LEC 9: Artificial Neural Networks (ANN)

Dr. Guangliang Chen

April 28, 2016

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:

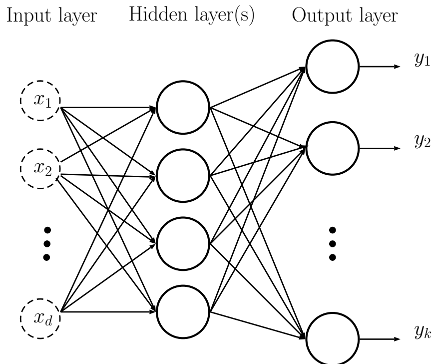
- **Olga Veksler's lecture on neural networks** at

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

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

  http://neuralnetworksanddeeplearning.com

What is an artificial neural network?



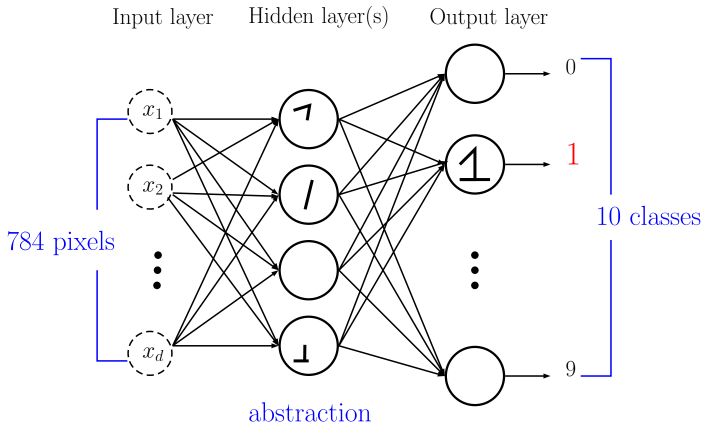
The leftmost layer inputs features.

The rightmost layer outputs results for the user.

The solid circles represent **neurons**, which process inputs from previous layer and output results for next layer (or the user).

The network may have more than 1 hidden layer (called a **deep** network).

ANN for MNIST handwritten digits recognition



The rise of ANNs

- Automatically make increasing levels of abstraction of input features
- Nowadays people can train deep networks with lots of neurons in each layer
- Can carve out arbitrarily complex decision boundaries without requiring as many terms as polynomial functions
- Have won many machine learning competitions
- Have achieved a 0.21% error rate (i.e., only 21 errors!) for MNIST digits classification

A little history about ANN

- Originally inspired by brain research (but cannot claim that this is how the brain actually works)
- 1958: Perceptron (a single-layer neural network) was first introduced by F. Rosenblatt of Cornell University, but no further progress until the 1980s
- 1986: Rediscovery of the backpropagation algorithm, making it possible to train multilayer neural networks
- 1998: Convolutional network (convnet) by Y. Lecun for digit recognition, very successful

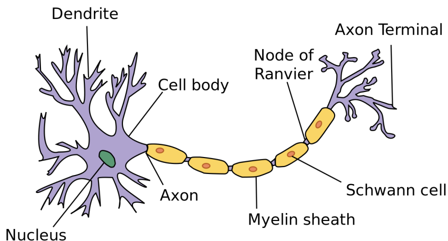
- 1990s: Research in NN moves slowly again
  - Networks with multiple layers are hard to train well (except convnet for digit recognition)
  - SVM becomes popular, works better
- Since 2006: deep networks are trained successfully
  - massive training data becomes available
  - better hardware: fast training on GPU
  - better training algorithms for network training when there are many hidden layers

- Breakthrough papers
  - Hinton, G. E, Osindero, S., and Teh, Y. W. (2006). A fast learning algorithm for deep belief nets. Neural Computation, 18:1527-1554.
  - Bengio, Y., Lamblin, P., Popovici, P., Larochelle, H. (2007). Greedy Layer-Wise Training of Deep Networks, Advances in Neural Information Processing Systems 19
- Extensive use
  - Government: automatic recognition of zip codes and license plates etc.
  - Industry: Facebook, Google, Microsoft, etc.

