LEC 8: Artificial Neural Networks (ANN)

Guangliang Chen

November 26, 2018

Outline

- Overview
  - What is a neural network
  - What is a neuron
- Perceptrons
- Sigmoid neurons network
- Summary

Acknowledgments

This presentation is based on the following references:

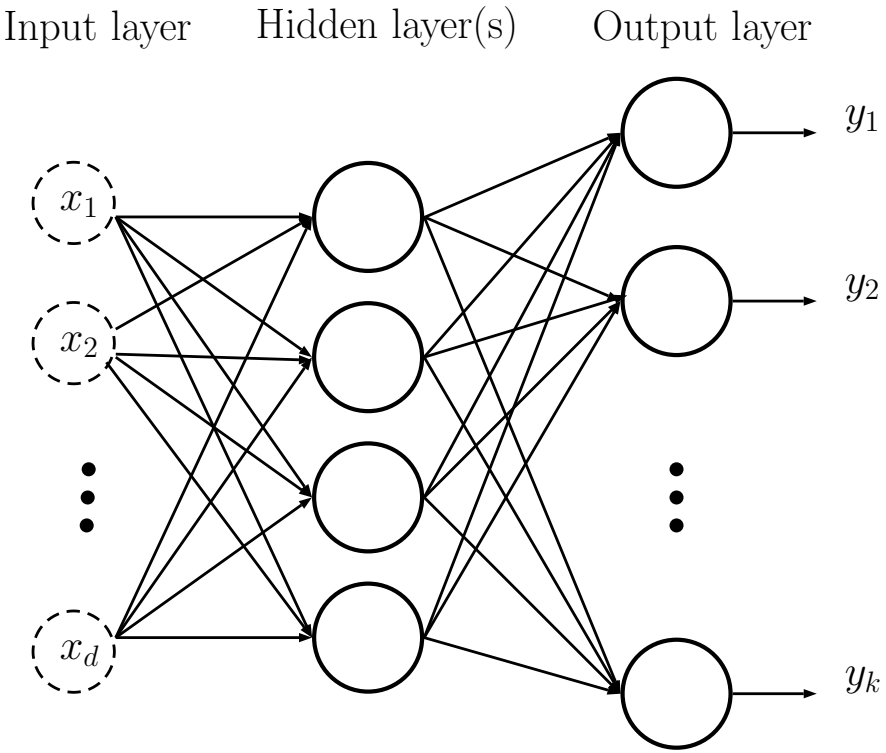
- **Michael Nielsen's book "Neural Networks and Deep Learning"** at

  `http://neuralnetworksanddeeplearning.com`

- **Olga Veksler's lecture on neural networks** at

  `http://www.csd.uwo.ca/courses/CS9840a/Lecture10_NeuralNets.pdf`

What is an artificial neural network?



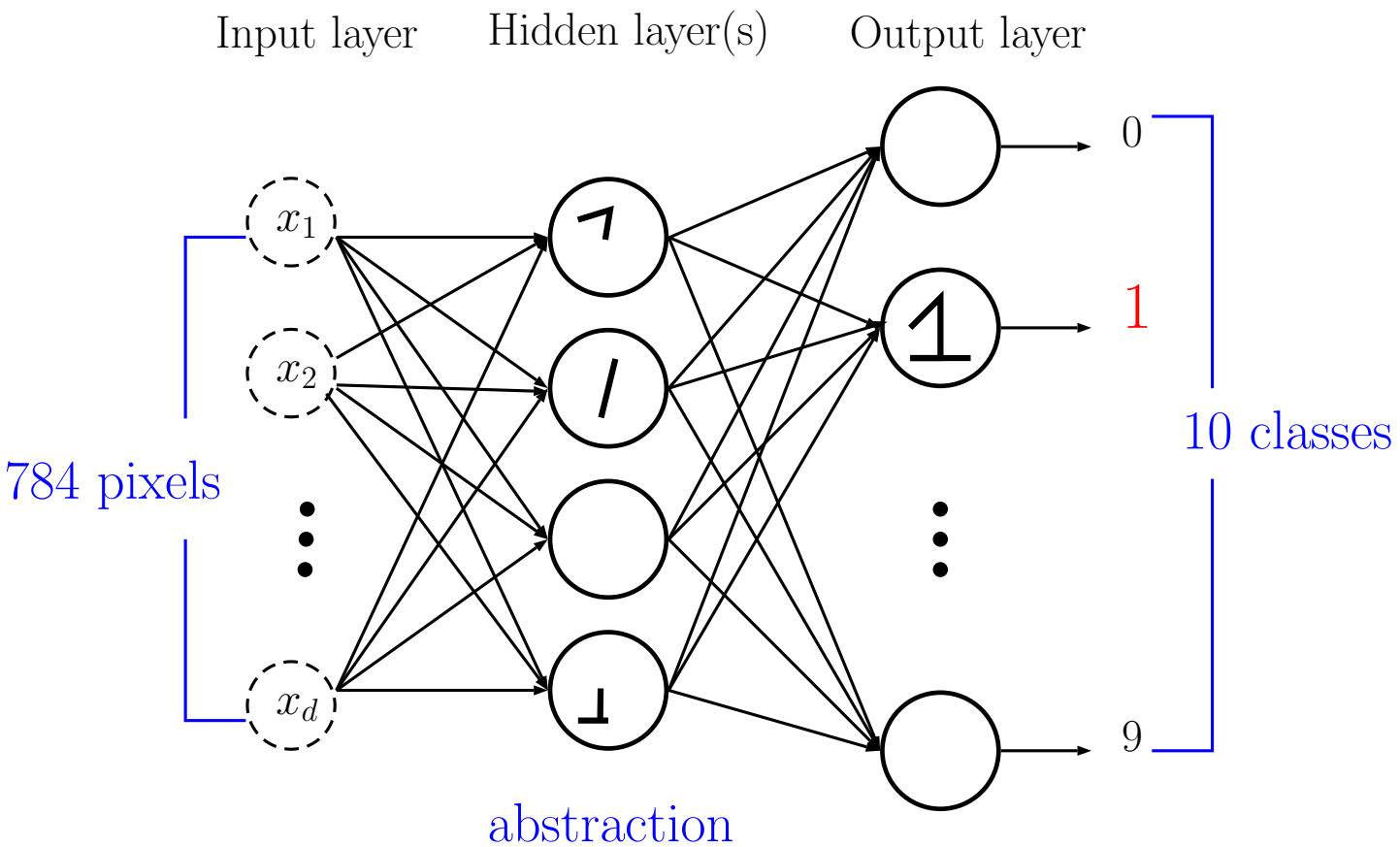
The leftmost layer inputs features x_i.

The rightmost layer outputs approximations to predictions y_i.

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

The network may have > 1 hidden layer (**shallow/deep** network).

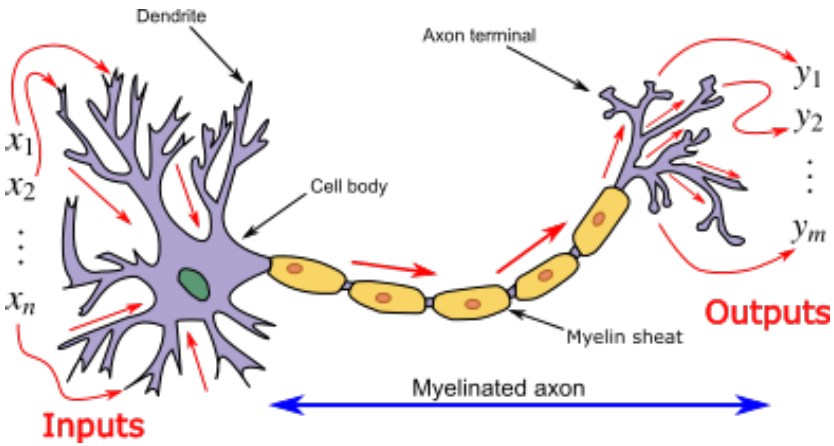
ANN for MNIST handwritten digits recognition



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:

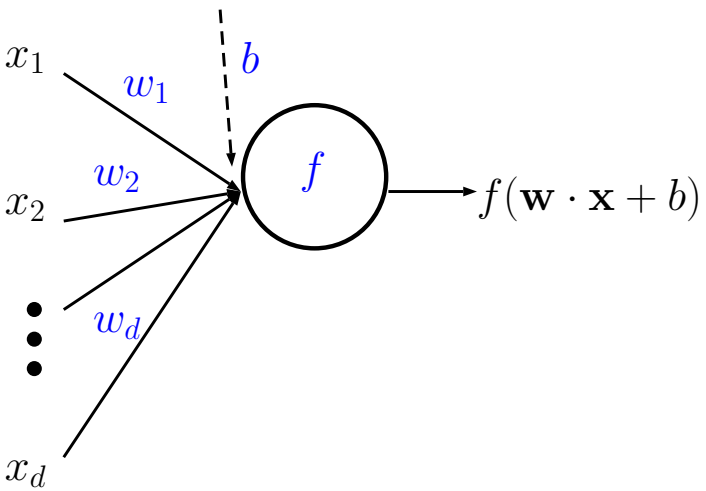
- "input wires", receive inputs from other neurons
- a neuron may have thousands of dendrites, usually short

