Rheinisch-Westfälische Technische Hochschule Aachen
Computer science chair 6
Prof. Dr.-Ing. Hermann Ney

Selected Topics in Human Language Technology and Pattern Recognition within the Winter Term 2016/2017

Convolutional Networks

Mike Lorang

Matriculation number 348496

Date of the presentation: 19^{th} January 2017

Supervisor: Harald Hanselmann

Contents

List of Tables

List of Figures

1 Introduction

Convolutional Networks which are also known as Convolutional Neural Networks (CNNs) are a kind of deep learning network which can process data with a clear grid-structured topology like an image.
The idea of CNNs came quite early. In 1980 Fukushima tried to reproduce how the brain processes the vision in the visual cortex in code, based on the experiments of Hubel and Weisel. With Neurocognitron Fukushima built a layered architecture where the neurons look at a small region of the input [5]. Because back-propagation was not invented at that time, Fukushima could not train his network efficiently. In 1998 Yann LeCun and Leon Bottou took roughly the same architecture as Fukushima and trained their neural network with backward propagation.
In the 1990s, CNNs were used to check the handwriting on checks. More than 10% of the checks were read by CNNs by the end of the 1990s. Thus CNNs were one of the first neural networks to be used in a commercial way [6].
The modern Convolutional Network is based on that research paper by Leon Bottou and Yann LeCunn.
Nowadays CNNs are used in many more domains like computer vision, object classification, video analysis, drug discovery, playing the game Go and many more.

In Section 2 it is described why CNNs are needed, and why MPLs are not suitable for certain tasks. Then in Section 4 the general architecture of CNNs is explained and why it performs better than Multilayer Perceptrons (MLP's). Afterwards in Section 5 the different layers will be elucidated in detail. Then in Section 6 the difference between training an MLP and training an CNN is elaborated. Furthermore in Section 7 some options for optimization are outlined. Afterwards in Section 8 the most famous and successful CNN architectures are presented and explained.

2 Motivation

Several problems like image recognition are very computational intense on an MLP due to the unmanageable amount of parameters. For example on a 512x512x3 image (512 height, 512 width, 3 color channels for Red Green and Blue) a single neuron in the first fully connected layer would have $512 \cdot 512 \cdot 3 = 786432$ weights to manage. If every hidden neuron has different weights, the amount of parameters would add up quickly and become unmanageable. If the first hidden layer consists of 100 neurons, the first layer only had to manage 7864320 parameter. Thus a more efficient method is needed in order to process grid structured data efficiently .
Another advantage that CNNs have to MLPs is that they can handle inputs of varying size.

3 Basic terminology

Parameters: The values that are needed to be saved are referred as parameters.
Weight matrix: A matrix that contains parameters. The weight matrix describes with which value the output of the previous layer is multiplied before being set as input of the

current layer.
Multilayer perceptron (MLP): Is an artificial neural network which consist of three or more layer. It has one input layer, one or more hidden layers and one output layer.
Training: Adjusting the entries of the weight matrices such that the neural networks result comes closer to the desired result.
Overfitting: A modeling error which occurs when a function is too closely fit to a limited set of data points. Overfitting typically occurs when the model is very complex but there is only few training data.

4 General architecture

In a Convolutional Neural Network the neurons are arranged in three dimensions (width, height, depth) whereas in an MLP the neurons are arranged in only 2 dimensions. Each layer of the CNN transforms the 3D input volume in a 3D output volume. This is illustrated in Figure 1.

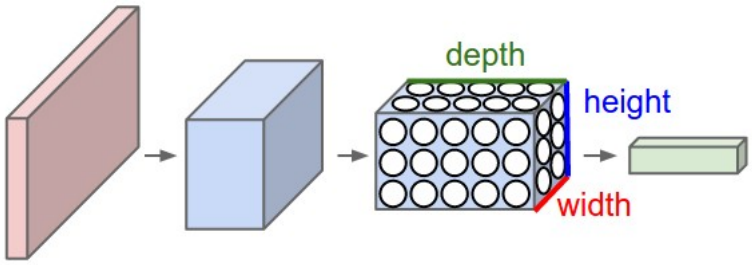


Figure 1: Picture from [8]

Each input unit of the CNN is looking for a certain pattern, in the input. The closer input gets to that pattern, the higher the output of the activation function rises. For example, the input is a picture and the pattern to detect is an horizontal edge. If a neuron detects a horizontal edge in the picture, the neurons activation function rises. By stacking these operations more complex patterns than edges can be detected.
The convolution operation plays a very important role and provides the core to increase the memory efficiency. Convolution provides three basic properties: sparse connectivity, parameter sharing and equivariant representations.

4.1 The Convolution Operation

The convolution operation is a mathematical operation which operates on two functions (e.g. f and g) and produces a third function (e.g. h). The convolution is defined as the integral of the product of two functions, after one is reversed and shifted. The convolution operation is usually denoted with a asterisk.

$$(f * g)(t) = \int_{-\infty}^{\infty} f(a)g(t-a)da \tag{1}$$

The symbol t does not need to represent the time, but if it does the integral represents a weighted average of the function $f(a)$ at the moment t and g represents the weight function.
In CNN terminology the function f is often referred as the input, the function g as the

kernel or filter and the output of the convolution operation is called the feature map. The input of a CNN is usually an array which contains discrete values, which means the integral has to be transformed in a sum to match the requirements.

