## History — early models of artificial neural networks

Historically the first neuron model introduced in 1943 (McCulloch and Pitts) was able to recognize two categories of objects based on thresholding values for the function $f(\boldsymbol{x}) = \Sigma_i\, w_i x_i$. However, the weights had to be selected by the operator.

In the late 1950s, a perceptron appeared (Rosenblatt) who was able to learn the correct weights based on samples of different categories (but only for a single-layer input/output network).

The ADALINE neuron model was introduced almost simultaneously (Widrow and Hoff) (*Adaptive Linear Neuron*). It returned the value $f(\boldsymbol{x})$ trying to predict a real number based on input values, and learned its weights from data.

## ANN history: the first wave of enthusiasm

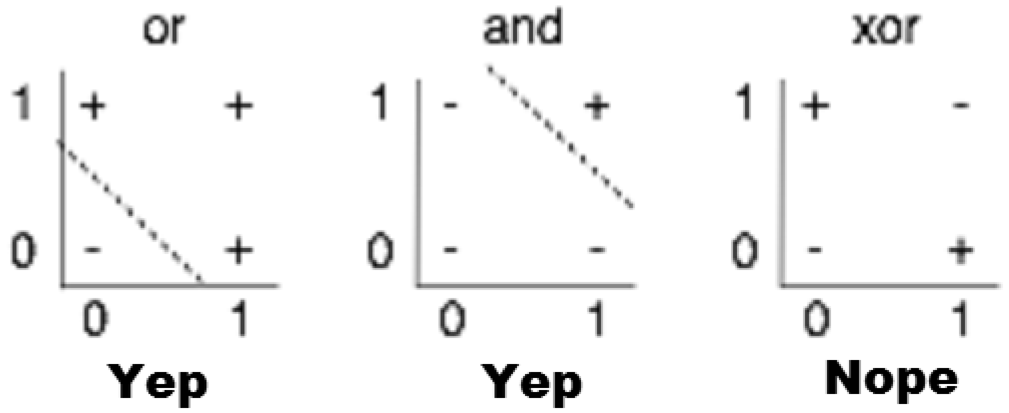
These early models ignited the first revolutionary wave of interest in artificial neural networks. This revolution appeared in times of the early computers, and was based mainly on promises and predictions, rather than on specific projects and applications.

A quote from the New York Times, July 8, 1958, “New Navy Device Learns By Doing”

> *“The Navy revealed the embryo of an electronic computer today that it expects will be able to walk, talk, see, write, reproduce itself and be conscious of its existence ... Dr. Frank Rosenblatt, a research psychologist at the Cornell Aeronautical Laboratory, Buffalo, said Perceptrons might be fired to the planets as mechanical space explorers”*

## ANN history: first disappointment

As might be expected, this enthusiasm quickly disappeared, especially after the publication of the famous article by Marvin Minsky and Seymour Papert *"Perceptrons,"* presenting a technical and critical analysis of the abilities of artificial neural networks. One of the key arguments was the inability of a (single layer) neural network to compute such simple logical function as XOR (Exclusive-OR):



The disappointment with neural networks, after the first wave of enthusiasm, was so deep that neural networks have ceased to be a popular topic for scientific research. For over 10 years (1970-1986) the scientific world actually ignored emerging independent results with the backpropagation algorithm.

## ANN history: second revolution

It wasn't until 1986 that the paper *"Learning representations by back-propagating errors "* (Rumelhart, Hinton, Williams) stirred new interest, or perhaps the second revolution in artificial neural networks. The error backpropagation algorithm allowed for effective learning in multilayer neural networks, overcoming the XOR function problem, and many other barriers.

This revolution resulted in an incredible amount of publications, projects, and applications, applying neural networks in many areas of science and technology. Just spelling out "artificial neural networks" almost guaranteed acceptance of a paper for publication, or obtaining funding for a research project.

This revolution was called the **connectionism**.

## ANN history: the second chill

In the late 1990s, the enthusiasm of the scientific world related to artificial neural networks began to disappear. ANN have become stable technology, their properties were well known, and new sensational reports ceased to appear.

The technical capabilities of artificial neural networks were limited by the then existing computer technology. For implementing even a minimally complex calculation, a multilayer network was needed, but training many layers with a large number of neurons was computationally impractical (weeks, months, or years of calculations). The view also dominated that the computational ability of networks with more than one hidden layer did not exceed those networks with just one hidden layer.

It should be noted that the 1990s are also the beginning years of the internet revolution. Access to the network and software resources were expanding, as were the new capabilities they allowed. It was not yet the „*big data*" era and creating neural networks with larger capabilities was limited by the amount of data available. Existing datasets stored thousands of samples at most, and the limit for obtaining more interesting results was the compromise between the noise in data and avoiding overfitting.

## ANN history: third revolution — deep networks

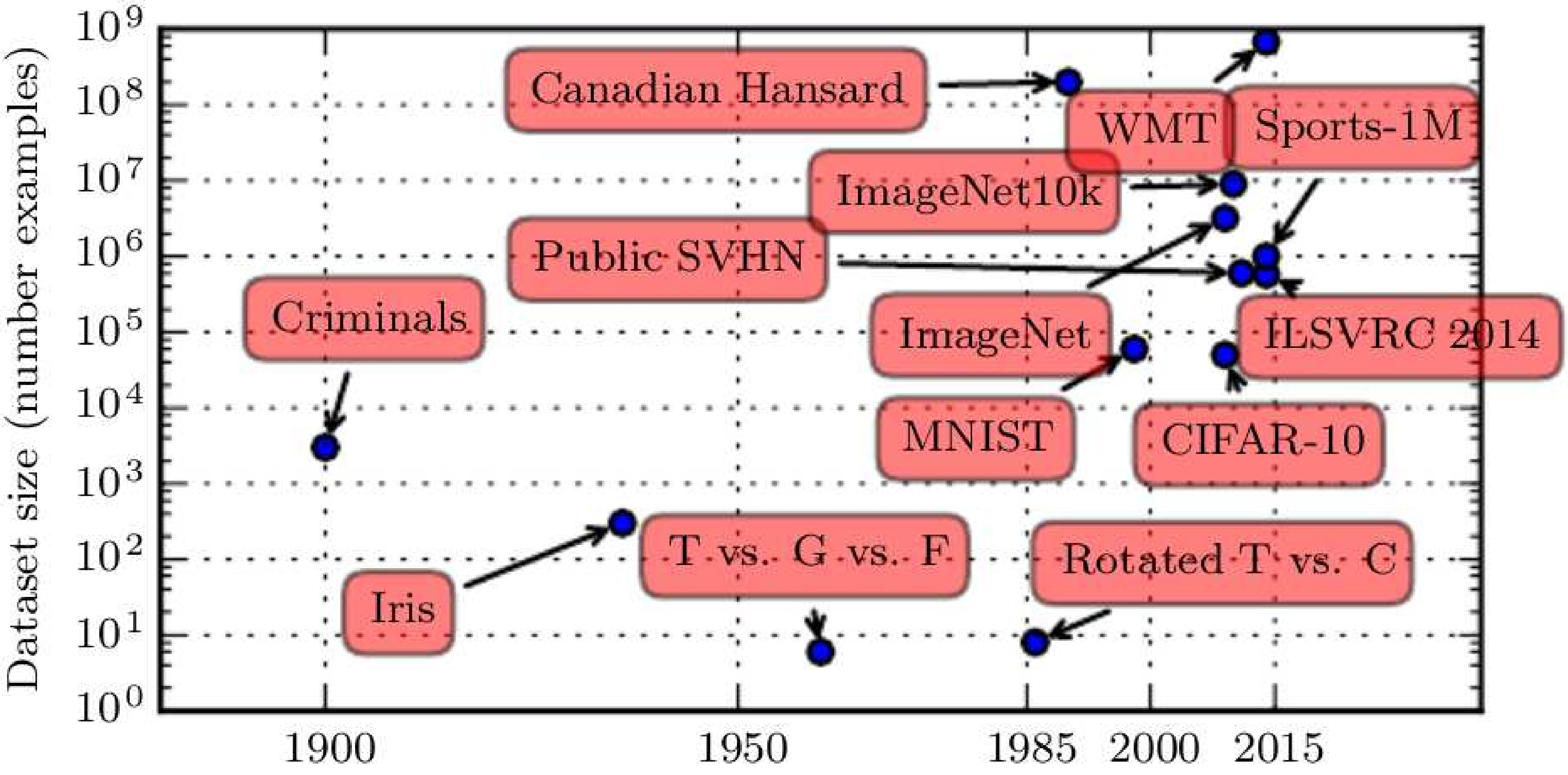
Despite the vanishing enthusiasm of the scientific world for the artificial neural networks, in the 1990s a number of new technologies were created, which were yet to find wider applications. An example is a distributed representation where dedicated neurons are responsible for coding separate aspects of an object representation (such as shape or color). Another example is the LSTM network (Long Short-Term Memory) that solves problems of modeling long sequences.

Another breakthrough began around 2006, when **deep networks** and effective methods of teaching them began to appear. Initial such networks used unsupervised learning algorithms and demonstrated the possibility of effective generalization based on small data sets.

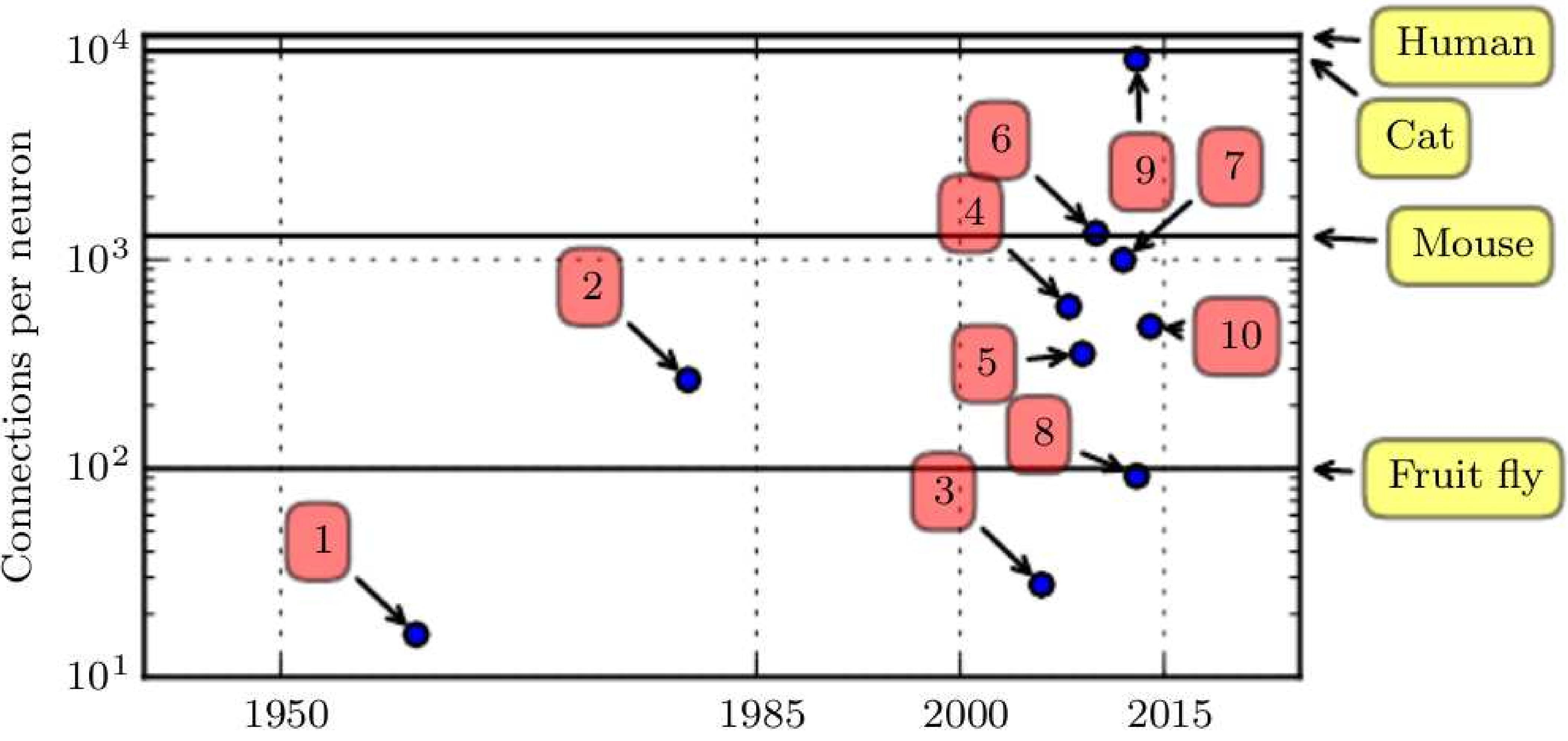
Since then, the mainstream has turned to exploitation new, huge data sets, but using the methods successfully applied before, such as the multilayer perceptron and learning with the backpropagation algorithm.

It should be noted that this breakthrough has become possible partly thanks to progress in the development of computer technology. A significant share of it was due to the emergence of the GPU graphics cards with thousand of cores, which are intended for calculating image elements, but can be used for any calculations.

