Fakultät für Mathematik, Informatik und Naturwissenschaften
Lehr- und Forschungsgebiet Informatik VIII
Computer Vision
Prof. Dr. Bastian Leibe

Seminar Report

Understanding Convolutional Neural Networks

David Stutz

March 7, 2015

Advisor: Lucas Beyer

Abstract

This seminar paper focusses on convolutional neural networks and a visualization technique allowing further insights into their internal operation. After giving a brief introduction to neural networks and the multilayer perceptron, we review both supervised and unsupervised training of neural networks in detail. In addition, we discuss several approaches to regularization.

The second section introduces the different types of layers present in recent convolutional neural networks. Based on these basic building blocks, we discuss the architecture of the traditional convolutional neural network as proposed by LeCun et al. [LBD+89] as well as the architecture of recent implementations.

The third section focusses on a technique to visualize feature activations of higher layers by backprojecting them to the image plane. This allows to get deeper insights into the internal working of convolutional neural networks such that recent architectures can be evaluated and improved even further.

Contents

1 Motivation

Artificial neural networks are motivated by the learning capabilities of the human brain which consists of neurons interconnected by synapses. In fact – at least theoretically – they are able to learn any given mapping up to arbitrary accuracy [HSW89]. In addition, they allow to easily incorporate prior knowledge about the task into the network architecture. As result, in 1989, LeCun et al. introduced convolutional neural networks for application in computer vision [LBD+89].
Convolutional neural networks use images directly as input. Instead of handcrafted features, convolutional neural networks are used to automatically learn a hierarchy of features which can then be used for classification purposes. This is accomplished by successively convolving the input image with learned filters to build up a hierarchy of feature maps. The hierarchical approach allows to learn more complex, as well as translation and distortion invariant, features in higher layers.
In contrast to traditional multilayer perceptrons, where deep learning is considered difficult [Ben09], deep convolutional neural networks can be trained more easily using traditional methods[1]. This property is due to the constrained architecture[2] of convolutional neural networks which is specific to input for which discrete convolution is defined, such as images. Nevertheless, deep learning of convolutional neural networks is an active area of research, as well.
As with multilayer perceptrons, convolutional neural networks still have some disadvantages when compared to other popular machine learning techniques as for example Support Vector Machines as their internal operation is not well understood [ZF13]. Using deconvolutional neural networks proposed in [ZKTF10], this problem is addressed in [ZF13]. The approach described in [ZF13] allows the visualization of feature activations in higher layers of the network and can be used to give further insights into the internal operation of convolutional neural networks.

1.1 Bibliographical Notes

Although this paper briefly introduces the basic notions of neural networks as well as network training, this topic is far too extensive to be covered in detail. For a detailed discussion of neural networks and their training several textbooks are available [Bis95, Bis06, Hay05].
The convolutional neural network was originally proposed in [LBD+89] for the task of ZIP code recognition. Both convolutional neural networks as well as traditional multilayer perceptrons were excessively applied to character recognition and handwritten digit recognition [LBBH98]. Training was initially based on error backpropagation [RHW86] and gradient descent.
The original convolutional neural network is based on weight sharing which was proposed in [RHW86]. An extension of weight sharing called soft weight sharing is discussed in [NH92]. Recent implementations make use of other regularization techniques as for example dropout [HSK+12].
Although the work by Hinton et al. in 2006 [HO06] can be considered as breakthrough in deep learning – as it allows unsupervised training of neural networks – deep learning is still considered difficult [Ben09]. A thorough discussion of deep learning including recent research is given in [Ben09] as well as [LBLL09, GB10, BL07]. Additional research on this topic includes discussion on activation functions as well as the effect of unsupervised pre-training [EMB+09, EBC+10, GBB11].
Recent architectural changes of convolutional neural networks are discussed in detail in [JKRL09] and [LKF10]. Recent success of convolutional neural networks is reported in [KSH12] and [CMS12].
This paper is mainly motivated by the experiments in [ZF13]. Based on deconvolutional neural networks [ZKTF10], the authors of [ZF13] propose a visualization technique allowing to visualize feature activations of higher layers.

[1] Here, traditional methods refers to gradient descent for parameter optimization combined with error backpropagation as discussed in section 2.3.

[2] Using weight sharing as discussed in section 2.5.4, the actual model complexity is reduced.

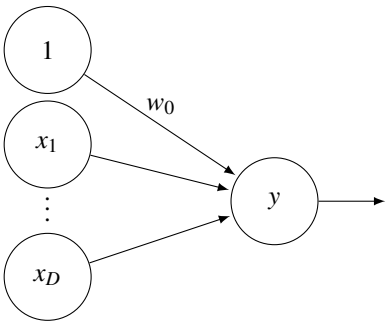


Figure 1: A processing unit consists of a propagation rule mapping all inputs $w_0, x_1 \ldots, x_D$ to the actual input z, and an activation function f which is applied on the actual input to form the output $y = f(z)$. Here, w_0 represents an external input called bias and $x_1, \ldots, x_D$ are inputs from other units of the network. In a network graph, each unit is labeled according to its output. Therefore, to include the bias w_0 as well, a dummy unit (see section 2.1) with value 1 is included.

2 Neural Networks and Deep Learning

An (artificial) neural network comprises a set of interconnected processing units [Bis95, p. 80-81]. Given input values $w_0, x_1, \ldots, x_D$, where w_0 represents an external input and $x_1, \ldots, x_D$ are inputs originating from other processing units within the network, a processing unit computes its output as $y = f(z)$. Here, f is called activation function and z is obtained by applying a propagation rule which maps all the inputs to the actual input z. This model of a single processing unit includes the definition of a neuron in [Hay05] where instead of a propagation rule an adder is used to compute z as the weighted sum of all inputs.
Neural networks can be visualized in the means of a directed graph[3] called network graph [Bis95, p. 117-120]. Each unit is represented by a node labeled according to its output and the units are interconnected by directed edges. For a single processing unit this is illustrated in figure 1 where the external input w_0 is only added for illustration purposes and is usually omitted [Bis95, p. 116-120].
For convenience, we distinguish input units and output units. An input unit computes the output $y := x$ where x is the single input value of the unit. Output units may accept an arbitrary number of input values. Altogether, the network represents a function $y(x)$ which dimensions are fixed by the number of input units and output units, this means the input of the network is accepted by the input units and the output units form the output of the network.

2.1 Multilayer Perceptrons

A $(L+1)$-layer perceptron, illustrated in figure 2, consists of D input units, C output units, and several so called hidden units. The units are arranged in layers, that is a multilayer perceptron comprises an input layer, an output layer and L hidden layers[4] [Bis95, p. 117-120]. The i^{th} unit within layer l computes the output

$$y_i^{(l)} = f\left(z_i^{(l)}\right) \quad \text{with} \quad z_i^{(l)} = \sum_{k=1}^{m^{(l-1)}} w_{i,k}^{(l)} y_k^{(l-1)} + w_{i,0}^{(l)} \tag{1}$$

where $w_{i,k}^{(l)}$ denotes the weighted connection from the k^{th} unit in layer $(l-1)$ to the i^{th} unit in layer l, and $w_{i,0}^{(l)}$ can be regarded es external input to the unit and is referred to as bias. Here, $m^{(l)}$ denotes the number of units in layer l, such that $D = m^{(0)}$ and $C = m^{(L+1)}$. For simplicity, the bias can be regarded as weight when introducing a dummy unit $y_0^{(l)} := 1$ in each layer:

$$z_i^{(l)} = \sum_{k=0}^{m^{(l-1)}} w_{i,k}^{(l)} y_k^{(l-1)} \quad \text{or} \quad z^{(l)} = w^{(l)} y^{(l-1)} \tag{2}$$

where $z^{(l)}$, $w^{(l)}$ and $y^{(l-1)}$ denote the corresponding vector and matrix representations of the actual inputs $z_i^{(l)}$, the weights $w_{i,k}^{(l)}$ and the outputs $y_k^{(l-1)}$, respectively.

[3] In its most general form, a directed graph is an ordered pair $G = (V, E)$ where V is a set of nodes and E a set of edges connecting the nodes: $(u, v) \in E$ means that a directed edge from node u to v exists within the graph. In a network graph, given two units u and v, a directed edge from u to v means that the output of unit u is used by unit v as input.

[4] Actually, a $(L+1)$-layer perceptron consists of $(L+2)$ layers including the input layer. However, as stated in [Bis06], the input layer is not counted as there is no real processing taking place (input units compute the identity).