$$(f * g)(t) = \sum_{a=-\infty}^{\infty} f(a)g(t-a) \tag{2}$$

The convolution operation is commutative:

$$\sum_{a=-\infty}^{\infty} f(a)g(t-a) = \sum_{a=-\infty}^{\infty} f(t-a)g(a) \tag{3}$$

This can be useful when writing proofs but does not affect the efficiency. The input is often not one-dimensional but a multidimensional array of data. Thus the kernel needs to be the same dimension as the input and contains parameters adapted to the use-case. These multidimensional arrays are also called tensors.
If we want to apply the convolution operation to a two-dimensional image I, we also need an two-dimensional kernel K. This results in the following equation:

$$S(i,j) = (I * K)(i,j) = \sum_m \sum_n I(m,n)K(i-m, j-n) \tag{4}$$

Each element of the input and kernel array needs to be explicitly stored and other entries are assumed to be zero. This way it is possible to implement the infinite summation as a finite summation of these array elements.
There is a operation called cross-correlation, which has apart from not being commutative, the same properties as the convolution operation. Cross-correlation is as efficient as convolution, but easier to implement. That is why it is often found in machine learning libraries under the name convolution. The cross-correlation equation is the following:

$$S(i,j) = (I * K)(i,j) = \sum_m \sum_n I(m+i, n+j)K(m,n) \tag{5}$$

Apart from being memory efficient, the convolution is good at learning features. During the training phase, the parameters of the kernels are adjusted in order to be able to extract the most important features. In the Figure 2 is an example of cross-correlation. The kernel slides over the input and performs a matrix multiplication. The boxes and arrows indicate the starting position of the kernel. The kernel is applied to all possible positions, which means in every position where it lies entirely in the image. Thus the output of the convolution function is smaller than the input, if the kernel-size is greater than one.

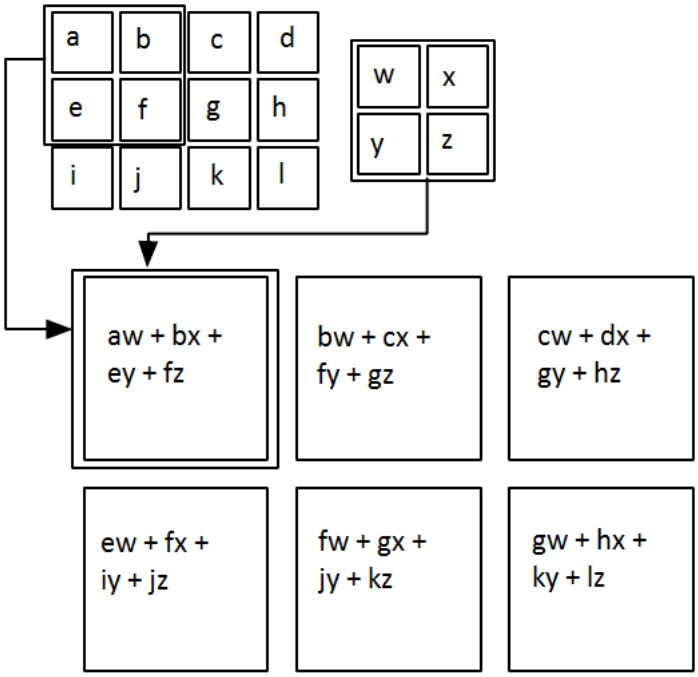


Figure 2: [6]

4.2 Sparse connectivity

The next subsection is based on section 9.2 from [6]. To be able to handle large amounts of input data, for example a large image, it is impractical to connect every neuron of the current layer to every neurons from the previous layer like in MLPs. Full matrix multiplication should be avoided in order to save memory. This is realized by the sparse connectivity which works as following. Each input neuron is responsible for a certain area of the input, the local receptive field. The local receptive field always includes the full depth of the input. The size of the local receptive field is always equal to the kernel size. In Figure 3 on the left, neurons from the first hidden layer have a local receptive field of 3x3. The hidden units gets their input from nine units from the input layer. In a MLP, a unit from the first hidden layer would get their input from all 36 input units. In Figure 4, sparse connectivity is illustrated in one dimension. [10]

This way not every input unit is connected to every unit of the next layer like in a multilayer perceptron, However if the kernel size is the same as the input size, the full time consuming matrix multiplication is still needed. Thus the kernel size needs to be smaller than the input size to gain efficiency. For example if the input is a large image, instead of looking at the whole picture at once (which could be a few million pixels), small regions (some hundred pixels) of the picture will be looked at one after another to detect small meaningful features like edges. This way much fewer parameters are need to be stored. Let m be the number of inputs and n be the number of outputs. Then $m \cdot n$ parameters are need to be stored. If we limit the number of inputs with the kernel to k. Only $k \cdot n$ parameters need to be stored for each input. Usually k is several orders of magnitude smaller than m which results in a big gain of efficiency.