## Datasets increase

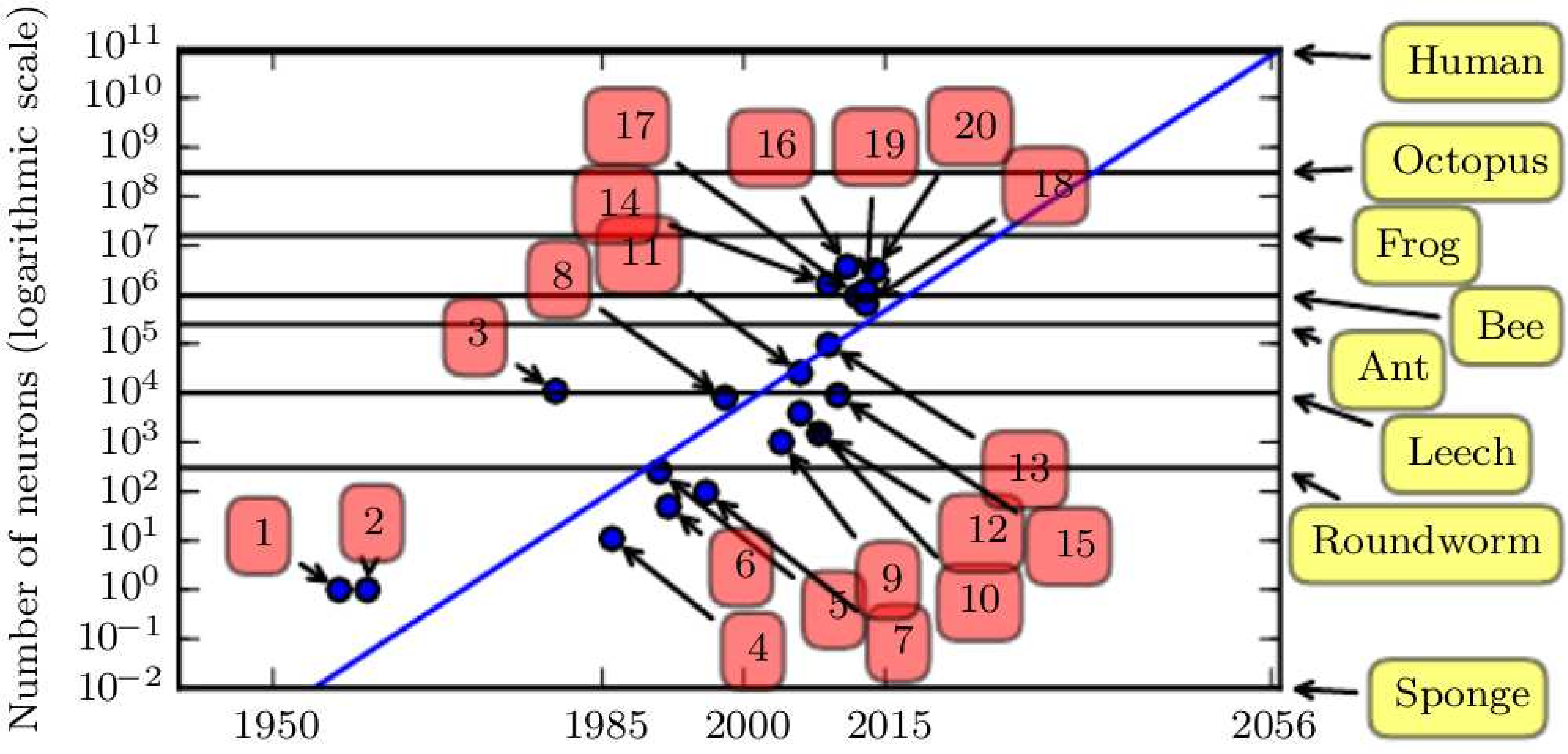


# Number in connections



1. Adaptive linear element (Widrow, Hoff, 1960)
2. Neocognitron (Fukushima, 1980)
3. GPU-accelerated convolutional network (Chellapilla et al., 2006)
4. Deep Boltzmann machine (Salakhutdinov, Hinton, 2009a)
5. Unsupervised convolutional network (Jarrett et al., 2009)
6. GPU-accelerated multilayer perceptron (Ciresan et al., 2010)
7. Distributed autoencoder (Le et al., 2012)
8. Multi-GPU convolutional network (Krizhevsky et al., 2012)
9. COTS HPC unsupervised convolutional network (Coates et al., 2013)
10. GoogLeNet (Szegedy et al., 2014a)

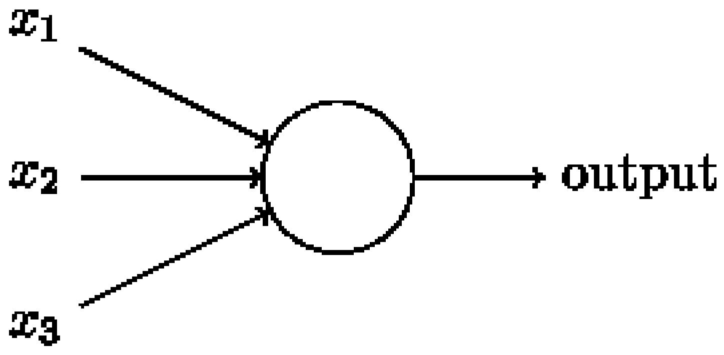
## Increase in network size



1. Perceptron (Rosenblatt, 1958, 1962)
2. Adaptive linear element (Widrow, Hoff, 1960)
3. Neocognitron (Fukushima, 1980)
4. Early back-propagation network (Rumelhart et al., 1986b)
5. Recurrent neural network for speech recognition(Robinson,Fallside,1991)
6. Multilayer perceptron for speech recognition (Bengio et al., 1991)
7. Mean field sigmoid belief network (Saul et al., 1996)
8. LeNet-5 (LeCun et al., 1998b)
9. Echo state network (Jaeger, Haas, 2004)
10. Deep belief network (Hinton et al., 2006)
11. GPU-accelerated convolutional network (Chellapilla et al., 2006)
12. Deep Boltzmann machine (Salakhutdinov, Hinton, 2009a)
13. GPU-accelerated deep belief network (Raina et al., 2009)
14. Unsupervised convolutional network (Jarrett et al., 2009)
15. GPU-accelerated multilayer perceptron (Ciresan et al., 2010)
16. OMP-1 network (Coates, Ng, 2011)
17. Distributed autoencoder (Le et al., 2012)
18. Multi-GPU convolutional network (Krizhevsky et al., 2012)
19. COTS HPC unsupervised convolutional network (Coates et al.,2013)
20. GoogLeNet (Szegedy et al., 2014a)

## The perceptron

**The perceptron** is one of the first neuron models used in artificial neural networks (ANN). It is a simple computational element implementing a weighted sum of many inputs, combined with thresholding:



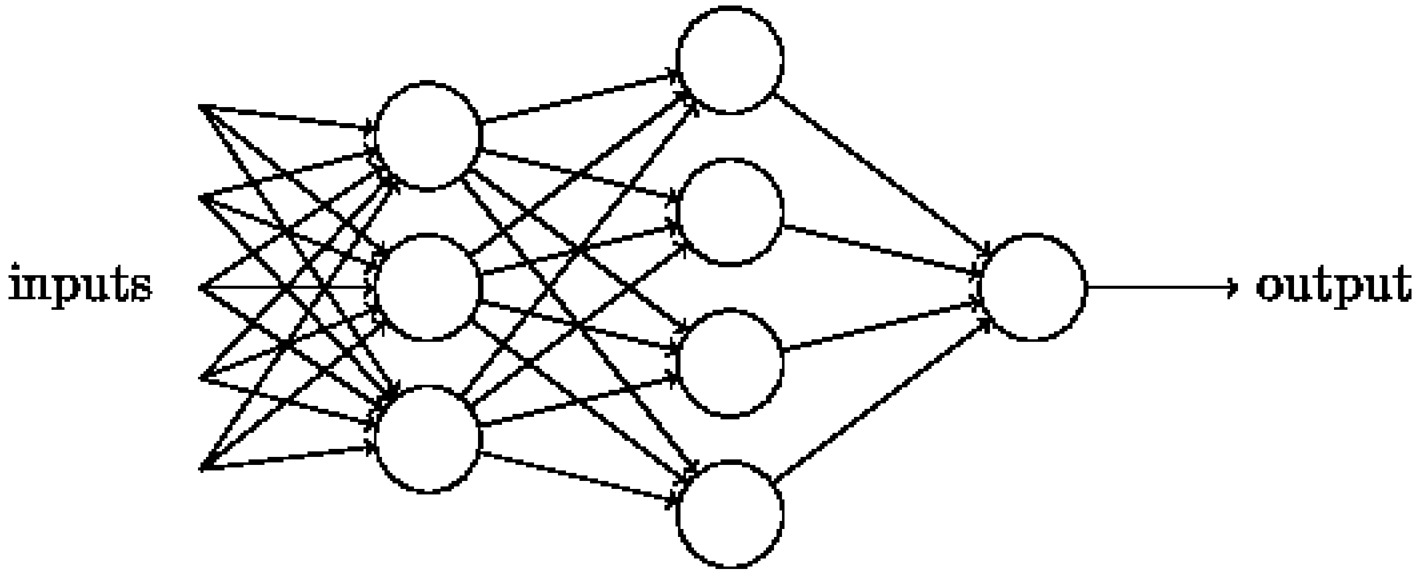
$$\text{output} = \begin{cases} 0 & \text{if } \Sigma_j w_j x_j \leq \text{threshold} \\ 1 & \text{if } \Sigma_j w_j x_j > \text{threshold} \end{cases}$$

The perceptron is the simplest decision element, calculating decisions based on the weighted sum of input variables. Choosing the appropriate weights $w_1, \ldots, w_n$ and the threshold value, we can model some simple decision-making schemes. Input signals having a large positive effect on the desired output value are given large weight values, and those inputs having a smaller effect on the output value are assigned smaller weights.

And, more importantly, we can <u>learn</u> the right values of these parameters based on the appropriate training set.

## Multilayer perceptron

Before we get to the methods of automatically training neural networks, let's consider a more complex case. A single perceptron can model making very simple decisions by calculating weighted sums with thresholding. Since these calculations are quite trivial, we can imagine many such elements connected together, for the purpose of calculating more complex decisions. One possible such model is the **multilayer perceptron MLP**.



MLP consists of a series of layers of neurons, of which the first (to which input signals are connected) we call the input layer, and the last one (from which the output signals are drawn) is called the output layer, and the intermediate layers (which can be many) are called hidden layers.

## Multilayer perceptron (ctd.)

Thanks to its layered structure, the decision making model in the MLP can be much more complex. Entry layer neurons, instead of making final decisions, may recognize some elementary features of the input signal. Similarly, subsequent layers can calculate more and more complex quantities, and the output layer neurons can make the final decisions based on far more significant categories than raw input variables.

## Neuron activation function

The perceptron activation formula can be rewritten into a slightly more convenient form:

$$\text{output} = \sigma(w^T \cdot x + b) = \begin{cases} 0 & \text{if } (w^T \cdot x + b) \leq 0 \\ 1 & \text{if } (w^T \cdot x + b) > 0 \end{cases}$$

where the form $w^T \cdot x$ denotes the scalar product of the weights and input signal vectors, the parameter $b$ (bias) is the inverse of the threshold value, the entire value of $(w^T \cdot x + b)$ is called the **weighted sum of inputs**, and the function $\sigma$ is called **activation function**.

The activation function $\sigma$ given by the above formula is called the **step function**.

Note that the bias parameter can be considered as the weight of an additional input with a fixed value of 1. It then disappears from the formula, at a cost of a larger network. It is also possible to use a network with the value of bias equal to 0 — in other words without the bias — if we know that such network correctly models the given decision-making process.

# Sigmoid activation function

The presented step activation function can correctly model some decision making processes, but is not suitable for automatic network training. If some weights, or bias, are slightly different from optimal values, two cases are possible. Either the perceptron gives the same response as for optimal values, and then the network will seemingly work fine, but it is not possible to detect any difference and make even small corrections. For data outside the training set we will obtain errors. Or the perceptron will "jump" to another output value, and then the network will operate incorrectly, but we will not be able to notice that the parameter values were close to correct.

step

$$\sigma(x) = \begin{cases} 0 & \text{iff } x \le 0 \\ 1 & \text{iff } x > 0 \end{cases}$$

sigmoid

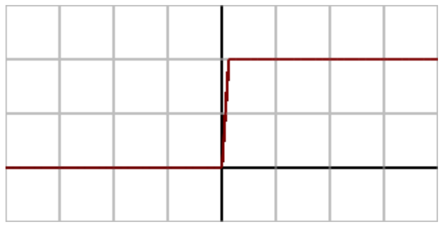
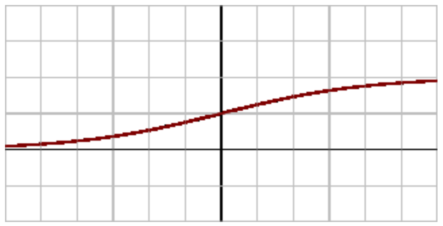
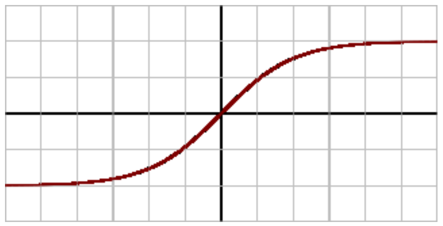
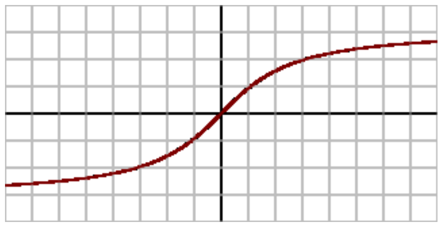
$$\sigma(x) = \frac{1}{1+e^{-x}}$$

tanh

$$\begin{aligned} \sigma(x) &= tanh(x) \\ &= \frac{e^x - e^{-x}}{e^x + e^{-x}} \end{aligned}$$

arctan

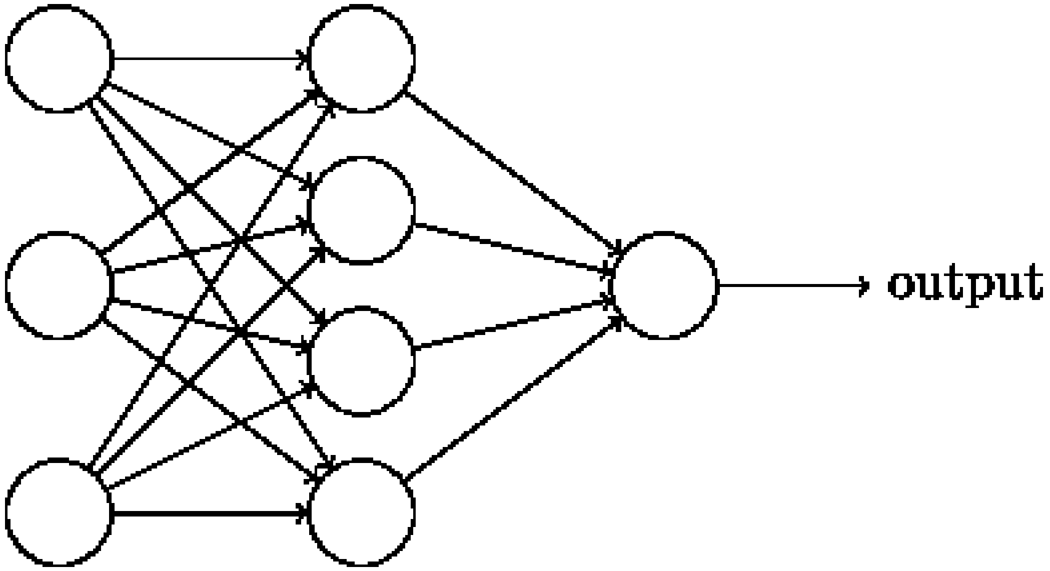
$$\sigma(x) = tan^{-1}(x)$$

Often used is the sigmoid activation function, also called the logistic curve.

## Neural network architecture

The MLP multilayer perceptron discussed so far, more generally called a **feedforward network** is an example of the neural network architecture — one of the most popular — but not the only one.



Such a network consists of at least two layers: input and output, and, optionally, of a number of intermediate layers (in the figure above there is one of them), also referred to as the hidden layers. (They are not hidden in any special sense; the term “hidden” only means that they do not belong to the external interface of the neural network.)

