j in layer ℓ

- a_j^{ℓ}: layer ℓ, neuron j output



Notation (vector form)

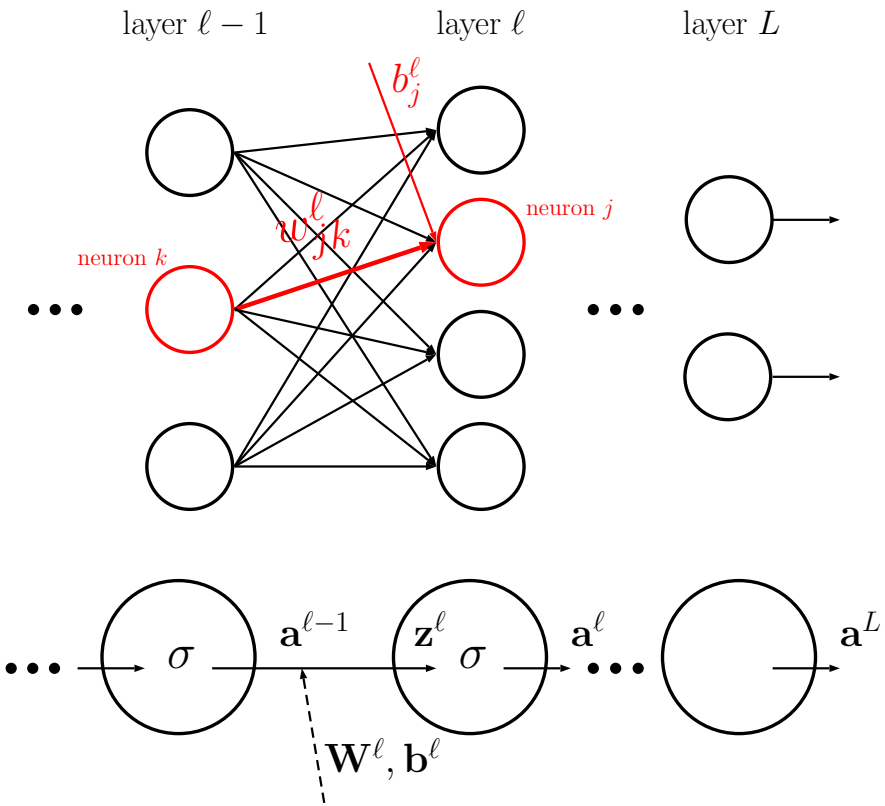
$\mathbf{W}^\ell = \left(w^\ell_{jk}\right)_{j,k}$: matrix of all weights between layers $\ell - 1$ and ℓ;

$\mathbf{b}^\ell = \left(b^\ell_j\right)_j$: vector of biases in layer ℓ

$\mathbf{z}^\ell = \left(z^\ell_j\right)_j$: vector of weighted inputs to neurons in layer ℓ

$\mathbf{a}^\ell = \left(a^\ell_j\right)_j$: vector of outputs from neurons in layer ℓ

Note that $\mathbf{a}^0 = \mathbf{x}$: input to network, and $\mathbf{a}^L$: network output.



The feedforward relationship

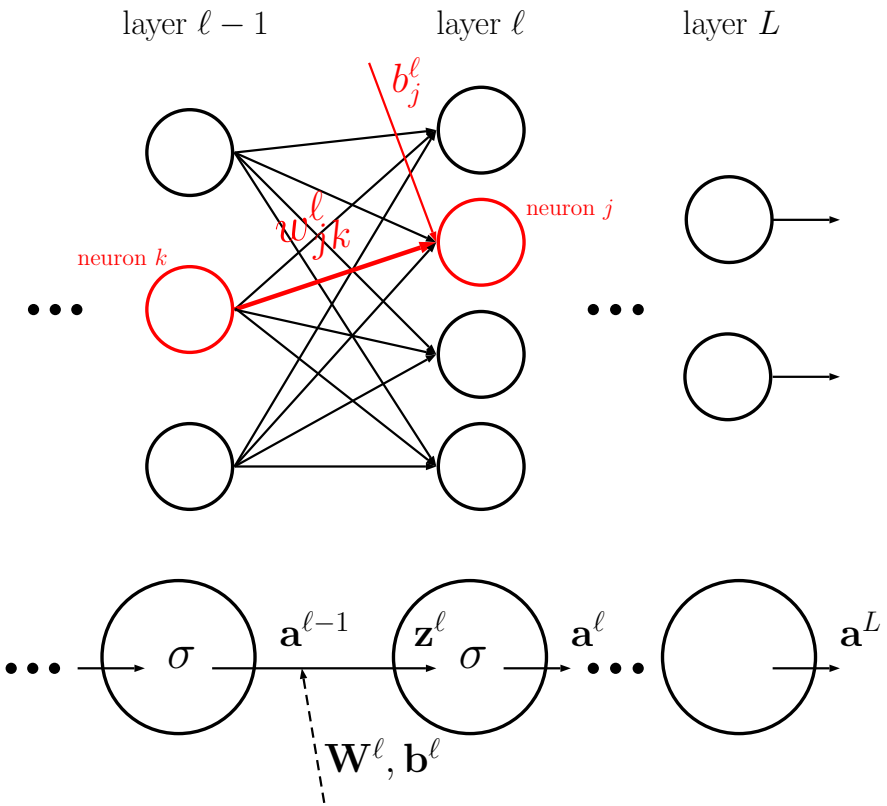
First note that at neuron j in layer ℓ:

$$z_j^\ell = \sum_k w_{jk}^\ell a_k^{\ell-1} + b_j^\ell,$$
$$a_j^\ell = \sigma(z_j^\ell)$$

On the entire layer ℓ:

$$\mathbf{z}^\ell = \mathbf{W}^\ell \mathbf{a}^{\ell-1} + \mathbf{b}^\ell,$$
$$\mathbf{a}^\ell = \sigma(\mathbf{z}^\ell)$$

where in the last equation, σ is applied in a componentwise fashion.



The network loss

To tune the weights and biases of a network of sigmoid neurons, we need to select a loss function.

We first consider the square loss due to its simplicity

$$C(\{\mathbf{W}^\ell, \mathbf{b}^\ell\}_{1\leq \ell \leq L}) = \frac{1}{2n}\sum_{i=1}^{n} \|\mathbf{a}^L(\mathbf{x}_i) - \mathbf{y}_i\|^2$$

where

- $\mathbf{a}^L(\mathbf{x}_i)$ is the network output when inputing a training example $\mathbf{x}_i$.

- $\mathbf{y}_i$ is the training label (coded by a vector). For example, in the case of MNIST handwritten digits, $\mathbf{y}_i$ takes one of the following vector values:

$$\text{digit } 0 = \begin{pmatrix} 1 \\ 0 \\ \vdots \\ 0 \end{pmatrix}, \text{digit } 1 = \begin{pmatrix} 0 \\ 1 \\ \vdots \\ 0 \end{pmatrix}, \ldots, \text{digit } 9 = \begin{pmatrix} 0 \\ 0 \\ \vdots \\ 1 \end{pmatrix}$$

The network loss has too many variables to be minimized analytically, so we'll use gradient descent to tackle the problem.

Gradient descent

For the specified network loss

$$C(\{\mathbf{W}^\ell, \mathbf{b}^\ell\}_{1\le\ell\le L}) = \frac{1}{2n}\sum_{i=1}^{n} \|\mathbf{a}^L(\mathbf{x}_i) - \mathbf{y}_i\|^2$$

computing all the partial derivatives $\frac{\partial C}{\partial w^\ell_{jk}}, \frac{\partial C}{\partial b^\ell_j}$ is highly nontrivial.