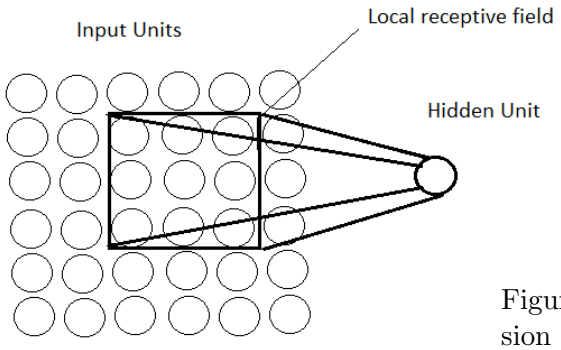


Figure 3: Sparse connectivity in two dimensions

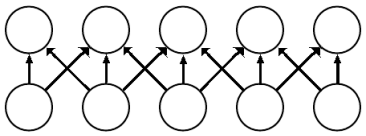


Figure 4: Sparse connectivity in one dimension

4.3 Parameter sharing

During one forward pass in an MLP, each element of the weight matrix is only used once when it gets multiplied by the input. Apart from that multiplication it is never used again. In CNNs the parameters stay the same for every kernel, which is applied multiple times during one forward pass. Another word for parameter sharing is tied weights because the weights applied to the input does not change no matter where it is applied. Rather than learning different parameters for every part of the input, the CNN only needs one set of parameters for every part of the input. This reduces the memory requirement.
For example the input size is 6x6 and a kernel has a size of 3x3 then there are $4 \cdot 4 = 16$ possible positions for the kernel. Without parameter sharing, $16 * 9 = 144$ parameters needed to be stored. With parameter sharing, only 9 need to be stored.
While using parameter sharing, the memory efficiency and the statistical run-time efficiency increases. To visualize how shared weights work see Figure 5. The red arrows indicate where a particular parameter is used. In the top picture we see that the same parameter is used for five connections due to parameter sharing. In the bottom picture, every parameter is used exactly once.

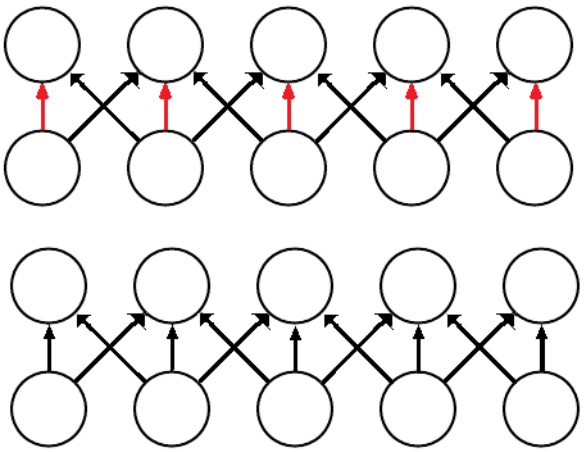


Figure 5: Picture from [6]

4.4 Equivariance to translation

Equivariance to translation means that if the input of one function f changes, the output changes the same way. Mathematically this means that if the function f is equivariant to the function g, $f(g(x)) = g(f(x))$. Convolution has this property due to the parameter sharing.
If we process time series data, the output shows exactly when an event appeared on the input, because of the property that the output changes the same way as the input. The result will be a time line that shows different events over time. If we move an event later in time in the input, the exact same representation of it will appear in the output just later in time. In Figure 6 there are two exact same events. The event 2 is later in time than event 1. The representation of the output of both events is exactly the same but event 2 appears later in the output than event 1.

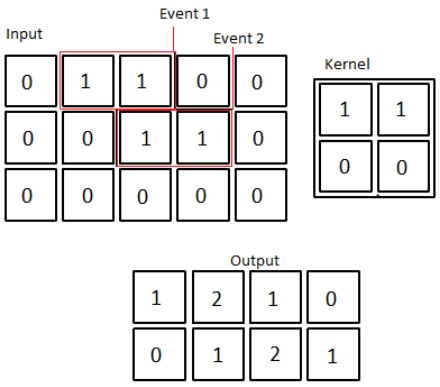


Figure 6: Equivariance to translation

This can be really useful when processing images. Usually the first convolution layer detects edges. Edges can be found in many parts of the image, thus it makes sense to share

the parameters over the entire picture and find out where the edges are getting detected in the input.

5 Layers used in a Convolutional Network

A CNN is usually build by stacking the following three layers: A convolution layer, a detector layer and a pooling layer (see Figure 7)
There are often a one to three fully connected layers at the end of the CNN like in the AlexNet which is described in Section 8.1. The fully connected layers work the same way as in an MLP.

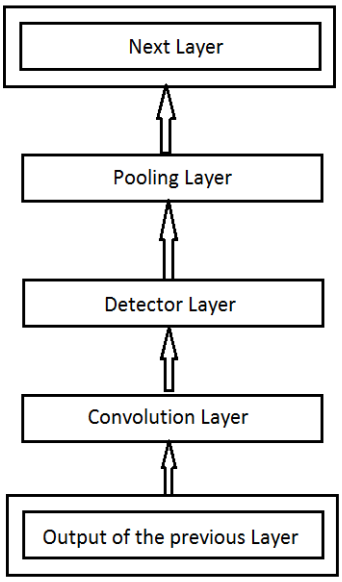


Figure 7: The Convolutional Network is often build stacking up these three layers.[6]

The Convolution Layer performs several convolutions with different kernels in parallel to produce a set of linear activation. In the detector layer, the linear activation runs through a nonlinear activation function. Finally the pooling layer is used to modify the output of the layer further.

