AARHUS
UNIVERSITET

Computer Vision & Machine Learning

Alexandros Iosifidis
@
Department of Electrical and Computer Engineering
Aarhus University

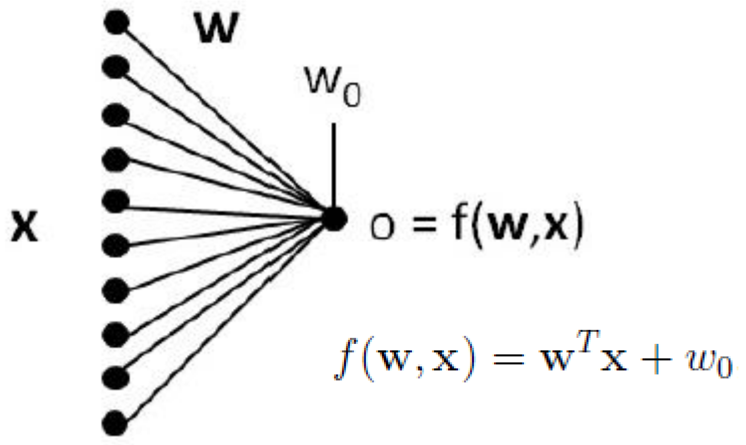
This week

(Multi-layer) Feedforward Neural Networks:
- Architectures
- Training

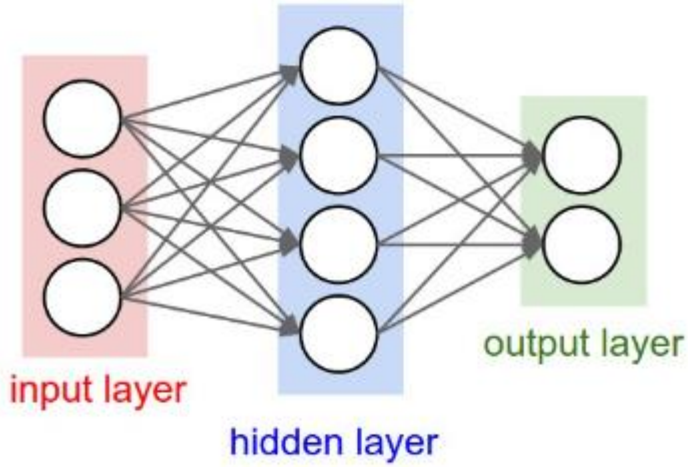
Convolutional Neural Networks:
- Extensions of concepts in ML-FNNs
- Architectures
- Applications

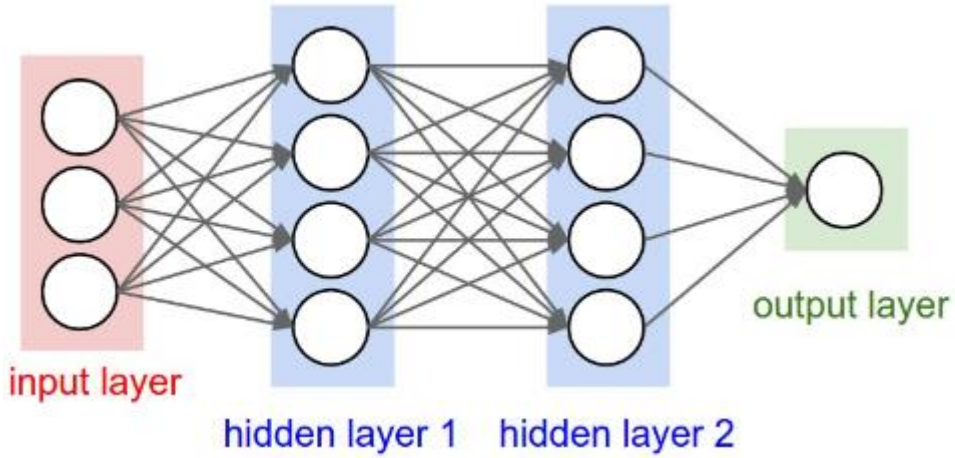
Slides based on:
- S.S. Shwartz & S.B. David: Understanding ML
- L. Fei-Fei: CNNs for Visual Recognition

Feedforward neural networks



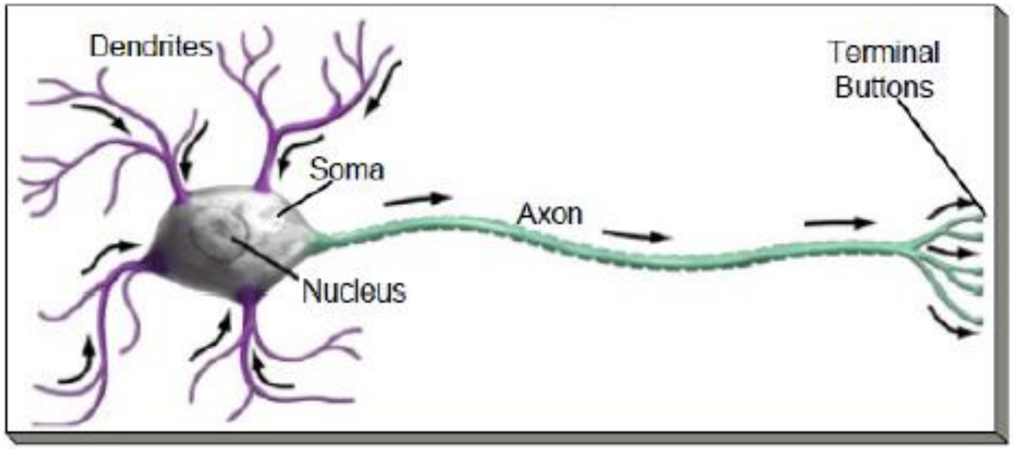
$$f(\mathbf{w},\mathbf{x}) = \mathbf{w}^T\mathbf{x} + w_0$$

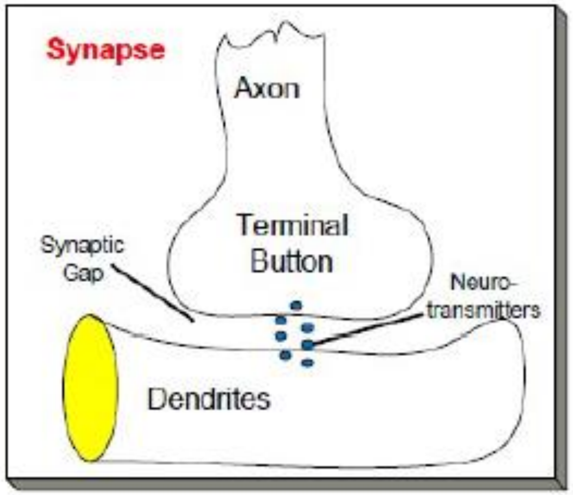


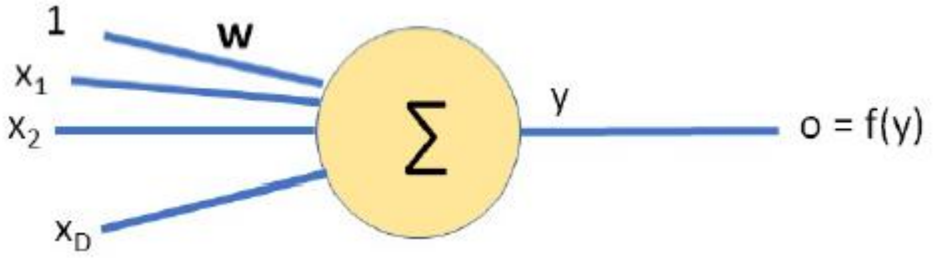


The neuron

Biological neuron and computational approximation





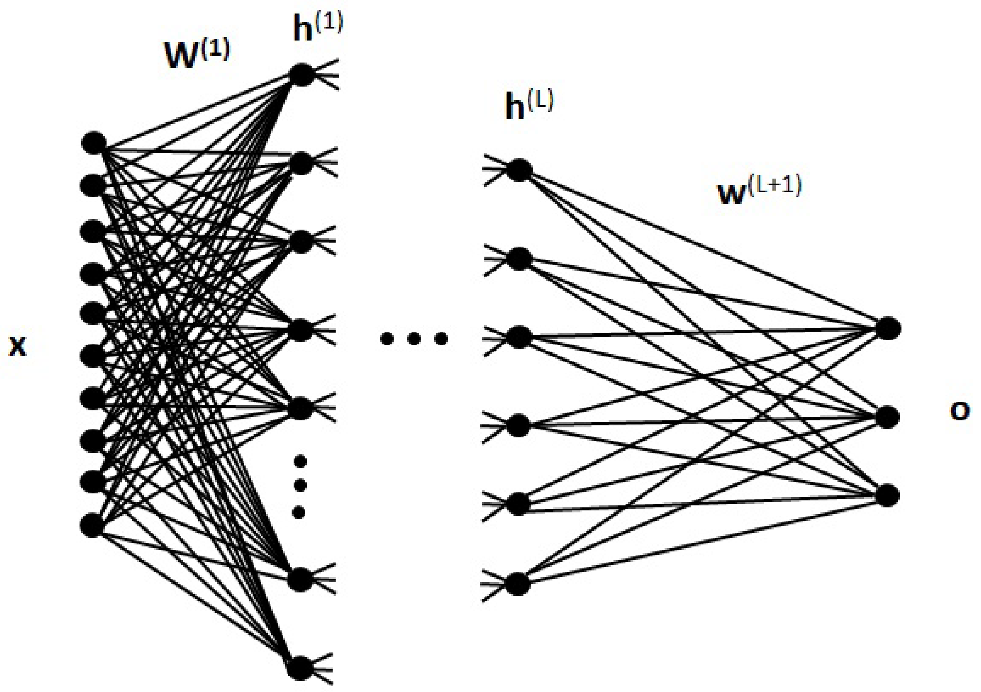


$$f(\mathbf{w},\mathbf{x}) = \mathbf{w}^T\mathbf{x} + w_0$$

Multilayer neural network

We can ‘stack’ many neurons on top of the other in order to create a ‘layer’.

We can stack one layer on top of the other in order to create a ‘deeper neural network’

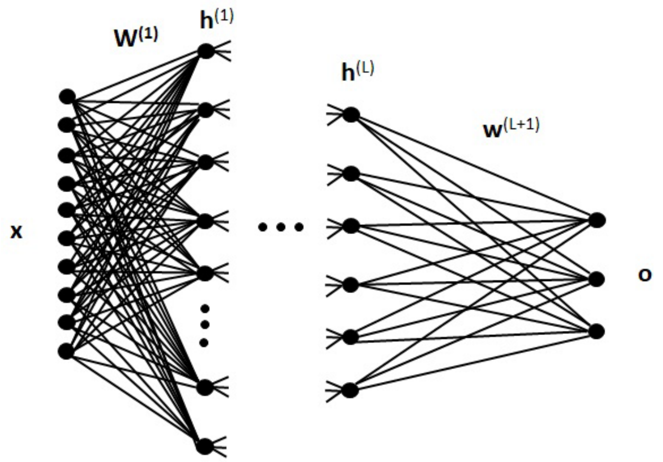


Multilayer neural network

We can ‘stack’ many neurons on top of the other in order to create a ‘layer’.

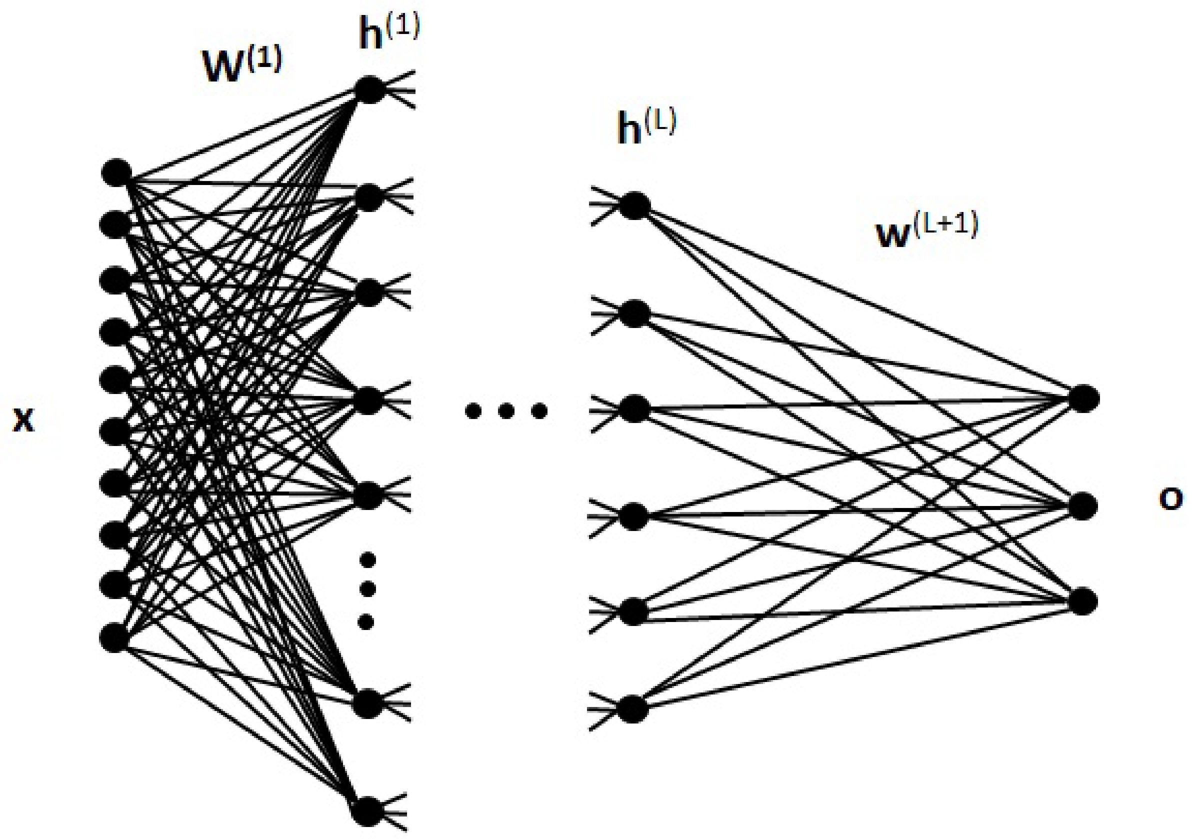
We can stack one layer on top of the other in order to create a ‘deeper neural network’.