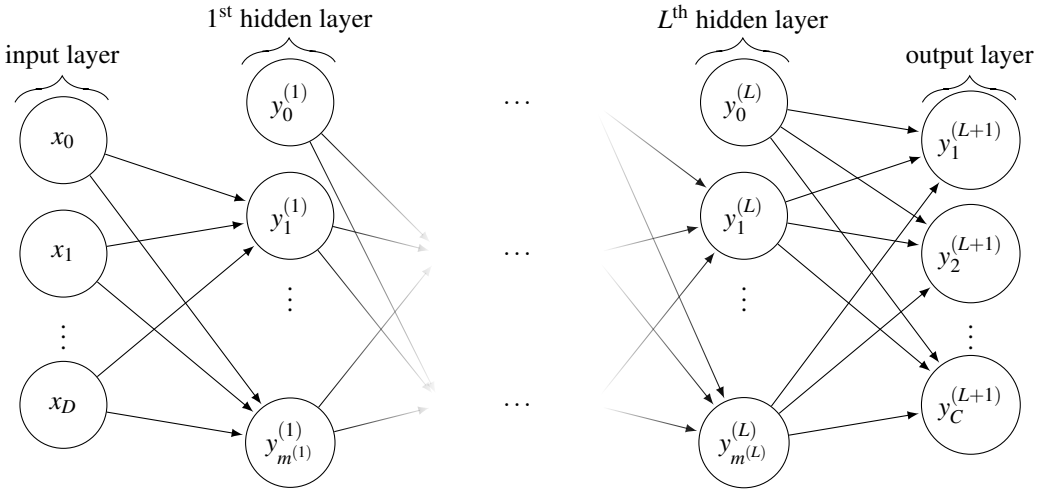


Figure 2: Network graph of a $(L+1)$-layer perceptron with D input units and C output units. The l^{th} hidden layer contains $m^{(l)}$ hidden units.

Overall, a multilayer perceptron represents a function

$$y(\cdot, w) : \mathbb{R}^D \to \mathbb{R}^C, x \mapsto y(x, w) \tag{3}$$

where the output vector $y(x,w)$ comprises the output values $y_i(x,w) := y_i^{(L+1)}$ and w is the vector of all weights within the network.
We speak of deep neural networks when there are more than three hidden layers present [Ben09]. The training of deep neural networks, referred to as deep learning, is considered especially challenging [Ben09].

2.2 Activation Functions

In [Hay05, p. 34-37], three types of activation functions are discussed: threshold functions, piecewise-linear functions and sigmoid functions. A common threshold function is given by the Heaviside function:

$$h(z) = \begin{cases} 1 & \text{if } z \geq 0 \\ 0 & \text{if } z < 0 \end{cases}. \tag{4}$$

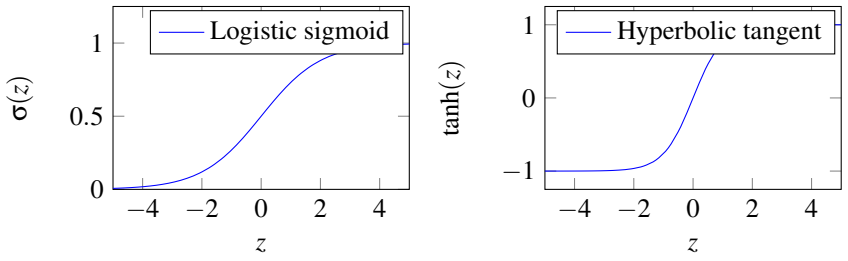
However, both threshold functions as well as piecewise-linear functions have some drawbacks. First, for network training we may need the activation function to be differentiable. Second, nonlinear activation functions are preferable due to the additional computational power they induce [DHS01, HSW89].
The most commonly used type of activation functions are sigmoid functions. As example, the logistic sigmoid is given by

$$\sigma(z) = \frac{1}{1+\exp(-z)}. \tag{5}$$

Its graph is s-shaped and it is differentiable as well as monotonic. The hyperbolic tangent $\tanh(z)$ can be regarded as linear transformation of the logistic sigmoid onto the interval $[-1,1]$. Note, that both activation functions are saturating [DHS01, p. 307-308].
When using neural networks for classification[5], the softmax activation function for output units is used to interpret the output values as posterior probabilities[6]. Then the output of the i^{th} unit in the output layer is

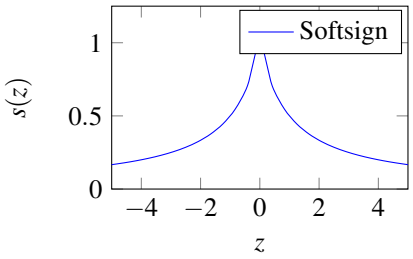
[5]The classification task can be stated as follows: Given an input vector x of D dimensions, the goal is to assign x to one of C discrete classes [Bis06].

[6]The outputs $y_i^{(L+1)}$, $1 \leq i \leq C$, can be interpreted as probabilities as they lie in the interval $[0,1]$ and sum to 1 [Bis06].

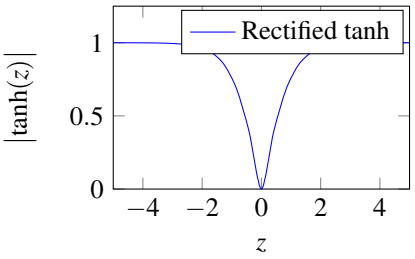


(a) Logistic sigmoid activation function.

(b) Hyperbolic tangent activation function.



(c) Logistic sigmoid activation function.



(d) Rectified hyperbolic tangent activation function.

Figure 3: Common used activation functions include the logistic sigmoid $\sigma(z)$ defined in equation (5) and the hyperbolic tangent $tanh(z)$. More recently used activation functions are the softsign of equation (7) and the rectified hyperbolic tangent.

given by

$$\sigma(z^{(L+1)}, i) = \frac{\exp(z_i^{(L+1)})}{\sum_{k=1}^{C} \exp(z_k^{(L+1)})}. \tag{6}$$

Experiments in [GB10] show that the logistic sigmoid as well as the hyperbolic tangent perform rather poorly in deep learning. Better performance is reported using the softsign activation function:

$$s(z) = \frac{1}{1+|z|}. \tag{7}$$

In [KSH12] a non-saturating activation function is used:

$$r(z) = \max(0, z). \tag{8}$$

Hidden units using the activation function in equation (8) are called rectified linear units[7]. Furthermore, in [JKRL09], rectification in addition to the hyperbolic tangent activation function is reported to give good results. Some of the above activation functions are shown in figure 3

2.3 Supervised Training

Supervised training is the problem of determining the network weights to approximate a specific target mapping g. In practice, g may be unknown such that the mapping is given by a set of training data. The training set

$$T_S := \{(x_n, t_n) : 1 \leq n \leq N\} \tag{9}$$

comprises both input values x_n and corresponding desired, possibly noisy, output values $t_n \approx g(x_n)$ [Hay05].

[7] Also abbreviated as ReLUs.

2.3.1 Error Measures

Training is accomplished by adjusting the weights w of the neural network to minimize a chosen objective function which can be interpreted as error measure between network output $y(x_n)$ and desired target output t_n. Popular choices for classification include the sum-of-squared error measure given by

$$E(w) = \sum_{n=1}^{N} E_n(w) = \sum_{n=1}^{N} \sum_{k=1}^{C} (y_k(x_n, w) - t_{n,k})^2, \tag{10}$$

and the cross-entropy error measure given by

$$E(w) = \sum_{n=1}^{N} E_n(w) = \sum_{n=1}^{N} \sum_{k=1}^{C} t_{n,k} \log(y_k(x_n, w)), \tag{11}$$

where $t_{n,k}$ is the k^{th} entry of the target value t_n. Details on the choice of error measure and their properties can be found in [Bis95].

2.3.2 Training Protocols

[DHS01] considers three training protocols:

Stochastic training An input value is chosen at random and the network weights are updated based on the error $E_n(w)$.

Batch training All input values are processed and the weights are updated based on the overall error $E(w) = \sum_{n=1}^{N} E_n(w)$.

Online training Every input value is processed only once and the weights are updated using the error $E_n(w)$.

Further discussion of these protocols can be found in [Bis06] and [DHS01]. A common practice (e.g. used for experiments in [GBB11], [GB10]) combines stochastic training and batch training:

Mini-batch training A random subset $M \subseteq \{1, \ldots, N\}$ (mini-batches) of the training set is processed and the weights are updated based on the cumulative error $E_M(w) := \sum_{n \in M} E_n(w)$.

2.3.3 Parameter Optimization

Considering stochastic training we seek to minimize E_n with respect to the network weights w. The necessary criterion can be written as

$$\frac{\partial E_n}{\partial w} = \nabla E_n(w) \stackrel{!}{=} 0 \tag{12}$$

where ∇E_n is the gradient of the error E_n.
Due to the complexity of the error E_n, a closed-form solution is usually not possible and we use an iterative approach. Let $w[t]$ denote the weight vector in the t^{th} iteration. In each iteration we compute a weight update $\Delta w[t]$ and update the weights accordingly [Bis06, p. 236-237]:

$$w[t+1] = w[t] + \Delta w[t]. \tag{13}$$

From unconstrained optimization we have several optimization techniques available. Gradient descent is a first-order method, this means it uses only information of the first derivative of E_n and can, thus, be used in combination with error backpropagation as described in section 2.3.5, whereas Newton's method is a second-order method and needs to evaluate the Hessian matrix H_n of E_n[8] (or an appropriate approximation of the Hessian matrix) in each iteration step.