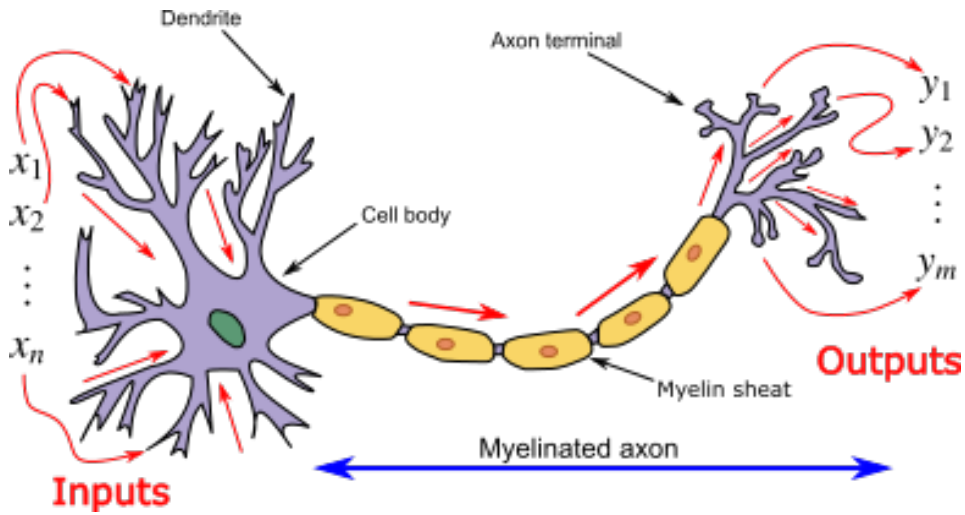
cell body: computational unit

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



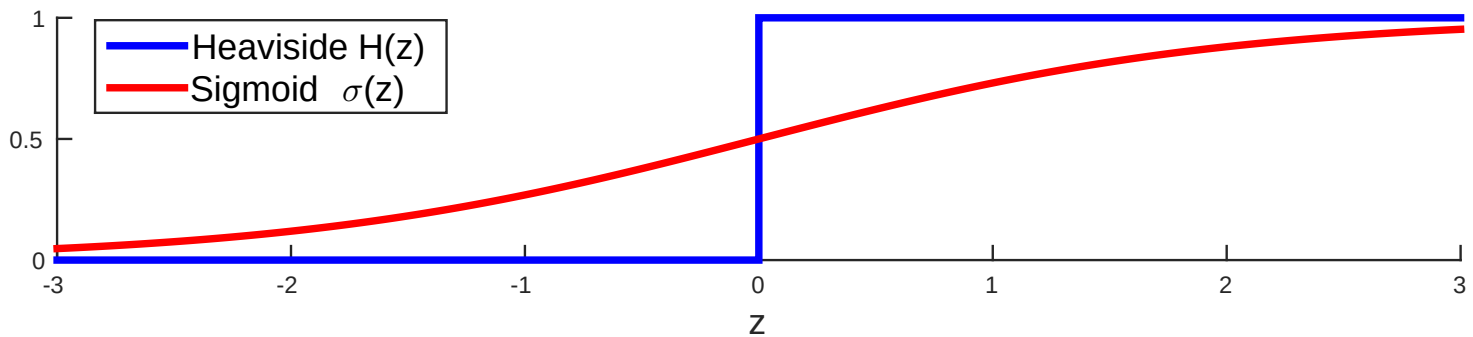


In the above,

- 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, resp.

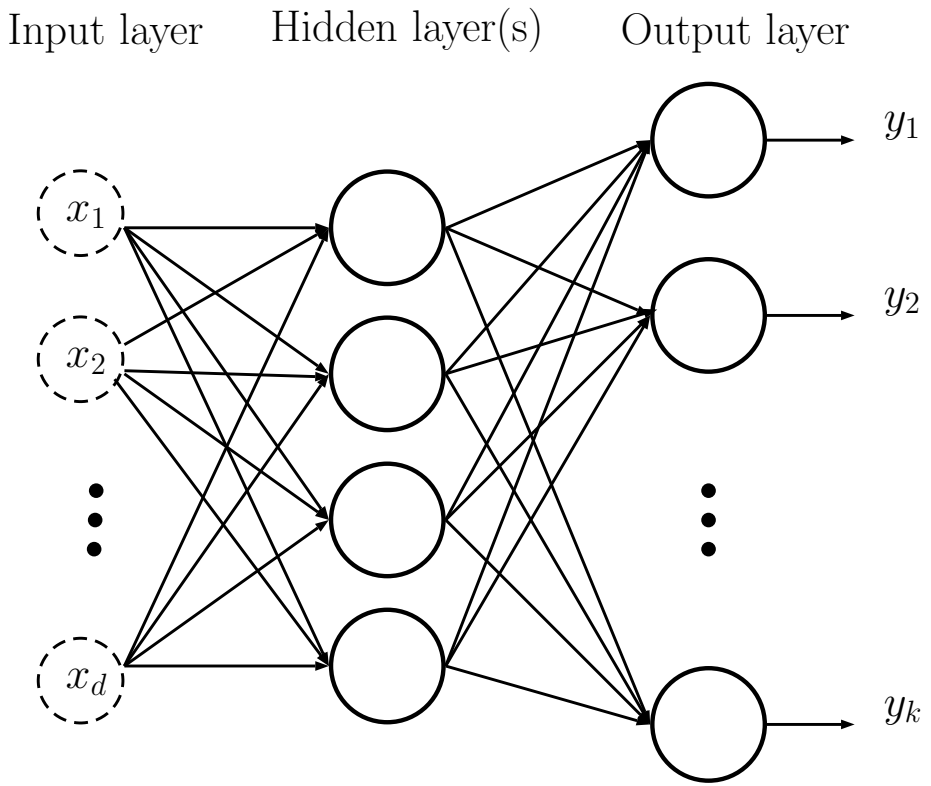
We will mention several other activation functions in the end.

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 more of theoretical than practical interest.

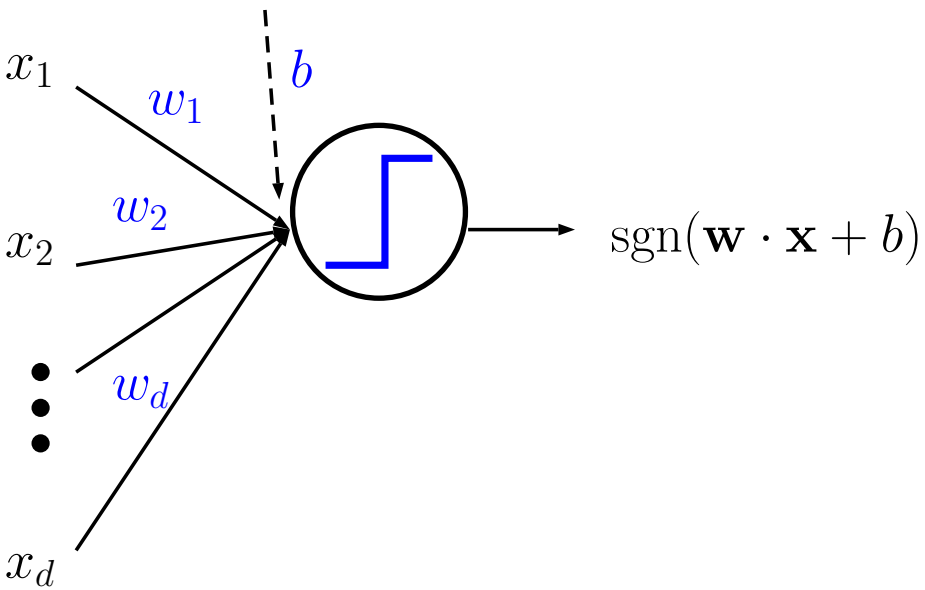


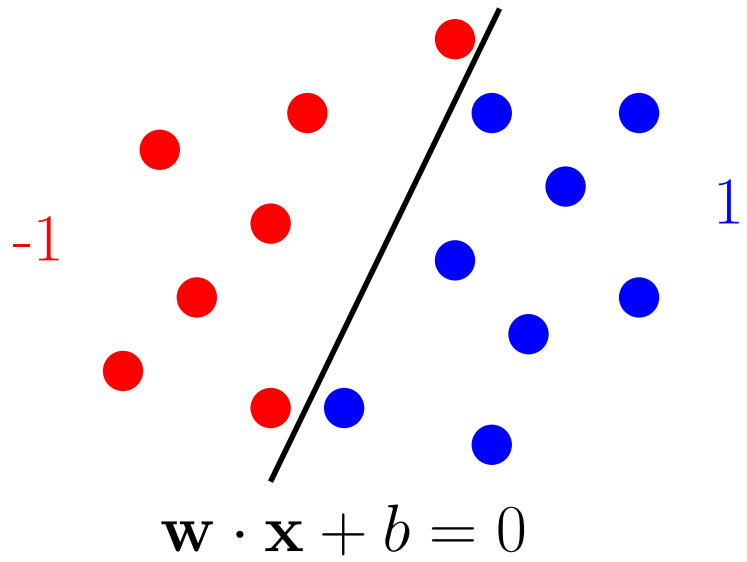
How to train ANNs in principle

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

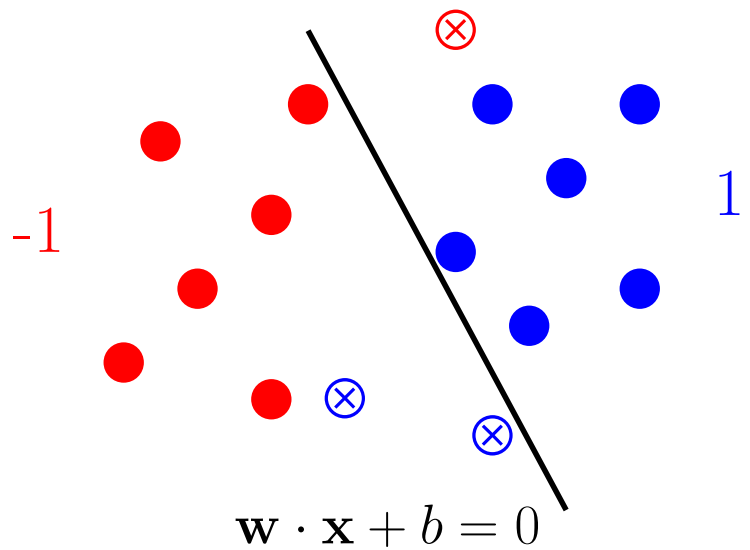
A perceptron is a neuron whose activation function is the Heaviside step function. It defines a linear, binary classifier (not necessarily optimal).





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 $\mathcal{M}$.
- The goal is to minimize the total loss

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

- 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:

- **Initialize** weights $\mathbf{w}$ and bias b (which would determine $\mathcal{M}$).
- **Iterate** until stopping criterion is met

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

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

$$\frac{\partial \ell}{\partial b} = -\sum_{i \in \mathcal{M}} y_i$$

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

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

$$b \longleftarrow b + \rho \sum_{i \in \mathcal{M}} y_i$$

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

Interpretation:

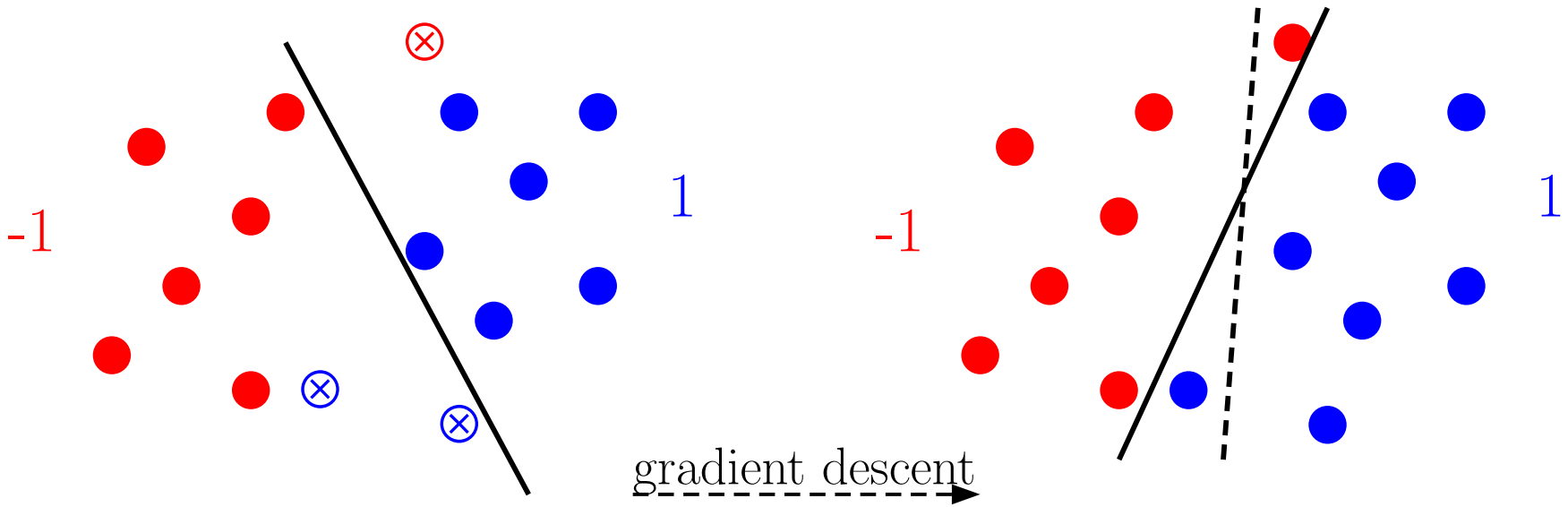
* Since $\sum_{i\in\mathcal{M}} y_i > 0$ (<0) if there are more errors in the positive (negative) class, b will be modified to fix the errors in the corresponding class.