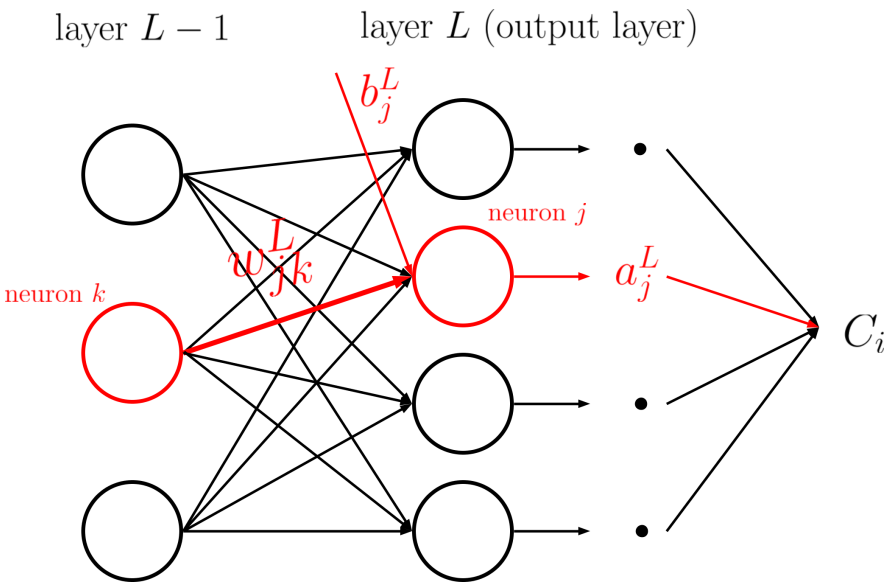
To simplify the task a bit, we consider a sample of size 1 consisting of only $\mathbf{x}_i$:

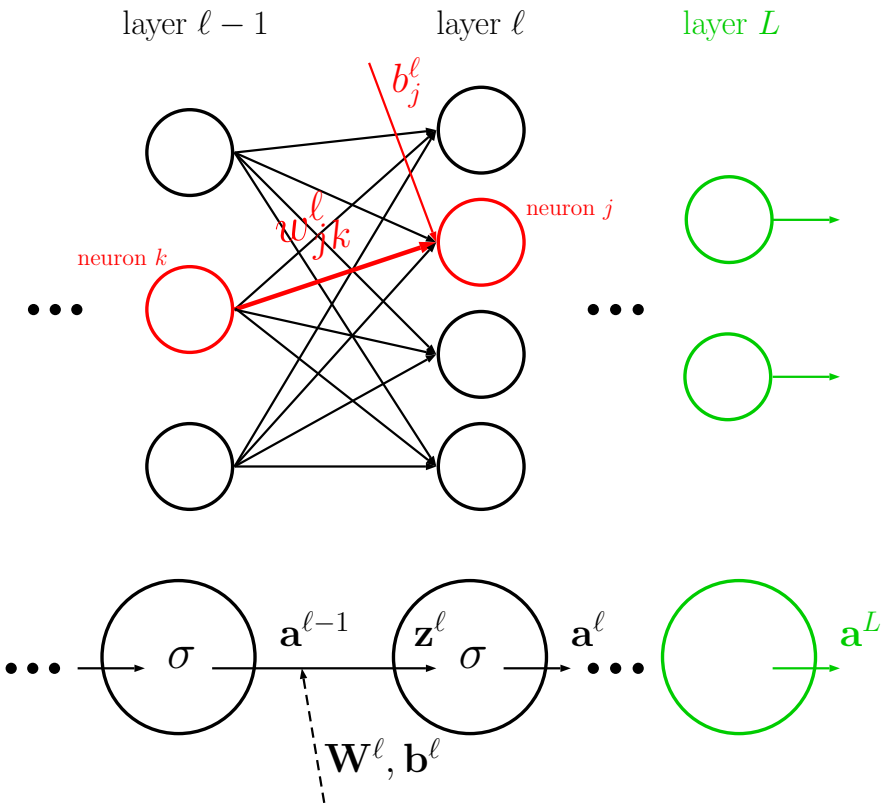
$$C_i(\{\mathbf{W}^\ell, \mathbf{b}^\ell\}_{1\le\ell\le L}) = \frac{1}{2}\|\mathbf{a}^L(\mathbf{x}_i) - \mathbf{y}_i\|^2 = \frac{1}{2}\sum_{j=1}^{c}(a^L_j - y_{ij})^2$$

which is enough as $\frac{\partial C}{\partial w^\ell_{jk}} = \frac{1}{n}\sum_i \frac{\partial C_i}{\partial w^\ell_{jk}}$ and $\frac{\partial C}{\partial b^\ell_j} = \frac{1}{n}\sum_i \frac{\partial C_i}{\partial b^\ell_j}$.

The output layer first

We start by computing $\frac{\partial C_i}{\partial w^L_{jk}}, \frac{\partial C_i}{\partial b^L_j}$ as they are the easiest.





Computing $\frac{\partial C_i}{\partial w_{jk}^L}, \frac{\partial C_i}{\partial b_j^L}$ for the output layer

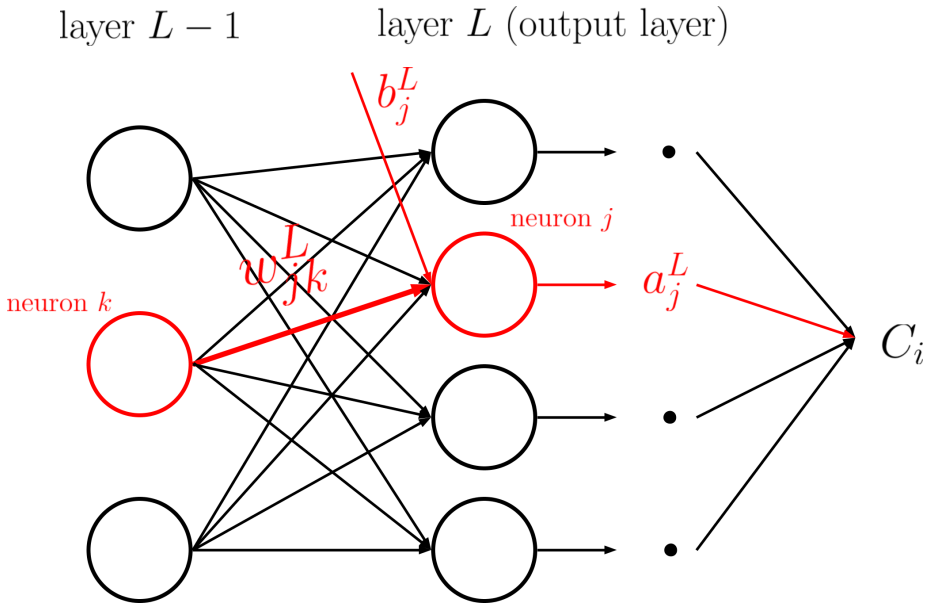
By chain rule we have

$$\frac{\partial C_i}{\partial w_{jk}^L} = \frac{\partial C_i}{\partial a_j^L} \cdot \frac{\partial a_j^L}{\partial w_{jk}^L}$$

where $\frac{\partial C_i}{\partial a_j^L} = a_j^L - y_{ij}$ for square loss and

$$\frac{\partial a_j^L}{\partial w_{jk}^L} = \frac{\partial a_j^L}{\partial z_j^L} \cdot \frac{\partial z_j^L}{\partial w_{jk}^L} = \sigma'(z_j^L) a_k^{L-1}$$

which is obtained by applying chain rule again with the formula for a_j^L.



$$a_j^L = \sigma\left(z_j^L\right), \; z_j^L = \sum_{k'} w_{jk'}^L a_{k'}^{L-1} + b_j^L$$

