UNIT 5

Artificial Neural Networks

JOHANNES KEPLER
UNIVERSITY LINZ

BIOINF

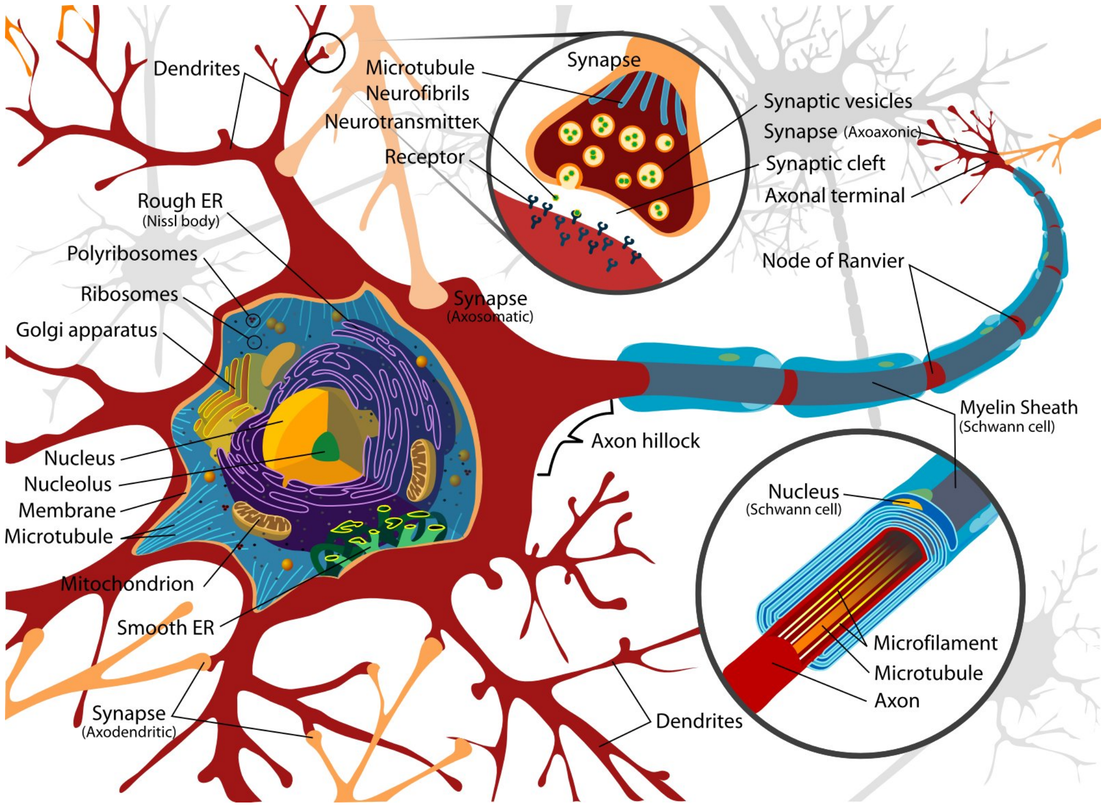
INTRODUCTION

- The most powerful and most versatile “learning machine” is still the human brain.
- Starting in the 1940ies, ideas for creating “intelligent‘” systems by mimicking the function of nerve/brain cells have been developed.
- An *artificial neural network* is a parallel processing system with small computing units (*neurons*) that work similarly to nerve/brain cells.

NEUROPHYSIOLOGICAL BACKGROUND

- The inside of every neuron (nerve or brain cell) carries a certain electric charge.
- Electric charge of connected neurons may raise or lower this charge (by means of transmission of ions through the synaptic interface).
- As soon as the charge reaches a certain threshold, an electric impulse is transmitted through the cell’s axon to the neighboring cells.
- In the synaptic interfaces, chemicals called neurotransmitters control the strength to which an impulse is transmitted from one cell to another.

NEUROPHYSIOLOGICAL BACKGROUND (cont'd)



[Wikimedia Commons]

FEED-FORWARD NEURAL NETWORKS

- We restrict to *feed-forward neural networks*, i.e. simple static input-output systems without any feedback loops between neurons and without any system dynamics over time.
- Within this class, we consider perceptrons and multi-layer perceptrons (along with the backpropagation algorithm).
- Finally, we will also highlight *deep learning*, i.e. different strategies for training networks with many layers (which mostly have large numbers of neurons too).

Important note: For notational simplicity, throughout this unit, all vectors are *column vectors*, in particular, weight vectors, input vectors and output vectors!

PERCEPTRONS

- A perceptron is a simple linear threshold unit:

$$g(\mathbf{x};\mathbf{w},\theta)=\begin{cases}1 & \text{if } \sum\limits_{j=1}^{d} w_j \cdot x_j > \theta \\ 0 & \text{otherwise}\end{cases} \tag{1}$$

- In analogy to the biological model, the inputs x_j correspond to the charges received from connected cells through the dentrites, the weights w_j correspond to the properties of the synaptic interface, and the output corresponds to the impulse that is sent through the axon as soon as the charge exceeds the threshold θ.
- Though it seems to be a (simplistic) model of a neuron, a perceptron is nothing else but a simple linear classifier.

THE PERCEPTRON LEARNING ALGORITHM

1. Given: data set $\mathbf{Z} = \{(\mathbf{x}^i, y^i) \mid i = 1, \ldots, l\}$, where $\mathbf{x}_i \in \mathbb{R}^d$, $y^i \in \{0, 1\}$; learning rate η; initial weight vector $\mathbf{w}$
2. For $k = 1, \ldots, l$ do:
   - If $g(\mathbf{x}^k; \mathbf{w}, \theta) = 0$ and $y^k = 1$
     - $\mathbf{w} := \mathbf{w} + \eta \cdot \mathbf{x}^k$
     - $\theta := \theta - \eta$
   - Else if $g(\mathbf{x}^k; \mathbf{w}, \theta) = 1$ and $y^k = 0$
     - $\mathbf{w} := \mathbf{w} - \eta \cdot \mathbf{x}^k$
     - $\theta := \theta + \eta$
3. Return to 2. if stopping condition not fulfilled
4. Output: vector of weights $\mathbf{w} \in \mathbb{R}^d$, threshold θ

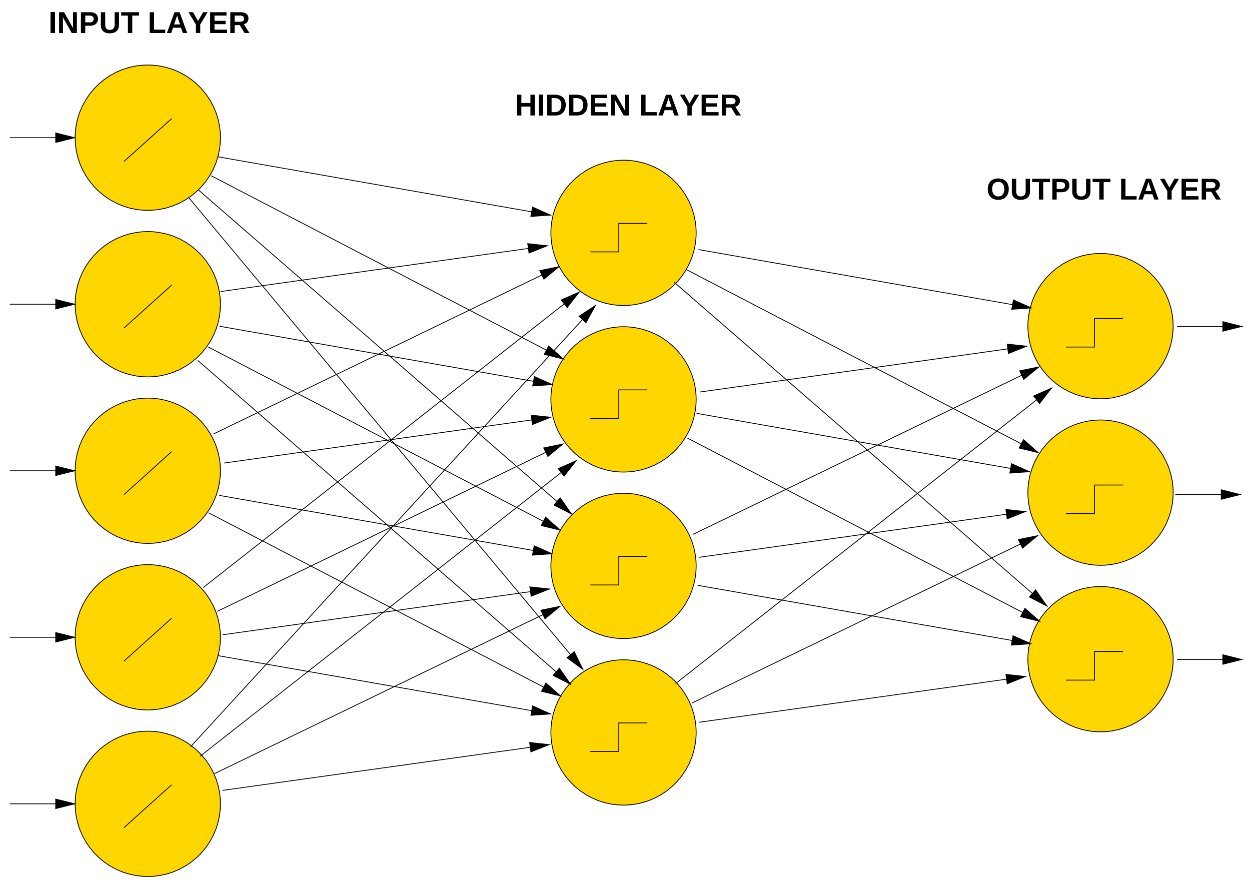
PERCEPTRONS AND LINEAR SEPARABILITY

- In case that the data set $\mathbf{Z}$ is linearly separable in $\mathbb{R}^d$, the perceptron learning algorithm terminates and finally solves the learning task.
- The final solution is not unique and the learning algorithm just gives one arbitrary solution (depending on initial weights).
- Obviously, perceptrons cannot solve classification tasks that are not linearly separable, e.g. like the simple XOR problem. In such a case, the perceptron learning algorithm not even terminates.

MULTI-LAYER PERCEPTRONS

- The only solution to the limitation of linear separability is to introduce intermediate layers.
- A multi-layer perceptron is a feed-forward artificial neural network consisting of a certain number of layers of perceptrons.
- The output of such a network is computed in the following way: The outputs of the first layer are initialized with the input $(x_1, \ldots, x_d)^T$, then the outputs of the other neurons are computed layer by layer using Formula (1).
- The "only problem" is how to find appropriate weights and thresholds that solve a given classification problem.

MULTI-LAYER PERCEPTRONS (cont’d)



SOME HISTORICAL REMARKS

- Minsky and Papert, the pioneers of perceptrons, conjectured in the late 1960ies that a training algorithm for multi-layer perceptrons—even if one could be found—is computationally infeasible and that, therefore, the study of multi-layer perceptrons is not worthwhile.
- Because of this conjecture, the study of multi-layer perceptrons was almost halted until the mid of the 1980ies.
- In 1986, Rumelhart and McClelland first published the *backpropagation algorithm* and, thereby, proved Minsky and Papert wrong.
- It turned out later that the backpropagation algorithm had already been described by Werbos in 1974 in his dissertation. In a different context, the algorithm first appeared in the work of Bryson *et al.* in the 1960ies.

CONTINUOUS ACTIVATION FUNCTIONS

The key idea is to replace the discontinuous threshold function in (1) by a *differentiable function* φ. Then the output of the neuron, its so-called *activation*, is computed as

$$g(\mathbf{x};\mathbf{w},\theta) = \varphi\Big(\sum_{j=1}^{d} w_j \cdot x_j - \theta\Big). \tag{2}$$

For simplicity, we consider the offset $-\theta$ as a “zero-th” weight w_0 (with the convention $x_0 = 1$) and denote the activation as a and the so-called *network input* of the neuron as $\mathrm{net} = \sum_{j=0}^{d} w_j \cdot x_j$:

$$g(\mathbf{x};\mathbf{w}) = a = \varphi\Big(\underbrace{\mathbf{w}^T \cdot \binom{1}{\mathbf{x}}}_{\mathrm{net}}\Big). \tag{3}$$

CONTINUOUS ACTIVATION FUNCTIONS (cont'd)

- For regression, the simplest choice is $\varphi(x) = x$ (i.e. resulting in a simple linear regressor).
- For 0/1 outputs, the most common choice is the so-called *sigmoid* or *logistic function*

$$\varphi(x) = \frac{1}{1 + e^{-x}}.$$

  Note that this is nothing else but the inverse logit function $\mathrm{logit}^{-1}(x)$, where $\mathrm{logit}(x) = \ln(\frac{x}{1-x})$. Further note that, for this particular case, $\varphi'(x) = \varphi(x) \cdot (1 - \varphi(x))$ holds.
- For -1/+1 data, a common choice of the activation function is the *hyperbolic tangent* $\varphi(x) = \tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$, which is nothing else but a transformation of the sigmoid function to the interval $[-1, +1]$.

TRAINING WITH DIFFERENTIABLE ACTIVATION FUNCTION

- The central problem with training a perceptron with threshold activation is the lack of differentiability. If a differentiable activation function and a differentiable loss function are used, we can compute the derivative of the loss of a single training sample (and, consequently, the empirical loss) according to the weights $\mathbf{w}$.
- The derivatives are the key to solving the learning problem:
  - □ Any $\mathbf{w}$ satisfying the equation $\frac{\partial L}{\partial \mathbf{w}}\big(y^i, g(\mathbf{x}^i, \mathbf{w})\big) = \mathbf{0}$ minimizes the loss for sample $(\mathbf{x}^i, y^i)$. A $\mathbf{w}$ satisfying the equation $\sum_{i=1}^{l} \frac{\partial L}{\partial \mathbf{w}}\big(y^i, g(\mathbf{x}^i, \mathbf{w})\big) = \mathbf{0}$ minimizes the empirical error for the training set $\mathbf{Z} = \{(\mathbf{x}^i, y^i) \mid i = 1, \ldots, l\}$.
  - □ If such a solution cannot be computed explicitly, an *iterative optimization method* can be used, e.g. *gradient descent*.

