Chapter 3

Mathematical/Technical Background

3.1 Used Models

The following models where used:

3.1.1 Tensor

In the context of artificial Neural Networks and most of the scientific computing frameworks like Torch7 or TensorFlow a tensor is nothing more than a potentially multidimensional matrix with variable number of dimensions. A tensor can be a simple value if the tensor has a dimension of 0. A tensor of dimension 1 is a vector. A tensor of dimension 2 is a matrix. A tensor of dimension 3 is a stack of matrices, but it is still possible to perform a lot of the matrix operations on it. The highest number of dimensions for a tensor used in this thesis was 4. In this case it was the weight tensor of a convolutional layer.
Tensors are used in most deep learning computing frameworks, because they grant a very fast access to each value of itself. If a GPU is handling multiple matrices it switches

between them all the time so it couldn't operate efficiently. In this context efficiently is defined by code length and GPU time. For example:

$$\mathbf{M}_1 = \begin{bmatrix} m_{11} & m_{12} & m_{13} & m_{14} \\ m_{21} & m_{22} & m_{23} & m_{24} \\ m_{31} & m_{32} & m_{33} & m_{34} \end{bmatrix} \qquad \mathbf{K}_1 = \begin{bmatrix} k_{11} & k_{12} & k_{13} & k_{14} \\ k_{21} & k_{22} & k_{23} & k_{24} \\ k_{31} & k_{32} & k_{33} & k_{34} \end{bmatrix}$$

$$\mathbf{M}_2 = \begin{bmatrix} m_{11} & m_{12} & m_{13} & m_{14} \\ m_{21} & m_{22} & m_{23} & m_{24} \\ m_{31} & m_{32} & m_{33} & m_{34} \end{bmatrix} \qquad \mathbf{K}_2 = \begin{bmatrix} k_{11} & k_{12} & k_{13} & k_{14} \\ k_{21} & k_{22} & k_{23} & k_{24} \\ k_{31} & k_{32} & k_{33} & k_{34} \end{bmatrix}$$

$$\mathbf{T_M} = [\mathbf{M}_1, \mathbf{M}_2] \qquad \mathbf{T_K} = [\mathbf{K}_1, \mathbf{K}_2]$$

$$\mathbf{T}_{\mathbf{M}'} = \mathbf{T_M} \odot \mathbf{T_K} = [\mathbf{M}_1 \odot \mathbf{K}_1, \mathbf{M}_2 \odot \mathbf{K}_2]$$

$$\mathbf{T}_{\mathbf{M}'} = \begin{bmatrix} \begin{bmatrix} m_{11 \cdot k11} & m_{12 \cdot k12} & m_{13 \cdot k13} & m_{14 \cdot k14} \\ m_{21 \cdot k21} & m_{22 \cdot k22} & m_{23 \cdot k23} & m_{24 \cdot k24} \\ m_{31 \cdot k31} & m_{32 \cdot k32} & m_{33 \cdot k33} & m_{34 \cdot k34} \end{bmatrix} \\ \begin{bmatrix} m_{11 \cdot k11} & m_{12 \cdot k12} & m_{13 \cdot k13} & m_{14 \cdot k14} \\ m_{21 \cdot k21} & m_{22 \cdot k22} & m_{23 \cdot k23} & m_{24 \cdot k24} \\ m_{31 \cdot k31} & m_{32 \cdot k32} & m_{33 \cdot k33} & m_{34 \cdot k34} \end{bmatrix} \end{bmatrix}$$

We want to perform a tensor × tensor element wise multiplication similar to a matrix × matrix element wise multiplication
(with tensors $\mathbf{T_M}$ and $\mathbf{T_K}$ of size $\mathbb{R}^{2\times3\times4}$, which I also write with the dimensions as an elevated index to be able to differentiate between different types of tensor: $\mathbf{T}_{\mathbf{M}}^{(2,3,4)}$ or $\mathbf{T}_{\mathbf{K_1}}^{(1,3,4)} \equiv \mathbf{K_1}$).
For that we just need a sequence of multiplications and summations to do this, which can be processed very efficient on a GPU. But with multiple matrices to handle the whole thing would look differently on the GPU because it would need CPU commands in between to find and access the next matrix it needs in the process, which would slow down the whole calculation. Another advantage is to use the multidimensional relation between elements in the tensor. For example it can look on a specific patch of pixels in all color channels of an image to perform a convolution with this patch of pixels in all color channels at the same time without searching for the certain spatial location on all color channels separately to perform the convolution. Tensors are often synonymously called volumes, since from a software engineering point of view it is nothing else than a volume of values addressed in a specific way so that they can be fast accessed.

3.1.2 Artificial Neural Network

The Figures 3.2a, 3.2b, 3.4 and 3.5 are part of Colah's Blog [Ola14b].

In the introduction I'm talking about the great achievements of artificial Neural Networks in computer vision, however there have been breakthroughs thanks to artificial Neural Networks in other areas as well. But artificial Neural Networks remain a challenging and unpredictable matter to work with. It is difficult to unveil the inner life of them, to measure their inner state and to understand what is really going on.
A well trained network can reach fascinating results, but the same network can also fail under worse conditions and in both cases it is hard to understand why. Deep Neural Networks are particularly obscure to retrace, but they are also the most promising ones. Luckily, deep networks with few neurons in each layer (low-dimensional Neural Networks) are much easier to study and they do principally the same on a lower level of com-

plexity.
This opens a new perspective to explore them, because it is possible to visualize those low-dimensional networks and use an area of mathematics called topology to get a better view of the networks' behavior and what they do with data they take in.
We start off with a simple example:
One layer of a Neural Network (not counting the input layer figure 3.1) also called a perceptron network will try to separate a given dataset into two or more user defined groups by optimizing a linear function which will split the two groups as good as possible, which is called logistic regression. The first image (figure 3.2a) shows the two groups and the linear function of the network at the beginning of the optimization progress. In this state the linear function is just defined randomly by the initialization values of the network. And in case of a two dimensional domain space, the neural network would only have to input and two output neurons.

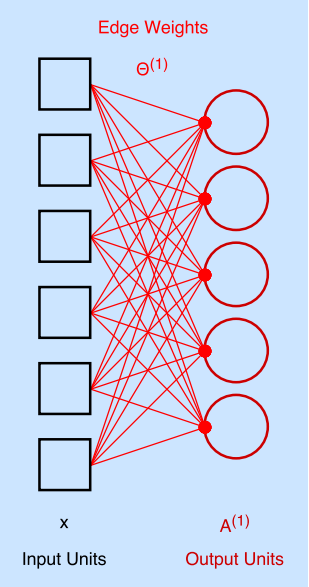
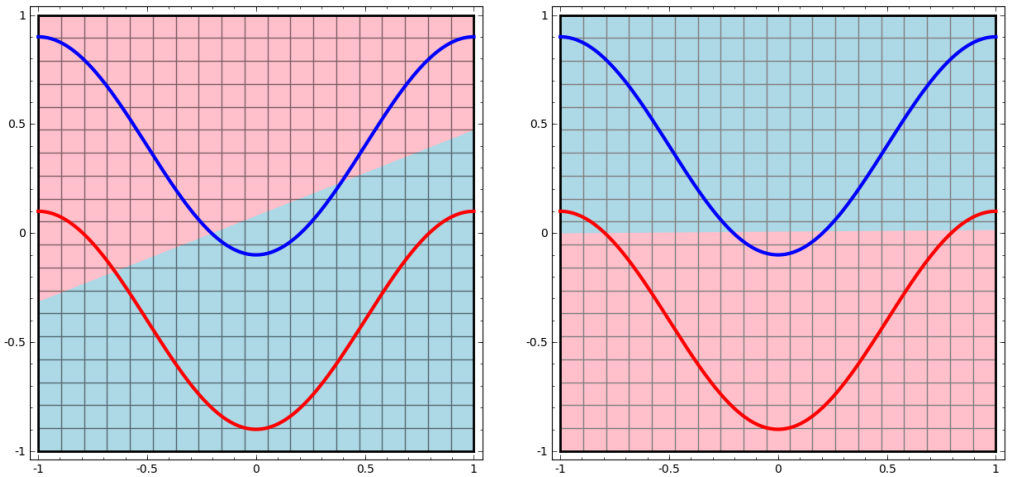


Figure 3.1: Perceptron-Network



(a) linear separation after initialization

(b) linear separation failed

Figure 3.2: illustration logistic regression

After some iterations of the perceptron network, the linear function looks like this. As shown, the network has difficulties to separate the two groups in this case entirely. This is natural for a perceptron network, because it is just capable of a linear separation and cannot tell apart groups in a dataset, which are too intertwined for a linear separation. Therefore it isn't even possible to simulate an XOR function with a perceptron network.
But adding a second layer of neurons may do the trick: Modern networks tend to have at least one "hidden layer" in between the input layer and the output layer. This would for example look like this 3.3

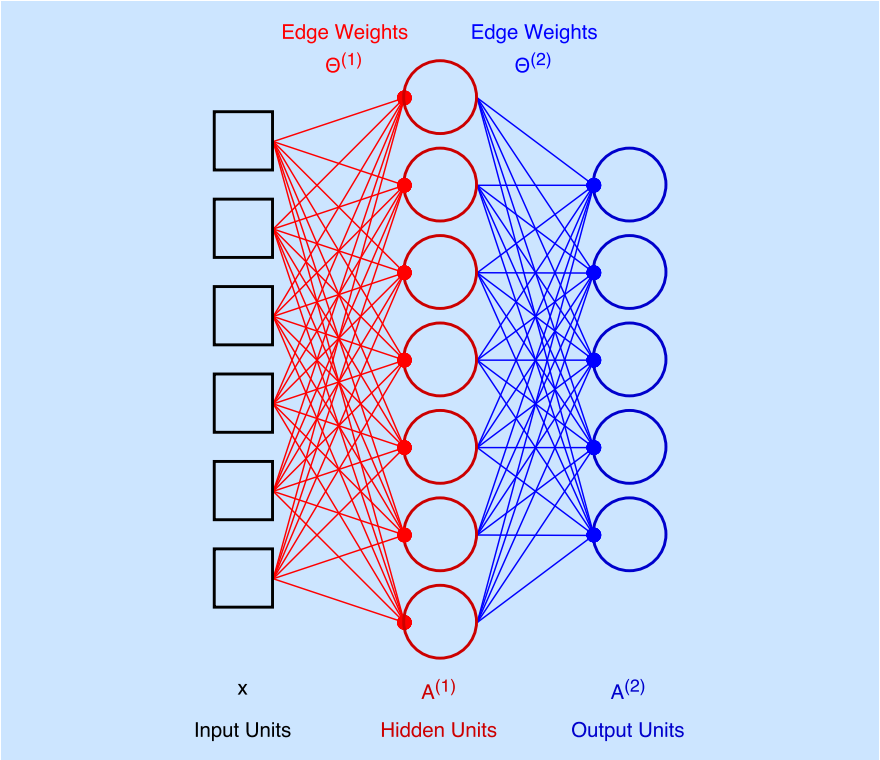


Figure 3.3: Feed Forward Network with one Hidden Layer

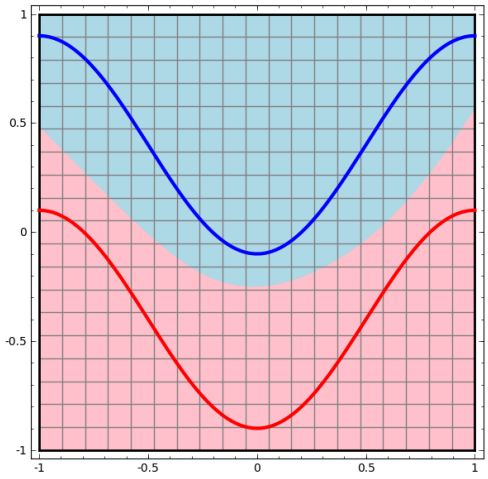


Figure 3.4: curved separation

Now the network is capable to create a curved line of separation, which is an important ability on the most datasets. But how exactly did the second layer make this possible? If we go into the second layer after some training of the network and see the output of the first layer before processing it in the second layer you would see something like this:

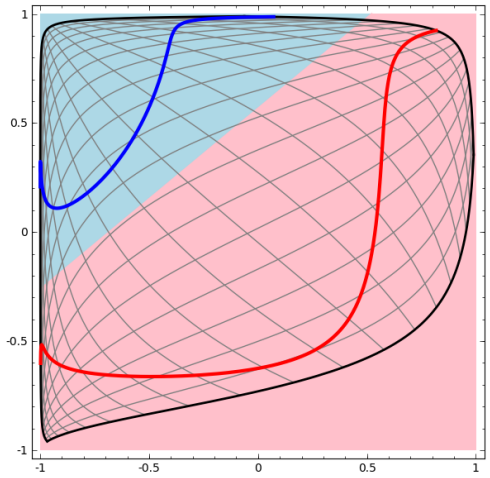


Figure 3.5: topological distortion of the domain space

Now the second layer is able to perform a linear separation on the two groups and split them completely. In this representation we used a topological transformation of the manifold to imitate the functionality of the first layer which culminates in the distortion of the whole space where the network is now operating on after the first layer.
So we saw how to manipulate the manifolds so that they can separate the codomain in different object class spaces, but what if the domain of the problem is multidimensional and cannot be solved by dividing the codomain with a simple line?
For this matter the concept of a hyperplane is useful:

Hyperplane

If the input layer of a Neural Network has more than two neurons, the network is forced to separate the classes in more than two dimensions. A line wouldn't do the trick in this case. In 3 dimensions we need a plane to divide the classes effectively and in 4 dimensions and more the multidimensional plane that separates the classes is called *hyperplane*.
However a great achievement of Neural Networks is, that they are able of dimensional reduction, which means that the number of output neurons in a neural network can be smaller than the input number. With this technique and the right layout of the network a dimensional reduction of the domain space can be achieved, which means that Neural Networks are able to reduce the number of unknown input variables in a way that they recognize not stochastically independent properties between the input variable and can map one or more variable on the others. There are a couple of useful applications for this matter, one is language processing or rather mapping one language on another for translation purposes. This can be realized with an unsupervised learning algorithm and a Recurrent Neural Network (RNN). A RNN is obligatory for language processing, because without feedback loops in the neural network itself the network wouldn't be able to decode the concept of before and afterwards. It would try to process every word in a whole text simultaneously and wouldn't comprehend, that a text is a sequence of words. But with unsupervised learning and enough learning matter a RNN would be able to process language by its content and could translate words to other languages.