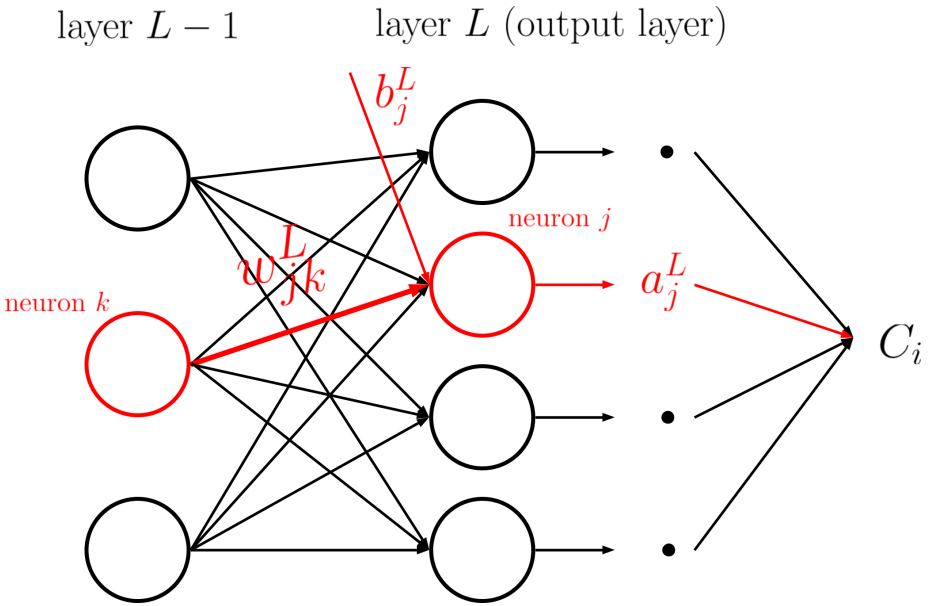
Computing $\frac{\partial C_i}{\partial w_{jk}^L}, \frac{\partial C_i}{\partial b_j^L}$ for the output layer

Combining results gives that

$$\begin{aligned}\frac{\partial C_i}{\partial w_{jk}^L} &= \frac{\partial C_i}{\partial a_j^L} \cdot \frac{\partial a_j^L}{\partial w_{jk}^L} \\ &= \left(a_j^L - y_{ij}\right) \sigma'(z_j^L) a_k^{L-1}.\end{aligned}$$

Similarly, we obtain that

$$\begin{aligned}\frac{\partial C_i}{\partial b_j^L} &= \frac{\partial C_i}{\partial a_j^L} \cdot \frac{\partial a_j^L}{\partial b_j^L} \\ &= \left(a_j^L - y_{ij}\right) \sigma'(z_j^L).\end{aligned}$$

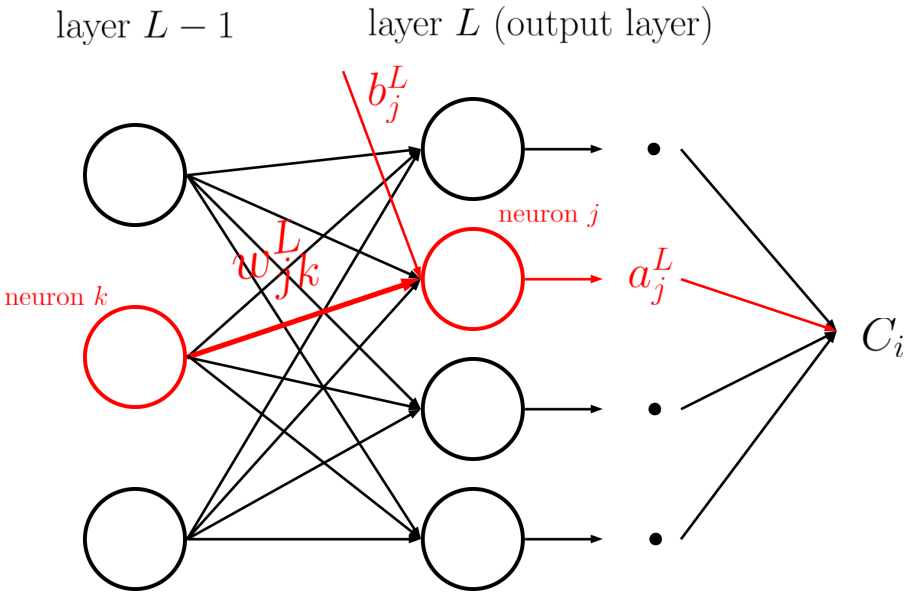


$$a_j^L = \sigma\left(z_j^L\right),\ z_j^L = \sum_{k'} w_{jk'}^L a_{k'}^{L-1} + b_j^L$$

Interpretation of the formula for $\frac{\partial C_i}{\partial w_{jk}^L}$

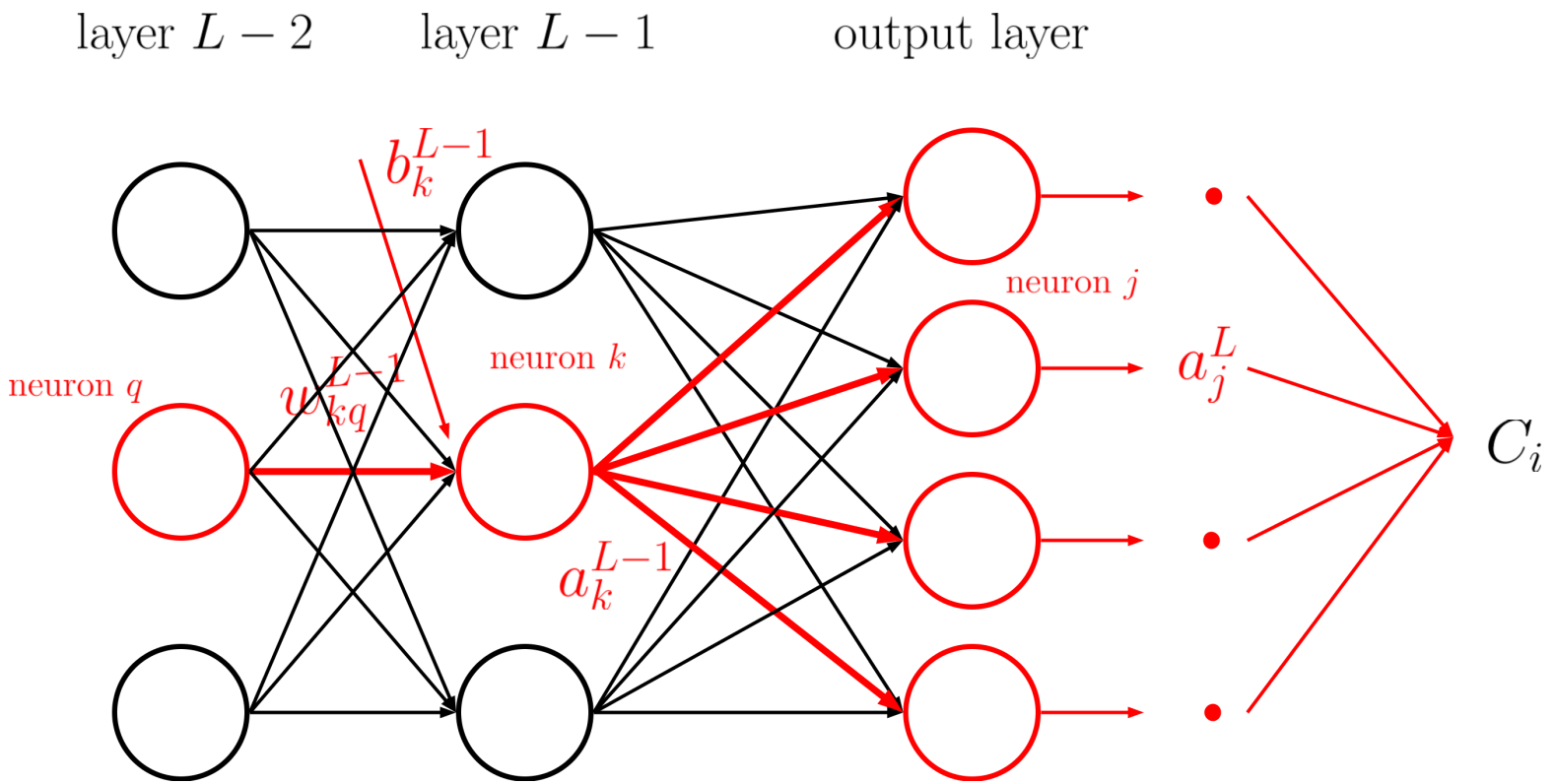
Observe that the rate of change of C_i w.r.t. w_{jk}^L depends on three factors ($\frac{\partial C_i}{\partial b_j^L}$ only depends on the first two):