DIFFERENTIATING THE LOSS W.R.T. WEIGHTS

Consider a sample $(\mathbf{x}^i, y^i)$. Then we obtain the following by using the chain rule:

$$\begin{aligned}
\frac{\partial L}{\partial \mathbf{w}}\big(y^i, g(\mathbf{x}^i; \mathbf{w})\big) &= \frac{\partial L}{\partial a}(y^i, a) \cdot \overbrace{\frac{\partial a}{\partial \mathrm{net}}(\mathrm{net})}^{=\varphi'(\mathrm{net})} \cdot \overbrace{\frac{\partial \mathrm{net}}{\partial \mathbf{w}}(\mathbf{x}^i, \mathbf{w})}^{=\left(1 \mid \mathbf{x}^{i^T}\right)} \\
&= \frac{\partial L}{\partial a}(y^i, a) \cdot \varphi'(\mathrm{net}) \cdot \left(1 \mid \mathbf{x}^{i^T}\right) \\
&= \underbrace{\varphi'(\mathrm{net}) \cdot \frac{\partial L}{\partial a}(y^i, a)}_{=\delta} \cdot \left(1 \mid \mathbf{x}^{i^T}\right) \qquad (4)
\end{aligned}$$

DIFFERENTIATING THE QUADRATIC LOSS

Consider a sample $(\mathbf{x}^i, y^i)$. Then we obtain the following for the halved quadratic loss:

$$\frac{\partial L}{\partial a}(y^i, a) = \frac{\partial}{\partial a}\Big(\frac{1}{2}(y^i - a)^2\Big) = (a - y^i)$$

Hence, (4) gives the following:

$$\frac{\partial L}{\partial \mathbf{w}}\big(y^i, g(\mathbf{x}^i; \mathbf{w})\big) = \underbrace{\varphi'(\mathrm{net}) \cdot (a - y^i)}_{=\delta} \cdot \big(1 \mid {\mathbf{x}^i}^T\big) \tag{5}$$

DIFFERENTIATING THE QUADRATIC LOSS: LINEAR ACTIVATION

For the special case $\varphi(x) = x$ (hence $\varphi'(x) = 1$), (5) further simplifies to

$$\frac{\partial L}{\partial \mathbf{w}}\big(y^i, g(\mathbf{x}^i; \mathbf{w})\big) = \underbrace{(a - y^i)}_{=\delta} \cdot \big(1 \mid \mathbf{x}^{i^T}\big).$$

Some routine computations show that the equation $\sum_{i=1}^{l} \frac{\partial L}{\partial \mathbf{w}}\big(y^i, g(\mathbf{x}^i, \mathbf{w})\big) = \mathbf{0}$ has the unique solution

$$\mathbf{w} = \underbrace{\big(\tilde{\mathbf{X}}^T \cdot \tilde{\mathbf{X}}\big)^{-1} \cdot \tilde{\mathbf{X}}^T}_{\tilde{\mathbf{X}}^+} \cdot \mathbf{y}$$

with $\tilde{\mathbf{X}} = \big(\mathbf{1} \mid \mathbf{X}^T\big)$ (compare with slide no. 78).

LOGISTIC REGRESSION AND CROSS ENTROPY

Suppose we have a perceptron with sigmoid activation function φ. Then the output $g(\mathbf{x}^i;\mathbf{w})$ can be interpreted as an estimate of the probability that $\mathbf{x}^i$ belongs to the positive class:

$$p(y=1\mid\mathbf{x}^i)=a=\varphi(\text{net})\quad\text{and}\quad p(y=0\mid\mathbf{x}^i)=1-a=1-\varphi(\text{net})$$

We can unify these two formulas as follows to get a single formula for the likelihood $p(y=y^i\mid\mathbf{x}^i;\mathbf{w})$:

$$\begin{aligned}p(y=y^i\mid\mathbf{x}^i;\mathbf{w})&=\varphi(\text{net})^{y^i}\cdot\big(1-\varphi(\text{net})\big)^{(1-y^i)}\\&=a^{y^i}\cdot(1-a)^{(1-y^i)}\end{aligned}$$

LOGISTIC REGRESSION AND CROSS ENTROPY (cont'd)

Maximizing the likelihood is equivalent to minimizing -1 times its natural logarithm:

$$-\ln\big(p(y = y^i \mid \mathbf{x}^i; \mathbf{w})\big) = -y^i \cdot \ln(a) - (1 - y^i) \cdot \ln(1 - a)$$

This formula corresponds to the *cross entropy* between prediction and true class, and is a well-established loss function of its own.

Training a linear classifier by minimizing the cross entropy of a logistic linear model is usually called *logistic regression*.

DIFFERENTIATING CROSS ENTROPY

Consider a sample $(\mathbf{x}^i, y^i)$. Then we obtain the following for the cross entropy loss:

$$\frac{\partial L}{\partial a}(y^i, a) = -y^i \cdot \frac{1}{a} - (1 - y^i) \cdot \frac{1}{1-a} \cdot (-1) = -y^i \cdot \frac{1}{a} - (y^i - 1) \cdot \frac{1}{1-a}$$

For the sigmoid function, $\varphi'(\mathrm{net}) = \varphi(\mathrm{net}) \cdot (1 - \varphi(\mathrm{net})) = a \cdot (1 - a)$ holds, and (4) simplifies to

$$\begin{aligned}\frac{\partial L}{\partial \mathbf{w}}\big(y^i, g(\mathbf{x}^i; \mathbf{w})\big) &= \overbrace{a \cdot (1-a)}^{\varphi'(\mathrm{net})} \cdot \overbrace{\Big(- y^i \cdot \frac{1}{a} - (y^i - 1) \cdot \frac{1}{1-a}\Big)}^{= \frac{\partial L}{\partial a}\left(y^i, a\right)} \cdot \big(1 \mid \mathbf{x}^{i^T}\big) \\ &= \big(- y^i \cdot (1-a) - (y^i - 1) \cdot a\big) \cdot \big(1 \mid \mathbf{x}^{i^T}\big) \\ &= \underbrace{(a - y^i)}_{=\delta} \cdot \big(1 \mid \mathbf{x}^{i^T}\big).\end{aligned}$$

TRAINING ALGORITHM (ONLINE VERSION)

1. Given: data set $\mathbf{Z} = \{(\mathbf{x}^i, y^i) \mid i = 1, \ldots, l\}$, where $\mathbf{x}^i \in \mathbb{R}^d$, $y^i \in \mathbb{R}$; learning rate η; initial weight vector $\mathbf{w}$
2. For all training samples $(\mathbf{x}^i, y^i)$ (in random order) do:
   a. Compute net, a, and δ (according to (4))
   b. Update: $\mathbf{w} := \mathbf{w} - \eta \cdot \delta \cdot \begin{pmatrix} 1 \\ \mathbf{x}^i \end{pmatrix}$
3. Return to 2. if stopping condition not fulfilled
4. Output: vector of weights $\mathbf{w} \in \mathbb{R}^{d+1}$

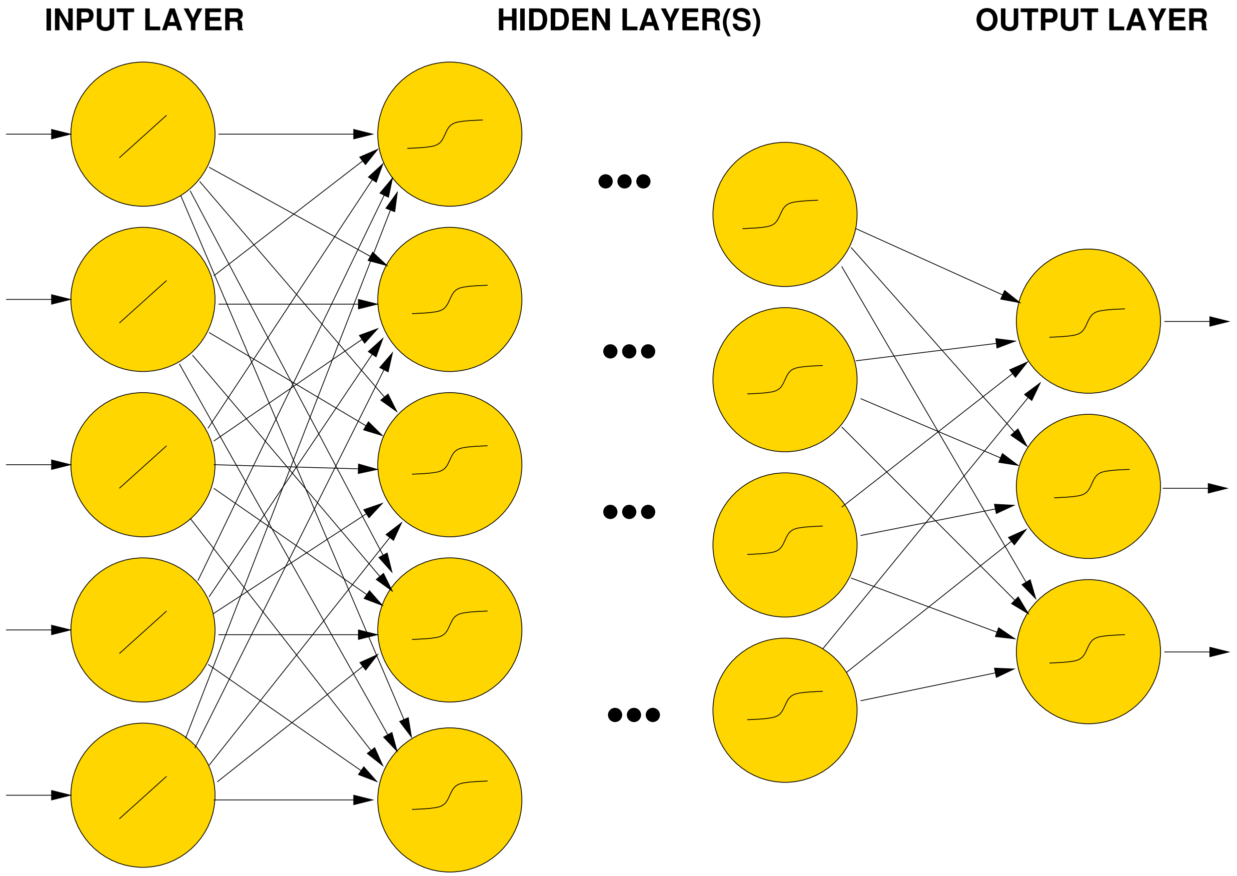
TRAINING ALGORITHM (BATCH VERSION)

1. Given: data set $\mathbf{Z} = \{(\mathbf{x}^i, y^i) \mid i = 1, \ldots, l\}$, where $\mathbf{x}^i \in \mathbb{R}^d$, $y^i \in \mathbb{R}$; learning rate η; initial weight vector $\mathbf{w}$
2. Set $\Delta\mathbf{w} = \mathbf{0}$
3. For all training samples $(\mathbf{x}^i, y^i)$ do:
   a. Compute net, a, and δ (according to (4))
   b. $\Delta\mathbf{w} := \Delta\mathbf{w} - \eta \cdot \delta \cdot \begin{pmatrix} 1 \\ \mathbf{x}^i \end{pmatrix}$
4. Update: $\mathbf{w} := \mathbf{w} + \Delta\mathbf{w}$
5. Return to 2. if stopping condition not fulfilled
6. Output: vector of weights $\mathbf{w} \in \mathbb{R}^{d+1}$

TRAINING WITH DIFFERENTIABLE ACTIVATION (COMMENTS)

- One pass through the training set is usually called a *training epoch*.
- The first variant is called *online learning* because $\mathbf{w}$ is updated for each sample individually. This local update performs one gradient descent step with the aim to reduce the loss for this sample. The *randomization of the order of training samples* is necessary to avoid that the result is biased to the order of samples.
- The second variant is called *batch learning* because the local updates are summed up for all samples before $\mathbf{w}$ is updated. This global update performs one gradient descent step with the aim to reduce the empirical error for the entire training set.
- A common variant is to train in *mini batches*, i.e. to apply batch training for small randomly sampled batches.

MULTI-LAYER PERCEPTRON: ARCHITECTURE AND NOTATION (1/3)



MULTI-LAYER PERCEPTRON: ARCHITECTURE AND NOTATION (2/3)

Suppose that we have a training set with I real-valued inputs and O outputs (continuous or binary), i.e.

$$\mathbf{Z} = \{(\mathbf{x}^i, \mathbf{y}^i) \mid i = 1, \ldots, l, \mathbf{x}^i \in \mathbb{R}^I, \mathbf{y}^i \in \mathbb{R}^O\}.$$

Then we consider a multi-layered network as follows:

- The network consists of an *input layer*, $N - 1$ *hidden/intermediate layer*, and one *output layer*.
- For each layer $k = 0, \ldots, N$, we denote the number of neurons in that layer with n_k. The input layer has I neurons, i.e. $n_0 = I$ and the output layer has O neurons, i.e. $n_N = O$.
- The activation of each layer is denoted with $\mathbf{a}^{(k)}$.

MULTI-LAYER PERCEPTRON: ARCHITECTURE AND NOTATION (3/3)

- For $k > 0$, each neuron of the k-th layer receives input from all neurons of the $k-1$-st layer and the *bias unit* (which has constant activation 1), but not from any other neurons. The *net input* of the k-th layer is denoted with $\mathbf{net}^{(k)}$. The activation function used by all units of the k-th layer is denoted with φ_k.
- For each $k > 0$, the weights between the $k-1$-st and the k-th layer are stored in a *weight matrix* $\mathbf{W}^{(k)} = \big(w_{ji}^{(k)}\big)_{j=1,\ldots,n_k}^{i=0,\ldots,n_{k-1}}$. So, $w_{ji}^{(k)}$ is the weight of the connection between the i-th unit of the $k-1$-st layer and the j-th unit of the k-th layer. Any $w_{j0}^{(k)}$ corresponds to the weight between the bias unit and the j-th unit of the k-th layer.
- Networks in which all units of one layer are connected with all units of the next layer are called *fully connected neural networks*.