Figure 3.6: cutout of t-SNE visualizations of word embeddings. from [TRB10]

(a) see complete image at [Tur10]

To visualize this, there are large maps of words of languages (figure 3.7a) where a RNN has reduced the dimension of different texts in different languages, so that all words of a text are positioned on a two dimensional plane with regards to two variables which are a combination of multiple variables mapped on them. Now we are able to lay two of those word maps, where one word map stand for one language, on top of each other and can now see which words of these two different languages lay close to each other, which will possible be a translation between these two words. Modern intelligent translation algorithms, like the Google translator work among other things with these technique. Because this way Google doesn't have to save a translation between every language to every other language. It can map multiple languages on top of each other.
Interestingly to notice is, that the Neural Network has its difficulties with words, that are used synonymously or are used often in the same way like days of a week or months. Maybe some month are more often associated with cold than others, but if the text sources don't work with the different time dates a lot it is impossible for this kind of network to tell the difference. So one can end up translating January with March in another language.

the Stochastic Gradient Descent (SGD):

Principally SGD is some kind of hill climbing algorithm that uses the gradient of a point in the domain space to determine a direction vector to change the weights in the network in this direction so that the loss function computes a lower value. By this means it is more a minimum descent than a hill climbing algorithm. In order to pass back the right gradients back from the loss function at the end of the network to the neuron in the previous layer, the algorithm uses the backpropagation algorithm by reverting the forwardpropagation of the network. The forwardpropagation algorithm calculates the output, in the case of this thesis the prediction for the object classification of the network by forwardchaining every calculation of every layer starting from the input layer. After the network has calculated the prediction vector in this way for the object recognition, the loss function computes the network's loss vector, by subtracting the correct

prediction vector (given by the supervised learning labels) from the prediction vector of the network. We will see later how the backpropagation is formed out of the forward-propagation in the mathematical background part of this chapter (page 46). Like all hill climbing algorithms the SGD method can encounter difficult situations where it can get stuck in the optimization process. Plateaus are a problem, in this case the gradient vanishes because no direction can point at any inclination. This problem is covered at chapter [4.2.2]. Another problem is a very noisy codomain, because the algorithm gets stuck easily at a local minimum. In this case Simulated Annealing would be far better. In the implementation for this thesis SGD is used for the training process (often called *learning process* interchangeably)
Does a deeper or wider network converge faster and stronger?
Since its called deep learning and not wide learning the answer is obvious, but the question is justified. First there is no reason to use a deep Neural Network at all. A wide enough Neural Network with just a single hidden layer is able to approximate any given function with enough training data and time, if the function is not too far-fetched. But extremely wide and shallow networks are problematic, because they are very good at memorization which leads to overfitting. It is possible to train a network with every possible input value and a wide enough network will eventually memorize all the output values that it should, but in reality it won't have every possible input value to train with.
The big advantage of multilayer networks is they can recognize features of different levels of abstraction, for example the layers in an image recognition CNN train itself to recognize different shapes; to say it with Matthew D. Zeiler and Rob Fergus words:

"Layer 2 responds to corners and other edge/color conjunctions.
Layer 3 has more complex invariances, capturing similar textures
(e.g. mesh patterns [...]).
Layer 4 shows significant variation, but is more class-specific:
dog faces [...]; bird's legs [...].
Layer 5 shows entire objects with significant pose variation,
e.g. keyboards [...] and dogs [...]" [ZF13]

(cf. figure 3.8 for illustration)
In this way multilayer networks are much better at generalizing than wide networks.

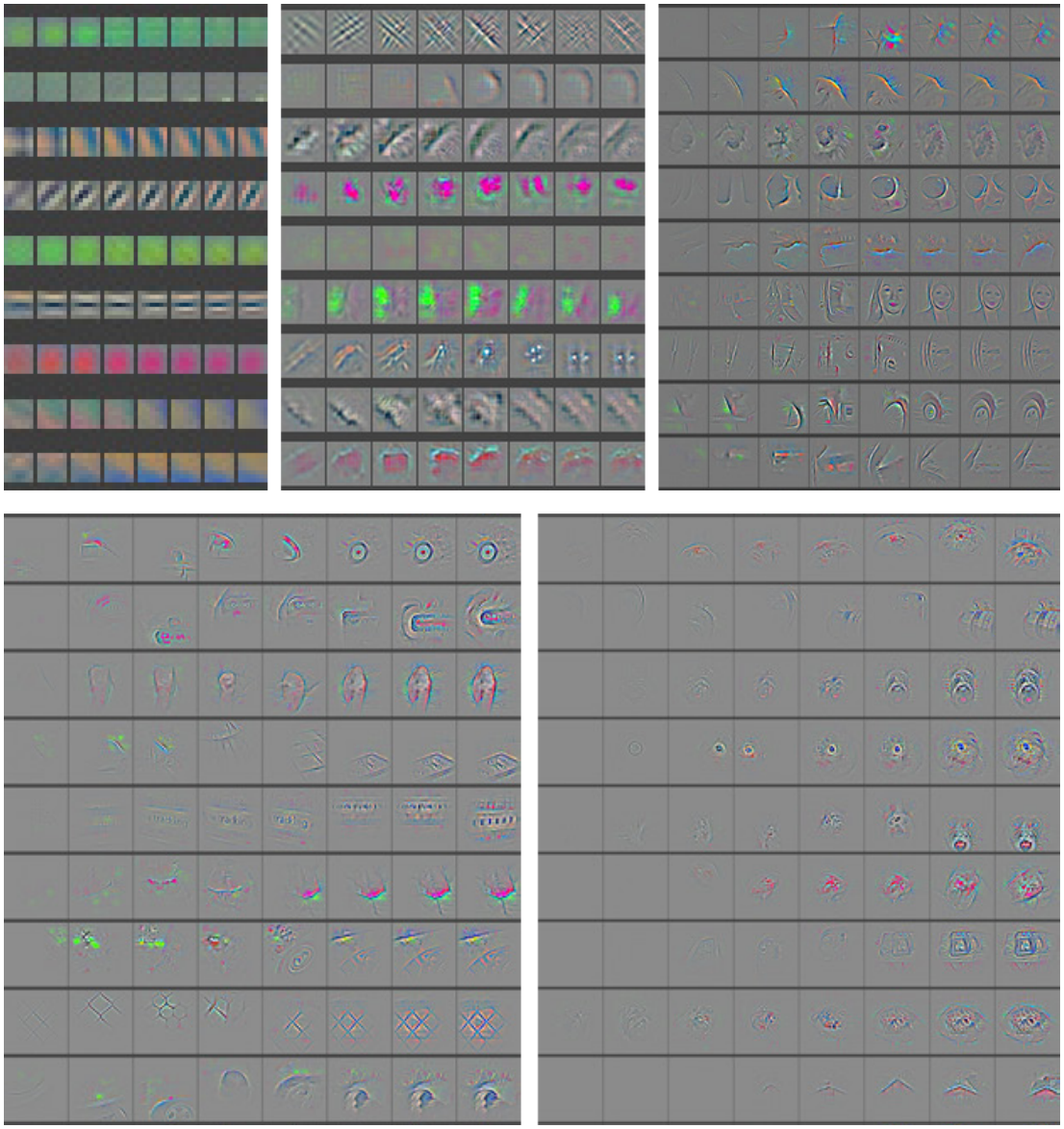


Figure 3.8: recognized features captured at different layers, from [ZF13]

from top to bottom / from left to right recognized features from layer 1 to layer 5, one specific feature per row

the Transfer Function:

The transfer function is an intrinsic part of a neuron and is often called activation function synonymously. Though, in Torch7 and many other scientific computing frameworks it is represented as an own layer of the network and it can actually be thought mathematical as an own solitary part of the network, which not every neuron must adopt, but it was originally intended to be an intrinsic part of a neuron. It passes the summed up and weighted input values through the transfer function to all outputs. The transfer function enables the neuron to imitate different logical, set and mathematical functions with the big advantage that the Neural Network can decide by training how intense it will use different functions for different input stimuli. There are different kind of transfer functions:

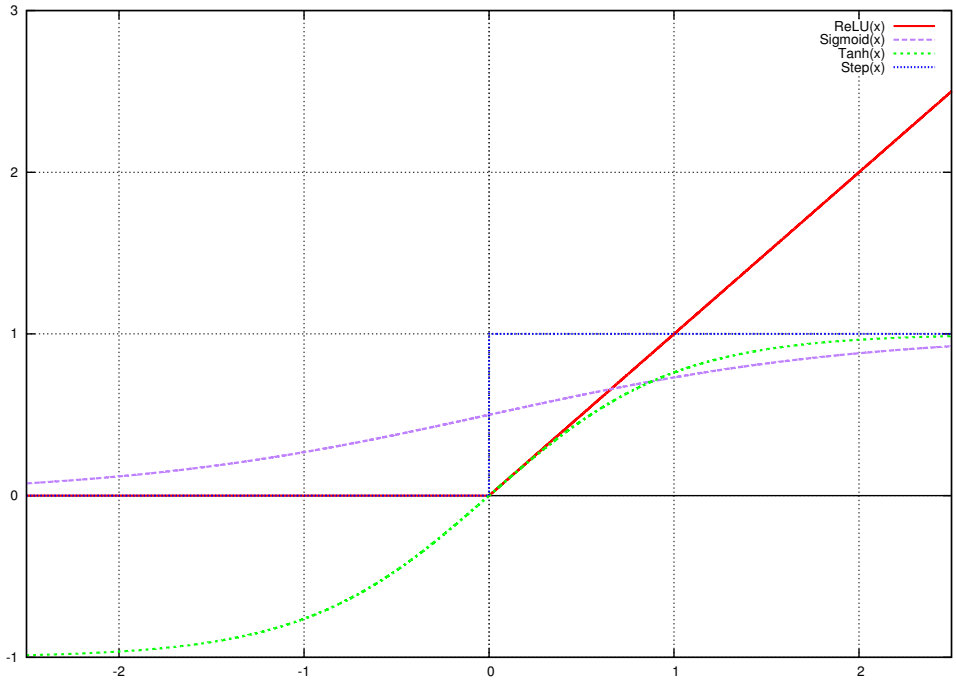


Figure 3.10: transfer functions

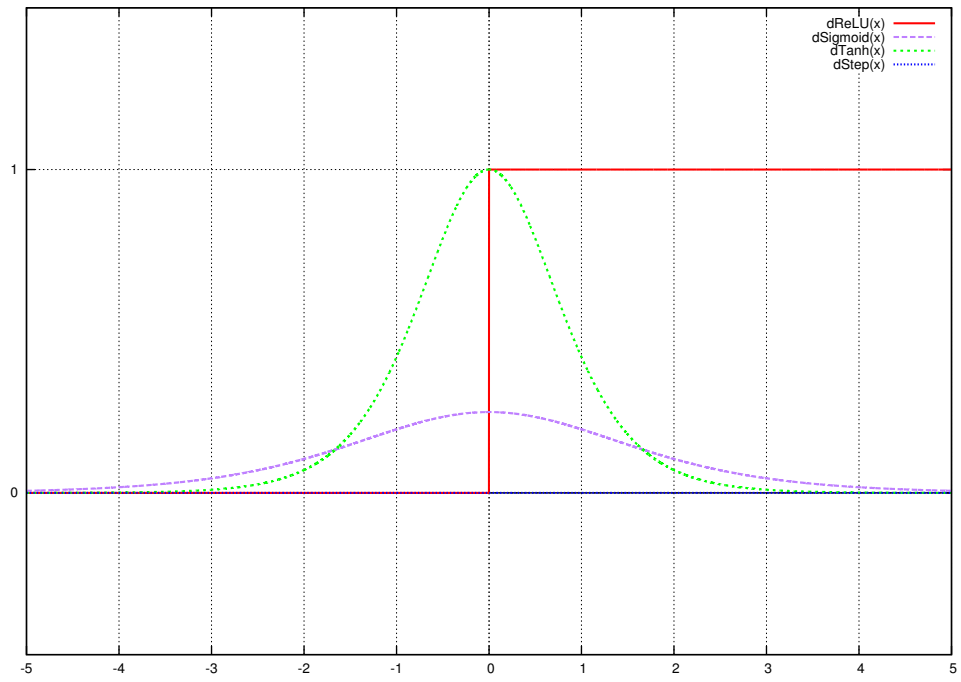


Figure 3.11: df/dx of transfer functions

$$\text{Sigmoid function: } \operatorname{sigmoid}(x) = \frac{1}{1+e^{-x}} \qquad \frac{d\,\text{sigm.}}{dx} = \frac{e^x}{(1+e^x)^2}$$

$$\text{hyperbolic tangent function: } \tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}} \qquad \frac{d\tanh}{dx} = \frac{4 \cdot e^{2\cdot x}}{(1+e^{2\cdot x})^2}$$

$$\text{RELU function: } \operatorname{relu}(x) = \begin{Bmatrix} 0 & | \, x \leq 0 \\ x & | \, x > 0 \end{Bmatrix} \qquad \frac{d\,\text{relu}}{dx} := \operatorname{step}(x),\ \{0 \mid x = 0\}$$

$$\text{step function: } \operatorname{step}(x) = \begin{Bmatrix} 0 & | \, x \leq 0 \\ 1 & | \, x > 0 \end{Bmatrix} \qquad \frac{d\,\text{step}}{dx} := 0,\ \{0 \mid x = 0\}$$

Most of the CNNs uses the Rectified Linear Unit (ReLU) transfer function today, because it can be processed very fast and its reduces the likelihood the vanishing gradients problem. For vanishing gradient problem see chapter [3.2.2].
There are many more transfer functions, but all transfer functions have to be differentiable for the backpropagation process to work properly. The easy explanation is, that the backpropagation algorithm would tear the manifold open through incontinuous topological transformations, if the function isn't differentiable. The network then wouldn't be able to derive a correct gradient during the training process, which leads to miscalculations in the SGD procedure. The complex explanation can be seen in the partial differentiations in the mathematical background (equation 3.15), where a non differentiable function wouldn't form a sound gradient.
It is possible to perform some non-linear separations (e.g. imitate an XOR function) with one perceptron layer of an Neural Network, if the neurons use a special transfer function, like normal distributed transfer functions, but isn't used in this approach, because using enough hidden layers to perform the required non-linear separations is the better option in nearly all cases.

the Batch Normalization:

Batch normalization is a method to reduce internal covariate shift in Neural Networks. The internal covariate shift occur in networks with many layers and it describes the effect that every layer shifts its input values slightly away from a zero-mean and unit variance condition. Since those effects increase exponentially it can lead to other unwanted effects in the network, such as that every pixel surpasses the ReLU activation function, which can lead to detecting features that aren't there and it is for the later layers overall harder to work on highly shifted volumes. In this way the batch normalization prevents a slow down at the training and lead to a higher overall accuracy. It also enables the use of a higher learning rate, potentially providing another boost in speed. I cannot confirm the latter, because I didn't run many tests with different learning speeds and I worked with the batch normalization throughout. Testing every scenario which might improve the network, is something many other related research has done and since I didn't aim to break records with my approach the purpose of my approach is another, which I describe at chapter [2.2]. But I think, that the batch normalization enable the use of higher learning rates, sounds plausible, which is however a strange interrelation, because a faster learning rate means more overswinging, which means that the batch normalization has to provides some damping as well.

3.1.3 Convolution

In the context of image processing a convolution is an operation using a kernel, which is a small matrix to do blurring, sharpening, embossing, edge detection, and more by adding each element of the image to its local neighbors, weighted by the kernel. The mathematics behind this is described in chapter [3.3.5]. It's enough to keep this in mind to understand the following and to know that the kernel also called convolution matrix is shifted over the image while operating so that each pixel has been affected by its local neighbors and the values of the convolution matrix. You should also take into account that the output image will be a few pixels smaller due to this operation, which can be prevented, as illustrated in chapter 3.1.3.

What is a Convolutional Neural Network (CNN)?
Generally speaking a CNN is a Feed Forward Neural Network which contains a number of convolutional layers figure 3.12,

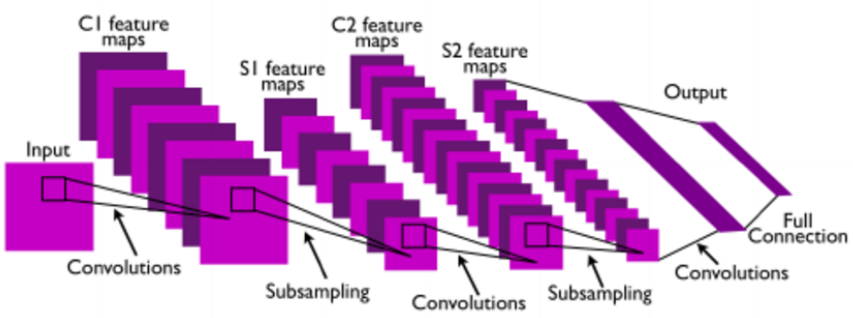


Figure 3.12: simple CNN, from [LKF10]

at least one but in most cases more. A convolutional layer is layer with conventional neurons like those we saw above in chapter [4.1.2], which form in specific groups together a small Neural Network on their own inside the CNN. This group of neurons could be seen as one combined neuron, but they are arranged in a specific way so that every combined neuron is there for one specific patch of pixels on the image. This patch of pixels in the size of the convolution matrix becomes the input values for this specific combined neuron. Every combined neuron contains a number of neurons in size of the convolution matrix and they are arranged in one layer like a convolution matrix (green orbs form one combined neuron, see 3.13).