- $a_j^L - y_{ij}$: how much current output is off from desired output
- $\sigma'(z_j^L)$: how fast the neuron reacts to changes of its input
- a_k^{L-1}: contribution from neuron k in layer $L-1$



Thus, w_{jk}^L will update slowly if the input neuron is in low-activation ($a_k^{L-1} \approx 0$), or the output neuron has saturated, i.e., $\sigma'(z_j^L) \approx 0$.

What about layer $L-1$ (and further inside)?

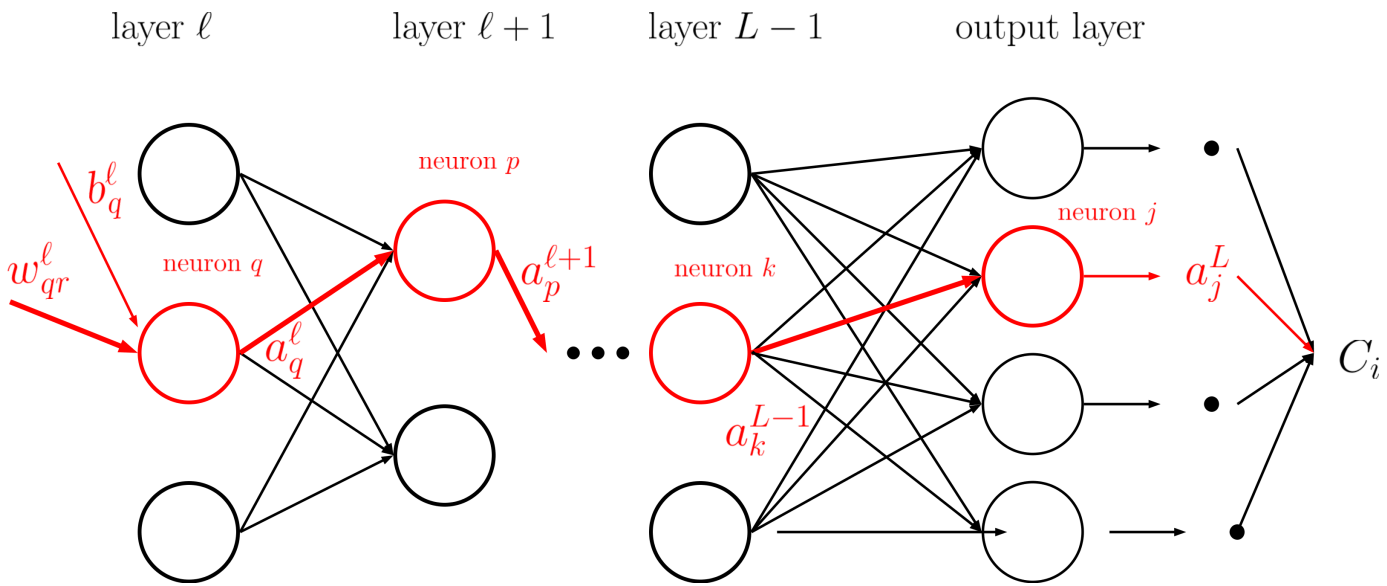


By chain rule,

$$\frac{\partial C_i}{\partial w_{kq}^{L-1}} = \sum_j \frac{\partial C_i}{\partial a_j^L} \frac{\partial a_j^L}{\partial w_{kq}^{L-1}} = \sum_j \frac{\partial C_i}{\partial a_j^L} \frac{\partial a_j^L}{\partial a_k^{L-1}} \frac{\partial a_k^{L-1}}{\partial w_{kq}^{L-1}}$$

where

- $\frac{\partial C_i}{\partial a_j^L} = a_j^L - y_{ij}$: current difference between output and target;
- $\frac{\partial a_j^L}{\partial a_k^{L-1}} = \sigma'(z_j^L) w_{jk}^L$: link between layers L and $L-1$;
- $\frac{\partial a_k^{L-1}}{\partial w_{kq}^{L-1}} = \sigma'(z_k^{L-1}) a_q^{L-2}$: Input from nuron q in layer $L-2$ and current response rate of neural k in layer $L-1$



As we move further inside the network (from the output layer), we will need to compute more and more links between layers:

$$\frac{\partial C_i}{\partial w_{qr}^\ell} = \sum_{p,\ldots,k,j} \frac{\partial a_q^\ell}{\partial w_{pq}^\ell} \frac{\partial a_p^{\ell+1}}{\partial a_q^\ell} \cdots \frac{\partial a_j^L}{\partial a_k^{L-1}} \frac{\partial C_i}{\partial a_j^L}$$

The backpropagation algorithm

The products of the link terms may be computed iteratively from right to left, leading to an efficient algorithm for computing all $\frac{\partial C_i}{\partial w_{jk}^{\ell}}, \frac{\partial C_i}{\partial b_j^{\ell}}$ (based on only $\mathbf{x}_i$):

- Feedforward $\mathbf{x}_i$ to obtain all neuron inputs and outputs:

$$\mathbf{a}^0 = \mathbf{x}_i; \quad \mathbf{a}^{\ell} = \sigma(\mathbf{W}^{\ell}\mathbf{a}^{\ell-1} + \mathbf{b}^{\ell}), \text{ for } \ell = 1, \ldots, L$$

- Backpropagate the network to compute

$$\frac{\partial a_j^L}{\partial a_q^{\ell}} = \sum_{p,\ldots,k} \frac{\partial a_p^{\ell+1}}{\partial a_q^{\ell}} \cdots \frac{\partial a_j^L}{\partial a_k^{L-1}}, \text{ for } \ell = L, \ldots, 1$$

The backpropagation algorithm (cont'd)

- Compute $\frac{\partial C_i}{\partial w_{qr}^{\ell}}, \frac{\partial C_i}{\partial b_q^{\ell}}$ for every layer ℓ and every neuron q or pair of neurons (q, r) by using

$$\frac{\partial C_i}{\partial w_{qr}^{\ell}} = \sum_j \frac{\partial a_q^{\ell}}{\partial w_{qr}^{\ell}} \cdot \frac{\partial a_j^L}{\partial a_q^{\ell}} \cdot \frac{\partial C_i}{\partial a_j^L}$$

$$\frac{\partial C_i}{\partial b_q^{\ell}} = \sum_j \frac{\partial a_q^{\ell}}{\partial b_q^{\ell}} \cdot \frac{\partial a_j^L}{\partial a_q^{\ell}} \cdot \frac{\partial C_i}{\partial a_j^L}$$

  Note that $\frac{\partial C_i}{\partial a_j^L}$ only needs to computed once.

Remark. The entire backpropagation process can be vectorized, in order to be implemented efficiently.