FORWARD PROPAGATION / FORWARD PASS

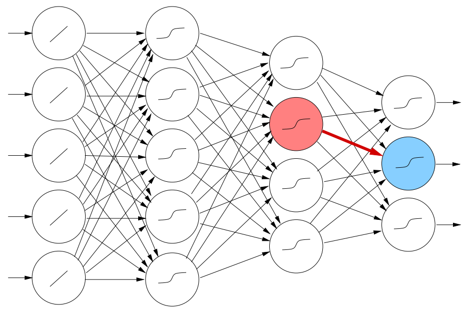
1. Given: multi-layer perceptron according to the specifications above, i.e. with I inputs, O outputs and $N-1$ hidden layers; input $\mathbf{x} \in \mathbb{R}^I$.
2. Set $\mathbf{a}^{(0)} := \mathbf{x}$.
3. For k from 1 to N do:
    a. Compute net input as $\mathbf{net}^{(k)} := \mathbf{W}^{(k)} \cdot \binom{1}{\mathbf{a}^{(k-1)}}$.
    b. Compute activation as $\mathbf{a}^{(k)} := \varphi_{k\bullet}\big(\mathbf{net}^{(k)}\big)$ (apply φ_k element-wise to $\mathbf{net}^{(k)}$).
4. Output: $\mathbf{y} := \mathbf{a}^{(N)}$.

So the input $\mathbf{x}$ is *propagated* through the network from the input layer to the output layer. As already noted, such a type of network is called *feed-forward network*.

DIFFERENTIATING THE LOSS W.R.T. WEIGHTS (1/5)

Suppose we have a training sample $(\mathbf{x}, \mathbf{y})$ that we have propagated through the network. Then the derivative of the loss function w.r.t. to weight $w_{ji}^{(N)}$, i.e. a weight between the $N-1$-st and the output layer, is given as:

$$\begin{aligned}\frac{\partial L}{\partial w_{ji}^{(N)}}\big(\mathbf{y}, \mathbf{a}^{(N)}\big) &= \frac{\partial L}{\partial a_j^{(N)}}\big(\mathbf{y}, \mathbf{a}^{(N)}\big) \cdot \overbrace{\frac{\partial a_j^{(N)}}{\partial \mathrm{net}_j^{(N)}}\big(\mathrm{net}_j^{(N)}\big)}^{=\varphi_N'\left(\mathrm{net}_j^{(N)}\right)} \cdot \overbrace{\frac{\partial \mathrm{net}_j^{(N)}}{\partial w_{ji}^{(N)}}\big(a_i^{(N-1)}, w_{ji}^{(N)}\big)}^{=a_i^{(N-1)}} \\ &= \underbrace{\varphi_N'\big(\mathrm{net}_j^{(N)}\big) \cdot \frac{\partial L}{\partial a_j^{(N)}}\big(\mathbf{y}, \mathbf{a}^{(N)}\big)}_{=\delta_j^{(N)}} \cdot a_i^{(N-1)} \qquad (6)\end{aligned}$$



DIFFERENTIATING THE LOSS W.R.T. WEIGHTS (2/5)

In matrix notation, (6) can be summarized as an outer product of deltas of the output layer and the activations of the $N-1$-st layer:

$$\frac{\partial L}{\partial \mathbf{W}^{(N)}}(\mathbf{y}, \mathbf{a}^{(N)}) = \underbrace{\operatorname{diag}\big(\varphi'_{L\bullet}(\mathrm{net}^{(N)})\big) \cdot \Big(\frac{\partial L}{\partial \mathbf{a}^{(N)}}\Big)^T}_{=\boldsymbol{\delta}^{(N)}} \cdot \big(1 \mid \mathbf{a}^{(N-1)^T}\big)$$

The deltas simplify to $\boldsymbol{\delta}^{(N)} = \big(\mathbf{a}^{(N)} - \mathbf{y}\big)$ in the following two cases:

1. Linear output units ($\varphi_N(x) = x$) are used in conjunction with the halved quadratic loss
$$L(\mathbf{y}, \mathbf{a}^{(N)}) = \frac{1}{2}\sum\nolimits_{i=1}^{O} \big(y_i - a_i^{(N)}\big)^2.$$
2. We have sigmoid output units and use the sum of binary cross entropy losses:

$$L(\mathbf{y}, \mathbf{a}^{(N)}) = -\sum\nolimits_{i=1}^{O} \big(y_i \cdot \ln(a_i^{(N)}) + (1 - y_i) \cdot \ln(1 - a_i^{(N)})\big)$$

DIFFERENTIATING THE LOSS W.R.T. WEIGHTS (3/5)

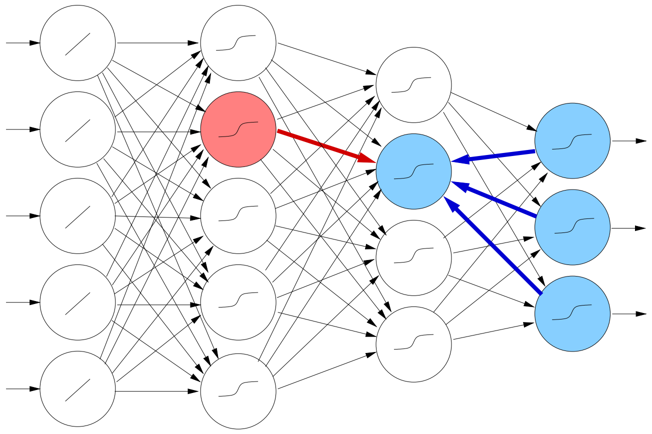
With the above assumptions, the derivative of the loss function w.r.t. to weight $w_{ji}^{(N-1)}$, i.e. a weight between the $N-2$-nd and the $N-1$-st layer, is given as:

$$\begin{aligned}
\frac{\partial L}{\partial w_{ji}^{(N-1)}}\left(\mathbf{y}, \mathbf{a}^{(N)}\right) &= \sum_{h=1}^{n_N} \frac{\partial L}{\partial a_h^{(N)}}\left(\mathbf{y}, \mathbf{a}^{(N)}\right) \cdot \overbrace{\frac{\partial a_h^{(N)}}{\partial a_j^{(N-1)}}\left(a_j^{(N-1)}\right)}^{=\varphi_N'\left(\mathrm{net}_h^{(N)}\right)\cdot w_{hj}^{(N)}} \cdot \overbrace{\frac{\partial a_j^{(N-1)}}{\partial w_{ji}^{(N-1)}}\left(a_i^{(N-1)}, w_{ji}^{(N-1)}\right)}^{=\varphi_{N-1}'\left(\mathrm{net}_j^{(N-1)}\right)\cdot a_i^{(N-2)}} \\
&= \underbrace{\varphi_{N-1}'\left(\mathrm{net}_j^{(N-1)}\right) \cdot \sum_{h=1}^{n_{N-1}} \overbrace{\varphi_N'\left(\mathrm{net}_h^{(N)}\right) \cdot \frac{\partial L}{\partial a_h^{(N)}}\left(\mathbf{y}, \mathbf{a}^{(N)}\right)}^{=\delta_h^{(N)}} \cdot w_{hj}^{(N)}}_{=\delta_j^{(N-1)}} \cdot a_i^{(N-2)} \\
&= \underbrace{\varphi_{N-1}'\left(\mathrm{net}_j^{(N-1)}\right) \cdot \left(\sum_{h=1}^{n_{N-1}} \delta_h^{(N)} \cdot w_{hj}^{(N)}\right)}_{=\delta_j^{(N-1)}} \cdot\ a_i^{(N-2)}
\end{aligned} \tag{7}$$

DIFFERENTIATING THE LOSS W.R.T. WEIGHTS (4/5)

The formula (7) tells us two important facts:

- The derivative of the loss w.r.t. a weight between the $N-2$-nd and the $N-1$-st layer is again an outer product of some deltas (of the $N-1$-st layer) and the activations of the $N-2$-nd layer.
- The deltas of the $N-1$-st layer are given as the derivative of the activation function times a weighted sum of deltas of the N-th layer, i.e. the deltas are *propagated back* through the network.



DIFFERENTIATING THE LOSS W.R.T. WEIGHTS (5/5)

In matrix notation, (7) can be summarized as

$$\frac{\partial L}{\partial \mathbf{W}^{(N-1)}}\big(\mathbf{y},\mathbf{a}^{(N)}\big) = \boldsymbol{\delta}^{(N-1)} \cdot \big(1 \mid \mathbf{a}^{(N-2)^T}\big),$$

with

$$\boldsymbol{\delta}^{(N-1)} = \mathrm{diag}\big(\varphi'_{N-1\bullet}(\mathrm{net}^{(N-1)})\big) \cdot \tilde{\mathbf{W}}^{(N)^T} \cdot \boldsymbol{\delta}^{(N)},$$

where $\tilde{\mathbf{W}}^{(N)}$ is the weight matrix without the first column of bias weights.

The same trick works for computing the derivative of the loss w.r.t. to weights between the $N-3$-rd and the $N-2$-nd layer, and so forth.

BACKPROPAGATION ALGORITHM (ONLINE VERSION)

1. Given: data set $\mathbf{Z} = \{(\mathbf{x}^i, \mathbf{y}^i) \mid i = 1, \ldots, l\}$, where $\mathbf{x}^i \in \mathbb{R}^I$, $\mathbf{y}^i \in \mathbb{R}^O$; learning rate η; network as above with some choice of initial weights.
2. For all training samples $(\mathbf{x}^i, \mathbf{y}^i)$ (in random order) do:
   a. Propagate $\mathbf{x}^i$ through network to compute all network inputs and activations.
   b. Compute $\boldsymbol{\delta}^{(N)}$.
   c. Set $\Delta\mathbf{W}^{(N)} := -\eta \cdot \boldsymbol{\delta}^{(N)} \cdot (1 \mid \mathbf{a}^{(N-1)^T})$.
   d. For all k from $N-1$ to 1 do:
      i. Compute $\boldsymbol{\delta}^{(k)} = \mathrm{diag}\big(\varphi'_{k\bullet}(\mathrm{net}^{(k)})\big) \cdot \tilde{\mathbf{W}}^{(k+1)^T} \cdot \boldsymbol{\delta}^{(k+1)}$.
      ii. Set $\Delta\mathbf{W}^{(k)} := -\eta \cdot \boldsymbol{\delta}^{(k)} \cdot \big(1 \mid \mathbf{a}^{(k-1)^T}\big)$.
   e. Update: For all k from 1 to N do
      i. $\mathbf{W}^{(k)} := \mathbf{W}^{(k)} + \Delta\mathbf{W}^{(k)}$.
3. Return to 2. if stopping condition not fulfilled
4. Output: weights $\mathbf{W}^{(1)}, \ldots, \mathbf{W}^{(N)}$.

BACKPROPAGATION ALGORITHM (BATCH VERSION)

```
1. Given: data set $\mathbf{Z} = \{(\mathbf{x}^i, \mathbf{y}^i) \mid i = 1, \ldots, l\}$, where $\mathbf{x}^i \in \mathbb{R}^I$, $\mathbf{y}^i \in \mathbb{R}^O$; learning rate
   $\eta$; network as above with some choice of initial weights.
2. Set $\Delta\mathbf{W}^{(1)}, \ldots, \Delta\mathbf{W}^{(N)}$ to zero matrices.
3. For all training samples $(\mathbf{x}^i, \mathbf{y}^i)$ do:
   a. Propagate $\mathbf{x}^i$ through network to compute all network inputs and activations.
   b. Compute $\boldsymbol{\delta}^{(N)}$.
   c. Set $\Delta\mathbf{W}^{(N)} := \Delta\mathbf{W}^{(N)} - \eta \cdot \boldsymbol{\delta}^{(N)} \cdot (1 \mid {\mathbf{a}^{(N-1)}}^T)$.
   d. For all $k$ from $N-1$ to $1$ do:
      i. Compute $\boldsymbol{\delta}^{(k)} = \mathrm{diag}\big(\varphi'_{k\bullet}(\mathrm{net}^{(k)})\big) \cdot {\tilde{\mathbf{W}}^{(k+1)}}^T \cdot \boldsymbol{\delta}^{(k+1)}$.
      ii. Set $\Delta\mathbf{W}^{(k)} := \Delta\mathbf{W}^{(k)} - \eta \cdot \boldsymbol{\delta}^{(k)} \cdot \big(1 \mid {\mathbf{a}^{(k-1)}}^T\big)$.
4. Update: For all $k$ from $1$ to $N$ do
   a. $\mathbf{W}^{(k)} := \mathbf{W}^{(k)} + \Delta\mathbf{W}^{(k)}$.
5. Return to 2. if stopping condition not fulfilled
6. Output: weights $\mathbf{W}^{(1)}, \ldots, \mathbf{W}^{(N)}$.
```

BACKPROPAGATION ALGORITHM (COMMENTS)

- When implementing the algorithm (any variant), it would be inefficient to expand the vector $\varphi'_{k\bullet}(\mathrm{net}^{(k)})$ to a diagonal matrix. Instead, this should be implemented by elementwise multiplication.
- One pass through the training set is called a *training epoch*.
- The first variant is called *online learning* because the weights are updated for each sample individually. This local update performs one gradient descent step with the aim to reduce the loss for this sample. The *randomization of the order of training samples* is necessary to avoid that the result is biased to the order of samples.

BACKPROPAGATION ALGORITHM (COMMENTS; cont'd)

- The second variant is called *batch learning* because the local updates are summed up for all samples before weights are updated. This global update performs one gradient descent step with the aim to reduce the empirical error for the entire training set.
- As mentioned before, training in *mini batches* is common, too.
- Online learning and mini batches do not consider the exact gradient, but an approximation that depends on the random choice of samples. This is called *stochastic gradient descent*.

BACKPROPAGATION ALGORITHM: SUMMARY

Forward pass: Set activation of input layer to input vector. For each layer (from first hidden layer to output layer), compute net input as product of activation with weight matrix and apply activation function to net input for each unit.