[8] The Hessian matrix H_n of a the error E_n is the matrix of second-order partial derivatives: $(H_n)_{r,s} = \frac{\partial^2 E_n}{\partial w_r \partial w_s}$

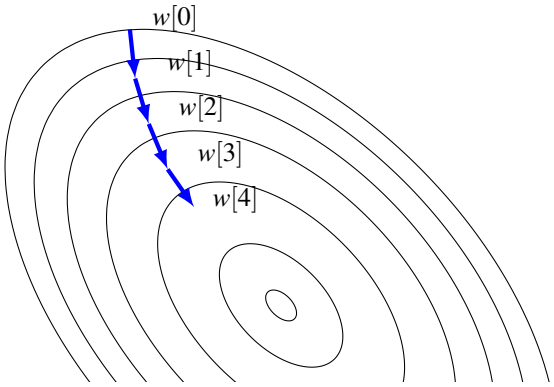


Figure 4: Illustrated using a quadratic function to minimize, the idea of gradient descent is to follow the negative gradient at the current position as it describes the direction of the steepest descent. The learning rate γ describes the step size taken in each iteration step. Therefore, gradient descent describes a first-order optimization technique.

Gradient descent Gradient descent is motivated by the idea to take a step in the direction of the steepest descent, that is the direction of the negative gradient, to reach a minimum [Bis95, p. 263-267]. This principle is illustrated by figure 4. Therefore, the weight update is given by

$$\Delta w[t] = -\gamma \frac{\partial E_n}{\partial w[t]} = -\gamma \nabla E_n(w[t]) \tag{14}$$

where γ is the learning rate. As discussed in [Bis06, p.263-272], this approach has several difficulties, for example how to choose the learning rate to get fast learning but at the same time avoid oscillation[9].

Newton's method Although there are some extensions of gradient descent available, second-order methods promise faster convergence because of the use of second-order information [BL89]. When using Newton's method, the weight update $\Delta w[t]$ is given by

$$\Delta w[t] = -\gamma \left(\frac{\partial^2 E_n}{\partial w[t]^2} \right)^{-1} \frac{\partial E_n}{\partial w[t]} = -\gamma \left(H_n(w[t]) \right)^{-1} \nabla E_n(w[t]) \tag{15}$$

where $H_n(w[t])$ is the Hessian matrix of E_n and γ describes the learning rate. The drawback of this method is the evaluation and inversion of the Hessian matrix[10] which is computationally expensive [BL89].

2.3.4 Weight Initialization

As we use an iterative optimization technique, the initialization of the weights w is crucial. [DHS01, p. 311-312] suggest choosing the weights randomly in the range

$$-\frac{1}{\sqrt{m^{(l-1)}}} < w_{i,j}^{(l)} < \frac{1}{\sqrt{m^{(l-1)}}}. \tag{16}$$

This result is based on the assumption that the inputs of each unit are distributed according to a Gaussian distribution and ensures that the actual input is approximately of unity order. Given logistic sigmoid activation functions, this is meant to result in optimal learning [DHS01, p. 311-312].
In [GB10] an alternative initialization scheme called normalized initialization is introduced. We choose the weights randomly in the range

$$-\frac{\sqrt{6}}{\sqrt{m^{(l-1)} + m^{(l)}}} < w_{i,j}^{(l)} < \frac{\sqrt{6}}{\sqrt{m^{(l-1)} + m^{(l)}}}. \tag{17}$$

The derivation of this initialization scheme can be found in [GB10]. Experimental results in [GB10] demonstrate improved learning when using normalized initialization.
An alternative to these weight initialization schemes is given by layer-wise unsupervised pre-training as discussed in [EBC+10]. We discuss unsupervised training in section 2.4.

[9] Oscillation occurs if the learning rate is chosen too large such that the algorithm successively oversteps the minimum.

[10] An algorithm to evaluate the Hessian matrix based on error backpropagation as introduced in section 2.3.5 can be found in [Bis92]. The inversion of an $n \times n$ matrix has complexity $O(n^3)$ when using the LU decomposition or similar techniques.

2.3.5 Error Backpropagation

Algorithm 2.1, proposed in [RHW86], is used to evaluate the gradient $\nabla E_n(w[t])$ of the error function E_n in each iteration step. More details as well as a thorough derivation of the algorithm can be found in [Bis95] or [RHW86].

Algorithm 2.1 (Error Backpropagation)
1. Propagate the input value x_n through the network to get the actual input and output of each unit.

2. Calculate the so called errors $\delta_i^{(L+1)}$ [Bis06, p. 241-245] for the output units:

$$\delta_i^{(L+1)} := \frac{\partial E_n}{\partial y_i^{(L+1)}} f'(z_i^{(L+1)}). \tag{18}$$

3. Determine $\delta_i^{(l)}$ for all hidden layers l by using error backpropagation:

$$\delta_i^{(l)} := f'(z_i^{(l)}) \sum_{k=1}^{m^{(l+1)}} w_{i,k}^{(l+1)} \delta_k^{(l+1)}. \tag{19}$$

4. Calculate the required derivatives:

$$\frac{\partial E_n}{\partial w_{j,i}^{(l)}} = \delta_j^{(l)} y_i^{(l-1)}. \tag{20}$$

2.4 Unsupervised Training

In unsupervised training, given a training set

$$T_U := \{x_n : 1 \leq n \leq N\} \tag{21}$$

without desired target values, the network has to find similarities and regularities within the data by itself. Among others, unsupervised training of deep architectures can be accomplished based on Restricted Boltzman Machines[11] or auto-encodes [Ben09]. We focus on auto-encoders.

2.4.1 Auto-Encoders

Auto-encoders, also called auto-associators [Ben09], are two-layer perceptrons with the goal to compute a representation of the input in the first layer from which the input can accurately be reconstructed in the output layer. Therefore, no desired target values are needed – auto-encoders are self-supervised [Ben09]. In the hidden layer, consisting of $m := m^{(1)}$ units, an auto-encoder computes a representation $c(x)$ from the input x [Ben09]:

$$c_i(x) = \sum_{k=0}^{D} w_{i,k}^{(1)} x_k. \tag{22}$$

The output layer tries to reconstruct the input from the representation given by $c(x)$:

$$\hat{x}_i = d_i(c(x)) = \sum_{k=0}^{m} w_{i,k}^{(2)} c_k(x). \tag{23}$$

As the output of an auto-encoder should resemble its input, it can be trained as discussed in section 2.3 by replacing the desired target values t_n used in the error measure by the input x_n. In the case where $m < D$, the auto-encoder is expected to compute a useful, dimensionality-reducing representation of the input. If $m \geq D$, the auto-encoder could just learn the identity such that $\hat{x}$ would be a perfect reconstruction of x. However, as discussed in [Ben09], in practice this is not a problem.

[11] A brief introduction to Restricted Boltzman Machines can be found in [Ben09].

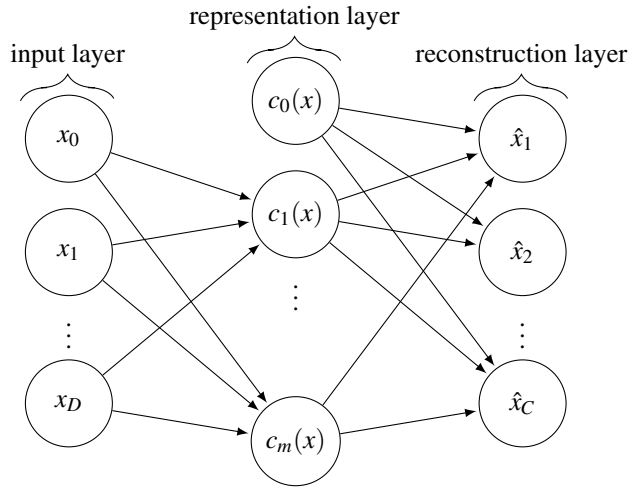


Figure 5: An auto-encoder is mainly a tow-layer perceptron with $m := m^{(1)}$ hidden units and the goal to compute a representation $c(x)$ in the first layer from which the input can accurately be reconstructed in the output layer.

2.4.2 Layer-Wise Training

As discussed in [LBLL09], the layers of a neural network can be trained in an unsupervised fashion using the following scheme:

1. For each layer $l = 1, \ldots, L+1$:
   - Train layer l using the approach discussed above taking the output of layer $(l-1)$ as input, associating the output of layer l with the representation $c(y^{(l-1)})$ and adding an additional layer to compute $\hat{y}^{(l)}$.