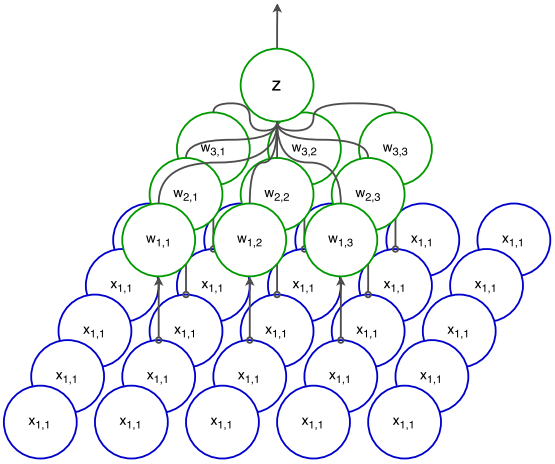


Figure 3.13: one convolutional unit over a certain path of input pixels

Since every Neuron has a weight for each input value the question is which weights have those neurons in the combined neurons? The answer is simple. Every combined neuron is a copy of the others, they are all similar and they have all the same weights for the same neuron inside of them (see figure 3.14). So one can map one convolution matrix onto the weights of the neurons in all combined neurons in one convolutional layer.

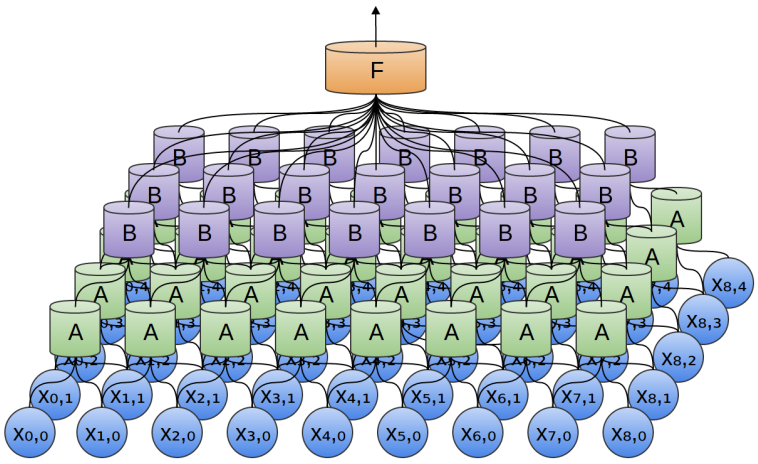


Figure 3.14: convolutional layer, from [Ola14a]

after the combined neuron processed one path of pixels follows a batch normalization and possibly a tranfer function

Now the convolutional layer is able to perform a convolution with the output of the last layer, which is in the first convolutional layer an image and in the other layers a tensor or a stack of images. This happens because the output of one convolution doesn't have to be one image it can be a multiple numbers of images for each convolution matrix. Though every combined neuron performs the convolution with the same convolution matrix, it is possible to use more then one convolution matrix on the same input tensor, but than the layer needs another set of combined neurons for each feature that should be detected with a separate filter (convolution matrix). Another convolution matrix for another feature will than maps their weights onto the input weights of the combined neurons. So one convolutional layer is able extract different features in one input image

or tensor (which is a stack of multiple preextracted features, so we are talking about features of features already). One layer of output features of one convolutional layer is called mini-batch layer, so we can differentiate now between the spatially dimensions of a feature image tensor (height, weight) and the depth of the stack (number of mini-batch layers).
At the end of the CNN after the convolutional layers has downsized the size of the input image but gained more mini-batch layers comes the part of the network which has to score every extracted feature, because since every convolutional layer is still a Neural Network layer it can be trained to optimize its feature extraction. This happens by the backpropagation algorithm, which I covered shortly in the SGD subchapter [3.1.2] and later more in detail in chapter [3.3.4].
But how does the backpropagation in the convolutional layers know how to relate it changes to those layers' feature extraction weights to the "real world" and not just by the gradient output of the next layer, which leads to the point at the end of the network where there is no layer with a gradient output. What we missed here is the last part of a CNN, the fully connected layer part. This part of the network tries to decide based on the output of the last convolutional layer of the network which class it has detected. But what is a class for that matter? In the field of object detection A class is the type of objects. For example a cat or a dog. Things that are captured on images which can be perceived and grouped by humans, machines which are capable of computer vision and also animals. But there are also a lot of other things which CNNs can perceive on certain images like genome sequences, microbes like plankton and tumor cells on MRT images.

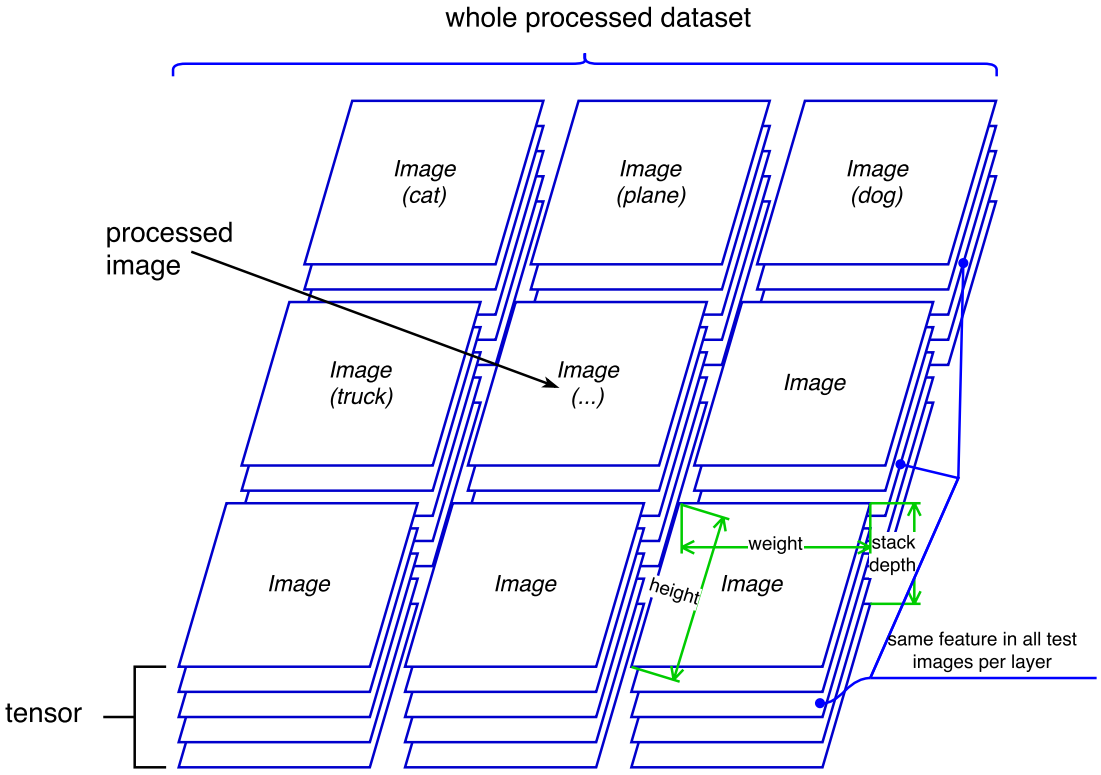


Figure 3.16: processed dataset

Back to the fully connected layer. After this layer has made it decision for one or usually a stack of thousands of images while it is been trained it receives a loss value for each

image based on a loss function. In the testing mode it is of course blind in the sense that it doesn't get any feedback, at least if we use supervised learning as a training and testing program. This is due to the principle of how supervised learning/training should be proceeded. If the CNN would in one epoch of training get to know the results of the test images it is very possible that it begins to learn by heart and scores perfectly in the next testing without recognizing anything, it would just repeat our answers, what we don't want. This effect is scientifically called overfitting, see subchapter *Overfitting* [4.2.1]. There are other learning methods than supervised learning, which for example wouldn't need any labeling at all.
We saw in the *"What is a hyperplane?"* subsection, that there are possible applications for CNNs to use unsupervised learning. There are also interesting applications for reinforcement learning for CNNs which I linked at the end chapter [6.3].
So the backpropagation in the convolutional layers can relate their changes in their feature extraction weights to the definition of the project by getting feedback during training so that the fully connected layer can propagate this feedback back to the parts of the network which needs improvements. Therefore each training dataset should have a label for each pictures which tells the network the class of the image. Each testing dataset (as long as the network completes the training epochs) should also have a label, because so we can compare after each training and testing epoch the score of both and realize faster when the network levels off (see overfitting[4.2.1]). The fully connected layer is in fact a basic non-convolutional layer, a "conventional“ layer so to speak, but it is necessary to connect at some point every neuron with every other neuron between two layers, because otherwise the network would see an image as multiple images and wouldn't see the whole picture. After the network completes the whole training process and operates in the real world there wouldn't be, of course, any labels. Then the network is on its own and can just perceive images the way it has learned to; unless it has something like a companion[1](see page 31) network, which has another channel to gain information about the real world.
So we saw, that the optimization forms naturally out of the backpropagation and can change each weight accordingly to reach a smaller loss score in the SGD process. But of course SGD do not provide the ultimate answer to reach in every given scenario the perfect loss score. There are simply to many possible weight combinations which can lead to a better score, than SGD can find. But there are other methods to find faster and to find a better combination of weights for a network. Beside the relative simple SGD algorithm the other more sophisticated algorithms for approximation the global optimum of a given non-linear function are far more complex so that I won't go much into detail about them at this point. The thesis approach only uses the SGD algorithm.
A very commonly used algorithm in this category is the SA (Simulated Annealing) algorithm. The SA heuristic marks at each step some neighboring states of the current state through a special method as a probable new target for moving the current state to. Then a probabilistically method decides between moving the system to a new state or staying at the current state. But after each cycle of considering a new location and a possible movement the method for marking neighboring states shrinks down the radius where it looks for new states, which leads possible to a fully annealed state, where the algorithm has found the global optimum.

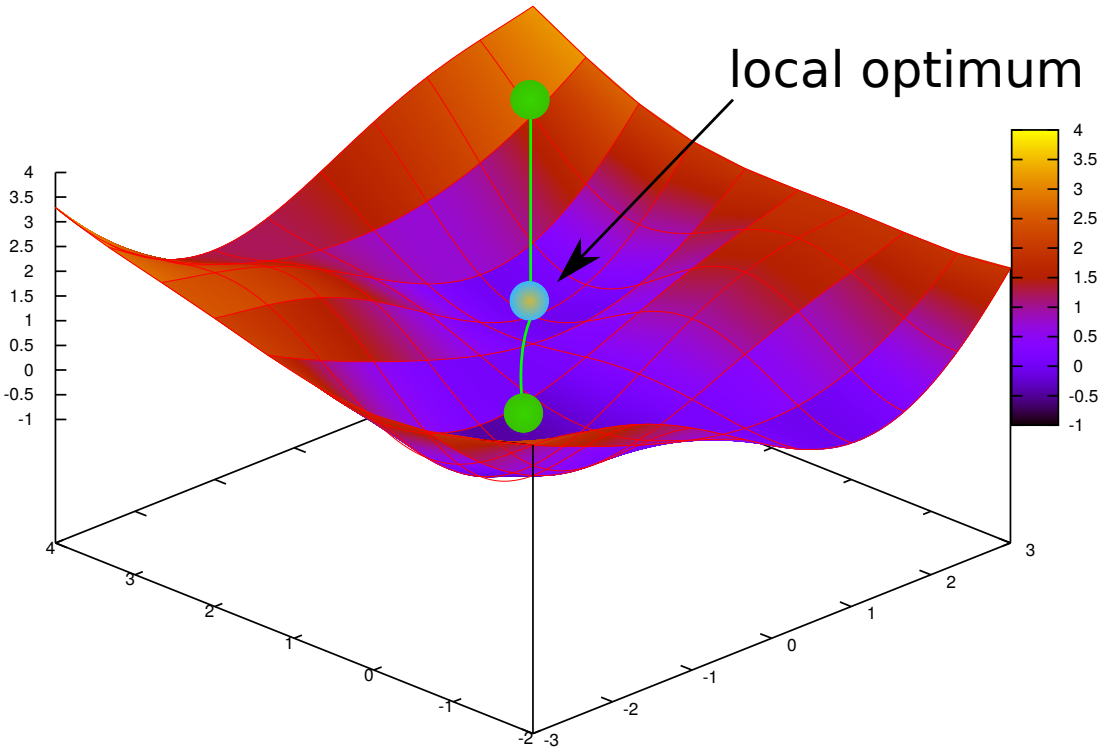


Figure 3.17: local optimum

Nag [SMDH13] is an algorithm in this category, that is a version of a SGD algorithm with an additional inertia attached to the current optimization vector, that makes sure the approximation process didn't get stuck so easily in a local optimum, because when it is reaching the local optimum it is still moved forward above a small ascent by its inertia vector. See figure 3.17:

There are other optimization algorithms, that has been proven to work better than SGD, that are preimplemented in torch7 like CMAES (Covariance Matrix Adaptation Evolution Strategy) [GMW15] or Adam [KB14]. more details on different optimizations algorithm can be found here: [Rud16].

Since this is a multidimensional problem, with millions of variables it becomes clear why human or object recognition performed by other lifeforms is a great achievement of nature. I assume, with enough time, probably a few thousand of years, a evolutionary algorithm for hill climbing (or loss minimization) would deliver the best results on a very complex classification problem, since those algorithms can handle tasks with numerous amounts of values yet at best, if time is no issue. A evolutionary algorithm has the big advantage, that it never stops trying out something new.

[1] I actually came up with this name by my self. But the principle of two networks which train each other and learn differently about the world (for example, one could learn supervised the other "reinforced") is old. The big advantage of reinforced learning is that the feedback for completely new events can be based on real world effects and doesn't have to be simulated in a lab at first with supervised labels and stuff like this. If, for example, a robot with a reinforced learning algorithm falls off a cliff and cannot get back to its known world so easily it would notice that something out of the normal definition of the project has occurred, because it would receive very bad results in its Markov Decision Process and can not find a way out of this bad loop without trying out something innovative new. Actually this covers a big problem in image recognition, which will above all bother the autonomous car industries for a longer time than you might think. (more about this on page 32[1part b].)

the Pooling Layer:

The pooling layer is some kind of down sampling layer to reduce the number of parameters in the network. The pooling layer has no trainable parameters. It basically disassembles the input volume in 2×2 (spatial dimensions)^2 fragments and outputs the maximum number in every fragment (per input volume layer), if it is a max pooling layer. There is also a average pooling and a L2-norm pooling, which I don't use in my own approach, but the official ResNet uses a average pooling layer at the end, because the max pooling does distort the hight of the values in the input volume, which isn't helpful for the fully connected part of the network. But the max pooling is in fact helpful for the convolutional layers with regards to the ReLU part of the neurons, because there is a higher activation value which triggers the right neurons, which would

Figure 3.18: Example sentences generated by multimodal RNN [KF15, Page 7, Fig. 6.]

a woman riding a horse on a dirt road | an airplane is parked on the tarmac at an airport | a group of people standing on top of a beach

Figure 3.19: unlikely events for a Neural Network. from [GDDM13, Page 26, Fig. 6.]