Backward pass: Perform forward pass and compute deltas for output layer. For each layer (from last hidden layer to first hidden layer), compute deltas as elementwise products of the activations' derivatives and the product of the transpose of the weight matrix times the deltas of the next layer (the one "to the right"). The weight updates are outer products of the deltas of the "right layer" and the activations of the "left layer".

MULTI-LAYER PERCEPTRONS APPLIED TO CLASSIFICATION

- Assume we are given a data set $\mathbf{Z} = \{(\mathbf{x}^i, y^i) \mid i = 1, \ldots, l\}$.
- If we have a binary classification problem, i.e. $y_i \in \{0, 1\}$ or $y_i \in \{-1, +1\}$ (in this case, replace -1's by 0's), we can use a single output neuron ($O = 1$).
- If we are given a problem with $M > 2$ classes, i.e. $y^i \in \{1, \ldots, M\}$, we can use a network with $O = M$ output neurons. The labels have to be mapped to M-dimensional output vectors $\mathbf{y}^i$ in the following way:

$$y^i_j = \begin{cases} 1 & \text{if } y^i = j \\ 0 & \text{otherwise} \end{cases}$$

When predicting a new sample $\mathbf{x}$, the output neuron with the highest activation determines the class prediction.

MULTI-LAYER PERCEPTRONS APPLIED TO CLASSIF. (cont'd)

- Though sigmoid activation functions are possible and not so uncommon for multi-class problems with more than 2 output neurons, the activations need not sum up to 1, therefore, they cannot be used as estimates of class probabilities.
- For multi-class classification, it is more common to use the *softmax* activation function:

$$a_i^{(N)} = \frac{\exp\left(\mathrm{net}_i^{(N)}\right)}{\sum_{j=1}^{O} \exp\left(\mathrm{net}_j^{(N)}\right)}$$

- If we use the softmax activation function in conjunction with the multi-class cross entropy loss

$$L(\mathbf{y}, \mathbf{a}^{(N)}) = -\sum_{i=1}^{O} y_i \cdot \ln(a_i^{(N)}),$$

the deltas of the output layer again simplify to $\boldsymbol{\delta}^{(N)} = \left(\mathbf{a}^{(N)} - \mathbf{y}\right)$.

MULTI-LAYER PERCEPTRONS APPLIED TO REGRESSION

- Assume we are given a data set $\mathbf{Z} = \{(\mathbf{x}^i, \mathbf{y}^i) \mid i = 1, \ldots, l\}$, where $\mathbf{x}^i \in \mathbb{R}^I$ and $\mathbf{y}^i \in \mathbb{R}^O$ (in the simplest case $O = 1$)
- There are two ways to make multi-layer perceptrons usable for regression:
  1. Transforming/scaling all desired output vectors $\mathbf{y}^i$ to $[0, 1]^O$
  2. The better option is to use linear neurons in the output layer, i.e., $\varphi_N(x) = x$ is used, while the other neurons remain unchanged; in this case, the outputs of the $N - 1$-st layer can be understood as basis functions; the output is then a linear combination of these basis functions.
- The most common loss function is the quadratic loss.
- Multi-layer perceptrons are universal approximators, however, this is only a theoretical result with minor practical value.

ANOTHER USAGE SCENARIO: AUTOENCODERS

- Assume we are given a data set $\mathbf{X} = \{\mathbf{x}^i \mid i = 1, \ldots, l\}$, where $\mathbf{x}^i \in \mathbb{R}^d$. (only inputs, no targets!)
- Now consider a multi-layer perceptron with $I = d$ input neurons and $O = d$ output neurons and suppose that we train the network with the training set $\mathbf{Z} = \{(\mathbf{x}^i, \mathbf{x}^i) \mid i = 1, \ldots, l\}$, i.e. the network is trained to produce its input as output. Such a network is called *autoencoder*.
- If the number of units in each hidden layer is smaller than d or if other precautions against overfitting are taken (see later), then such an autoencoder learns an efficient representation of the data set.
- It is also possible to train an autoencoder on a data set whose input vectors have been perturbed by noise. Then the autoencoder learns to recover original data from noisy data. Such a network is commonly called *denoising autoencoder*.

PRACTICAL CONSIDERATIONS (1/4)

Input scaling: It is advisable to standardize all inputs to the same range, e.g. to $[-1, 1]$ or to mean 0 and variance 1. This ensures that all input features have the same prior importance. Otherwise large input features are overrated and it might take long to properly adjust the weights for small input features.

Initial weights: A common strategy is to use small random values, e.g. drawn from a uniform distribution over $[-0.1, 0.1]$. Weights around 0 result in net inputs around 0. Therefore, the network is almost linear in the beginning. Moreover, the derivatives of the sigmoid activation function is maximal around 0, which speeds up learning in the beginning.

PRACTICAL CONSIDERATIONS (2/4)

Number of hidden layers: For most learning tasks, two hidden layers are sufficient. For simpler tasks, even one or no hidden layer may be sufficient. Larger numbers of hidden layers would make sense for more difficult tasks, but appropriate precautions have to be taken against overfitting. Moreover, training becomes increasingly difficult for deeper networks (see later).

Numbers of hidden units: Too few hidden units result in underfitting. Too many hidden units may result in overfitting unless appropriate precautions, such as, regularization are taken (see later). Moreover, pruning or growing strategies can be applied during training.

PRACTICAL CONSIDERATIONS (3/4)

Learning rates: Typically, learning rates have to be low for online learning, e.g. 0.1 or 0.01. For batch training, it is common to use something like, e.g., $\eta = \frac{0.7}{l}$. The best choice, however, is to adapt the learning rate during training. This can be done by help of the Hessian (advanced topic, see literature).

Online vs. batch: With an appropriately chosen learning rate (or an adaptive learning rate), online learning is known to converge faster than batch learning (see literature). It is also possible to choose a strategy in between: by training in *mini-batches*.

Selective δ propagation: Learning can be sped up by marking samples with sufficiently large deltas and only considering those for a certain number of epochs.

PRACTICAL CONSIDERATIONS (4/4)

Momentum term: In order to avoid oscillations, it is possible to augment updates with the previous update, i.e. $\mathbf{w} := \mathbf{w} + \Delta\mathbf{w} + \mu \cdot \Delta\mathbf{w}^{\text{old}}$, where $\Delta\mathbf{w}^{\text{old}}$ is the weight update of the previous iteration/epoch. μ is the momentum parameter that controls the influence of the previous update.

Stopping criteria: The following stopping criteria can be used (possibly in combination):

- Maximal number of epochs/iterations reached
- Empirical error below certain threshold
- Relative improvement of empirical below certain threshold
- Maximal weight change below certain threshold

REGULARIZATION (1/4)

- Support vector machines include a capacity term in its objective: the flatness of discriminant/regressor function (which corresponds to margin maximization for classification). In theory, this capacity term aims to avoid overfitting by limiting model complexity.
- The neural networks considered so far have no such built-in mechanism against overfitting.
- In neural networks, complexity is mainly controlled by the architecture of the network (number of hidden layers along with numbers of neurons in hidden layers).
- Since it is hard to find the optimal architecture for a given learning task, an option would be to allow for a larger (more complex) network, but to control its model complexity by appropriate additional measures during training.

REGULARIZATION (2/4)

Early stopping: As mentioned above, if we start from initial weights around 0, the neurons' activation is nearly linear. So the model complexity is low in the beginning. During the course of training, weights will become larger and the neurons' activation becomes more non-linear, i.e. the model complexity grows. The idea of early stopping is to quit learning when the right model complexity is reached. There are no well-founded strategies for that. Instead, the use of a validation set is more or less the only option.

Training with noise: In order to avoid that the networks adapts too much to individual training samples (overfitting), noise is added either to the inputs or to the weights during training. Different strategies are available for that (see literature).

REGULARIZATION (3/4)

Weight decay: As mentioned above, weights around 0 correspond to low complexity models. A higher model complexity necessitates higher weights (in terms of their absolute values). Therefore, a mechanism that favors weights around 0 can be used to control model complexity. In other words, we add $\lambda \cdot \Omega(\mathcal{W})$ to the learning objective, where the *regularization term* $\Omega(\mathcal{W})$ measures the overall size of the weights, $\mathcal{W}$ is the set of all weights (resp. weight matrices) in the network and λ is the *regularization parameter* that controls the influence of the regularization term. A basic variant is *Tikhonov regularization* in which $\Omega(\mathcal{W})$ is the (halved) *sum of squares of all weights*. This results in a derivative of $\frac{\partial \Omega}{\partial w_{ji}^{(k)}}(\mathcal{W}) = w_{ji}^{(k)}$ resp. $\frac{\partial \Omega}{\partial \mathbf{W}^{(k)}}(\mathcal{W}) = \mathbf{W}^{(k)}$. Hence, the update in backpropagation changes to

$$\mathbf{W}^{(k)} := \mathbf{W}^{(k)} + \Delta \mathbf{W}^{(k)} - \lambda \cdot \mathbf{W}^{(k)},$$

where $-\lambda \cdot \mathbf{W}^{(k)}$ obviously tends to push the weight matrix towards $\mathbf{0}$.

REGULARIZATION (4/4)

Growing and pruning strategies: We already noted that the numbers of hidden neurons control the complexity of the network. *Growing* starts from a small (unterfitted) network and successively increases model complexity by adding hidden neurons until the right model complexity is reached. The best known algorithm is *cascade correlation* (see literature). *Pruning*, on the other hand, is concerned with removing unnecessary neurons from an already trained (possibly overfitted) network. Two well-known strategies are *Optimal Brain Damage (OBD)* and the *Optimal Brain Surgeon (OBS)*. Both methods rely on second-order derivatives (Hessian) to estimate the relative importance of neurons (advanced topic, see literature).

ALTERNATIVE LEARNING ALGORITHMS

As backpropagation with a constant learning rate converges slowly in many cases, add-ons and alternatives have been established:

Adaptive learning rate: The learning rate is not constant, but varied according to some strategy/heuristics.

Second-order methods: methods that not only use first derivatives, but also second-order derivatives (Newton method) or approximations of second-order derivatives (quasi-Newton methods, BFGS, Levenberg-Marquardt).

Resilient backpropagation (Rprop): only the signs of gradients are taken into account to determine the direction of the weight update; the absolute amount of update is specific for each weight and changes for each iteration. The more updates appear consistently in the same direction, the more the amount grows. The more the updates oscillate, the more the amount is shrunk.

PRO’S AND CON’S OF ARTIFICIAL NEURAL NETWORKS

+ Universal
+ Relatively easy to apply
– Black box
– Only applicable for vectorial data
– Large effort for training
– Solution is not guaranteed to be a global minimum, but only a local one
– Large number of different variants and parameters; thorough parameter selection is not feasible because of large number of parameters and long training times.

DEEP NETWORKS: MOTIVATION

- Neural networks that are deep (many hidden layers) and broad (many neurons in each hidden layers) would allow for representing, storing, and recognizing many complex patterns.
- However, there are practical obstacles to the application of standard backpropagation (and similar variants) to such networks:
  - Overfitting is highly likely to occur (which could possibly be counteracted with some regularization).
  - Backpropagation cannot be used for training more than 2–3 hidden layers because of *vanishing gradients* (see next page).

THE VANISHING GRADIENT PROBLEM

We have seen above that

$$\begin{aligned}\frac{\partial L}{\partial \mathbf{W}^{(k-1)}}\big(\mathbf{y},\mathbf{a}^{(N)}\big) &= \boldsymbol{\delta}^{(k-1)}\cdot\big(1 \mid \mathbf{a}^{(k-2)^T}\big)\\ &= \operatorname{diag}\big(\varphi'_{k-1\bullet}(\mathrm{net}^{(k-1)})\big)\cdot\tilde{\mathbf{W}}^{(k)^T}\cdot\boldsymbol{\delta}^{(k)}\cdot\big(1 \mid \mathbf{a}^{(k-2)^T}\big),\end{aligned}$$

i.e. each backward propagation step includes a multiplication with derivatives of activation functions. If we have back-propagated deltas from the N-th layer to the $N-q$-th layer, we have performed such multiplications q times. Since the maximal value of the derivative of the sigmoid/logistic function is $\frac{1}{4}$, this means that the magnitude of the deltas (and therefore the magnitude of the gradients and the updates) decreases exponentially layer by layer. This makes backpropagation training of deep networks intractable, as weights at the back are updated extremely slowly.

DEEP LEARNING

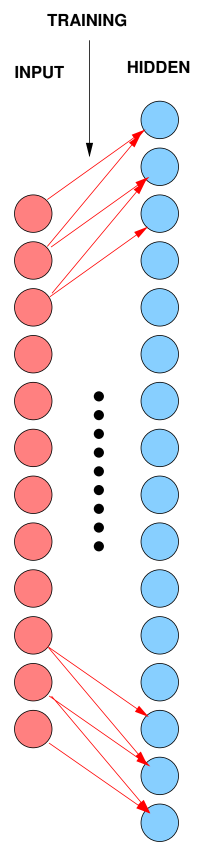
- *Deep learning* is a class/framework of strategies for training deep networks that are aimed to learn multiple levels of representations of the data and to allow for accurate predictions from these representations.
- In the first successful supervised applications, deep learning was a two-step procedure:

  Pre-training: levels of representations are learned layer by layer (unsupervised or supervised).

  Fine-tuning: a supervised learning algorithm is applied that makes predictions from the activations of the last layer of the pre-trained network. Typically a single-layer ANN is used, but any other predictor would be possible too.