2.5 Regularization

It has been shown, that multilayer perceptrons with at least one hidden layer can approximate any target mapping up to arbitrary accuracy [HSW89]. Thus, the training data may be overfitted, that is the training error may be very low on the training set but high on unseen data [Ben09]. Regularization describes the task to avoid overfitting to give better generalization performance, meaning that the trained network should also perform well on unseen data [Hay05]. Therefore, the training set is usually split up into an actual training set and a validation set. The neural network is then trained using the new training set and its generalization performance is evaluated on the validation set [DHS01].
There are different methods to perform regularization. Often, the training set is augmented to introduce certain invariances the network is expected to learn [KSH12]. Other methods add a regularization term to the error measure aiming to control the complexity and form of the solution [Bis95]:

$$\hat{E}_n(w) = E_n(w) + \eta P(w) \tag{24}$$

where $P(w)$ influences the form of the solution and η is a balancing parameter.

2.5.1 L_p-Regularization

A popular example of L_p-regularization is the L_2-regularization[12]:

$$P(w) = \|w\|_2^2 = w^T w. \tag{25}$$

The idea is to penalize large weights as they tend to result in overfitting [Bis95]. In general, arbitrary p can be used to perform L_p-regularization. Another example sets $p = 1$[13] to enforce sparsity of the weights, that is many of the weights should vanish:

$$P(w) = \|w\|_1. \tag{26}$$

[12] The L_2-regularization is often referred to as weight decay, see [Bis95] for details.
[13] For $p = 1$, the norm $\|\cdot\|_1$ is defined by $\|w\|_1 = \sum_{k=1}^{W} |w_k|$ where W is the dimension of the weight vector w.

2.5.2 Early Stopping

While the error on the training set tends to decrease with the number of iterations, the error on the validation set usually starts to rise again once the network starts to overfit the training set. To avoid overfitting, training can be stopped as soon as the error on the validation set reaches a minimum, that is before the error on the validation set rises again [Bis95]. This method is called early stopping.

2.5.3 Dropout

In [HSK+12] another regularization technique, based on observation of the human brain, is proposed. Whenever the neural network is given a training sample, each hidden unit is skipped with probability $\frac{1}{2}$. This method can be interpreted in different ways [HSK+12]. First, units cannot rely on the presence of other units. Second, this method leads to the training of multiple different networks simultaneously. Thus, dropout can be interpreted as model averaging[14].

2.5.4 Weight Sharing

The idea of weight sharing was introduced in [RHW86] in the context of the T-C problem[15]. Weight sharing describes the idea of different units within the same layer to use identical weights. This can be interpreted as a regularization method as the complexity of the network is reduced and prior knowledge may be incorporated into the network architecture. The equality constraint is replaced when using soft weight sharing, introduced in [NH92]. Here, a set of weights is encouraged not to have the same weight value but similar weight values. Details can be found in [NH92] and [Bis95].
When using weight sharing, error backpropagation can be applied as usual, however, equation (20) changes to

$$\frac{\partial E_n}{\partial w_{j,i}^{(l)}} = \sum_{k=1}^{m^{(l)}} \delta_k^{(l)} y_i^{(l-1)} \tag{27}$$

when assuming that all units in layer l share the same set of weights, that is $w_{j,i}^{(l)} = w_{k,i}^{(l)}$ for $1 \leq j,k \leq m^{(l)}$. Nevertheless, equation (20) still needs to be applied in the case that the errors need to be propagated to preceding layers [Bis06].

2.5.5 Unsupervised Pre-Training

Results in [EBC+10] suggest that layer-wise unsupervised pre-training of deep neural networks can be interpreted as regularization technique[16]. Layer-wise unsupervised pre training can be accomplished using a similar scheme as discussed in section 2.4.2:

1. For each $l = 1,\ldots,L+1$:
   - Train layer l using the approach discussed in section 2.4.1.
2. Fine-tune the weights using supervised training as discussed in section 2.3.

A formulation of the effect of unsupervised pre-training as regularization method is proposed in [EMB+09]: The regularization term punishes weights outside a specific region in weight space with an infinite penalty such that

$$P(w) = -\log(p(w)) \tag{28}$$

where $p(w)$ is the prior for the weights, which is zero for weights outside this specific region [EBC+10].

[14] Model averaging tries to reduce the error by averaging the prediction of different models [HSK+12].

[15] The T-C problem describes the task of classifying images into those containing a "T" and those containing a "C" independent of position and rotation [RHW86].

[16] Another interpretation of unsupervised pre-training is that it initializes the weights in the basin of a good local minimum and can therefore be interpreted as optimization aid [Ben09].

3 Convolutional Neural Networks

Although neural networks can be applied to computer vision tasks, to get good generalization performance, it is beneficial to incorporate prior knowledge into the network architecture [LeC89]. Convolutional neural networks aim to use spatial information between the pixels of an image. Therefore, they are based on discrete convolution. After introducing discrete convolution, we discuss the basic components of convolutional neural networks as described in [JKRL09] and [LKF10].

3.1 Convolution

For simplicity we assume a grayscale image to be defined by a function

$$I : \{1,\dots,n_1\} \times \{1,\dots,n_2\} \to W \subseteq \mathbb{R}, (i,j) \mapsto I_{i,j} \tag{29}$$

such that the image I can be represented by an array of size $n_1 \times n_2$[17]. Given the filter $K \in \mathbb{R}^{2h_1+1 \times 2h_2+1}$, the discrete convolution of the image I with filter K is given by

$$(I * K)_{r,s} := \sum_{u=-h_1}^{h_1} \sum_{v=-h_2}^{h_2} K_{u,v} I_{r+u,s+v} \tag{30}$$

where the filter K is given by

$$K = \begin{pmatrix} K_{-h_1,-h_2} & \dots & K_{-h_1,h_2} \\ \vdots & K_{0,0} & \vdots \\ K_{h_1,-h_2} & \dots & K_{h_1,h_2} \end{pmatrix}. \tag{31}$$

Note that the behavior of this operation towards the borders of the image needs to be defined properly[18]. A commonly used filter for smoothing is the discrete Gaussian filter $K_{G(\sigma)}$ [FP02] which is defined by

$$\left(K_{G(\sigma)}\right)_{r,s} = \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(\frac{r^2+s^2}{2\sigma^2}\right) \tag{32}$$

where σ is the standard deviation of the Gaussian distribution [FP02].

3.2 Layers

We follow [JKRL09] and introduce the different types of layers used in convolutional neural networks. Based on these layers, complex architectures as used for classification in [CMS12] and [KSH12] can be built by stacking multiple layers.

3.2.1 Convolutional Layer

Let layer l be a convolutional layer. Then, the input of layer l comprises $m_1^{(l-1)}$ feature maps from the previous layer, each of size $m_2^{(l-1)} \times m_3^{(l-1)}$. In the case where $l = 1$, the input is a single image I consisting of one or more channels. This way, a convolutional neural network directly accepts raw images as input. The output of layer l consists of $m_1^{(l)}$ feature maps of size $m_2^{(l)} \times m_3^{(l)}$. The i^{th} feature map in layer l, denoted $Y_i^{(l)}$, is computed as

$$Y_i^{(l)} = B_i^{(l)} + \sum_{j=1}^{m_1^{(l-1)}} K_{i,j}^{(l)} * Y_j^{(l-1)} \tag{33}$$

[17] Often, W will be the set $\{0,\dots,255\}$ representing an 8-bit channel. Then, a color image can be represented by an array of size $n_1 \times n_2 \times 3$ assuming three color channels, for example RGB.

[18] As example, consider a gray scale image of size $n_1 \times n_2$. When applying an arbitrary filter of size $2h_1+1 \times 2h_2+1$ to the pixel at location $(1,1)$ the sum of equation (30) includes pixel locations with negative indices. To solve this problem, several approaches can be considered, as for example padding the image in some way or applying the filter only for locations where the operation is defined properly resulting in the output array being smaller than the image.