5.1 Convolution Layer

In the convolution layer the convolution operation is applied. It is the most computational expensive layer due to the matrix multiplication (see Section 4.1). Moreover there are variations of the convolution operation. These variations are used to maintain certain properties or to gain a bit of efficiency. Two of these variations are described in the two following subsections, the stride and zero padding. [12]

5.1.1 Stride

To save some computational cost, there is a possibility to skip certain convolution operations. In order to do that, the kernel is applied to every s-th position. We refer to s as

the stride. The down-sampled convolution is defined as follow.

$$S(i,j,s) = (I * K)(i,j) = \sum_m \sum_n I((i \cdot s) + m, (j \cdot s) + n)K(m,n) \tag{6}$$

In the Figure 8 there is an example of the convolution operation but this time with a stride of $s = 2$ applied. The kernel now only takes every second place of the input in each direction. Making the stride the same size as the kernel width might lead to missing features. If certain features are only partly visible on the border they will not be detected. The higher the stride the smaller the output gets. The stride samples the input further down. In the Equation (6) the stride is the same for both axes but it is also possible to define a different stride for each axis. For example s = 1 for the height axis and s = 2 for the width axis.

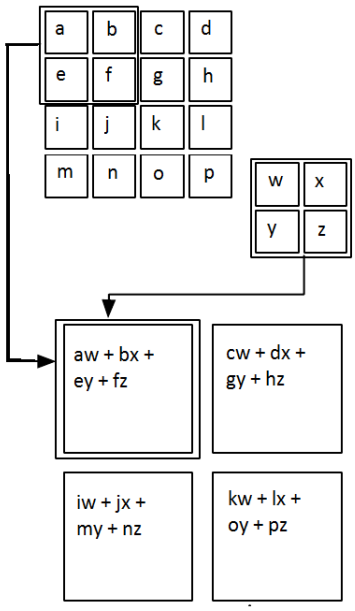


Figure 8: Stride $s = 2$

5.1.2 Zero padding

Zero padding is used to expand the input and make it wider. In this paper when it is said that zero padding by size one is applied, it is meant that we insert one film of zeros all around the border of the input. If a 3x3 input gets zero padded by size one, it has size 5x5 after the padding. In Figure 9 there is a 3x3 input that gets zero padded by size one. After the zero padding is has size 5x5. The Green circles represent input which is zero padded and the black circles represent the original input.

If the kernel-size is greater than one, the number of positions it can take in the input is smaller than the number of input values. This means than if the kernel is greater than one, the output is smaller than the input. The larger the kernel the smaller the output of the convolution. If we have a square input of (N x N), a kernel with width (m x m) and stride s the convolution layer output will be of size ($\frac{N-m}{s} + 1$) x ($\frac{N-m}{s} + 1$). That means

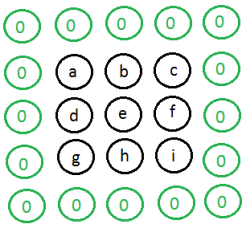


Figure 9: Zero padding

if the input would be a 32x32 picture, the kernel would have size (5x5) (with stride 1), the output of the convolution operation without zero padding would have size 28x28 pixels. Zero padding can prevent the input from shrinking by making the input bigger. It is also possible to control the kernel width and the output size independently. If zero padding by 2 is applied, we get a 36x36 input out of the 32x32 input. This way the output will be the same size as the input without zero padding (32x32). Without zero padding it has to be chosen between a small kernel or rapidly shrinking the spatial extent of the network.

5.2 Detector Layer

The detector layer contains the activation function and takes the linear activation of the convolution function and applies a nonlinear function to it such as the Sigmoid function, the tanh function or the max(0,x) function from the rectified linear units.

5.2.1 ReLU Units

This section is based on [9]. One way to model a neuron's output f as a function of it's input x was $f(x) = tanh(x)$ or the Sigmoid function $f(x) = \frac{1}{1+e^{-x}}$. If we train the CNN with gradient descent like explained in Section 6 the training will take much longer with these two saturating nonlinear functions as with the f(x)=max(0,x) function. Neurons using this activation function are referred as rectified linear units (ReLU units). Convolutional Neural Networks train several times faster than their equivalent tanh units [9]. In the Figure 10 the tanh units and the ReLU units are compared when training on an image data-set for a particular 4-layer CNN.

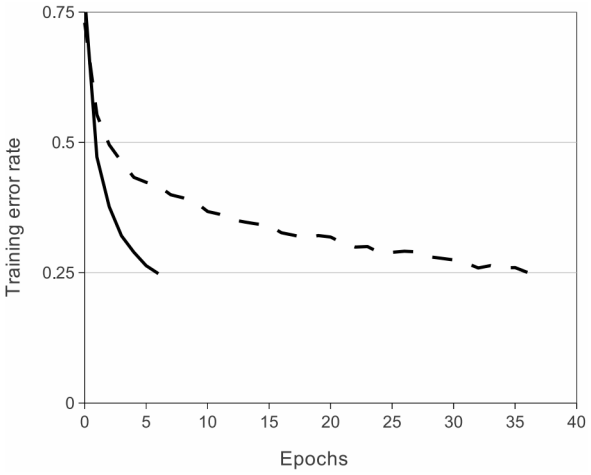


Figure 10: The four-layer CNN which uses rectified linear units in the detector layer (solid line), reaches the 25% error rate several times faster as the one using the tanh units (dashed line).

In this specific example, the network with the ReLU units trains about 4 times faster. This rate depends on the architecture but networks with ReLU units consistently learn several times faster than their equivalents with saturation neurons.

5.3 Pooling layer

In the pooling layer the pooling function is applied which is used to progressively reduce the spatial size of the representation and thus reduce the amount of parameters and computation in the network, hence also control over-fitting. It takes the output of the detector layer and combines it with it's nearby outputs. There are several pooling functions. The most popular is the maxpooling. It reports the maximum output within a rectangular area. Another common pooling function is average pooling. Here the pooling function reports the average output of a rectangular area.