* For any $j \in \mathcal{M}$,

$$\mathbf{w}\cdot\mathbf{x}_j \longleftarrow \mathbf{w}\cdot\mathbf{x}_j + \rho y_j \|\mathbf{x}_j\|_2^2 + \sum_{i\in\mathcal{M}-\{j\}} y_i(\mathbf{x}_i\cdot\mathbf{x}_j)$$

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

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



(Computer demonstration)

How to set the learning rate ρ

- Can adjust ρ at the training time
- The loss function $\ell(\mathbf{w}, b)$ should decrease during gradient descent
  - If $\ell(\mathbf{w}, b)$ oscillates: ρ is too large, decrease it
  - If $\ell(\mathbf{w}, b)$ goes down but very slowly: ρ is too small, increase it

Stochastic gradient descent

The gradient descent approach we use so far assumes that we have access to the full training set, and uses all training data to iteratively update weights and bias.

This may be slow for large data sets, or impractical in the setting of online learning (where data comes sequentially).

A variant of gradient descent, called **stochastic gradient descent**, uses

- only a single training point, or
- a small subset of examples (called mini-batch),

each round to update weights and bias.

- **Single-sample** update rule:
  - Start with a random hyperplane (with corresponding $\mathbf{w}$ and b)
  - Randomly select a new point $\mathbf{x}_i$ from the training set: if it lies on the correct side, no change; otherwise update

$$\mathbf{w} \longleftarrow \mathbf{w} + \rho y_i \mathbf{x}_i$$
$$b \longleftarrow b + \rho y_i$$

  - Repeat until all examples have been visited (this is called an **epoch**)

- **Mini-batch** update rule:
  - Divide training data into mini-batches (of size 5, or 10), and update weights after processing each mini-batch

$$\mathbf{w} \longleftarrow \mathbf{w} + \sum \rho y_i \mathbf{x}_i$$
$$b \longleftarrow b + \rho \sum y_i$$

  - Middle ground between single sample and full training set
  - One iteration over all mini-batches is called an epoch

Comments on stochastic gradient descent

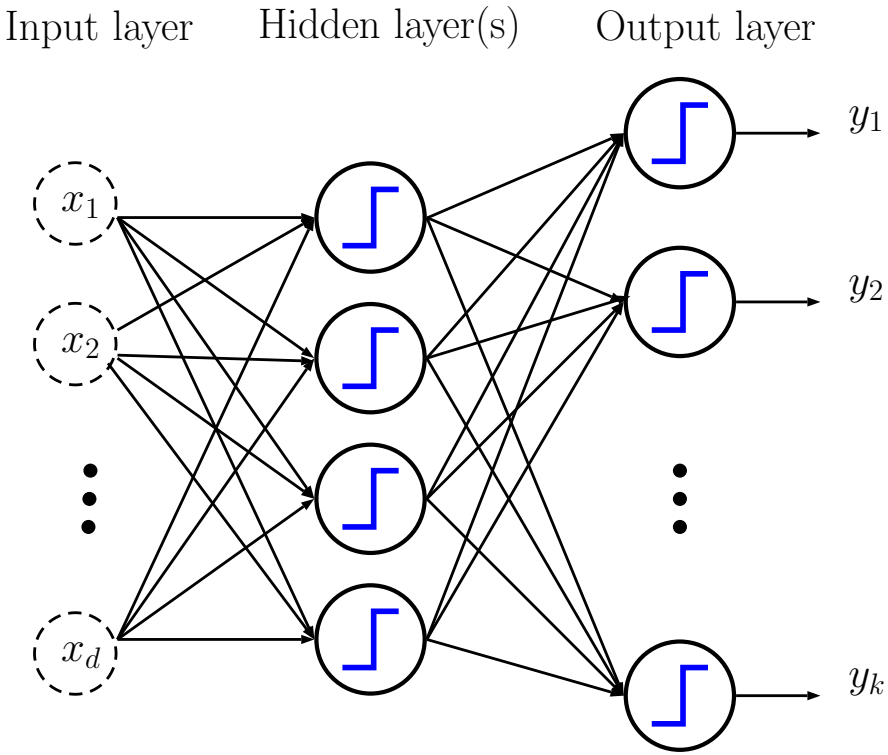
- Single-sample update rule applies to online learning
- Faster than full gradient descent, but maybe less stable
- Batch update rule might achieve some balance between speed and stability
- May find only a local minimum (the hyperplane is trapped in a suboptimal location)

(Click here to see pictures)

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

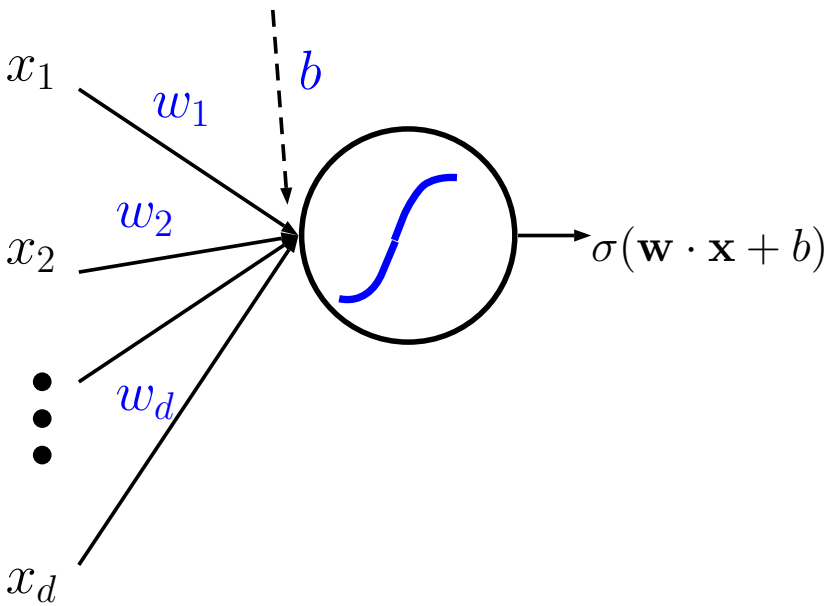
Sigmoid neurons are smoothed-out (or soft) versions of the perceptrons:

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

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

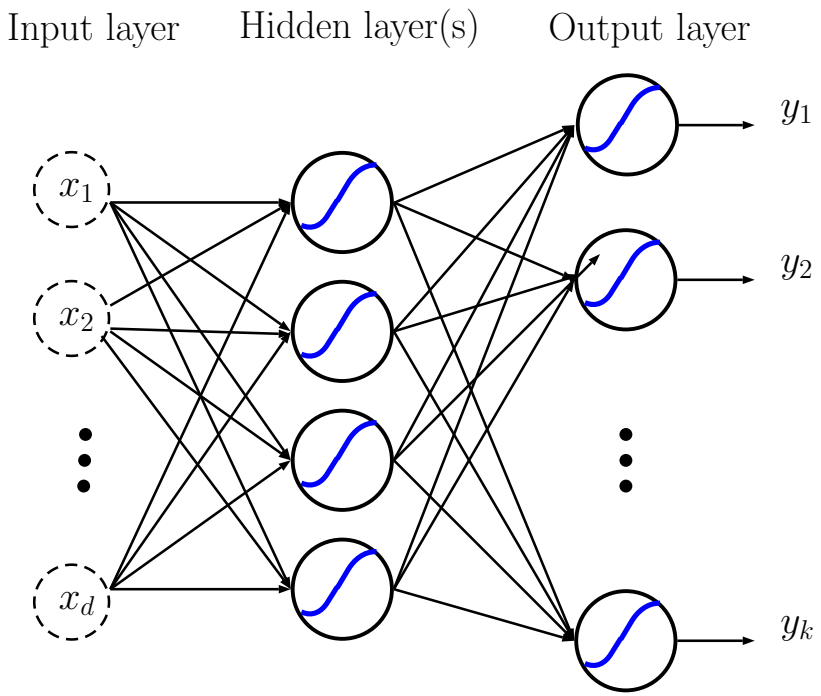
We say the neuron is in low (high) activation if the output is near 0 (1).

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

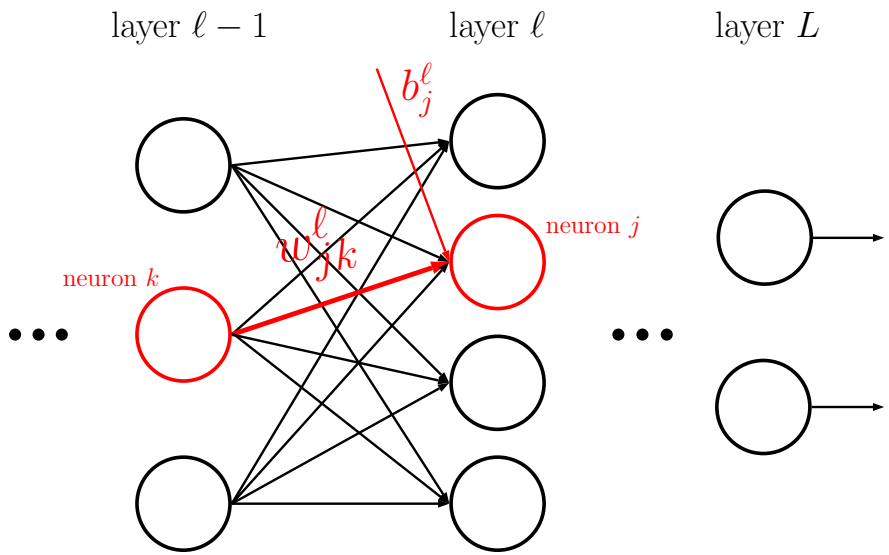
Notation

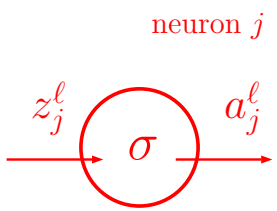
w_{jk}^{ℓ}: layer ℓ, "j back to k" weight;

b_j^{ℓ}: layer ℓ, neuron j bias

a_j^{ℓ}: layer ℓ, neuron j output

$z_j^{\ell} = \sum_k w_{jk}^{\ell} a_k^{\ell-1} + b_j^{\ell}$: weighted input to neuron j in layer ℓ





Note that $a_j^{\ell} = \sigma(z_j^{\ell})$.