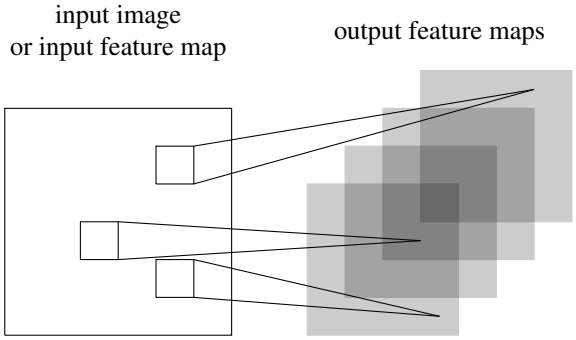


Figure 6: Illustration of a single convolutional layer. If layer l is a convolutional layer, the input image (if $l = 1$) or a feature map of the previous layer is convolved by different filters to yield the output feature maps of layer l.

where $B_i^{(l)}$ is a bias matrix and $K_{i,j}^{(l)}$ is the filter of size $2h_1^{(l)}+1 \times 2h_2^{(l)}+1$ connecting the j^{th} feature map in layer $(l-1)$ with the i^{th} feature map in layer l [LKF10][19]. As mentioned above, $m_2^{(l)}$ and $m_3^{(l)}$ are influenced by border effects. When applying the discrete convolution only in the so called valid region of the input feature maps, that is only for pixels where the sum of equation (30) is defined properly, the output feature maps have size

$$m_2^{(l)} = m_2^{(l-1)} - 2h_1^{(l)} \quad \text{and} \quad m_3^{(l)} = m_3^{(l-1)} - 2h_2^{(l)}. \tag{34}$$

Often the filters used for computing a fixed feature map $Y_i^{(l)}$ are the same, that is $K_{i,j}^{(l)} = K_{i,k}^{(l)}$ for $j \neq k$. In addition, the sum in equation (33) may also run over a subset of the input feature maps.
To relate the convolutional layer and its operation as defined by equation (33) to the multilayer perceptron, we rewrite the above equation. Each feature map $Y_i^{(l)}$ in layer l consists of $m_2^{(l)} \cdot m_3^{(l)}$ units arranged in a two-dimensional array. The unit at position (r,s) computes the output

$$\left(Y_i^{(l)}\right)_{r,s} = \left(B_i^{(l)}\right)_{r,s} + \sum_{j=1}^{m_1^{(l-1)}} \left(K_{i,j}^{(l)} * Y_j^{(l-1)}\right)_{r,s} \tag{35}$$

$$= \left(B_i^{(l)}\right)_{r,s} + \sum_{j=1}^{m_1^{(l-1)}} \sum_{u=-h_1^{(l)}}^{h_1^{(l)}} \sum_{v=-h_2^{(l)}}^{h_2^{(l)}} \left(K_{i,j}^{(l)}\right)_{u,v} \left(Y_j^{(l-1)}\right)_{r+u,s+v}. \tag{36}$$

The trainable weights of the network can be found in the filters $K_{i,j}^{(l)}$ and the bias matrices $B_i^{(l)}$.
As we will see in section 3.2.5, subsampling is used to decrease the effect of noise and distortions. As noted in [CMM+11], subsampling can be done using so called skipping factors $s_1^{(l)}$ and $s_2^{(l)}$. The basic idea is to skip a fixed number of pixels, both in horizontal and in vertical direction, before applying the filter again. With skipping factors as above, the size of the output feature maps is given by

$$m_2^{(l)} = \frac{m_2^{(l-1)} - 2h_1^{(l)}}{s_1^{(l)} + 1} \quad \text{and} \quad m_3^{(l)} = \frac{m_3^{(l-1)} - 2h_2^{(l)}}{s_2^{(l)} + 1}. \tag{37}$$

3.2.2 Non-Linearity Layer

If layer l is a non-linearity layer, its input is given by $m_1^{(l)}$ feature maps and its output comprises again $m_1^{(l)} = m_1^{(l-1)}$ feature maps, each of size $m_2^{(l-1)} \times m_3^{(l-1)}$ such that $m_2^{(l)} = m_2^{(l-1)}$ and $m_3^{(l)} = m_3^{(l-1)}$, given by

$$Y_i^{(l)} = f\left(Y_i^{(l-1)}\right). \tag{38}$$

[19] Note the difference between a feature map $Y_i^{(l)}$ comprising $m_2^{(l)} \cdot m_3^{(l)}$ units arranged in a two-dimensional array and a single unit $y_i^{(l)}$ as used in the multilayer perceptron.

where f is the activation function used in layer l and operates point wise. In [JKRL09] additional gain coefficients are added:

$$Y_i^{(l)} = g_i f\left(Y_i^{(l-1)}\right). \tag{39}$$

A convolutional layer including a non-linearity, with hyperbolic tangent activation functions and gain coefficients is denoted by F_{CSG}[20]. Note that in [JKRL09] this constitutes a single layer whereas we separate the convolutional layer and the non-linearity layer.

3.2.3 Rectification

Let layer l be a rectification layer. Then its input comprises $m_1^{(l-1)}$ feature maps of size $m_2^{(l-1)} \times m_3^{(l-1)}$ and the absolute value for each component of the feature maps is computed:

$$Y_i^{(l)} = \left|Y_i^{(l)}\right| \tag{40}$$

where the absolute value is computed point wise such that the output consists of $m_1^{(l)} = m_1^{(l-1)}$ feature maps unchanged in size[21]. Experiments in [JKRL09] show that rectification plays a central role in achieving good performance.
Although rectification could be included in the non-linearity layer [LKF10], we follow [JKRL09] and add this operation as an independent layer. The rectification layer is denoted by R_{abs}.

3.2.4 Local Contrast Normalization Layer

Let layer l be a contrast normalization layer. The task of a local contrast normalization layer is to enforce local competitiveness between adjacent units within a feature map and units at the same spatial location in different feature maps. We discuss subtractive normalization as well as brightness normalization. An alternative, called divisive normalization, can be found in [JKRL09] or [LKF10]. Given $m_1^{(l-1)}$ feature maps of size $m_2^{(l-1)} \times m_3^{(l-1)}$, the output of layer l comprises $m_1^{(l)} = m_1^{(l-1)}$ feature maps unchanged in size. The subtractive normalization operation computes

$$Y_i^{(l)} = Y_i^{(l-1)} - \sum_{j=1}^{m^{(l-1)}} K_{G(\sigma)} * Y_j^{(l-1)} \tag{41}$$

where $K_{G(\sigma)}$ is the Gaussian filter from equation (32).
In [KSH12] an alternative local normalization scheme called brightness normalization is proposed to be used in combination with rectified linear units. Then the output of layer l is given by

$$\left(Y_i^{(l)}\right)_{r,s} = \frac{\left(Y_i^{(l-1)}\right)_{r,s}}{\left(\kappa + \mu \sum_{j=1}^{m_1^{(l-1)}} \left(Y_j^{(l-1)}\right)_{r,s}^2\right)^{\mu}} \tag{42}$$

where κ, λ, μ are hyperparameters which can be set using a validation set [KSH12]. The sum in equation (42) may also run over a subset of to the feature maps in layer $(l-1)$. Local contrast normalization layers are denoted N_S and N_B, respectively.

3.2.5 Feature Pooling and Subsampling Layer

The motivation of subsampling the feature maps obtained by previous layers is robustness to noise and distortions [JKRL09]. Reducing the resolution can be accomplished in different ways. In [JKRL09] and

[20] C for convolutional layer, S for sigmoid/hyperbolic tangent activation functions and G for gain coefficients. In [JKRL09] the filter size is added as subscript such that $F_{\text{CSG}}^{7\times7}$ denotes the usage of 7×7 filters. Additionally, the number of used filters is added as follows: $32F_{\text{CSG}}^{7\times7}$. We omit the number of filters as we assume full connectivity such that the number of filters is given by $m_1^{(l)} \cdot m_1^{(l-1)}$.
[21] Note that equation (40) can easily be applied to fully-connected layers as introduced in section 3.2.6, as well.

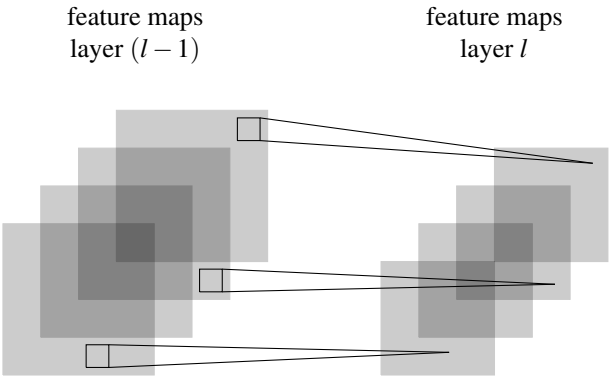


Figure 7: Illustration of a pooling and subsampling layer. If layer l is a pooling and subsampling layer and given $m_1^{(l-1)} = 4$ feature maps of the previous layer, all feature maps are pooled and subsampled individually. Each unit in one of the $m_1^{(l)} = 4$ output feature maps represents the average or the maximum within a fixed window of the corresponding feature map in layer $(l-1)$.

[LKF10] this is combined with pooling and done in a separate layer, while in the traditional convolutional neural networks, subsampling is done by applying skipping factors.

Let l be a pooling layer. Its output comprises $m_1^{(l)} = m_1^{(l-1)}$ feature maps of reduced size. In general, pooling operates by placing windows at non-overlapping positions in each feature map and keeping one value per window such that the feature maps are subsampled. We distinguish two types of pooling:

Average pooling When using a boxcar filter[22], the operation is called average pooling and the layer denoted by P_A.

Max pooling For max pooling, the maximum value of each window is taken. The layer is denoted by P_M.