"Perceiving scenes without intuitive physics, intuitive psychology, compositionality, and causality. Image captions are generated by a deep Neural Network [KF15]
Image credits: Gabriel Villena Fern'andez (left), TVBS Taiwan, Agence France-Presse (middle) and AP Photo , Dave Martin (right)." [from the same paper]

[1part b]In figure 3.18 we can see, that this image labeling RNN can in fact cover most of the situations nearly perfectly but the very rare are a big problem (see figure 3.19). I assume, that we humans will at least demand an AI of a 10 times lower failure quota than a human failure quota while driving before we let them drive on their own. And since that every very rare incidence will delay the day AI takes over driving for a respective amount of humans, it could take decades until autonomous driving is a real thing. And this somewhat makes sense, because some very rare incidence will happen when there are enough autonomous cars which drives frequently and everybody who experienced a situation, where he was forced to reconsider his reactions in a very short time, knows that he has to act differently to prevent a fatal mistake. A normal labeling RNN can just react at its best. Recognizing the out of the ordinary and than act properly isn't that easy, since everything is theoretical possible or so they say.

the original not down sampled input also have done. The risk by an average pooling is, that the main features can be lost due to a average to mean effect. So every time a transfer function like a ReLU unit follows a pooling layer at some point in the network it is recommended to use max pooling instead of average pooling.
The other idea behind a pooling layer is, that the relative location to other features is more important than its exact location. The network has now 75% less parameters to worry about without loosing the main information of the input volume.
Pooling can also be a prevention method for overfitting which we will se in chapter [4.2.1].

the AlexNet:

The AlexNet was the first CNN, which successfully performed an image classification with groundbreaking results and this led to the research for CNN on a wide base. There where some previous related works in this interrelationship, especially Yann LeCun's paper [LBD+90], but they were to a greater extent of a theoretical nature. The AlexNet however was used to win the 2012 ImageNet Large Scale Visual Recognition Challenge (ILSVRC), which is something like the Olympics of computer vision, with different competitions such as classification, localization and detection. But the AlexNet didn't just enable the use of deep learning for image classification, it achieved a top 5 test error rate of 15.4% in 1000 possible object classes. The next best entry achieved a top 5 error rate of 26.2%, which is a surprising change, considering that before this improvement there was just a very slow improvement for a long time. The AlexNet is at its core pretty much like the convolutional network i described above. It has two parallel network paths, though, because this way the network was able to parallelize more of the calculations on two graphics cards, but there are crossovers between the two parallel paths at some points in the network. The network has 5 convolutional layers, max pooling layers, dropout layers and 3 fully connected layers, where all this layers are split up on those two network paths except for the last fully connected layer, which is the last layer of the network.

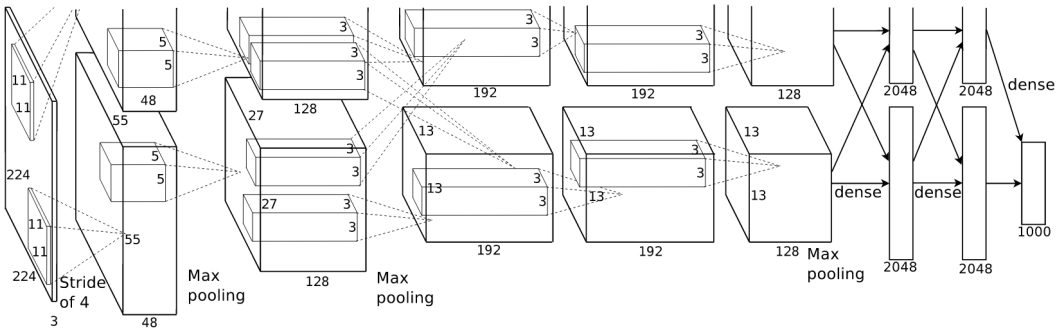


Figure 3.21: model AlexNet [KSH12, Page 2, Fig. 2.]

The network uses ReLU nonlinearity functions, like all other convolutional networks I described below. ReLU decrease the training time in comparison to the tanh transfer function. The network uses also dropout layers in order to combat the problem of overfitting. The network implemented data augmentation techniques like image translations, horizontal reflections, and patch extractions. For training the network uses SGD with a training time of five to six days on two GTX 580 GPUs to achieve the results listed above

Since the AlexNet performed so well on the historically difficult ImageNet dataset and kinda started the whole development in this area, it has been cited over 6,184 ti mes and is acknowledged as one of the most influential publications in the field.

the VGG Net:

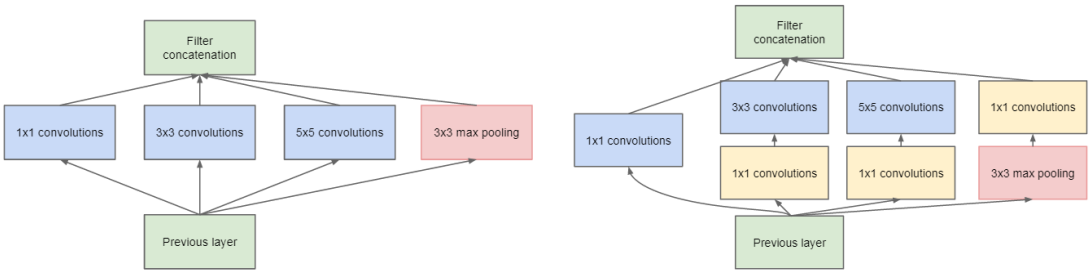
The plain but quantitative augmented shot:

Karen Simonyan and Andrew Zisserman created a 19 layer CNN that simply uses 3×3 filters with stride and pad of 1, along with 2×2 maxpooling layers with stride of 2, which is close to the ResNet's and the VResNet. They achieved an error rate of 7,4% on the ILSVRC 2014, but weren't the winners. In comparison to the AlexNet, which uses 11×11 filters in the first layer and other networks before the VGG Net which often uses 7×7 filters, this is a is quite different way to improve the feature recognition. The authors' intention is that the combination of two 3×3 convolutional layers in series has an effective receptive field of 5×5 pixels, while keeping the number of parameters lower than using one 5×5 filter. Three in a row would add up to 7×7 effective receptive field, but smaller features would be filtered more often this way, which improves the feature recognition of features with less pixels in the input volumes of a layer in comparison to one 7×7 filter. Unlike the AlexNet, the convolutional and the pooling layers decreases the spatial size of the input volumes at each layer and with every added filter at each layer the depth of the volumes increases. This is also similar to the ResNet. The model was built with the Caffe toolbox, which is a c++ based deep learning framework and it was trained two to three weeks on 4 Nvidia Titan Black GPUs. The parallelization of two network paths like in the AlexNet was not necessary due to the use of the Caffe toolbox which is able to parallelize like torch7 the mini-batches.

the GoogLeNet:

The GoogLeNet is a very successful and complex CNN. I won't go too much into detail about this network because it didn't have much in common with this thesis approach, But I will outline its main features:
The GoogLeNet is a 22 layer CNN and won the ILSVRC 2014 with a top 5 error rate of 6.7%. It was one of the first CNNs that give up the simple linear approach of stacking the network's layers on top of each other. It consists out of a stack of inception modules, that look like this figure 3.22 and some other parts. It has the big advantage over the VGG Net, that it needs much less neurons, GPU time and storage due to the smart layout of the network modules.

[2] there are of course other possible fragment sizes than 2×2, but 2×2 already decreases the amount of information to handle by the network to 25%, which is for most of the applications enough.



(a) Inception module, naïve version (b) Inception module with dimension reductions

Figure 3.22: inception modules [SLJ+14, Page 5, Fig. 2.]

the ResNet:

For this thesis approach I choose the ResNet, because it achieved with the GoogLeNet one of the highest scores in object recognition in the recent years, but has a more straightforward design than the GoogleNet has, which is simpler to reverse for the deconvolutional part of the network. The ResNet is comparable to the VGG Net, but it comes with a clever tweak. Some layers are skipped by additional shortcuts, which breaks with the VGG Net concept of a true 100% linear network, but improves the SGD adjustment rate of the front layers during the backpropagation drastically. This is on the one hand a great improvement for such a relatively small enhancement, but on the other hand this slightly change has a huge impact on the inner processes of the network and made the network nearly impossible for me to revert for the deconvolutional part of the network, which we will see in chapter [4]. The case would be much worse with the use of the GoogLeNet, so I didn't made the worst choice using the ResNet, because all newer CNNs uses nonlinear and more complex structures and I wasn't aware of this problematic before choosing the network architecture. But I found a way to handle those nonlinearities and was finally able to perform a deconvolution on a ResNet.
Since I implemented the ResNet in this thesis approach I describe the ResNet in the next chapter so here just some key data to outline the ResNet:

- Up to 152 layers.
- 3,6% error rate on the same dataset the GoogLeNet used above.
- No dropout layers.
- Just two pooling layers. One at the beginning one at the end.
- Was the first network that beats humans on object recognitions tasks at a regular bases.
- Just one fully connected layer at the end.

3.1.4 Deconvolution

Deconvolutional Layer

Deconvolution, upconvolution or full convolution describes in the context of Neural Networks a deep learning application, that takes scored mini-batch layers from a CNN as an input and outputs the scored feature location on an reassembled original input

image of the CNN. The deconvolution is an operation that inverts the forward and backward information flow by inverting every layer in the network and than put the layers back together in a reverse order. Through this operation it is possible to extract spatially dimension information of each recognized feature on an input image. This process can be used without training, but a supervised training of the deconvolutional layers is also possible, which was showed in [ZF13].
The thesis approach only uses an untrained version of the deconvolution as a purely information extraction operation. I explain the deconvolutional Neural Network as a whole in chapter [4.1.2]; In this chapter I will explain how one deconvolutional layer is formed out of a convolutional layer and how to invert batch normalization-, pooling- and ReLU layers.

The inversion of the convolutional layer is as simple as transposing the kernel filters of the convolution and convolve with it the output of the ReLU layer, which was the subsequent layer of the convolution before the inversion. The settings for padding and the stride sizes need some updates, too. If use a padding before, we now have to make sure its effect is correctly inverted to prevent checker board artifact effects, which means if a layer decreases the rim of an image we have to make sure to increase it in the same way again. The stride size can be left untouched in most cases, but some combinations of stride sizes and filter sizes cause some inference patterns as well.

What a good combination of stride, filter and padding sizes for a deconvolution is, is a science in itself and can be found here [ODO16] and here [DV16].

Inverted Batch Normalization

The inversion of the batch normalization is as simple as using the same batch normalization with the same trained weights again. Because if the network didn't use the batch normalization it won't use it this way in the inverted network and if there was an internal covariate shift in the output of the previews convolutional layer, the inverted version of that layer won't have problems to recreate its previews input to reassemble as good as possible original image with a normalized version of its original output, because the con- and deconvolutional operation is invariant to a pure shifting operation, because it primarily detects edges and other patterns, which remain the same if shifted. The ReLU layer however reacts differently to shifted values, but the first layer after an inverted batch normalization is the inverted convolution in a deconvolutional network.

Unpooling

The unpooling is more complex and has an specific function in torch7, which is based on this approach:
"Unpooling: In the convnet, the max pooling operation is non-invertible, however we can obtain an approximate inverse by recording the locations of the maxima within each pooling region in a set of switch variables. In the deconvnet, the unpooling operation uses these switches to place the reconstructions from the layer above into appropriate locations, preserving the structure of the stimulus." [ZF13, page 2]. See figure 3.a. for an illustration of this procedure.

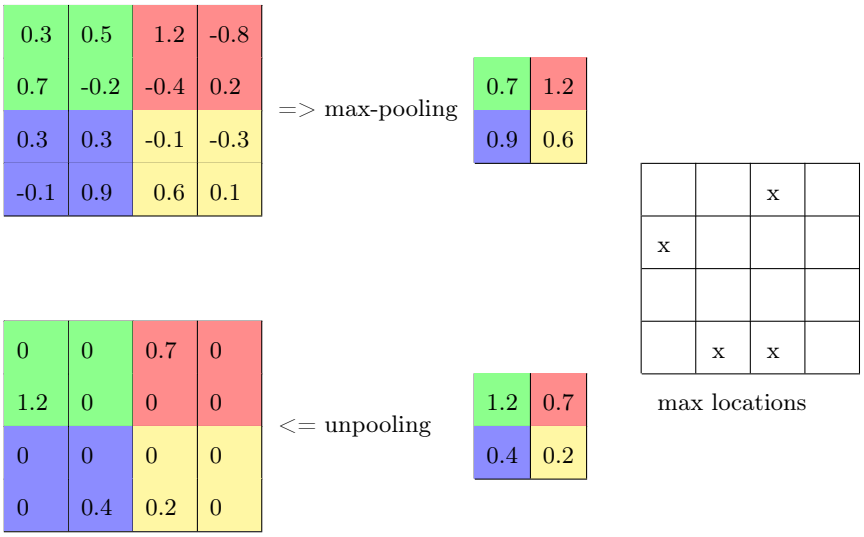


Figure 3.a: unpooling, region 2×2, stride 2×2

It is important to point out, that the values for unpooling might have changed trough the other layers of the deconvolution process and are not the values that were pooled down before. The only thing that is preserved is the original location of the maximum values of the pooling operation. In this way location heat maps from recognized object features can be traced back more precisely to there original pixel location source, which can be seen in figure 2.1.

Inverted ReLU Layer

The inversion of the ReLU Layer is a mathematical inversion of the transfer function. Which is in the case for a ReLU Laye exactly the same for the positive domain of the function. The negative domain of the function should be set to zero so that no negative values lead to an activation. In this way we achieve a normal ReLU Layer like we had before in the convolutional part of the network.

3.2 Unwanted Effects in Neural Networks

3.2.1 Overfitting

Deep Learning in the Literal Sense:

The "deep" in deep learning comes at least in two flavors. First it means that deep Neural Networks with at least one hidden layer are involved. This is also why we called them "deep" in the first place because they have got hidden layers.
The second meaning is the crucial one for this chapter:
Deep Neural Networks have proven, that they are capable of recognizing known patterns in complex and unknown data, which i will call from now on *"learning in deed"* or *"generalizing"*. That contrasts with simple memorizing or *"learning by heart"*, which is a basic function of every computerized system anyway and therefore not the objective of deep learning.
But by contrast with humans they need thousands of sample data, considering humans are in best cases able to recognizing a pattern with just one description of a vague sample. But I leave here the realm of well grounded research. There are some discussions

about whether humans actually do recognize patterns very similar to deep Neural Networks and for example need to render thousands of images to recognize the patterns of them on new images. When a child sees a specific object starting from its fourth age to its tenth age from time to time so that is has watched this object about 10 hour overall it would already has processed a million of images of this class of object in its brain and so it is able to distinguish its special features from other object classes' features. Probable most of the processing has to been made to equalize the perception of different receptors on the child's retina which records and blend together the different view angles of the same pattern to form one sound and vision independent[1] understanding of this pattern. The same is true to Neural Networks. If we could give the network more universal image data which wouldn't rely on pixel position and angle the network would perhaps just need a couple of images to recognize a class. The other theory to debate is why and whether humans can elevate abstract data to more applicable data to fulfill a task. The big advantage humans got here is that humans can think consciously and can analyze a task selectively. For that we need at least a recurrent Neural Network, which is a whole other league of artificial intelligence, we don't really talk about in this thesis.