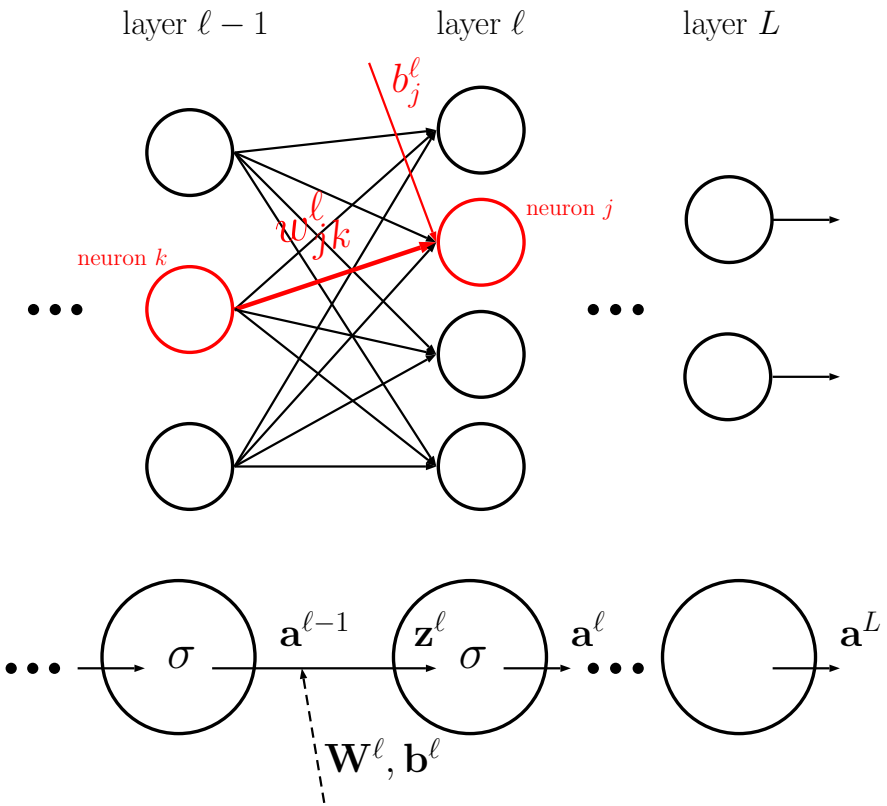
Notation (vector form)

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

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

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

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



We write $\mathbf{a}^\ell = \sigma\left(\mathbf{z}^\ell\right)$ (componentwise).

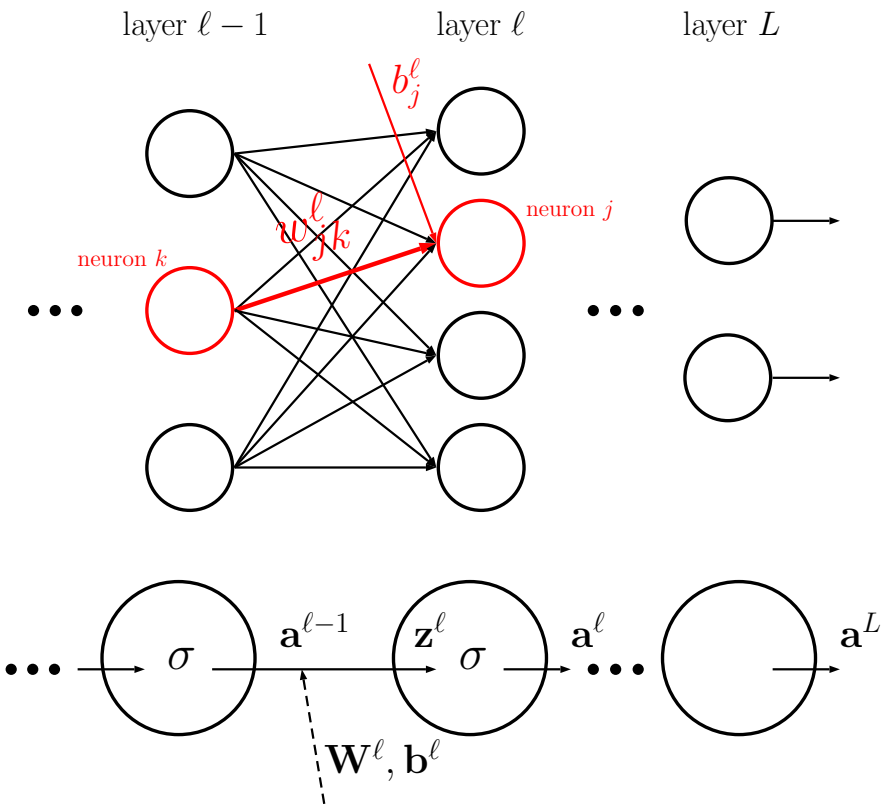
The feedforward relationship

First note that

- Input layer is indexed by $\ell = 0$ so that $\mathbf{a}^0 = \mathbf{x}$.
- $\mathbf{a}^L$ is the network output.

For each $1 \leq \ell \leq L$,

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



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

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

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).

Remark. In our setting, the labels are coded as follows:

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

Therefore, by varying the weights and biases, we try to minimize the difference between each network output $\mathbf{a}^L(\mathbf{x}_i)$ and one of the vectors above (associated to the training class that $\mathbf{x}_i$ belongs to).

Gradient descent

The network loss has too many variables to be minimized analytically:

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

We'll use gradient descent to attack the problem. However, computing all the partial derivatives $\frac{\partial C}{\partial w^\ell_{jk}}, \frac{\partial C}{\partial b^\ell_j}$ is highly nontrivial.

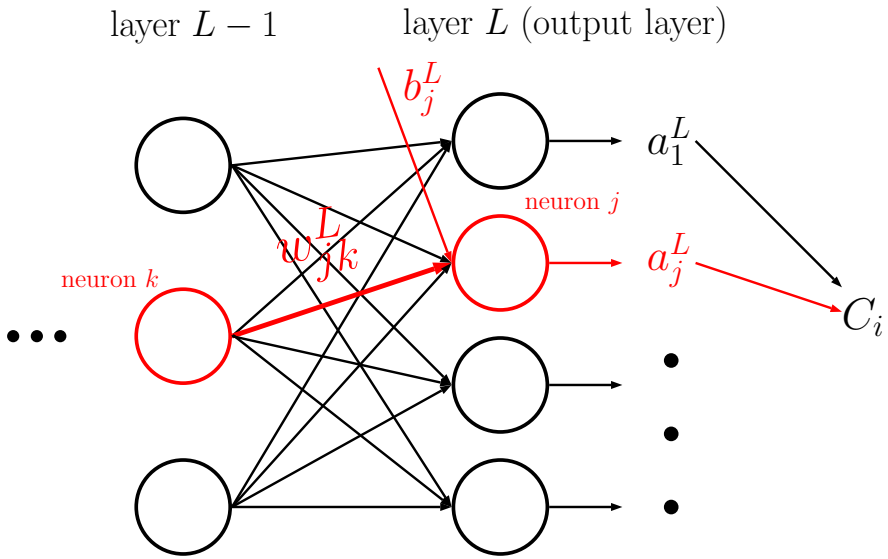
To simplify the task a bit, we consider a sample of size 1 consisting of only $\mathbf{x}_i$:

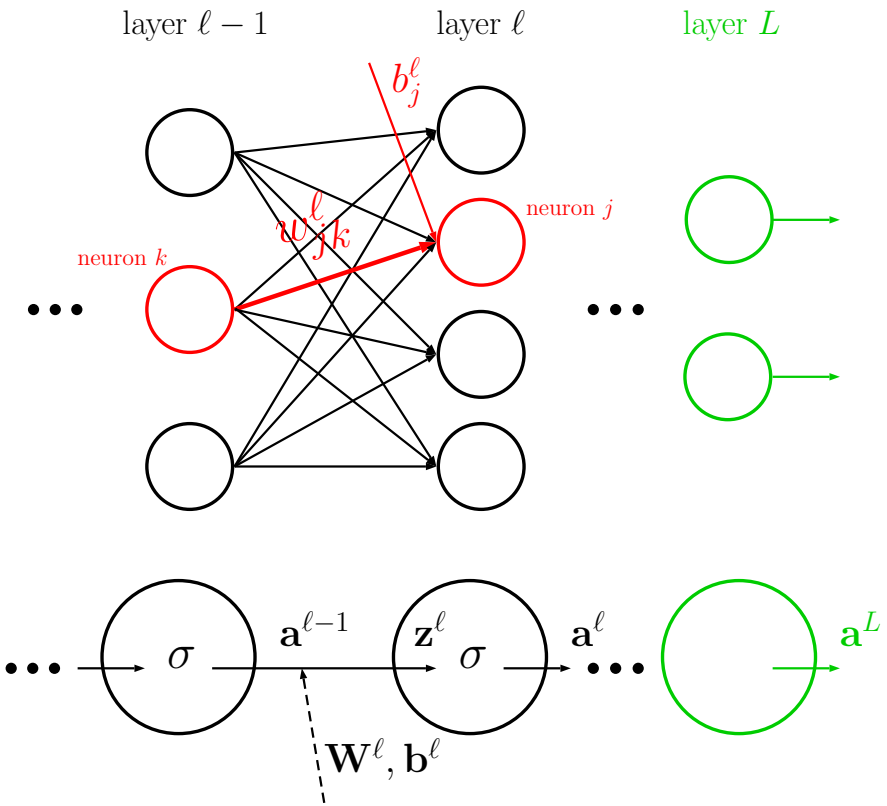
$$C_i(\{\mathbf{W}^\ell, \mathbf{b}^\ell\}_{1\leq \ell \leq L}) = \frac{1}{2}\|\mathbf{a}^L(\mathbf{x}_i) - \mathbf{y}_i\|^2 = \frac{1}{2}\sum_j (a^L_j - y_i(j))^2$$