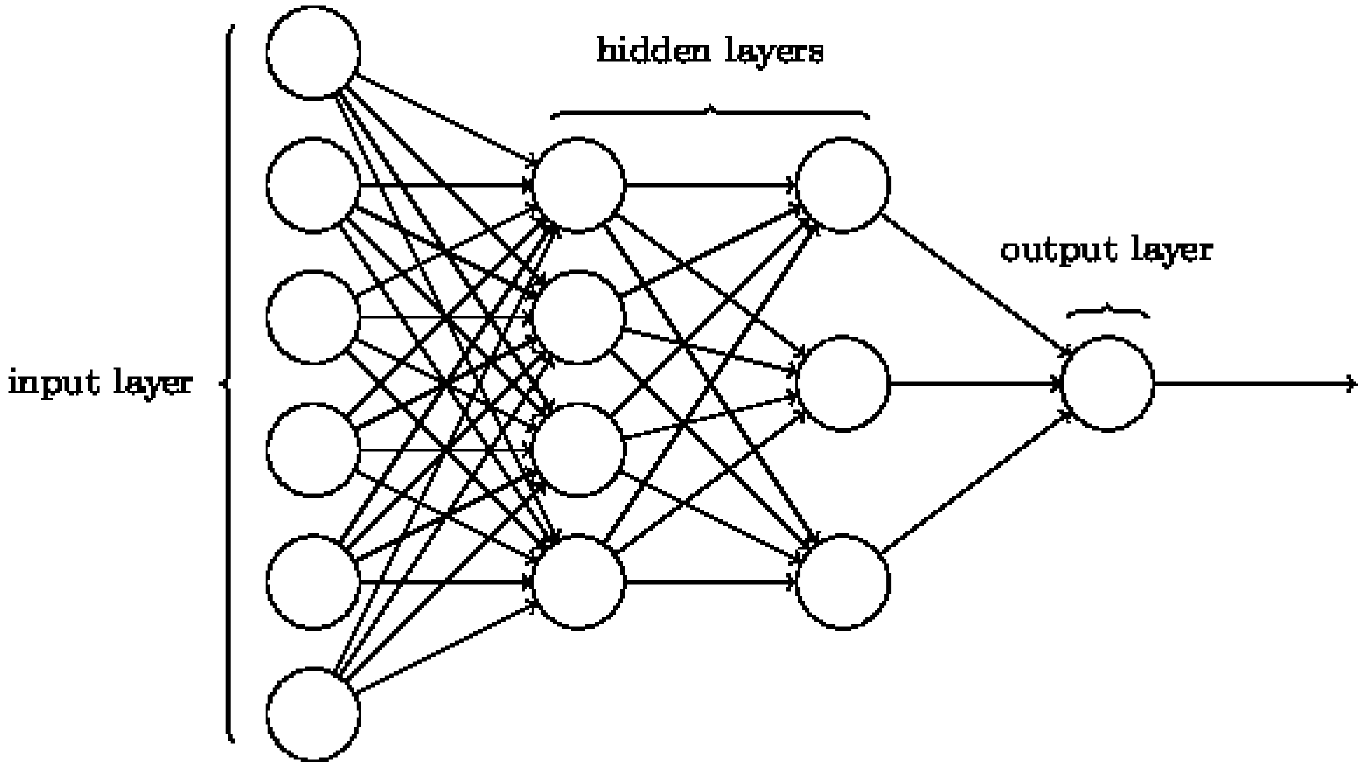
## Other network architectures

Feedforward networks are simple and popular, but by no means they are the only model of neural networks. At least equally interesting are **recursive networks** in which the signal from neurons from further layers can be sent back to the initial layer. Due to the signal propagation time resulting from nonzero excitation time of neurons, such loops do not lead to infinite oscillations, they only create possibilities for completely different computing processes.

Additions: examples of recursive networks.

## Feedforward networks

The architecture of a feedforward network is directly related to its application. A network to be used as a binary classifier of the objects described by six parameters should have six neurons in the input layer, activated by the appropriate input parameters. Its single output will be interpreted as the object class value (true or false).

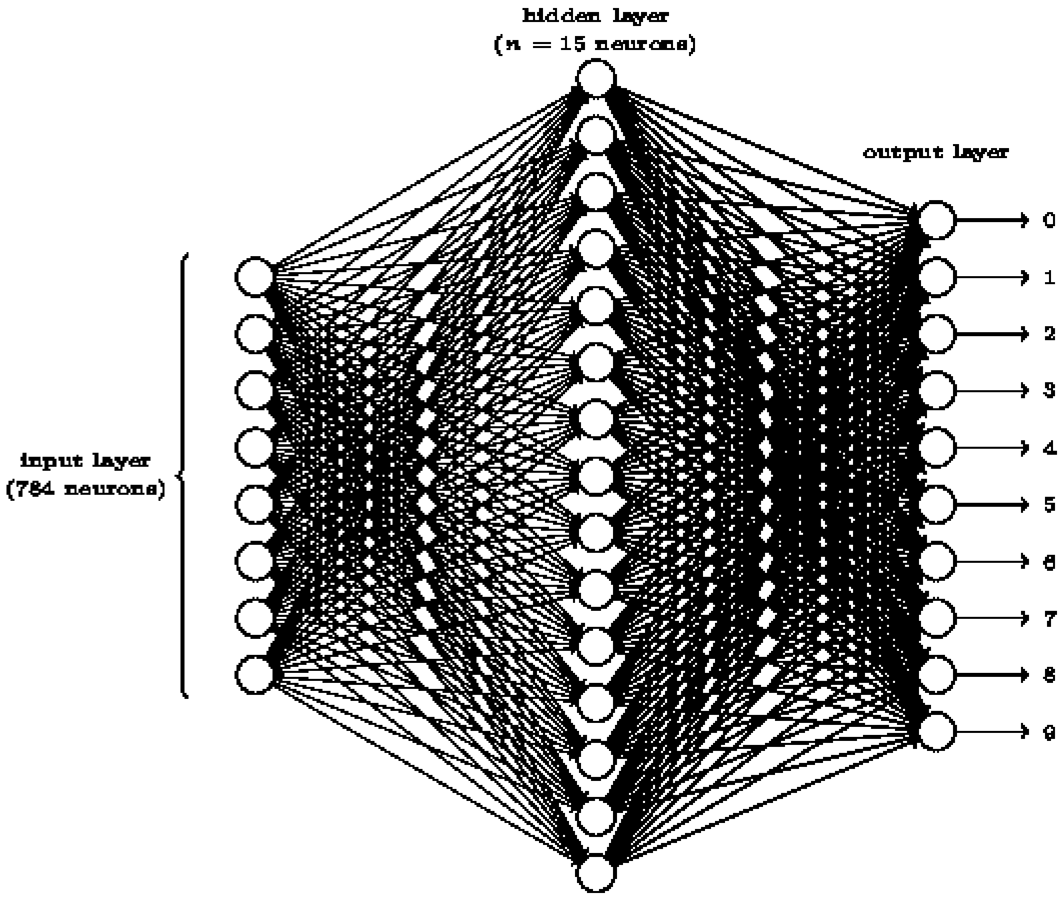


Note that the network has all possible connections between the neurons of subsequent layers. This is how a network is typically initialized. Then, during the training, some of the connections may be turned off, by reducing their weights to zero, or near zero.

## Feedforward networks: example

Let's now consider a specific neural network for handwriting recognition. The network is to determine which decimal digit is depicted in a $28 \times 28$ raster image, grayscale encoded.

The simplest approach is to use all pixels of the image as separate input signals. So the network input layer should have $28 \times 28 = 784$ neurons. They can be driven directly by the appropriate image pixel brightness values, for some reasons scaled to the range of $[0, 1]$. The output layer will have ten neurons representing individual digits, with the threshold set to $0.5$.



## Feedforward networks: construction of internal layers

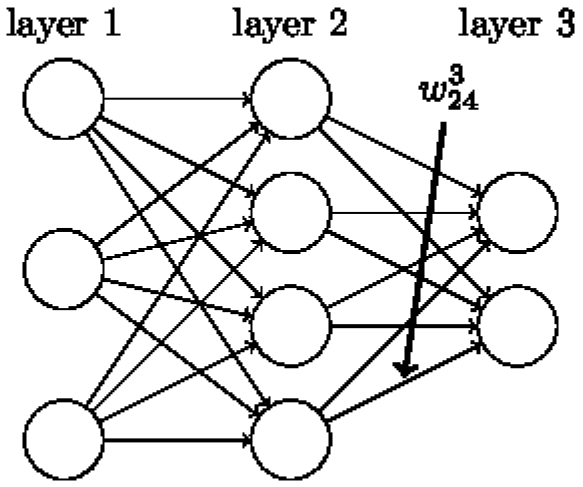
As we have seen, the construction of input and output layers, constituting the interface of the neural network is determined by the use of the network and existing/required external signals. The construction of the internal layer(s) requires more complex considerations. On the one hand, the number of internal layers and neurons in them determines the ability of the network to implement a specific calculation. A network with too few neurons will not be able to implement complex processes. On the other hand, a network with too many neurons will tend to learn very long and to overfit.

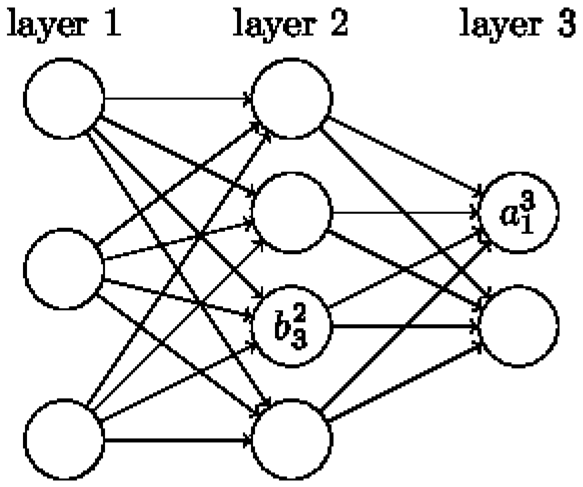
Generally, we want the inner layer to have fewer neurons than the input layer, to force the network to generalize — build a more complex representation of input data.

The division of the internal neuron pool into layers is determined by other rules. The traditional approach (until around 2010) was to have one or two internal layers, depending on the complexity of the computation process. The rationale was that more layers with the same pool of neurons would greatly extend the training, without significantly increasing the ability of the network to implement complex processing.

## Training neural networks — designations

We will use the following designations for the weights of the connections between neurons: $w^l_{jk}$ will denote the weight of the connection from neuron $k$ in $(l-1)$th layer to neuron $j$ in $l$th layer:





Likewise, we will use $b^l_j$ and $a^l_j$, respectively, to denote the bias and activation of the neuron $j$ in the $l$th layer. In this notation, activation of the neuron $j$ in layer $l$ is given by the formula:

$$a^l_j = \sigma(\sum_k w^l_{jk} a^{l-1}_k + b^l_j) \quad \text{in matrix algebra notation:} \quad a^l = \sigma(w^l a^{l-1} + b^l)$$

## Training Neural Networks — the cost function

Learning algorithms typically choose weights and bias values to achieve the minimum of the following quadratic cost function, sometimes also referred to as the **mean square error (MSE)**:

$$C = \frac{1}{2n} \sum_x \parallel y(x) - a^L(x) \parallel^2$$

where $n$ is the number of samples $x$ of the training set, $y = y(x)$ is the desired output value, $L$ is the output layer number (or the number of layers), and $a^L = a^L(x)$ is the activation vector of the network output for the input sample $x$.

The above formula can be treated as the average cost value $C = \frac{1}{n} \Sigma_x C_x$ of the quadratic cost function values of all individual samples $x$: $C_x = \frac{1}{2} \parallel y - a^L \parallel^2$.

The **backpropagation** algorithm presented below determines the error $\delta_j^l$ of the computation of neuron $j$ in layer $l$, and on this basis partial derivatives $\partial C / \partial w_{jk}^l$ and $\partial C / \partial b_j^l$. As a result, updates for $w_{jk}^l$ and $b_j^l$ minimizing the cost function can be computed by the maximum gradient method. Frequently the calculations are averaged over a certain small batch of samples, which is called the **stochastic gradient descent method.**

## Training neural networks — the backpropagation algorithm

**Step 0 (initialize):** for the sample $x$ set the input layer activations $a^1$.

**Step 1 (propagate):** for subsequent layers $l = 2, 3, ..., L$ calculate $z^l = w^l a^{l-1} + b^l$ and $a^l = \sigma(z^l)$.

**Step 2 (output error):** calculate the error vector $\delta^L = \nabla_a C \odot \sigma'(z^L)$.
(the symbol $\odot$ means the Hadamard product, available in Matlab as `.*`)

**Step 4 (backpropagate error):** going back, for subsequent layers $l = L, L-1, ..., 3, 2, 1$ calculate error vectors $\delta^l = ((w^{l+1})^T \delta^{l+1}) \odot \sigma'(z^l)$.

**Step 5 (corrections):** The cost function gradient is given as $\frac{\partial C}{\partial w^l_{jk}} = a^{l-1}_k \delta^l_j$ and
$\frac{\partial C}{\partial b^l_j} = \delta^l_j$
so the negations of these gradients should be added as corrections to weights $w^l_{jk}$ and biases $b^l_j$, with some low learning rate $\eta$, providing stable learning (immunity to noise in training data).

Additions: update formulas for mini-batch

## Network training improvements

Humans learns much faster when they see they make big errors. Their reactions are determined, and the corrections they make are significant. When the errors they make become minimal, the learning process slows down. The question is whether a similar phenomenon can be observed in training artificial neural networks.

As we saw in the back propagation algorithm, weights and bias corrections are determined by the values of $\frac{\partial C}{\partial in}$ and $\frac{\partial C}{\partial b}$, where a quadratic cost function is given by the formula (again adjusted to the case when the desired output value is $y = 1.0$):

$$C = \frac{(y - a(x))^2}{2}$$

where $a(x) = \sigma(wx + b)$ is the neuron activation value.

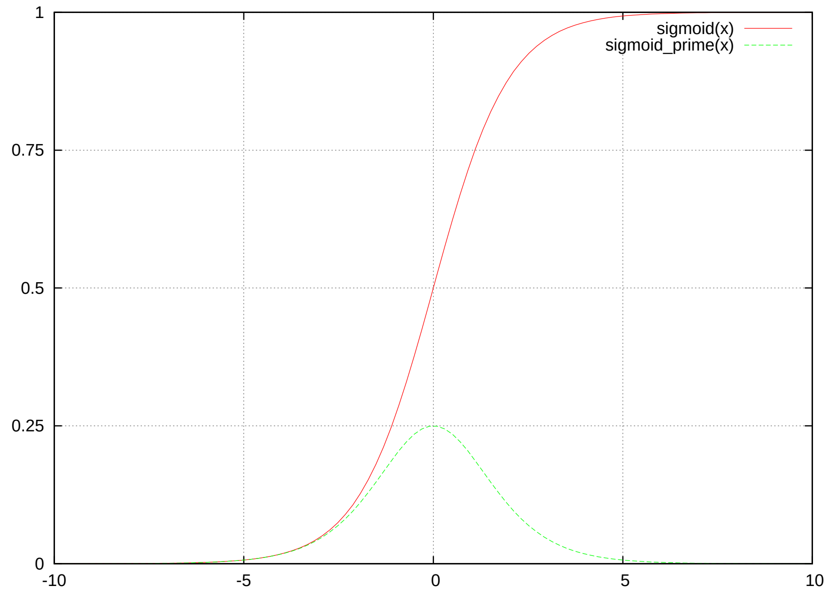
The above partial derivatives of the cost function can be calculated as proportional to the derivative of the activation function $a'(x)$. So the magnitude of the corrections made in the training process are associated with the activation function derivative.

