# Deep Learning and Knowledge Generalization


**Guido Tascini**

**Abstract** This work concerns the new studies related to the deep learning of machines. In particular, it tries to see what lies behind the behavior of deep neural networks, which have been very successful in various fields, such as image recognition, interpretation of natural language and much more. The work analyzes the deep networks and the behavior of the backpropagation algorithm. From this it seems that the success of these networks, which amazed the authors of the algorithms themselves, seems to lie in the processing of information and in the ability of the algorithms to extract relevant knowledge, discarding that which is not relevant for the purposes of the learning target. The bottleneck principle (Tishby & Zaslavsky, 2015 IEEE Information Theory Workshop (ITW), Jerusalem, pp. 1–5, 2015), in particular, appears to be a promising vision for the design of deep artificial neural networks, based on a general principle related to the processing of knowledge.

**Keywords** Autonomous knowledge learning · Backpropagation · Deep artificial neural network · Deep learning · Input data compression · Output prediction · Knowledge generalization · Relevant knowledge extraction · Shallow artificial neural network

## 1 Introduction

A Shallow Neural Network, has a single hidden layer, between an input layer and an output layer. The algorithm that associates the set of input patterns to the set of output patterns, named back-propagation algorithm, derives its name from the backward propagation of the errors on the output units: difference between real outputs and expected outputs. If the network has more than one hidden layers, we are talking about deep networks.

G. Tascini (✉)
ISAR-Lab, ITPI, Fermo, Italy



G. Minati (ed.), *Multiplicity and Interdisciplinarity*, Contemporary Systems Thinking, https://doi.org/10.1007/978-3-030-71877-0_12

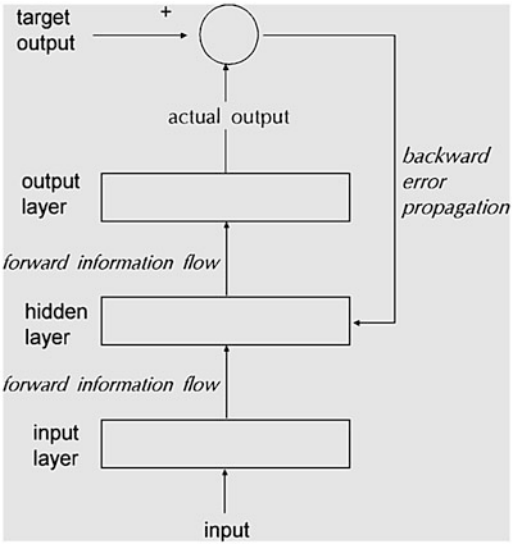


**Fig. 1** Backpropagation algorithm scheme

The scheme of the Backpropagation algorithm, for the shallow network, is represented in Fig. 1. As it can be seen, the normal information flows forward, while the errors flow backward and the errors derive from the comparison between the output that the network must have (target) and the calculated output from the network (actual). The algorithm tries to minimize these errors by varying the weights of the links and stops when the target output substantially coincides with the actual output (minimum of errors).

**Weights Initialization** Normally weights and thresholds are set, at the beginning, equal to small random numbers. The activation level of the 'entry unit' is set by the instance; as the activation level $O_j$ of the hidden unit or the output unit, it is determined by the expression:

$$O_j = F\left(\sum w_{ji} O_i - \theta_j\right)$$

where

$$F(a) = \frac{1}{1 + e^{-a}}$$

**Weight Training** We start from the output layer and work backward on the hidden layers by recursively updating the weights with the relation:

$$w_{ji}\,(t+1) = w_{ji}(t) + \Delta w_{ji}$$

where the variation of weights is given by the expression delta:

$$\Delta w_{ji} = \eta \delta_j O_i$$

with η speed parameter. A term, called 'moment', is added to this variation to speed up convergence:

$$w_{ji}(t+1) = w_{ji}(t) + \eta \delta_j O_i + \alpha \cdot \left\lfloor w_{ji}(t) - w_{ji}(t-1) \right\rfloor$$

where $0 < \alpha < 1$.

$\boldsymbol{\delta_j}$, gradient of the error, is calculated with the expression:

$$\delta_j = O_j \left(1 - O_j\right) \sum_k \delta_k w_{kj}$$

where $\delta_k$ is the gradient of the error corresponding to the unit k to which a connection from unit j points.

The iterations are repeated until convergence. Then *backpropagation algorithm* for shallow networks, with only one hidden layer, is the following:

1. Initialize the weights randomly
2. Do {
3. Initialize the global error $E = 0$;
4. For each $(X_k, t_k) \in TS\}$ {
5. Calculate $y_k$ and the $E_k$ error;
6. Calculate the $\delta_j$ on the output layer:
7. Calculate the $\delta_i$ on the hidden layer:
8. Update the network weights: $\Delta w = \eta \delta x$:
9. Update the global error: $E = E + E_k$;}
10. } while $(E < \varepsilon)$;

Training is carried out in the following three phases.

**Learning** In this phase the training set patterns set $(X_k, y_{dk})$, $k = 1, \ldots M\}$ and weights modified according to the Error-Backpropagation rule. It is important to choose the patterns of the training set well so that they are as representative as possible of the information that the network has to learn.

Such patterns can be presented:

*In a Bach* (or cumulative) mode. All the patterns are presented first, the error committed on each one is calculated, the error is added up and then the connection coefficients are modified;

*In a on-line* mode in which the connection coefficient values are updated after the presentation of each single pattern of the training set. Convergence occurs when it is reached a reduction of the global error, $E = \Sigma_k E_k$, so that the weights adapt to the input pattern: in practice so that it becomes $E < \varepsilon$.

**Generalization** A well-trained network must be able to generalize information. In the learning phase, after minimizing the errors committed at the exit, the weights are

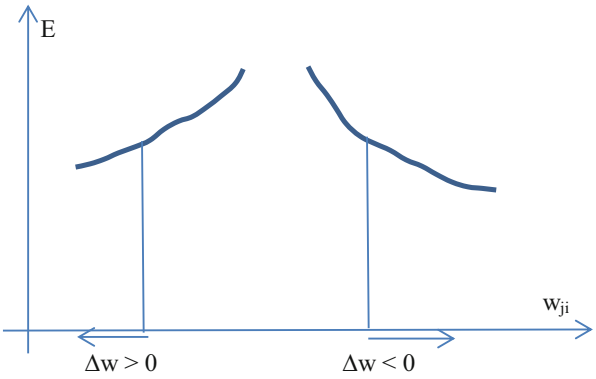


**Fig. 2** Gradient Descent method

frozen to proceed to the generalization phase in which the network responds well to examples never seen before.

