Deep Convolutional Networks for Image Processing

An Overview on Convolutional Neural Networks for Image Classification and Segmentation

F. Riegger

TUDelft

Deep Convolutional Networks for Image Processing

An Overview on Convolutional Neural Networks for Image Classification and Segmentation

by

F. Riegger

TU Delft

Contents

1 Introduction

Digital Image Processing (DIP) is the procedure of extracting useful information from a given image or generating a modified version of it by applying mathematical operations [2]. Although this seems to be somehow abstract, DIP became a daily component of many people's life.

Numerous smart phone applications such as bar code scanner or social networks make heavily use of DIP techniques. In Facebook for instance, it was common for a user to upload an image and link the persons shown in it with their profile. DIP however, recognizes the faces in images without the need of an explicit, manual link and informs the person automatically.

In spite of that, it affects lives in more essential means from a medical view point. DIP does not only pave new ways to detect cancer in earlier stages [47] but can also provide a substitution of the human eyesight. The detection of a smile triggers the vibration of a device and gives a blind person the possibility to interact with her or his counterpart [5].

DIP techniques also found their ways into academic life by tracking the attendance and attention of students during lectures, as lately introduced in universities in China and Paris [3]. Evaluating satellite images to explore new areas for oil and gas extraction or monitor the wildlife on the North Slope of Alaska and in Africa's National Parks is based on DIP methods [4]. Hence, even far away from civilization in the outer space, DIP is applicable and even necessary.

Although these applications could barely be more different, they are all instances of either *Image Classification* and *Image Segmentation*. Both are subfields of DIP and assign parts or the entirety of an input image to a class or category which, e.g., comprises all images that show the same object.

While classification aims at coarsely correlating one image with a unique class, segmentation can be interpreted as its refined version, where each pixel is assigned to one category.

The difference can be clarified by considering a potential application. Assume for instance, a sailing ship keeled over. Before leaving the sinking ship and escaping in the life boat, the captain released an appeal for help. Immediately, the coast guard moves out in a helicopter to search the survivors. To see a wider range of the ocean, a DIP capable camera tracks the happenings from above. Whenever this instrument detects an object, an image classification program tells the coast guard what exactly they see. Is it only a heap of rubbish, a tiny island or the desired life boat? As soon as, the latter is found, an image segmentation application is brought into operation. It analyses the exact position of the persons seeking for help, and the rescue mission succeeded.

The vast amount of different applications of classification and segmentation motivated the development of a wide range of diverse approaches. In the last few years, however, mainly one of these methods attracted the attention of scientists and researchers. One of them are so-called *Deep Convolutional Networks*, an approach that is based on artificial intelligence.

It is a special kind of *neural networks*, which in turns are learning machines.

This report aims at giving an overview about the most recent developments in the field of image classification and segmentation with deep convolutional networks. Preceding this, the relevant basics of machine learning including neural networks are introduced. These explanations are accompanied by a real-world image classification problem which I first approach by implementing a very simple neural network. My analysis reveals that this does not suffice to yield the expected results and motivates further theoretical studies of learning theory. Gradually, I add improvements to the simple neural network and based on that, the concept of Deep Convolutional Networks is developed. To finally emphasize the power of this kind of neural networks, I implement a more elaborate model and compare it to its simple predecessor.

The understanding and designing of machine learning systems is a complex task and so is their implementation. That is what the researchers of the Google Brain Team within Google's Machine Intelligence research organization felt like in 2015, when they decided to ease the task of implementation by developing the open source software library *Tensorflow*. While initially tailored for machine learning systems and deep neural networks, it is applicable to a wide range of different problems.

As indicated by the expression *"Tensorflow"*, models are abstracted to *tensor*-represented data that flows through a *graph*. Each node of this graph, represents a computation unit, and the edges express the data consumed or produced by a computation.
However, at this point the graph can be imagined to be a static construction that merely defines the computations but does not perform them. This is in the responsibility of so-called *sessions*. Not before a graph is run in a session, the operations are executed and tensors evaluated. While for designing the graphs a convenient, Python-based front-end API can be used, sessions provide the links between this API and the high-performance C++ runtime. Apart from executing the computation on local devices, sessions provide also access to remote devices using the distributed TensorFlow runtime [52]. Within this report, models of neural networks are implemented in the *Tensorflow* framework.

A further application of image segmentation is the automation of processes that would either take too long or would suffer from subjectivity if they were done by humans. One such process, is the analysis of material properties based on its microstructure. Often, this is used as a tool of quality assurance. In particular in the aerospace industry, where damages are avoided at any price, the condition of material is directly responsible for the safety and scrap rate of products. Hence, the German engine manufacturer *MTU Aero Engines* who is responsible for the entire product development process of aircraft engines, faces the challenge of microstructure analysis every day. For this reason, the company puts particular effort in the development of high-quality, automated image segmentation procedures. At the same time, the complexity and number of different manifestations of certain substructures hampers automation and demands further investigation of potential image segmentation methods. In this context, MTU commissions a research project, posing the question:

Is it possible to apply an image segmentation technique based on Deep Learning to determine different phenotypical structures of a material with nearly human-like precision?