There are also other pooling functions, which are not used as frequently as those two just mentioned, like L2 pooling, where the pooling function reports the square-root of the sum of the squares of the activation in the pooling region is taken. A modification of the average pooling is the weighted average function which takes the weighted average based on the distance to the central pixel.
Which kind of pooling function one should use in the different use-cases is not that trivial. There is some theoretical work done by Y-Lan Boureau [2] on how to choose the right pooling function. Their experiments proved that average-pooling and max-pooling are performing best.
Due to the summary of the pixels in a certain neighborhood, the pooling function makes the input approximately invariant to small translations of the input. This means that if there is a small variation, translation or shift in the input the pooling function still reports the same result. This is a very useful property if the goal is rather to know if there is a certain feature in an input than the exact location of that certain feature. The exact location of the feature is lost. The intuition is that once a feature has been found, it's exact location is not as important as the fact that the feature is there and the position

of the discovered feature relative to the other features. If we want to detect a face for example, it is more important to know that there is a nose an eyes in the picture and the relative positions than knowing the exact location of these features.
Another ability pooling can provide is learning the invariances. This means that if a pooling unit pools over several features that are learned with a different set of parameters, it can learn to be invariant to transformations of the input. An example of this ability is shown in Figure 12. In the picture there are three different kernels to detect a handwritten five. Each of the handwritten fives has a slightly different orientation. The input in the left picture and the input in the right picture of the five have also different orientations. Because the outputs of these three detector units are getting pooled, there is a large response in the maxpool unit and the five gets detected no matter what the orientation it is.

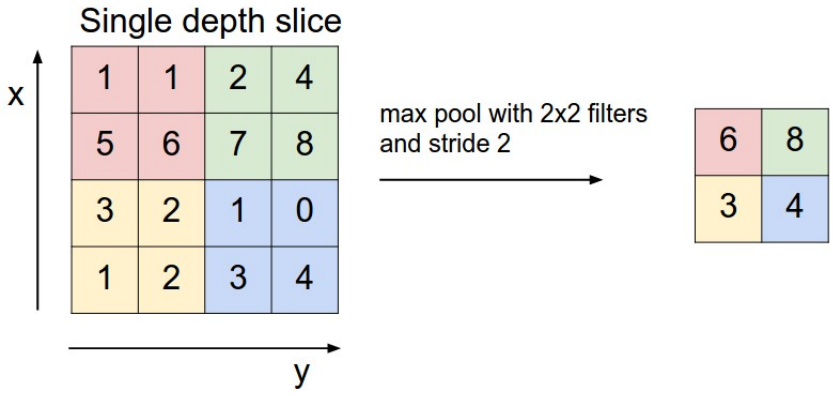


Figure 11: Maxpooling is applied to a 2x2 rectangular neighbourhood with stride 2. The maximum number of every 2x2 square (4 numbers) is taken.

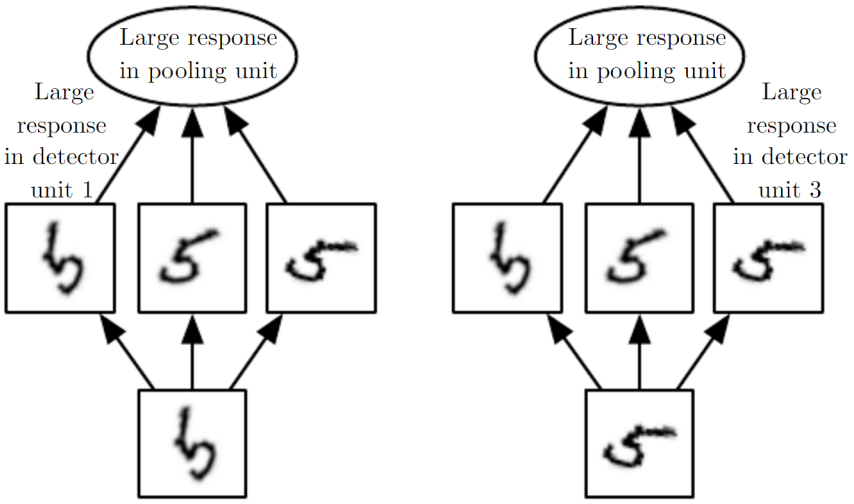


Figure 12: Picture from [6]

6 Training of Convolutional Networks

6.1 Backward propagation with Convolutional Neural Networks

This section is based on the calculations of Andrew Gibiansky [1]. One method how CNNs can be trained is with backward propagation and gradient descent. The backward propagation in CNNs works similarly to the back-propagation in MPLs but there are a few modifications of the back-propagation that are need to be done in the convolution and pooling layer. In the following two subsections it is assumed that the input is two-dimensional and has (N x N) neurons.
P : the number of predictions
E : Error function ${a_p}^L$: the predicted network output
t_p :targeted value.
l : the number of the layer, where l=1 is the first layer, and l=L is the last layer. Learning will be achieved by adjusting the weights such that the output matrix A^L is as close to the Target Matrix T as possible.