**Convergence** The 'gradient descent' method is universally adopted for the convergence phase. Conceptually, the method consists in reducing the global error going down towards the minimum, along the curve E = f (w), with the calculation of the gradient.:

- if the gradient, $\partial E/\partial w_{ji}$ is positive, you must go towards the decrease of the weights ($\Delta w < 0$),
- if the gradient, $\partial E/\partial w_{ji}$ is negative, you need to go towards weight gain ($\Delta w > 0$). See Fig. 2.

The error is calculated every time a training pattern is presented to the network and then a descent towards the minimum is performed along the curve E = f (w) following the decrease in the gradient. And there will be a gradient for each weight.

**Extended Delta Rule** y1 (actual output) is compared with y1 *(expected output) and the coefficients are increased by $\Delta w_{ij}$:

$\Delta w_{ij} = \eta \partial E/\partial w_{ij}$, with η = learning parameter. If as a measure of the error we have.

$$E = \frac{1}{2}\sum_i \left(y_i - y_i^*\right)^2$$

It leads to the explicit formula:

$$\Delta w_{ij} = -\eta \left(y_i - y_i^*\right) \frac{\partial F\left(P_i\right)}{\partial P_i} x_j$$

While if we are dealing with a finite set of training patterns it is more convenient to use the GLOBAL error:

$$E = \frac{1}{2} \sum_{k} \sum_{i} \left( y_{ik} - y_{ik}^{*} \right)^{2}$$

That give the *Extended delta rule.*

$$\Delta w_{ij} = -\eta \sum_{k} \left[ \left( y_{ik} - y_{ik}^{*} \right) \frac{\partial F\left( P_{ik} \right)}{\partial P_{ik}} x_{jk} \right]$$

In this last case, it is equivalent to the search for a local minimum of the value of E moving in the direction of the maximum decrease (gradient method).

## 2 Deep Neural Networks

Deep Learning (DL) is a branch of Machine Learning. It allows you to extract very complex information from a set of data, making it possible to carry out very complicated tasks, such as those related to the perceptual sphere. Deep learning models have the characteristic of being made up of different processing layers, each of which extracts a representation of the previous layer.

In the context of supervised deep learning, the most used class of models is the multi-layer neural network, or deep neural network (DNN). So it is a type of network built model, the main components of which are nodes, or neurons. As known, there are different classes of neural networks, depending on the type of nodes, and how they are connected to each other. The neural networks, on the basis of which the types of networks used in deep learning have been developed, are *feed-forward neural networks* (*FFNN*), whose operation is normally based on the "Back-Propagation" algorithm.

We can define the *FFNN* as follows: a network in which, if we number the vertices, all the connections go from one vertex to another of greater number. In practice the vertices are grouped into layers, and the connections go only from one layer to the higher layers.

The layers of the nodes form a hierarchical structure: the lowest layer is the *input* layer; the highest is the *output* layer. All the layers located inside are called *hidden* layers; see Fig. 3.

The Deep Neural Networks, with multiple layers of neurons, of the type feed-foreward, with many more then two hidden layers, and accelerated by the use of GPUs, have recently seen enormous successes in many fields. They have passed the previous state of the art in speech recognition, object recognition, images, linguistic modeling and translation.

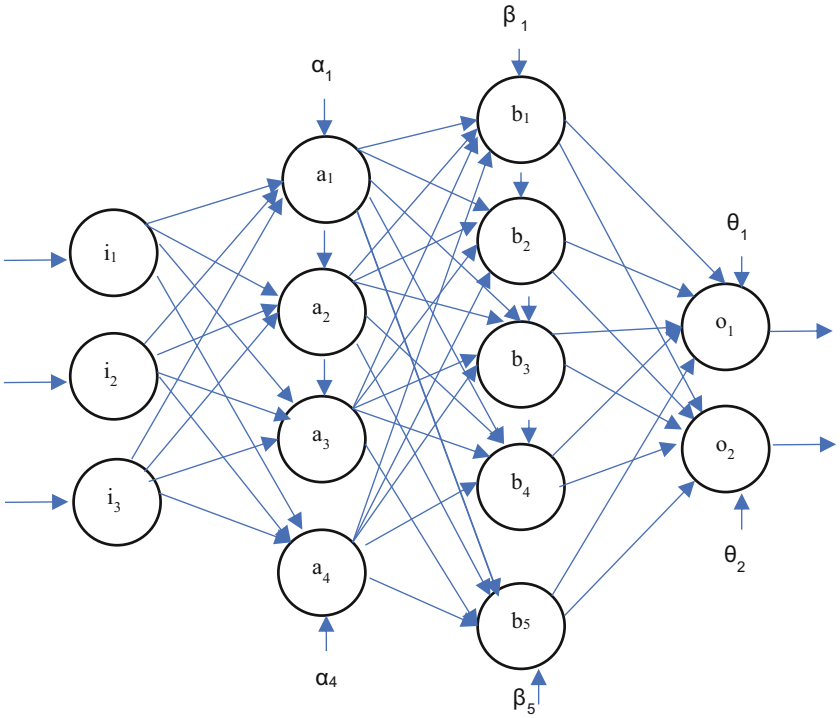


**Fig. 3** Deep Neural Network, of the type *feed-forward,* described by the sequence 3–4–5-2, with four layers: input layer, two hidden layers, output layer

The Fig. 3 illustrates a *deep neural network* with only two hidden layers. The show nn has three inputs (**$i_1$, $i_2$, $i_3$**), a first hidden layer ("**A**") with four neurons, a second hidden layer ("**B**") with five neurons and two outputs ($O_1$, $O_2$), that may be described by the sequence **3–4–5-2.** This network requires a total of (**3** * 4) weights +4 *bias* + (**4** * 5) weights +5 bias + (**5** * 2) weights +2 bias = **42** weights and 11 bias.

The example use as activation function the *hyperbolic tangent* for the outputs of the two hidden layers and the *softmax* for the output of the network. Then the formulas that calculate the feed-forward are as follows:

$A_i = \tanh (i_1 p_{1i} + i_2 p_{2i} + i_3 p_{3i} + \alpha_i)$—first hidden layer,
$B_i = \tanh (A_1 p_{1i} + A_2 p_{2i} + A_3 p_{3i} + \alpha_i)$—second hidden layer,
$O_i = \text{softmax} (B_1 p_{1i} + B_2 p_{2i} + B_3 p_{3i} + B_3 p_{3i} + \beta_i)$—outputs.

The training standard of deep NN uses back-propagation algorithm. The deep neural network training, with multiple hidden layers, is more difficult than the *shallow neural network* training with a single layer of hidden nodes. This factor

is the main obstacle to overcome in order to process networks with many hidden layers.