As discussed in [SMB10], max pooling is used to get faster convergence during training. Both average and max pooling can also be applied using overlapping windows of size $2p \times 2p$ which are placed q units apart. Then the windows overlap if $q < p$. This is found to reduce the chance of overfitting the training set [KSH12].

3.2.6 Fully Connected Layer

Let layer l be a fully connected layer. If layer $(l-1)$ is a fully connected layer, as well, we may apply equation (2). Otherwise, layer l expects $m_1^{(l-1)}$ feature maps of size $m_2^{(l-1)} \times m_3^{(l-1)}$ as input and the i^{th} unit in layer l computes:

$$y_i^{(l)} = f\left(z_i^{(l)}\right) \quad \text{with} \quad z_i^{(l)} = \sum_{j=1}^{m_1^{(l-1)}} \sum_{r=1}^{m_2^{(l-1)}} \sum_{s=1}^{m_3^{(l-1)}} w_{i,j,r,s}^{(l)} \left(Y_j^{(l-1)}\right)_{r,s}. \tag{43}$$

where $w_{i,j,r,s}^{(l)}$ denotes the weight connecting the unit at position (r,s) in the j^{th} feature map of layer $(l-1)$ and the i^{th} unit in layer l. In practice, convolutional layers are used to learn a feature hierarchy and one or more fully connected layers are used for classification purposes based on the computed features [LBD+89, LKF10]. Note that a fully-connected layer already includes the non-linearities while for a convolutional layer the non-linearities are separated in their own layer.

3.3 Architectures

We discuss both the traditional convolutional neural network as proposed in [LBD+89] as well as a modern variant as used in [KSH12].

3.3.1 Traditional Convolutional Neural Network

In [JKRL09], the basic building block of traditional neural networks is $F_\mathrm{CSG} - P_\mathrm{A}$, while in [LBD+89], the subsampling is accomplished within the convolutional layers and there are no gain coefficients used. In

[22] Using the notation as used in section 3.1, the boxcar filter K_B of size $2h_1+1 \times 2h_2+1$ is given by $(K_B)_{r,s} = \frac{1}{(2h_1+1)(2h_2+1)}$.

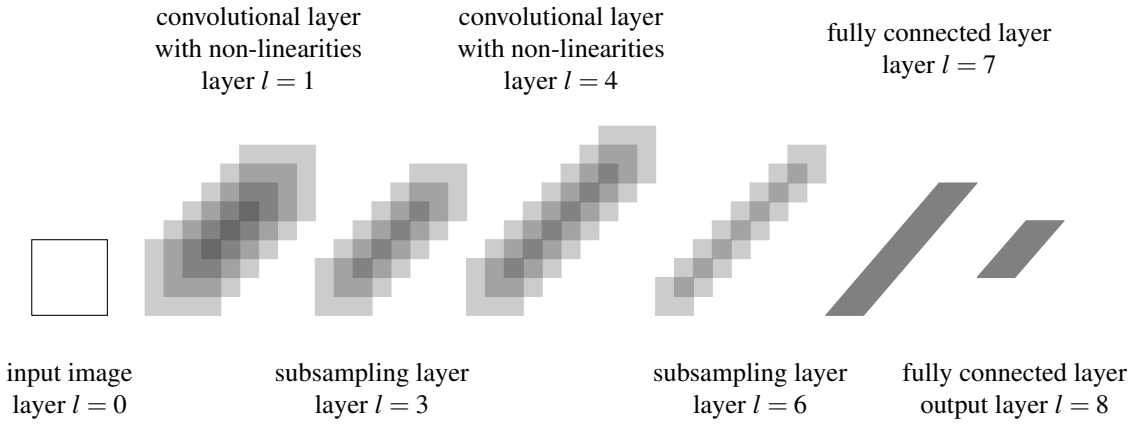


Figure 8: The architecture of the original convolutional neural network, as introduced in [LBD+89], alternates between convolutional layers including hyperbolic tangent non-linearities and subsampling layers. In this illustration, the convolutional layers already include non-linearities and, thus, a convolutional layer actually represents two layers. The feature maps of the final subsampling layer are then fed into the actual classifier consisting of an arbitrary number of fully connected layers. The output layer usually uses softmax activation functions.

general, the unique characteristic of traditional convolutional neural networks lies in the hyperbolic tangent non-linearities and the weight sharing [LBD+89]. This is illustrated in figure 8 where the non-linearities are included within the convolutional layers.

3.3.2 Modern Convolutional Neural Networks

As example of a modern convolutional neural network we explore the architecture used in [KSH12] which gives excellent performance on the ImageNet Dataset [ZF13]. The architecture comprises five convolutional layers each followed by a rectified linear unit non-linearity layer, brightness normalization and overlapping pooling. Classification is done using three additional fully-connected layers. To avoid overfitting, [KSH12] uses dropout as regularization technique. Such a network can be specified by $F_{\text{CR}} - N_B - P$ where F_{CR} denotes a convolutional layer followed by a non-linearity layer with rectified linear units. Details can be found in [KSH12].
In [CMS12] the authors combine several deep convolutional neural networks which have a similar architecture as described above and average their classification/prediction result. This architecture is referred to as multi-column deep convolutional neural network.

4 Understanding Convolutional Neural Networks

Although convolutional neural networks have been used with success for a variety of computer vision tasks, their internal operation is not well understood. While backprojection of feature activations from the first convolutional layer is possible, subsequent pooling and rectification layers hinder us from understanding higher layers as well. As stated in [ZF13], this is highly unsatisfactory when aiming to improve convolutional neural networks. Thus, in [ZF13], a visualization technique is proposed which allows us to visualize the activations from higher layers. This technique is based on an additional model for unsupervised learning of feature hierarchies: the deconvolutional neural network as introduced in [ZKTF10].

4.1 Deconvolutional Neural Networks

Similar to convolutional neural networks, deconvolutional neural networks are based upon the idea of generating feature hierarchies by convolving the input image by a set of filters at each layer [ZKTF10]. However, deconvolutional neural networks are unsupervised by definition. In addition, deconvolutional neural networks are based on a top-down approach. This means, the goal is to reconstruct the network input from its activations and filters [ZKTF10].

4.1.1 Deconvolutional Layer

Let layer l be a deconvolutional layer. The input is composed of $m_1^{(l-1)}$ feature maps of size $m_2^{(l-1)} \times m_3^{(l-1)}$. Each such feature map $Y_i^{(l-1)}$ is represented as sum over $m_1^{(l)}$ feature maps convolved with filters $K_{j,i}^{(l)}$:

$$\sum_{j=1}^{m_1^{(l)}} K_{j,i}^{(l)} * Y_j^{(l)} = Y_i^{(l-1)}. \tag{44}$$

As with an auto-encoder, it is easy for the layer to learn the identity, if there are enough degrees of freedom. Therefore, [ZKTF10] introduces a sparsity constraint for the feature maps $Y_j^{(l)}$, and the error measure for training layer l is given by

$$E^{(l)}(w) = \sum_{i=1}^{m_1^{(l-1)}} \left\| \sum_{j=1}^{m_1^{(l)}} K_{j,i}^{(l)} * Y_j^{(l)} - Y_i^{(l-1)} \right\|_2^2 + \sum_{i=1}^{m_1^{(l)}} \left\| Y_i^{(l)} \right\|_p^p \tag{45}$$

where $\|\cdot\|_p$ is the vectorized p-norm and can be interpreted as L_p-regularization as discussed in section 2.5.1. The difference between a convolutional layer and a deconvolutional layer is illustrated in figure 9. Note that the error measure $E^{(l)}$ is specific for layer l. This implies that a deconvolutional neural network with multiple deconvolutional layers is trained layer-wise.

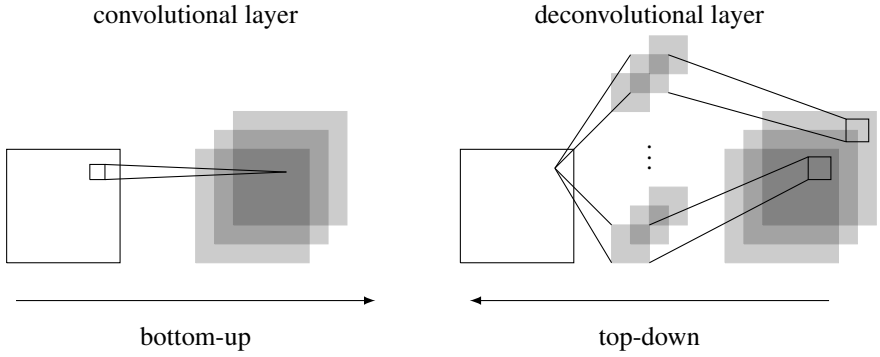


Figure 9: An illustration of the difference between the bottom-up approach of convolutional layers and the top-down approach of deconvolutional layers.