Stochastic gradient descent

- Initialize all the weights w_{jk}^{ℓ} and biases b_j^{ℓ};
- For each training example $\mathbf{x}_i$,
  - Use backpropagation to compute the partial derivatives $\frac{\partial C_i}{\partial w_{jk}^{\ell}}, \frac{\partial C_i}{\partial b_j^{\ell}}$
  - Update the weights and biases by:

$$w_{jk}^{\ell} \longleftarrow w_{jk}^{\ell} - \eta \cdot \frac{\partial C_i}{\partial w_{jk}^{\ell}}, \qquad b_j^{\ell} \longleftarrow b_j^{\ell} - \eta \cdot \frac{\partial C_i}{\partial b_j^{\ell}}$$

  This completes one epoch in the training process.
- Repeat the preceding step until convergence.

Remark. The previous procedure uses single-sample update rule (one training time each time). We can also use mini-batches $\{\mathbf{x}_i\}_{i\in B}$ to perform gradient descent (for faster speed):

- For every $i \in B$, use backpropagation to compute the partial derivatives $\frac{\partial C_i}{\partial w_{jk}^{\ell}}, \frac{\partial C_i}{\partial b_j^{\ell}}$
- Update the weights and biases by:

$$w_{jk}^{\ell} \longleftarrow w_{jk}^{\ell} - \eta \cdot \frac{1}{|B|} \sum_{i\in B} \frac{\partial C_i}{\partial w_{jk}^{\ell}},$$

$$b_j^{\ell} \longleftarrow b_j^{\ell} - \eta \cdot \frac{1}{|B|} \sum_{i\in B} \frac{\partial C_i}{\partial b_j^{\ell}}$$

Adaptive learning rates

Learning rates do not need to be fixed all the time: Initially, they can be set larger to speed up convergence and then gradually reduced:

```
opts = trainingOptions('sgdm', ...
    'InitialLearnRate', 0.005, ...
    'LearnRateSchedule', 'piecewise', ...
    'LearnRateDropFactor', 0.5, ...
    'LearnRateDropPeriod', 5);
```

For more detail, type 'help trainingOptions' in MATLAB.

Software for neural networks

MATLAB: Neural networks is not part of the MATLAB Statistics and Machine Learning Toolbox; there is a separate Deep Learning Toolbox.

Python: Nielson has written from scratch excellent Python codes exactly for MNIST digits classification, which is available at `https://github.com/mnielsen/neural-networks-and-deep-learning/archive/master.zip`.

Otherwise, you can directly use the Python MLP function available at `https://scikit-learn.org/stable/modules/generated/sklearn.neural_network.MLPClassifier.html`.

MATLAB demonstration

Nielson's Python codes for neural networks

```
# load MNIST data into python
import mnist_loader
training_data, validation_data, test_data = mnist_loader.
load_data_wrapper()

# define a 3-layer neural network with number of neurons on each layer
import network
net = network.Network([784, 30, 10])

# execute stochastic gradient descent over 30 epochs and with mini-batches
of size 10 and a learning rate of 3
net.SGD(training_data, 30, 10, 3.0, test_data=test_data)
```

Other activation functions

We have introduced Heaviside step and Sigmoid. Below are a few more:

- **Hyperbolic tangent**

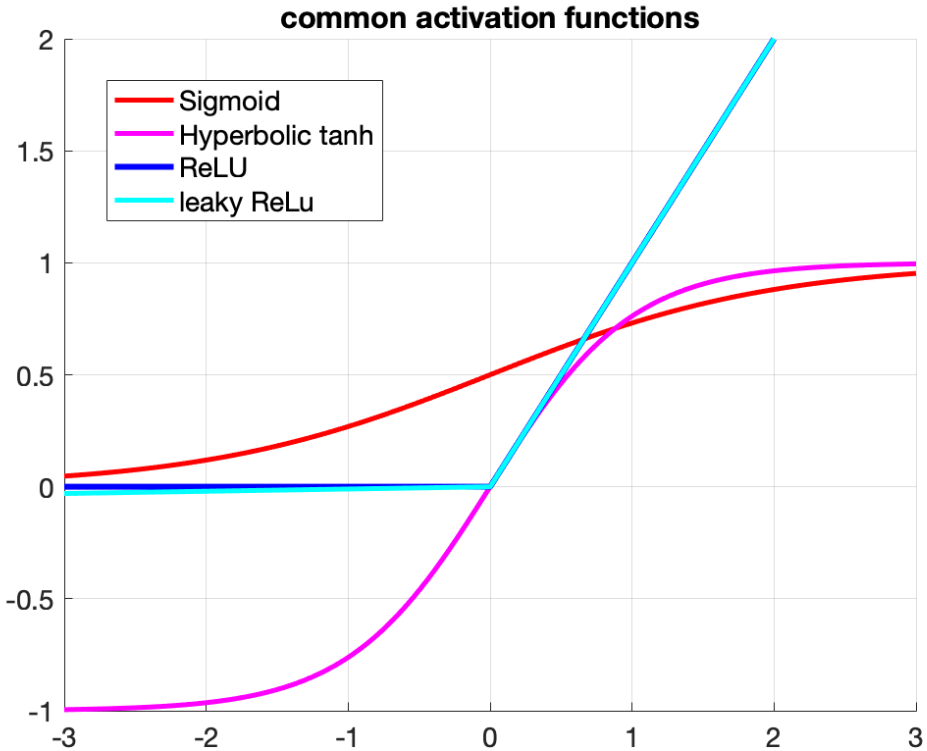
$$g(z) = \tanh(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}} = 2\,\text{sigmoid}(2z) - 1$$

- **Rectified Linear Unit (ReLU)** $\longleftarrow$ converges much faster than sigmoid

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

- **Leaky ReLU** (or parameterized ReLU if changing $0.01 \to a$)

$$g(z) = \begin{cases} 0.01z, & z < 0 \\ z, & z > 0 \end{cases}$$



More on activation functions:

`http://cs231n.stanford.edu/slides/2020/lecture_7.pdf`

Practical issues and techniques for improvement

We have covered the main ideas of neural networks. There are a lot of practical issues to consider:

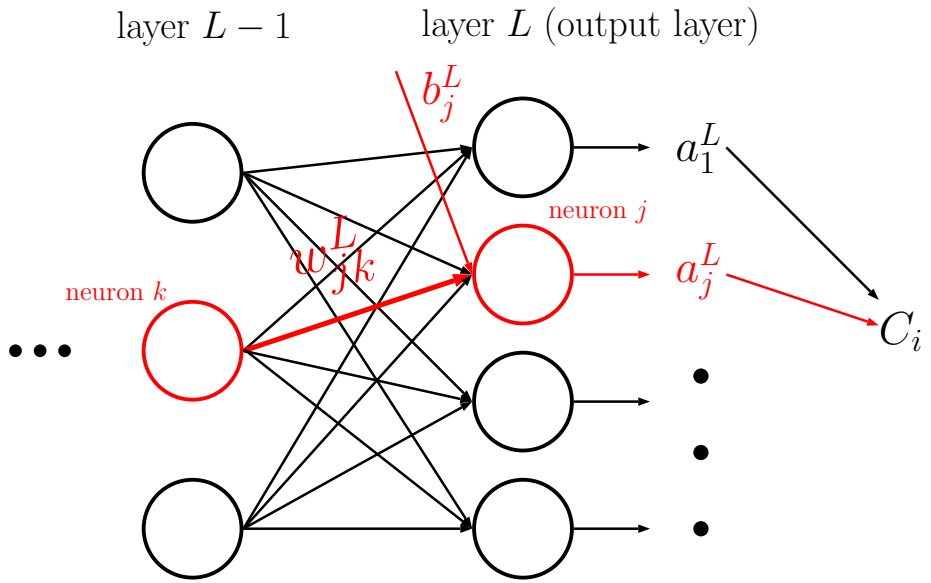
- How to fix learning slowdown
- How to avoid overfitting
- How to initialize the weights and biases for gradient descent
- How to choose the hyperparameters, such as the learning rate, regularization parameter, and configuration of the network, etc.