Because the actual operation of the network transfers the information from the input to the output, these networks are usually called feedforward neural networks.



Multilayer neural network

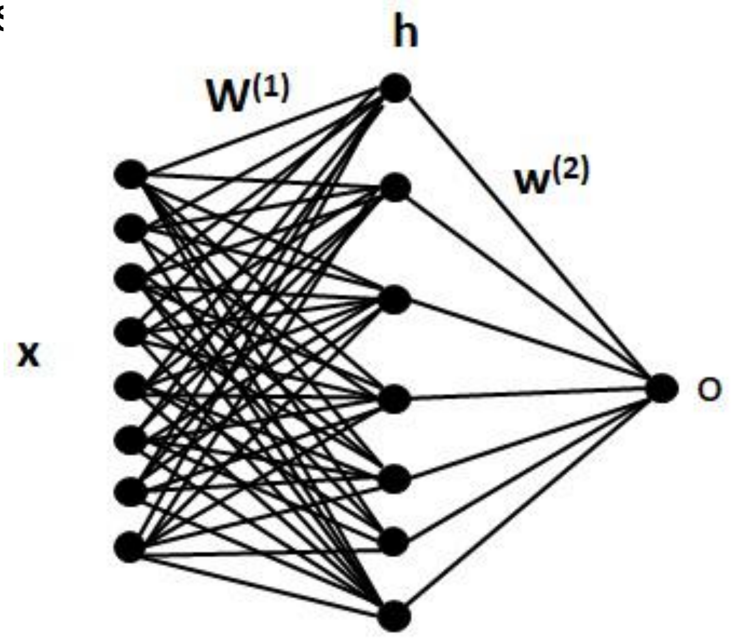
Which are the parameters of the network?



Three-layer neural network

Let us focus on the three-layer topology.

Since this neural networks having this topology (usually called architecture) have one input, one output and one hidden layer, they are usually called Single-hidden Layer Feedforward Neural networks

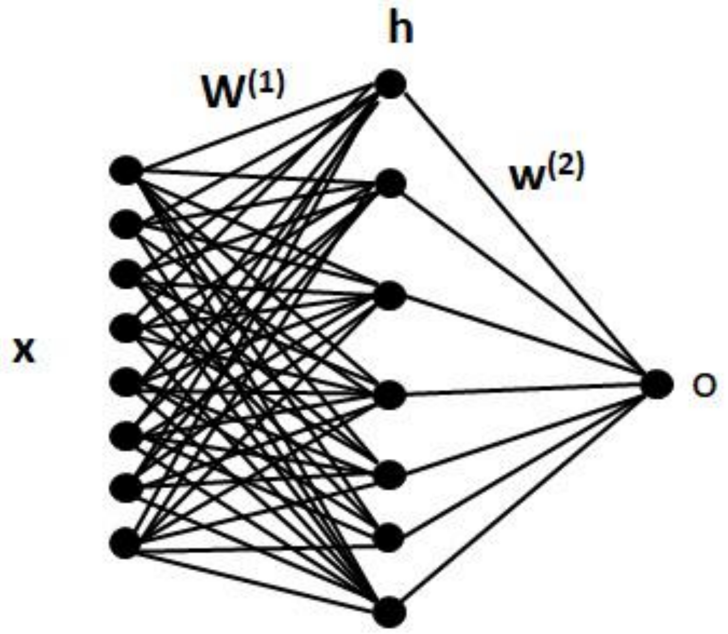


Three-layer neural network

Let us focus on the three-layer topology.

Each of the hidden neurons has an output equal to

$$h_k = f\left(\mathbf{W}_k^{(1)T}\mathbf{x}\right),\ k = 1,\dots,K = 7$$



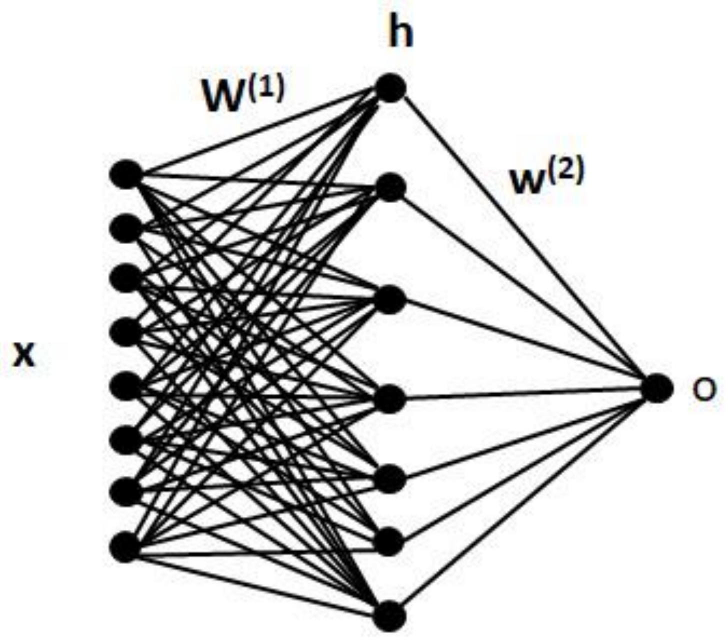
Three-layer neural network

Let us focus on the three-layer topology.

Each of the hidden neurons has an output equal to

$$h_k = f\left(\mathbf{W}_k^{(1)T}\mathbf{x}\right),\ k = 1, \ldots, K = 7$$

All h_k outputs are used to form the vector **h**



Three-layer neural network

Let us focus on the three-layer topology.

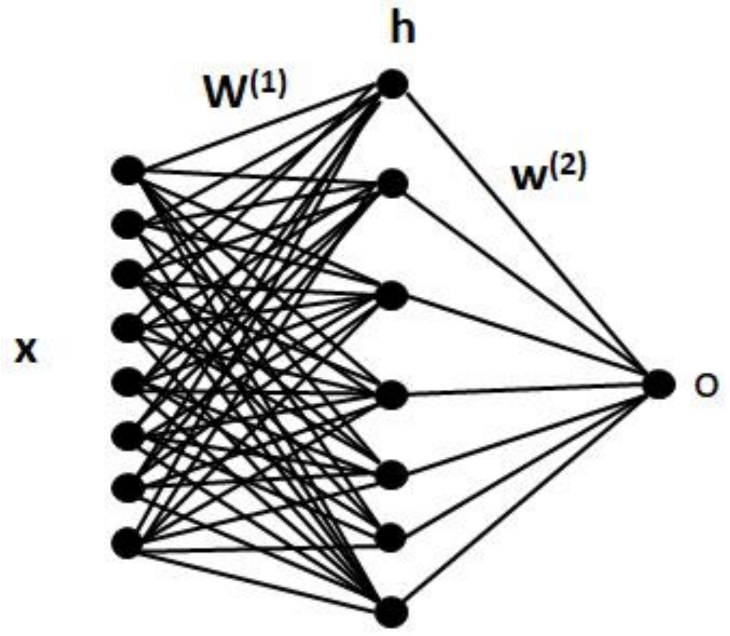
Each of the hidden neurons has an output equal to

$$h_k = f\left(\mathbf{W}_k^{(1)T}\mathbf{x}\right),\ k = 1,\ldots,K = 7$$

All h_k outputs are used to form the vector **h**

The output neuron has an output equal to

$$\begin{aligned} o &= f\left(\mathbf{w}^{(2)T}\mathbf{h}\right) = f\left(\sum_{k=1}^{K} w_k^{(2)} h_k\right) \\ &= f\left(\sum_{k=1}^{K} w_k^{(2)} f(\mathbf{W}_k^{(1)T}\mathbf{x})\right). \end{aligned}$$



Three-layer neural network

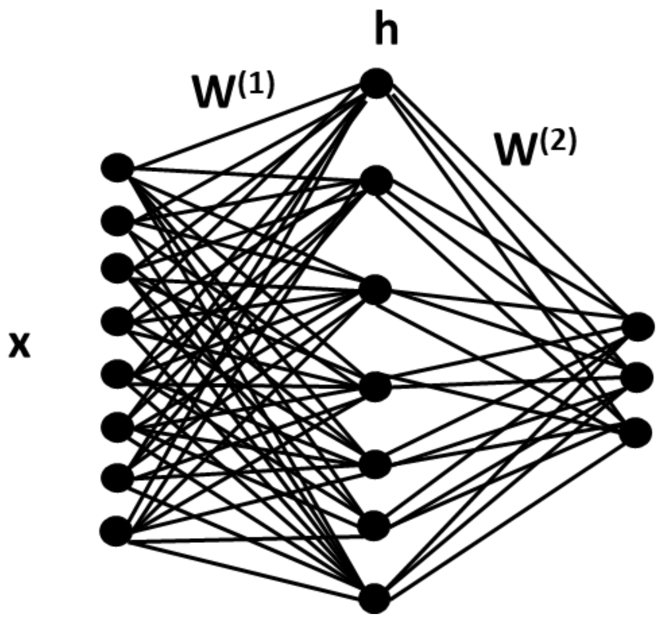
One case of particular interest is when the activation function of the output layer is linear, i.e. when f(**o**) = **o**. Then, we have:

$$o = \mathbf{w}^{(2)T}\mathbf{h} = \sum_{k=1}^{K} w_k^{(2)} h_k = \sum_{k=1}^{K} w_k^{(2)} f(\mathbf{W}_k^{(1)T}\mathbf{x})$$

How can we create a three-layer network solving a multi-class classification problem?

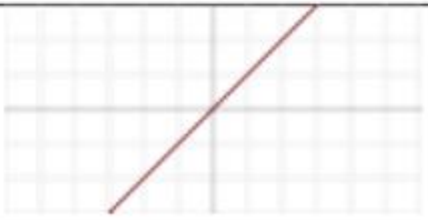
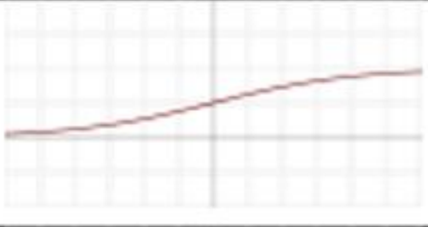
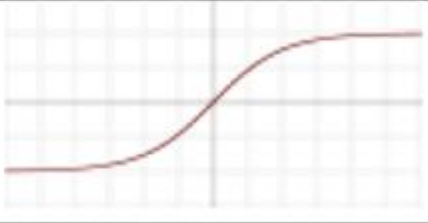
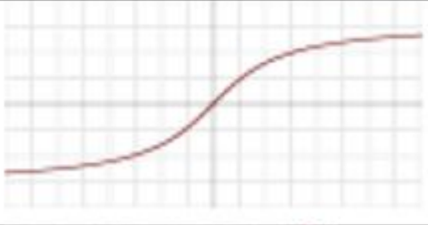
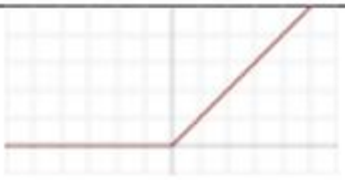
Three-layer neural network

In order to solve a K-class classification problem, the output layer is formed by K neurons



Activation functions

Name	Equation	Derivative
Identity	$f(x) = x$	$f'(x) = 1$
Logistic	$f(x) = \frac{1}{1+e^{-x}}$	$f'(x) = f(x)(1 - f(x))$
HyperbTan	$f(x) = \tanh(x) = \frac{2}{1+e^{-2x}} - 1$	$f'(x) = 1 - f(x)^2$
ArcTan	$f(x) = \tanh^{-1}(x)$	$f'(x) = \frac{1}{x^2+1}$
ReLU	$f(x) = \begin{cases} 0, \text{ if } x \leq 0 \\ x, \text{ if } x > 0 \end{cases}$	$f(x) = \begin{cases} 0, \text{ if } x \leq 0 \\ 1, \text{ if } x > 0 \end{cases}$
Softmax	$f_i(\mathbf{x}) = \frac{e^{x_k}}{\sum_{l=1}^{K} e^{x_l}}, \quad k = 1, \ldots, K$	$\frac{\vartheta f_i(\mathbf{x})}{\vartheta x_j} = f_i(\mathbf{x})(\delta_{ij} - f_j(\mathbf{x}))$

Name	Plot
Identity	
Logistic	
HypTan	
ArcTan	
ReLU	

Training multi-layer neural networks

After defining the parameters of a feedforward neural network (including the architecture and the activation functions), we need to determine good parameters for a specific classification task.