## The derivative of the sigmoid function

The derivative of a sigmoid function can be calculated as follows. (Note that that the second derivative form is important in the training process. The backpropagation algorithm calculates all activation values during the first propagation phase. They can be stored by the algorithm, and used for quick calculation of derivative values in the backpropagation phase.)

$$\begin{aligned}\sigma(x) &= \frac{1}{1+e^{-x}} \\ \sigma'(x) &= \frac{1}{1+e^{-x}}\left(1-\frac{1}{1+e^{-x}}\right) \\ \sigma'(x) &= \sigma(x)\cdot(1-\sigma(x))\end{aligned}$$

```
$ gnuplot
sigmoid(x)=1/(1+exp(-x))
sigmoid_prime(x)=sigmoid(x)*(1.-sigmoid(x))
plot sigmoid(x),sigmoid_prime(x)
```



As can be seen, when the network is far from the desired value $y = 0.0$, the sigmoid function is flat, and its derivative has very low values. For this reason, the training progress is very slow when the network has large errors, and only accelerates in the immediate vicinity of the target values.

## The cross-entropy cost function

The slowdown in multi-layer network training can be seen by running a small example: http://neuralnetworksanddeeplearning.com/chap3.html#saturation2_anchor

This problem of network training slowdown can be solved by using, in place of the mean square cost function, the following **cross-entropy** function:

$$C = -\frac{1}{n} \sum_x \left[ y \ln a + (1-y) \ln(1-a) \right]$$

where the sum is computed over all $n$ training samples $\langle x, y \rangle$.

The sense of using this function results from some statistical properties that will not be discussed here. Let us only note that the above function: (i) is non-negative, (ii) goes down to zero near the correct solution (assuming that the desired value $y = 0.0$ and then $a \approx 0.0$).

It can be obtained that the partial derivatives $\frac{\partial C}{\partial w}$ and $\frac{\partial C}{\partial b}$ are in this case proportional to the difference between the correct output value and the neuron activation. This causes larger updates when the network is far from desirable region, and reducing them as we approach the solution.

The formula for the total cost (error) for all neurons $j$ in layer $L$ of the network:

$$C = -\frac{1}{n} \sum_x \sum_j \left[ y_j \ln a_j^L + (1 - y_j) \ln(1 - a_j^L) \right]$$

## The total network output value vector interpretation

Neurons of the output layer of a multilayer neural network calculate their values individually, which together constitute the network's response to the input. Neurons change their values smoothly between 0 a 1, but the sigmoid activation function mostly goes to the extreme values of 0 or 1. We may expect one of the neurons to definitely approach 1, and others to go to 0, or to have many neurons to simultaneously approach 1.

The first case corresponds to the classifier function when the network response indicates one specific output out of $n$ possible, while the second case corresponds to any network response function.

It is sometimes beneficial to replace the sigmoid activation function with another function that determines the combined response of all neurons. Then, using the network as a classifier, the vector of the network output values can be treated as the probability distribution of choosing a specific output, instead of their individual indication by a logistic function.

## Application of *softmax* neurons

The above characteristic of network outputs, which can be interpreted as probability distribution, can be achieved by using in the output layer a different activation function called *softmax*:

$$a_j^L = \frac{e^{z_j^L}}{\sum_k e^{z_k^L}}$$

As can be seen, the activation value $a_j^L$ of a single neuron does not depend only on the value of $z_j^L$ (weighted sum of inputs), but also on the activation values of all other neurons. When the activation of one increases, then the activations of all other neurons decrease.

It is easy to see that the sum of the activation values of all neurons is 1:

$$\sum_j a_j^L = \frac{\sum_j e^{z_j^L}}{\sum_k e^{z_k^L}} = 1$$

## The cost function for *softmax* neurons

When using the *softmax* activation function it is appropriate to use another cost function called the **log-likelihood** cost function, which has the value (for a specific learning sample):

$$C = -\ln a_y^L$$

When the network correctly recognizes the value of $y$ for this sample, then the value $a_y^L$ will be close to 1 and the cost will be calculated as 0. When the network does not recognize the sample correctly, then $a_y^L$ will be small, and the inverse of its logarithm will be a large positive value.

It can be shown that the values of partial derivatives of the cost function relative to weights and biases are proportional to the difference between the correct and network-calculated output value, just as in the case of cross-entropy for the logistic function.

So using the *softmax* output layer makes sense with the log-likelihood cost function, just as using the standard sigmoid activation works well with the cross-entropy cost. The advantage of the first combination is the interpretation of the result as a probability distribution.

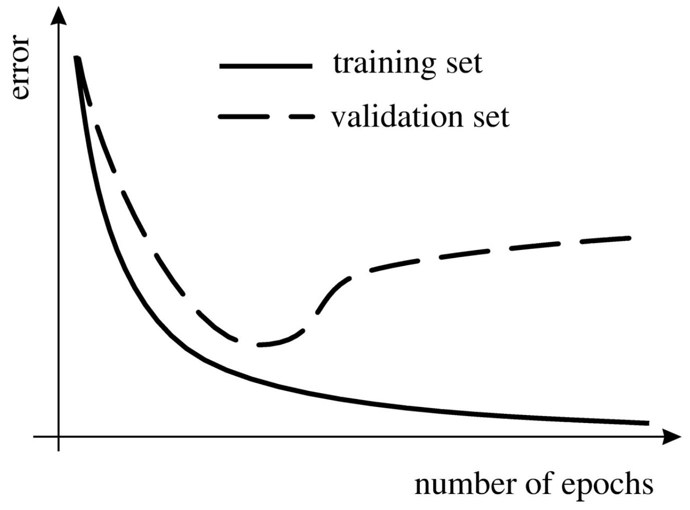
## Overfitting in neural networks

An artificial neural network with many neurons, layers, and connections, has very many parameters. These parameters make it possible to adjust the network response to the desired function in many ways, of which only some (few) may be correct. Even worse, when there are small errors (noise) in the training data, a chance to determine the correct parameters decreases as the network learns, because it tends to match this noise exactly.

This is a problem of overfitting, which we have already seen in different scenarios, and which is one of major machine learning problems.

## Regularization

The standard method of avoiding overfitting is to use a validation dataset,[1] and stop learning when the validation set error begins to increase, while the error on the training set continues to decrease.



However, this is the input/output method, treating the learning algorithm like a black box. It is sometimes possible to embed the overfitting avoidance in the learning algorithm. One such technique is the **regularization**, which involves modifying the cost function to force the network to train in a more desirable way (to make the network function more regular).

[1]Of course, another standard method of avoiding or limiting overfitting is to increase the training set.

## $L_2$ Regularization

One of the most commonly used forms of regularization, called $L_2$ regularization, works by adding a positive component to the cost function according to scheme:

$$C = C_0 + \frac{\lambda}{2n} \sum_w w^2$$

where $C_0$ is the original, non-regularized cost function.

The goal of this is to get the network to prefer small weight values. Large weights will only appear if the original part of the cost function (non-regularized) decreases significantly. The $\lambda$ parameter is used to manage the compromise between the tendency to minimize the main cost function, and obtaining low weight values.

The intuitive justification for regularization is usually the Ockham's razor principle: a model with smaller weights is considered simpler, and if it works as well as a more complex model, then it usually will work better. In reality, however, the correct application of regularization requires experimenting, in particular with the $\lambda$ parameter.

## Remarks on regularization

The effect of regularization is usually:

- avoiding overfitting,
- faster convergence to greater calculations accuracy,
- less sensitivity to the choice of initial values and avoidance of local cost functions minima.

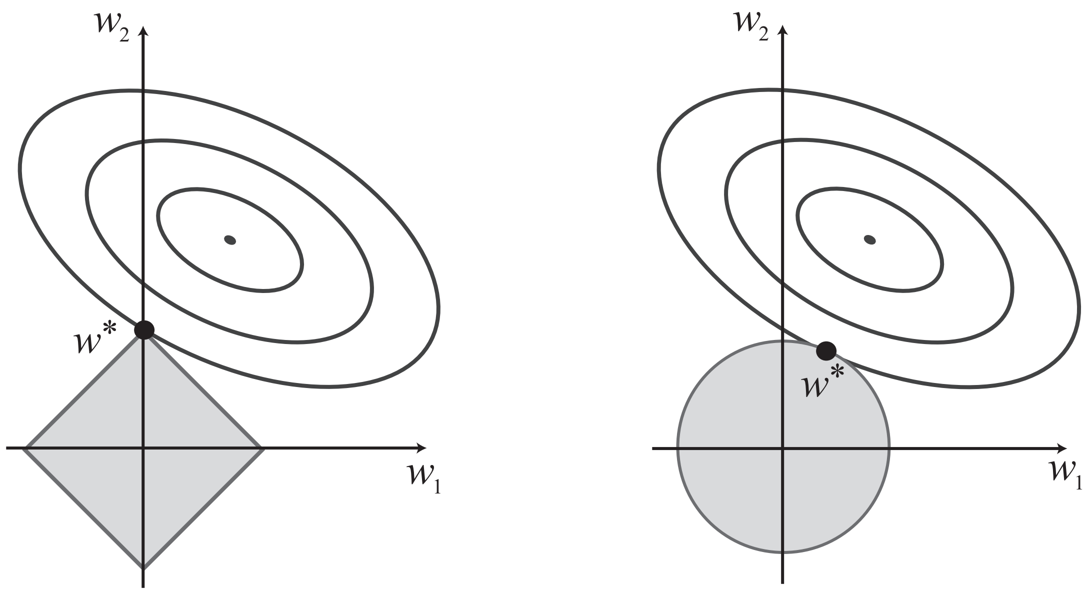
Note that the regularization factor in the formula given above includes weights, but does not include the bias. So the bias, unlike the weights, can have a large value. This is usually not a problem, which is why the bias is not included in the regularization formula.

## $L_1$ Regularization

An alternative approach is to use the so-called $L_1$ regularization according to the formula:

$$C = C_0 + \frac{\lambda}{n} \sum_w |w|$$

In general, $L_1$ regularization works similarly to $L_2$, enforcing smaller weight values. However, $L_1$ has some properties different from $L_2$. With $L_1$ regularization often the result is a model with many weights equal/close to 0. This effect can be explained on the diagram below. Because weight search space with $L_1$ have the shape of an oblique square, the contact point with the minimum of the cost function will often be on any of the coordinate axes.



## $L_1$ and $L_2$ Regularization

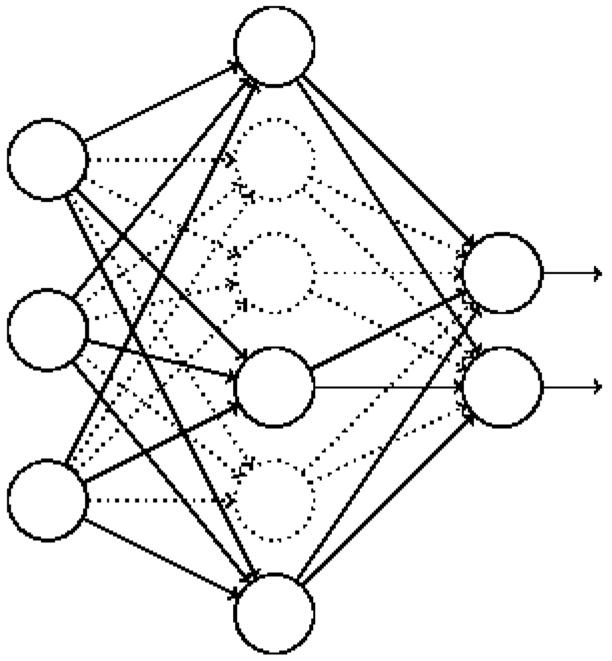
Terminological remark: In machine learning literature, the designations $L_1$ and $L_2$ are often used to denote the absolute and mean square cost functions, as well as, the absolute value and mean square regularization. The context of using these two signage systems are close, but they mean different things.

# The *dropout* method

**Dropout** is a regularization method quite different from the previous ones. It consists in repeatedly removing randomly selected neurons during training (temporarily). After the learning phase with deleted neurons, these are restored, then another random set of neurons removed, and learning continued. The final trained network works with all original neurons.

For example, removing each time half of the neurons from the selected hidden layer for training purposes, in the actual application the network will have twice as many active neurons as at the time of training.

This can be compensated by dividing all learned weights of this layer by two.



The *dropout* procedure is a heuristic method and like other similar methods require testing of the details of its use. However, overall, it produces similar effects to other regularization methods.

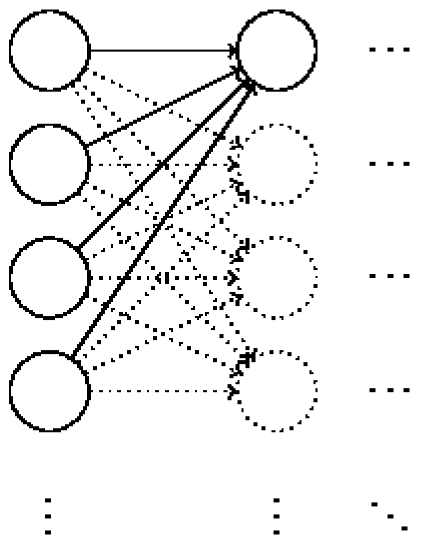
There are several interpretations of this effect. Basic interpretation assumes that individual neurons need to “learn” more reliable activation function (weights and bias) because they cannot count on assistance from other neurons.

Multiple network training, however more expensive (because each *dropout* phase causes a restart and initiates training from start, in a sense), it also creates an averaging effect that eliminates overfitting, avoids the local minima of the cost function, and makes the training result more independent from more or less (un)lucky initialization.

## Initialization of weights

The selection of the initial values of all weights and bias can have significant impact for the final effect of training, due to possible local cost function minima. Therefore, random initialization of weights and bias is used as a minimum requirement. It is unclear, however, what distribution this random initialization should use. In the absence of a better idea, the default approach may be the normal distribution with the mean of 0, and the standard deviation equal to 1.

However, it turns out that this is not an optimal method, and even a simple analysis allows to find a better method of initializing weights and bias.



The problem arises when there are a lot of neurons in a certain layer (typically this can be in the input layer). At the initial configuration, with full connections, the activation of the neurons of the next (second) layer is a sum of a large number of elements, each with a random value with normal distribution. The weighted sums of inputs of such neurons will usually be high values and these neurons will for a significant initial learning time be saturated.

This is the same phenomenon of initial learning slowdown that was solved by replacing the cost function, but only for the output layer.

For example, if there are a thousand neurons in the input layer, then we can expect half of the inputs to have value 1 (and the other half 0), then the weighted sum at any neuron in the second layer will have a normal distribution with zero mean but standard deviation (from 500 activated inputs and the bias) equal to $\sqrt{501} \approx 22.4$. With high probability this will be a number with the absolute value large enough so that the activation of this (and each) second layer neuron will be very close to 0 or 1.