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 1000 10,000 other neurons

Main components of a biological neuron



cell body: computational unit

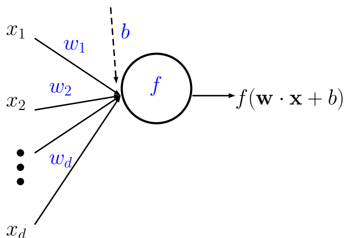
dendrites:

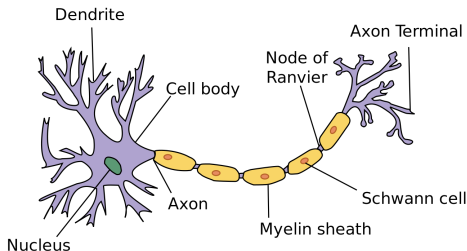
- "input wires", receive inputs from other neurons
- a neuron may have thousands of dendrites, usually short

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



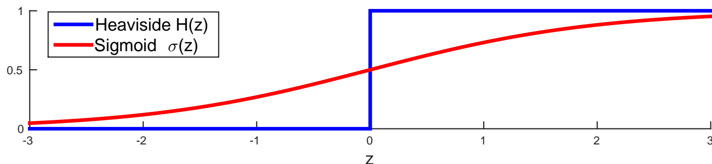


Artificial neurons are mathematical functions from $\mathbb{R}^d \mapsto \mathbb{R}$ defined by the

- w_i: **weights**, b: **bias**, and f: rule (called **activation function**)

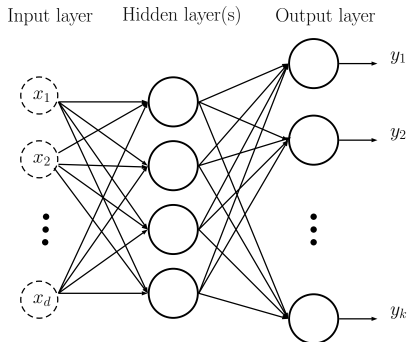
Two common activation functions

- 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.

The functional perspective of ANN



ANN is a composition of many functions!

- easier to visualize as a network
- notation gets ugly

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

How to train ANNs in principle

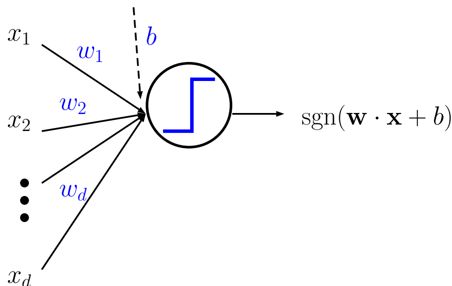
First, we need to select an activation function for all neurons.

Afterwards, we 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 gradient complexity
  - lots of tricks to get gradient descent work reasonably well

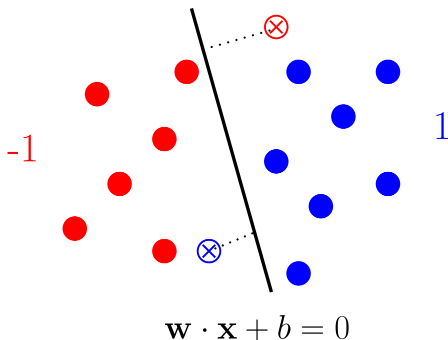
Perceptrons

A perceptron is a linear, binary classifier with 0/1 output (we say that the perceptron fires when the output is 1).



The Perceptron loss function

Given two classes codes by $y_i = \pm 1$, the goal is to find a separating hyperplane by minimizing the distances of misclassified points to the decision boundary.