Training multi-layer neural networks

After defining the parameters of a feedforward neural network (including the architecture and the activation functions), we need to determine good parameters for a specific classification task.

We usually do not make assumptions regarding the distributions of the classes involved in the classification problem. Instead, we define the goodness of the network (and its parameters) based on the performance of the network on a set of data.

Training multi-layer neural networks

After defining the parameters of a feedforward neural network (including the architecture and the activation functions), we need to determine good parameters for a specific classification task.

We usually do not make assumptions regarding the distributions of the classes involved in the classification problem. Instead, we define the goodness of the network (and its parameters) based on the performance of the network on a set of data.

After initializing the network's parameters, we apply an iterative optimization process, which refines the network's parameters based on its output (usually called response) to a given input vector.

Training multi-layer neural networks

After defining the parameters of a feedforward neural network (including the architecture and the activation functions), we need to determine good parameters for a specific classification task.

We usually do not make assumptions regarding the distributions of the classes involved in the classification problem. Instead, we define the goodness of the network (and its parameters) based on the performance of the network on a set of data.

After initializing the network's parameters, we apply an iterative optimization process, which refines the network's parameters based on its output (usually called response) to a given input vector.

This is done by calculating the error for the specific vector and propagating (the changes required to reduce) the error from the output layer to the input layer.

The (error) Backpropagation algorithm

Let us consider as an error function the following criterion

$$\mathcal{J}(\mathbf{w}) = \frac{1}{2}\sum_{k=1}^{K}(t_k - o_k)^2 = \frac{1}{2}\|\mathbf{t} - \mathbf{o}\|_2^2$$

When the training error is measured using the MSE criterion

where t $\in \mathbb{R}^K$ is the target vector for a K-class classification problem.

The (error) Backpropagation algorithm

Let us consider as an error function the following criterion

$$\mathcal{J}(\mathbf{w}) = \frac{1}{2}\sum_{k=1}^{K}(t_k - o_k)^2 = \frac{1}{2}\|\mathbf{t} - \mathbf{o}\|_2^2$$

where $\mathbf{t} \in \mathbb{R}^K$ is the target vector for a K-class classification problem.

We usually set the values of t equal to:

- $t_k = 1$, if x belongs to class c_k
- $t_k = 0$, if x belongs to class $c_l \neq c_k$

Can we use target values equal to {-1,1}?

The (error) Backpropagation algorithm

Let us consider as an error function the following criterion

$$\mathcal{J}(\mathbf{w}) = \frac{1}{2}\sum_{k=1}^{K}(t_k - o_k)^2 = \frac{1}{2}\|\mathbf{t} - \mathbf{o}\|_2^2$$

where t $\in \mathbb{R}^K$ is the target vector for a K-class classification problem.

We usually set the values of t equal to:

- $t_k = 1$, if x belongs to class c_k
- $t_k = 0$, if x belongs to class $c_l \neq c_k$

Can we use target values equal to {-1,1}?

In principle, the answer is yes.

It depends on the activation function of the output layer!

The (error) Backpropagation algorithm

Let us consider as an error function the following criterion

$$\mathcal{J}(\mathbf{w}) = \frac{1}{2}\sum_{k=1}^{K}(t_k - o_k)^2 = \frac{1}{2}\|\mathbf{t} - \mathbf{o}\|_2^2$$

where t $\in \mathbb{R}^K$ is the target vector for a K-class classification problem.

The backpropagation update rule is based on gradient descent

$$\Delta\mathbf{w} = -\eta\frac{\vartheta\mathcal{J}}{\vartheta\mathbf{w}}$$

or in an element-wise form $$\Delta w_{ij} = -\eta\frac{\vartheta\mathcal{J}}{\vartheta w_{ij}}$$

The (error) Backpropagation algorithm

The backpropagation update rule is based on gradient descent

$$\Delta \mathbf{w} = -\eta \frac{\vartheta \mathcal{J}}{\vartheta \mathbf{w}}$$

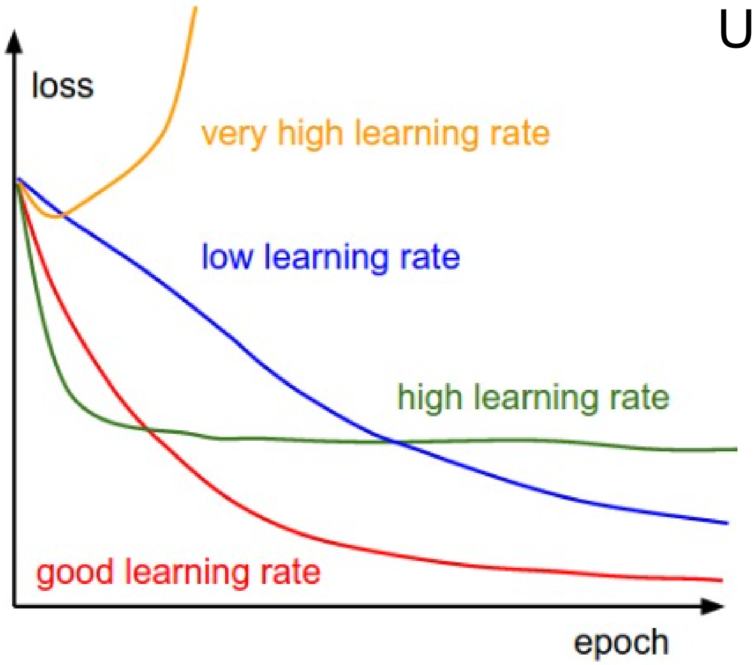
or in an element-wise form

$$\Delta w_{ij} = -\eta \frac{\vartheta \mathcal{J}}{\vartheta w_{ij}}$$

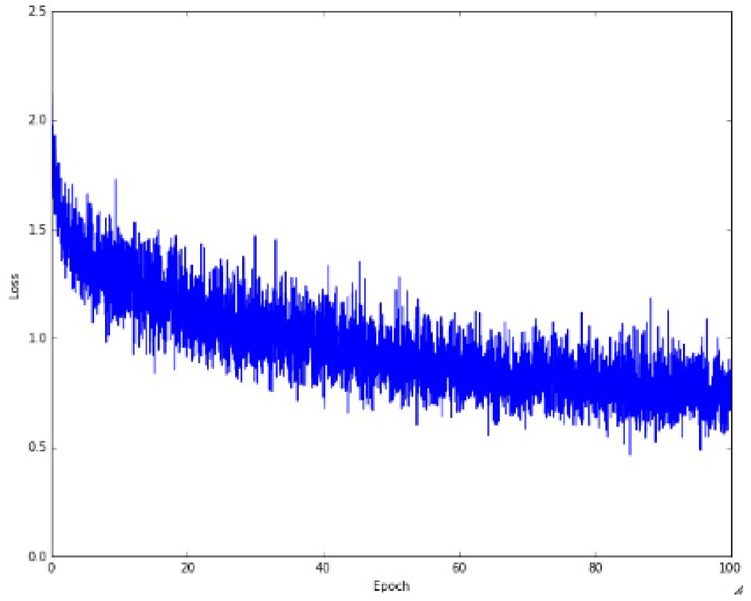
After calculating the gradient, the weights are updated as follows

$$\mathbf{w}(t+1) = \mathbf{w}(t) + \Delta \mathbf{w}.$$

The (error) Backpropagation algorithm



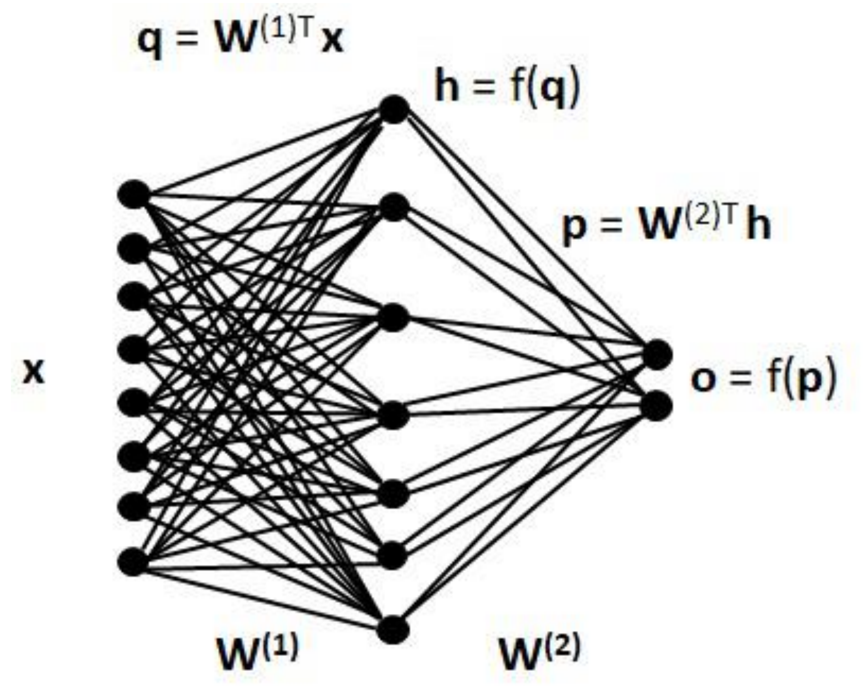
Usually the case in deep networks



Left: A cartoon depicting the effects of different learning rates. With low learning rates the improvements will be linear. With high learning rates they will start to look more exponential. Higher learning rates will decay the loss faster, but they get stuck at worse values of loss (green line). This is because there is too much "energy" in the optimization and the parameters are bouncing around chaotically, unable to settle in a nice spot in the optimization landscape. **Right:** An example of a typical loss function over time, while training a small network on CIFAR-10 dataset. This loss function looks reasonable (it might indicate a slightly too small learning rate based on its speed of decay, but it's hard to say), and also indicates that the batch size might be a little too low (since the cost is a little too noisy).

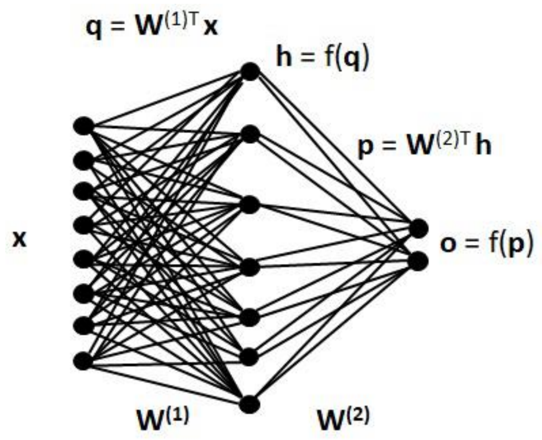
The (error) Backpropagation algorithm

Let's take as an example the following three-layer neural network



The (error) Backpropagation algorithm

Let's take as an example the following three-layer neural network



First we focus on the weights $\mathbf{W}^{(2)}$.