The learning slowdown issue

Learning slowdown may happen anywhere in a sigmoid neural network, when the combined input z_j^ℓ is on the wrong side of the sigmoid curve such that $\sigma'(z_j^\ell) \approx 0$.

In particular, it may happen on the output layer (when the square loss is used):

$$\begin{aligned}\frac{\partial C_i}{\partial w_{jk}^L} &= \frac{\partial C_i}{\partial a_j^L} \cdot \frac{\partial a_j^L}{\partial w_{jk}^L} \\ &= \left(a_j^L - y_{ij}\right) \sigma'(z_j^L) a_k^{L-1} \\ \frac{\partial C_i}{\partial b_j^L} &= \frac{\partial C_i}{\partial a_j^L} \cdot \frac{\partial a_j^L}{\partial b_j^L} \\ &= \left(a_j^L - y_{ij}\right) \sigma'(z_j^L).\end{aligned}$$



How to fix learning slowdown on the output layer

There are two simple ways to prevent learning slowdown from happening on the output layer (the task is much harder on the hidden layers).

First method: Use the logistic loss (also called the cross-entropy loss) instead:

$$C_i(\{\mathbf{W}^\ell, \mathbf{b}^\ell\}_{1\le\ell\le L}) = \sum_{j=1}^{c} -y_{ij}\log\left(a_j^L\right) - (1-y_{ij})\log\left(1-a_j^L\right)$$

With this loss, we can show that the $\sigma'(z_j^L)$ term is gone:

$$\frac{\partial C_i}{\partial w_{jk}^L} = \left(a_j^L - y_{ij}\right)a_k^{L-1}, \quad \frac{\partial C_i}{\partial b_j^L} = \left(a_j^L - y_{ij}\right)$$

so that gradient descent will move fast when a_j^L is on the opposite side to y_{ij}.

Detailed calculation:

$$\begin{aligned}
\frac{\partial C_i}{\partial w_{jk}^L} &= \frac{\partial C_i}{\partial a_j^L} \cdot \frac{\partial a_j^L}{\partial w_{jk}^L} \\
&= \left(-y_{ij} \frac{1}{a_j^L} - (1 - y_{ij}) \frac{-1}{1 - a_j^L} \right) \cdot \sigma'(z_j^L) a_k^{L-1} \\
&= \frac{a_j^L - y_{ij}}{a_j^L (1 - a_j^L)} \cdot \sigma'(z_j^L) a_k^{L-1} \\
&= \left(a_j^L - y_{ij} \right) a_k^{L-1}
\end{aligned}$$

$$\frac{\partial C_i}{\partial b_j^L} = \frac{\partial C_i}{\partial a_j^L} \cdot \frac{\partial a_j^L}{\partial b_j^L} = \frac{a_j^L - y_{ij}}{a_j^L (1 - a_j^L)} \cdot \sigma'(z_j^L) = \left(a_j^L - y_{ij} \right)$$

Second method: Use a "softmax output layer" with log-likelihood cost (see Nielson's book, Chapter 3):

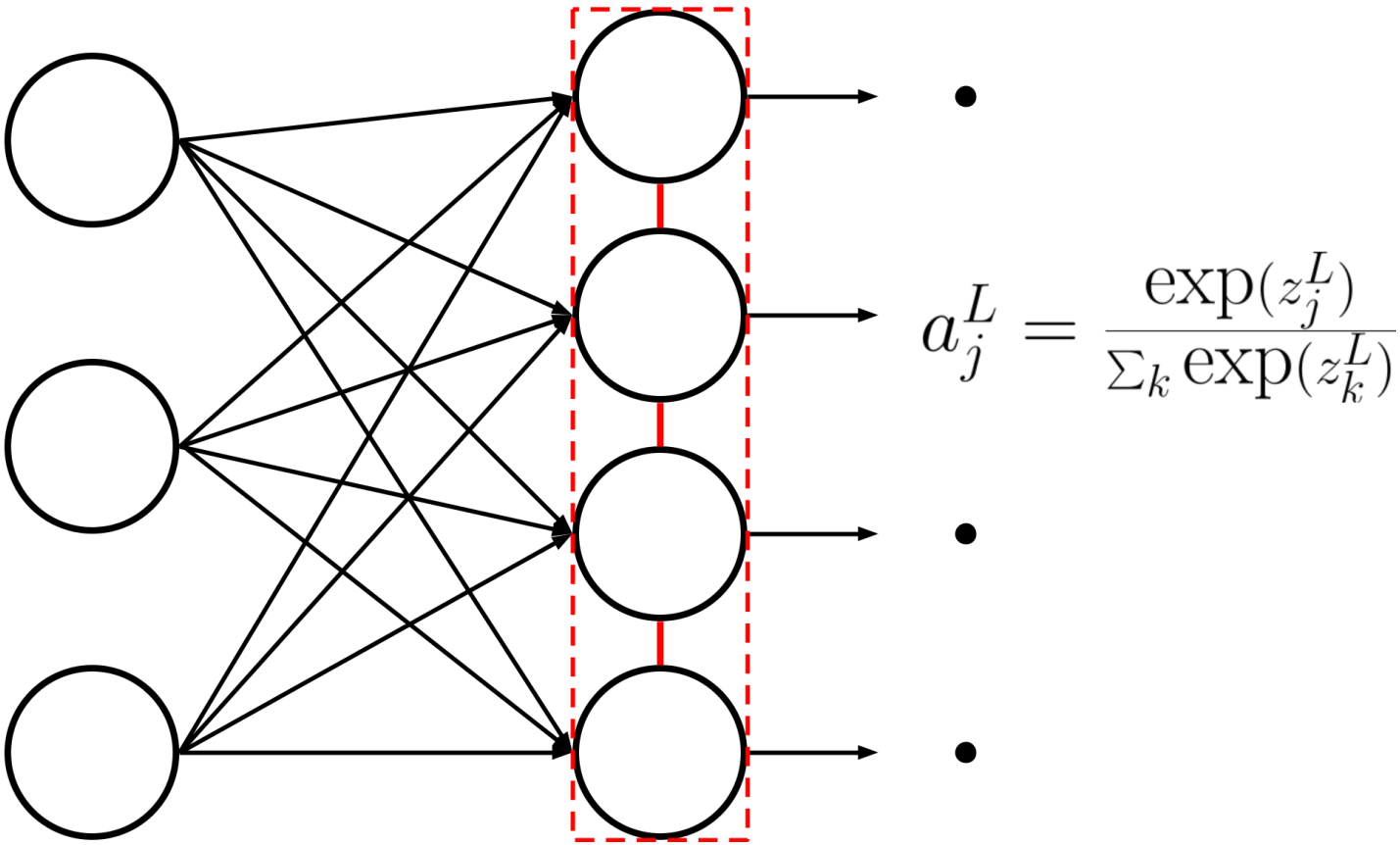
- Define a new type of output layer by changing the activation function from sigmoid to softmax (and letting all neurons share the total inputs $\{z_j^L\}$ with each other):

$$a_j^L = \text{softmax}(z_1^L, \ldots, z_c^L; j) = \frac{e^{z_j^L}}{\sum_k e^{z_k^L}}$$

- Use the log-likelihood cost (instead of logistic loss)

$$C = \sum_{i=1}^{n} C_i, \quad C_i = -\log a_{y_i}^L$$

layer $L-1$ layer L (softmax layer)