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

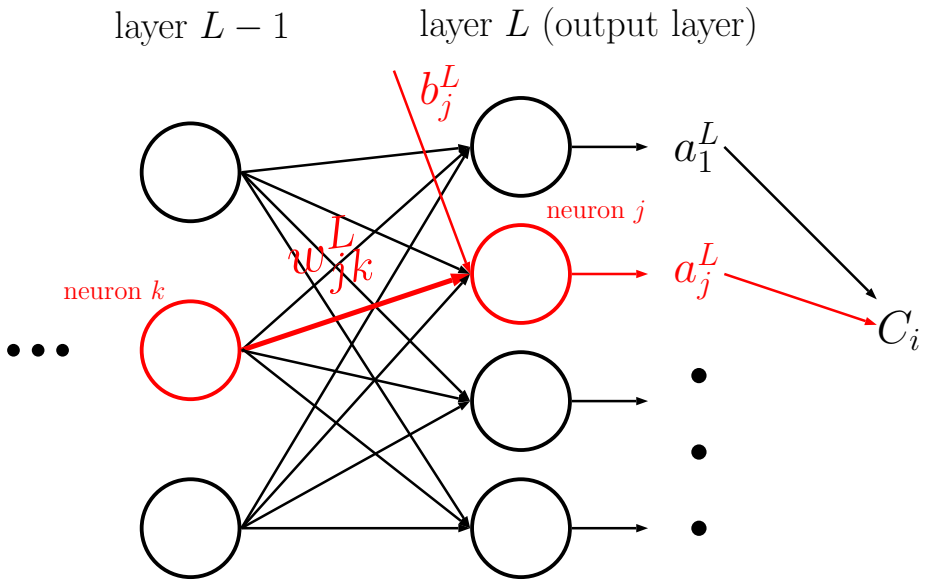
By chain rule we have

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

where $\frac{\partial C_i}{\partial a_j^L} = a_j^L - y_i(j)$ 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.



$$a_j^L = \sigma\left(z_j^L = \sum_k w_{jk}^L a_k^{L-1} + b_j^L\right)$$

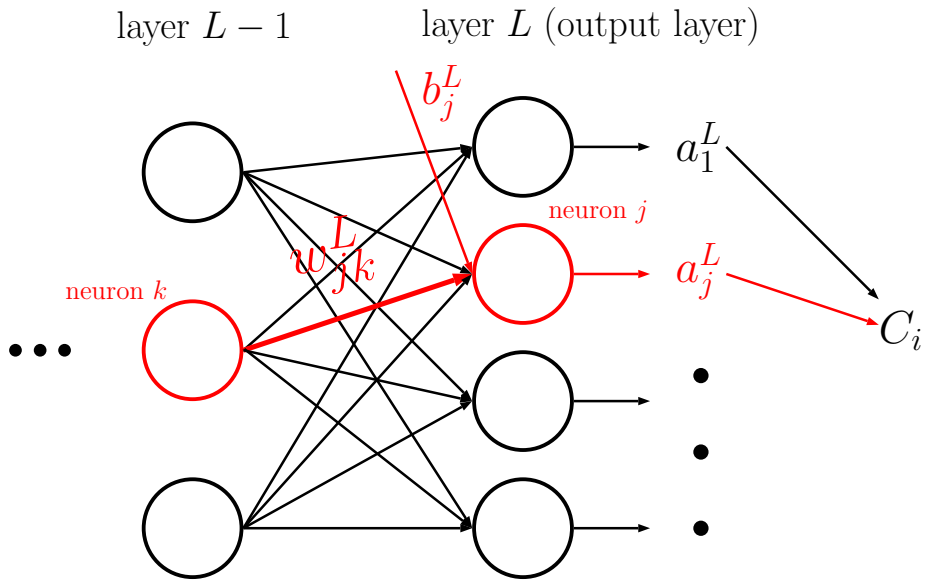
Computing $\frac{\partial C_i}{\partial w_{jk}^L}, \frac{\partial C_i}{\partial b_j^L}$ for the output layer

Combining results gives that

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

Similarly, we obtain that

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

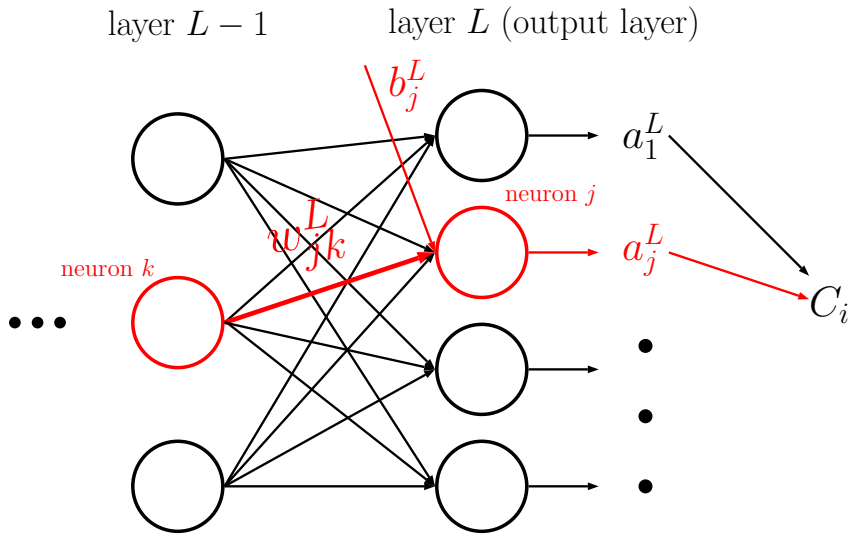


$$a_j^L = \sigma\left(z_j^L = \sum_{k'} w_{jk'}^L a_{k'}^{L-1} + b_j^L\right)$$

Interpretation of the formula for $\frac{\partial C_i}{\partial w^L_{jk}}$

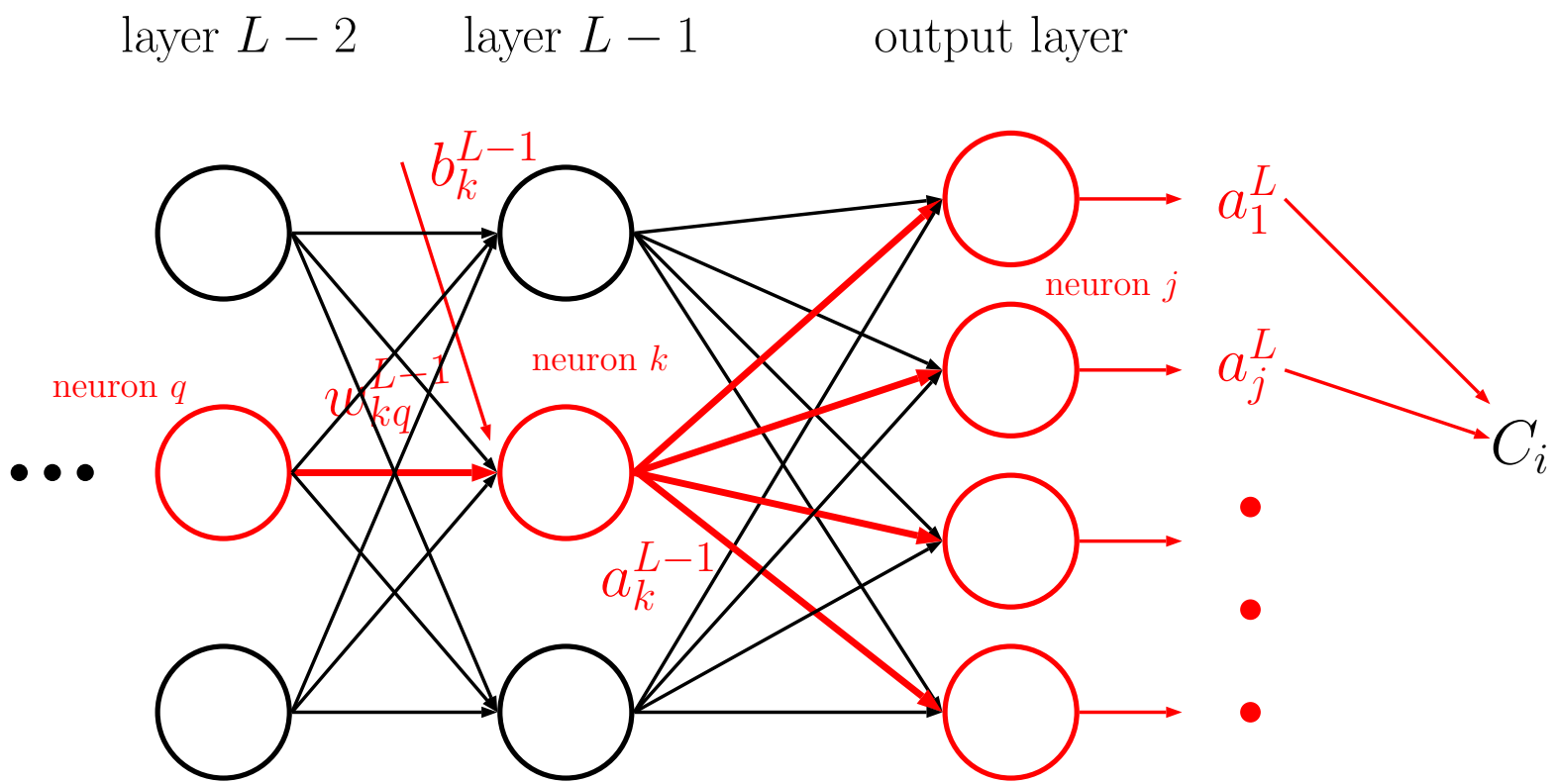
Observe that the rate of change of C_i w.r.t. w^L_{jk} depends on three factors ($\frac{\partial C_i}{\partial b^L_j}$ only depends on the first two):

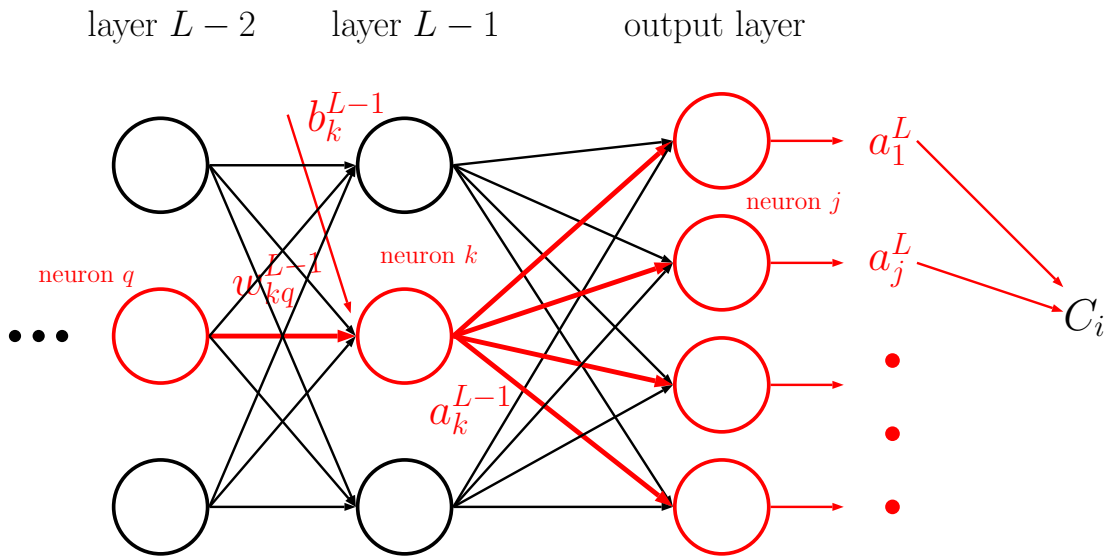
- $a^L_j - y_i(j)$: how much current output is off from desired output
- $\sigma'(z^L_j)$: how fast the neuron reacts to changes of its input
- a^{L-1}_k: contribution from neuron k in layer $L-1$



Thus, w^L_{jk} will learn slowly if the input neuron is in low-activation ($a^{L-1}_k \approx 0$), or the output neuron has "saturated", i.e., is in either high- or low-activation (in both cases $\sigma'(z^L_j) \approx 0$).