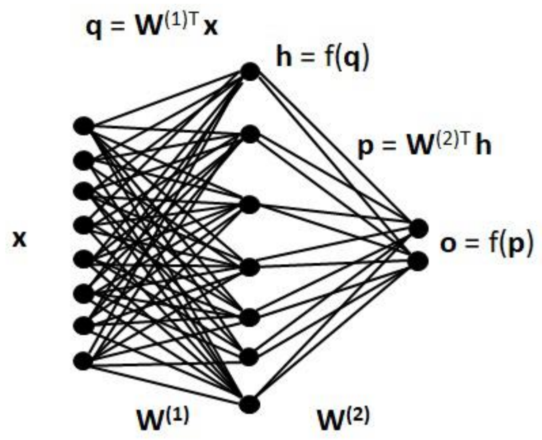
For the weights $\mathbf{W}_{jk}^{(2)}$ connecting the hidden neuron j with the output neuron k, we have

$$\frac{\vartheta \mathcal{J}}{\vartheta W_{jk}^{(2)}} = \frac{\vartheta \mathcal{J}}{\vartheta o_k} \frac{\vartheta o_k}{\vartheta p_k} \frac{\vartheta p_k}{\vartheta W_{jk}^{(2)}}$$

The (error) Backpropagation algorithm

For the weights $\mathbf{W}_{jk}^{(2)}$ connecting the hidden neuron j with the output neuron k, we have

$$\frac{\vartheta \mathcal{J}}{\vartheta W_{jk}^{(2)}} = \frac{\vartheta \mathcal{J}}{\vartheta o_k} \frac{\vartheta o_k}{\vartheta p_k} \frac{\vartheta p_k}{\vartheta W_{jk}^{(2)}}$$



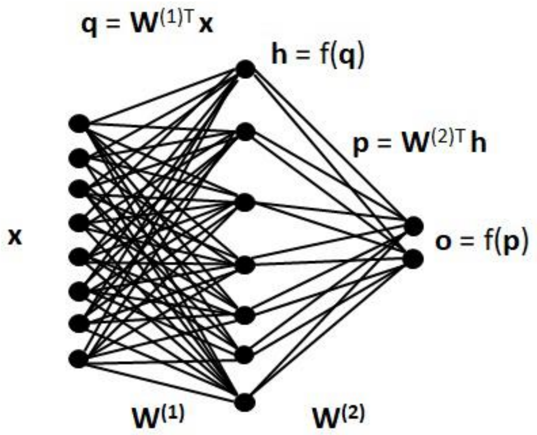
The last term is equal to

$$\frac{\vartheta p_k}{\vartheta W_{jk}^{(2)}} = \frac{\vartheta}{\vartheta W_{jk}^{(2)}} \left(\sum_{l=1}^{|\mathbf{h}|} W_{lk}^{(2)} h_l \right) = \frac{\vartheta}{\vartheta W_{jk}^{(2)}} \left(W_{jk}^{(2)} h_j \right) = h_j$$

The (error) Backpropagation algorithm

For the weights $\mathbf{W}_{jk}^{(2)}$ connecting the hidden neuron j with the output neuron k, we have

$$\frac{\vartheta \mathcal{J}}{\vartheta W_{jk}^{(2)}} = \frac{\vartheta \mathcal{J}}{\vartheta o_k} \frac{\vartheta o_k}{\vartheta p_k} \frac{\vartheta p_k}{\vartheta W_{jk}^{(2)}}$$


$\mathbf{q} = \mathbf{W}^{(1)T}\mathbf{x}$
$\mathbf{h} = f(\mathbf{q})$
$\mathbf{p} = \mathbf{W}^{(2)T}\mathbf{h}$
$\mathbf{x}$
$\mathbf{o} = f(\mathbf{p})$
$\mathbf{W}^{(1)}$
$\mathbf{W}^{(2)}$

The second term is equal to

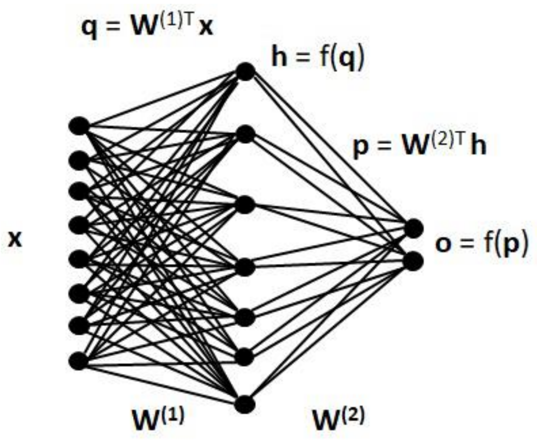
$$\frac{\vartheta o_k}{\vartheta p_k} = \frac{\vartheta}{\vartheta p_k} f(p_k) = f'(p_k)$$

This is why we need the activation function f(.) to be differentiable.

The (error) Backpropagation algorithm

For the weights $\mathbf{W}_{jk}^{(2)}$ connecting the hidden neuron j with the output neuron k, we have

$$\frac{\vartheta \mathcal{J}}{\vartheta W_{jk}^{(2)}} = \frac{\vartheta \mathcal{J}}{\vartheta o_k} \frac{\vartheta o_k}{\vartheta p_k} \frac{\vartheta p_k}{\vartheta W_{jk}^{(2)}}$$



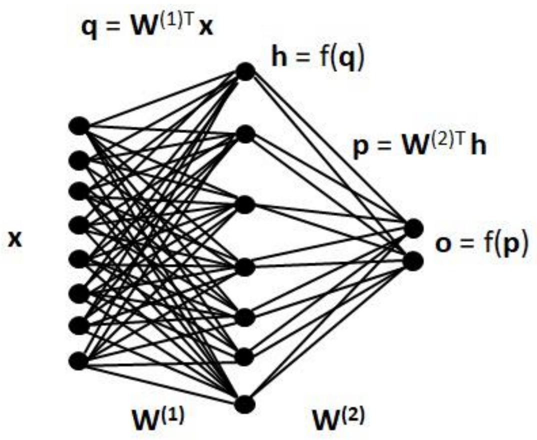
The first term is equal to

$$\frac{\vartheta \mathcal{J}}{\vartheta o_k} = \frac{\vartheta}{\vartheta o_k}\left(\frac{1}{2}(t_k - o_k)^2\right) = o_k - t_k$$

The (error) Backpropagation algorithm

For the weights $\mathbf{W}_{jk}^{(2)}$ connecting the hidden neuron j with the output neuron k, we have

$$\frac{\vartheta \mathcal{J}}{\vartheta W_{jk}^{(2)}} = \frac{\vartheta \mathcal{J}}{\vartheta o_k} \frac{\vartheta o_k}{\vartheta p_k} \frac{\vartheta p_k}{\vartheta W_{jk}^{(2)}}$$



Thus, we have

$$\frac{\vartheta \mathcal{J}}{\vartheta W_{jk}^{(2)}} = (o_k - t_k) f'(p_k) h_k$$

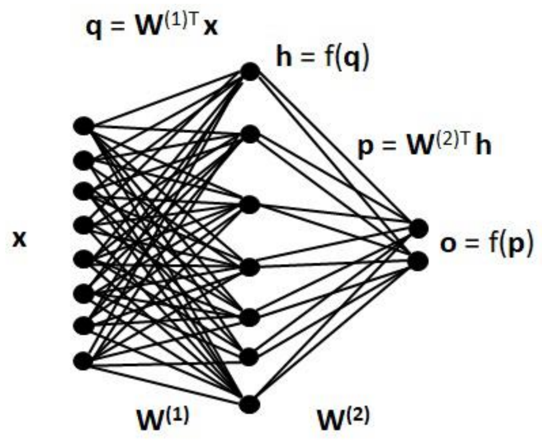
Is this easy to be calculated?

The (error) Backpropagation algorithm

Let's now focus on the weights $\mathbf{W}^{(1)}$.

For the weights $\mathbf{W}_{ij}^{(1)}$ connecting the input neuron i with the hidden neuron j, we have

$$\frac{\vartheta \mathcal{J}}{\vartheta W_{ij}^{(1)}} = \frac{\vartheta \mathcal{J}}{\vartheta h_j} \frac{\vartheta h_j}{\vartheta q_j} \frac{\vartheta q_j}{\vartheta W_{ij}^{(1)}}$$

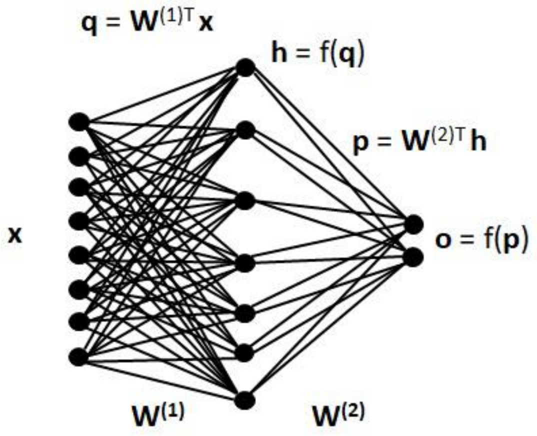


The (error) Backpropagation algorithm

Let's now focus on the weights $\mathbf{W}^{(1)}$.

For the weights $\mathbf{W}_{ij}^{(1)}$ connecting the input neuron i with the hidden neuron j, we have

$$\frac{\vartheta \mathcal{J}}{\vartheta W_{ij}^{(1)}} = \frac{\vartheta \mathcal{J}}{\vartheta h_j} \frac{\vartheta h_j}{\vartheta q_j} \frac{\vartheta q_j}{\vartheta W_{ij}^{(1)}}$$



The last term is equal to

$$\frac{\vartheta q_j}{\vartheta W_{ij}^{(1)}} = \frac{\vartheta}{\vartheta W_{ij}^{(1)}} \left(\sum_{l=1}^{D} W_{lj}^{(1)} x_l \right) = \frac{\vartheta}{\vartheta W_{ij}^{(1)}} \left(W_{ij}^{(1)} x_i \right) = x_i$$

The (error) Backpropagation algorithm

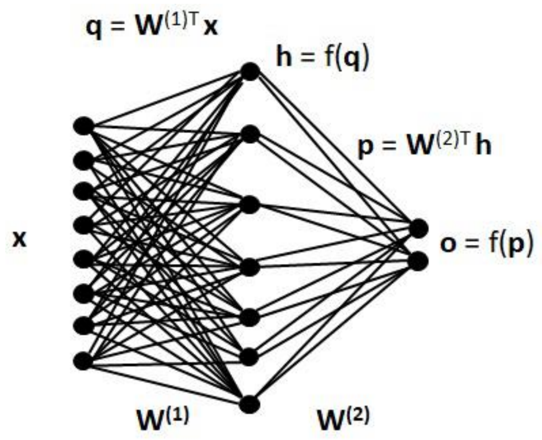
Let's now focus on the weights $\mathbf{W}^{(1)}$.

For the weights $\mathbf{W}_{ij}^{(1)}$ connecting the input neuron i with the hidden neuron j, we have

$$\frac{\vartheta \mathcal{J}}{\vartheta W_{ij}^{(1)}} = \frac{\vartheta \mathcal{J}}{\vartheta h_j} \frac{\vartheta h_j}{\vartheta q_j} \frac{\vartheta q_j}{\vartheta W_{ij}^{(1)}}$$

The second term is equal to

$$\frac{\vartheta h_j}{\vartheta q_j} = \frac{\vartheta}{\vartheta q_j} f(q_j) = f'(q_j)$$

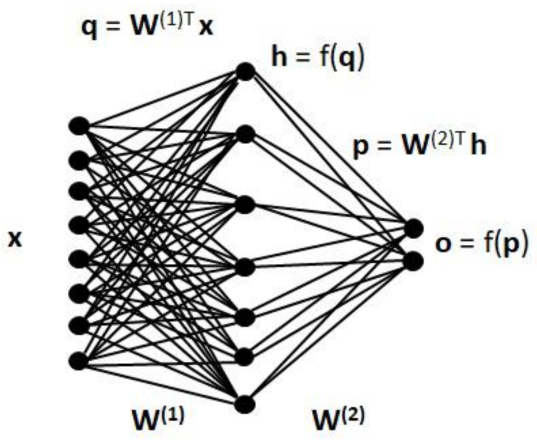


The (error) Backpropagation algorithm

Let's now focus on the weights $\mathbf{W}^{(1)}$.

For the weights $\mathbf{W}_{ij}^{(1)}$ connecting the input neuron i with the hidden neuron j, we have

$$\frac{\vartheta \mathcal{J}}{\vartheta W_{ij}^{(1)}} = \frac{\vartheta \mathcal{J}}{\vartheta h_j} \frac{\vartheta h_j}{\vartheta q_j} \frac{\vartheta q_j}{\vartheta W_{ij}^{(1)}}$$



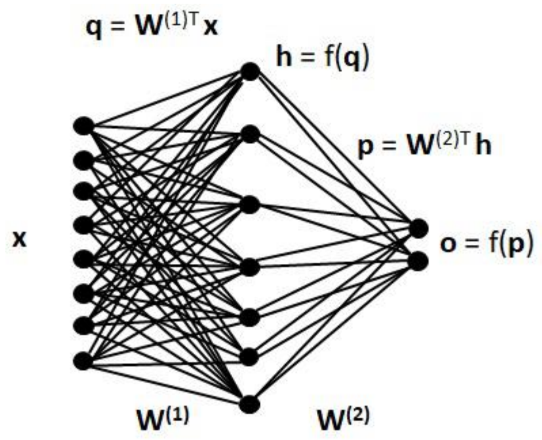
The calculation of the first term seems complicated, because neuron j belongs to the hidden layer.

The (error) Backpropagation algorithm

Let’s now focus on the weights $\mathbf{W}^{(1)}$.