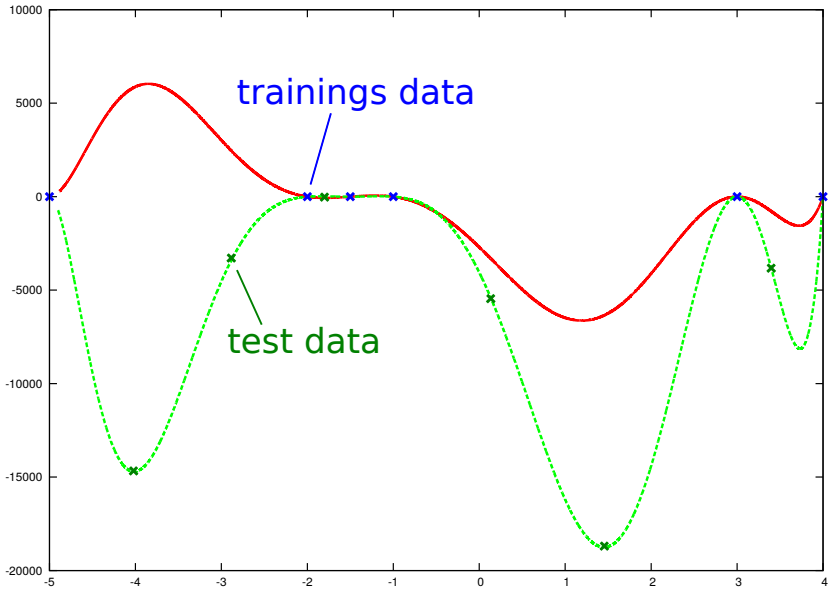


Figure 3.24: overfitting

But recognizing known patterns in complex and unknown data is a big step from learning by heart and simple repeating of knowledge towards an artificial entity we would call intelligent by todays norm. And this is exactly what the overfitting problem is about. Because just because an entity (either human or artificial) is able of learning in deep rather than learning by heart doesn't mean that it would always do that automatically. In fact, if we train or design a deep Neural Network badly we can loose this great achievement, and the network just learned by heart and didn't generalized its new learned data to something someone would call knowledge. If a potentially able of learning in deep algorithm does learn by heart its called overfitting.

[1] view angle, image position on retina and other vision dependent influences

the Overfitting Effect:

The effect overfitting also called *"leveling off"* describes mathematically, that the prediction graph (red line in figure 3.24) of the network goes through most of the correct points for the trainings samples (blue points) but deviates drastically from the correct points for the testing samples (green points) . So we should always keep an eye on this deviation between the scores of training and testing. But its not always a problem when the network starts overfitting. When the network approximates its maximum testing score it is at some point no matter when it begins to do overfitting. Because at some point this is the only thing the network is capable of learning additionally and it will most likely tend to do so. In this case it doesn't really do overfitting because its learned as much as it could in deep and the rest by heart. The deviation between the two scores shouldn't that big either. Since the trainings score will most likely hit a maximum value of near 100% at some point anyway, the trainings score is at any given epoch of training more like a scale for how much the network will be able to learn in deep additionally at best in the remaining training epochs, because if at any point the training score is near 100% it neither won't be able to improve its testing score.
But how to prevent overfitting?
There are two sources which increase overfitting. One lays inside the trainings process the other in the design of the network itself.
Optimizing the training process to prevent overfitting as good as possible:
You should try to provide as much training samples as possible for the network. As more samples you provide as more the network will learn in deep as a side effect even if it tends to do overfitting as well. To get a feeling for how much training samples you need, you can set up a smaller program of training with fewer classes to detect. Every additional class has to be told apart from each other. So doubling the number of different classes will quadruple the complexity of the problem for the network; in average, though. Because you can add pretty hard to differentiate classes.
The next thing you can do is shuffling the training examples, if the network is already updating its weight parameters during one training cycle. The Information Theory states - *"Learning that an unlikely event has occurred is more informative than learning that a likely event has occurred"* and if an unlikely events happening multiple times in a row the network couldn't value the importance of this unlikely event correctly.
Another important influence to consider is the right difficulty of a training dataset. A dataset that is too easy wouldn't use the full potential of a network and on a too difficult dataset the network wouldn't learn nothing at all. But if the dataset contains the right mix of difficulty levels the network is able to learn in multiple training iterations one step after another. A good learning curve can lead to better results. If a network already receives a better score by detecting some minor score relevant but easy to detect features it can at least simplify the dimensional complexity of the whole classification problem. After a some epochs of training the network then might be able to detect some score relevant features between a proper subset of all classes and not have to deal with all classes at once on one feature. In one of my experiments on the CIFAR-10 dataset the Neural Network first searched only for images with much blue on it. Later it separated the planes from the ships, which are the only images were the whole background is blue, by scanning the image for a horizon, which were in most cases only present on images of ships, because most of the planes are photographed from below. Im sure, if the network has from beginning scanned all the images for horizontal edges, it would have made confusions with the other 8 classes of the CIFAR-10 dataset.
The source for overfitting that lay inside the network design itself in the design of the

network itself, are many. The main important one is the flatness of the network itself. A deep and narrow network is not as vulnerable for overfitting as a wider and shallow one. As wider a Neural Network becomes as faster it can approximate a test dataset, but without understanding its deeper patterns. This leads to the effect, that it won't recognize features of features, that are important to master a test dataset with features, that are a variation of the original features, but the features of the features stayed the same. For example, a flat network looks for specific directed edges on an image but doesn't care about the arrangement of the edges. But in the test dataset all edges are rotated together around a certain point. the arrangement is still the same, but the flat network wouldn't recognize any of that rotated edge anymore.

3.2.2 Vanishing Gradient Problem

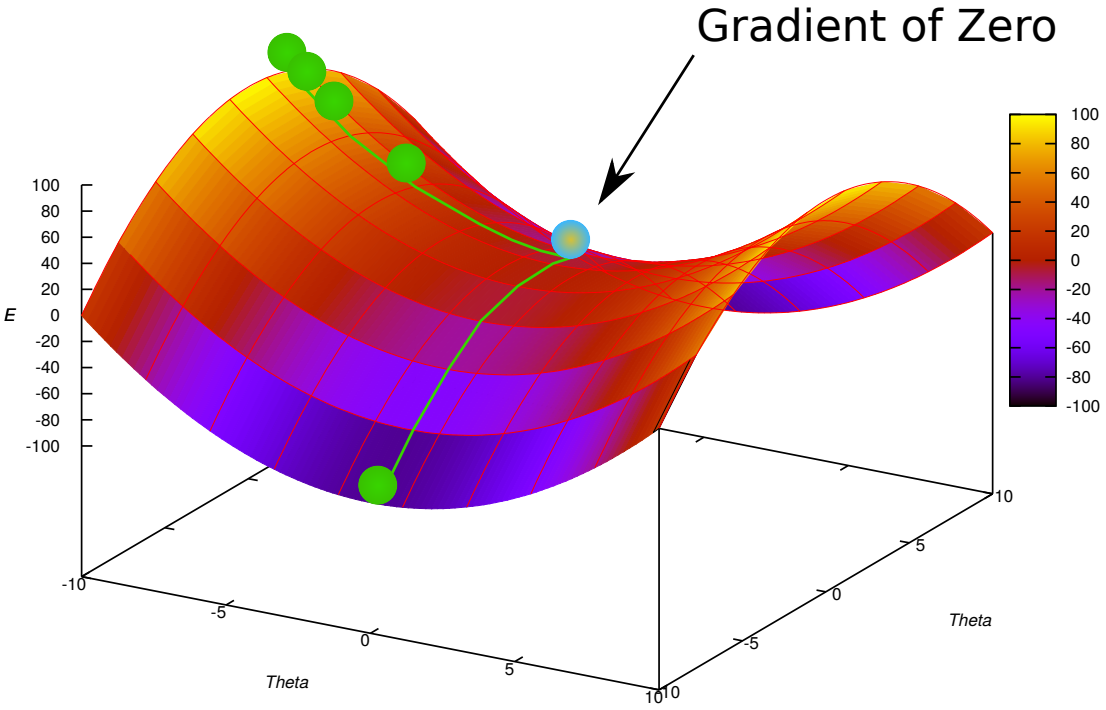


Figure 3.25: vanishing gradient problem

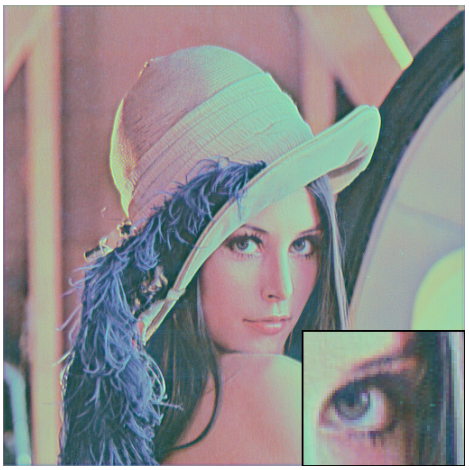
During the training process the network can get stuck in a condition that all or a part of the gradient parameters becoming zero, but we are either in a local- nor global minimum. In this case a vanilla version of the SGD algorithm isn't able to train the weight parameters of these gradient parameters any more. More on gradient parameters in chapter [3.3.3] and [3.3.4]. This effect can be prevented through dropout layers which leave out neurons, that have "dead" gradients (gradients that are zero). If all gradient parameters of a are dead neuron we talk about a dead neuron. There are other gradient descent training's methods, which gives a small momentum to the update vector, so that it didn't get stuck if it receives in one epoch of training a gradient of zero. Randomizing the gradients with a noise function with a very small amplitude could also solve this problem. After every epoch of training a gradient of zero would become a gradient of zero $\pm$ 0.01 or something like that, so it would get back to live after some training cycles. The problem about randomizing the weights is, that these small changes add up to big errors in the later layers of the network. There are many more solutions for

this problem, but I just used the one above.

3.2.3 Exploding Gradient Problem

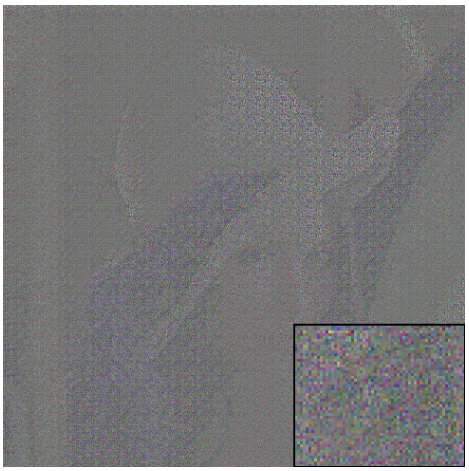
Exploding gradients means that the neural network is going to be unable to learn long term temporal relations, which is a problem that occur during the optimization of RNNs over many time steps, more on this in the book [GBC16, chapter: 10.11]. In our CNN networks, that are nor recurrent and have no understanding of temporal relations, this effect doesn't turn out to be as fatal as the vanishing gradient problem, but extreme high gradients are still problematic, because they are often caused by a very noisy domain of the error function and would result in a very high update vector, which leads to random update values in the weight parameters of the network.

(a) Lenna (input image)
image size 3×512×512



(b) convolution $\curvearrowright$ deconvolution :
kernel size = 3×3
stride = 1; padding = 1;
mini-batch transformations:
$3 \to 64 \to 128 \to 256 \curvearrowright 128 \to 64 \to 3$

(c) convolution $\curvearrowright$ deconvolution :
kernel size = 4×4
stride = 2; padding = 2;
mini-batch transformations:
$3 \to 64 \to 128 \to 256 \curvearrowright 128 \to 64 \to 3$



(d) convolution $\curvearrowright$ deconvolution :
kernel size = 3×3
stride = 3; padding = 1;
mini-batch transformations:
$3 \to 64 \to 128 \to 256 \curvearrowright 128 \to 64 \to 3$

Figure 3.23: inference pattern (checker board artifact)

3.3 Which Mathematical Methods are Known in Literature?

3.3.1 The Neuron

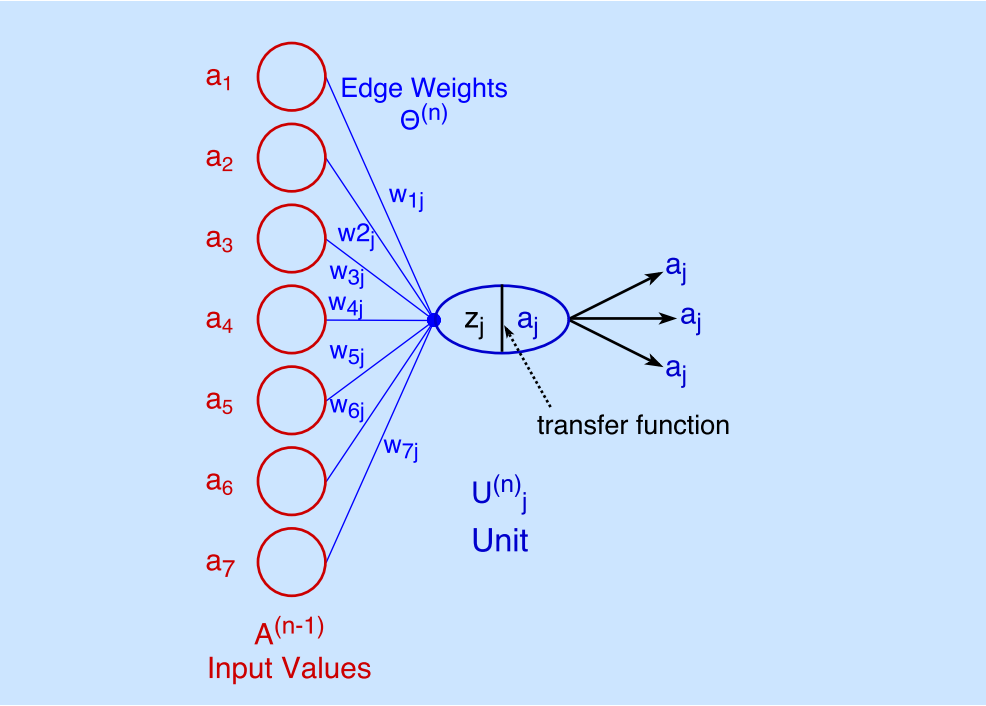


Figure 3.26: a single neuron

A single neuron (unit $U_j^{(n)}$) consists out of three types of elements mathematically speaking, where n is the index number (starting by 1) of the layer the neuron lies in and i the position (starting by 1) of that neuron in a layer. The input edges signal (value) $a_i^{(n-1)}$, the output edges value $a_j^{(n)}$ and the nucleus N, where the transfer function $f_N(z_j)$ is executed and j stands for an edge index going into the jth neuron in the reception layer and coming from the ith neuron in the transmission layer of a Neural Network. Though $a_i^{(n-1)}$ is the value of the input edge of an nth layer neuron, the signal a on an edge is always counted as the output of the transmitting unit. At this point we indeed look at single neurons, but let correctly define the i and j and n indices to be able to define each input edge of the neuron distinguishable. The input edges are weighted each by its own weight $w_{ij}^{(n)}$. The output edges receive all the same value from the nucleus and their weights are defined by the next neurons, which gets the output as receiving units, so that to each neuron belongs the weights of its input connections. A neuron is graph theoretically speaking a single node with two bundles directed edges on each side of the node and builds with itself and/or other neurons a directed and weighted graph, which we call a Neural Network. A neuron can additionally have a bias unit. This is a input that has a constant value. To keep things clearly represented we will neglect the bias of each neuron in this mathematical background chapter, because there influence on the derivations is very low. So if we illustrate a neuron we always have to consider where it gets it input stimulations (values) from and where goes its

output values to.
At first we execute a weighted sum $z_{ij}^{(n)}$ over all input values with the weights in the input edges of a neuron $U_j^{(n)}$ (in this example with a following ReLU transfer function as $f_N(z_j^{(n)})$):

$$\varepsilon_{ij}^{(n)} = a_i^{(n-1)} \cdot w_{ij}^{(n)} \tag{3.1}$$

$$z_j^{(n)} = \sum_i \varepsilon_{ij}^{(n)} = \sum_i (a_i^{(n-1)} \cdot w_{ij}^{(n)}) \tag{3.2}$$

$$a_j^{(n)} = f_N(z_j^{(n)}) = \max(0, z_j^{(n)}) = \begin{Bmatrix} 0 & | \ z_j^{(n)} \leq 0 \\ z_j^{(n)} & | \ z_j^{(n)} > 0 \end{Bmatrix} \tag{3.3}$$

After that the transmission unit becomes the reception unit for the next layer in the network and $a_j^{(n)}$ is passed to the next neuron in line or to the output of the network.