We can either leave the correction of this situation to the training process and ... wait, or we can adjust the initial connection weights to rectify it.

In practice, a good solution is the initialization of the weights of each connection to a neuron with $n_{in}$ input signals with normal distribution with mean 0 and standard deviation equal to $1/\sqrt{n_{in}}$. In most cases, this results in faster initial network learning. In some cases it also gives a better learning outcome (better generalization, i.e. lower errors on the test set).

Initializing the value of the bias does not affect the learning speed, and they can be initialized with a standard deviation of 1, or even initialized with a zero value.

## Hyper-parameters for training neural networks

The main parameters of a neural network are those that the network is able to automatically learn: the set of weights and biases.

However, to build a network and even start a training process, a number of other parameters have to be assigned: the number and type of neurons, the number of layers and the division of neurons between the hidden layers, and the various parameters related to the learning process such as: the learning rate $\eta$, the regularization parameter $\lambda$, the mini-batch size $m$, etc. They are sometimes referred to as the **hyper-parameters**.

An important difficulty in building and training neural networks is that all these parameters affect the ability of training of networks, performing their calculations, and achieving the desired accuracy and resistance to overfitting. There are no solid methods for determining all these parameters, and often it is necessary to experiment a little, to find the optimal (or at least a working) value for them. Unfortunately, experimenting with all parameters at the same time is often impossible, because a network with many parameters set incorrectly may not be making any progress at all.

To help with this situation, a number of heuristics exist for setting these parameters.

## Neural network project management strategy

The first goal in a neural network project should be to achieve any non-trivial, i.e. significantly better than random, learning effect. Having achieved this result, one can start to fine-tune the individual parameters. However, it can be amazingly difficult to achieve, especially for beginners, or for a new class of problems.

It is worth starting the project for a radically simplified problem (e.g. with a reduced number of classes), the training set "cleaned" of all artifacts that can cause network learning problems, network size and complexity near the minimum of what we expect to be appropriate, and small learning mini-batches to shorten the time of those preliminary experiments, and be able to respond more easily to phenomena appearing in the process.

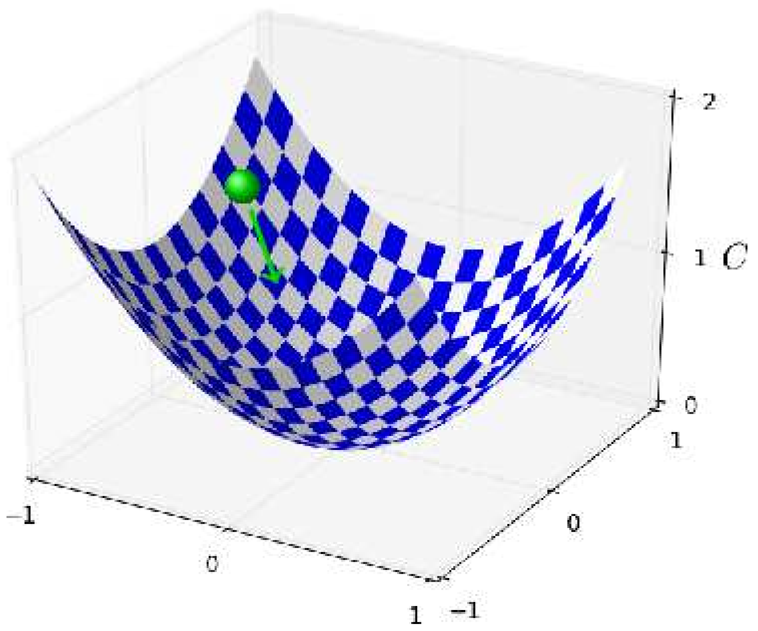
After getting the first indications that the network is learning the desired properties, one can start experimenting with the hyper-parameters of the learning process, including the cost function etc.

## Hyper-parameter values: learning rate $\eta$

Recall that the learning rate $\eta$ controls the magnitude of updates in the backpropagation algorithm.

Too small $\eta$ will cause very slow learning.

Too high a value may cause the gradient descend algorithm to jump around the minimum point, not being able to reach it.

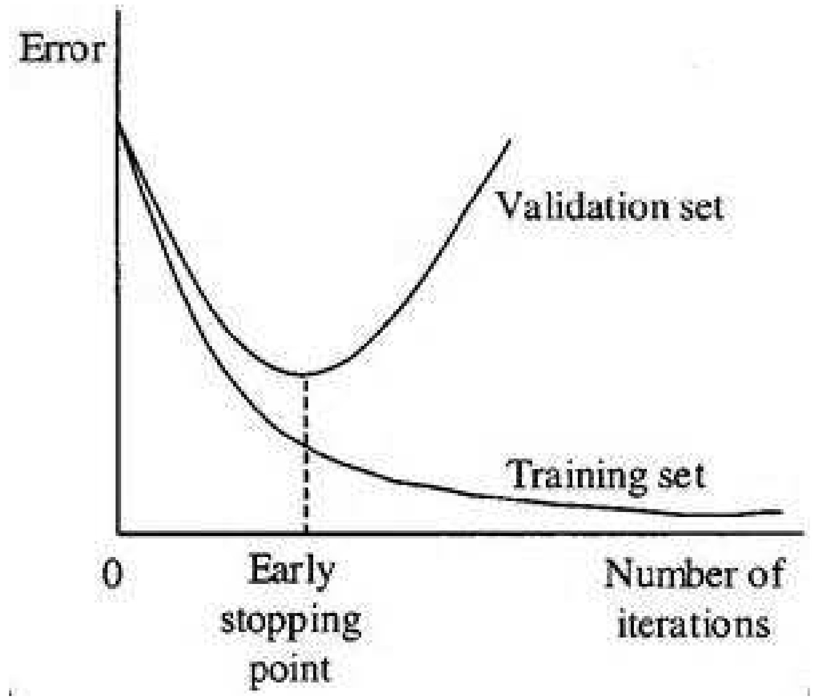


It is therefore useful first to get a border $\eta$ value, from which the cost criterion starts improving from the beginning. Starting with an arbitrary initial value, such as $\eta = 0.01$, if improvement occurs, increase $\eta$ according to the scheme: $\eta = 0.1, 1.0, ...$ until oscillations, or deterioration of the criterion, are encountered. If there was no improvement from the beginning, only deterioration or oscillation, then reduce $\eta$ according to the scheme: $\eta = 0.001, 0.0001, ...$, until improvement is achieved.

After determining the $\eta$ border value, one should specify the working value for optimization. It should be smaller than the border value, eg. several times smaller.

## Early stopping

**Early stopping** of the training process is a technique of monitoring the progress using a validation set. When the accuracy calculated on the validation set stops improving, we stop learning, even though the accuracy may continue to improve on the training set.



However, it is not always easy to use this method. The error curve is never a solid and monotonic line, and to notice that the validation error stopped decreasing, it has to be analyzed over a long period of time. It happens also that the validation error remains at a certain level fairly long, and then decreases again.

Note that the early stopping method automatically determines the **number of learning epochs**.

## Tuning the learning rate $\eta$

Given the method of automatically stopping learning, we can improve slightly the earlier procedure for determining the learning rate $\eta$. Instead of keeping it constant, we can reduce it as network optimization proceeds.

Initially, the procedure should be as described above. After stopping, the learning rate $\eta$ should be reduced several times, and retry learning. It will almost always be possible to further improve accuracy, however with a lower magnitude, corresponding to the learning speed.

This process can be repeated many times, in fact as long as we are able to reliably detect a real improvement of the learning criterion.

## Hyper-parameter values: regularization parameter $\lambda$

There is no well-established method for selecting and optimizing the ratio regularization parameter $\lambda$. It seems worth starting without regularization ($\lambda = 0.0$) and first determine the $\eta$ parameter.

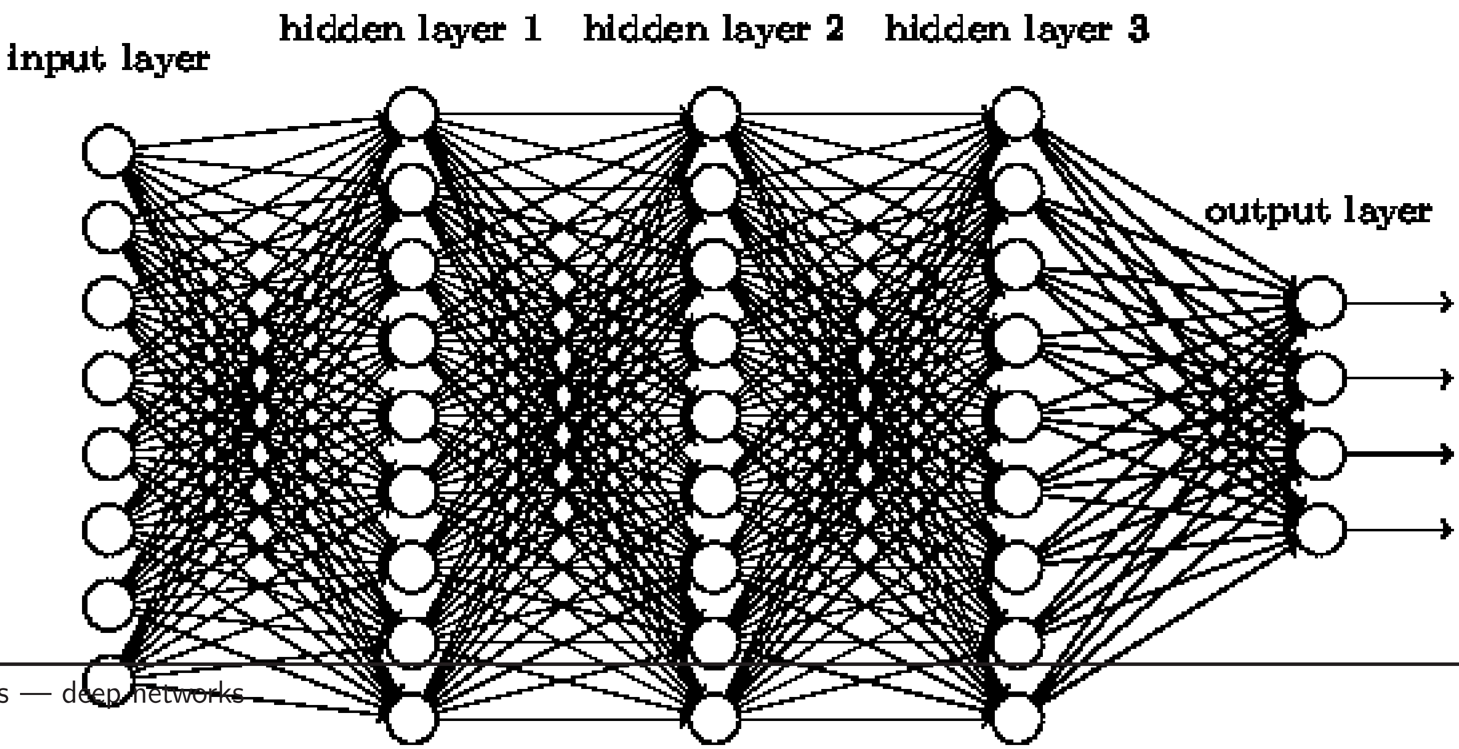
After specifying a good value for $\eta$ one can experiment with $\lambda$ starting from an arbitrarily selected value (e.g. $\lambda = 1.0$) and increasing or decreasing this value, expecting the accuracy on the validation set to improve. After reaching a series of good $\lambda$ values, an attempt to fine-tune may be made, by gradually decreasing it.

After determining a good value of $\lambda$, one can resume tuning the value of $\eta$.

## Training deep networks

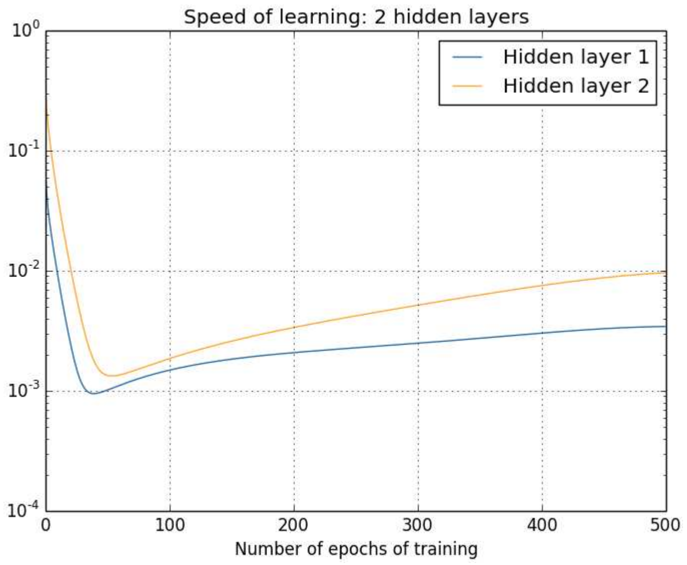
For a long time working with feedforward ANNs, mainly **shallow** networks were built, typically with one hidden layer. This was because of the fact that a network with one or two hidden layers can model any function. Further, teaching deeper networks is slow.

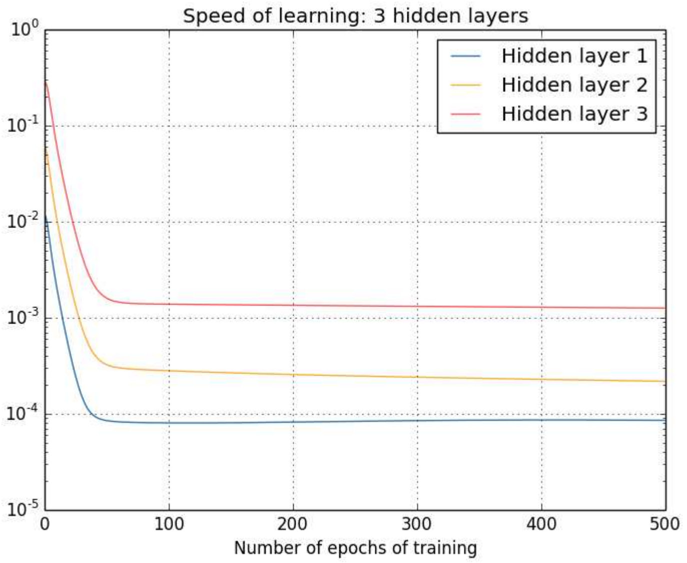
However, it could intuitively be expected that a **deep** network, with many hidden layers, could potentially have more capabilities than a simpler shallow network. Additional layers could, for example, enable the network to create additional levels of abstraction for the problem analysis, instead of generating the final results in one step. This is particularly important in the forthcoming big data era, when more and more complex issues are undertaken, and we expect neural networks to be able to understand and learn the regularities hidden in this data.