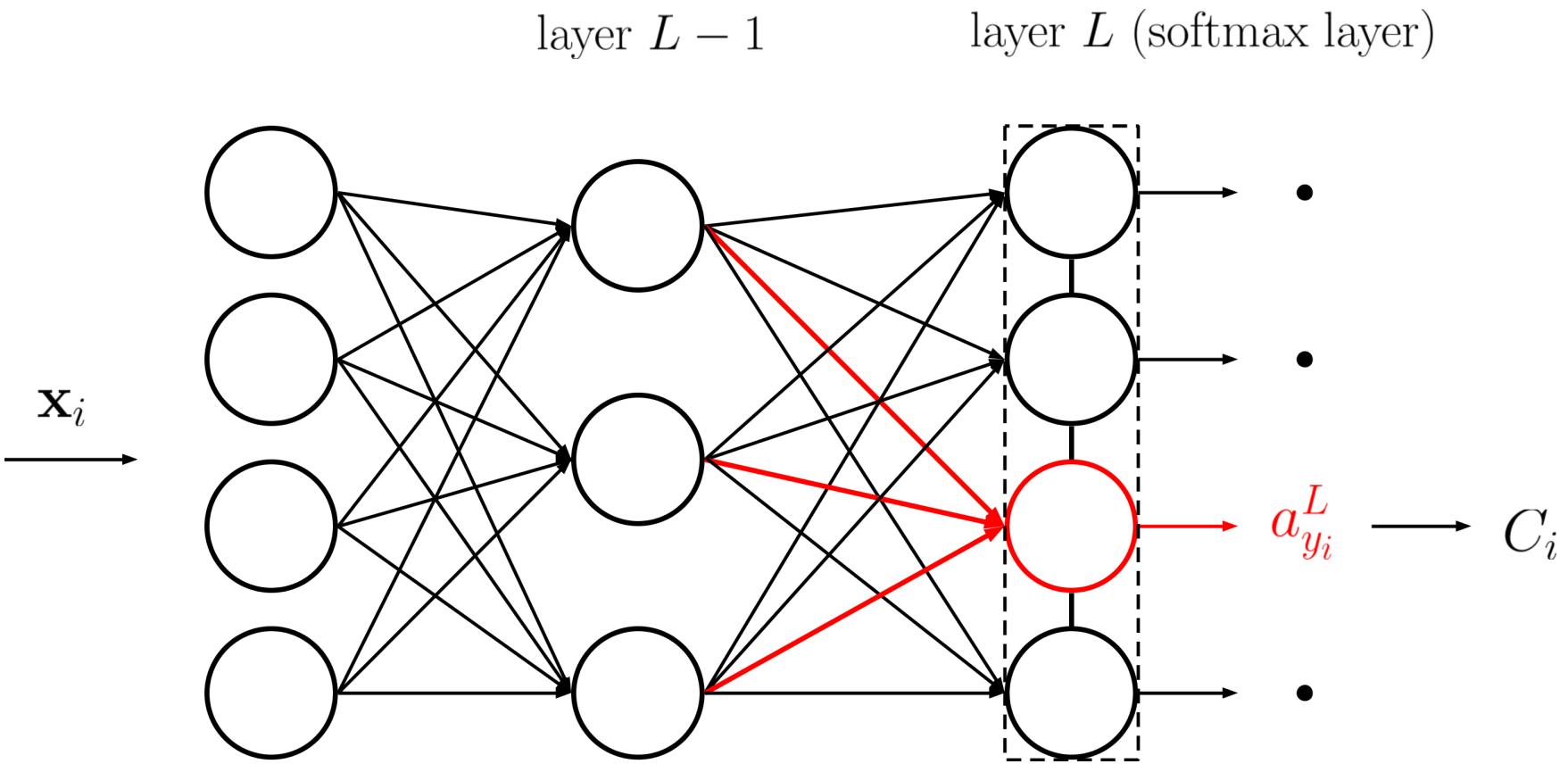
$$a_j^L = \frac{\exp(z_j^L)}{\Sigma_k \exp(z_k^L)}$$

We then have that

$$\frac{\partial C_i}{\partial w_{jk}^L} = \begin{cases} (a_j^L - 1)a_k^{L-1}, & j = y_i \\ a_j^L a_k^{L-1}, & j \neq y_i \end{cases}, \quad \text{and} \quad \frac{\partial C_i}{\partial b_j^L} = \begin{cases} a_j^L - 1, & j = y_i \\ a_j^L, & j \neq y_i \end{cases}$$

To verify these formulas, first we let $j = y_i$ and calculate $\frac{\partial C_i}{\partial w_{jk}^L}$:

$$\begin{aligned} \frac{\partial C_i}{\partial w_{jk}^L} &= -\frac{1}{a_{y_i}^L} \cdot \frac{\partial a_{y_i}^L}{\partial z_j^L} \cdot \frac{\partial z_j^L}{\partial w_{jk}^L} \\ &= -\frac{1}{a_{y_i}^L} \cdot \frac{e^{z_j^L} \sum_k e^{z_k^L} - e^{z_j^L} e^{z_j^L}}{\left(\sum_k e^{z_k^L}\right)^2} \cdot a_k^{L-1} \\ &= -\frac{1}{a_{y_i}^L} \cdot a_j^L (1 - a_j^L) \cdot a_k^{L-1} \\ &= (a_j^L - 1) a_k^{L-1} \end{aligned}$$


layer $L-1$
layer L (softmax layer)
$\mathbf{x}_i$
$a^L_{y_i}$
C_i

Now, we suppose $j \neq y_i$ and calculate $\frac{\partial C_i}{\partial w^L_{jk}}$ again:

$$\begin{aligned}
\frac{\partial C_i}{\partial w^L_{jk}} &= -\frac{1}{a^L_{y_i}} \cdot \frac{\partial a^L_{y_i}}{\partial z^L_j} \cdot \frac{\partial z^L_j}{\partial w^L_{jk}} \\
&= -\frac{1}{a^L_{y_i}} \cdot \frac{-e^{z^L_{y_i}} e^{z^L_j}}{\left(\sum_k e^{z^L_k}\right)^2} \cdot a^{L-1}_k \\
&= -\frac{1}{a^L_{y_i}} \cdot (-a^L_{y_i} a^L_j) \cdot a^{L-1}_k \\
&= a^L_j a^{L-1}_k
\end{aligned}$$

The formulas for $\frac{\partial C_i}{\partial b^L_j}$ in both cases can be verified by replacing $\frac{\partial z^L_j}{\partial w^L_{jk}}$ with $\frac{\partial z^L_j}{\partial b^L_j} = 1$.

In general, both techniques

- a sigmoid output layer and cross-entropy, or
- a softmax output layer and log-likelihood

work similarly well.

One advantage of the softmax layer is the interpretation of its outputs a_j^L as probabilities:

$$a_j^L > 0, \quad \sum_j a_j^L = 1$$

Nielson's implementation for neural networks with cross-entropy loss

```
# define a 3-layer neural network with cross-entropy cost
import network2
net = network2.Network([784, 30, 10],
cost=network2.CrossEntropyCost)

# stochastic gradient descent
net.large_weight_initializer()
net.SGD(training_data, 30, 10, 0.5, evaluation_data=test_data,
monitor_evaluation_accuracy=True)
```

How to avoid overfitting

Neural networks due to their many parameters are likely to overfit especially when given insufficient training data.

Like regularized logistic regression, we can add a regularization term of the form

$$\frac{\lambda}{2n}\sum_{j,k,\ell}|w_{jk}^{\ell}|^{p}$$

to any cost function used in order to avoid overfitting.

Typical choices of p are

- $p=2$ (L_2-regularization) and
- $p=1$ (L_1-regularization)

When L_2-regularization is used, the single-sample based gradient descent update rule for the weights is

$$w_{jk}^{\ell} \longleftarrow w_{jk}^{\ell} - \eta \left(\frac{\partial C_i}{\partial w_{jk}^{\ell}} + \frac{\lambda}{n} w_{jk}^{\ell} \right) = \left(1 - \frac{\eta\lambda}{n} \right) w_{jk}^{\ell} - \eta \cdot \frac{\partial C_i}{\partial w_{jk}^{\ell}}$$

Notice the weight decay due to the factor $1 - \frac{\eta\lambda}{n}$.