The whole process as one function would look like this:

$$a_j^{(n)} = f_N(z_j^{(n)}) = \max(0, \sum_i (a_i^{(n-1)} \cdot w_{ij}^{(n)})) \tag{3.4}$$

The forwardpropagation algorithm has i numbers of variables, i constants (the weights) and resolves j unknown. This is all about the forwardpropagation of a single neuron. We will later see how the backpropagation work where the weights are the unknown variables to resolve.

3.3.2 Forwardpropagation

In figure 3.3 we can see, that in a Feed Forward Network every neuron of one layer is connected to every neuron in the next layer. But there are no feedback loops or connections between two neurons within one layer. All connections in a Neural Network can be represented with one matrix, but this matrix would than have a lot of preset zero values (that can not be changed through the training process of the network), because no connection skips a layer. The red marked cells in table 3.1 are the red colored connection of figure 3.3 and the blue marked cells are the blue colored connections.
A Neural Network consists out of multiple neurons $U_j^{(n)}$ which can be connected in many different ways to achieve different purposes, but in this thesis we only use Neural Networks working with layers of neurons, where each layer is connected to the next in a series with directed connections and the whole network has no cycles and no connection within one layer. Such a network's forwardpropagation can be described as the following mathematical construct, where $\Theta^{(n)}$ is a Matrix filled with the weights w_{ij} of different connections between two layers, where n is the index of the receiving layer. $A^{(n)}$ is a vector filled with output values a_j of a whole layer, where n is the index of the sending layer. $\mathbf{x}$ or $A^{(0)}$ is a vector of input values for the whole network and specifically the

↙	x_1	x_2	x_3	x_4	x_5	x_6	$a_1^{(1)}$	$a_2^{(1)}$	$a_3^{(1)}$	$a_4^{(1)}$	$a_5^{(1)}$	$a_6^{(1)}$	$a_7^{(1)}$	$a_1^{(2)}$	$a_2^{(2)}$	$a_3^{(2)}$	$a_4^{(2)}$	$a_5^{(2)}$
x_1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
x_2	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
x_3	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
x_4	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
x_5	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
x_6	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
$a_1^{(1)}$							0	0	0	0	0	0	0	0	0	0	0	0
$a_2^{(1)}$							0	0	0	0	0	0	0	0	0	0	0	0
$a_3^{(1)}$							0	0	0	0	0	0	0	0	0	0	0	0
$a_4^{(1)}$							0	0	0	0	0	0	0	0	0	0	0	0
$a_5^{(1)}$							0	0	0	0	0	0	0	0	0	0	0	0
$a_6^{(1)}$							0	0	0	0	0	0	0	0	0	0	0	0
$a_7^{(1)}$							0	0	0	0	0	0	0	0	0	0	0	0
$a_1^{(2)}$	0	0	0	0	0	0								0	0	0	0	0
$a_2^{(2)}$	0	0	0	0	0	0								0	0	0	0	0
$a_3^{(2)}$	0	0	0	0	0	0								0	0	0	0	0
$a_4^{(2)}$	0	0	0	0	0	0								0	0	0	0	0
$a_5^{(2)}$	0	0	0	0	0	0								0	0	0	0	0

Table 3.1: weight matrix of the whole Neural Network of figure 3.3

first layer which is followed by the first hidden layer:

$$
\begin{aligned}
A^{(0)} &= \mathbf{x} && \text{(I.)}\\
A^{(1)} &= \Theta^{(1)} \cdot A^{(0)} \\
A^{(2)} &= \Theta^{(2)} \cdot A^{(1)} && (3.5)\\
A^{(3)} &= \Theta^{(3)} \cdot A^{(2)}
\end{aligned}
$$

... until n is the last layer ℓ, $\mathrm{A}^{(\ell)}$ would be the output of the last layer

Every forwardpropagation from one to the next layer can be Universally formulated as this Matrix Vector product:

$$A^{(n)} = \Theta^{(n)} \cdot A^{(n-1)} \qquad \text{(II.1 a)}$$

This calculation however excludes the transfer function f_N of the nucleus, which can be added[2] between every layer $L^{(n)}$ and looks like this (where $A^{(n)}$ is the output vector and $Z^{(n)}$ the input vector (consisting out of $z_i^{(n)}$ elements) of the transfer function f_L,

[2]Theoretically every neuron has a transfer function in the nucleus, but the function can be a $f_N(x) = x$ function, which doesn't change the input at all. But practically the transfer function can be left away between some layers in software solutions, which we will see later in chapter [4.1].

which is now a $f\colon \mathbb{R}^n \to \mathbb{R}^n$ vector-valued function):

If we use a f_L we have to change function II.1a to :

$$Z^{(n)} = \Theta^{(n)} \cdot A^{(n-1)} \tag{II.1}$$

$$A^{(n)} = f_L[Z^{(n)}] \tag{II.2}$$

in case of a ReLU$f_L\colon A^{(n)} = \max[0, Z^{(n)}]$ where the max function is applied to each vector value separately. (II.2b)

from II.1 and II.2 follows that: $A^{(n)} = f_L[\Theta^{(n)} \cdot A^{(n-1)}]$ (3.6)

3.3.3 Backpropagation

Most of the derivations in this subchapter [3.3.3] were made by Alfredo Canziani and are based on known related works like [Roj96]. His derivations are featured in his YouTube video "Practical 2.1 – NN backward", which is part of his "Torch Video Tutorial" series, which I can highly recommend for educational purposes about Neural Networks and Torch7. Alfredo Canziani has granted me permission to use his derivations in my thesis.

The backpropagation algorithm is the functionality of the Neural Network where the magic happens. But at first we have to talk about loss functions and labels. Since this is all about supervised learning, the network is just able to learn something when it is able to compare its own output to a correctly labeled result of the training set. The deviation between the network predicted results $h_\Theta[\mathbf{x}_k]$ and the labeled results $\mathbf{y}_k$ is called loss, where $\mathbf{x}_k$ is the kth network's input vector of the whole training's set $\mathbf{X}$ and $\mathbf{y}_k$ is the kth vector of the corresponding training's label $\mathbf{Y}$. The loss function $\mathcal{L}(\Theta)$ is just depending[3] on the values of all the variable weights Θ in the neural network and is a summation of all errors E_k in each individual training's sample.

$$\mathcal{L}(\Theta) = \frac{1}{l} \sum_{k}^{l} \mathrm{E}_k \tag{3.7}$$

The error of each training sample is calculated by this:

$$\mathrm{E}_k = f_E[h_\Theta[\mathbf{x}_k]] \tag{3.8}$$

at this point I won't keep track of the index k any longer because we will look on the error function for every trainings sample in general, but we will from now on use the index i instead for describing an specific element in the vector y_j, the vector x_j and $a_j^{(n)}$, where n is the layer from where the output emerge. I will especially use $\hat{a}_j$ for an element in the output vector of the last layer of the network, so that $\hat{a}_j = [h_\Theta[\mathbf{x}]]_j$ applies, when this is a notation of an element of an vector-valued function: $[f[\mathbf{x}]]_j$.

$$f_E[h_\Theta[\mathbf{x}]] = \frac{1}{2} \cdot \left\| y - h_\Theta[\mathbf{x}] \right\|^2 = \frac{1}{2} \cdot \sum_j (y_j - \hat{a}_j)^2 \tag{3.9}$$

[3] Of course it also depends on the training samples, but since the network isn't able to change them it can just change the loss by changing its weights. So the training samples are assumed to be constants during the backpropagation.

With the knowledge about the network's error we can try to change the $\mathbf{\Theta}$ value slightly, so that we lower the network's error of a training sample. This slightly change can be represented with a partially derivation for the direction of the change and a value η for the magnitude of the change:

$$\mathbf{\Theta} \rightarrow \mathbf{\Theta} - \eta \frac{\partial \mathrm{E}}{\partial \mathbf{\Theta}} \tag{3.10}$$

The problem is, that the $\mathbf{\Theta}$ variables we want to derivate in our function $f_E[h_\Theta(\mathbf{x})]$ are yet encapsulated inside a series of other functions, so that we have to make some other derivations and transformations at first:
A slightly change in the weighted input of the transfer function would result in a change of the error:

$$z_j^{(n)} \rightarrow z_j^{(n)} + \Delta z_j^{(n)} \tag{3.11}$$

$$\Rightarrow \quad E \rightarrow E + \frac{\partial \mathrm{E}}{\partial z_j^{(n)}} \cdot \Delta z_j^{(n)} \tag{3.12}$$

Let us define a variable $\delta_j^{(n)}$ for the derivation over the error with respect to the weighted input $z_j^{(n)}$ in function 3.12, where $\hat{\delta}_j$ would be the "δ" with respect to the weighted input $\hat{z}_j$ of the transfer function of the last layer:

$$\delta_j^{(n)} := \frac{\partial \mathrm{E}}{\partial z_j^{(n)}} \tag{3.13}$$

$$\hat{\delta}_j = \frac{\partial \mathrm{E}}{\partial \hat{a}_j} \cdot \frac{\partial \hat{a}_j}{\partial \hat{z}_j} \tag{3.14}$$

In function 3.14 we made use of the chain rule to compute the derivative of the composition of two functions and do now derivate over the error with respect to the output of the transfer function. With doing so it emerges another derivation over the output of the transfer function with respect to our weighted input $\hat{z}_j$.
We can replace the second derivation over $\hat{a}_j$ with a derivation over the transfer function, preconditioned that $f_L(\hat{z}_j)$ is fully differentiable (in this case our ReLU transfer function):

$$\hat{\delta}_j = \frac{\partial \mathrm{E}}{\partial \hat{a}_j} \cdot \frac{\partial \hat{a}_j}{\partial \hat{z}_j} = \frac{\partial \mathrm{E}}{\partial \hat{a}_j} \cdot f_L'(\hat{z}_j) \tag{3.15}$$

$$\text{for ReLU}: \Rightarrow \frac{\partial \mathrm{E}}{\partial \hat{a}_j} \cdot \mathrm{H}(\hat{z}_j)) = \begin{Bmatrix} 0 & |\hat{z}_j \leq 0 \\ 1 & |\hat{z}_j > 0 \end{Bmatrix} \tag{3.15b}$$

$$\Rightarrow \hat{\delta}_j = \begin{Bmatrix} 0 & |\hat{z}_j \leq 0 \\ \frac{\partial \mathrm{E}}{\partial \hat{a}_j} & |\hat{z}_j > 0 \end{Bmatrix} \tag{3.15c}$$

Where $\mathrm{H}(x)$ in function 3.15b is the Heaviside step function.
We can express the $\hat{\delta}_j$ in the equation 3.15 as a vector $\hat{\mathfrak{d}}$ for the whole output error of

the network:

$$\hat{\mathfrak{d}} = \underbrace{\nabla E(A^{(\ell)})}_{A^{(\ell)}-\mathbf{y}} \odot f_L'[Z^{(\ell)}] \tag{III.}$$

$$\text{for ReLU}: \Rightarrow \hat{\mathfrak{d}} = A^{(\ell)} - \mathbf{y} \odot \underbrace{\begin{pmatrix} \hat{z}_1 \leq 0\ ? & 0:1 \\ \hat{z}_2 \leq 0\ ? & 0:1 \\ \vdots & \\ \hat{z}_n \leq 0\ ? & 0:1 \end{pmatrix}}_{\mathrm{H}[Z^{(\ell)}]} \tag{III.b}$$

we get a vector $\hat{\mathfrak{d}}$ with elements like this : $\hat{a}_1 - y_1$ or 0 depending on $\hat{z}_1 > 0$ (III.c)

III. is a component wise multiplication of the derivation of the transfer function and a component wise difference between the "netoutput" $A^{(\ell)}$ and the label $\mathbf{y}$, because $A^{(\ell)} - \mathbf{y}$ equates the gradient ∇ of E with respect to $A^{(\ell)}$.
We will later see how we use the "*local error*" $\mathfrak{d}^{(n)}$ to update the weights $\Theta^{(n)}$ of the network. But at first we have to determine $\mathfrak{d}^{(n)}$ of the other layers by backpropagating the local error back through the network, where $f_{\mathfrak{d}^{(n)}}$ is a function, which resolves the local error from one layer dependent on the local error of the next layer.

$$\mathfrak{d}^{(n)} = f_{\mathfrak{d}^{(n)}}[\mathfrak{d}^{(n+1)}] \tag{3.16}$$

$$\text{def. 3.13:}\ \ \delta_i^{(n)} = \frac{\partial \mathrm{E}}{\partial z_i^{(n)}} \qquad \Rightarrow \delta_j^{(n+1)} = \frac{\partial \mathrm{E}}{\partial z_j^{(n+1)}} \tag{3.17}$$

$$\overset{(\mathtt{J}\mathtt{c}f)}{\Rightarrow}\ \ \delta_i^{(n)} = \sum_j \left(\frac{\partial \mathrm{E}}{\partial z_j^{(n+1)}} \cdot \frac{\partial z_j^{(n+1)}}{\partial z_i^{(n)}} \right) \tag{3.18}$$

$(\mathtt{J}\mathtt{c}f)$ is the Jacobian of a Composition function, which we used in step 3.18 to express $\delta_i^{(n)}$ dependent on a ∂z_j of the next layer, which is defined like that (where $\mathtt{J}_f$ is the Jacobian-Matrix of a variable or vector):

$$f\colon \mathbb{R}^n \to \mathbb{R} \text{ differentiable in } \mathbf{x}_0 = g[u_0], \quad g\colon \mathbb{R} \to \mathbb{R}^2 \text{ differentiable in } u_0 \tag{3.19}$$

$$h = f \circ g\colon \mathbb{R} \to \mathbb{R} \text{ derivation in } u_0\colon \tag{3.20}$$

$$h'(u_0) = \mathtt{J}_f[\mathbf{x}_0] \cdot \mathtt{J}_g(u_0) \tag{3.21}$$

$$= \begin{bmatrix} \frac{\partial f}{\partial([\mathbf{x}_0]_1)}[\mathbf{x}_0] & \cdots & \frac{\partial f}{\partial([\mathbf{x}_0]_n)}[\mathbf{x}_0] \end{bmatrix} \cdot \begin{bmatrix} g_1'(u_0) \\ \vdots \\ g_n'(u_0) \end{bmatrix} \tag{3.22}$$

$$= \langle\ \nabla f[\mathbf{x}_0]\ ,\ g'(u_0)\ \rangle \text{ which is a scalar product} \tag{3.23}$$

Back to our equation in 3.18:

$$\delta_i^{(n)} = \sum_j \left(\frac{\partial \mathrm{E}}{\partial z_j^{(n+1)}} \cdot \frac{\partial z_j^{(n+1)}}{\partial z_i^{(n)}} \right) \overset{(3.17)}{=} \sum_j \left(\delta_j^{(n+1)} \cdot \underbrace{\frac{\partial z_j^{(n+1)}}{\partial z_i^{(n)}}}_{\text{to resolve}} \right) \tag{3.24}$$

this time we look at two layers at the same time, where j is the index for a neuron at the $(n+1)$th layer and i for a neuron at the nth layer.

$$z_j^{(n+1)} \overset{(3.2)}{=} \sum_i \left(a_i^{(n)} \cdot w_{ij}^{(n+1)} \right) \overset{(3.3)}{=} \sum_i \left(f_N(z_i^{(n)}) \cdot w_{ij}^{(n+1)} \right) \tag{3.25}$$

$$\Rightarrow \frac{\partial z_j^{(n+1)}}{\partial z_i^{(n)}} = w_{ij}^{(n+1)} \cdot f_N'(z_i^{(n)}) \tag{3.26}$$