This is examined in the use case of nickel-base superalloys. The latter can be precipitation hardened by a so-called γ'-phase which appears in a cubic shape. The automated identification of these precipitations, has to take the following properties and difficulties into account:

1. two size-depending instances of γ'-phase need to be distinguished
2. their cubic shape can be deformed due to temperature or force effect
3. the quality metallographic images does not always allow the detection or visibility of all four edges

4. the magnification of the given images and hence snippet of the microstructure which they show is crucial for the final phenotypic occurrence

Given this research problem, the following explanations are firstly intended to explore the abilities and borders of image processing techniques based on Deep Learning. As summarized in chapter 9, this allows the elaboration of a strategy how to approach the research question.

2

Artificial Intelligence and Deep Learning

In the early 19th century, Augusta Ada King, Countess of Lovelace developed the basics of programmable languages and conceived the first programmable computers. Since then humanity has wondered whether such machines can become intelligent. A lot of effort has been put in the attempt to answer this question of the existence of *artificial intelligence* (AI).

It soon became evident, that machines perform well in solving abstract issues such as manipulating strings or playing chess. For instance, while a machine easily solves a system of linear equations within a couple of seconds, this problem is intellectually challenging for humans and a manual solution might take minutes.

However, when it comes to classifying an object such as a randomly chosen digit in an image, computers can not compete with humans. While solving a linear system of equations relies on a few basic arithmetic rules, the human abilities to recognize objects arise from an intuitive knowledge which was derived from the immense amount of environmental impressions. Humans saw digits a thousands times before and it became natural to recognize them. However, the impalpable and most often subjective nature of this knowledge aggravates the formulation of general rules that define what a particular digit looks like. This reveals one of the main challenges in AI - fetching the same informal knowledge that humans have and process it in such a way that computers can employ it.

A general approach to capture the necessary knowledge about the world is to hard-code it in a formal language. It clearly, for example, defines what a particular digit has to look like. Given this formulation, a computer can automatically reason about statements by using inference rules. This *knowledge-based* branch of AI suffers from a particular difficulty. It assumes that AI systems are able to acquire their knowledge by extracting patterns from raw data. From thousands of pictures with digits, the machine has to learn the appearance of any of them. This capability of *machine learning* empowers computers to make apparently subjective decisions or solve tasks that involve knowledge about the real world - skills that have been claimed by humanity for long time [19].

A crucial factor that heavily influences the performance of machine learning algorithms is the *representation* of data. To decide whether an image contains a four, the computer does not have to know if the shown digit is red or blue. Moreover, the only relevant information is the shape of the digit. This kind of useful information that is included in the representation is called *feature*. A simple visual example for the dependency on representation is given in figure 2.1.

Machine learning algorithms which are aware of the meaningful features that need to be extracted, are capable of solving the corresponding AI tasks. For instance, assume one wants to write a program that recognizes cars in an image. For humans wheels and spotlights are obvious features that clearly describe a car. The machine, yet, only sees the pixels of an image and the actual difficulty is to describe a wheel in terms of pixels. Of course, the geometric

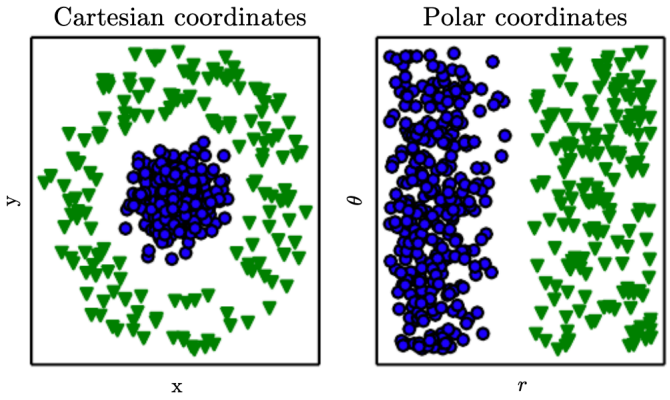

Figure 2.1: Assuming the scatter plot of some data has to be separated into two categories, using only one straight light.*(Left)* For data in Cartesian coordinates, this is impossible.*(Right)* If the data is represented in polar coordinates, the task becomes easy. [19]

shape of a car is simple and easy to translate into pixel values. However, these values are strongly affected by shadows falling on the wheel or objects obscuring the front of the car [19]. Hence, features that might be meaningful for humans are less useful for machines.