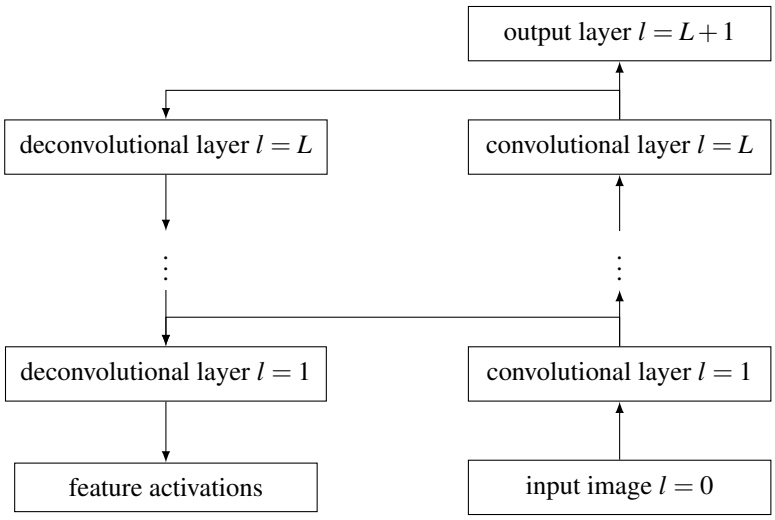


Figure 10: After each convolutional layer, the feature activations of the previous layer are reconstructed using an attached deconvolutional layer. For $l > 1$ the process of reconstruction is iterated until the feature activations are backprojected onto the image plane.

4.1.2 Unsupervised Training

Similar to unsupervised training discussed in section 2.4, training is performed layer-wise. Therefore, equation (45) is optimized by alternately optimizing with respect to the feature maps $Y_i^{(l)}$ given the filters $K_{j,i}^{(l)}$ and the feature maps $Y_i^{(l-1)}$ of the previous layer and with respect to the filters $K_{j,i}^{(l)}$ [ZKTF10]. Here, the optimization with respect to the feature maps $Y_i^{(l)}$ causes some problems. For example when using $p = 1$, the optimization problem is poorly conditioned [ZKTF10] and therefore usual gradient descent optimization fails. An alternative optimization scheme is discussed in detail in [ZKTF10], however, as we do not need to train deconvolutional neural networks, this is left to the reader.

4.2 Visualizing Convolutional Neural Networks

To visualize and understand the internal operations of a convolutional neural network, a single deconvolutional layer is attached to each convolutional layer. Given input feature maps for layer l, the output feature maps $Y_i^{(l)}$ are fed back into the corresponding deconvolutional layer at level l. The deconvolutional layer reconstructs the feature maps $Y_i^{(l-1)}$ that gave rise to the activations in layer l [ZF13]. This process is iterated until layer $l = 0$ is reached resulting in the activations of layer l being backprojected onto the image plane. The general idea is illustrated in figure 10. Note that the deconvolutional layers do not need to be trained as the filters are already given by the trained convolutional layers and merely have to be transposed[23]. More complex convolutional neural networks may include non-linearity layers, rectification layers as well as pooling layers. While we assume the used non-linearities to be invertible , the use of rectification layers and pooling layers cause some problems.

4.2.1 Pooling Layers

Let layer l be a max pooling layer, then the operation of layer l is not invertible. We need to remember which positions within the input feature map $Y_i^{(l)}$ gave rise to the maximum value to get an approximate inverse [ZF13]. Therefore, as discussed in [ZF13], switch variables are introduced.

[23] Given a feature map $Y_i^{(l)} = K_{i,j}^{(l)} * Y_j^{(l-1)}$ (here we omit the sum of equation (33) for simplicity) and using the transposed filter $\left(K_{j,i}^{(l)}\right)^T$ gives us: $Y_j^{(l-1)} = \left(K_{i,j}^{(l)}\right)^T * Y_i^{(l)}$.

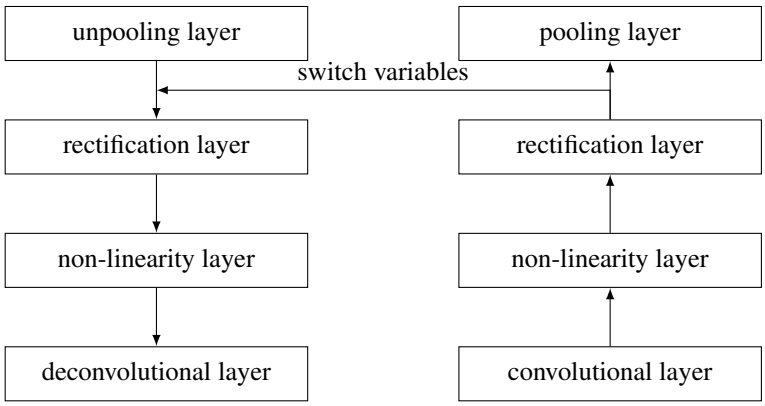


Figure 11: While the approach described in section 4.2 can easily be applied to convolutional neural networks including non-linearity layers, the usage of pooling and rectification layers imposes some problems. The max pooling operation is not invertible. Therefore, for each unit in the pooling layer, we remember the position in the corresponding feature map which gave rise to the unit's output value. To accomplish this, so called switch variables are introduced [ZF13]. Rectification layers can simply be inverted by prepending a rectification layer to the deconvolutional layer.

4.2.2 Rectification Layers

The convolutional layer may use rectification layers to obtain positive feature maps after each non-linearity layer. To cope with this, a rectification layer is added to each deconvolutional layer to obtain positive reconstructions of the feature maps, as well [ZF13]. Both the incorporation of pooling layers and rectification layers is illustrated in figure 11.

4.3 Convolutional Neural Network Visualization

The above visualization technique can be used to discuss several aspects of convolutional neural networks. We follow the discussion in [ZF13] which refers to the architecture described in section 3.3.2.

4.3.1 Filters and Features

Backprojecting the feature activations allows close analysis of the hierarchical nature of the features within the convolutional neural network. Figure 12, taken from [ZF13], shows the activations for three layers with corresponding input images. While the first and second layer comprise filters for edge and corner detection, the filters tend to get more complex and abstract with higher layers. For example when considering layer 3, the feature activations reflect specific structures within the images: the patterns used in layer 3, row 1, column 1; human contours in layer 3 row3, column 3. Higher levels show strong invariances to translation and rotation [ZF13]. Such transformations usually have high impact on low-level features. In addition, as stated in [ZF13], it is important to train the convolutional neural network until convergence as the higher levels usually need more time to converge.

4.3.2 Architecture Evaluation

The visualization of the feature activations across the convolutional layers allows to evaluate the effect of filter size as well as filter placement. For example, by analyzing the feature activations of the first and second layer, the authors of [ZF13] observed that the first layer does only capture high frequency and low frequency information and the feature activations of the second layer show aliasing artifacts. By adapting the filter size of the first layer and the skipping factor used within the second layer, performance could be improved. In addition, the visualization shows the advantage of deep architectures as higher layers are able to learn more complex features invariant to low-level distortions and translations [ZF13].

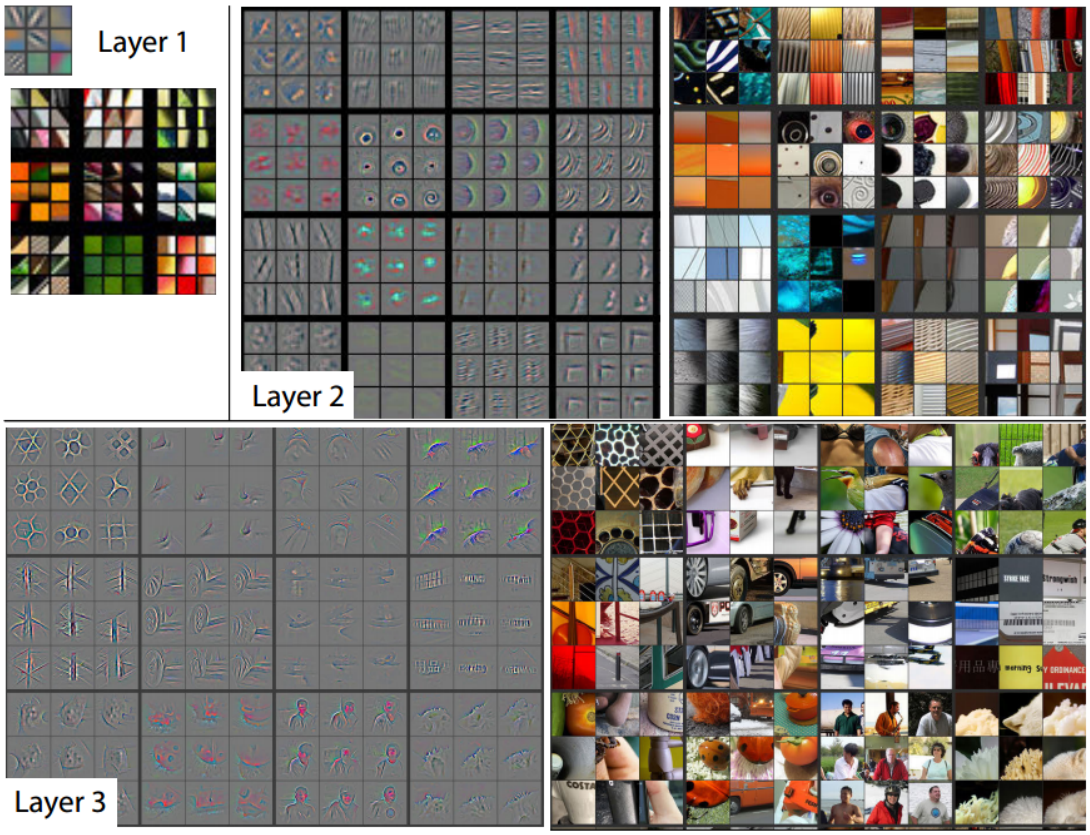


Figure 12: Taken from [ZF13], this figure shows a selection of features across several layers of a fully trained convolutional network using the visualization technique discussed in section 4.