The connections, represented by arcs, are unidirectional and connect only nodes of one layer with those of the next layer. Each arc is associated with a parameter, called weight. In the initial modeling the arcs represented the synapses, that is, nerve impulses that are transmitted from one neuron to another and the purpose of these models was to identify which neurons were crossed by a sufficiently intense signal, omitting neurons, whose signal was below a certain threshold. We present the relationship between the layers of the network as a univariate relationship. For this we define:

- L: number of layers of the network, consisting of an input layer, an output layer and L—2 hidden layers;
- $p_1$: number of input nodes;
- $p_l$: number of nodes present in the l-th layer;
- $x_i$: value of the i-th input node;
- •$a_j^{(l)}$: value of the j-th node of the l-th layer;
- w $(l)_{ij}$: coefficient associated with the arc that connects the i-th node of the l-th layer with the j-th node of the (l + 1) -th layer;
- $y_k$: value of the k-th output node.

The relationship between the input layer and the first hidden layer is:

$$z_j^{(2)} = w_{0j}^{(1)} + \sum_{i=1}^{p_1} w_{ij}^{(1)} x_i,$$

$$a_j^{(2)} = g^{(2)}\left(z_j^{(2)}\right).$$

Note how the j-th node of the first hidden layer takes on a value equal to $g^{(2)}(z^{(2)})$, where $g^{(2)}(\cdot)$ is a non-linear function, called activation function, while $z^{(2)}$ is the linear combination of the input nodes and the parameters $w^{(1)}$. To this linear combination is added the term:

$$w_{0j}^{(l)}$$

that is the parameter associated with the arc that connects a constant node equal to 1 with the j-th node of the (l + 1)-th layer.

This quantity acts as an intercept in the linear combination, and is introduced to model any distortion.

The relationship between the (l–1)-th layer and the l-th layer is defined as:

$$z_j^{(l)} = w_{0j}^{(l-1)} + \sum_{i=1}^{p_{l-1}} w_{ij}^{(l-1)} a_i^{(l-1)},$$

$$a_j^{(l)} = g^{(l)}\left(z_j^{(l)}\right). \tag{1}$$

The activation function $g^{(l)}(\cdot)$ is specific for the *l-th* layer, although a single activation function g (·) common in all layers is often used for the entire network. Finally, the output layer is produced through the relationship between the *(L-1) -th* layer and the following *L-th* layer:

$$z_k^{(L)} = w_{0k}^{(L-1)} + \sum_{i=1}^{p_{L-1}} w_{ik}^{(L-1)} a_i^{(L-1)},$$

$$y_k = g^{(L)}\left(z_k^{(L)}\right).$$

Both the number of output nodes K and the transformation function $g^{(L)}(\cdot)$ depend on the problem in question. For a unchanged regression problem, there is typically only one output node, therefore $K = 1$, while, a suitable choice of transformation function is the identity function, $g^{(L)}(z^{(L)}) = z^{(L)}$. For a classification problem, the number of nodes K coincides with the number of classes of the response variable that you want to model. Each node *k* indicates the probability of belonging to the *k-th* class. As a transformation function, it is often convenient to use the multinomial logistic function,

$$g^{(L)}\left(z_k^{(L)}\right) = \frac{e^{z_k^{(L)}}}{\sum_{j=1}^{K} e^{z_j^{(L)}}}$$

which is called the *softmax* function. Now ask:

$$\mathbf{a}^{(1)} = x = \left[1 \;\; x_1 \ldots x_{p_1}\right]^T;$$

$$\mathbf{a}^{(l)} = \left[1 \; a_1^{(l)} \ldots a_{p_l}^{(l)}\right]^T;$$

$$\mathbf{w}_j^{(l)} = \left[w_{0j}^{(l)} \; w_{1j}^{(l)} \ldots w_{p_l j}^{(l)}\right]^T;$$

$$W^{(l)} = \left[\mathbf{w}_0^{(l)} \; \mathbf{w}_1^{(l)} \ldots \mathbf{w}_{p_{l+1}}^{(l)}\right]^T;$$

$$\mathbf{W} = \left[ W^{(1)} W^{(2)} \ldots W^{(L)} \right];$$

$$\boldsymbol{y} = [y_1 \ldots y_K]^T.$$

**Vector Notation** Adopting vector notation makes it easier and more intuitive formulate the relationship between two generic layers of the network:

$$\mathbf{z}^{(l)} = W^{(l-1)} \mathbf{a}^{(l-1)}, \tag{2}$$

$$\mathbf{a}^{(l)} = g^{(l)} \left( \mathbf{z}^{(l)} \right), \tag{3}$$

where the function *g (l) (·)* is applied element by element to the vector $z^{(l)}$. Consequently, the complete relationship between the input vector *x* and the output vector *y* is the following:

$$\mathbf{y} = \boldsymbol{f}(\mathbf{x}; \mathbf{W}) = \boldsymbol{g}^{(\boldsymbol{L})} \left( \boldsymbol{W}^{(\boldsymbol{L-1})} \boldsymbol{g}^{(\boldsymbol{L-1})} \left( \ldots \boldsymbol{W}^{(\mathbf{2})} \boldsymbol{g}^{(\mathbf{2})} \left( \boldsymbol{W}^{(\mathbf{1})} \mathbf{x} \right) \right) \right) \tag{4}$$

## *2.1 Calculation of Parameters Via Backpropagation*

For *regression* problems, we generally have a quantitative response variable $y = (y_1, ..., y_n) \in \mathrm{R}^n$, while for *classification* problems we use a qualitative response variable $y = (y_1, ..., y_n) \in T(y)^n = \{t_1, ..., t_K\}^n$, where T (y) is the set of modalities that can assume y. Consider a whole of data, consisting of n observations, for each of which are detected p explanatory variables, $x_i = (x_{i1}, ..., x_{ip}) \in \mathrm{R}^p$.

We want to adapt a neural network to the set of data, with the minimization of a given loss function $L[y, f(x; \mathbf{W})]$. This is achieved by looking for those values of the parameters $\hat{\mathbf{W}}$, such that

Loss function to be minimized is chosen from the following:

$$\hat{\mathbf{W}} = \arg\min_{\mathbf{W}} \left\{ \frac{1}{n} \sum_{i=1}^{n} L\left[ y_i, f(x_i; \mathbf{W}) \right] \right\}. \tag{5}$$

For **regression** problems

*Mean square error*

$$\mathrm{MSE}(\mathbf{W}) = \frac{1}{n} \sum\nolimits_{i=1}^{n} (y_i - f(x_i; \mathbf{W}))^2;$$

*Root of the MSE*

$$\text{rMSE}\,(\mathbf{W}) = \sqrt{\frac{1}{n}\sum\nolimits_{i=1}^{n}\left(y_i - f\,(x_i;\mathbf{W})\right)^2};$$

*Mean absolute error*

$$\text{MAE}\,(\mathbf{W}) = \frac{1}{n}\sum\nolimits_{i=1}^{n}\left|y_i - f\,(x_i;\mathbf{W})\right|.$$

For **classification** problems,
*Misclassification rate*

$$\text{H}\,(\mathbf{W}) = -\sum\nolimits_{i=1}^{n}\sum\nolimits_{k=1}^{K} y_{ik}\log f_k\,(x_i;\mathbf{W})$$

where $y_{ik} = 1$ if $y_i = t_k$, 0 otherwise. Then minimizing cross-entropy corresponds to maximizing the log-likelihood (Hastie et al., 2009). The algorithm most widely used to estimate and calculate neural networks, is the backpropagation algorithm, adapted to Deep Neural Networks (see Rumelhart et al. 1986).