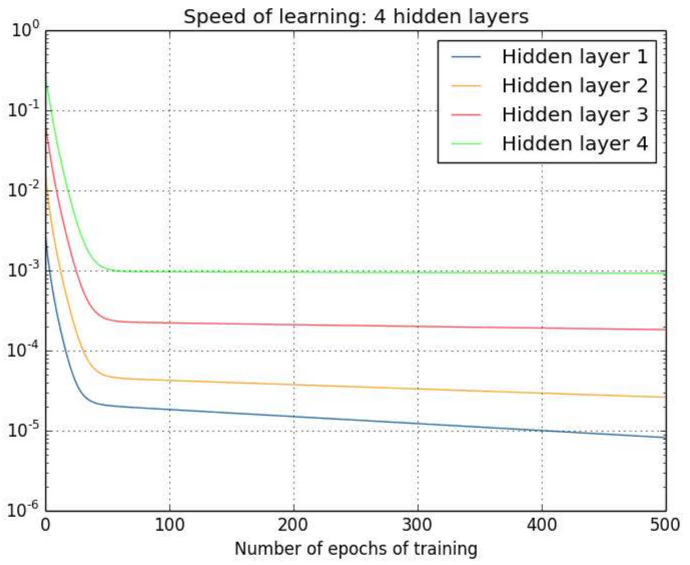


## The problem of vanishing gradient

An attempt to experimentally analyze the learning process of networks of a larger number of hidden layers can show, that in the normal backpropagation learning process the gradients of network parameters (weights and bias) decrease dramatically in the subsequently added layers.





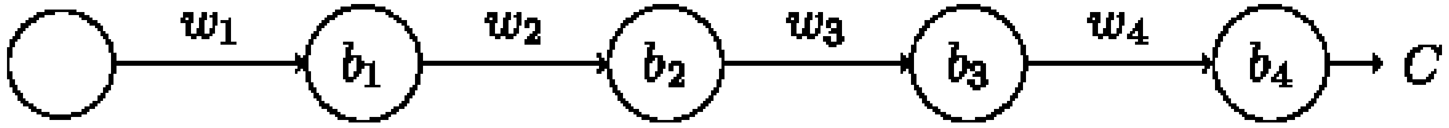


It turns out that this is a common phenomenon. In deep networks, when training by error backpropagation with the cross-entropy cost function, the gradients controlling the speed of weight updates, determining the speed of learning, disappear dramatically.

## The problem of vanishing gradient (continued)

The question is whether this **vanishing gradient effect** is a significant problem in network learning? Note that the neuron weights were initially initialized with random values. Thus, the signal fed to the network input is initially completely neutralized by these random weights. For the network to begin to learn effectively, the weights of the first layer must quickly grow to reasonable values.
However, they change extremely slowly.

An explanation of the vanishing gradient problem can be obtained by analyzing a trivially simple deep network, with single neurons in all layers.



The following formula can be derived for the partial derivative of the cost relative to the bias in the first layer:

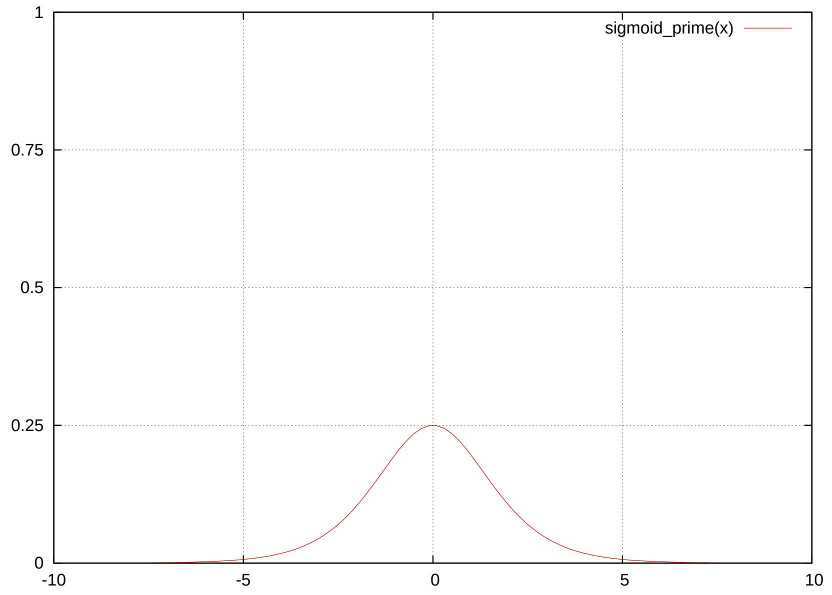
$$\frac{\partial C}{\partial b_1} = \sigma'(z_1)\, w_2\sigma'(z_2)\, w_3\sigma'(z_3)\, w_4\sigma'(z_4)\, \frac{\partial C}{\partial a_4}$$

## The problem of vanishing/exploding gradient

To study the behavior of the expression from the previous slide, recall the logistic function derivative formula:

$$\sigma'(x) \quad = \quad \frac{1}{1+e^{-x}}\left(1-\frac{1}{1+e^{-x}}\right)$$

```
$ gnuplot
sigmoid_prime(x)=1/(1+exp(-x))*(1-1/(1+exp(-x)))
plot sigmoid_prime(x)
```



As we can see, it has quite small values, especially for arguments significantly different from zero. Multiplying many such small factors. like in the formula in the previous slide, we get very small gradient values.

With high weights — either learned or enforced in the algorithm — it is also possible to get very large gradient values, which is called the **problem of exploding gradient**.

As a result, it can be concluded that the error gradient in training deep feedforward networks is <u>unstable</u>, and does not lead to effective learning.

So how can one achieve effective learning in deep networks?

## Convolution networks

The problem of the vanishing (unstable) gradient of the network parameters is not the only problem obstructing the construction and training of deep neural networks. Among others, the problem of overwhelming computational complexity for a long time hindered the creation of deep networks. These problems can be solved by building specialized networks, whose design leads to effective learning.

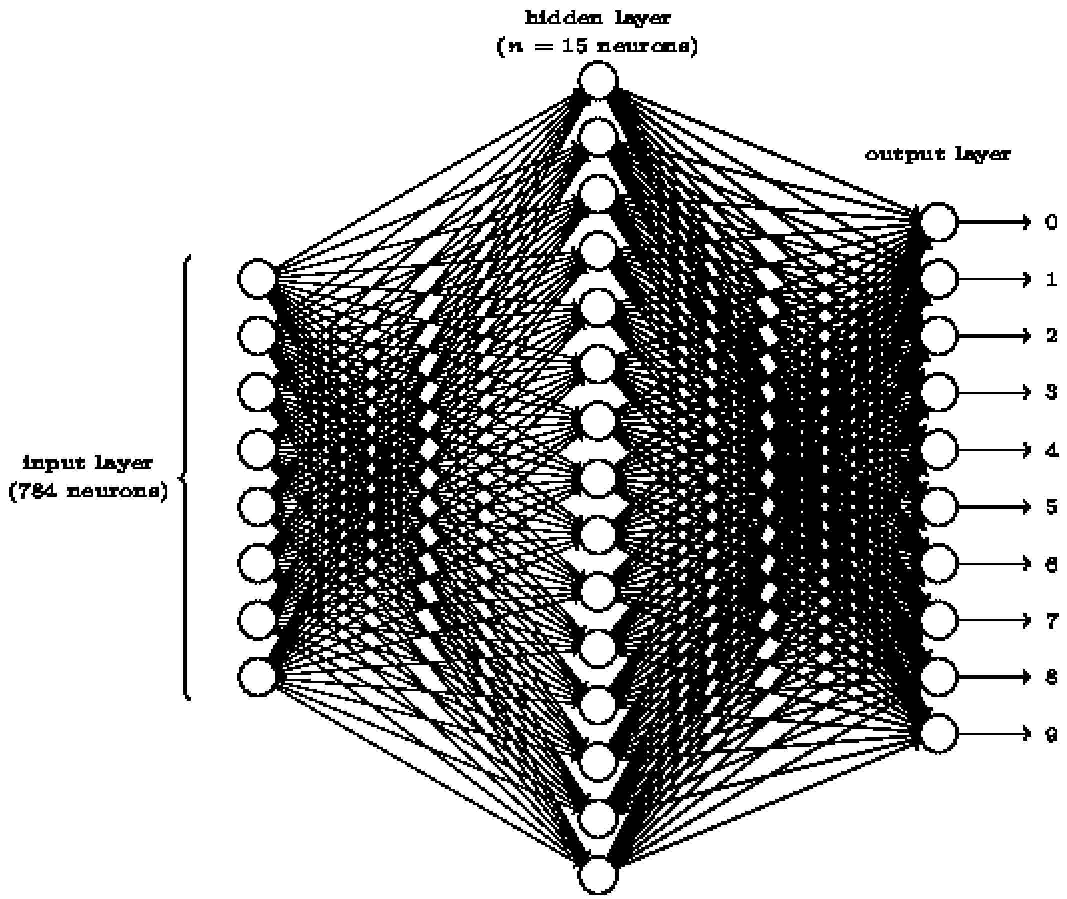
There are a number of deep network models that give good learning outcomes. Among them are the **convolution networks**, especially useful for image processing. This is due to the operation principle of the **convolution** operation, processing a large data set with a smaller filter, gradually moving along the set, and processing it section by section. But this type of processing is also used in others areas such as audio signal processing and other measurement data, as well as processing natural language utterances, and others.

The deep convolution network consists of a number of layers with designated functions, enforced by the specific structure and type of connections. Its construction takes advantage of three main ideas: local receptive fields, sharing weights, and image areas pooling.

## Convolution networks: why?

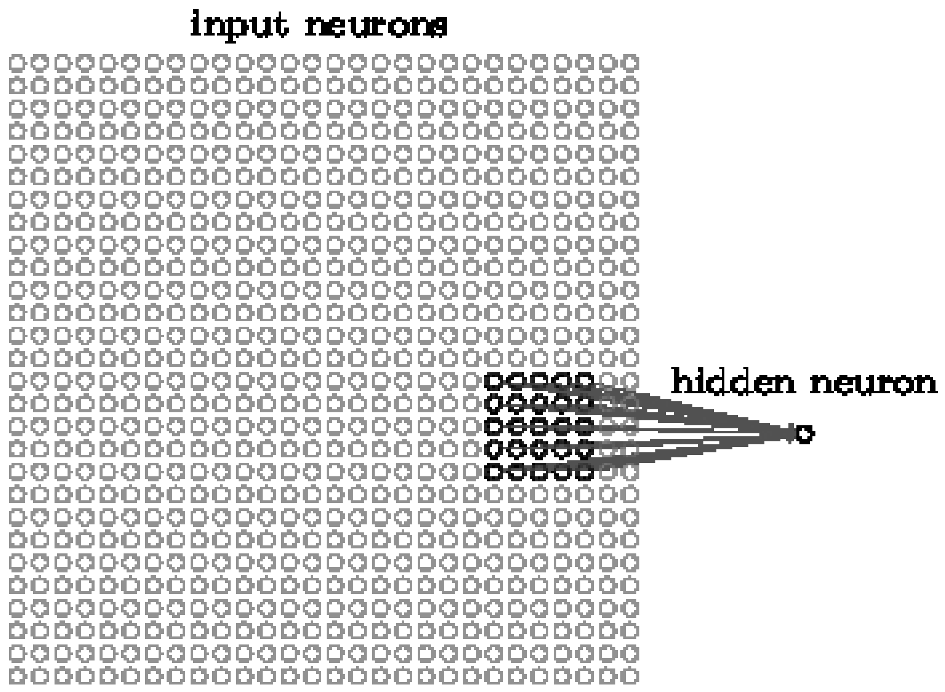
In the earlier example of analyzing images of handwritten digits (the MNIST database) of 28x28 pixels each, the network was used with the first (input) layer of 784 (=28x28) pixels, with a full set of connections to all neurons of the second (hidden) layer. This makes sense from the point view of processing complete instances of input data. But from the image processing point of view this is strange because the second layer is designed to learn how to process pixels which are both adjacent and far apart.

The input layer is seen as a one-dimensional row of neurons, not reflecting their neighborhood.



## The convolution layer

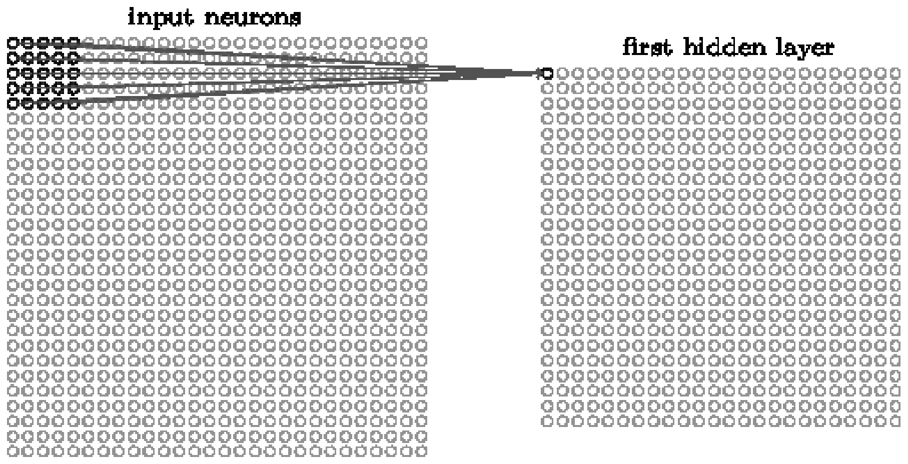
Building a convolution network we will now consider the neurons of the input layer to correspond to the pixels of the image to be processed. These pixels, in turn, are considered to represent **local receptive fields**, which are small areas of the image.

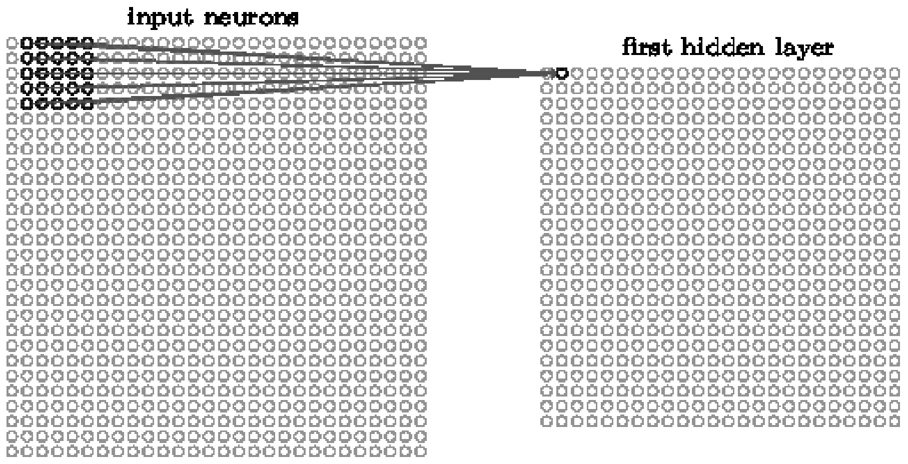


The convolution layer will be the second layer, processing each local receptive field with a convolution filter. This filter is a small array, processing the neurons of each local receptive field neurons, mapping to a single second layer neuron. For example, the convolution filter may be 5x5 neurons in size. Instead of the full connection from the input layer to the convolution layer, there will now only be connections from the local receptive field neurons to the single neuron of the convolution layer.