6.1.1 Backward Propagation in the Convolution Layers

The size of the kernel w is (m x m). $w_{a,b}$ means the entry (a,b) of the kernel. x^l is the linear output of layer l and y^l the nonlinear output and y^l- is the input from the previous layer. Thsi means:

$$x^l_{i,j} = \sum_{a=0}^{m-1}\sum_{b=0}^{m-1} w_{a,b} y^{l-1}_{(i+a)(j+b)} \tag{7}$$

$$y^l_{i,j} = \sigma(x^l_{i,j}) \tag{8}$$

$\sigma(x)$ is a nonlinear function such as the tanh function of the ReLU function (see section 5.2).
The first thing to figure out is the gradient of each weight. This is done by applying the chain-rule.

$$\frac{\delta E}{\delta w_{a,b}} = \sum_{i=0}^{N-m}\sum_{j=0}^{N-m} \frac{\delta E}{\delta x^l_{i,j}} \frac{\delta x^l_{i,j}}{\delta w_{a,b}} = \sum_{i=0}^{N-m}\sum_{j=0}^{N-m} \frac{\delta E}{\delta x^l_{i,j}} y^{l-1}_{(i+a),(j+b)} \tag{9}$$

The sum is taken over all the $x^l_{i,j}$ inputs where the weight $w_{a,b}$ occurs due to the shared weights. This equation $\frac{\delta x^l_{i,j}}{\delta w_{a,b}} = y^{l-1}_{(i+a),(j+b)}$ is the result of looking at the convolution equation.
To compute the gradient the so called "deltas" needs to be computed first. Deltas are the derivatives of the error function E in respect of an input unit x which means $\frac{\delta E}{\delta x^l_{i,j}}$
These deltas are also computed by the chain-rule as following:

$$\frac{\delta E}{\delta x^l_{i,j}} = \frac{\delta E}{\delta y^l_{i,j}} \frac{\delta y^l_{i,j}}{\delta x^l_{i,j}} = \frac{\delta E}{\delta y^l_{i,j}} \frac{\delta}{\delta x^l_{i,j}} (\sigma(x^l_{i,j})) = \frac{\delta E}{\delta y^l_{i,j}} (\sigma'(x^l_{i,j})) \tag{10}$$

Since $\frac{\delta E}{\delta y^l_{i,j}}$ the error of the current layer is already known, we can easily compute the deltas just by derivation of the activation function $\sigma'(x)$.
Furthermore the weights need to be computed and the errors need to be propagated back

to the previous layer. This is again done with the chain rule and the following formula:

$$\frac{\delta E}{\delta y_{i,j}^{l-1}} = \sum_{a=0}^{m-1} \sum_{b=0}^{m-1} \frac{\delta E}{\delta x_{(i-a),(j-b)}^{l}} \frac{\delta x_{(i-a),(j-b)}^{l}}{\delta y_{i,j}^{l-1}} = \sum_{a=0}^{m-1} \sum_{b=0}^{m-1} \frac{\delta E}{\delta x_{i-a,j-b}^{l}} w_{a,b} \tag{11}$$

Looking again at the convolution function, the following equation could be elucidated $\frac{\delta x_{(i-a),(j-b)}^{l}}{\delta y_{i,j}^{l-1}} = w_{a,b}$
The equation (11) provides the error value of previous layer. That equation only makes sense for points that are at least m points away from the edges. This can be fixed by zero padding the edges.

6.1.2 Backward propagation in the Pooling Layers

At the pooling layer, the forward propagation reduces a (N x N) block to a single value, the value of the "winning unit". The error afterwards is only dependent of the value of that "winning unit". That is why there is a need to keep track of the "winning unit". This is realized by memorizing the index of that unit during the forward pass which can then be used for gradient routing during back-propagation.
When using maxpooling, the error will be assigned to where it comes from, which means from the "winning unit". This is because units other than the "winning unit" did not contribute to the error and it does not make sense to change anything there.
When using average pooling the error is divided equally between all the units. That means every units gets the error value times $\frac{1}{N \cdot N}$

7 Optimization of Convolutional Networks

As the Convolutional Networks get bigger there is also a need for better hardware and exploiting parallel computing. Other than that, it is also possible to speed up the convolution by selecting the right convolution algorithm.
One possible way to do that is to separate the d-dimensional kernel in d vectors. Performing d one-dimensional convolutions is equivalent to the convolution operation elucidated in Section 4.1 but much more efficient. The kernel takes fewer parameters to represent as vectors. If the kernel has a width of w, the convolution operation requires $O(w^d)$ runtime and memory, while performing d one-dimensional convolutions requires $O(w \cdot d)$ runtime and memory. However separating the kernel is not always possible. The d-dimensional kernel needs the property to be able to be expressed as the outer product of d vectors.
Furthermore there are also possibilities to lower the cost of training the network.
Typically CNNs are trained in a supervised way, which shows to perform well on image classification tasks. The features learned by the CNNs are achieving state-of-the-art performance (see Section8). But the downside of supervised training is the need for expensive labeling as the amount of training data increases quickly the lager the network gets. Successful CNNs are performing good due to the amount do manually label images. For this reason, unsupervised learning, even though it is currently underperforming, remains an appealing paradigm, because it can make use of unlabeled images and videos [4]. There are several possibilities how the kernels can be obtained without supervised training. One way is to initialize the kernels randomly which works surprisingly well.

Another strategy is to use hand-crafted kernels for example kernels that detect vertical or horizontal edges.

7.1 Memory requirements

The main requirement to be aware of when working with Convolutional Networks is the memory requirement.
There are three main sources of memory to be aware of:
The raw numbers of activation at every layer of the CNN. They are needed later on for the training with backward propagation. A clever implementation that only stores these parameters during training and not during testing time which reduces the memory requirements during the testing.
The same goes with the kernels and their gradients that need to be stored for the backward propagation.
Thirdly every CNN implementation has to maintain miscellaneous memory to store for example the image data batches or the augmented version of the image.
As an example VGGNet (which is also presented in subsection 8.2) needs about 180MB of memory per image.