## 2.2 Backpropagation Algorithm for Deep Neural Networks

1. Calculate the value of the node $a^{(l)}$ for each layer l = 2, ..., L, using the current values of W,
2. For the output layer l = L, calculate

$$\delta^{(L)} = \frac{\partial L\left[y_i,\,\hat{f}\,(x_i;\mathbf{W})\right]}{\partial\,\hat{f}\,(x_i;\mathbf{W})} \circ \dot{g}^{(L)}\left(\mathbf{z}^{(L)}\right); \tag{6}$$

3. For the hidden layers (l = L—1, ...,2) obtain

$$\delta^{(l)} = \left(W^{(l)'}\delta^{(l+1)}\right) \circ \dot{g}^{(l)}\left(\mathbf{z}^{(l)}\right); \tag{7}$$

4. Having $\delta_2$,...,$\delta_L$ it Is Possible to Derive the Partial Derivatives with

$$\frac{\partial L\left[y_i, \hat{f}\left(x_i; \mathbf{W}\right)\right]}{\partial W^{(l)}} = \delta^{(l+1)}\mathbf{a}^{(l)'}; \tag{8}$$

5. Update the W parameters using the gradient descent;
6. Start over with a new iteration from step 1, using the new values for the W parameters.

This algorithm solve Eq. (8), with a low computational cost. Normally most of the numerical optimization algorithms are iterative and require the calculation of the gradient of the loss function with respect to the parameters, of first and second order.

Keeping in mind that a multi-layered neural network has a very high number of parameters, the computational cost of calculating the second order gradient becomes excessive. If L are the layers, the network has L matrices of parameters W (l), each of which contains $p_l \times p_l + 1$ coefficients, where the number of nodes $p_l$ can reach a few thousand. For each iteration of the algorithm, the calculation of the first gradient requires a number of operations equal to the number of coefficients, while the operations required for the calculation of the second degree gradient grow quadratically as the number of parameters increases.

The advantage of the backpropagation algorithm is that, on the one hand, it does not require the second order gradient, and on the other, it calculates the first gradient only in the last layer, and then propagates it backwards in the other layers.

The algorithm alternates, for a given observation (xi, yi), with i = 1, ..., n, two steps iteratively: with the step *forward* you get f^ (xì; W) through (4), keeping W fixed, while with the step *backwards* you get the gradients and the parameters are updated. In machine learning, each iteration is called an *epoch*.

In the step forward, the value of the nodes a (l) for each layer l = 2, ..., L is calculated, using the current values of W (point 1 of algorithm backpropagation). Through formula (4), it is possible to obtain all the values of the nodes, a (l), and of the linear combinations, z (l), saving the intermediate quantities in progress. It is therefore necessary to initialize the parameters with randomly chosen values, close to 0.

Then the step backwards develops. This includes a *propagation phase* (points 2–4) and an *update phase* (point 5). The purpose of the propagation step is to compute all the partial derivatives $\partial L[y_i, \hat{f}(x_i; \mathbf{W})]$, with respect to the parameters. In practice, the quantities $\delta_L, ..., \delta_2$ are obtained, useful for calculating the partial derivatives, in an iterative way. The generic $\delta_l$ must be calculated as $\partial L[y_i, \hat{f}(x_i; \mathbf{W})]$ with respect to $z^{(l)}$. The $\delta_L$ of the output layer can be calculated with the "chain rule"; in substance δL is calculated as follows:

$$\begin{aligned}\delta^{(L)} &= \frac{\partial L\left[y_i, \hat{f}(x_i;\mathbf{W})\right]}{\partial \mathbf{z}^{(L)}} \\ &= \frac{\partial L\left[y_i, \hat{f}(x_i;\mathbf{W})\right]}{\partial \hat{f}(x_i;\mathbf{W})} \frac{\partial \hat{f}(x_i;\mathbf{W})}{\partial \mathbf{z}^{(L)}} \\ &= \frac{\partial L\left[y_i, \hat{f}(x_i;\mathbf{W})\right]}{\partial \hat{f}(x_i;\mathbf{W})} \circ \dot{g}^{(L)}\left(\mathbf{z}^{(L)}\right),\end{aligned}$$

where$\dot{g}^{(L)}$ indicates the first derivative of $g^{(L)}(\mathbf{z}^{(L)})$, and is easily obtained by deriving the expression (3) for l = L; the symbol ° indicates the Hadamard product (element by element product).

The $\delta^{(l)}$ of the generic layer l is obtained as follows:

$$\begin{aligned}\delta^{(l)} &= \frac{\partial L\left[y_i, \hat{f}(x_i;\mathbf{W})\right]}{\partial \mathbf{z}^{(L)}} \\ &= \frac{\partial L\left[y, \hat{f}(x_i;\mathbf{W})\right]}{\partial \mathbf{z}^{(l+1)}} \frac{\partial \mathbf{z}^{(l+1)}}{\partial \mathbf{a}^{(l)}} \frac{\partial \mathbf{a}^{(l)}}{\partial \mathbf{z}^{(l)}} \\ &= \delta^{(l+1)} \frac{\partial \mathbf{z}^{(l+1)}}{\partial \mathbf{a}^{(l)}} \frac{\partial \mathbf{a}^{(l)}}{\partial \mathbf{z}^{(l)}} \\ &= \left(W^{(l)'} \delta^{(l+1)}\right) \circ \dot{g}^{(l)}\left(\mathbf{z}^{(l)}\right),\end{aligned}$$

where

$$\frac{\partial \mathbf{z}^{(l+1)}}{\partial \mathbf{a}^{(l)}} = W^{(l)'}$$

is the first order gradient of (2). This expression correspond to (7) of the backpropagation algorithm and is named *backpropagation equation.*

Having $\delta_2$,...,$\delta_L$ it is possible to derive the partial derivatives with

$$\frac{\partial L\left[y_i, f\left(x_i; \mathbf{W}\right)\right]}{\partial W^{(l)}} = \frac{\partial L\left[y_i, f\left(x_i; \mathbf{W}\right)\right]}{\partial \mathbf{z}^{(l+1)}} \frac{\partial \mathbf{z}^{(l+1)}}{\partial W^{(l)}} = \delta^{(l+1)} \mathbf{a}^{(l)'},$$

In the updating phase, the parameter values are modified by means of the gradient descent, which uniquely uses the first-order partial derivatives, calculated in the propagation phase. The descent of the gradient is a numerical optimization technique that allows to find the minimum point of a function, using only the first derivatives.