## The convolution processing

Processing an image with a convolution filter works by applying the filter to the first section of the image, and computing the filter output value producing the output value. Then the filter slides by some distance, and processing repeats.





The slide distance — called the **stride** — showed in the pictures above, is 1 neuron.

## The arithmetic of the convolution

Consider the specific example: 28x28 pixel image processing network for processing the handwritten digits of the MNIST database.

Suppose we apply a 5x5 convolution matrix. Assuming that the matrix will be shifted by a one pixel stride in each direction (horizontal and vertical), there are 24 positions of the convolution matrix in both dimensions, and therefore the method of sliding the convolution matrix will produce the output data for 24x24 neurons, and such will be the size of the convolution (first hidden) layer.

Thus, only 25 weights and one bias value support the connection between the first and second network layers. This is the fundamental difference between this architecture and the full set of connections between these layers, which would require almost half a million weights (28x28x24x24) and 576 biases. Network training can thus be effective.

Note also that the second layer could be the exact same size as the processed image. This would require applying the center of the convolution matrix to the edges of the image, partially outside its boundary, and feeding it with values of non-existent pixels. We could also make the convolution layer much smaller by applying the filter with a stride of two (or more) neurons, producing the convolution layer of size 12x12.

## Shared weights of the convolution layer

The convolution filtering is very well known and popular in image processing. Various types of convolution filters are employed: blurring, sharpening, edge detection, and others.

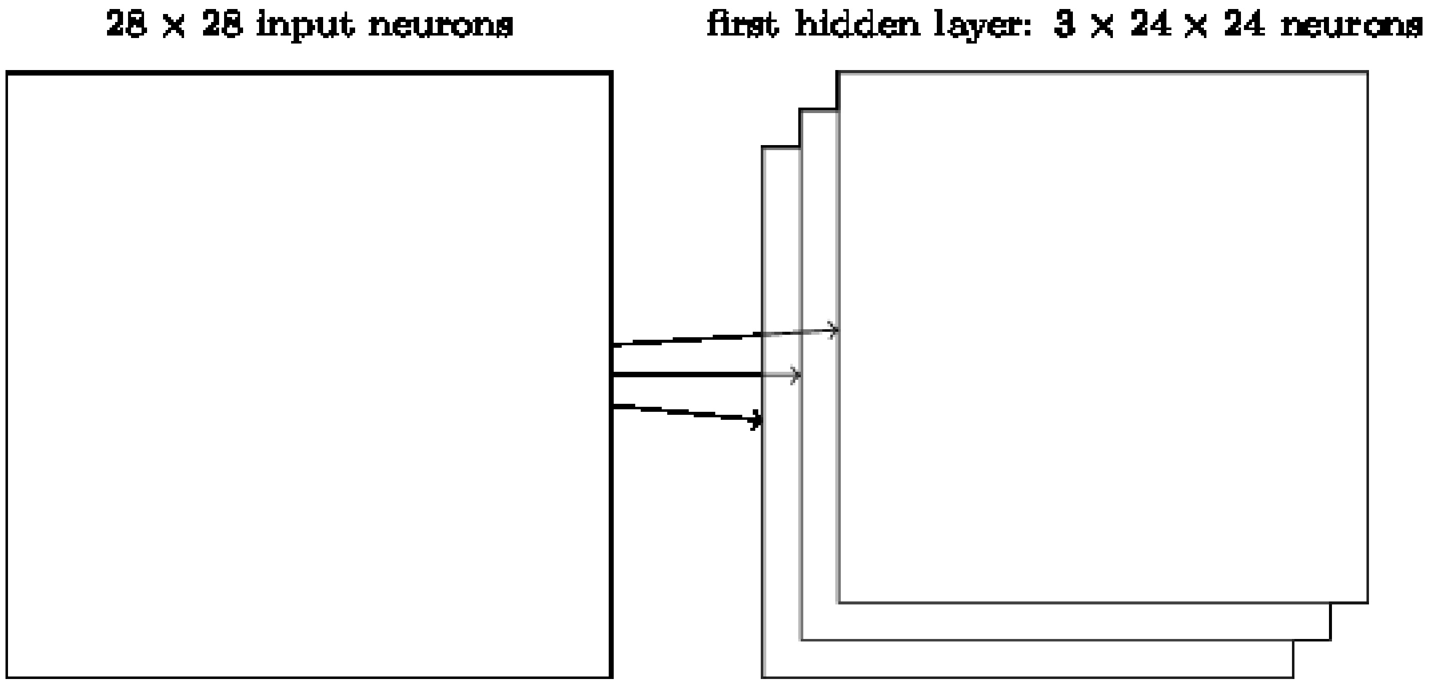
So which convolution filter will be used in our network? The key idea used in deep convolutional networks is the **sharing of weights**. All individual convolutional filters processing the local receptive fields will learn the same set of 25 weights (plus bias). This speeds up the training, and forces the weights to adjust to the type of processing that is identical for all the image receptive fields.

In effect, the convolution layer we have just created will learn to recognize some feature in the image. This feature can have a simple interpretation, like a central blob, or an edge running at a specific angle, but it can also be some hard to describe pattern (although must be simple, due to the size of the convolution filter). The rectangular structure of neurons produced by the convolution filtration is called the **feature map**.

It is important to see that the feature our network learns to recognize is going to be random — due to the random initialization of the filter's weights and bias. At the same time, this learned feature will be detected regardless of its location within the image.

## Multiple convolution filters

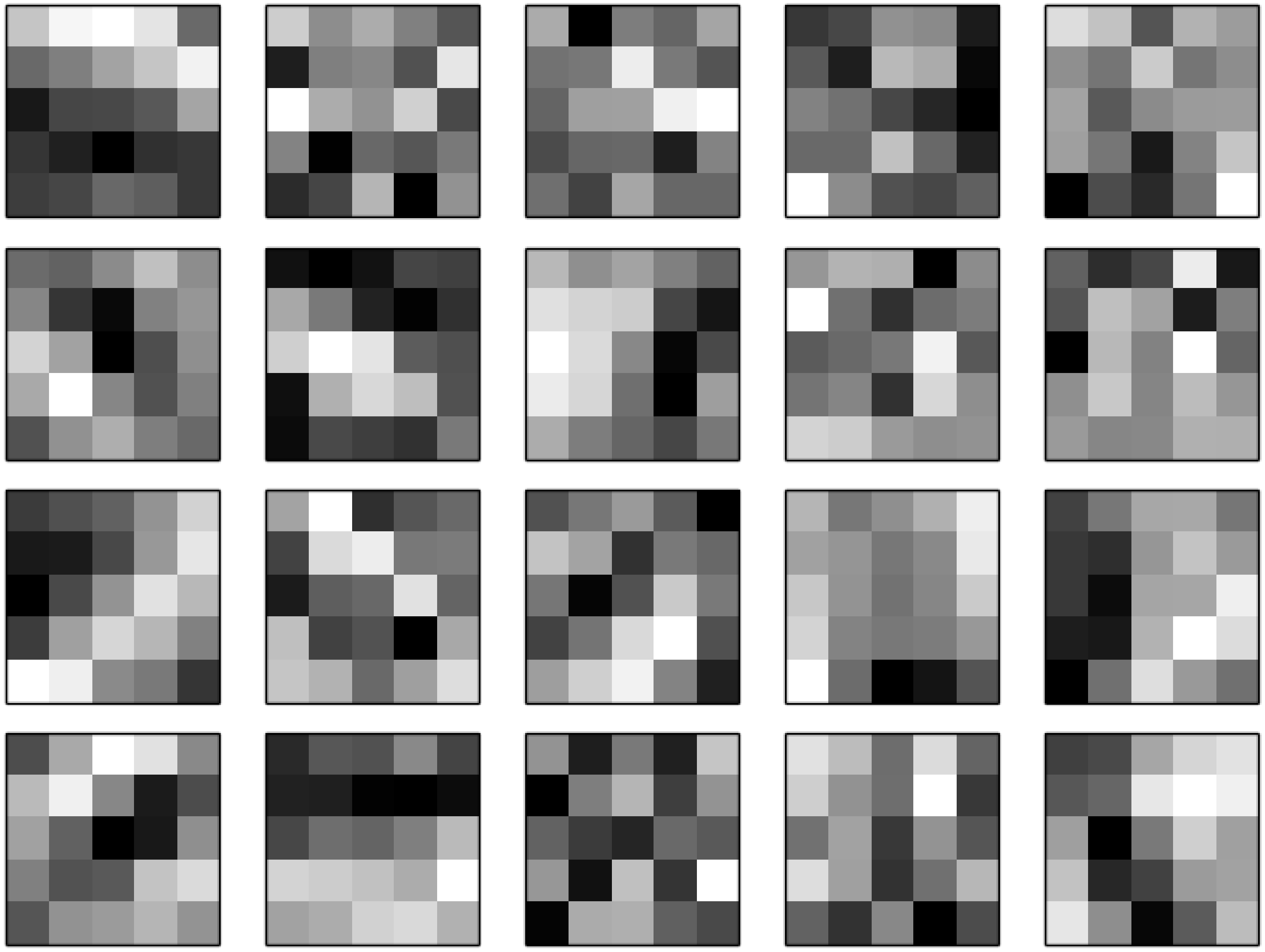
We have seen, that the convolution layer will learn to recognize some random image feature. This feature may be more of less useful in interpreting the whole image. But we can maximize the chance of obtaining useful feature-detecting filters, by creating several/many of them. Due to randomization, we can expect them each to be different.



In the above image we have three feature maps. They are trained in parallel, thus together forming the second (convolution) layer of our network (which is also the first hidden layer).

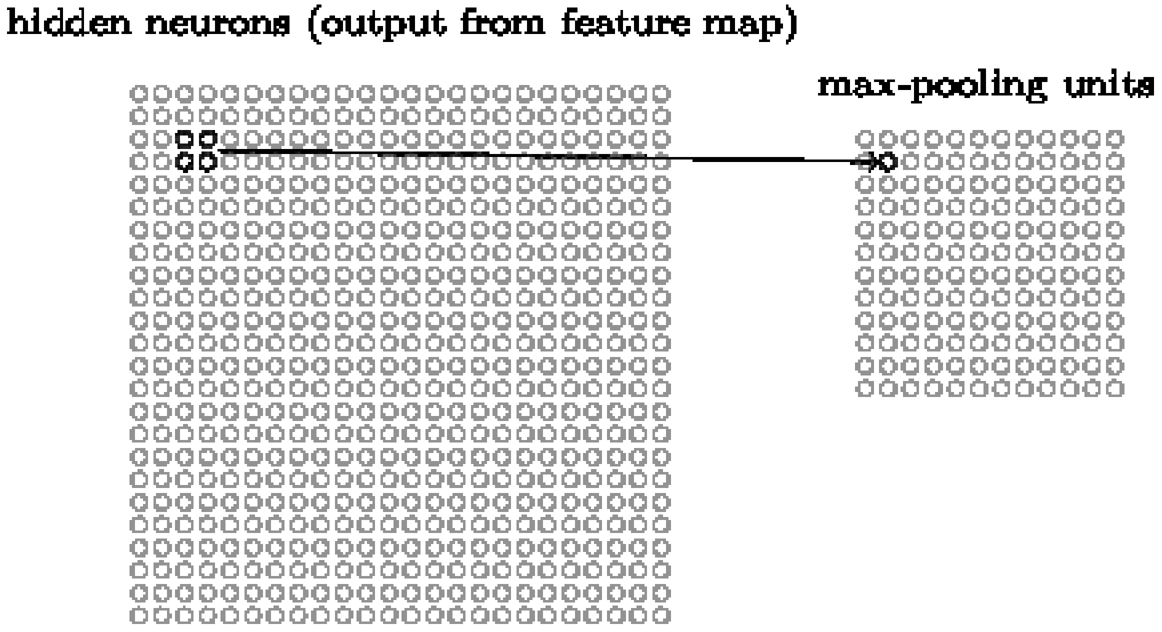
## Multiple convolution filters (ctd.)

Three feature maps depicted above is just an example. Many more can be created. Here are example filters trained on the MNIST database:



## Pooling layers

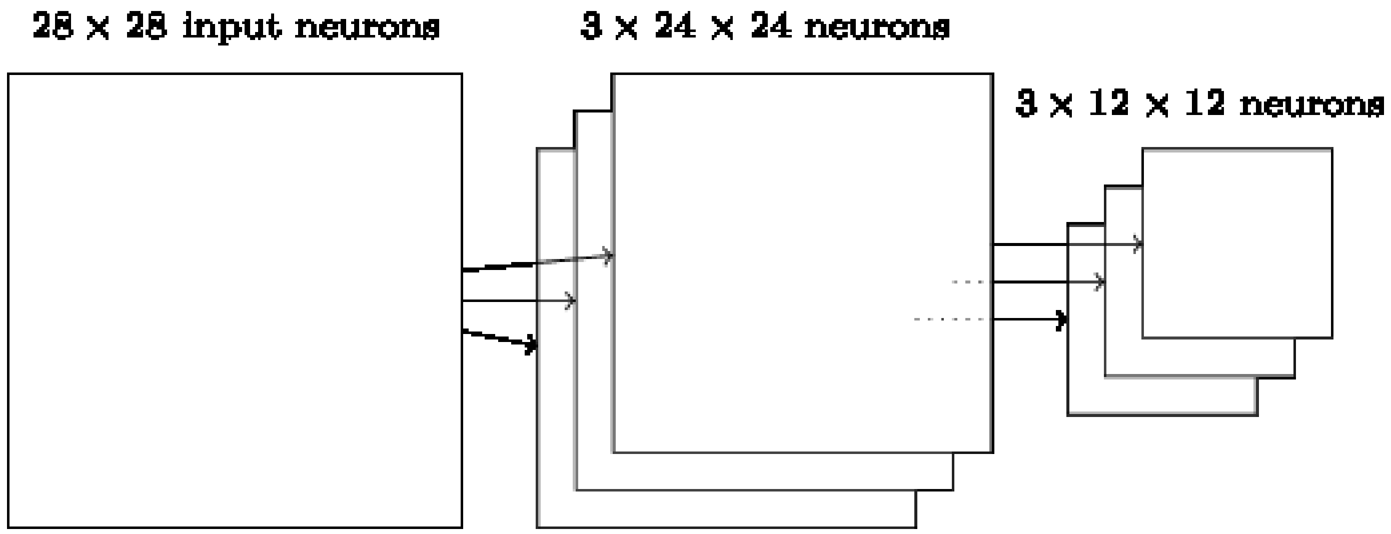
The feature maps produced in the convolution layer recognize small features in the original image. They are, however, almost as large each, as the original image (this depends on the stride size, plus the margins resulting from the size of the convolution filter). We can say, that the feature maps are created at full resolution of the original image. But we are, in fact, interested in finding macroscopic features in the image. We need a way to aggregate the image in a way which preserves the features learned.