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

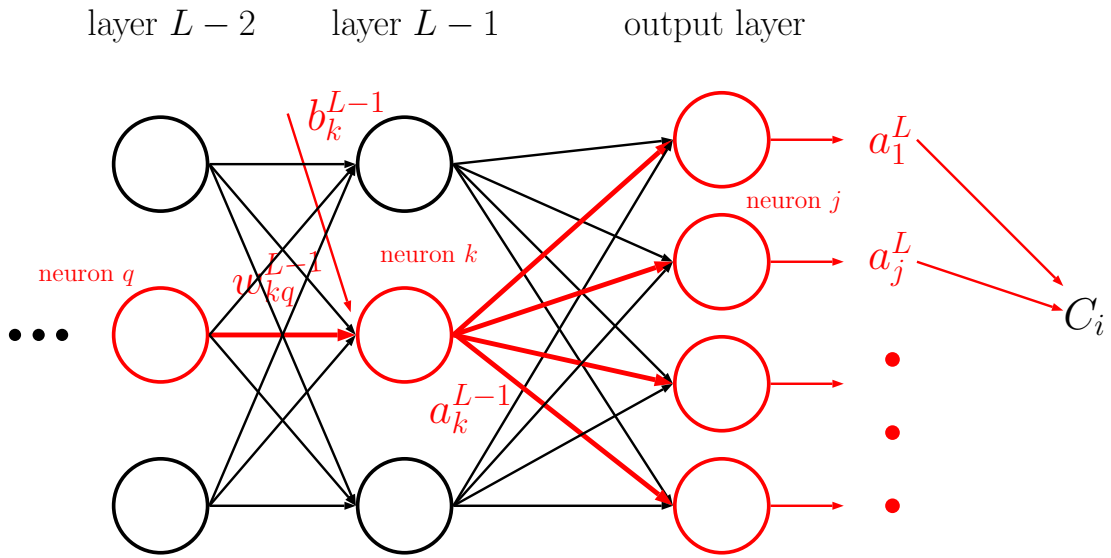




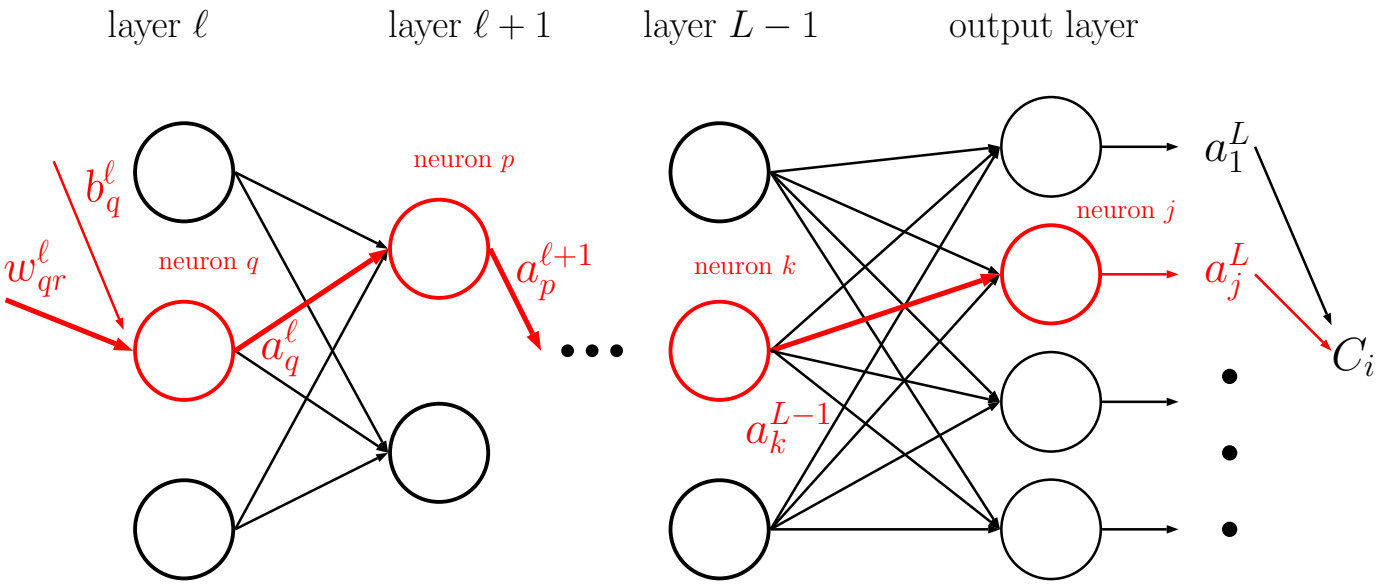
By chain rule,

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

where



- $\frac{\partial C_i}{\partial a_j^L}$: already computed (in the output layer);
- $\frac{\partial a_j^L}{\partial a_k^{L-1}} = \sigma'(z_j^L) w_{jk}^L$: link between layers L and $L-1$;
- $\frac{\partial a_k^{L-1}}{\partial w_{kq}^{L-1}}$: similarly computed as in the output layer



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, thus can 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:

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

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

Software for neural networks

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

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)
```

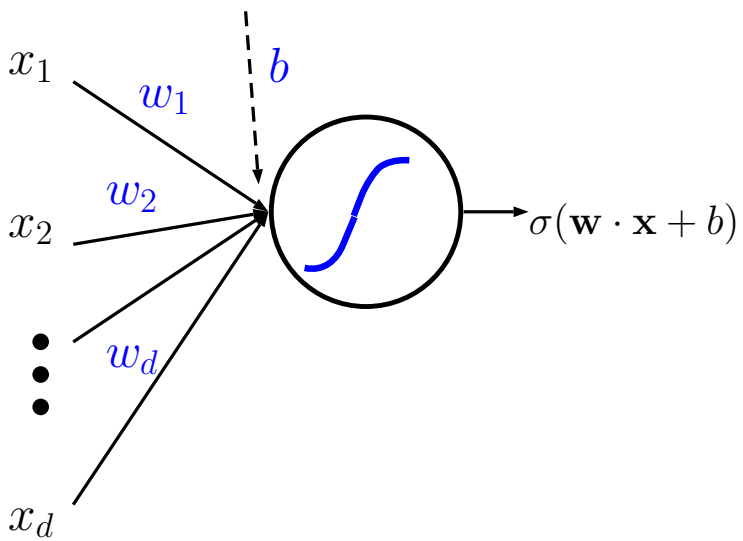
Practical issues and techniques for improvement

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 with square loss

Consider for simplicity a single sigmoid neuron



The total input and output are $z = \mathbf{w} \cdot \mathbf{x} + b$ and $a = \sigma(z)$, respectively.

Under the square loss $C(\mathbf{w}, b) = \frac{1}{2}(a-y)^2$ we obtain that

$$\frac{\partial C}{\partial w_j} = (a-y)\frac{\partial a}{\partial w_j} = (a-y)\,\sigma'(z)\,x_j$$
$$\frac{\partial C}{\partial b} = (a-y)\frac{\partial a}{\partial b} = (a-y)\,\sigma'(z)$$

When z is initially large in magnitude, $\sigma'(z) \approx 0$. This shows that both w_j, b will initially learn very slowly (which could be good or bad):

$$w_j \longleftarrow w_j - \eta \cdot (a-y)\,\sigma'(z)\,x_j,$$
$$b \longleftarrow b - \eta \cdot (a-y)\,\sigma'(z).$$

Therefore, the $\sigma'(z)$ term may cause a learning slowdown when the initial weighted input z is large in the wrong direction.

How to fix the learning slowdown issue

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

$$C(\mathbf{w}, b) = -y \log(a) - (1 - y) \log(1 - a)$$

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

$$\frac{\partial C}{\partial w_j} = (a - y)\, x_j$$
$$\frac{\partial C}{\partial b} = a - y$$

so that gradient descent will move fast when a is far from y.

Remark. A second solution is to add a "softmax output layer" with log-likelihood cost (see Nielson's book, Chapter 3).

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

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

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)

Remark. Two more techniques to deal with overfitting are dropout and artificial expansion of training data (see Nielson's book, Chapter 3).

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)
```

How to initialize weights and biases

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

Regarding weight initialization:

- **First idea**: Initialize w_{jk}^ℓ also as standard Gaussian random variables.
- **Better idea**: For each neuron, initialize the input weights as Gaussian random variables with mean 0 and standard deviation $1/\sqrt{n_{\text{in}}}$, where n_{in} is the number of input weights to this neuron.

Why the second idea is better: the total input to the neuron $z_j^\ell = \sum w_{jk}^\ell a_k^{\ell-1} + b_j^\ell$ has small standard deviation around zero so that the neuron starts in the middle, not from the two ends (see Nielson's book, Chapter 3).