8 Case studies

The following CNNs all played an important role in the development of deep learning and image recognition. They all participated in the ImageNet Large Scale Visual Recognition Challenge (ILSVRC) which makes them easy to compare. The challenge is to classify images of size 224x224x3 into 1000 different classes. The ILSVRC provided 1.2 million training images, 50000 validation images and 100000 test images. The networks were rated by their top-5-error rate.

8.1 AlexNet

AlexNet is a CNN which won the Imagenet challenge in 2012. It achieved a top-five error rate of 16.4%, compared to the second best entry which had a top-5 error rate of 26.2%.

The AlexNet has 5 convolution layers and 3 fully connected layers at the end of the network. In Figure 13 the architecture is illustrated.
The AlexNet uses ReLU units in the detector layer.
Furthermore, overlapping pooling is used. Traditionally, the neighborhoods summarized by adjacent pooling units do not overlap. This means in detail that if the stride of the pooling is s pixels, and the pooling grid has size $(z \times z)$, then $s < z$. Whereas in the traditional way, $s = z$. Using overlapping pooling reduced the top-5 and top-1 error rate by 0.4% absolute. With this technique the risk of over-fitting gets also slightly decreased.
The first convolution layer has 96 kernels, the seconds has 256, the third has 384 kernels, the fourth 384, and the fifth 256 kernels. The fully-connected layers have 4096 neurons each. The whole network has about 60 million parameters. [9].

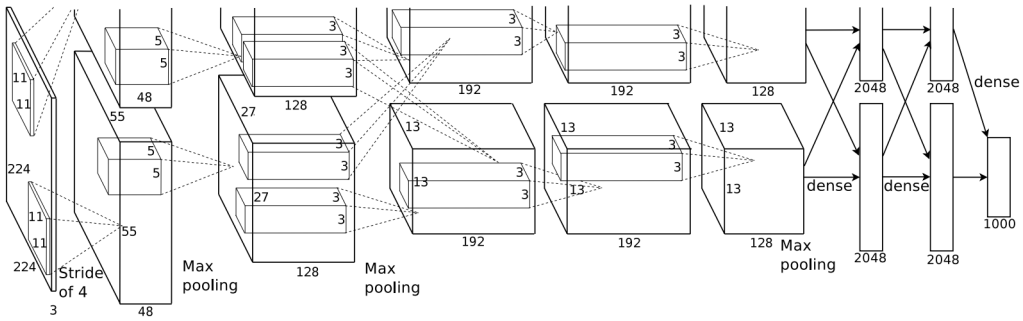


Figure 13: In the picture the architecture of the AlexNet is illustrated. It has 5 Convolution layers and 3 fully connected layers. Picture from [9]

8.2 VGGNet

VGGNet participated in the ILSVRC-2014. It had a top-5 error rate of 7.3% and is easy to implement due to it's simple architecture. An ensemble of seven networks were used for the ILSVRC-2014.
The VGGNet is very similar to the AlexNet. It also stacks convolution layers, max-pooling layers and ReLU layers and uses fully connected layers at the end of the network (see Figure 14). Kernels with constant size of 3 x 3 are used (which is the smallest size to capture the notion of left, right, up, down, center). The stride is fixed to one pixel on each axis. Zero padding is applied such as that the spatial resolution stays the same before and after each convolution operation. As pooling operation max-pooling is used on a 2 x 2 pixel window with a stride of 2. That means that there no overlapping pooling is performed. At the end there are a few fully connected layers.
As the images decrease spatially the number of kernels are increasing. First 64 kernels are used, then 128, 256 and 512. This way more high-level features are extracted. [11]

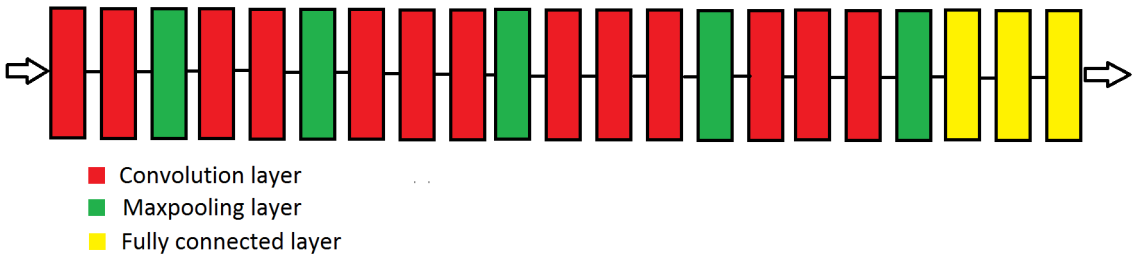


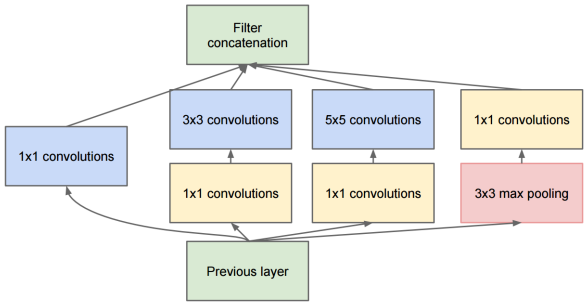
Figure 14: VGGNet Architecture

8.3 GoogLeNet

GoogLeNet is a deep Convolutional Network which won the ImageNet competition in 2014. GoogLeNet had a top-5 error rate of 6.67%