For the weights $\mathbf{W}_{ij}^{(1)}$ connecting the input neuron i with the hidden neuron j, we have

$$\frac{\vartheta \mathcal{J}}{\vartheta W_{ij}^{(1)}} = \frac{\vartheta \mathcal{J}}{\vartheta h_j} \frac{\vartheta h_j}{\vartheta q_j} \frac{\vartheta q_j}{\vartheta W_{ij}^{(1)}}$$



The first term is equal to

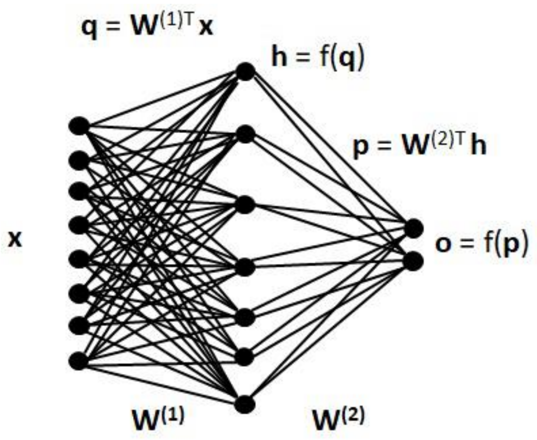
$$\frac{\vartheta \mathcal{J}}{\vartheta h_j} = \sum_{k=1}^{K} \left(\frac{\vartheta \mathcal{J}}{\vartheta o_k} \frac{\vartheta o_k}{\vartheta p_k} \frac{\vartheta p_k}{\vartheta h_j} \right) = \sum_{k=1}^{K} \left(\frac{\vartheta \mathcal{J}}{\vartheta o_k} \frac{\vartheta o_k}{\vartheta p_k} W_{jk}^{(2)} \right)$$

The (error) Backpropagation algorithm

Let's now focus on the weights $\mathbf{W}^{(1)}$.

The first term is equal to

$$\frac{\vartheta \mathcal{J}}{\vartheta h_j} = \sum_{k=1}^{K} \left(\frac{\vartheta \mathcal{J}}{\vartheta o_k} \boxed{\frac{\vartheta o_k}{\vartheta p_k}} \frac{\vartheta p_k}{\vartheta h_j} \right) = \sum_{k=1}^{K} \left(\frac{\vartheta \mathcal{J}}{\vartheta o_k} \frac{\vartheta o_k}{\vartheta p_k} W_{jk}^{(2)} \right)$$



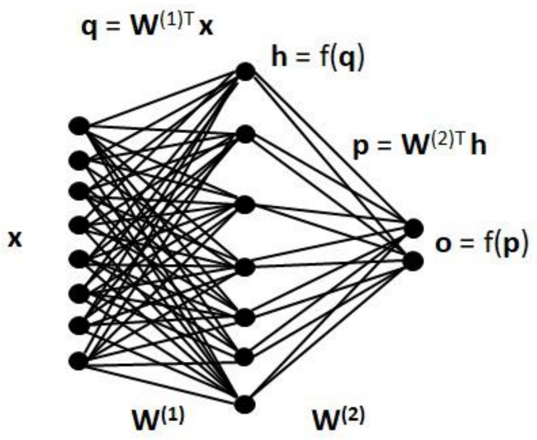
$$\frac{\vartheta o_k}{\vartheta p_k} = \frac{\vartheta}{\vartheta p_k} f(p_k) = f'(p_k)$$

The (error) Backpropagation algorithm

Let's now focus on the weights $\mathbf{W}^{(1)}$.

The first term is equal to

$$\frac{\vartheta \mathcal{J}}{\vartheta h_j} = \sum_{k=1}^{K} \left(\boxed{\frac{\vartheta \mathcal{J}}{\vartheta o_k}} \frac{\vartheta o_k}{\vartheta p_k} \frac{\vartheta p_k}{\vartheta h_j} \right) = \sum_{k=1}^{K} \left(\frac{\vartheta \mathcal{J}}{\vartheta o_k} \frac{\vartheta o_k}{\vartheta p_k} W_{jk}^{(2)} \right)$$


$\mathbf{q} = \mathbf{W}^{(1)T}\mathbf{x}$
$\mathbf{h} = f(\mathbf{q})$
$\mathbf{p} = \mathbf{W}^{(2)T}\mathbf{h}$
$\mathbf{x}$
$\mathbf{o} = f(\mathbf{p})$
$\mathbf{W}^{(1)}$
$\mathbf{W}^{(2)}$

$$\frac{\vartheta \mathcal{J}}{\vartheta o_k} = \frac{\vartheta}{\vartheta o_k} \left(\frac{1}{2} (t_k - o_k)^2 \right) = o_k - t_k$$

The (error) Backpropagation algorithm

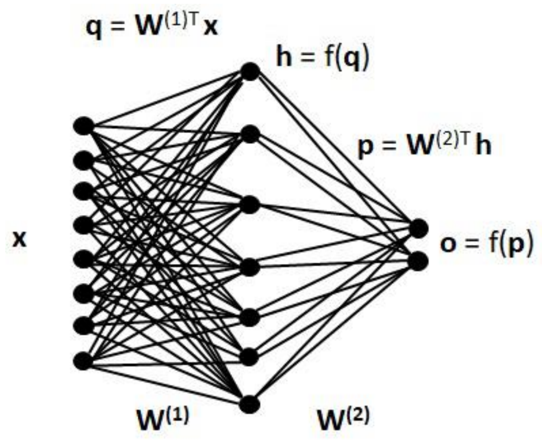
Let's now focus on the weights $\mathbf{W}^{(1)}$.

Thus the derivative

$$\frac{\vartheta \mathcal{J}}{\vartheta W_{ij}^{(1)}} = \frac{\vartheta \mathcal{J}}{\vartheta h_j} \frac{\vartheta h_j}{\vartheta q_j} \frac{\vartheta q_j}{\vartheta W_{ij}^{(1)}}$$

becomes

$$\frac{\vartheta \mathcal{J}}{\vartheta W_{ij}^{(1)}} = \sum_{k=1}^{K} \left(\frac{\vartheta \mathcal{J}}{\vartheta o_k} \frac{\vartheta o_k}{\vartheta p_k} W_{jk}^{(2)} \right) f'(q_j) x_i$$



The (error) Backpropagation algorithm

In the general case where the neural network if formed by more than three layers, the weight connecting the i-th neuron in layer l with the j-th neuron in layer l+q is updated using the gradient

$$\frac{\vartheta \mathcal{J}}{\vartheta W_{ij}^{(l)}} = \delta_j^l h_i^l$$

where $\delta_j^l = \left(\sum_{q \in \mathcal{L}_{l+1}} \delta_q^{l+1} W_{jq}^{l+1} \right) f'(h_j^{l+1})$

and the gradient is used to update the weight using $W_{ij}^{(l)} = W_{ij}^{(l)} - \eta \frac{\vartheta \mathcal{J}}{\vartheta W_{ij}^{(l)}}$

The (error) Backpropagation algorithm

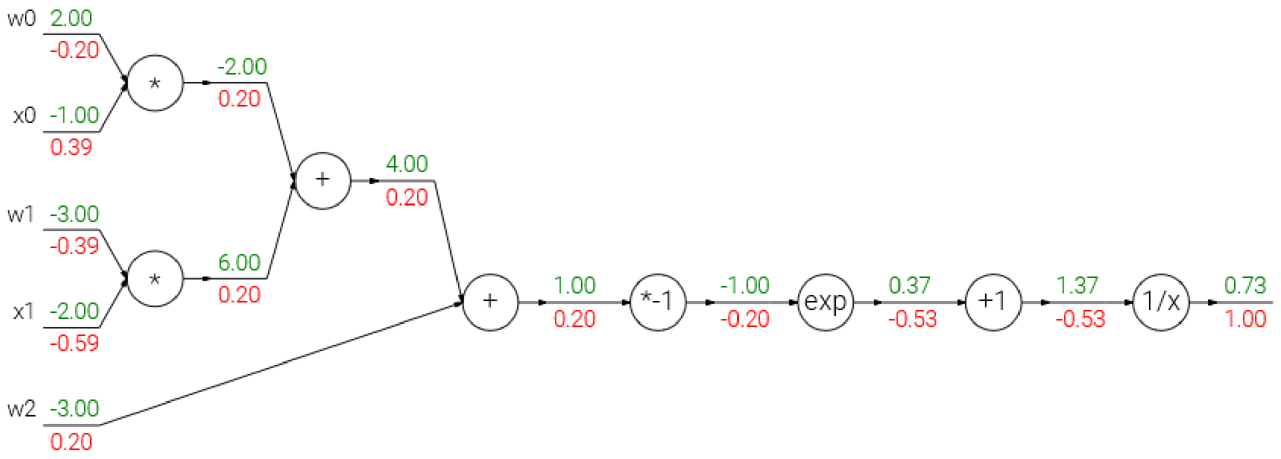
Example using the sigmoid function: $f(w, x) = \frac{1}{1 + e^{-(w_0 x_0 + w_1 x_1 + w_2)}}$

$$f(x) = \frac{1}{x} \quad \rightarrow \quad \frac{df}{dx} = -1/x^2$$

$$f_c(x) = c + x \quad \rightarrow \quad \frac{df}{dx} = 1$$

$$f(x) = e^x \quad \rightarrow \quad \frac{df}{dx} = e^x$$

$$f_a(x) = ax \quad \rightarrow \quad \frac{df}{dx} = a$$



Practical matters

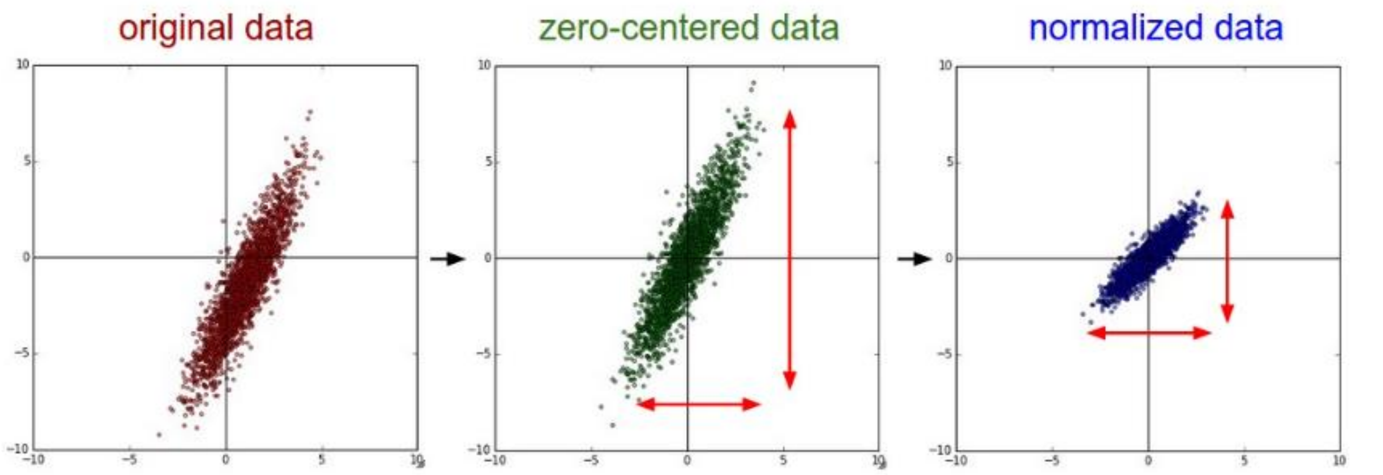
There are several issues that need to be appropriately handled when training a multilayer neural network:

- What type of architecture, how many layers, how many neurons?

Practical matters

There are several issues that need to be appropriately handled when training a multilayer neural network:

- What type of architecture, how many layers, how many neurons?
- Data pre-processing?