Then the algorithm is restarted with a new iteration, using the new values for the W parameters.

## 3 The Gradient Descent

Let's now see the updating of the parameters, carried out through the descent of the gradient, which is what happens in point *5* of backpropagation algorithm. The

gradient descent, based on the delta rule, is the most common and immediate method for updating the W (l) parameters (point *5* of algorithm) (Bengio, 2012). In this case, the updating of the parameters, at step t, takes place according to the Formula

$$W_{t+1}^{(l)} = W_t^{(l)} - \eta \cdot \Delta L\left(W_t^{(l)}; x, y\right), \quad \text{per} \quad l = 1, \ldots, L-1$$

where

$$\Delta L\left(W_t^{(l)}; x, y\right)$$

Is the gradient respect to $W_t(i)$ of the argument of expression (5), that is gradient of

$$\frac{1}{n}\sum_{i=1}^{n} L = [y_i, f\,(x_i; W)]$$

the

$$\Delta L\left(W_t^{(l)}; x, y\right)$$

corresponds to

$$\Delta L\left(W_t^{(l)}; x, y\right) = \frac{1}{n}\sum_{i=1}^{n} \frac{\partial L\,[y_i, f\,(x_i; W)]}{\partial W_t^{(l)}}. \tag{9}$$

Essentially, if the gradient is negative, the loss function at that point is decreasing, which means that the parameter has to move towards larger values to reach a minimum point. Conversely, if the gradient is positive, the parameters have to shift towards smaller values to reach lower values of the loss function. The parameter $\eta \in (0, 1]$ is called the *learning rate*, and it determines the magnitude of the displacement.

## *3.1 Mini Batch Gradient Descent*

The previous method has several problems and limitations when applied to multi-layered neural networks. The use of all data to perform a single update step involves considerable computational costs and greatly slows down the estimation procedure. Furthermore, it is not possible to estimate the model if the dataset is too large and cannot be loaded entirely into memory. In this regard, the mini-batch gradient descent technique is introduced. This consists in dividing the dataset into subsamples of fixed number mxn, after a random permutation of the entire data set.

The update is then implemented using each of these subsets, through the formula

$$W_{t+1}^{(l)} = W_t^{(l)} - \eta \cdot \Delta L \left( W_t^{(l)}; x^{(i:i+m)}, y^{(i:i+m)} \right),$$

where (*i: i + m*) is the index to refer the observation subset from *i-th* to *(i + m) th.* Then, for each epoch, instead of a single updating (with all data) they are done many updatings (mini-batch) by using the mini-batch data.

Advantages of this technique are:

- With little part of observations it is possible to meet better minima.
- The algorithm steps are so much faster and this fact guarantees a fastest convergence towards the minimum point.

**The learning rate problem:** setting too small values can lead to a very slow convergence, while large values can make the parameters fluctuate around the minimum without bringing the algorithm to convergence. Furthermore, dealing with this quantity with classic regularization methods (such as cross-validation) can be computationally too expensive. Finally, it seems inappropriate to think that all parameters need the same learning rate value to converge optimally. The problem of entrapment in local minima far from the absolute minimum. Since the models covered are highly parameterized, the loss functions previous discussed are generally convex in f (x; W), but not in W. This means that L [y, f (x; W)] has a single point of minimum for f (x; W), which is obviously the absolute minimum. Conversely, L [y, f (x; W)] has several local minima for W, of which only one is absolute. Then solve Eq. (5) and find the absolute minimum for W is somewhat complex, due to the high risk of obtaining a local minimum (Hastie et al., 2009). The attempt to solve the aforementioned problems allowed the development of subsequent improvements with the mini-batch gradient descent (Duchi et al., 2014).

$$w_{t+1,ij}^{(l)} = w_{t,ij}^{(l)} - \frac{\eta}{\sqrt{G_{t,ij} + \varepsilon}} \cdot g_{t,ij},$$

where $G_{t,ij}$ is the sum of the squares of the gradients with respect to $w_{t,\,ij}^{(l)}$, up to time t, that is $G_{t,ij} = \Sigma_1{}^T (g_{t,ij})^2$. $\varepsilon$ instead is a is a smoothing term that serves to avoid a null term in denominator, and is usually set to values of order of $10^{-8}$. This allows to avoid the adjustment of the learning rate parameter, of which only an initial value is set, usually equal to 0.01.

Since, $G_{ij}$ is a sum of positive terms, this quantity continues to increase with each epoch, and the learning rate decreases until it tends to 0. This problem can be solved by iteratively redefining $G_{ij}$ as an average exponential mobile (EWMA). The mean at time t is then

$$\mathbf{E}\left[g^2\right]_{t,ij} = \gamma \mathbf{E}\left[g^2\right]_{t-1,ij} + (1-\gamma)\, g_{t,ij}^2,$$

where $\gamma$ is normally updated around 0.9.

The updating of the parameters therefore becomes

$$w_{t+1,ij}^{(l)} = w_{t,ij}^{(l)} - \frac{\eta}{\sqrt{\mathbf{E}\left[g^2\right]_{t,ij} + \varepsilon}} \cdot g_{t,ij}.$$

A further improvement is obtained by keeping in memory past values also of the term $g_{t,ij}$, and also applying an exponential moving average to the latter. This innovative method of gradient descent is called *Adam* (Kingma & Lei, 2015; Sebastian, 2016). It is determined with $m_{t,ij} = \mathbf{E}[g]_{t,ij}$ e $v_{t,ij} = \mathbf{E}[g^2]_{t,ij}$. The quantities are then defined

$$\begin{aligned} m_{t,ij} &= \beta_1 m_{t-1,ij} + (1-\beta_1)\, g_{t,ij}, \\ v_{t,ij} &= \beta_2 v_{t-1,ij} + (1-\beta_2)\, g_{t,ij}^2, \end{aligned}$$

where $m_{0,ij}$ *and* $v_{0,ij}$ are initialized to 0. and it is shown the correction:

$$\tilde{m}_{t,ij} = \frac{m_{t,ij}}{1-\beta_1^t}, \quad \tilde{v}_{t,ij} = \frac{v_{t,ij}}{1-\beta_2^t}.$$

Then the parameter updated becomes:

$$w_{t+,ij}^{(l)} = w_{t,ij}^{(l)} = \frac{\eta}{\sqrt{\tilde{m}_{t,ij} + \varepsilon}} \cdot \tilde{v}_{t,ij}, \tag{10}$$

with $\beta_1$, $\beta_2$ that must have values, respectively, 0.9 and 0.999. The method appears very efficient.