Derivation:

- If a point $\mathbf{x}_i$ is misclassified, then $y_i(\mathbf{w} \cdot \mathbf{x}_i + b) < 0$.
- The distance from any $\mathbf{x}_i$ to the hyperplane $\mathbf{w} \cdot \mathbf{x}_i + b = 0$ is $\frac{|\mathbf{w} \cdot \mathbf{x}_i + b|}{\|\mathbf{w}\|_2}$.
- The distance from a misclassified point $\mathbf{x}_i$ to the hyperplane can be expressed as $\frac{-y_i(\mathbf{w} \cdot \mathbf{x}_i + b)}{\|\mathbf{w}\|_2}$.
- Denote the set of misclassified points by $\mathcal{M}$.
- The goal is to minimize a scalar multiple of the total distance:

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

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 an iterative procedure:

- **Initialize** weights $\mathbf{w}$ and bias b (which would determine a $\mathcal{M}$).
- **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 use gradient descent to update $\mathbf{w}, b$

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

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

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

Interpretation:

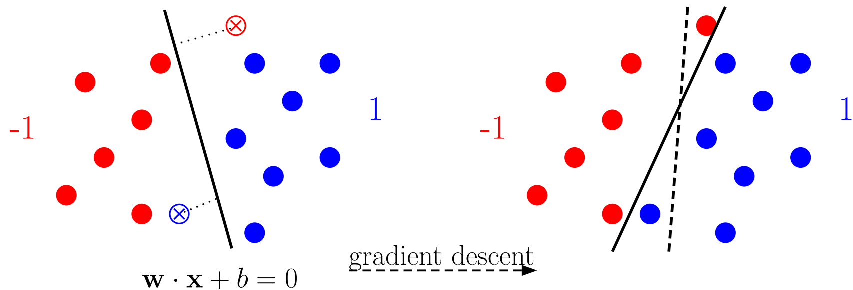
* Since $\sum_{i\in\mathcal{M}} y_i > 0$ (<0) if there are more errors in the positive (negative) class, b will be modified in favor of the dominant class of errors.

* 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:

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


-1
1
$\mathbf{w} \cdot \mathbf{x} + b = 0$
gradient descent
-1
1

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 previous method is called **full gradient descent**, as weights and bias are updated only after all examples are seen and processed (which might take time for large data sets).

A more efficient way is to use **stochastic gradient descent**:

- **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**)

- **Batch** update rule:
  - Divide training data into mini-batches, and update weights after processing each batch
  - Middle ground between single sample and full training set
  - One iteration over all mini-batches is called an epoch

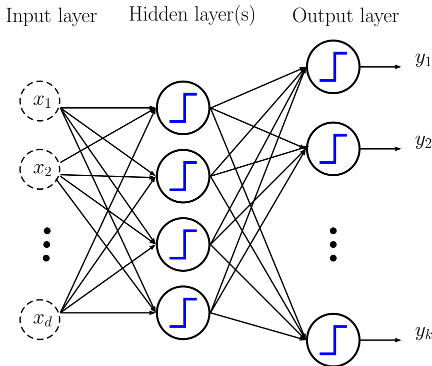
Comments on stochastic gradient descent

- Single-sample update rule applies to online learning (when data comes sequentially)
- 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)

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.
- A few issues
  - When the data are separable, there are many solutions, and which one is found depends on the starting values.
  - The number of steps can be very large. The smaller the gap (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).

Multilayer perceptrons (MLP)



MLP is a network of perceptrons.

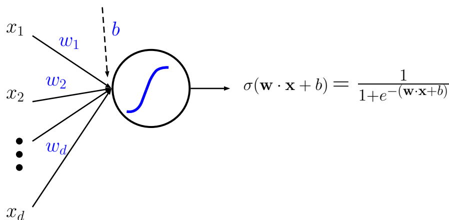
However, each perceptron has a discrete behavior, making its effect on latter layers hard to predict.

Next time we will look at the network of sigmoid neurons.

Sigmoid neurons

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

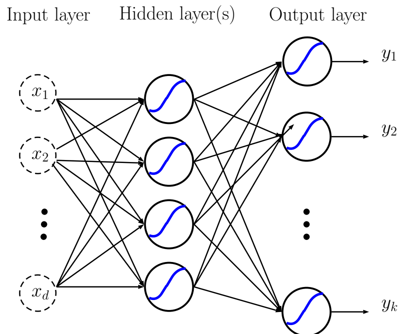
- We say the neuron is in low (high) activation if the output is near 0 (1).
- A "small" change in any weight or bias causes only a "small" change in the output.