The next network layer can be build to serve this purpose. One commonly used aggregation, or **pooling**, function is the max, computing the maximum pixel value over some area of the large feature map.

## Pooling layers (ctd.)

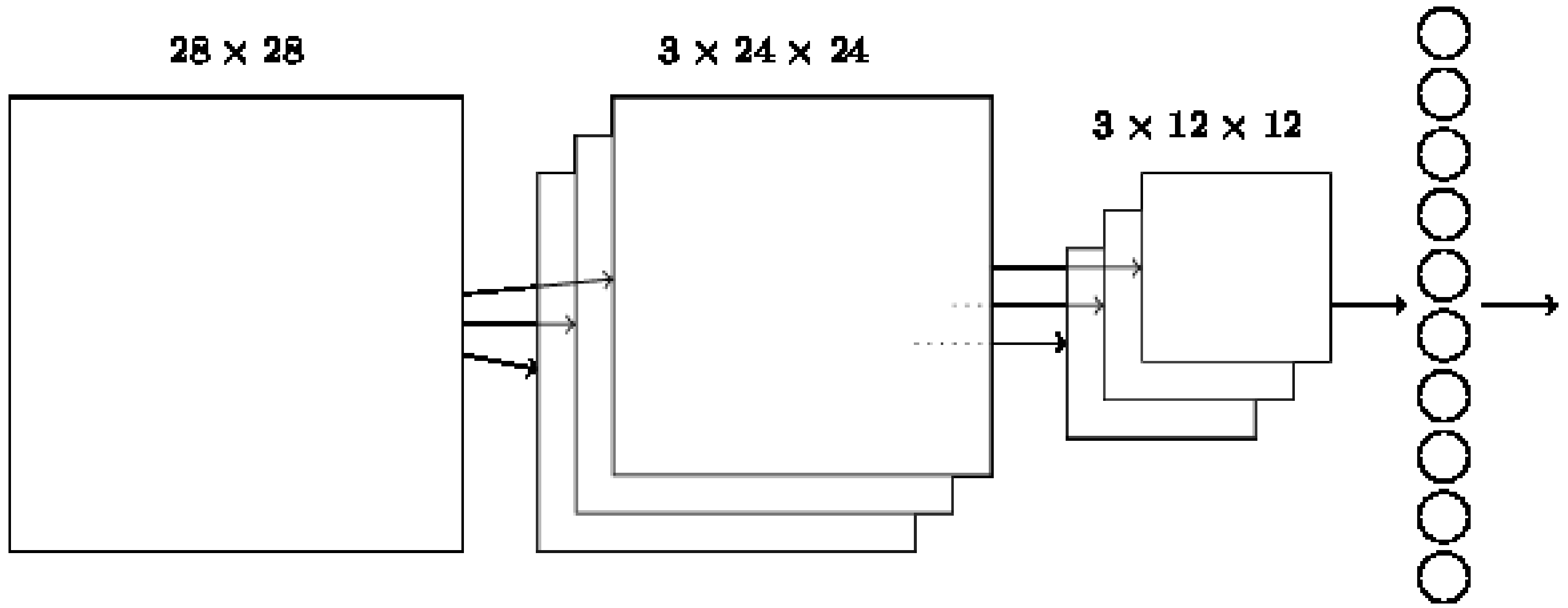
In the presented example the pooling of 2x2 neuron areas was applied. This is equivalent to reducing the image pixel size in half, from 24x24 to 12x12. Each feature map from the convolution layer should be pooled in the same way, resulting in the following structure:



Instead of max-pooling, another aggregation function could have been used. An example of such is the **L2 pooling**, which computes the square root of the sum of squared activations of neurons in the pooled area. The optimal version can be determined by experimenting with them all.

## Completing the network

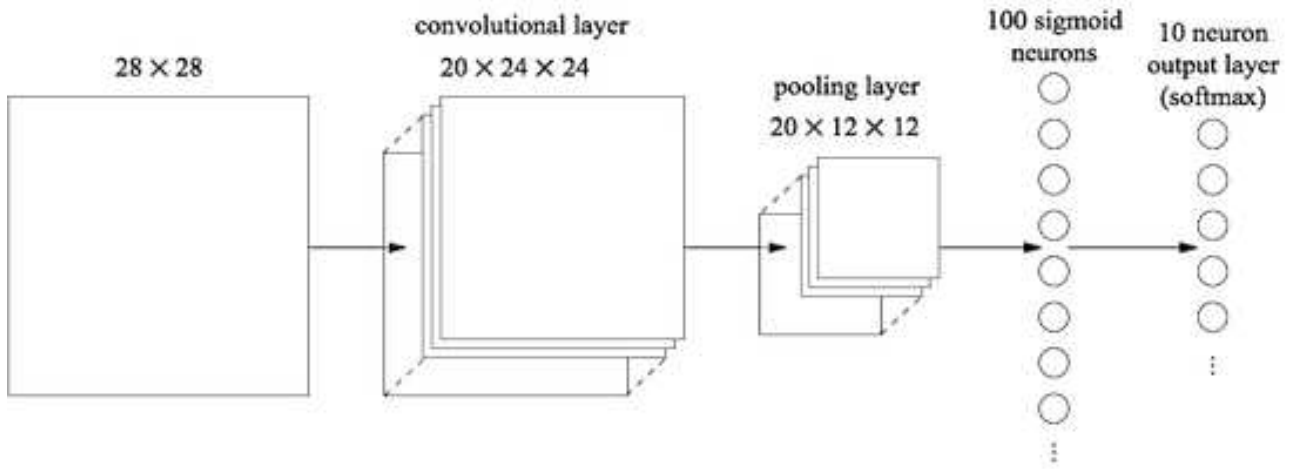
Assuming we decided that the network layers created thus far should be sufficient to recognized the hand-written digits, we can finish up by adding the output layer consisting of ten neurons, each trained to respond to a different digit, as before:



The neurons from the output layer are expected to recognize the digit contained in the input image by examining the features identified by the network. For this reason, each output neuron must have connections from all the neurons in the preceding network layer (pooling). In other words, there must be a full connection initialized between the final two layers.

## Further details

When building a complete network for recognizing handwritten digits from the MNIST database, it makes sense to use softmax neurons in the last layer with the simultaneous use of the log-likelihood cost function. Using twenty parallel filters in the convolution layer, the network looks like this:



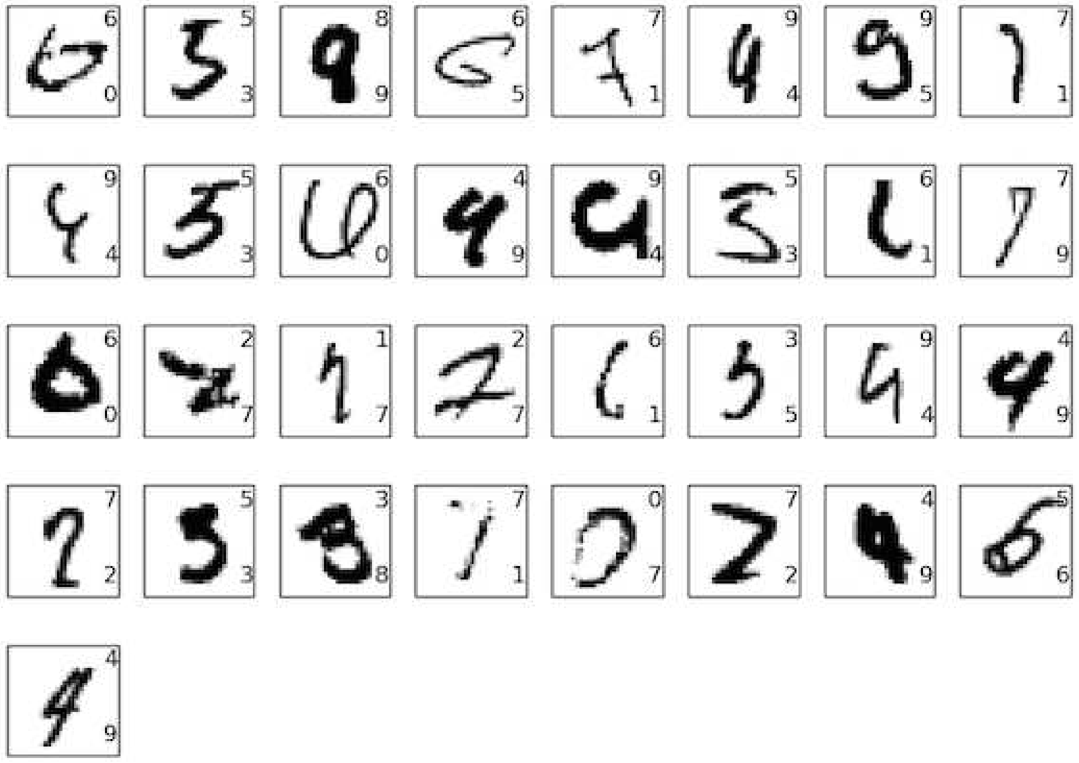
An additional, fully connected layer of neurons with a sigmoid activation function aims to integrate the information generated by individual convolution filters, allowing the final softmax neuron layer to make the final decision.

## Other variants

The classification of handwritten digits in the MNIST database has been studied extensively, and many classifiers and scientific papers have been produced to demonstrate effective techniques to improve the results obtained. Among them are:

- Application of the second set of convolution and grouping layers, analyzing the results from the first such layer. This has an effect introducing an additional level of abstraction to detect features image even more macroscopic. However, the second convolutional layer has input all outputs combined as input of the first grouping layer.
- Use a larger dataset. Although there is often no additional data, as in the case of the MNIST database, it is possible to "multiply" the data by transforming existing images, such as: rotations and shifts. In addition to greater learning efficiency, such an extension of the data set also reduces the tendency to overfitting.
- Using the ensemble learning method, ie training several independent networks, and adding a voting layer to select the result by majority method. The application of this method is based on the assumption that different networks can make different errors and such errors can be eliminated.
- Use of other activation functions, such as tanh, or especially relu.

33 cases of misclassification on 10,000 MNIST images (the upper index shows the actually written digit, and the lower index shows the digit recognized by the deep net):



## Training convolutional networks

Convolutional network training can be carried out similarly to training a multilayer perceptron with a full set of connections between successive layers. That is, it is possible to use error backpropagation, and optimization using the stochastic greatest gradient method.

The backpropagation procedure has to be adapted to the specifics of the convolutional network, but this is not a problem. Moreover, due to the independence of the convolutional network elements, the learning can proceed in a parallel manner, and in addition to the use of multi-processor units and/or multi-cores, it is possible to use multiple processors on the GPU graphics card.

## Transfer learning

**Transfer learning** is a group of techniques aiming at use the knowledge gained during machine learning of one task, or ready-made models created in its course, for another machine learning task.

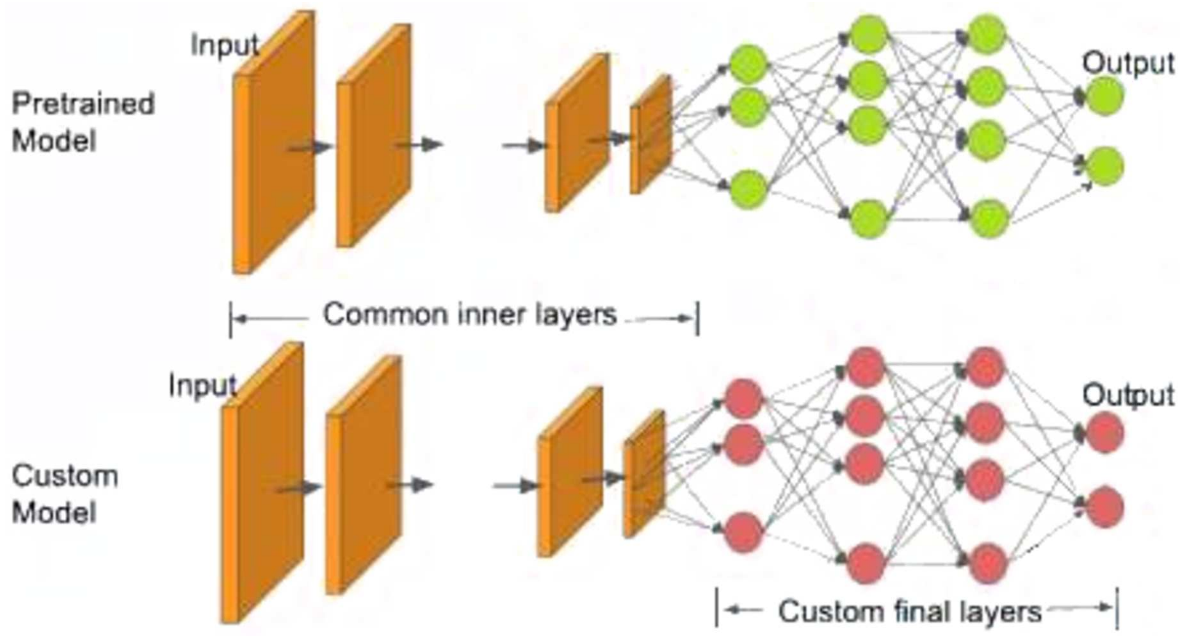
This approach, which has related mechanisms in psychology, has been considered in machine learning research since the 1970s. Currently, it is more and more often used in relation to deep networks as an alternative to building any application, i.e. training a neural network from scratch.

Transfer learning is generally used:

- to save time and resources from having to train multiple machine learning models from scratch to complete similar tasks,
- as an efficiency saving in areas of machine learning that require high amounts of resources such as image categorization or natural language processing,
- to negate a lack of labeled training data held by an organization, by using pre-trained models.

## Transfer learning in convolution networks

An example of the use of transfer learning is the use of neural networks for image recognition. If the deep network has been trained on a large training set, usually the initial layers of the network recognize basic graphic features such as points and edges, further layers recognize more complex elements, specific shapes that are elements of different images, and more complex structures useful for recognizing specific objects.



Let's note, that for the custom training of the adopted deep model to be possible, the size and format of the new samples must be compatible with the architecture of the initial layer of the network, which does not undergo training. Failing that, a preprocessing step would be required to adjust such size and format.

## Transfer learning in convolution networks (ctd.)

Although the last layer is fine-tuned to recognize the objects for which the network was built, only this layer can be trained during transfer learning, leaving the initial layers unchanged. To make it possible, it is necessary to **freeze** the trained parameters of all the layers which are to be preserved. This allows the network to use the knowledge accumulated from the large training set used to build the original network, and complete the network using a smaller set of new images.

Examples of convolutional deep models trained to recognize certain classes of images, and available over the Internet:

- VGG-16
- VGG-19
- Inception V3
- XCeption
- ResNet-50

There also exist trained deep models used in natural language processing.

## Resources

This presentation contains some materials from the following:

1. Michael A. Nielsen: Neural Networks and Deep Learning, Determination Press, 2015
`http://neuralnetworksanddeeplearning.com/`

2. Ian Goodfellow, Yoshua Bengio, Aaron Courville: Deep Learning, MIT Press, 2016
`http://www.deeplearningbook.org`