## 4 Deep Neural Networks and Convolutional Neural Networks

Deep neural networks are more difficult to train than shallow neural networks. On the other hand, deep networks are much more powerful than flat networks (Goodfellow et al., 2016). A widely used type of deep network is the convolutional deep neural network (CDNN).

Starting from shallow networks, through many iterations, we can build ever more powerful networks. The techniques to be inserted later are: *convolutions*, *pooling* and *GPU* (LeCun et al., 2015; Ronen & Shamir, 2015). To this we add the *algorithmic expansion of data training* to reduce overfitting, the use of the *dropout technique* (Srivastava et al., 2014) and network composition. Let's consider as an example: Manuscript classification, using figures from the MNIST datase.

Starting with convolutional networks (Delalleau & Bengio, 2011) with shallow networks, through successive iterations, we gradually build more complex networks:

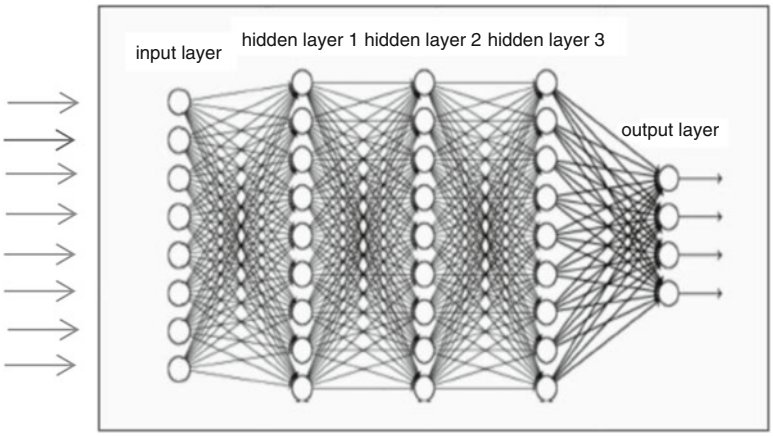


**Fig. 4** Convolutional N N, with: one input layer, three hidden layers, one output layer

The result will be a system that offers performance close to human. We will use the images not seen during training for the generalization test.

There have been spectacular recent advances in image recognition with convolutional networks; and also with recurrent neural networks, long- and short-term memory units, models that can be applied in speech recognition and natural language processing (Nielsen, 2015).

## *4.1 Convolutional Networks*

Here we have image recognition using networks with adjacent layers completely connected to each other (Krizhevsky et al., 2012). That is, every neuron in the network is connected to every neuron in the adjacent layers: Three basic ideas apply in convolutional neural networks: *local receptive fields, shared weights, and pools*. The input comes from squares of neurons, whose values correspond to the intensity of the pixels we are using (Fig. 4).

These squares are located in regions of the input image. Basically each neuron in the first hidden layer is connected to a small region of the input neurons, This region in the input image is called the *local receptive field*. Let's start with the top left corner and by scrolling the local receptive field over the entire input image we will have a different hidden neuron 'i' for each local receptive field (Fig. 5).

Steps greater than '1' and a direction different from the horizontal can be used. Shared weights and forecasts: each hidden neuron has a bias and weights connected to its local receptive field. We will use the same weights and biases for each of the hidden neurons. In practice, for the n.th hidden neuron, the output is: The use of the receptive field does not alter the recognizability of the image. The

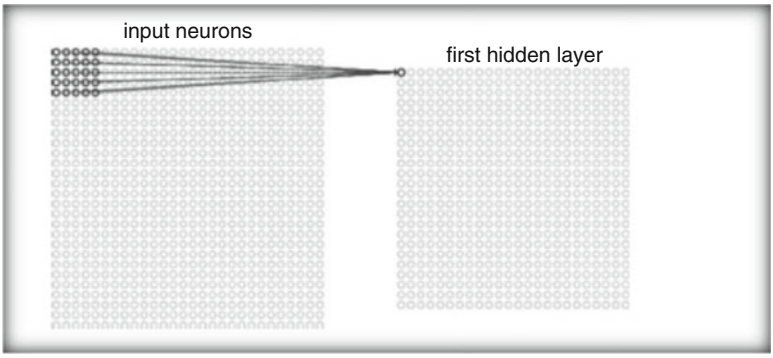


**Fig. 5** Receptive field connected to the first hidden layer

translation invariance of images also applies: the map from the input layer to the hidden layer is the feature map. We call the weights that define the characteristics in the map shared weights. The bias that defines the shared bias map file. The network map just described concerns only a localized feature (functionality). Image recognition requires multiple feature maps, so a full convolutional layer consists of several feature maps. Each map is defined by a set of shared weights and a single shared bias. The network can detect different types of feature-files, and each feature is detectable on the whole image. The images correspond to different feature maps (or filters). Each map is represented as a block image, corresponding to the weights in the local receptive field. Feature map example (see Fig. 7): The lighter blocks correspond to a smaller weight and the feature map responds less to the corresponding input pixels. Darker corresponds to greater weight, and the feature map responds more to corresponding input pixels.

Intuitively, it seems likely that the use of the translation invariance by the convolutional layer will reduce the number of parameters required to obtain the same performance as a fully connected model. This will also result in a faster workout. Intuitively, it seems likely that the use of the translation invariance by the convolutional layer reduces the number of parameters required to obtain the same performance as a fully connected model. This will also result in a faster training (Fig. 6).

**Pooling Layers** Pooling layers are placed immediately after the convolutional layers. The pooling layers simplify the information file that exits the convolutional layer: a pooling layer takes the output of each map of the characteristics of the convolutional layer and creates another map of condensed features.

For example, it condenses a region in the previous layer. Common procedure for pooling is *max-pooling*: the pooling unit takes only the maximum activation value in the input region (Fig. 7).

Example: max-pooling applied to each of three feature maps (see Fig. 8). The convolutional and max-pooling layers are similar to Neural Networks for Deep Learning.