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

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).



So how do we train a neural network?

- Notation
- Backpropagation
- Practical issues and solutions

Notation

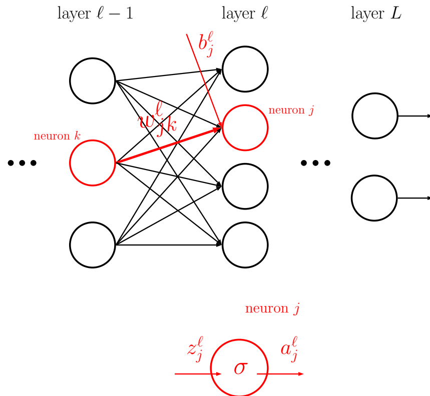
w^{ℓ}_{jk}: layer ℓ, "j back to k" weight;

b^{ℓ}_{j}: layer ℓ, neuron j bias

a^{ℓ}_{j}: layer ℓ, neuron j output

$z^{\ell}_{j} = \sum_k w^{\ell}_{jk} a^{\ell-1}_{k} + b^{\ell}_{j}$: weighted input to neuron j in layer ℓ

Note that $a^{\ell}_{j} = \sigma(z^{\ell}_{j})$.



Notation (vector form)

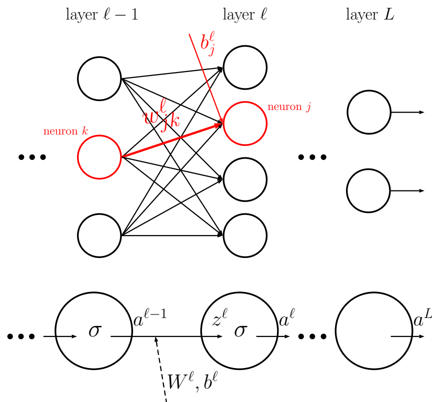
$\mathbf{W}^\ell = \left(w^\ell_{jk}\right)_{j,k}$: matrix of all weights between layers $\ell - 1$ and ℓ;

$\mathbf{b}^\ell = \left(b^\ell_j\right)_j$: 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

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



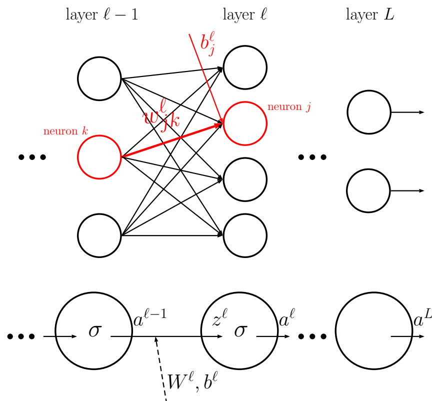
The feedforward relationship

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:

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

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:

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

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

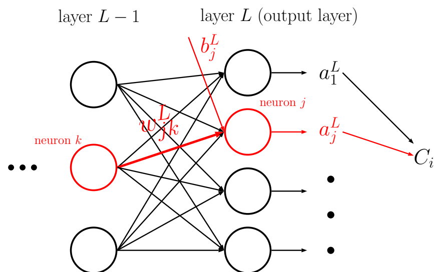
To simplify the task a bit, we consider a sample of size 1 consisting of only $\mathbf{x}_i$:

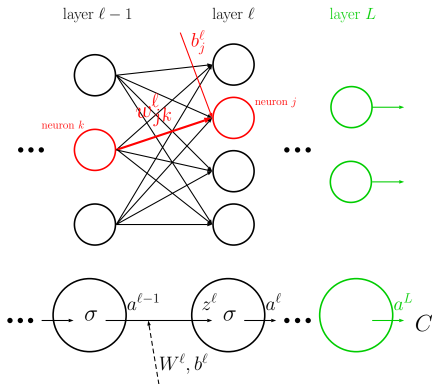
$$C_i(\{\mathbf{W}^\ell, \mathbf{b}^\ell\}_{1\le \ell\le L}) = \frac{1}{2}\|\mathbf{a}^L(\mathbf{x}_i) - \mathbf{y}_i\|^2 = \frac{1}{2}\sum_j (a_j^L - y_i(j))^2$$

which is enough as $\frac{\partial C}{\partial w_{jk}^\ell} = \frac{1}{n}\sum_i \frac{\partial C_i}{\partial w_{jk}^\ell}$ and $\frac{\partial C}{\partial b_j^\ell} = \frac{1}{n}\sum_i \frac{\partial C_i}{\partial b_j^\ell}$.

The output layer first

We start by computing $\frac{\partial C_i}{\partial w_{jk}^L}, \frac{\partial C_i}{\partial b_j^L}$ 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

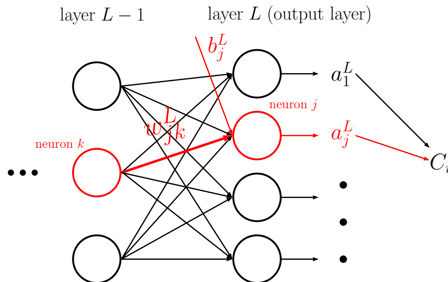
By chain rule we have

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

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.



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

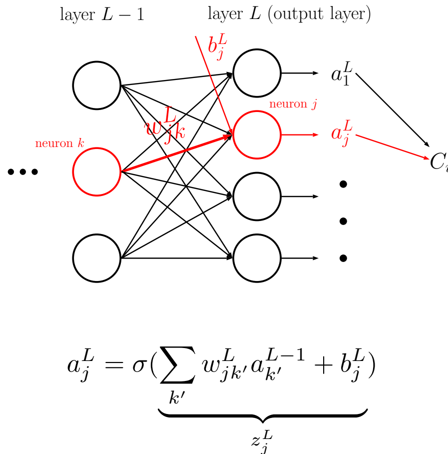
Computing $\frac{\partial C_i}{\partial w_{jk}^L}, \frac{\partial C_i}{\partial b_j^L}$ for the output layer

Combining results gives that

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

Similarly, we obtain that

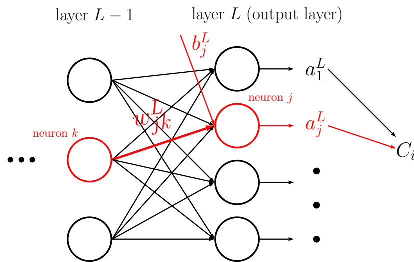
$$\begin{aligned}\frac{\partial C_i}{\partial b_j^L} &= \frac{\partial C_i}{\partial a_j^L} \cdot \frac{\partial a_j^L}{\partial b_j^L} \\ &= \left(a_j^L - y_i(j)\right) \sigma'(z_j^L).\end{aligned}$$



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

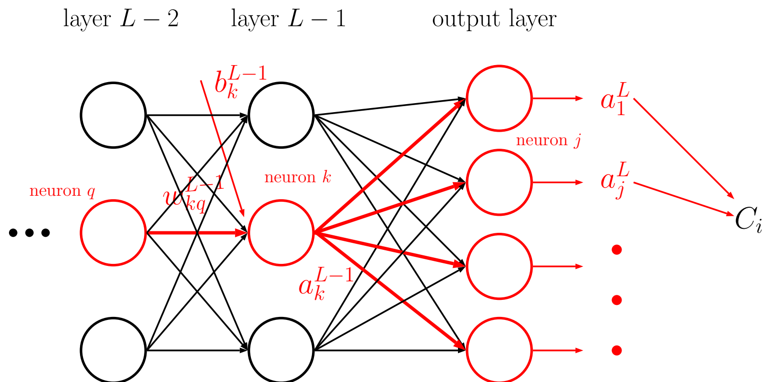
Observe that the rate of change of C_i w.r.t. w_{jk}^L depends on three factors ($\frac{\partial C_i}{\partial b_j^L}$ only depends on the first two):