The main idea behind the successful architecture are the inception modules. They try to optimize the local sparse structure of a Convolutional Vision Network. The module acts as multiple convolution kernels, that are processed on the same input and does pooling at

Figure 15: Inception Module



the same time. The result is concatenated afterwards. It can extract for instance more general features with the (5x5) kernels and local features with the (1x1) kernel at the same time. It stacks up the inception modules and does an average pooling before the fully connected layer at the end. The whole network only needs about 5 million parameters. This is a huge memory saving compared to the 60 million parameters AlexNet uses or the 138 million parameters of the VGGNet. [13]

8.4 ResNet

ResNet a Convolutional Network which won the ILSVRC-2015. It has an top-5 error rate of 3.6%. The training time was between 2 and 3 weeks on a 8 GPU machine. The Network has 152 layers which is roughly eight times more than the VGGNet. With it's 152 layers ResNet is the deepest network ever participating in the ILSVRC but is still faster than the VGGNet. The good performance has to do with the particular architecture. The network reduces the size of the input quite fast. After the two first layers (convolution and pooling layer) the spatial size is reduced from 224x224 to 56x56. This way many parameters and conputational cost is saved in the following layers.

When deeper networks are able to start converging, there is a degradation problem. With the network depth increasing, accuracy gets saturated and then degrades rapidly. Unexpectedly, such degradation is not caused by overfitting, and adding more layers to a suitably deep model leads to higher training error. The solution to that problem was identity mapping. The so called shortcut connection (see Figure 16) is simply an identity function which output gets added up with the output of layers it skipped. In the plain network, the original input gets lost due to the transformations, whereas here the weight layers compute $F(x)$ which gets added to the original input x. It turned out that the shortcut connection works most efficient when skipping two or three layers. [7]

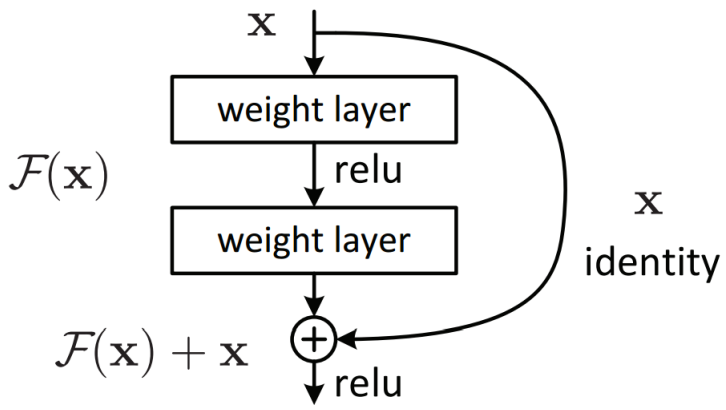


Figure 16: ResNet shortcut connection

8.5 Case Studies Overview

	AlexNet	VGGNet	GoogLeNet	ResNet
Top-5 error rate	16.4%	7.3%	6.67%	3.67%
Number of layers	8	16	22	152
Number of parameters	60 million	138 million	5 million	/
Number of networks used an ensemble	1	7	6	6

9 Conclusion

Overall Convolutional Neural Networks are a powerful tool and provide a way to specialize MLPs to work with data that has a grid-like structure. CNNs won apart from the ILSVR many contests and challenges like the CIFAR competition. Moreover a computer program using a CNN, called AlphaGo, was the first to beat a professional player at the board game GO [3]. Other tasks like classifying images or detecting objects in images and other jobs like playing games can be solved with Convolutional Neural Networks. Most architectures stack convolution layer, detection layer and the pooling layer. Often there is a fully connected layer at the end of the network. The trend seems to go towards using smaller kernels and more layers. Furthermore it seems like pooling layers and fully connected layers at the end of the network will disappear in the future.

References

[1] Convolutional neural networks, 2014.

[2] Y-Lan Boureau, Jean Ponce, and Yann LeCun. A theoretical analysis of feature pooling in vision algorithms. In *Proc. International Conference on Machine learning (ICML'10)*, 2010.

[3] Patricia S Churchland and Terrence J Sejnowski. *The computational brain*. MIT press, 2016.

[4] Alexey Dosovitskiy, Jost Tobias Springenberg, Martin Riedmiller, and Thomas Brox. *Discriminative Unsupervised Feature Learning with Convolutional Neural Networks*. Curran Associates, Inc., 2014.

[5] Kunihiko Fukushima. *Neocognitron: A Self-Organizing Neural Network Model for a Mechanism of Pattern Recognition Unaffected by Shift in Position*. 1980.

[6] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. Deep learning. Book in preparation for MIT Press, 2016.

[7] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. *Deep Residual Learning for Image Recognition*. 2015.

[8] Andrej Karpathy. Convolutional neural networks for visual recognition, 2015.

[9] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. *ImageNet Classification with Deep Convolutional Neural Networks*. Curran Associates, Inc., 2012.

[10] Michael A. Nielsen. *Neural Networks and Deep Learning*. Determination Press, 2015.

[11] K. Simonyan and A. Zisserman. *Very Deep Convolutional Networks for Large-Scale Image Recognition*. 2014.

[12] Yusuke Sugomori. *Java Deep Learning Essentials*. PACKT Publishung, 2016.

[13] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott E. Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. *Going Deeper with Convolutions*. 2014.