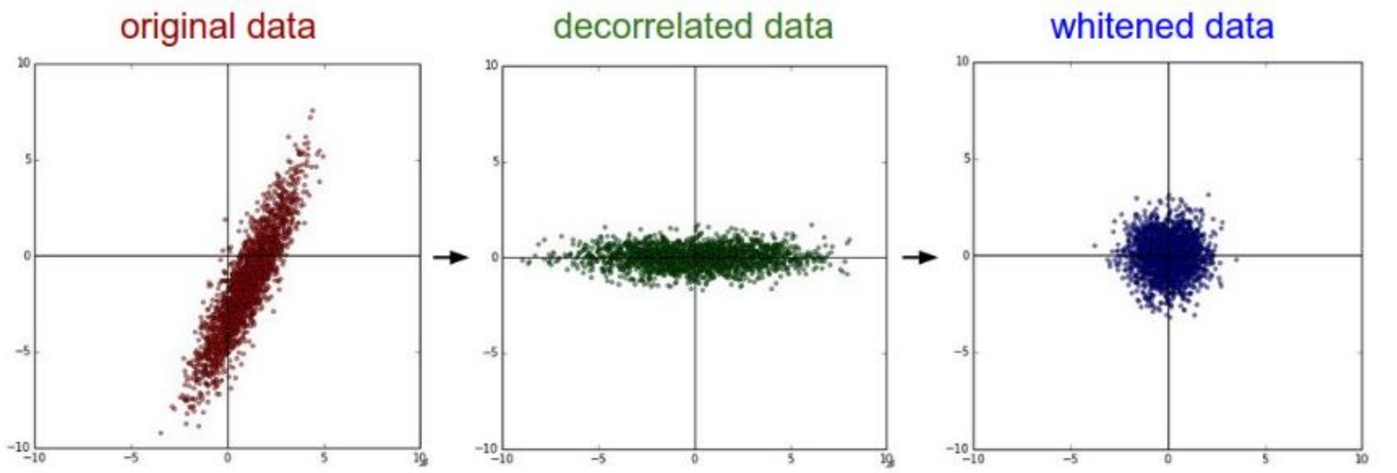


Common data preprocessing pipeline. **Left**: Original toy, 2-dimensional input data. **Middle**: The data is zero-centered by subtracting the mean in each dimension. The data cloud is now centered around the origin. **Right**: Each dimension is additionally scaled by its standard deviation. The red lines indicate the extent of the data - they are of unequal length in the middle, but of equal length on the right.

Practical matters

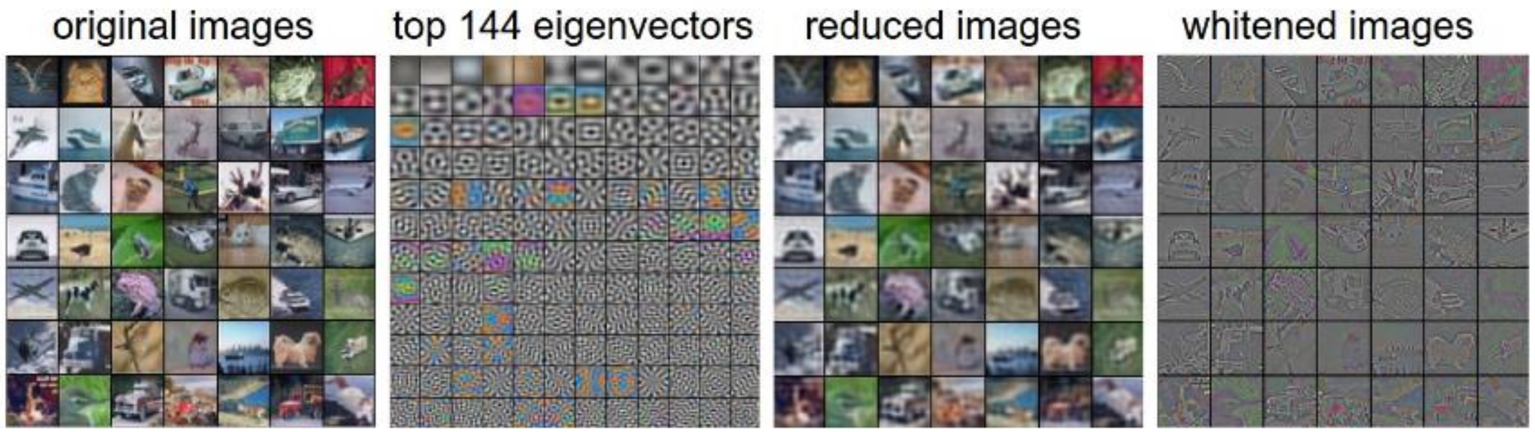
There are several issues that need to be appropriately handled when training a multilayer neural network:

- What type of architecture, how many layers, how many neurons?
- Data pre-processing?



PCA / Whitening. **Left**: Original toy, 2-dimensional input data. **Middle**: After performing PCA. The data is centered at zero and then rotated into the eigenbasis of the data covariance matrix. This decorrelates the data (the covariance matrix becomes diagonal). **Right**: Each dimension is additionally scaled by the eigenvalues, transforming the data covariance matrix into the identity matrix. Geometrically, this corresponds to stretching and squeezing the data into an isotropic gaussian blob.

Practical matters



Left: An example set of 49 images. **2nd from Left:** The top 144 out of 3072 eigenvectors. The top eigenvectors account for most of the variance in the data, and we can see that they correspond to lower frequencies in the images. **2nd from Right:** The 49 images reduced with PCA, using the 144 eigenvectors shown here. That is, instead of expressing every image as a 3072-dimensional vector where each element is the brightness of a particular pixel at some location and channel, every image above is only represented with a 144-dimensional vector, where each element measures how much of each eigenvector adds up to make up the image. In order to visualize what image information has been retained in the 144 numbers, we must rotate back into the "pixel" basis of 3072 numbers. Since U is a rotation, this can be achieved by multiplying by U.transpose()[:144,:], and then visualizing the resulting 3072 numbers as the image. You can see that the images are slightly blurrier, reflecting the fact that the top eigenvectors capture lower frequencies. However, most of the information is still preserved. **Right:** Visualization of the "white" representation, where the variance along every one of the 144 dimensions is squashed to equal length. Here, the whitened 144 numbers are rotated back to image pixel basis by multiplying by U.transpose()[:144,:]. The lower frequencies (which accounted for most variance) are now negligible, while the higher frequencies (which account for relatively little variance originally) become exaggerated.

Practical matters

There are several issues that need to be appropriately handled when training a multilayer neural network:

- What type of architecture, how many layers, how many neurons?
- Data pre-processing?
- Weights initialization?
  - Usually random initialization within a uniform distribution [-a,a]

Practical matters

There are several issues that need to be appropriately handled when training a multilayer neural network:

- What type of architecture, how many layers, how many neurons?
- Data pre-processing?
- Weights initialization?
- What learning rate?
  - Usually a small number 10^{-q}, q being -1 up to -5
  - linear decrease over the course of learning

Practical matters

There are several issues that need to be appropriately handled when training a multilayer neural network:

- What type of architecture, how many layers, how many neurons?
- Data pre-processing?
- Weights initialization?
- What learning rate?

Learning with momentum:

$$W_{ij}^{(l)}(t+1) = W_{ij}^{(l)}(t) - \eta \left(\frac{\vartheta \mathcal{J}}{\vartheta W_{ij}^{(l)}(t)} + \alpha \frac{\vartheta \mathcal{J}}{\vartheta W_{ij}^{(l)}(t-1)} \right)$$

Practical matters

The above type of update in BP is known under the name Vanilla update. Other types:

- Update with momentum:
  motivated from a physical perspective of the optimization problem. The optimization problem is interpreted as climbing down from a hill. In this case the force pushing the object downward is affected by its momentum and the local variance of the forces at a particular position
- Nesterov momentum:
  Using the same ideas as above, but calculating the gradient at the predicted position of the object

Practical matters

Second order methods:

- Newton's method:
  Update the parameters using the second order derivative information stored in the Hessian matrix Hf(w):

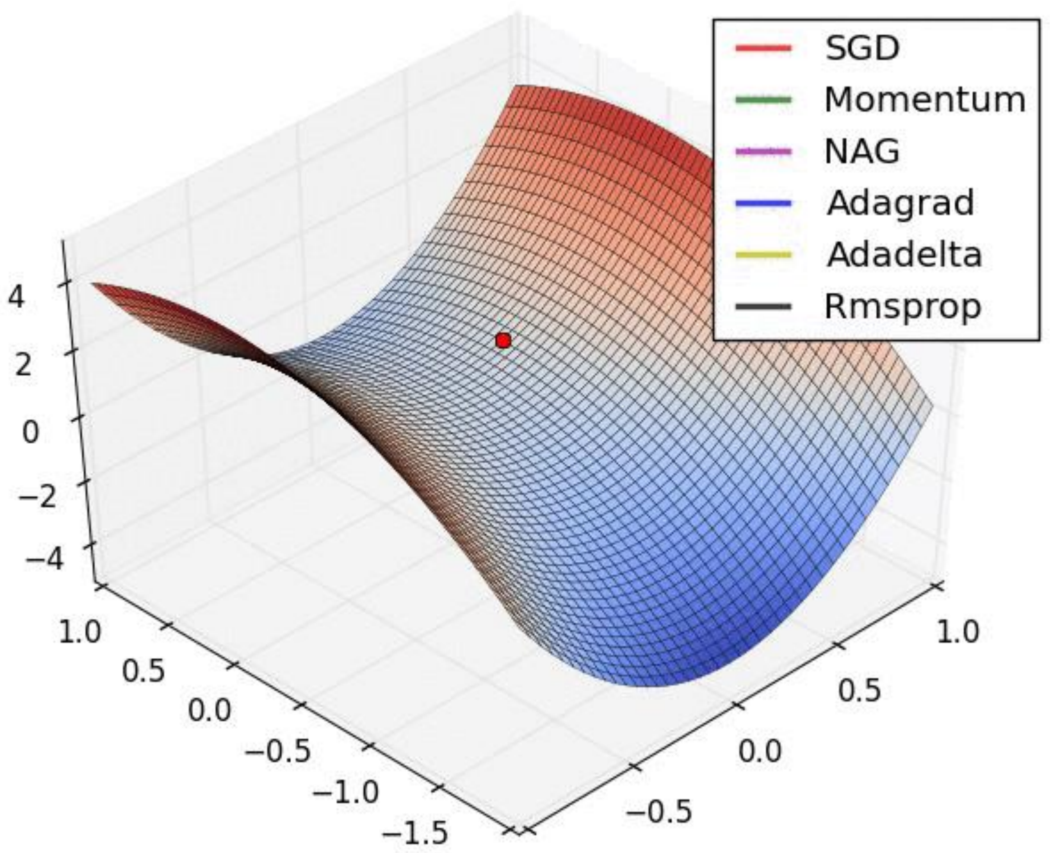
$$w \leftarrow w - [Hf(w)]^{-1} \nabla f(w)$$

Adaptive (per-parameter) update methods:

- Adagrad, RMSprop, Adam, etc.
- They use local information of the gradient over the last updates to change the learning rate for each parameter

Practical matters

Example of the behavior of different update rules at a saddle point



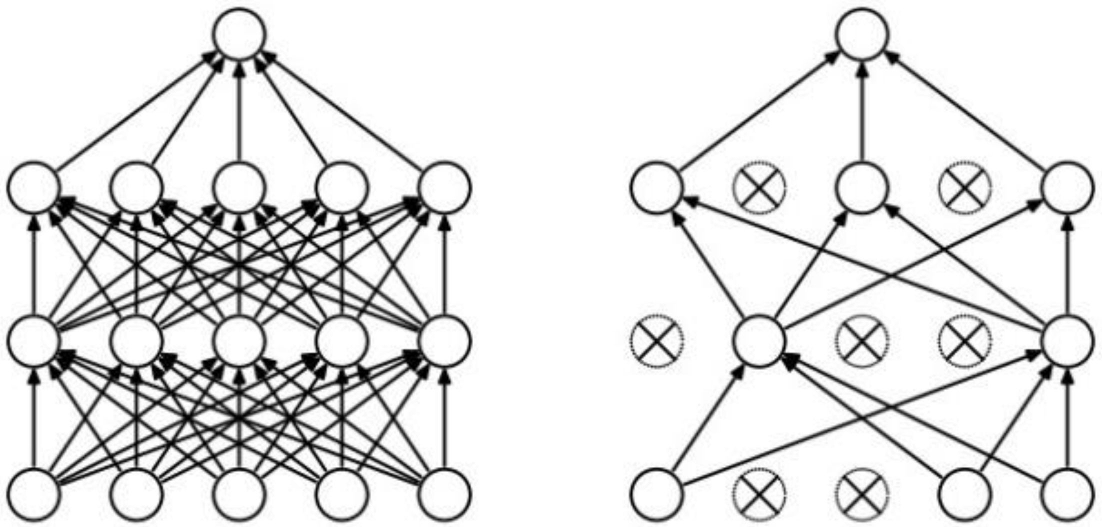
Practical matters

There are several issues that need to be appropriately handled when training a multilayer neural network:

- What type of architecture, how many layers, how many neurons?
- Data pre-processing?
- Weights initialization?
- What learning rate?
- DropOut?
  - A probabilistic way to 'augment' the training set during online training
  - At each training iteration, some of the weights (randomly chosen) are discarded and the response is based on a weighted version of the remaining weights

Practical matters

DropOut



(a) Standard Neural Net (b) After applying dropout.

Figure taken from the Dropout paper that illustrates the idea. During training, Dropout can be interpreted as sampling a Neural Network within the full Neural Network, and only updating the parameters of the sampled network based on the input data. (However, the exponential number of possible sampled networks are not independent because they share the parameters.) During testing there is no dropout applied, with the interpretation of evaluating an averaged prediction across the exponentially-sized ensemble of all sub-networks (more about ensembles in the next section).