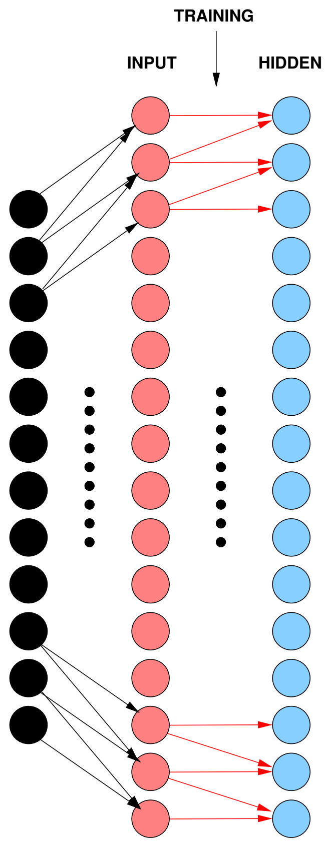
PRE-TRAINING AND FINE-TUNING

Pre-training: hidden layer no. 1



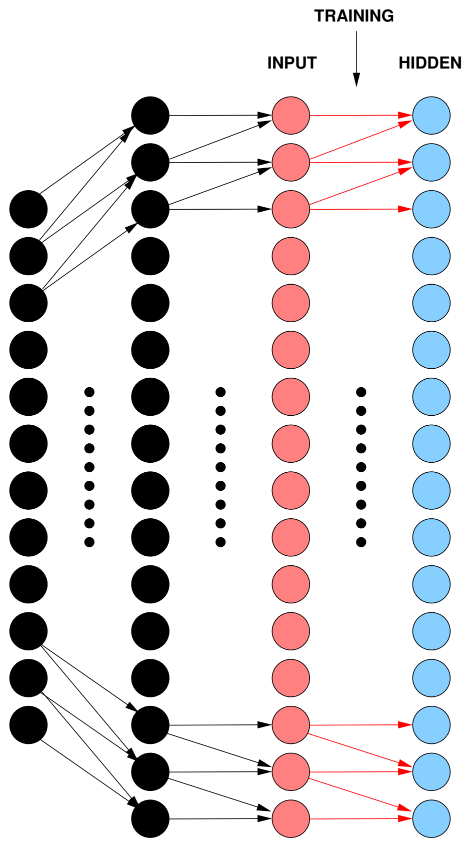
PRE-TRAINING AND FINE-TUNING

Pre-training: hidden layer no. 2



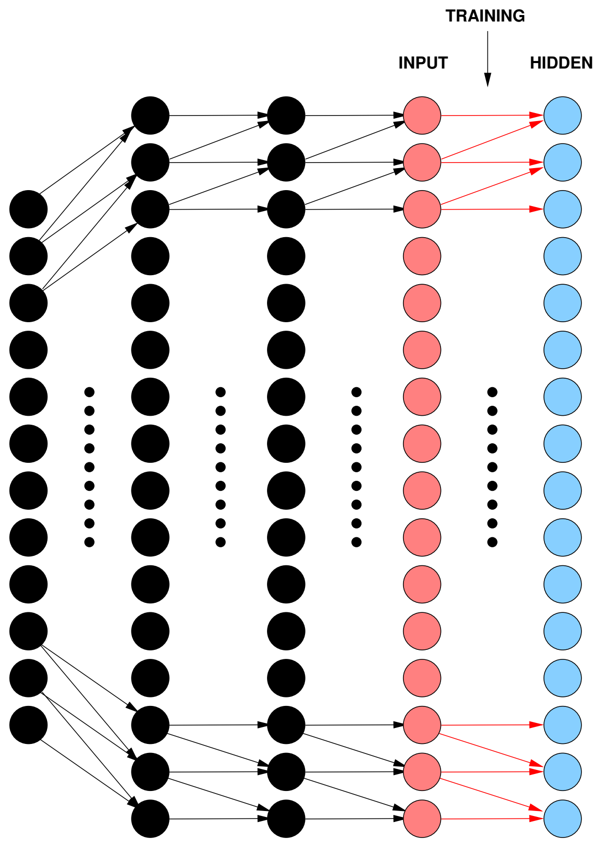
PRE-TRAINING AND FINE-TUNING

Pre-training: hidden layer no. 3



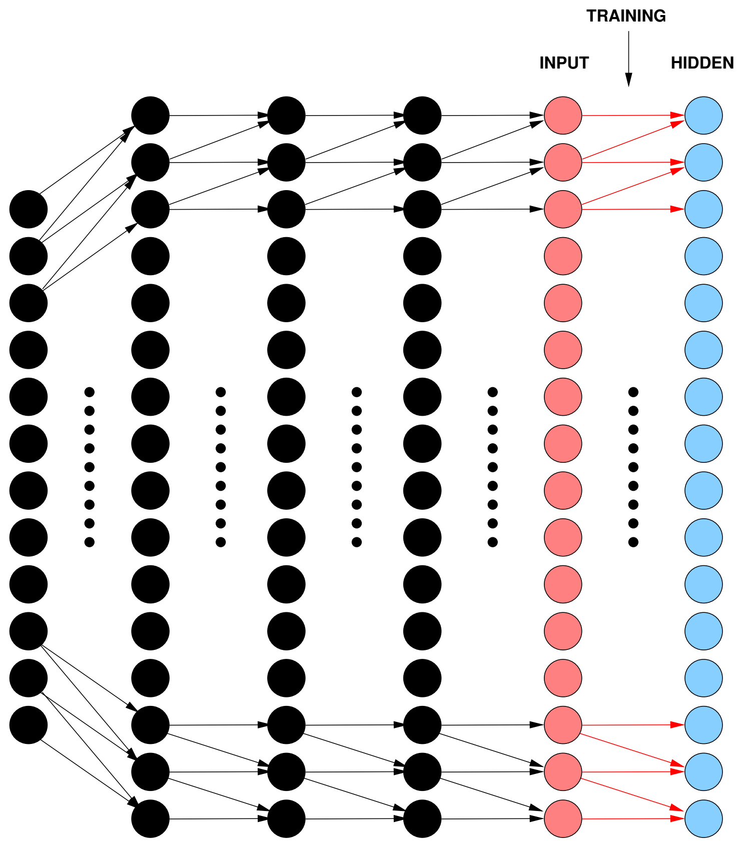
PRE-TRAINING AND FINE-TUNING

Pre-training: hidden layer no. 4



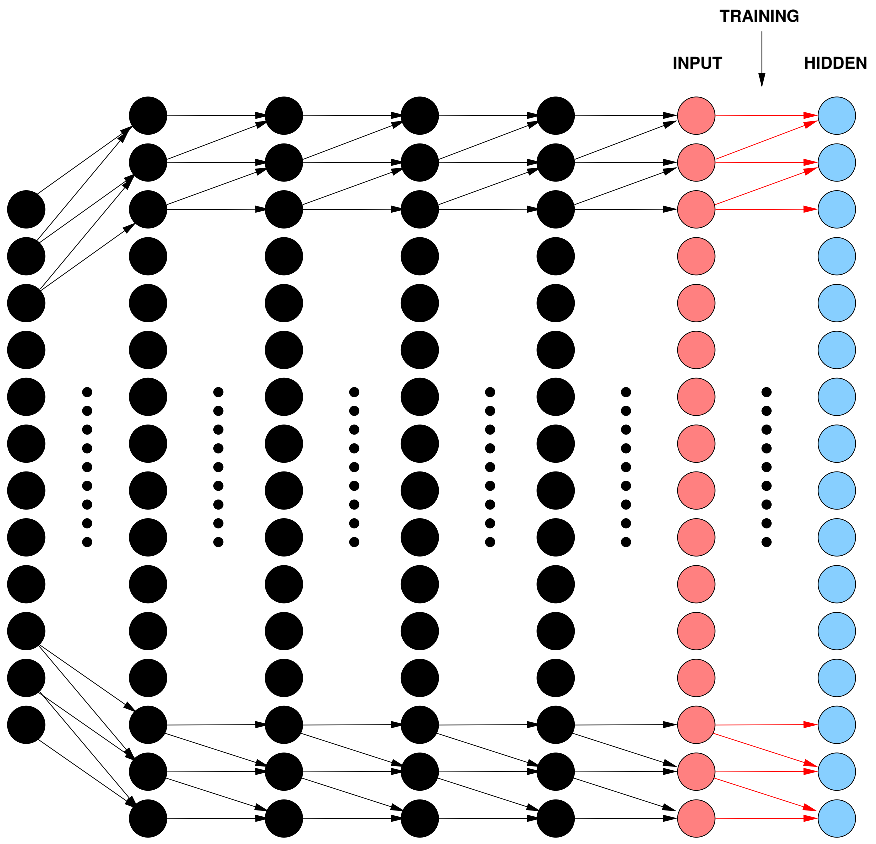
PRE-TRAINING AND FINE-TUNING

Pre-training: hidden layer no. 5



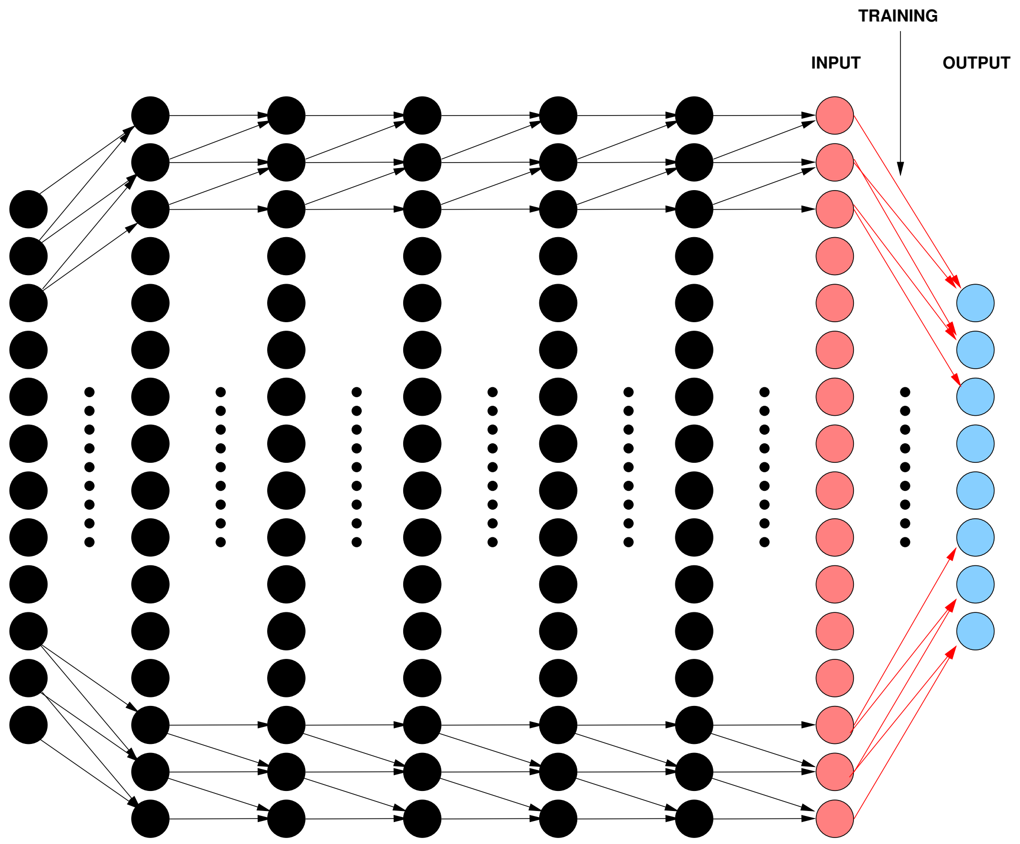
PRE-TRAINING AND FINE-TUNING

Pre-training: hidden layer no. 6



PRE-TRAINING AND FINE-TUNING

Fine-tuning:



METHODS FOR PRE-TRAINING

Restricted Boltzmann machine (RBM): A simple stochastic neural network with an input layer and one hidden layer that are connected in both directions with symmetric weights; RBMs aim to learn a probability distribution over the inputs. The learning algorithm uses sampling of inputs and hidden activations along with gradient descent.

Autoencoders: A (denoising) autoencoder with one hidden layer is trained in each pre-training step. After training, the output layer of the autoencoder is discarded and only the hidden layer remains. In the subsequent step, another autoencoder is trained with the inputs being the activations of the hidden neurons of the previously trained autoencoder.

Supervised pre-training: A network with one hidden layer is trained in each pre-training step. After training, the output layer is discarded and only the hidden layer remains. In the subsequent step, another network is trained with the inputs being the activations of the hidden neurons of the previous network.

DEEP LEARNING (cont'd))

- Unsupervised deep learning only consists of unsupervised pre-training and omits fine-tuning.
- In the meantime, it has become common to apply *deep learning in a purely supervised fashion*, i.e. without pre-training. In order to avoid the vanishing gradient problem and to facilitate meaningful data representations, appropriate regularization measures need to be employed (see below).

HOW TO LEARN GOOD REPRESENTATIONS?

- The success of a deep network is determined by how meaningful the representations in the hidden layers are.
- What is a meaningful representation?
  - Each hidden unit corresponds to a specific pattern (hidden) in the data.
  - Different hidden units correspond to different patterns, i.e. the patterns are disentangled. Otherwise, further representations are difficult to learn and/or the final supervised learning in the fine-tuning step is difficult.

In the following, we highlight a few approaches aimed at meaningful representations.

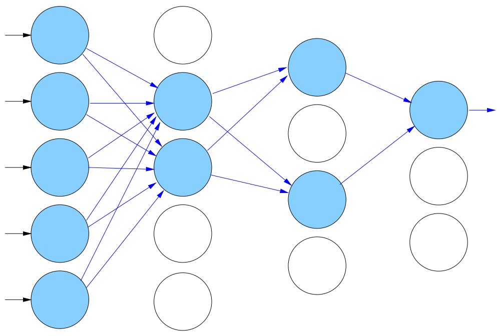
LEARNING SPARSE REPRESENTATIONS

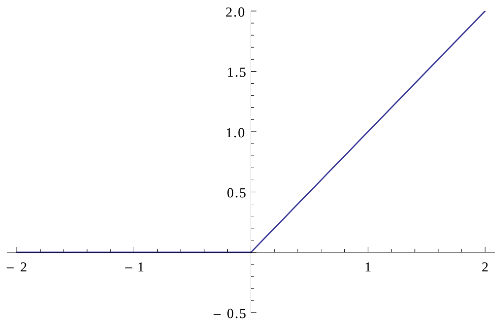
Disentangling of representations can also be achieved by ensuring sparse activation, i.e. only a fraction of hidden neurons are activated for a given input.