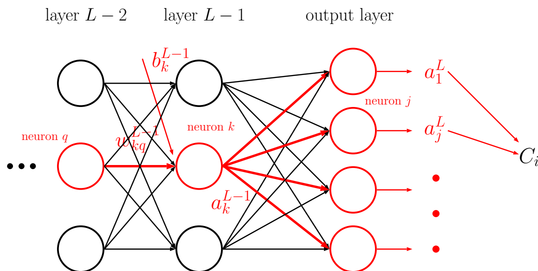
- $a_j^L - y_i(j)$: how much current output is off from desired output
- $\sigma'(z_j^L)$: how fast the neuron reacts to changes of its input
- a_k^{L-1}: contribution from neuron k in layer $L-1$



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

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

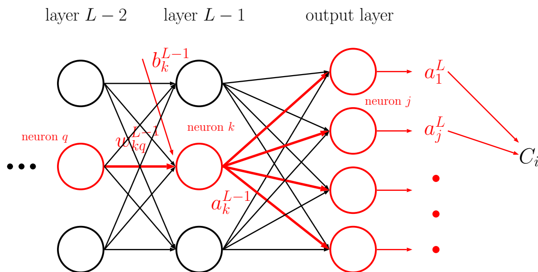




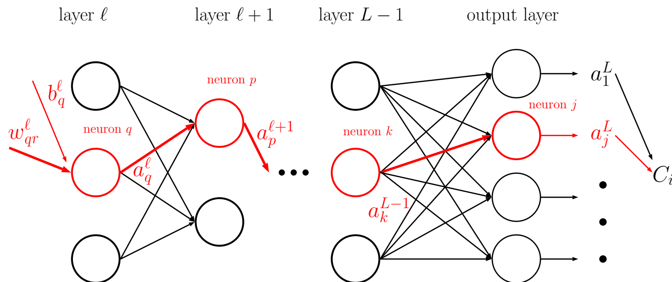
By chain rule,

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

where



- $\frac{\partial C_i}{\partial a_j^L}$: already computed (in the output layer);
- $\frac{\partial a_j^L}{\partial a_k^{L-1}}$: 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 (i.e., to the left), 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 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.

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:

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

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

Codes for neural networks

Bad news: Neural networks is not part of the MATLAB Statistics and Machine Learning Toolbox, and SJSU has not purchased a license for the MATLAB Neural Networks Toolbox.

Good news: 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`. So we'll use his Python codes for demonstration.

Codes for neural networks (cont'd)

```
# 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)
```

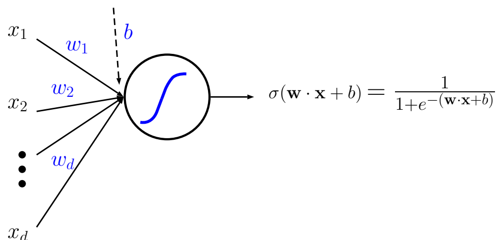
Practical issues and techniques for improvement

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

- Vector form of backpropogation for fast calculations
- 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, the regularization parameter, 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 learn very slowly (for a while):

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

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).

Python codes 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).

Python codes 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 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)
```

(The line `net.large_weight_initializer()` is struck through on the slide.)

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."

Further study (if you are interested)

- Other kinds of neurons such as RBF, tanh, and rectified linear
- Recurrent neural networks
- Convolutional nets
- Deep learning

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

Optional HW6b (due Wed. noon, May 17)

This homework tests neural networks on the MNIST digits. In both questions below report your results using graphs and/or texts.

4. Try creating a network with just two layers - only input and output, no hidden layer - with 784 and 10 neurons, respectively. Train the network using stochastic gradient descent. What classification accuracy can you achieve?

5. Now train a neural network with 4 layers [784, 25, 4, 10] and apply it to the MNIST digits. What is your best possible result?

Midterm project 7: Neural networks

Summarize the ideas of neural networks as well as the results obtained on the MNIST digits. You are also encouraged to try new options and compare with other relevant methods.