Practical matters

There are several issues that need to be appropriately handled when training a multilayer neural network:

- What type of architecture, how many layers, how many neurons?
- Data pre-processing?
- Weights initialization?
- What learning rate?
- DropOut?
- Loss function?
  - In most recent applications MSE is replaced by the softmax function:

$$p_j = \frac{e^{o_j}}{\sum_k e^{o_k}} \quad \Rightarrow \quad P(y = j \mid \mathbf{x}) = \frac{e^{\mathbf{x}^\top \mathbf{w}_j}}{\sum_{k=1}^{K} e^{\mathbf{x}^\top \mathbf{w}_k}}$$

and the cross-entropy loss function is used for training $L = -\sum_j y_j \log p_j$

Convolutional Neural Networks

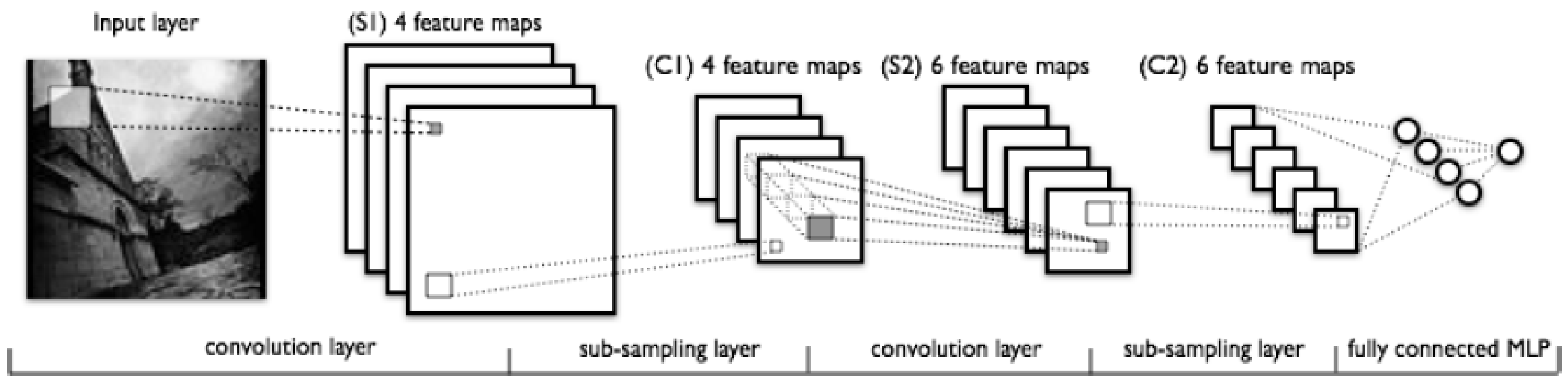
All neural networks we have seen until now take vectors as inputs.

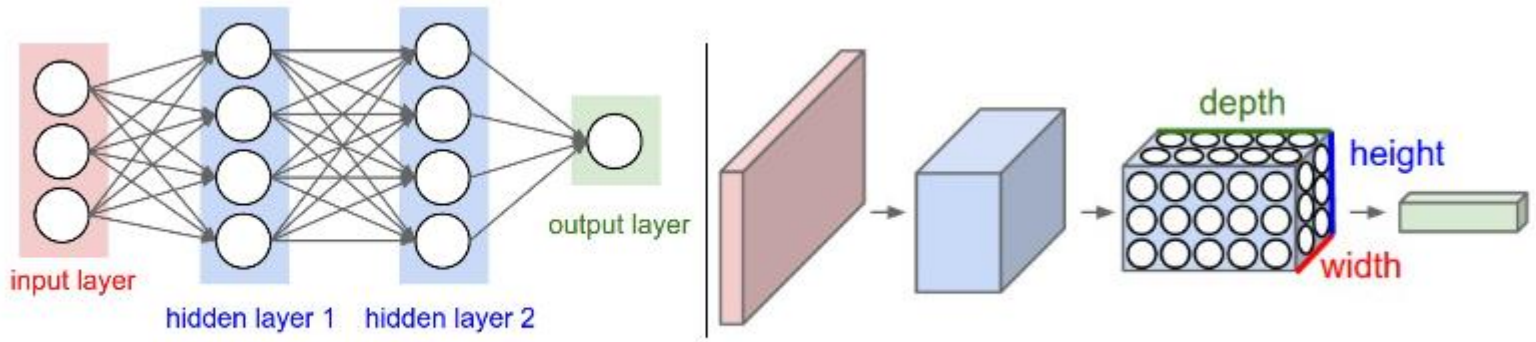
That is, in order to use them in a classification problem, we need to find a way to convert the input data (e.g. an image) to a vector.

Such data representations are usually referred to as hand-crafted data representations.

CNNs can (usually) lead to higher performances since they take as input the raw (image) data and optimize both the representation and classification parameters in a combined manner.

Convolutional Neural Networks



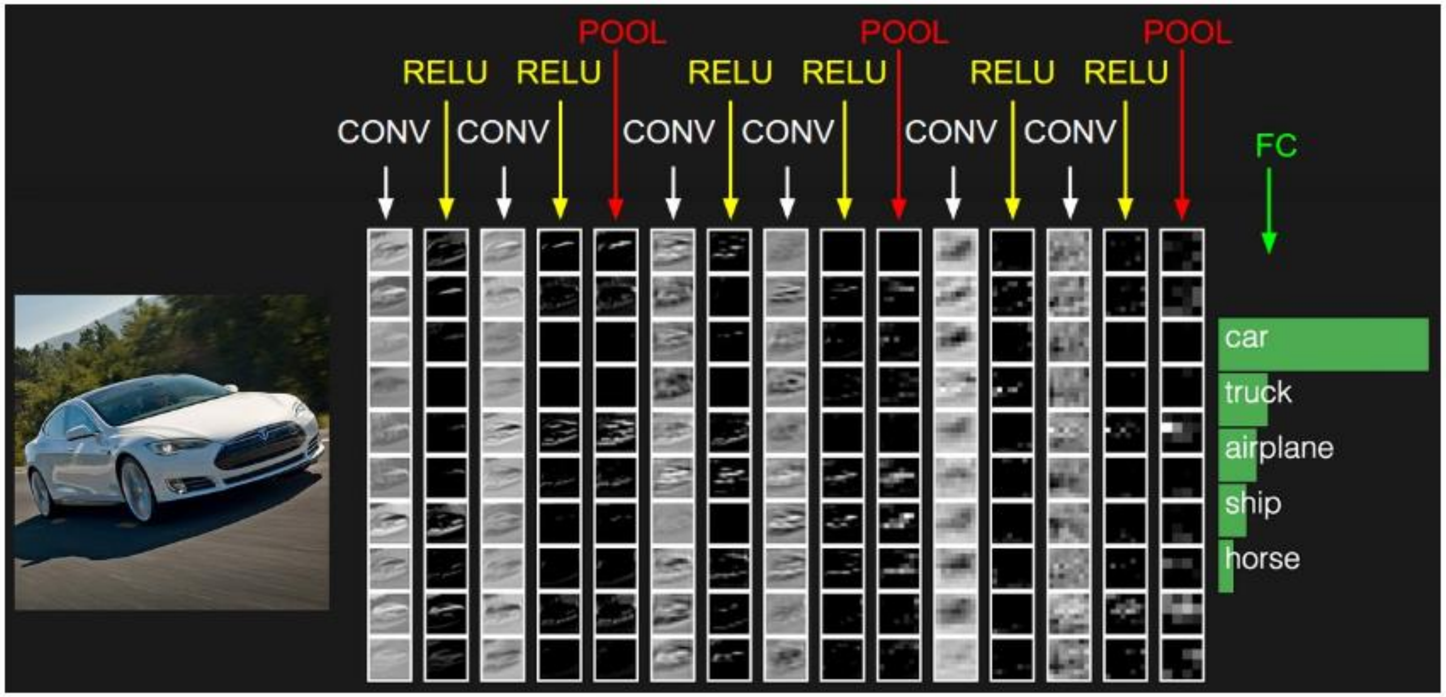


Left: A regular 3-layer Neural Network. Right: A ConvNet arranges its neurons in three dimensions (width, height, depth), as visualized in one of the layers. Every layer of a ConvNet transforms the 3D input volume to a 3D output volume of neuron activations. In this example, the red input layer holds the image, so its width and height would be the dimensions of the image, and the depth would be 3 (Red, Green, Blue channels).

Convolutional Neural Networks

A CNN is formed by three types of layers:

- Convolutional layers
- Subsampling/Pooling layers
- Fully-connected layers



Convolutional Neural Networks

Convolutional layer

It is formed by multiple neurons, each neuron equipped with:

- a filter with (h x w) elements
- a nonlinear activation function

The filter of each neuron 'scans' the image/tensor I_{m-1} and at each position it calculates

$$I_m(y,x) = f\left(\sum_{d=1}^{D}\sum_{i=0}^{1}\sum_{j=0}^{1} w_{ij}^{0d} I_{m-1}^{d}(y+1, x+j)\right)$$

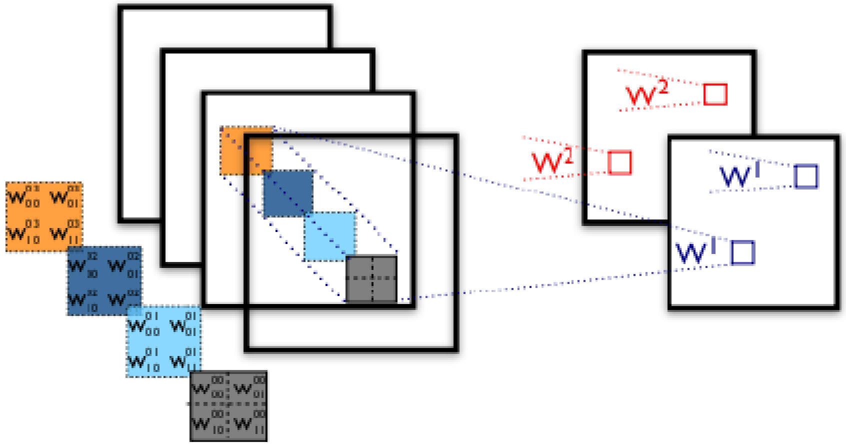
Doing the above for all positions in the image I_{m-1} leads to the creation of a new image I_m.

Convolutional Neural Networks

Convolutional layer

The filter of each neuron 'scans' the image/tensor I_{m-1} and at each position it calculates

$$I_m(y,x) = f\left(\sum_{d=1}^{D}\sum_{i=0}^{1}\sum_{j=0}^{1} w_{ij}^{0d} I_{m-1}^{d}(y+1, x+j)\right)$$



Convolutional Neural Networks

Convolutional layer

The filter of each neuron ‘scans’ the image/tensor I_{m-1} and at each position it calculates

$$I_m(y,x) = f\left(\sum_{d=1}^{D}\sum_{i=0}^{1}\sum_{j=0}^{1} w_{ij}^{0d} I_{m-1}^{d}(y+1, x+j)\right)$$

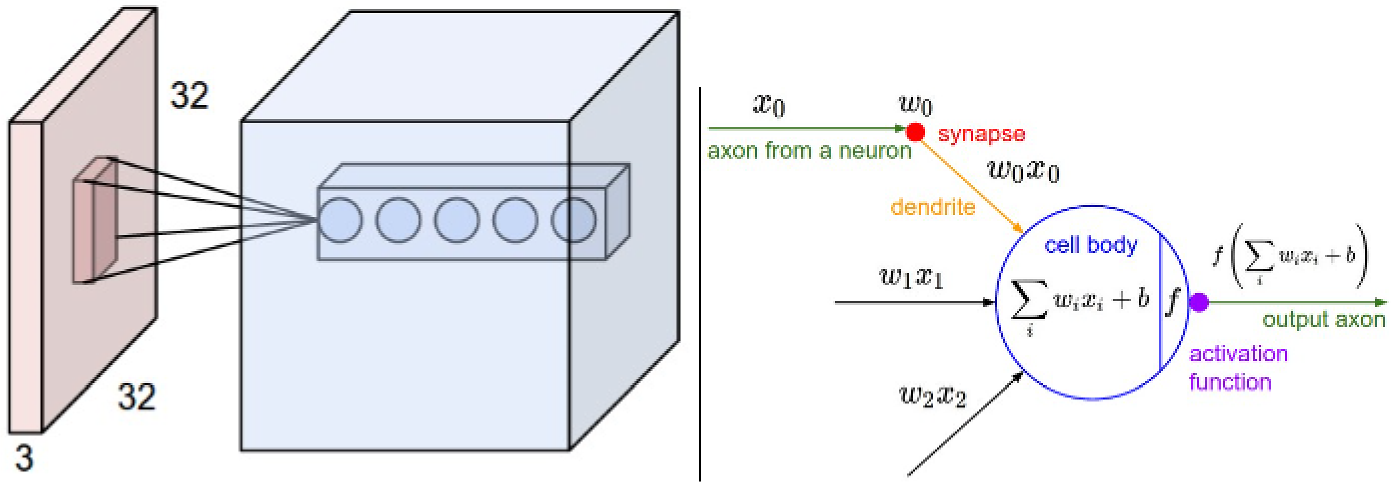
Doing the above for all positions in the image I_{m-1} leads to the creation of a new image I_m.