5 Conclusion

In the course of this paper we discussed the basic notions of both neural networks in general and the multi-layer perceptron in particular. With deep learning in mind, we introduced supervised training using gradient descent and error backropagation as well as unsupervised training using auto encoders. We concluded the section with a brief discussion of regularization methods including dropout [HSK+12] and unsupervised pre-training.

We introduced convolutional neural networks by discussing the different types of layers used in recent implementations: the convolutional layer; the non-linearity layer; the rectification layer; the local contrast normalization layer; and the pooling and subsampling layer. Based on these basic building blocks, we discussed the traditional convolutional neural networks [LBD+89] as well as a modern variant as used in [KSH12].

Despite of their excellent performance [KSH12, CMS12], the internal operation of convolutional neural networks is not well understood [ZF13]. To get deeper insight into their internal working, we followed [ZF13] and discussed a visualization technique allowing to backproject the feature activations of higher layers. This allows to further evaluate and improve recent architectures as for example the architecture used in [KSH12].

Nevertheless, convolutional neural networks and deep learning in general is an active area of research. Although the difficulty of deep learning seems to be understood [Ben09, GB10, EMB+09], learning feature hierarchies is considered very hard [Ben09]. Here, the possibility of unsupervised pre-training had a huge impact and allows to train deep architectures in reasonable time [Ben09, EBC+10]. Nonetheless, the reason for the good performance of deep neural networks is still not answered fully.

References

[Ben09] Y. Bengio. Learning deep architectures for AI. *Foundations and Trends in Machine Learning*, (1):1–127, 2009.

[Bis92] C. Bishop. Exact calculation of the hessian matrix for the multilayer perceptron. *Neural Computation*, 4(4):494–501, 1992.

[Bis95] C. Bishop. *Neural Networks for Pattern Recognition*. Clarendon Press, Oxford, 1995.

[Bis06] C. Bishop. *Pattern Recognition and Machine Learning*. Springer Verlag, New York, 2006.

[BL89] S. Becker and Y. LeCun. Improving the convergence of back-propagation learning with second-order methods. In *Connectionist Models Summer School*, pages 29–37, 1989.

[BL07] Y. Bengio and Y. LeCun. Scaling learning algorithms towards AI. In *Large Scale Kernel Machines*. MIT Press, 2007.

[CMM+11] D. C. Cireşan, U. Meier, J. Masci, L. M. Gambardella, and J. Schmidhuber. Flexible, high performance convolutional neural networks for image classification. In *Artificial Intelligence, International Joint Conference*, pages 1237–1242, 2011.

[CMS12] D. C. Ciresan, U. Meier, and J. Schmidhuber. Multi-column deep neural networks for image classification. *Computing Research Repository*, abs/1202.2745, 2012.

[DHS01] R. Duda, P. Hart, and D. Stork. *Pattern Classification*. Wiley-Interscience Publication, New York, 2001.

[EBC+10] D. Erhan, Y. Bengio, A. Courville, P.-A. Manzagol, P. Vincent, and S. Bengio. Why does unsupervised pre-training help deep learning? *Journal of Machine Learning Research*, 11:625–660, 2010.

[EMB+09] D. Erhan, P.-A. Manzagol, Y. Bengio, S. Bengio, and P. Vincent. The difficulty of training deep architectures and the effect of unsupervised pre-training. In *Artificial Intelligence and Statistics, International Conference on*, pages 153–160, 2009.

[FP02] D. Forsyth and J. Ponce. *Computer Vision: A Modern Approach*. Prentice Hall Professional Technical Reference, New Jersey, 2002.

[GB10] X. Glorot and Y. Bengio. Understanding the difficulty of training deep feedforward neural networks. In *Artificial Intelligence and Statistics, International Conference on*, pages 249–256, 2010.

[GBB11] X. Glorot, A. Bordes, and Y. Bengio. Deep sparse rectifier neural networks. In *Artificial Intelligence and Statistics, International Conference on*, pages 315–323, 2011.

[GMW81] P. Gill, W. Murray, and M. Wright. *Practical optimization*. Academic Press, London, 1981.

[Hay05] S. Haykin. *Neural Networks A Comprehensive Foundation*. Pearson Education, New Delhi, 2005.

[HO06] G. E. Hinton and S. Osindero. A fast learning algorithm for deep belief nets. *Neural Computation*, 18(7):1527–1554, 2006.

[HSK+12] G. E. Hinton, N. Srivastava, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov. Improving neural networks by preventing co-adaptation of feature detectors. *Computing Research Repository*, abs/1207.0580, 2012.

[HSW89] K. Hornik, M. Stinchcombe, and H. White. Multilayer feedforward networks are universal approximators. *Neural Networks*, 2(5):359–366, 1989.

[JKRL09] K. Jarrett, K. Kavukcuogl, M. Ranzato, and Y. LeCun. What is the best multi-stage architecture for object recognition? In *Computer Vision, International Conference on*, pages 2146–2153, 2009.

[KRL10] K. Kavukcuoglu, M.'A. Ranzato, and Y. LeCun. Fast inference in sparse coding algorithms with applications to object recognition. *Computing Research Repository*, abs/1010.3467, 2010.

[KSH12] A. Krizhevsky, I. Sutskever, and G. E. Hinton. ImageNet classification with deep convolutional neural networks. In *Advances in Neural Information Processing Systems*, pages 1097–1105, 2012.

[LBBH98] Y. LeCun, L. Buttou, Y. Bengio, and P. Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86:2278–2324, 1998.

[LBD+89] Y. LeCun, B. Boser, J. S. Denker, D. Henderson, R. E. Howard, W. Hubbard, and L. D. Jackel. Backpropagation applied to handwritten zip code recognition. *Neural Computation*, 1(4):541–551, 1989.

[LBLL09] H. Larochelle, Y. Bengio, J. Louradour, and P. Lamblin. Exploring strategies for training deep neural networks. *Journal of Machine Learning Research*, 10:1–40, 2009.

[LeC89] Y. LeCun. Generalization and network design strategies. In *Connectionism in Perspective*, 1989.

[LKF10] Y. LeCun, K. Kavukvuoglu, and C. Farabet. Convolutional networks and applications in vision. In *Circuits and Systems, International Symposium on*, pages 253–256, 2010.

[NH92] S. J. Nowlan and G. E. Hinton. Simplifying neural networks by soft weight-sharing. *Neural Computation*, 4(4):473–493, 1992.

[RHW86] D. E. Rumelhart, G. E. Hinton, and R. J. Williams. Parallel distributed processing: Explorations in the microstructure of cognition. chapter Learning Representations by Back-Propagating Errors, pages 318–362. MIT Press, Cambridge, 1986.

[Ros58] F. Rosenblatt. The perceptron: A probabilistic model for information storage and organization in the brain. *Psychological Review*, 65, 1958.

[SMB10] D. Scherer, A. Müller, and S. Behnke. Evaluation of pooling operations in convolutional architectures for object recognition. In *Artificial Neural Networks, International Conference on*, pages 92–101, 2010.

[SSP03] P. Y. Simard, D. Steinkraus, and J. C. Platt. Best practices for convolutional neural networks pplied to visual document analysis. In *Document Analysis and Recognition, International Conference on*, 2003.

[ZF13] M. D. Zeiler and R. Fergus. Visualizing and understanding convolutional networks. *Computing Research Repository*, abs/1311.2901, 2013.

[ZKTF10] M. D. Zeiler, D. Krishnan, G. W. Taylor, and R. Fergus. Deconvolutional networks. In *Computer Vision and Pattern Recognition, Conference on*, pages 2528–2535, 2010.