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 often 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.

  and play with the parameters in order 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

Further study (if you are interested)

- Other kinds of neurons such as **tanh**, and **rectified linear**
- **Convolutional neural networks (CNN)**: specially designed for image data
- **Recurrent neural networks (RNN)**: image captioning, speech recognition, text sentiment classification, and language translation
- **Deep learning**

Other activation functions

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

- **Hyperbolic tangent**

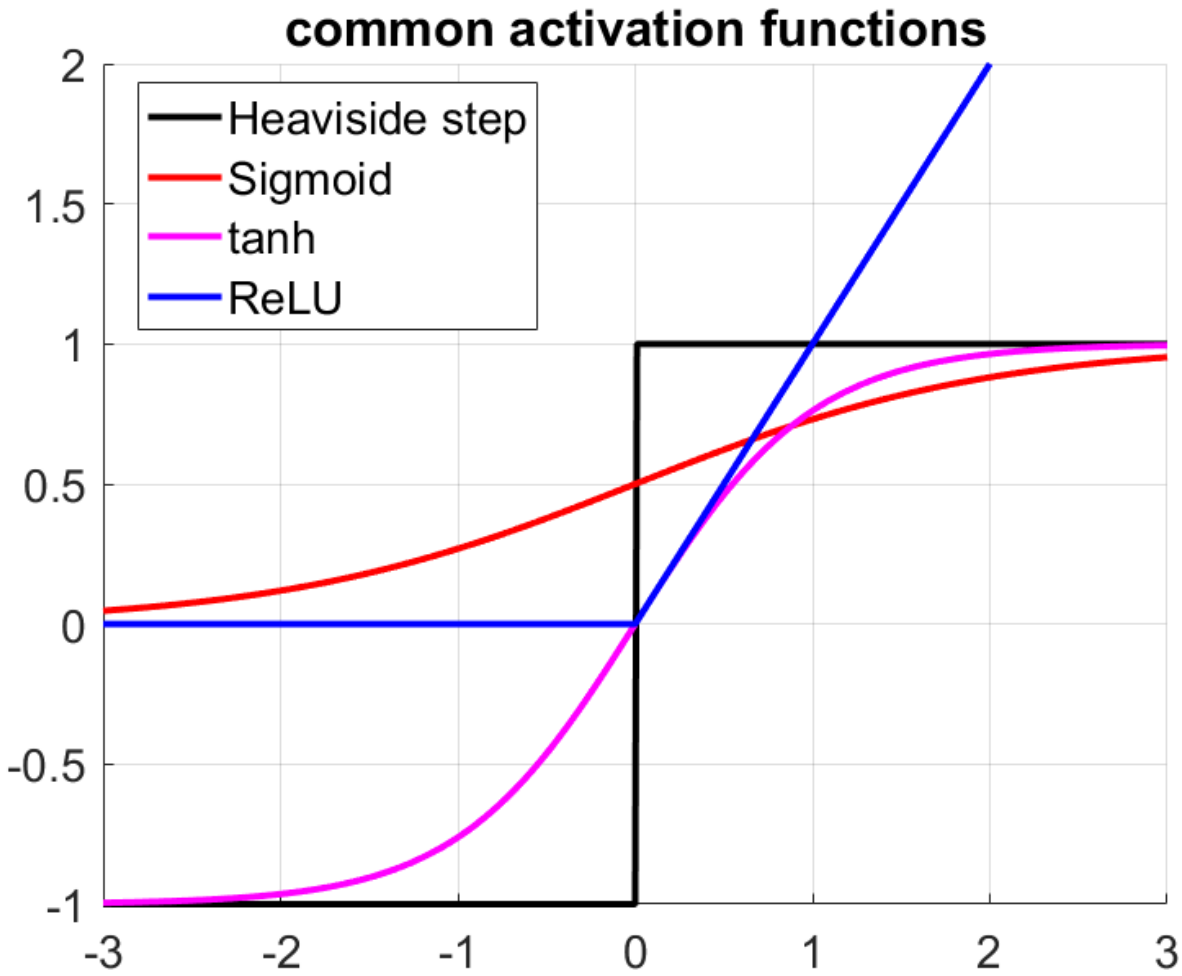
$$g(z) = \tanh(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}} = 2\,\text{sigmoid}(2z) - 1$$

- **Rectified Linear Unit (ReLU)** $\longleftarrow$ very popular

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

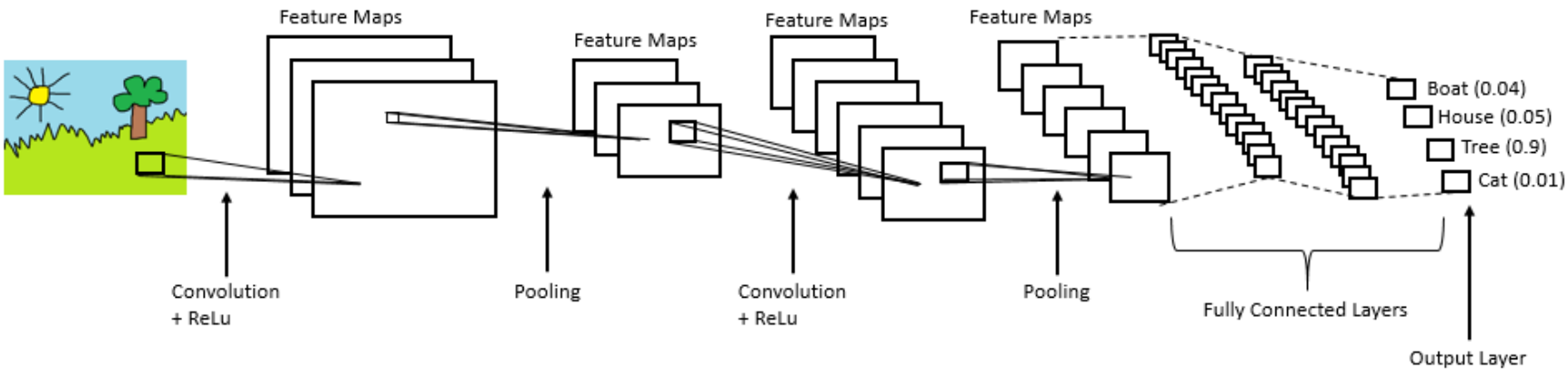
- **Leaky ReLU** (or parameterized ReLU if changing $0.01 \to a$)

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


common activation functions
Heaviside step
Sigmoid
tanh
ReLU
2
1.5
1
0.5
0
-0.5
-1
-3
-2
-1
0
1
2
3

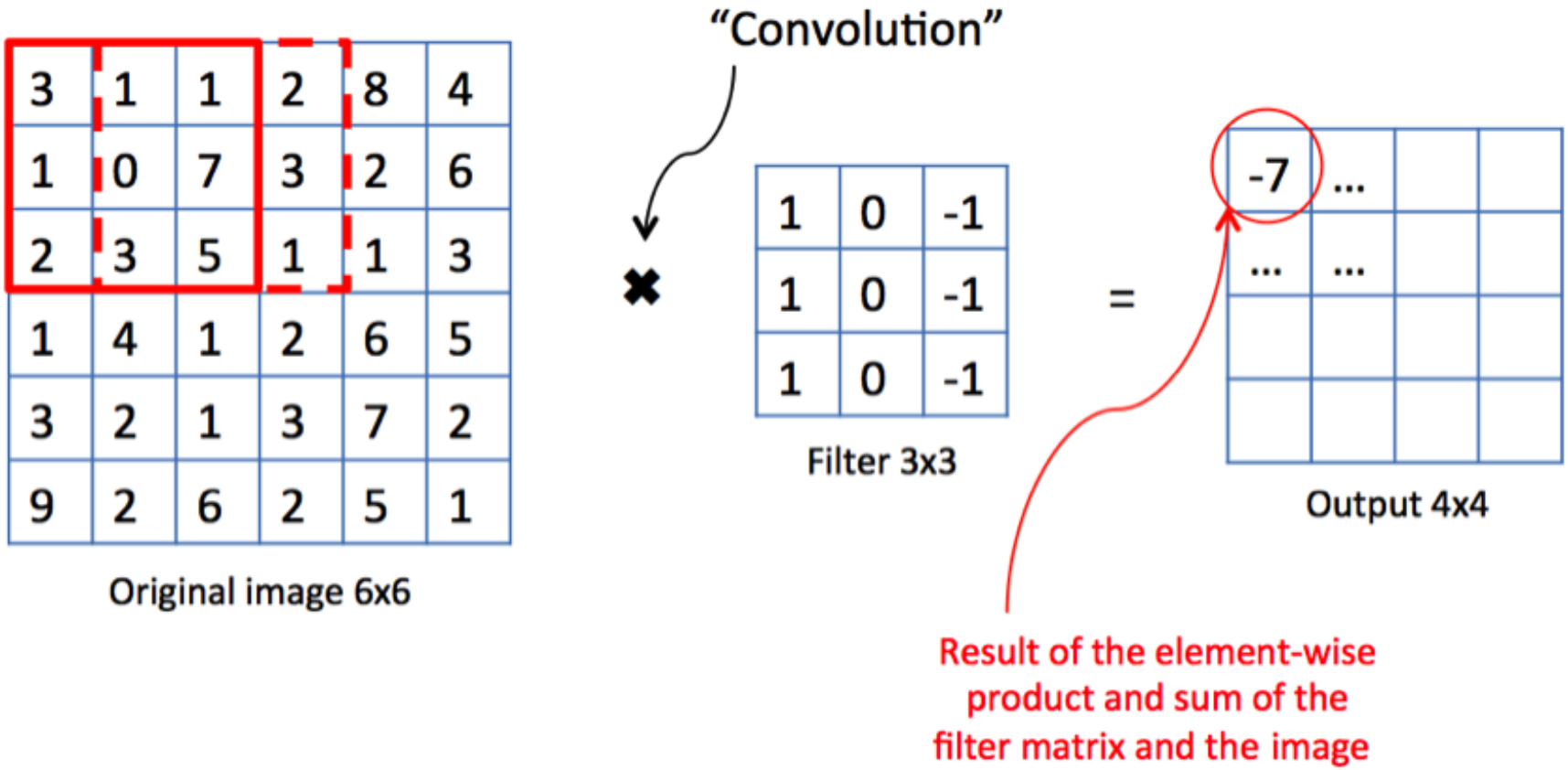
What is a CNN?

Neurons are arranged in 2D arrays and only nearby neurons (pixels) are connected to the same neuron in next layer, in a weights-sharing fashion.