Back to our equation in 3.24:

$$\delta_i^{(n)} = \sum_j \left(\delta_j^{(n+1)} \cdot \frac{\partial z_j^{(n+1)}}{\partial z_i^{(n)}} \right) \overset{(3.26)}{=} \sum_j \left(w_{ij}^{(n+1)} \cdot \delta_j^{(n+1)} \cdot f_N'(z_i^{(n)}) \right) \tag{3.27}$$

The vectorial form of the equation would look like this (remember $\Theta^{(n)}$ is a matrix of $w_{ij}^{(n)}$ weight elements):

$$\mathfrak{d}^{(n)} = f_{\mathfrak{d}^{(n)}}[\mathfrak{d}^{(n+1)}] = \left[\left(\underbrace{\Theta^{(n+1)}}_{\substack{\text{previous} \\ \text{epoch of} \\ \text{training}}} \right)^{\mathrm{T}} \cdot \underbrace{\mathfrak{d}^{(n+1)}}_{\substack{\text{back–} \\ \text{propa–} \\ \text{gation}}} \right] \odot f_L'[\underbrace{Z^{(n)}}_{\substack{\text{forward–} \\ \text{propa–} \\ \text{gation}}}] \tag{IV.}$$

To come back to our problem of the weight $\boldsymbol{\Theta}$ update in 3.10 we use another chain rule:

$$\frac{\partial \mathrm{E}}{\partial w_{ij}^{(n)}} = \frac{\partial \mathrm{E}}{\partial z_j^{(n)}} \cdot \frac{\partial z_j^{(n)}}{\partial w_{ij}^{(n)}} \qquad , \forall\, i, j \tag{3.28}$$

$$= \underbrace{\delta_j^{(n)}}_{\text{back-/forward pass}} \cdot \underbrace{a_i^{(n-1)}}_{\text{pass}} \tag{3.29}$$

The vectorial form of the equation would look like this:

$$\frac{\partial \mathrm{E}}{\partial \Theta^{(n)}} = \underbrace{A^{(n-1)} \cdot \left(\mathfrak{d}^{(n)} \right)^{\mathrm{T}}}_{\textbf{Matrix}} \tag{V.}$$

to summarize all necessary steps that are necessary to perform the backpropagation:

$$
\begin{aligned}
&\text{(I.)}: && A^{(0)} = \mathbf{x} \\
&\text{(II.1)}: && Z^{(n)} = \Theta^{(n)} \cdot A^{(n-1)} \qquad \text{(II.2)}: \quad A^{(n)} = f_L[Z^{(n)}] \\
&\text{(III.)}: && \hat{\mathfrak{d}} = \nabla E(A^{(\ell)}) \odot f_L'[Z^{(\ell)}] \\
&\text{(IV.)}: && \mathfrak{d}^{(n)} = \left[\left(\Theta^{(n+1)}\right)^{\mathrm{T}} \cdot \mathfrak{d}^{(n+1)}\right] \odot f_L'[Z^{(n)}] \\
&\text{(V.)}: && \frac{\partial \mathrm{E}}{\partial \Theta^{(n)}} = A^{(n-1)} \cdot \left(\mathfrak{d}^{(n)}\right)^{\mathrm{T}}
\end{aligned}
$$

step (I.) to pass the networks input to the first layer.
step (II.) n times, to perform the forwardpropagation.
step (III.) to calculate the local error $\hat{\mathfrak{d}}$ of the last layer (output error).
step (IV.) n times, to propagate the local error $\mathfrak{d}^{(n)}$ back to all layers.
step (V.) n times, to calculate the gradient of the error in all layers.

in torch7 the gradient over the error with respect to the weight of an edge is called `"gradWeight"` and is saved along with the weights of the edges in the module tensors of the layers.

3.3.4 SGD

The Stochastic Gradient Descent updates the weight matrix $\Theta^{(n)}$ by simply multiplying the learning rate η with the gradient of the error $\frac{\partial \mathrm{E}}{\partial \Theta^{(n)}}$ and subtracting this from the current weight matrix $\Theta^{(n)}$:

$$
3.10 : \boldsymbol{\Theta} \rightarrow \boldsymbol{\Theta} - \eta \frac{\partial \mathrm{E}}{\partial \boldsymbol{\Theta}} \qquad \Rightarrow \Theta^{(n)} \rightarrow \Theta^{(n)} - \eta \frac{\partial \mathrm{E}}{\partial \Theta^{(n)}} \tag{3.30}
$$

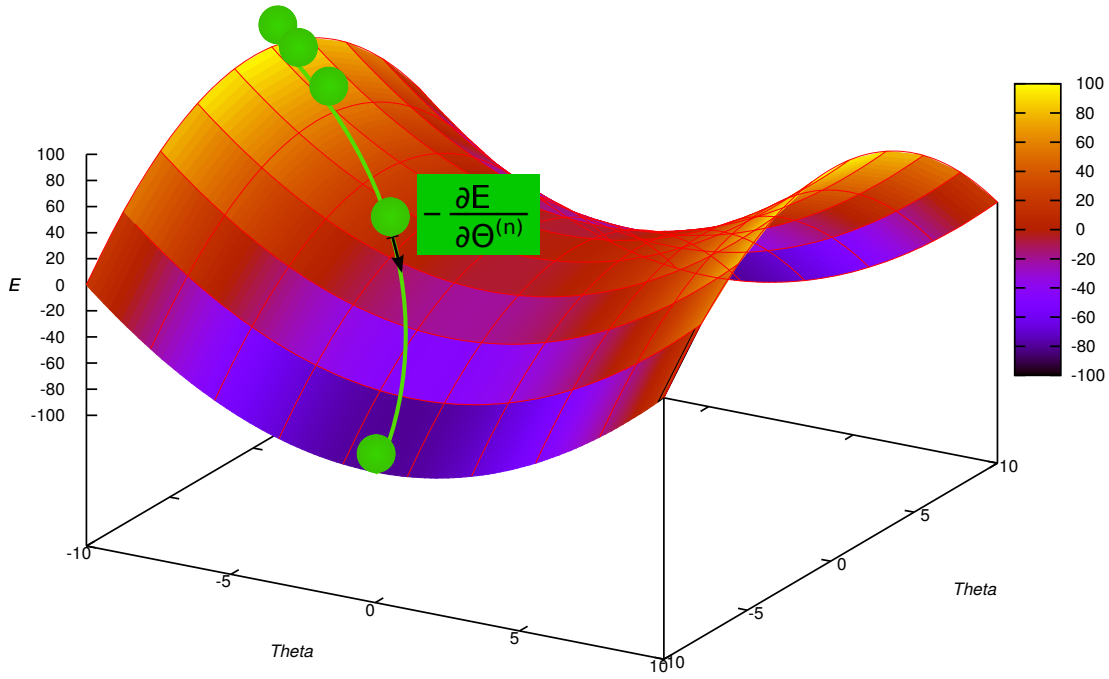


Figure 3.27: SGD illustration

You can see in Figure 3.27, that over multiple training's epochs the error is lowered by adjusting the weights $w_{ij}^{(n)}$ in $\Theta(n)$ in opposite direction of the gradient $\frac{\partial \mathrm{E}}{\partial \Theta^{(n)}}$ with a step size of the learning rate η, which corresponds to the length of the shown black vector arrow in this Figure. To be able to plot the SGD operation I simplified the dimension of Theta to two dimensions.
Since we want to use the whole trainings set $\mathbf{Y}$ and not just one training's sample $\mathbf{y}_k$ for the weight update, we have to calculate an average direction for the change in $\Theta^{(n)}$ of l numbers of training's samples:
Batch (or mini-Batch) Gradient Descent:

$$\Theta^{(n)} \rightarrow \Theta^{(n)} - \eta \, \frac{1}{l} \sum_{k=1}^{l} \left(\left[\frac{\partial \mathrm{E}}{\partial \Theta^{(n)}} \right]_k \right) \tag{3.31}$$

$\left[\frac{\partial \mathrm{E}}{\partial \Theta^{(n)}}\right]_k$ is the the gradient over the error E with respect to the weight $\Theta^{(n)}$ of the kth training's sample. After we have performed this algorithm over all layers of the network we have completed one epoch of training.

3.3.5 Convolution

Before I explain the mathematics about the convolutional Neural Networks, I will recap the mathematics about convolutions in image processing at first:
The convolution is an operation, where each pixel value of an image $\mathbf{I}$ is added to its local neighbors, weighted by the kernel Matrix $\mathbf{K}$. It should be noted, that the convolution is not a traditional matrix matrix multiplication, despite being often similarly denoted by the $*$ operator. I will use the $*$ operator for convolutions and the $\cdot$ operator for matrix

matrix, matrix vector and vector transposed vector multiplications.

$$\mathbf{I}^* = \sum_{v}^{q} \sum_{u}^{p} \Big(\mathbf{I}(x+u-\lceil p \rceil, y+v-\lceil q \rceil) \cdot \mathbf{K}(u,v) \Big) \quad , \forall\, q,p = \{2k+1 : k \in \mathbb{N}_0\}$$
$$\Leftrightarrow \; \mathbf{I}^* = \mathbf{I} * \mathbf{K} \tag{3.32}$$

The here used $\cdot$ operator is just a scalar scalar multiplication. $\mathbf{I}(x,y)$ and $\mathbf{K}(i,j)$ denote a scalar element i_{xy} and k_{uv} by the matrices $\mathbf{I}$ and $\mathbf{K}$. q and p have to be odd numbers, so that the kernel has a center at $(\lceil p \rceil, \lceil q \rceil)$.
Let us see how this effects images, for example the Sobel filter:

Sobel Filter:

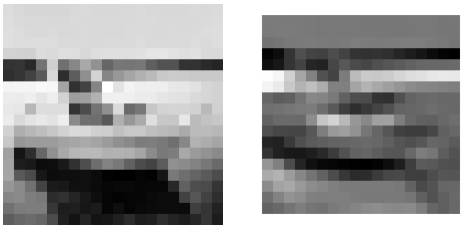
$$\mathbf{K} = \mathbf{G}_y = \begin{bmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix}$$

$$\mathbf{I} = \begin{matrix}
223 & 220 & 219 & 219 & 219 & 219 & 220 & 222 & 220 & 220 & 220 & 221 & 222 & 220 & 220 & 220 & 220 & 222 & 221 & 222 \\
225 & 222 & 220 & 221 & 220 & 220 & 222 & 221 & 221 & 221 & 221 & 221 & 222 & 222 & 221 & 221 & 221 & 221 & 222 & 223 \\
226 & 223 & 221 & 220 & 222 & 220 & 221 & 222 & 221 & 220 & 220 & 222 & 222 & 221 & 221 & 221 & 222 & 221 & 223 & 224 \\
226 & 223 & 223 & 224 & 223 & 222 & 222 & 222 & 221 & 222 & 222 & 222 & 223 & 222 & 222 & 222 & 222 & 223 & 222 & 224 \\
216 & 215 & 214 & 223 & 222 & 226 & 225 & 220 & 225 & 226 & 226 & 226 & 227 & 230 & 231 & 228 & 226 & 227 & 221 & 221 \\
93 & 94 & 85 & 142 & 213 & 140 & 158 & 184 & 166 & 132 & 145 & 144 & 145 & 127 & 117 & 117 & 93 & 92 & 86 & 81 \\
89 & 94 & 104 & 123 & 221 & 88 & 125 & 164 & 217 & 156 & 171 & 181 & 190 & 197 & 209 & 211 & 210 & 208 & 209 & 209 \\
242 & 243 & 243 & 246 & 244 & 180 & 91 & 104 & 156 & 250 & 229 & 241 & 237 & 238 & 237 & 232 & 231 & 227 & 219 & 216 \\
236 & 235 & 238 & 230 & 237 & 216 & 175 & 176 & 190 & 238 & 239 & 238 & 235 & 235 & 230 & 228 & 227 & 221 & 219 & 216 \\
233 & 231 & 195 & 224 & 223 & 224 & 93 & 112 & 142 & 165 & 222 & 135 & 145 & 206 & 216 & 218 & 217 & 215 & 196 & 214 \\
235 & 221 & 229 & 233 & 229 & 237 & 161 & 139 & 138 & 121 & 189 & 170 & 185 & 226 & 208 & 207 & 232 & 198 & 203 & 207 \\
215 & 188 & 195 & 200 & 207 & 216 & 222 & 221 & 214 & 212 & 202 & 208 & 208 & 201 & 197 & 195 & 185 & 198 & 198 & 195 \\
207 & 169 & 173 & 173 & 174 & 179 & 182 & 185 & 180 & 173 & 165 & 159 & 149 & 151 & 147 & 139 & 158 & 191 & 187 & 181 \\
177 & 69 & 89 & 133 & 165 & 184 & 195 & 196 & 196 & 190 & 192 & 189 & 181 & 161 & 147 & 150 & 183 & 173 & 167 & 160 \\
179 & 76 & 73 & 62 & 64 & 73 & 91 & 114 & 138 & 154 & 147 & 128 & 112 & 105 & 124 & 165 & 161 & 154 & 146 & 135 \\
181 & 122 & 111 & 71 & 63 & 64 & 64 & 66 & 62 & 55 & 53 & 53 & 57 & 55 & 70 & 147 & 138 & 131 & 123 & 112 \\
169 & 160 & 141 & 105 & 74 & 73 & 66 & 65 & 62 & 59 & 60 & 60 & 58 & 59 & 59 & 84 & 128 & 114 & 111 & 94 \\
157 & 160 & 151 & 147 & 87 & 91 & 82 & 72 & 64 & 57 & 56 & 57 & 58 & 61 & 63 & 63 & 104 & 119 & 104 & 89 \\
141 & 149 & 150 & 150 & 96 & 76 & 77 & 70 & 74 & 58 & 64 & 57 & 60 & 62 & 63 & 63 & 62 & 102 & 94 & 80 \\
125 & 133 & 137 & 132 & 111 & 78 & 68 & 81 & 69 & 70 & 78 & 87 & 76 & 78 & 89 & 81 & 76 & 83 & 95 & 75
\end{matrix}$$

$$\mathbf{I}^* = (\mathbf{I} * \mathbf{K} \underbrace{+ 4 * 256 \;) \; / \; 8}_{\text{some minor image normalization}}$$

$$\mathbf{I}^* = \begin{matrix}
130 & 129 & 129 & 129 & 129 & 129 & 129 & 129 & 129 & 129 & 129 & 129 & 129 & 129 & 129 & 129 & 129 & 129 \\
129 & 130 & 130 & 130 & 129 & 129 & 129 & 129 & 129 & 129 & 129 & 129 & 129 & 129 & 129 & 129 & 129 & 129 \\
124 & 126 & 128 & 130 & 130 & 130 & 129 & 130 & 131 & 131 & 131 & 131 & 133 & 133 & 132 & 131 & 130 & 128 \\
62 & 68 & 89 & 105 & 99 & 97 & 104 & 99 & 89 & 88 & 90 & 87 & 82 & 77 & 73 & 67 & 63 & 60 \\
69 & 73 & 90 & 98 & 81 & 79 & 101 & 111 & 103 & 100 & 106 & 109 & 113 & 117 & 119 & 120 & 120 & 122 \\
204 & 200 & 178 & 154 & 134 & 107 & 99 & 131 & 167 & 176 & 175 & 177 & 183 & 187 & 189 & 194 & 196 & 195 \\
199 & 193 & 174 & 162 & 169 & 158 & 134 & 133 & 154 & 163 & 157 & 152 & 146 & 141 & 137 & 136 & 135 & 133 \\
118 & 112 & 114 & 126 & 137 & 135 & 129 & 115 & 105 & 103 & 90 & 88 & 106 & 117 & 121 & 122 & 121 & 121 \\
124 & 125 & 127 & 129 & 131 & 123 & 111 & 96 & 86 & 93 & 99 & 106 & 117 & 119 & 121 & 124 & 121 & 120 \\
115 & 120 & 120 & 120 & 141 & 173 & 181 & 166 & 147 & 138 & 152 & 153 & 133 & 120 & 116 & 115 & 120 & 124 \\
105 & 100 & 100 & 100 & 110 & 132 & 148 & 151 & 144 & 128 & 118 & 109 & 98 & 95 & 95 & 101 & 115 & 120 \\
81 & 79 & 93 & 106 & 112 & 115 & 117 & 118 & 119 & 121 & 119 & 114 & 109 & 105 & 111 & 119 & 118 & 113 \\
89 & 78 & 74 & 74 & 77 & 84 & 94 & 107 & 116 & 118 & 114 & 110 & 109 & 120 & 132 & 128 & 114 & 108 \\
145 & 133 & 103 & 80 & 69 & 64 & 63 & 62 & 61 & 60 & 62 & 67 & 77 & 96 & 112 & 112 & 107 & 106 \\
157 & 161 & 149 & 136 & 127 & 116 & 104 & 91 & 84 & 86 & 94 & 101 & 102 & 96 & 96 & 105 & 110 & 110 \\
140 & 153 & 155 & 147 & 140 & 137 & 132 & 130 & 130 & 130 & 130 & 130 & 129 & 117 & 102 & 108 & 119 & 119 \\
123 & 135 & 144 & 140 & 133 & 132 & 133 & 132 & 130 & 129 & 128 & 129 & 130 & 127 & 115 & 108 & 115 & 121 \\
116 & 120 & 126 & 131 & 126 & 124 & 130 & 132 & 135 & 139 & 141 & 139 & 138 & 139 & 133 & 119 & 115 & 120
\end{matrix}$$

the transformation on the image looks like this:



(a) input image (b) filtered image

Figure 3.28: (3.28a source *CIFAR-10 dataset*)

The output image is one pixel smaller into every direction. This is because the filter has to fit entirely inside the image while sliding over every pixel of the image, and so the center of the 3×3 kernel cannot reach a pixel on the rim of the image and the center of the kernel lays on the pixel, where the one filter value is calculated for. There multiple edge solutions to prevent the scaling down of the image. Some involve using just zero values beyond the image's edge for the filter operation, others mirrors the image beyond the edge. But in most neural networks the scaling down is volitional anyway.
In case of the Sobel filter in y direction, the filter extracts horizontal edges and smoothen the image in the horizontal direction, so that the horizontal edges become perfectly visible. In CNNs the kernels are configured by the network itself and when a CNN is searching for horizontal edges one of it feature extracting kernel could look very alike the Sobel filter. In case of our picture of a boat (Figure 3.28a) you can see, that the Sobel filter clearly detects the horizontal edge of the horizon which separates air and water. In fact, most of the ships in the CIFAR-10 dataset where detected this way by VResNet and it doesn't matter if there was a ship on the image or not. Therefore I should have trained the network with pictures of the ocean without ships as well to train the network to look at other things than the horizon on the images. But nearly all pictures, which involve a clearly visible horizon featured ships on it, so the network didn't learn the real thing here.

Convolutional Neural Networks:

We saw in chapter 3.1.3 how CNNs work in general, but now let us look inside one convolutional layer (and forget about the $^{(n)}$ layer denominator and use a' for $a^{(n-1)}$ and A' for $A^{(n-1)}$) and how it extract some features. We start with our equation3.2, because the equation for a convolutional layer is derivable from it and we use the index (x, y) for i and (g, h) for j, because now the input of the network isn't a vector any longer, it is now an image. Before that, w_{ij} described an edge $i \to j$ now $w_{(x,y)\to(g,h)} (\equiv w_{(x+u-\lceil p \rceil, y+v-\lceil q \rceil)}$ for u and $v = 0$) describes an edge going from pixel (x, y) to pixel (g, h) of the output

image. This time we use as an example a 3×3 kernel for the convolution:

$$3.2: \quad z_j = \sum_i (a'_i \cdot w_{ij}) \qquad \Rightarrow z_{(g,h)}$$

$$= \underbrace{\sum_{(u,v)}}_{\text{over all}} \left[\mathbf{A'} \odot \underbrace{\begin{bmatrix} \ddots & & \vdots & & \iddots \\ 0 & 0 & 0 & 0 & 0 \\ 0 & w_{(x-1,y-1)\to(g,h)} & w_{(x,y-1)\to(g,h)} & w_{(x+1,y-1)\to(g,h)} & 0 \\ \cdots\ 0 & w_{(x-1,y)\to(g,h)} & w_{(x,y)\to(g,h)} & w_{(x+1,y)\to(g,h)} & 0\ \cdots \\ 0 & w_{(x-1,y+1)\to(g,h)} & w_{(x,y+1)\to(g,h)} & w_{(x+1,y+1)\to(g,h)} & 0 \\ 0 & 0 & 0 & 0 & 0 \\ \iddots & & \vdots & & \ddots \end{bmatrix}}_{\text{3×3 kernel matrix (consits out of all edge weights running into one neuron)}} \right]_{(u,v)}$$

$$\Rightarrow \mathbf{Z} = \mathbf{A'} * \mathbf{\Theta} \tag{3.33}$$

In equation 3.33 we see the vectorial form of the equation, but this time for every output pixel of the convolution. The kernel matrix $\mathbf{\Theta}$ is basically that what in (II.1) was the Θ weight matrix, but this time it is a matrix of spatially dimensions, which can be lain on an image. Θ was a matrix of w_{ij} edge elements, which describes every connection between two network layers:

$$\Theta = \begin{bmatrix} w_{11} & w_{12} & w_{13} \\ w_{21} & w_{22} & w_{23} \\ w_{31} & w_{32} & w_{33} \end{bmatrix}, \quad \mathbf{\Theta} = \begin{bmatrix} w_{(x-1,y-1)\to(g,h)} & w_{(x,y-1)\to(g,h)} & w_{(x+1,y-1)\to(g,h)} \\ w_{(x-1,y)\to(g,h)} & w_{(x,y)\to(g,h)} & w_{(x+1,y)\to(g,h)} \\ w_{(x-1,y+1)\to(g,h)} & w_{(x,y+1)\to(g,h)} & w_{(x+1,y+1)\to(g,h)} \end{bmatrix}$$

Both matrices looks very similar, but they are totally different. First of all we have to talk about the fundamental principle, that lies behind the theory of CNNs. The idea is to use many similar neurons, that do different things at different connections in the network, to harness the maximum computational power of graphics cards. Every Pixel of the input image runs through the same kernel $\mathbf{\Theta}$ with the same wights as all other pixels of the input image. The only factor that changes the output is the arrangement of the pixels to each other in the near neighborhood. I choose the big 3×3 kernel matrix with all the zeros above to demonstrate, that, if this matrix has instead of the zeros w elements and has the size of the image $\mathbf{A'}$, it would be the same operation like in a normal neuron, which we call for that fact a fully connected layer, because every input pixel is connected to every output pixel. When the filter has the size of the image it cannot longer be shifted over the image so that all pixels would connect to one neuron j. The only difference is that the vector A' is than viewed as a Matrix $\mathbf{A'}$. The indices would than transform like this: $i = x + (y-1)*$(image size in x dimension) and $j = u + (v-1) * p$(kernel size in u dimension).
To show the difference between the two Θ matrices, let us fill the indices with of $\mathbf{\Theta}$ with some example values, too and transform them into i and j values:

$$\Theta = \begin{bmatrix} w_{11} & w_{12} & w_{13} \\ w_{21} & w_{22} & w_{23} \\ w_{31} & w_{32} & w_{33} \end{bmatrix}, \qquad \mathbf{\Theta} = \begin{bmatrix} w_{1\to1} & w_{2\to1} & w_{3\to1} \\ w_{4\to1} & w_{5\to1} & w_{6\to1} \\ w_{7\to1} & w_{8\to1} & w_{9\to1} \end{bmatrix}$$

Where every column in Θ is being used to form a weighted sum for every neuron in a layer of a neural network, $\mathbf{\Theta}$ just computes the weighted sum with $\mathbf{A'}$ for one neuron and it no longer connects every neuron from one layer to the next.
But with those simplifications is room for more feature extractions in a single layer. We can use multiple kernels $\mathbf{\Theta}_m$, where every $\mathbf{\Theta}_m$ extract one feature $\mathfrak{f}_m$ in an input image. This results in multiple output images. This stack of images can be seen as an tensor $\mathbf{T_Z}$ with two dimensions x and y for spatial dimensions and one dimension m where one image $\mathbf{A_m}$ of one layer m in the stack of size r is called a mini-batch (for GPU purposes). So the tensor has three dimensions in total for one input image $\mathbf{x}$ of the Neural Network. That makes 4 dimensions for the whole training set $\mathbf{X}$ (because every image has multiple color channels, which form one mini-batch layer).

$$\mathbf{Z}_m = \mathbf{A'} * \mathbf{\Theta}_m \qquad \mathbf{T_Z} = \left[\mathbf{Z}_1, \mathbf{Z}_2, \cdots, \mathbf{Z}_r\right] \tag{3.34}$$

$$\Rightarrow \mathbf{T_Z} = \mathbf{A'} \circledast \mathbf{T_\Theta} \text{ where } \circledast \text{ is a convolution between each tensor layer } m \text{ with the image } \mathbf{A'} \text{ and a subsequent layer stacking operation to form the output tensor } \mathbf{T_Z}. \tag{3.35}$$

After the stack of images $\mathbf{Z}_m$ passed the possibly existing transfer function between two convolutional layers of the network we got our new $\mathbf{A}_m$ stack for the next layer. But this time we try to extract features not just from one image, but also on all layers l of the input tensor $\mathbf{T}^{(l,v,u)}_{\mathbf{A'}}$ with a stack size of s layers. For that we need a three dimensional tensor $\mathbf{T}^{(l,v,u)}_{\mathbf{\Theta}}$ with two spatial dimensions u and v and dimension l for every mini-batch in the input image:

$$z_{(g,h)} = \sum_{l}^{s} \left(\sum_{(u,v)} \left(\underbrace{\left[\mathbf{A'}_l \odot \mathbf{\Theta}_l\right]_{(u,v)}}_{\text{p×q matrices center (x,y)}} \right) \right) \tag{3.36}$$

$$:= \sum_{(u,v,l)} \left(\mathbf{T}^{(l,v,u)}_{\mathbf{A'}} \odot \mathbf{T}^{(l,v,u)}_{\mathbf{\Theta}} \right) = z_{(g,h)} \tag{3.37}$$

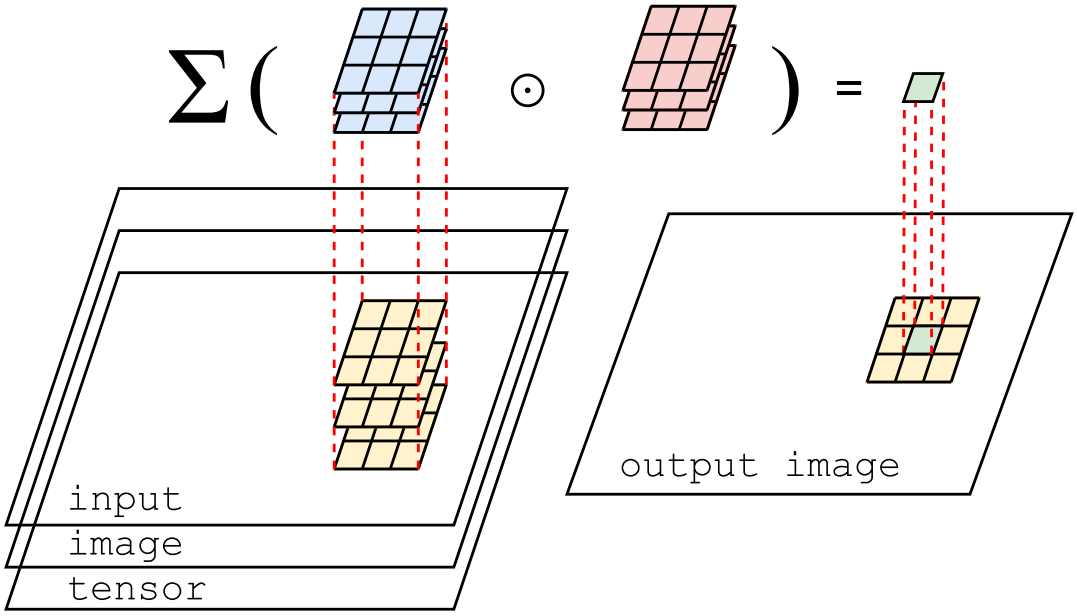


Figure 3.29: convolution on multiple batches

Where tensor $\mathbf{T}_{\mathbf{A}'}^{(l,v,u)} \odot \mathbf{T}_{\mathbf{\Theta}}^{(l,v,u)}$ is an element wise tensor multiplication.
For the whole output image:

$$\mathbf{Z} = \mathbf{T}_{\mathbf{A}'}^{(l,y,x)} * \mathbf{T}_{\mathbf{\Theta}}^{(l,v,u)} \tag{3.38}$$

With multiple features to extract:

$$\mathbf{T}_{\mathbf{Z}}^{(m,h,g)} = \mathbf{T}_{\mathbf{A}'}^{(l,y,x)} \circledast \mathbf{T}_{\mathbf{\Theta}}^{(m,l,v,u)} \tag{3.39}$$

Where $\mathbf{T}_{\mathbf{\Theta}}^{(m,l,v,u)}$ is a tensor $\in \mathbb{R}^{r \times s \times q \times p}$

Strides and Padding

If we want to use a certain step wide instead of one for moving the convolution kernel over the image, we can apply a so called stride value $\mathfrak{s}$ to the convolutional operation. There is not really a mathematical difference in the operation through another stride size than one, but the output tensor's $\mathbf{T}_{\mathbf{Z}}^{(m,h,g)}$ spatially dimension (q,p) size will be $\left(\frac{1}{\mathfrak{s}_y}\ \text{q},\ \frac{1}{\mathfrak{s}_x}\ \text{p}\right)$, where $\mathfrak{s}_y$ is the step width in y direction and $\mathfrak{s}_x$ is the step width in x direction. Normally the stride size in both direction is the same.
Another problem I mentioned in chapter [3.1.3] is that the output image/tensor of a convolution will be smaller by the size of the filter - 1. This can lead to problems when a special tensor size is required for a specific operation, like the element wise addition in the ResNet when two network paths merge together. To combat this problem we can apply a padding to the convolutional operation.

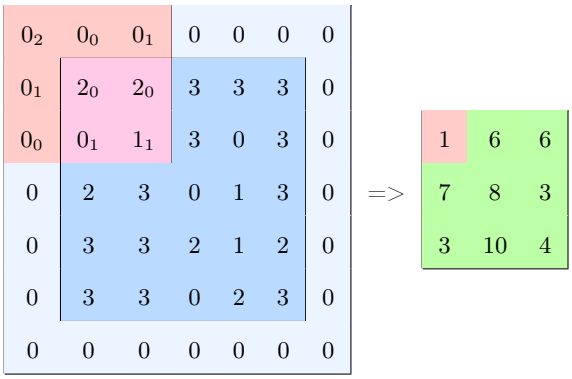


Figure 3.b: padding of zero (and stride of 2)

We can see in figure 3.b, that the original image is simply surrounded by zero valued elements temporary during the convolutional operation. The size of the thickness of the surrounding zero elements layer is the value of the padding parameter.