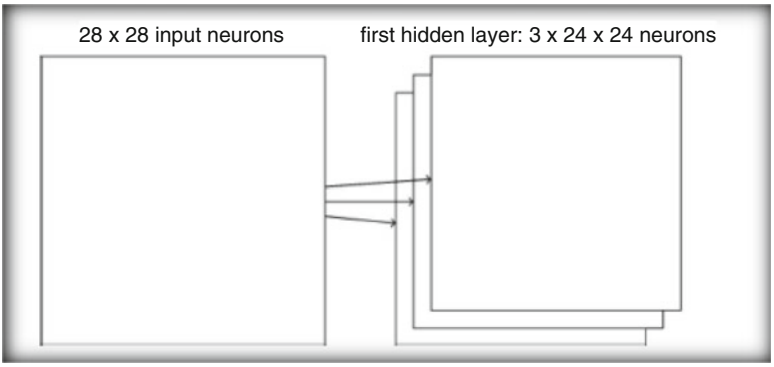


**Fig. 6** Input layer connected to three feature maps

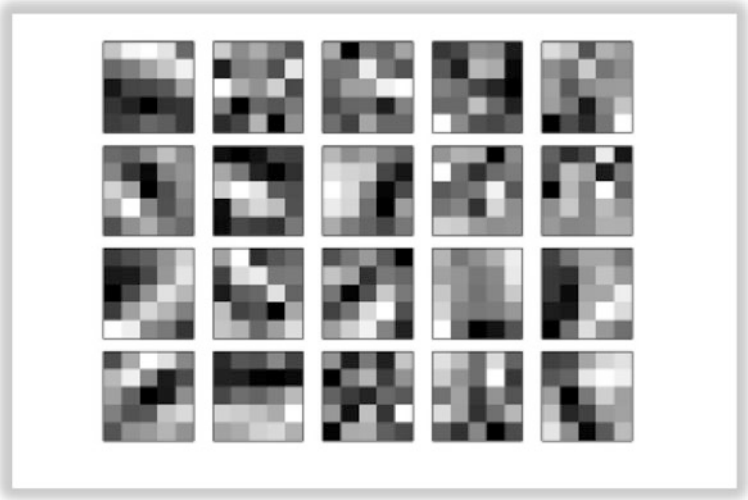

**Fig. 7** Feature map: block image, corresponding to the weights in the local receptive fields

**Using Rectified Linear Units** There are many ways to vary the network in an attempt to improve results.

For instance we can change neurons: ***instead of using the sigmoid activation, we use rectified linear units***. In practice We'll train for epochs. I also found soma advamtage by uinge some regularization, *with regularization parameter.*

**Expanding the Training Data** Another way to improve the results is by *algorithmically expanding the training data.* A simple way of expanding the training data is to ***displace each training image by a single pixel***, either up one pixel, down one pixel, left one pixel, or right one pixel. Using the expanded training data we can obtain a *better percent training accuracy*.

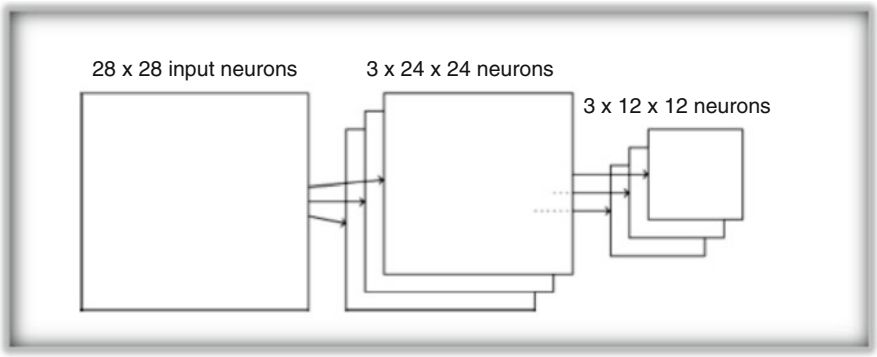


**Fig. 8** From Imput layer to 3 feature maps and then to 3 pooling maps

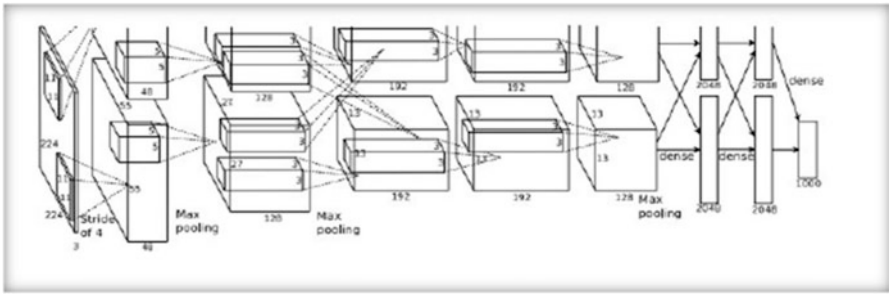


**Fig. 9** DCNN of Krizhevsky, Sutskever and Hinton

**Progress in Image Recognition** A best paper of Krizhevsky, Sutskever and Hinton appears in 2012 (Krizhevsky et al., 2012). **They** trained and tested a **DCNN** by a restricted subset of the ImageNet data. They used the ImageNet Large-Scale Visual Recognition Challenge (ILSVRC-2012). The used competition dataset gave them the possibility of comparing their approach with others. The ILSVRC-2012 training set contained about 1.2 million ImageNet images, from 1000 categories. From the same 1000 categories performed validation and test sets containing, respectively, 50,000 and 150,000 images.

As an example of good architecture it is intresting to see the DCNN of Krizhevsky, Sutskever and Hinton.

The DCNN of Krizhevsky, Sutskever and Hinton has layers of hidden neurons.

*The first hidden layers* are *convolutional* layers and some with max-pooling, the *next layers* are *fully-connected* layers.

Note the layers split into 2 parts, corresponding to the 2 GPUs.

*The input layer* contains neurons, representing the RGB values for a image. ImageNet contains images of varying resolution, while a neural network's input

layer is usually of a fixed size. The net dealt with this by rescaling each image so the shorter side had length .

The *first hidden* layer is a *convolutional layer*, with a max-pooling step. It uses 11 × 11 local receptive fields, and a stride length of 4 pixels. There are 96 feature maps, split into 48 feature maps on each *GPU.*

A *max-pooling* is in this and later layers, and done in 3 × 3 regions; pooling regions may be overlapped.

The *second hidden* layer is also *convolutional*, with a max pooling step. It uses 5 × 5 local receptive fields. There are 256 feature maps, split into 128 on each GPU.

The *input channels* are used *only by the feature maps*. This is because any single feature map only uses inputs from the same GPU.

The *third, fourth and fifth hidden* layers are also *convolutional*, but they do not involve max-pooling; their parameters are respectively:

– (3) 384 feature maps, with 3 × 3 local receptive fields, and 256 input channels;
– (4) 384 feature maps, with 3 × 3 local receptive fields, and 192 input channels;
– (5) 256 feature maps, with 3 × 3 local receptive fields, and 192 input channels.