For mini-batch gradient descent, the update rule for the weights is similar:

$$w_{jk}^{\ell} \longleftarrow \left(1 - \frac{\eta\lambda}{n} \right) w_{jk}^{\ell} - \eta \cdot \frac{1}{|B|} \sum_{i \in B} \frac{\partial C_i}{\partial w_{jk}^{\ell}}$$

For either version of gradient descent, the update rule for the biases b_j^{ℓ} remains unchanged.

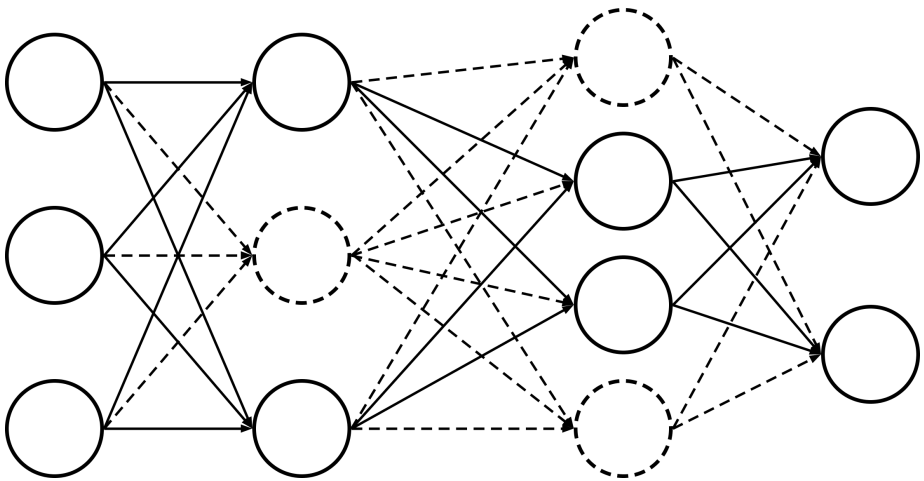
Nielson's implementation for regularized neural networks

```
# define a 3-layer neural network with cross-entropy cost
import network2
net = network2.Network([784, 30, 10], cost=network2.CrossEntropyCost)

# stochastic gradient descent
net.large_weight_initializer()
net.SGD(training_data, 30, 10, 0.5, evaluation_data=test_data, lmbda=5.0, monitor_evaluation_accuracy=True, monitor_training_accuracy=True)
```

Remark. Two more techniques to deal with overfitting are (see Nielson's book, Chapter 3)

- **artificial expansion of training data**, and
- **dropout**: Randomly and temporarily delete half of the hidden neurons (and their connections in the network) in each training iteration.



How to initialize weights and biases

The biases b_j^ℓ for all neurons are initialized as standard Gaussian random variables.

Regarding weight initialization, a first (and naive) idea is to initialize all weights $\{w_{jk}^\ell\}$ as standard Gaussian random variables.

For neuron j on layer ℓ, letting n_{in} be the number of inputs to this neuron,

$$z_j^\ell = \sum_{k=1}^{n_{\text{in}}} w_{jk}^\ell a_k^{\ell-1} + b_j^\ell \quad \sim \quad N\left(0,\, 1 + \sum_{k=1}^{n_{\text{in}}} \left(a_k^{\ell-1}\right)^2\right)$$

In particular, on the first hidden layer ($\ell = 1$),

$$z_j^1 = \sum_{k=1}^{d} w_{jk}^1 x_k + b_j^1 \quad \sim \quad N\left(0,\, 1 + \sum_{k=1}^{d} x_k^2\right)$$

Because of the possibly very large variance, this technique has the risk of forcing the neuron to start from very wrong positions.

A better idea is to initialize, for each neuron, the input weights as Gaussian random variables with mean 0 and standard deviation $1/\sqrt{n_{\text{in}}}$.

This implies that

$$z_j^{\ell} \sim N\left(0,\, 1 + \frac{1}{n_{\text{in}}} \sum_{k=1}^{n_{\text{in}}} \left(a_k^{\ell-1}\right)^2\right)$$

Such an initialization lets the neuron start in the middle of the activation curve, and thus will converge faster (see Nielson's book, Chapter 3).

Current standard weight initializations[3]

- Sigmoid and Tanh activation functions: "glorot" or "xavier" initialization

$$w_{jk}^{\ell} \sim \text{Unif}\left(-\frac{1}{\sqrt{n_{\text{in}}}}, \frac{1}{\sqrt{n_{\text{in}}}}\right)$$

- ReLU: "He" weight initialization

$$w_{jk}^{\ell} \sim N\left(0, \sqrt{\frac{2}{n_{\text{in}}}}\right)$$

Both have been implemented in MATLAB. Type "help fullyConnectedLayer" to see the option 'WeightsInitializer'.

[3] https://machinelearningmastery.com/weight-initialization-for-deep-learning-neural-networks/

Python codes for neural networks with better initialization

```
# define a 3-layer neural network with cross-entropy cost
import network2
net = network2.Network([784, 30, 10], cost=network2.CrossEntropyCost)

# stochastic gradient descent
net.large_weight_initializer()
net.SGD(training_data, 30, 10, 0.5, evaluation_data=test_data, lmbda=5.0, monitor_evaluation_accuracy=True, monitor_training_accuracy=True)
```

How to set the hyper-parameters

Parameter tuning for neural networks is hard and requires specialist knowledge.

- **Rules of thumb**: Start with subsets of data and small networks, e.g.,
  - consider only two classes (digits 0 and 1)
  - train a (784,10) network first, and then sth like (784, 30, 10) later
  - monitor the validation accuracy more often, say, after every 1,000 training images
  - stop early when the accuracy has saturated

  and play with the parameters to get quick feedback from experiments.

Once things get improved, vary each hyperparameter separately (while fixing the rest) until the result stops improving (though this may only give you a locally optimal combination).

- **Automated approaches**:
  - Grid search
  - Bayesian optimization

  See the references given in Nielson's book (Chapter 3).

Finally, remember that "the space of hyper-parameters is so large that one never really finishes optimizing, one only abandons the network to posterity."

Summary

- Presented what neural networks are and how to train them
  - Backpropagation
  - Gradient descent
  - Practical considerations
- Neural networks are new, flexible and powerful
- Neural networks are also an art to master

Next time: Introduction to deep learning

Main resources: *MIT 6.S191 Introduction to Deep Learning*[4]

- Lecture 1: Introduction to deep learning[5]
- Lecture 3: Deep computer vision[6]

[4] http://introtodeeplearning.com
[5] http://introtodeeplearning.com/slides/6S191_MIT_DeepLearning_L1.pdf
[6] http://introtodeeplearning.com/slides/6S191_MIT_DeepLearning_L3.pdf

Assignment 6

Question 1. Use gradient descent to find the local minimum of the function $f(x) = \frac{1}{4}x^4 - 3x^3 + 7x - 5$ around $x = 0$. Try three different learning rates (0.01, 0.05, 0.1) and plot the convergence of the function values in at least 30 iterations. What is the local minimum value and which learning rate is the best?

Question 2. Train a shallow neural network containing a single hidden layer of 100 or more neurons on the Fashion-MNIST training data by using the following options:

- logistic loss
- stochastic gradient descent (with mini-batch size of 300 or your own choice)

- the best learning rate (out of at least 5 different values such as 0.0001, 0.0005, 0.001, 0.005, 0.01)

and use it to classify the test data. What kind of accuracy did you achieve, and how long did it take to train the network (with the best learning rate you found)?

Question 3 (open exploration). Try a combination of new things or techniques, such as a new activation function like ReLu, more neurons and/or layers, and regularization/drop out to improve the test accuracy obtained in Question 2 by at least 0.5%.