Dropout: during training, activations are randomly set to 0 (typically with a probability of 0.5);

Rectified linear units (ReLU): instead of a sigmoid activation function, a function is used that gives 0 below a certain threshold. The most common choice is $\varphi(x) = \max(0, x)$.

These approaches allow for training a deep network directly without pre-training.





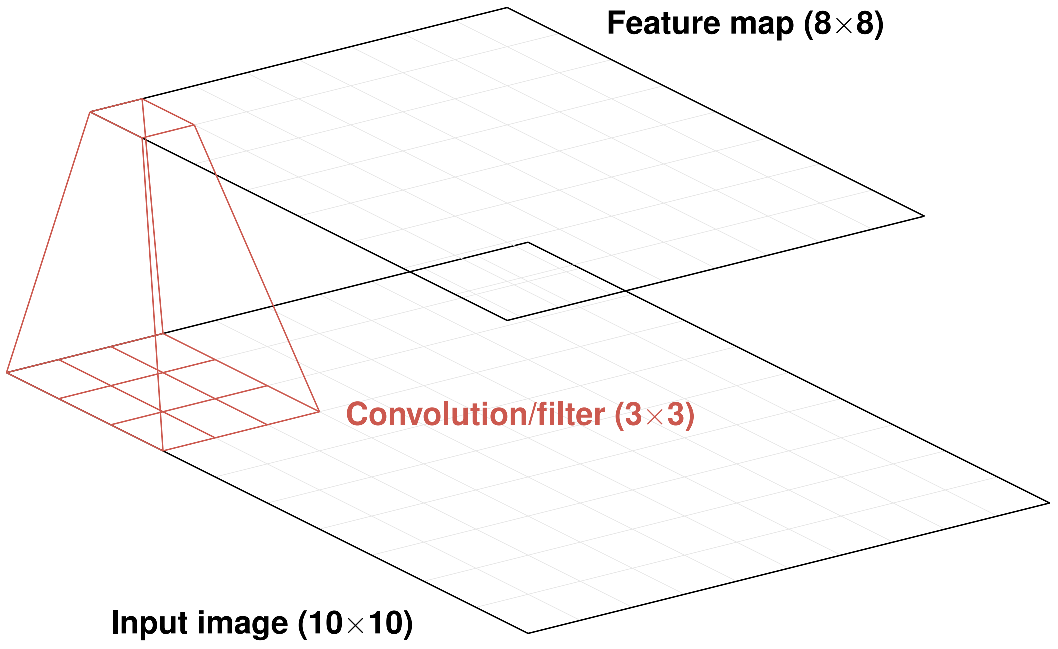
CONVOLUTIONAL NEURAL NETWORKS (CNNs): MOTIVATION

- In principle, classical feed-forward neural networks (fully connected networks) could be used for image analysis by simply connecting the pixels to input units.
- However, if at all, this only makes sense for small aligned images (e.g. in character recognition).
- For the analysis of larger and more complex images, this standard architecture is not useful.
- Instead, it is common to have stacked layers of units that operate on small overlapping patches/windows. Such networks are called *convolutional neural networks (CNNs)*.

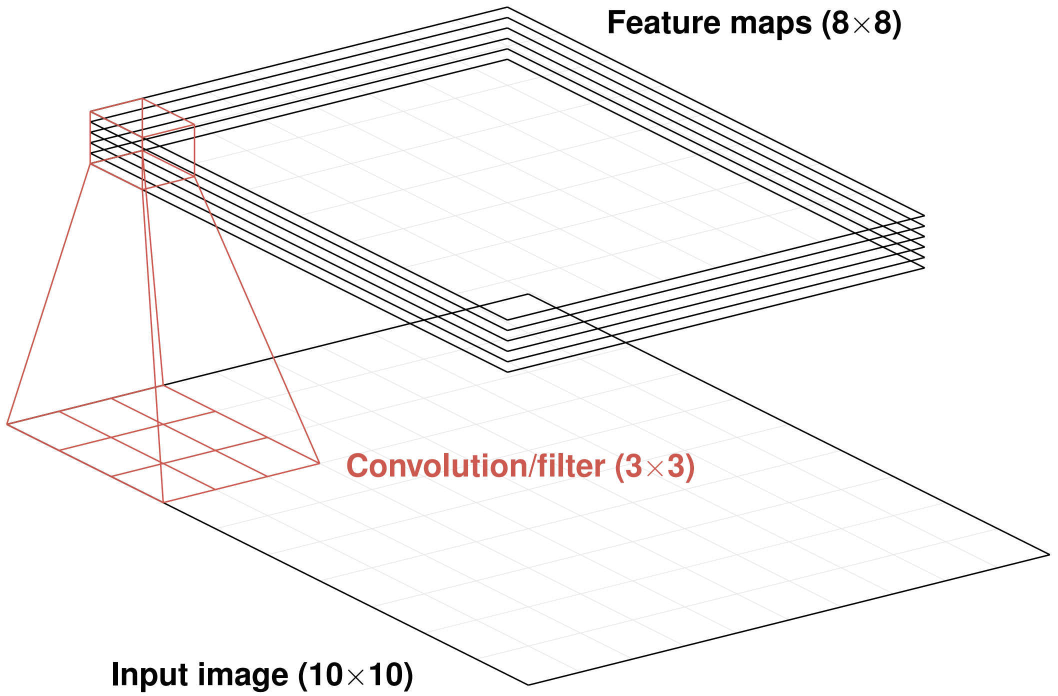
CNNs: ARCHITECTURE

- The first convolutional layer usually consists of multiple units that operate on small image patches (3×3, 5×5, or 7×7).
- Each unit corresponds to one simple feature of a patch. Such units are often called *filters*.
- The activations of all units are computed for all patches, thereby creating a *feature map* of the image.
- Convolutional layers can be stacked.
- It can be useful to down-sample feature maps by local *max pooling* (e.g. with non-overlapping 2×2 windows).
- Such networks can either have fully connected layers on top (e.g. for image classification) or can also be *fully convolutional* (output is an image; e.g. for segmentation of detected objects).

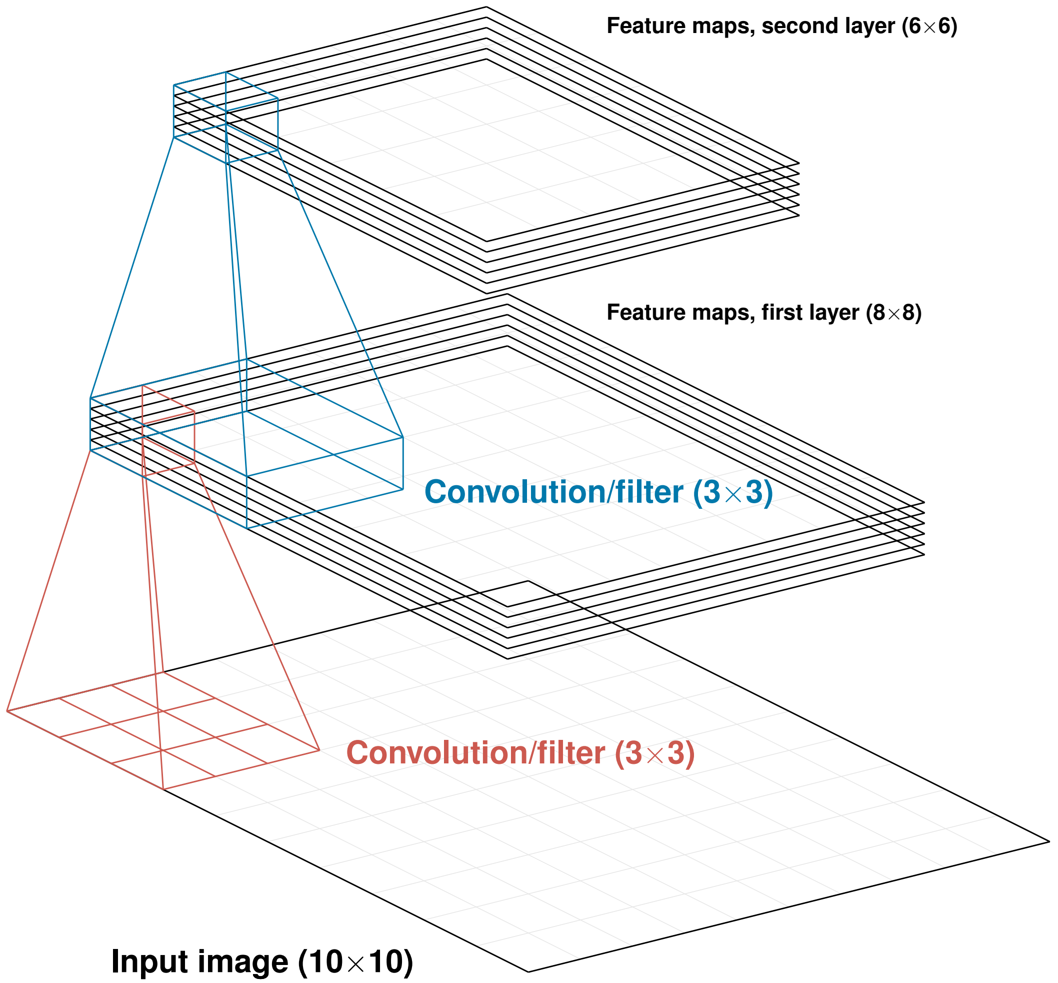
CNNs: ARCHITECTURE (cont'd)



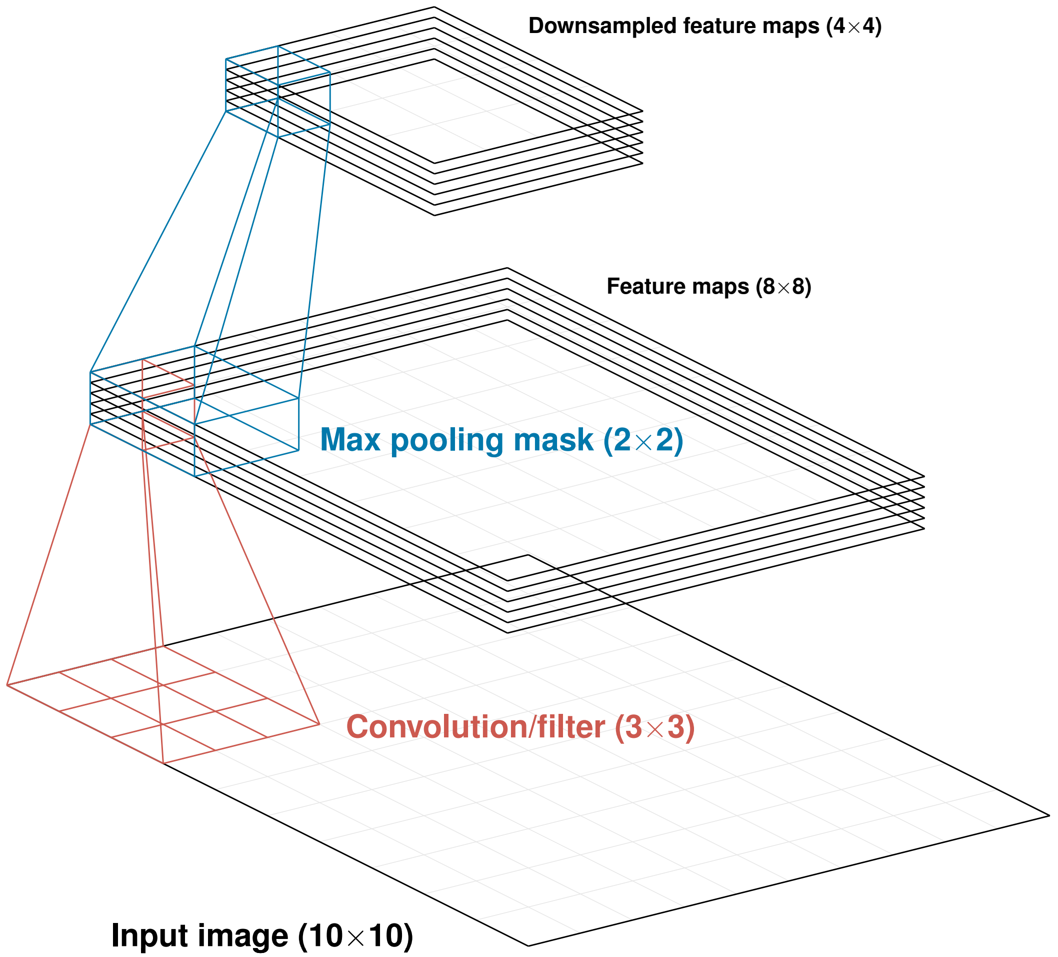
CNNs: ARCHITECTURE (cont'd)



CNNs: ARCHITECTURE (cont'd)



CNNs: ARCHITECTURE (cont’d)

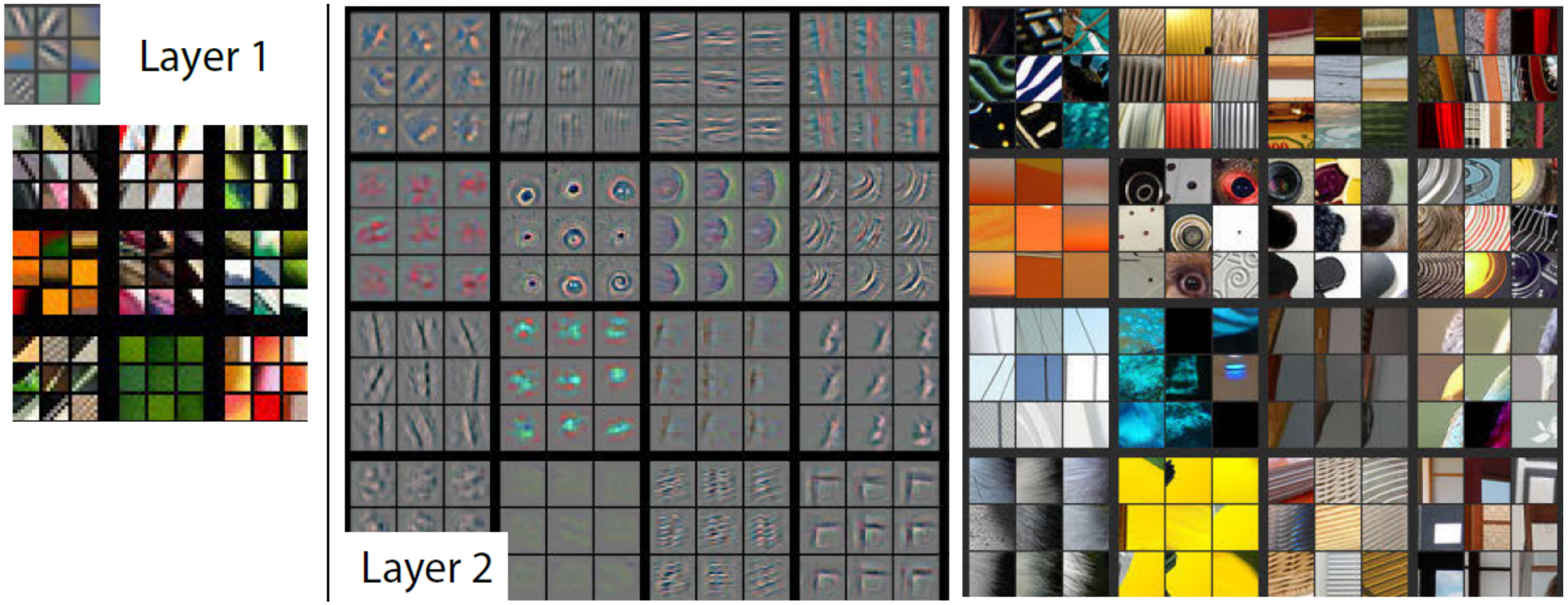


CNNs: TRAINING

- Fully connected layers are trained as usual.
- In convolutional layers, each feature map/filter has only one set of weights and all windows contribute weight updates. This is called *weight sharing*.
- In may pooling layers, the error signal is only propagated to the input from which the maximal activation came.