The third layer involves some inter-GPU communication (see figure) so the feature maps use all 256 input channels.

The *sixth and seventh hidden* layers are *fully-connected* layers, with 4096 neurons in each layer.

***The output layer*** is a 1000-unit softmax layer.

## *4.2 Deep Learning and Knowledge Relevance*

The new era of Artificial Intelligence, linked to deep learning, was born with the overcoming of Expert Systems and the difficulties encountered in defining all the rules necessary to create a useful and efficient Expert System. In practice, the A.I. has gone from trying to provide the machine with the necessary knowledge, to making the machine learn this knowledge automatically. And this is how Machine Learning was born, and Deep Learning in its field, with the successes we know in the field of image recognition, speech, natural language, and in many other sectors in which Machine Learning is applicable.

In practice, the turning point took place by abandoning the design of systems that contained all the necessary knowledge for the intelligent machine, turning to the design of systems that independently learned the necessary knowledge. Machine Learning, after a period of interesting but not optimal results, has recently accelerated, thanks to progress in computer technology on the one hand, and to the development of decidedly efficient algorithms, based on innovative artificial neural networks, and, in this context, of Deep Learning. The singular aspect of this breakthrough is linked to the successes of these algorithms, whose dynamics and founding principles possessed dark sides and all to be investigated.

However, some glimmer is making its way. In particular by analyzing one of the best known and most effective algorithms: the Backpropagation Algorithm (Rumelhart et al., 1986; Shamir et al., 2010). The machine that learns to recognize things never seen before selects the information it treats based on its importance. The degree of importance of the information corresponds to its generalization. In practice, the machine that learns to recognize objects does so by evaluating the importance of the information that the object carries with it. In this regard, in the behavior of the Backpropagation Algorithm, we have seen just what has just been said. The algorithm in its iterations ends up filtering the unimportant information, and preserving the broader one, that is, of a general type. And therefore, after the training phase with the training set, the machine will be able to recognize objects never seen before. That is, the machine is able to generalize its knowledge.

The phases observed by Tishby and Zaslavsky (2015) during the run of back-propagation algorithm, in a deep network, can be summarized as follows:

*Initial state*: Layer 1 neurons encode everything about the input data including all information on its labels. In the higher layers, in which neurons are located, they are in almost random state, with little or no relationship to the data or their labels.

*Adaptation phase*. As the DL begins, neurons in the upper layers gain information on the input and get the best of adapting labels to it.

*Phase of change*. The layers suddenly change their behavior and begin to forget information about the input.

*Compression Phase*: the higher layers compress their representation of the input data, taking what is relevant to the output label. They take the best to predict the label.

*Balance between security and compression*. The last layer achieves a good balance, retaining only what is necessary to predict the label.

Naftali Tishby and others have analyzed deep neural networks and defined the 'Information Bottleneck Principle' (Tishby et al., 1999; Tishby & Zaslavsky, 2015).

In practice, this principle allow to reach the theoretical limits of the optimal information in the DNNs: that is, they say, obtain the generalization limits of finite samples. This is quantifiable both by the constrained generalization and by the simplicity of the network.

We can analise the compromise between the compression of input data (due to bottlneck) and the output layer that preserves the prediction of supervised target. Closely connected to this could be the optimal architecture of nn: layers number, characteristics, connections.

In their experiments, Tishby and Shwartz-Ziv monitored the amount of information each layer of a deep neural network held on the input data and the amount of information each held on the output label. The networks appear to converge at the theoretical limit of the information bottleneck: theoretical limit that represents the optimal system for extracting relevant information: the network appears to compress the input as much as possible without sacrificing the ability to accurately predict its label.

We can argue that this trade-off between input compression and output prediction can correspond to reducing (compressing) knowledge of the input, distinguishing

what is not necessary, and is lost, and preserving what is relevant (general) for the output.

If this can be seen as more then the behaviour of some algorithms, but will become a general computational method, we would revolutionize the design of deep learning systems by designing their optimal architecture.

## References

Bengio, Y. (2012). Practical recommendations for gradient-based training of deep architectures. In *Neural networks: Tricks of the trade* (pp. 437–478). Springer.

Delalleau, O., & Bengio, Y. (2011). Shallow vs. deep sum-product networks. In *Advances in neural information processing systems* (pp. 666–674).

Duchi, J. C., Jordan, M. I., Winwright, J., & Wibisono, A. (2014). Optimal rates for zero-order convex optimization:the power of two function evaluations. *arXiv:1312.2139v2.Mat oc.*

Goodfellow, I., Bengio, Y., & Courville, A. (2016). *Deep learning*. MIT Press.

Hastie, T., Friedman, J., & Tibshirani, R. (2009). *The elements of statistical learning: Data mining, inference, and prediction*. Springer.

Kingma, D. P., & Lei, J. (2015). Adam: A method for stochastic optimization. In *Proceedings of ICLR 2015*.

Krizhevsky, A., Sutskever, I., & Hinton, G. E. (2012). ImageNet classification with deep convolutional neural networks. In *Advances in neural information processing systems (NIPS)* (pp. 1106–1114).

LeCun, Y., Bengio, Y., & Hinton, G. (2015). Deep learning. *Nature, 521*(7553), 436–444.

Nielsen, M. A. (2015). *Neural networks and deep learning*. Determination Press.

Ronen E., Shamir O. (2015). The power of depth for feedforward neural networks. *arXiv preprint*.

Rumelhart, D., Hinton, G. E., & Williams, R. J. (1986). Learning representations by back-propagating errors. *Nature, 323*(6088), 533–536.

Sebastian, R. (2016). An overview of gradient descent optimization algorithms. *arXiv preprint ar Xiv:*1609.*04747.*

Shamir, O., Sabato, S., & Thishby, N. (2010). Learning and generalization with the information bottelneck. *Theoretical Computer Science, 411*(29–30), 2696–2711.

Srivastava, N., Hinton, G. E., Krizhevsky, A., Sutskever, I., & Salakutdinov, R. (2014). Dropout: A simple way to prevent neural networks from overfitting. *The Journal of Machine Learning Research, 15*(1), 1929–1958.

Tishby, N., Pereira, F. C., & Bialek, W. (1999). The information bottleneck method. In *Proceeding of the 37th Annual Allerton Conference on Communication, Control and Computing* (pp. 368–377).

Tishby, N., & Zaslavsky, N. (2015). Deep learning and the information bottleneck principle. In *2015 IEEE Information Theory Workshop (ITW), Jerusalem* (pp. 1–5). https://doi.org/10.1109/ITW.2015.7133169