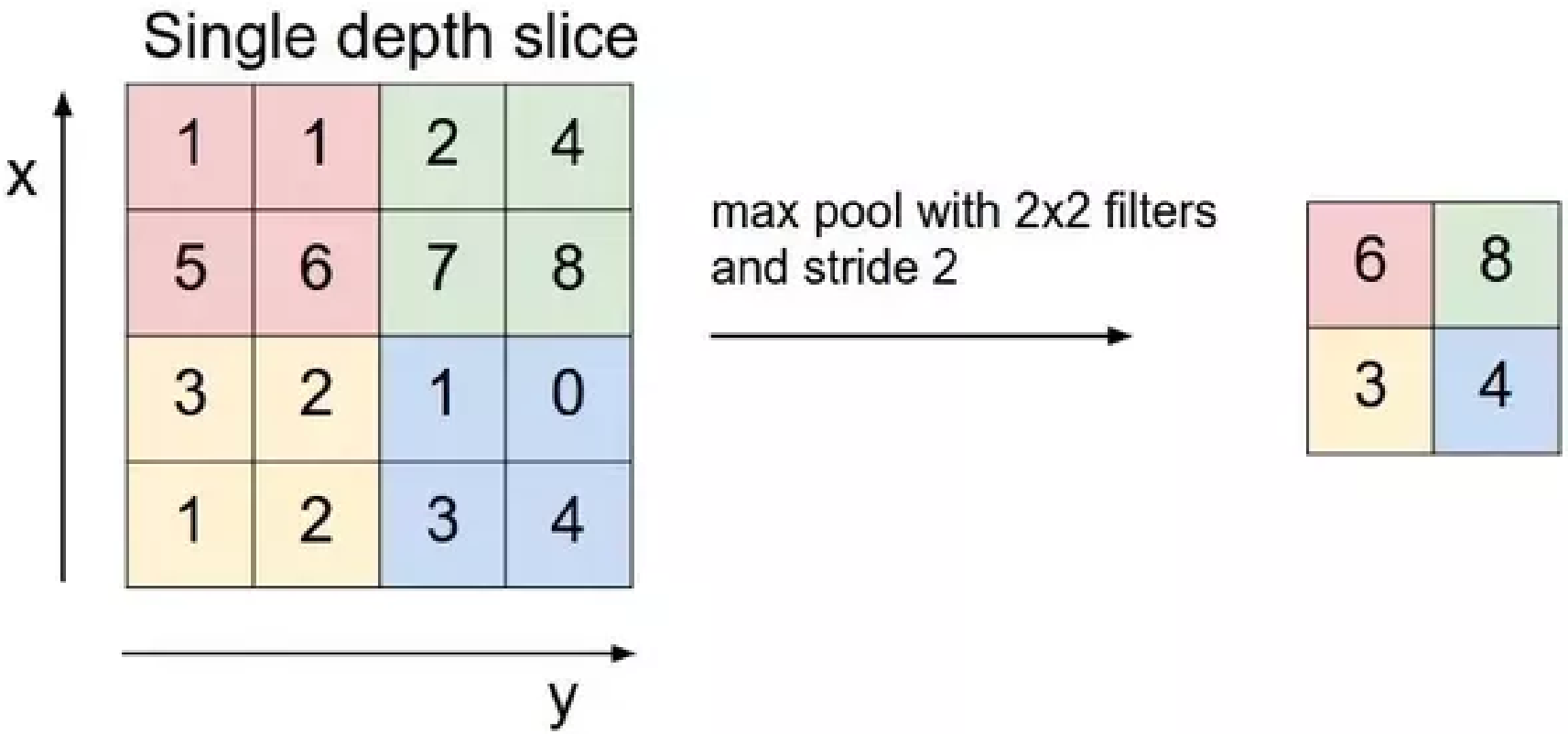


(Copied from Prabhu's online tutorial)

The convolutional layer



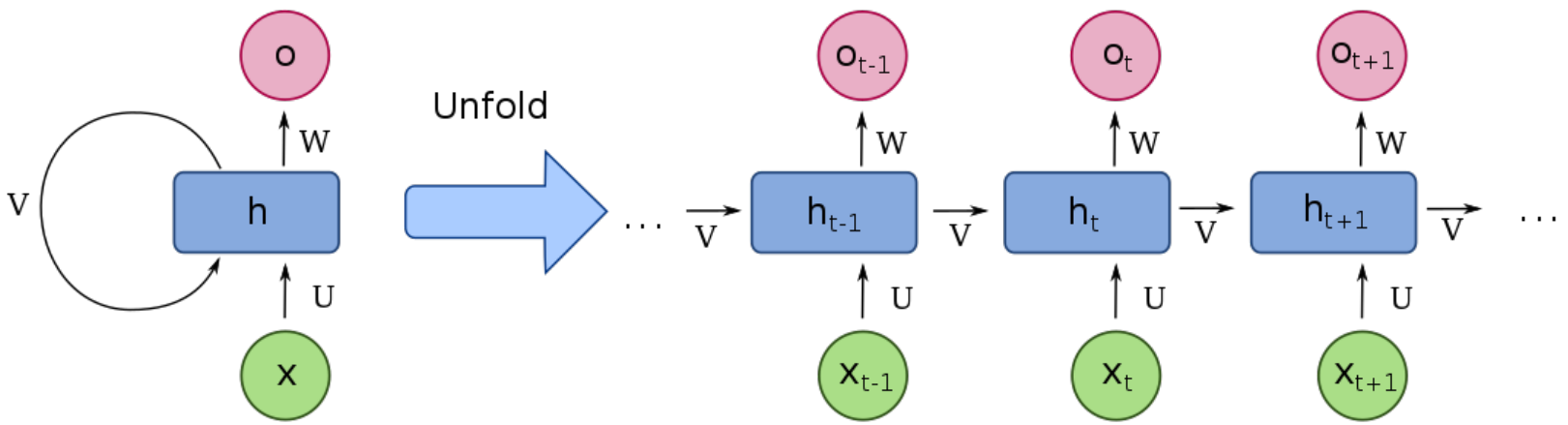
The pooling layer

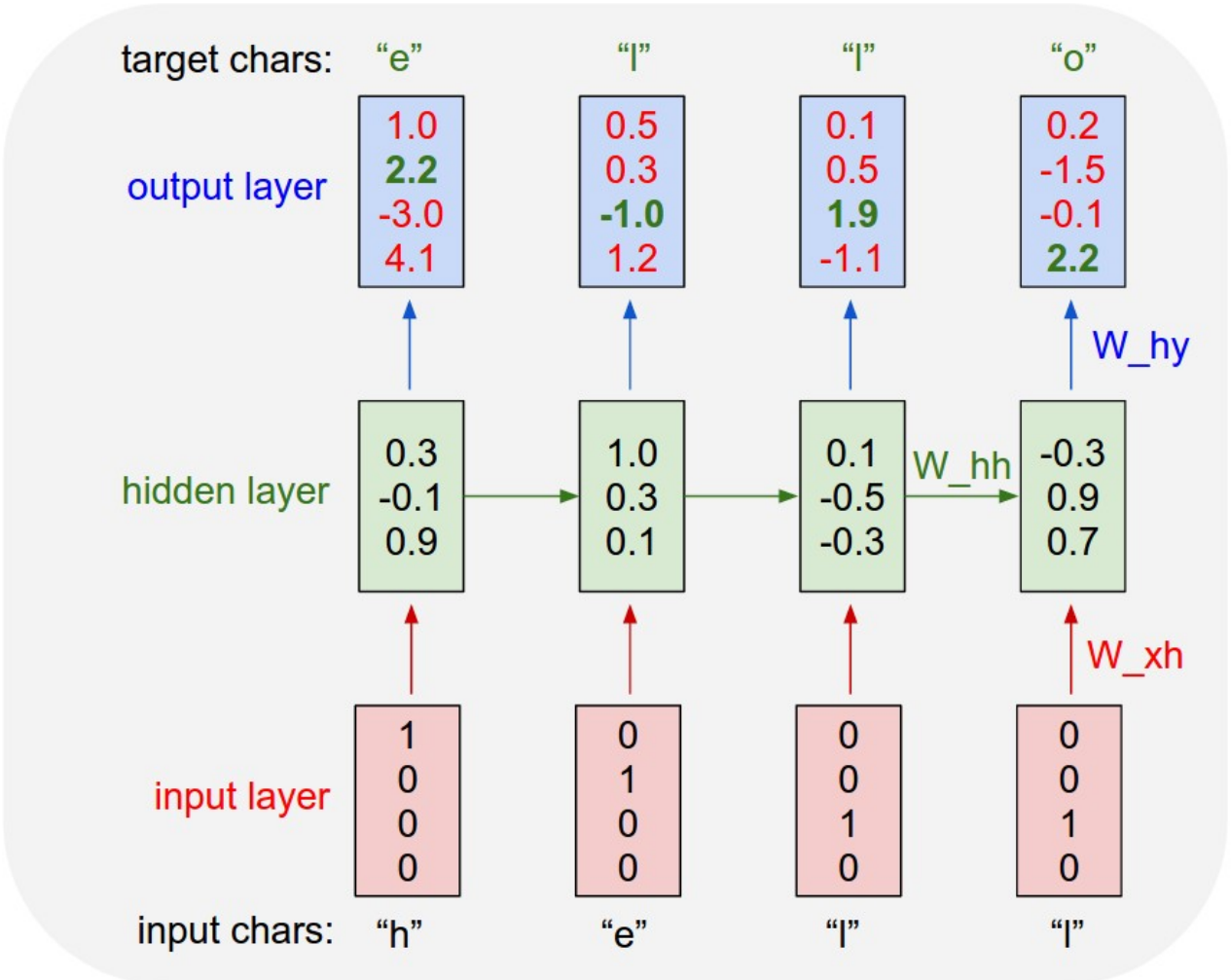


What is an RNN?

Neuron network with 1 hidden layer being replicated in discrete time t:

$$\underbrace{\mathbf{h}_t}_{\text{hidden state at } t} = \tanh(\mathbf{U}\underbrace{\mathbf{x}_t}_{\text{input at } t} + \mathbf{V}\underbrace{\mathbf{h}_{t-1}}_{\text{past state}}), \quad \underbrace{\mathbf{o}_t}_{\text{output at } t} = \text{softmax}(\mathbf{W}\mathbf{h}_t)$$




target chars:
"e"
"l"
"l"
"o"
output layer
1.0
2.2
-3.0
4.1
0.5
0.3
-1.0
1.2
0.1
0.5
1.9
-1.1
0.2
-1.5
-0.1
2.2
W_hy
hidden layer
0.3
-0.1
0.9
1.0
0.3
0.1
0.1
-0.5
-0.3
W_hh
-0.3
0.9
0.7
W_xh
input layer
1
0
0
0
0
1
0
0
0
0
1
0
0
0
1
0
input chars:
"h"
"e"
"l"
"l"

Deep learning

- Chapter 6 of Nielson's book (which also gives a nice introduction to CNN)

- SJSU CMPE 258 Deep Learning (offered in Spring 2019!):
  Deep neural networks and their applications to various problems, e.g., speech recognition, image segmentation, and natural language processing. Covers underlying theory, the range of applications to which it has been applied, and learning from very large data sets. Prerequisite: CMPE 255 Data Mining or CMPE 257 Machine Learning or instructor consent

Practice problems

1. Create a network with just two layers - only input and output, no hidden layer - with 256 and 10 neurons, respectively. Train the network on the USPS data set using stochastic gradient descent. What classification accuracy can you achieve?

2. Now train a neural network with 4 layers [256, 100, 50, 10] and apply it to the USPS digits. What is your best possible result?