CONVOLUTIONAL NETWORK: EXAMPLE (M.D. Zeiler & R. Fergus; arXiv, 2013)

Two layers of a convolutional network: hypothetical inputs maximizing activation and real images that lead to a high activation of the considered neuron



DEEP NETWORKS: THE HYPE

- Although the foundations of deep learning have been layed 15–20 years ago, a major hype emerged only recently in the machine learning community.
- Deep networks have won numerous competitions in music, speech and image recognition, drug discovery, and other fields.
- Deep learning has been called "*...the biggest data science breakthrough of the decade*" (J. Howard).
- The New York Times covered the subject twice with front-page articles in 2012.

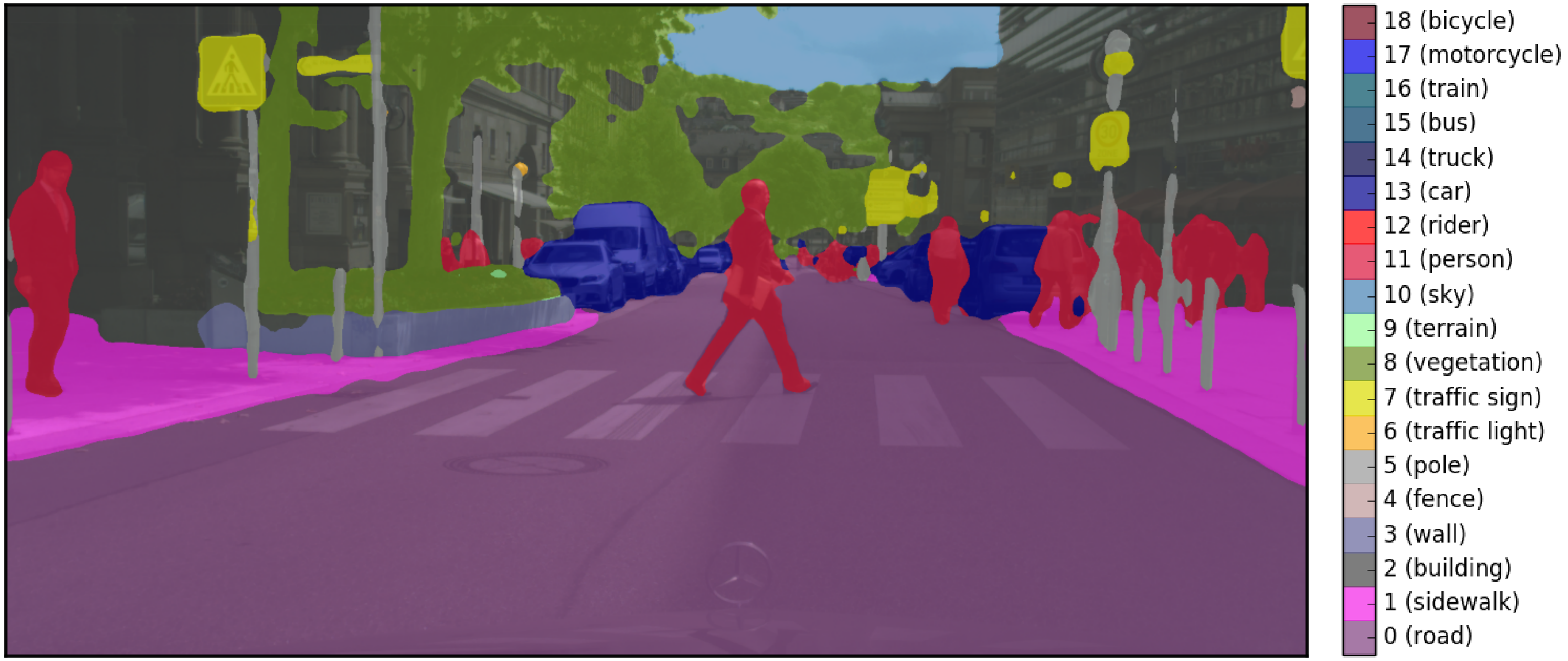
DEEP NETWORKS: THE HYPE (cont'd)

- Major companies, such as, Google, Microsoft, Apple, facebook, etc. have recently invested in deep learning and are using deep networks in their products and services.
- Google has acquired companies specialized in deep learning: DNNresearch (founded by G. Hinton, U. Toronto; March 2013; price not revealed) and Deepmind (London-based company founded by D. Hassabis; January 2014; price approx. $400–650m)

DEEP LEARNING: SUCCESS STORIES

- Have become standard in object recognition in images; ImageNet Challenge dominated by CNNs since several years
- NIH Tox21 Challenge on predicting toxicity of chemical structures won by deep learning system
- Deep learning is an essential ingredient in recent attempts to autonomous driving, e.g. for semantic segmentation and even end-to-end learning of driving decisions
- AlphaGo: first system to play Go on the same level as world-class players
- Generating image captions
- DeepArt: paint images in given style

EXAMPLE: SEMANTIC SEGMENTATION



EXAMPLE: TOX21 CHALLENGE (1/3)

- Computational challenge set up by the US agencies NIH, EPA, and FDA
- Unprecedented multi-million-dollar effort
- 12,000 compounds tested experimentally for twelve different toxic effects
- Goal: predict toxicity computationally

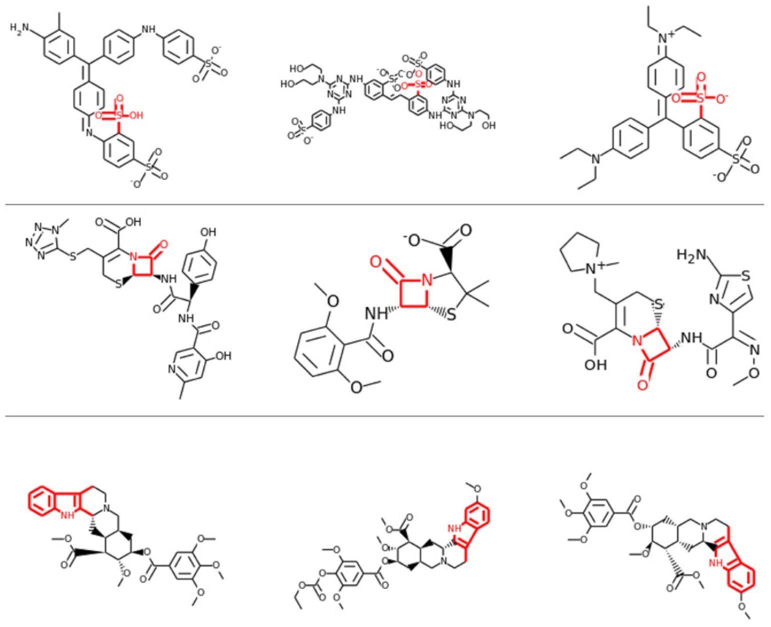
EXAMPLE: TOX21 CHALLENGE (2/3)

Input features:

- 40,000 very sparse features: Extended Connectivity Finger-Print (ECFP4) presence count of chemical sub-structures
- 5,057 additional features:
    - 2,500 toxicophore features
    - 200 common chemical scaffolds
    - various chemical descriptors

EXAMPLE: TOX21 CHALLENGE (3/3)

- Deep learning-based solution by JKU's Institute of Bioinformatics won the grand challenge, both panels (nuclear receptor panel and stress response panel), and six single prediction tasks.
- The hierarchical representation of deep networks allowed for the identification of novel toxicophores.



DEEP LEARNING: COMMENTS

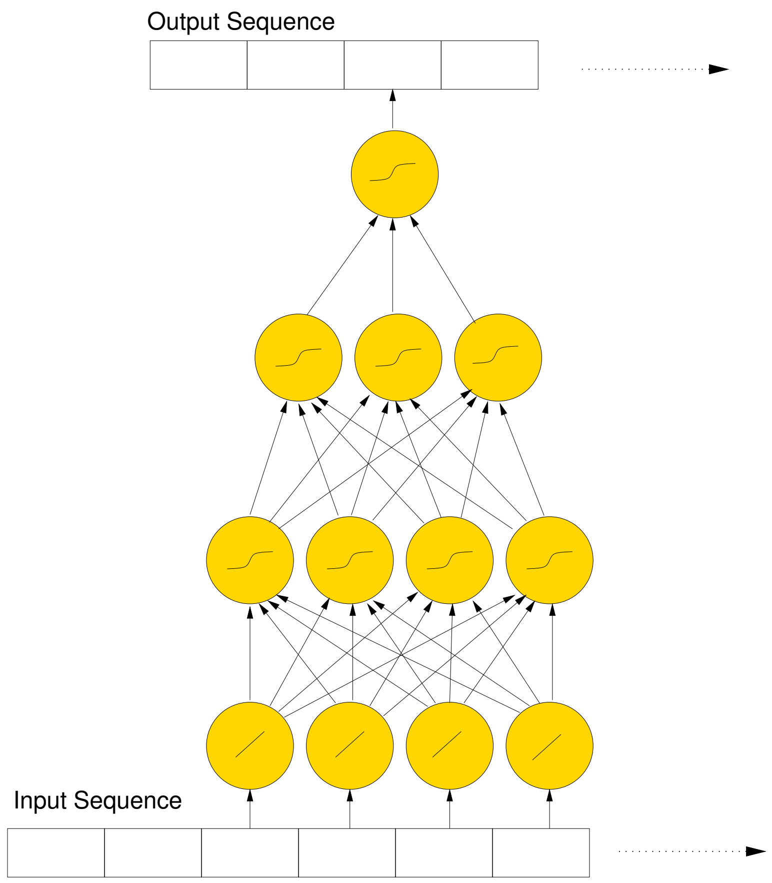
- Without any doubt, deep networks are the most powerful tools for audio and image recognition and other fields, also outperforming support vector machines.
- Despite the practical successes, the theoretical foundations why and under which conditions deep networks work are lagging far behind.
- The spectrum of variants is hard to survey, and the choice of good parameters is both crucial and tricky.
- Learning good representations of complex data, such as, high-res images, requires excessive amounts of training data and excessive computational power (supercomputers, GPUs).

TIME SERIES/SEQUENCE ANALYSIS WITH NEURAL NETS?

- Feedforward neural networks require vectorial inputs. Therefore, they cannot be applied to time series or sequences directly.
- One option is to apply them to (sliding) windows.
- The obvious disadvantage of this simple approach is that windows are treated independently and no learning across windows can take place.

TIME SERIES/SEQUENCE ANALYSIS WITH NEURAL NETS?

- Feedforward neural networks require vectorial inputs. Therefore, they cannot be applied to time series or sequences directly.
- One option is to apply them to (sliding) windows.
- The obvious disadvantage of this simple approach is that windows are treated independently and no learning across windows can take place.

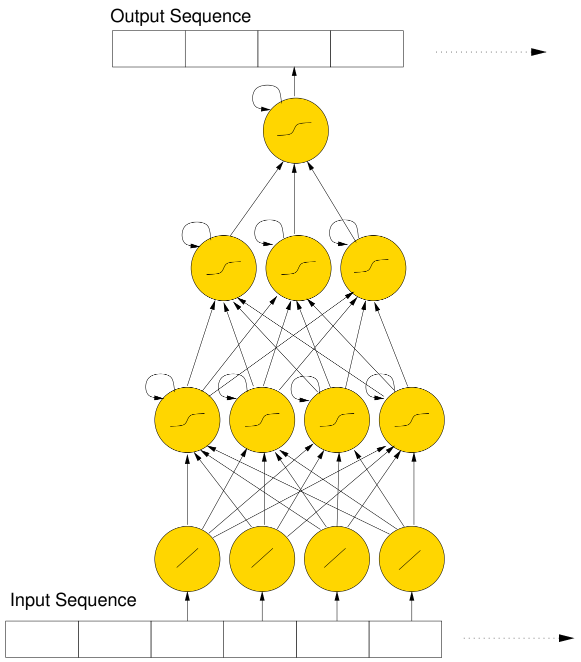


RECURRENT NEURAL NETS (RNNs)

- *Recurrent neural networks (RNNs)* provide an alternative, where “recurrent” means that the network has connection cycles.
- There are several different RNN architectures.
- After each evaluation (for one window, in time step t), the activations are kept and potentially used as inputs in time step $t+1$; so the generalization of the forward pass is straightforward for RNNs.
- The backpropagation algorithm can also be generalized to RNNs; this is typically called *backpropagation through time*.