The new image I_m is called feature map.

Multiple neurons in a layer create a new feature map each.

Convolutional Neural Networks

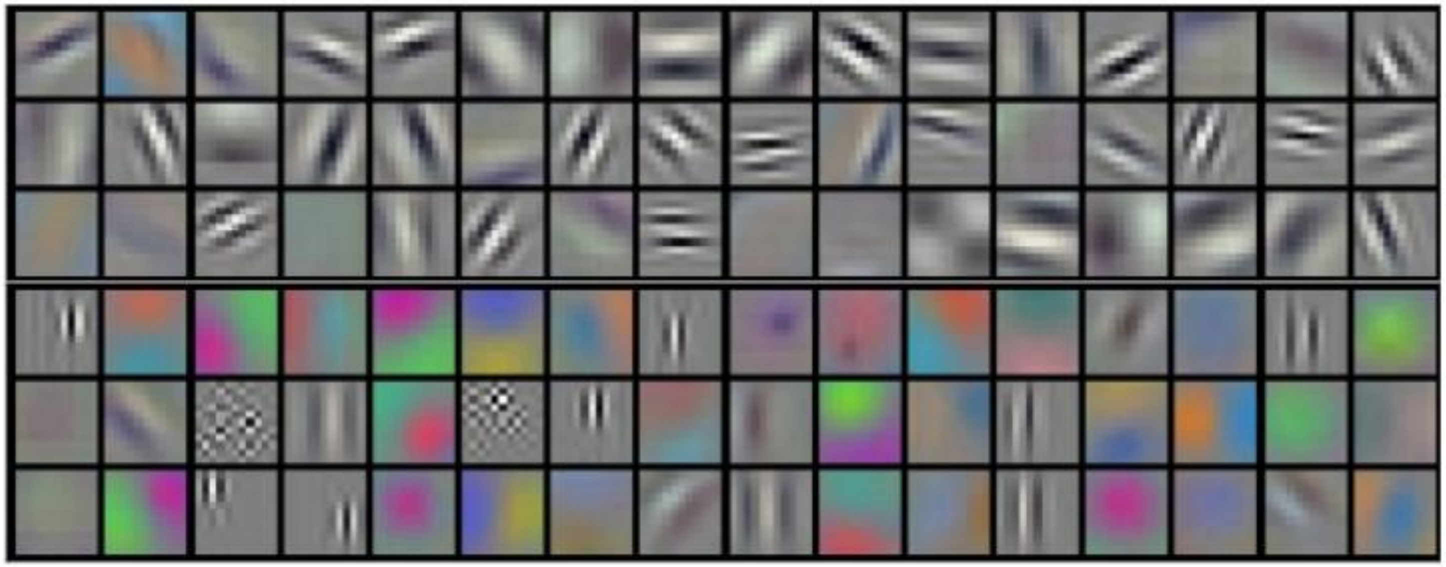
Convolutional layer → calculation of convolutions



Left: An example input volume in red (e.g. a 32x32x3 CIFAR-10 image), and an example volume of neurons in the first Convolutional layer. Each neuron in the convolutional layer is connected only to a local region in the input volume spatially, but to the full depth (i.e. all color channels). Note, there are multiple neurons (5 in this example) along the depth, all looking at the same region in the input - see discussion of depth columns in text below. **Right:** The neurons from the Neural Network chapter remain unchanged: They still compute a dot product of their weights with the input followed by a non-linearity, but their connectivity is now restricted to be local spatially.

Convolutional Neural Networks

Convolutional layer



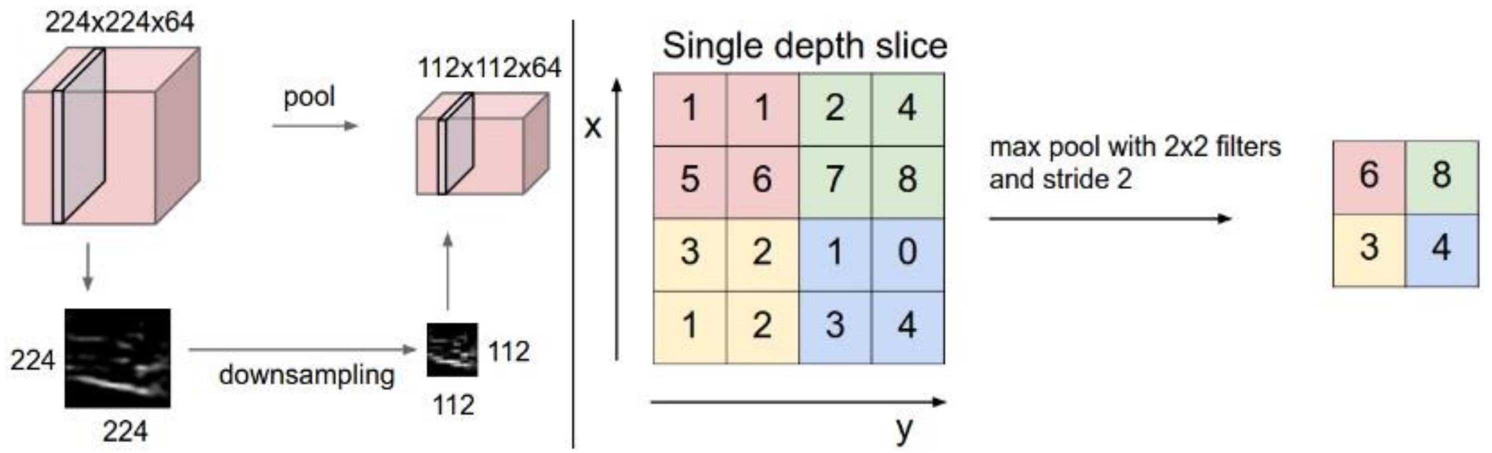
Example filters learned by Krizhevsky et al. Each of the 96 filters shown here is of size [11x11x3], and each one is shared by the 55*55 neurons in one depth slice. Notice that the parameter sharing assumption is relatively reasonable: If detecting a horizontal edge is important at some location in the image, it should intuitively be useful at some other location as well due to the translationally-invariant structure of images. There is therefore no need to relearn to detect a horizontal edge at every one of the 55*55 distinct locations in the Conv layer output volume.

Convolutional Neural Networks

Subsampling/Pooling layer

It resizes the feature maps of layer l to create a new layer l+1:

- average/max/rand pooling
- Sometimes this layer is removed and is replaced by using a higher convolutional step stride

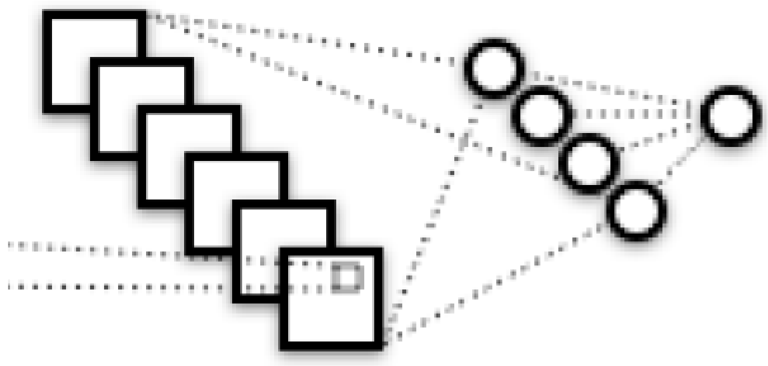


Pooling layer downsamples the volume spatially, independently in each depth slice of the input volume. **Left:** In this example, the input volume of size [224x224x64] is pooled with filter size 2, stride 2 into output volume of size [112x112x64]. Notice that the volume depth is preserved. **Right:** The most common downsampling operation is max, giving rise to **max pooling**, here shown with a stride of 2. That is, each max is taken over 4 numbers (little 2x2 square).

Convolutional Neural Networks

Fully-connected layer

Standard vector-based neural network layer (as discussed above)



Training protocols

When determining some parameters of an algorithm that need to be selected by the user (these are called hyper-parameters), we need to find a way to compare the goodness of models.

Training protocols

When determining some parameters of an algorithm that need to be selected by the user (these are called hyper-parameters), we need to find a way to compare the goodness of models.

Comparing the goodness of different models using the performance on the training data might lead to overfitting.

How to select the hyper-parameters?

Training protocols

There are two ways to select the hyper-parameters:

- Evaluate the goodness on a separate set for which we know the truth (e.g. labels). This set is called validation set. For example, we can split the training set in 70%-30% splits

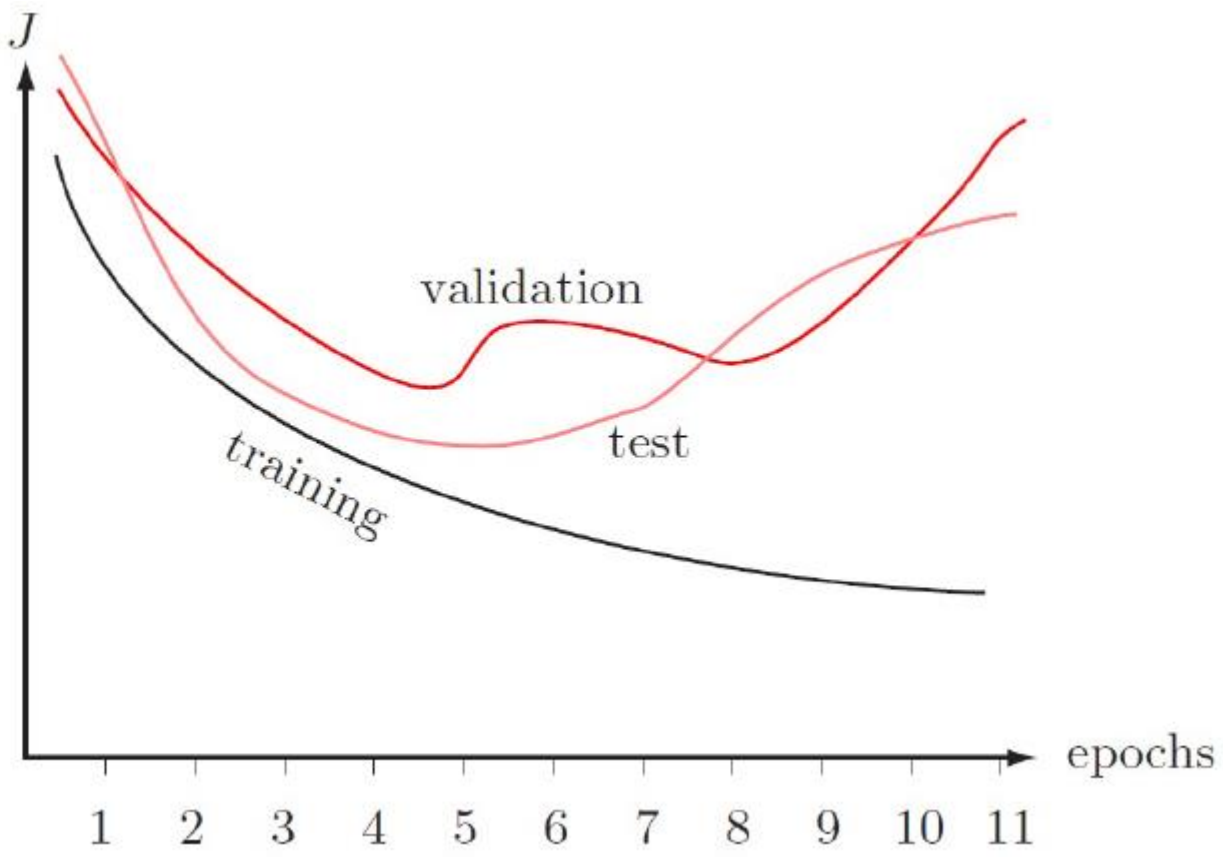
Training protocols

There are two ways to select the hyper-parameters:

- Evaluate the goodness on a separate set for which we know the truth (e.g. labels). This set is called validation set. For example, we can split the training set in 70%-30% splits
- Apply cross-validation:
  - We split the training set in K (mutually exclusive) subsets (K-fold cross validation).
  - For a set of hyper-parameter values, we train the model K times using K-1 sets and we test it with the remaining set. Then, we calculate the average performance over all K experiments and the corresponding standard deviation.
  - We select the hyper-parameter values providing the best overall performance
- Apply Leave-One-Out cross validation

Training protocols

Validation error as criterion for stopping training.


J
validation
test
training
epochs