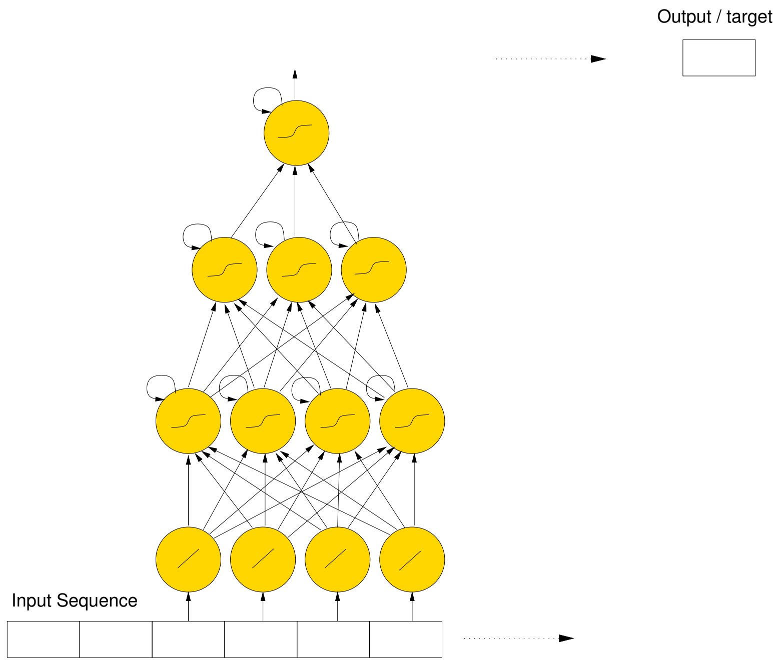
RECURRENT NEURAL NETS (cont'd)

Example of RNN with output sequence:



RECURRENT NEURAL NETS (cont'd)

Example of RNN with single output/target (output emitted only in last step):



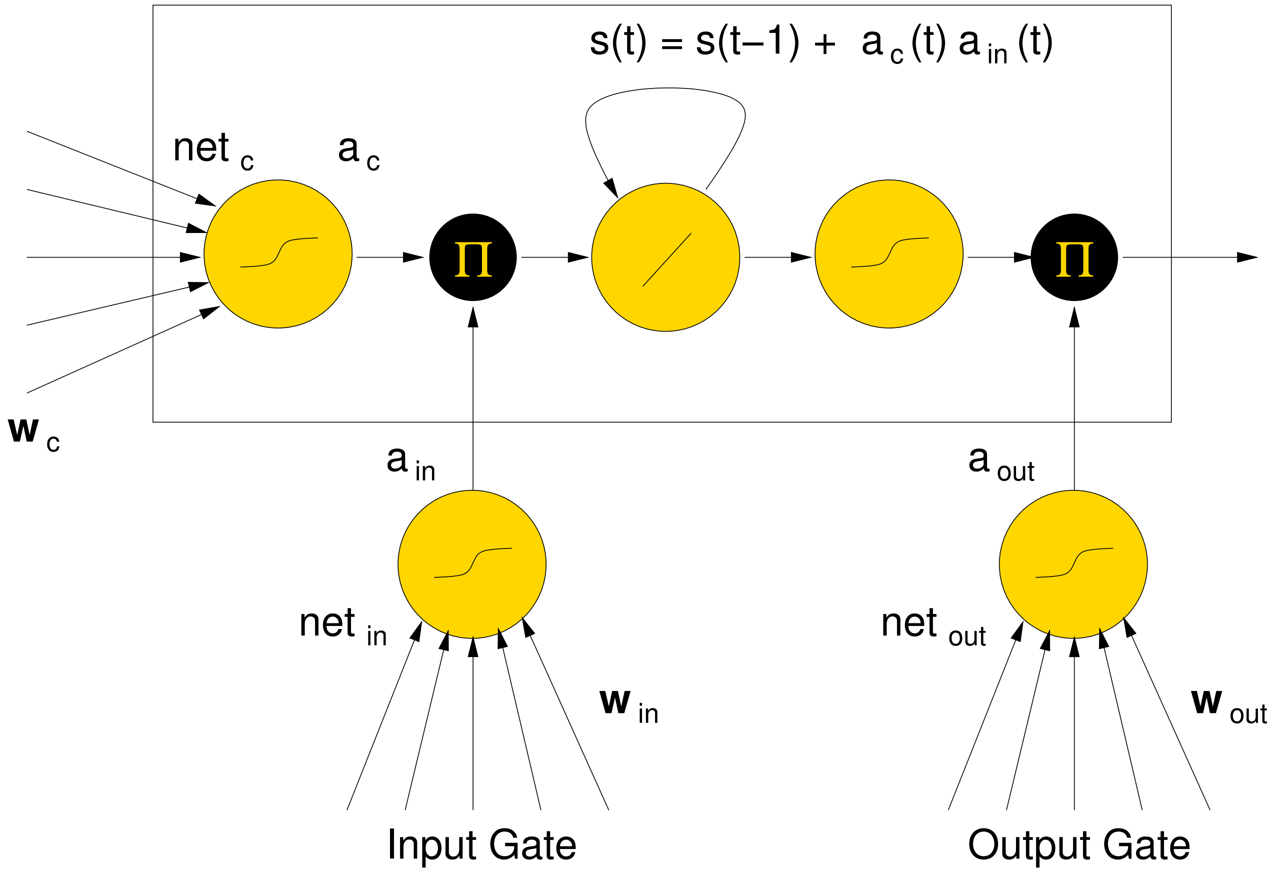
RNNs AND VANISHING GRADIENTS

- Standard RNNs with sigmoid activations are particularly prone to the vanishing gradient problem (actually, this problem has been formulated/discussed for RNNs first): errors/deltas decline (or explode) quickly when back-propagating through time.
- The consequence is that only short time lags between inputs and output signals can be learned correctly (up to about 10 time steps).

LONG SHORT-TERM MEMORY (LSTM)

- In order to overcome the vanishing gradient problem in RNNs, Hochreiter and Schmidhuber (1997) have introduced *Long Short-Term Memory (LSTM)* networks.
- Apart from a standard input unit, an LSTM memory cell has three main components:
  1. A *linear self-connected memory unit* (the linear activation facilitates constant error flow and thereby avoids vanishing gradients)
  2. A *multiplicative input gate* to protect the memory cell from irrelevant inputs
  3. A *multiplicative output gate* to protect the outputs (or other connected untis) from currently irrelevant memory contents

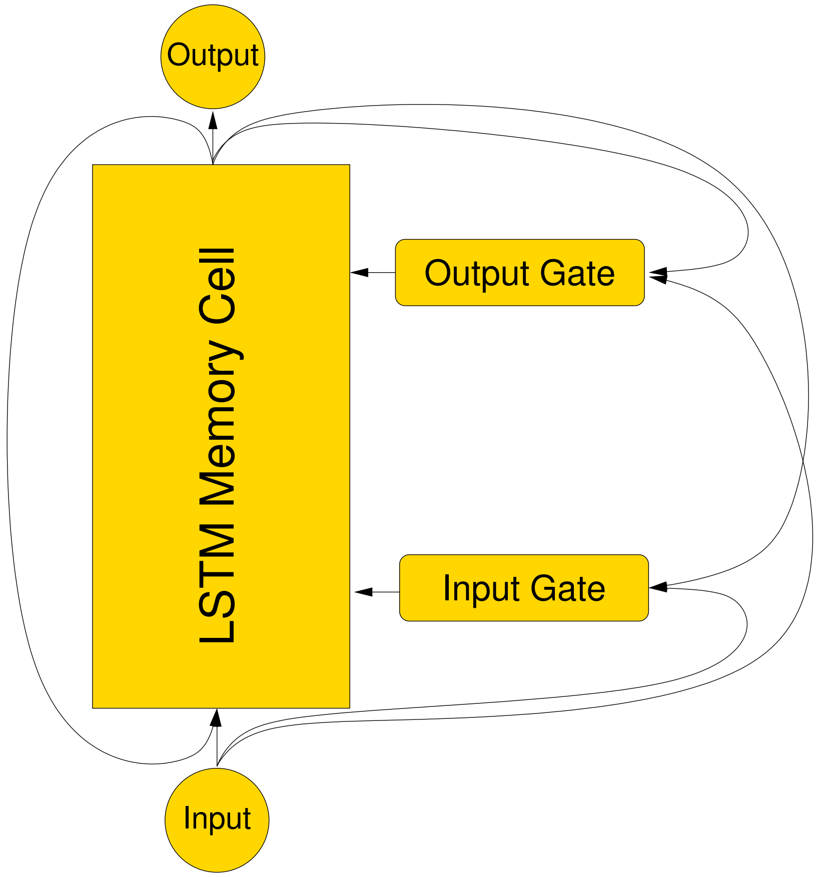
ARCHITECTURE OF AN LSTM MEMORY CELL



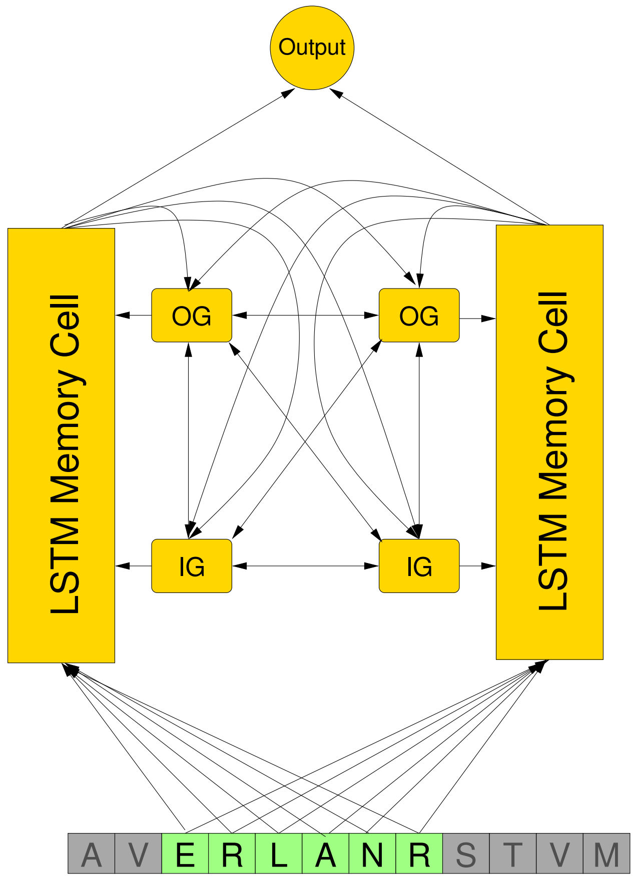
LONG SHORT-TERM MEMORY (LSTM; cont'd)

- LSTM memory units are trained with a so-called *truncated backpropagation* algorithm.
- Special update rules are necessary to account for the multiplicative gates.
- By avoiding the vanishing gradient problem, LSTM networks can learn across time lags of 1,000 or more time steps.
- The LSTM memory cell is just a building block. LSTM memory cells can be combined in various ways (see examples below).
- Since the introduction of LSTM, various variants have been introduced (e.g. forget gates, peepholes).

EXTREMELY SIMPLE LSTM NET FOR TIME SERIES ANALYSIS



LSTM NETWORK FOR SEQUENCE CLASSIFICATION



LSTM: THE CURRENT HYPE

Some benchmark records of 2014 achieved by LSTM:

- Text-to-speech synthesis (Fan *et al.*, Microsoft, Interspeech 2014)
- Language identification (Gonzalez-Dominguez *et al.*, Google, Interspeech 2014)
- Large vocabulary speech recognition (Sak *et al.*, Google, Interspeech 2014)
- Prosody contour prediction (Fernandez *et al.*, IBM, Interspeech 2014)
- Medium vocabulary speech recognition (Geiger *et al.*, Interspeech 2014)
- English to French translation (Sutskever *et al.*, Google, NIPS 2014)
- Audio onset detection (Marchi *et al.*, ICASSP 2014)
- Social signal classification (Brueckner & Schulter, ICASSP 2014)
- Arabic handwriting recognition (Bluche *et al.*, DAS 2014)
- Image caption generation (Vinyals *et al.*, Google, 2014)
- Video to textual description (Donahue *et al.*, 2014)

LSTM: THE CURRENT HYPE (cont’d)

LSTM @ Google:

- Neural Machine Translation System (NMT)
- Google Voice Transcription (Android speech recognizer)

LSTM @ Microsoft:

- Photo-real talking head with deep bidirectional LSTM
- Spoken language understanding using LSTM
- Text-to-speech synthesis with bidirectional LSTM-based RNN

LSTM @ facebook:

- Text analysis

LSTM @ Apple:

- Siri

EXAMPLE: IMAGE CAPTIONS

"man in black shirt is playing guitar."

"construction worker in orange safety vest is working on road."

"two young girls are playing with lego toy."

"boy is doing backflip on wakeboard."

"girl in pink dress is jumping in air."

"black and white dog jumps over bar."

"young girl in pink shirt is swinging on swing."

"man in blue wetsuit is surfing on wave."

OPEN-SOURCE SOFTWARE FRAMEWORKS (SELECTION)

CAFFE: by Berkeley Vision and Learning Center; interfaces for C++, command line, Python, and MATLAB;

LASAGNE: lightweight interface for Theano (see below);

MXNet: by Distributed (Deep) Machine Learning Community; interfaces for C++, Python, Julia, Matlab, JavaScript, Go, R, and Scala;

TensorFlow: by Google Brain; Python interface;

Theano: by Université de Montréal; Python interface;

Torch: by R. Collobert, K. Kavukcuogl, and C. Farabet; based on the Lua programming language; interface for Lua and C;

All of these frameworks support running code on GPUs (via CUDA); beside fully connected networks, all feature CNNs and RNNs.