Instead of trying to prescribe the features, why shouldn't the machine learn them by itself? Until now, machine learning detects mappings from representation to outputs, e.g. wheels are clearly associated with a car. *Representation learning* is a field of machine learning that further leaves the machine the freedom to not only learn the mapping but also the representation. Hence, the computer learns from given data which set of features to extract. Often, learned representations outperform hand-designed representations [19].

Meaningful features are learned by separating factors of variation that explain the data. Often, these factors are not observable. Instead, they could be thought of as concepts or abstractions that are used to make sense of the wide variety of the given data. When analyzing cars in an image the factors of variation include, e.g., the position of the car, its color, angle, the brightness of the sun and so on [19].

However, this reveals the central difficulty of representation learning: how can these abstract or high-level features be obtained from raw data?
Deep Learning (DL) is one approach to solve this essential issue. It introduces representations that are expressed in terms of other, simpler representations. Construction-kit-like, complex and meaningful concepts are built up by simple ones. As shown in figure 2.2, a human can be recognized by combining simple concepts. For instance, edges are used to define corners and contours which are parts of higher-level objects.

The most fundamental example of a deep learning model is the *Multilayer Perceptron* (MLP). Basically, it is a mathematical function, mapping input to output. This function might by complicated and is formed by composing many simple functions.

As summarized in figure 2.3, DL is an approach to AI. In general, AI is concerned with finding methods to extract knowledge from data. Most often capturing knowledge is difficult to achieve, since the input data is complicated and arises from real-world problems. In particular, ML resolves this problem by extracting only patterns and relevant features. However, this prompts the further question: how does the computer know which features are meaningful? For a long time, these features were hand-crafted and stipulated by the programmer. Opposed to that, representation learning allows the machine to learn the meaningful features on its own. DL is a special kind of representation learning. It achieves great flexibility by representing the world as a nested hierarchy of concepts with each concept being defined in relation to simpler ones.

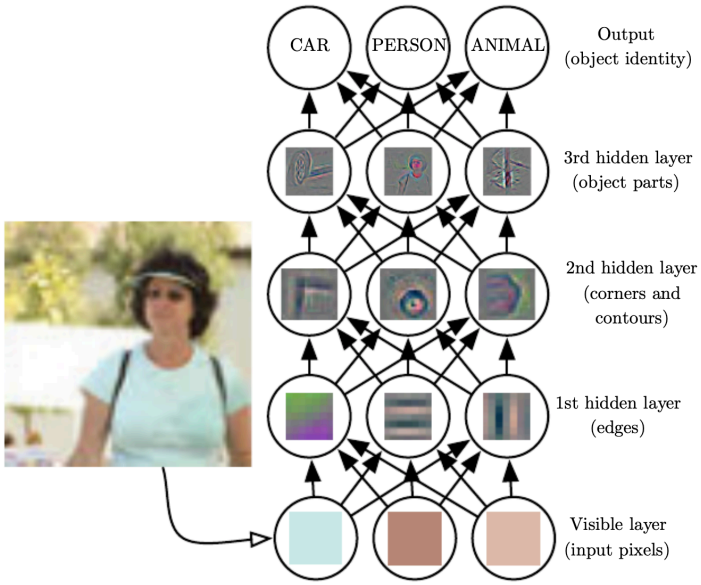

Figure 2.2: The shown model illustrates the approach of deep learning. The AI task is to classify whether an input image shows a car, person or an animal. However, computers can barely directly map the input data to the categories since it is difficult to analyze the raw input data. Deep learning resolves this issue by reducing the complicated mapping to several, nested sub-stages. The raw input data is visible for humans and hence represented by the *visible layer*. Opposed to that *hidden layers* extract increasingly abstract features which are not directly observable from the input data. The images in each hidden layer visualize the extracted features. From input data, edges are extracted which can easily be described in terms of raw input pixels. These are then used to resolve the more complicated corners and contours which are the basic for object parts. Merging them together indicates which object is depicted in the input image [19].

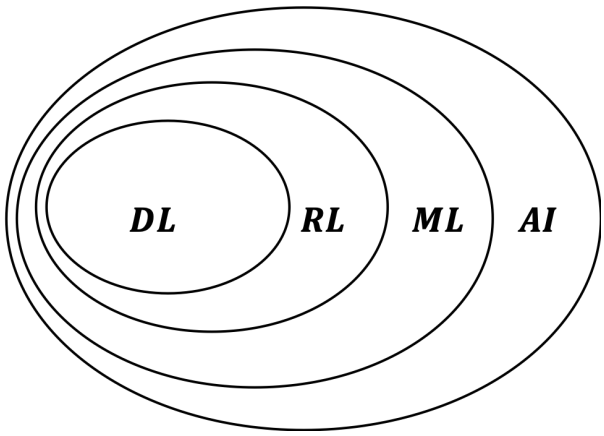

Figure 2.3: The diagram shows the relation of each sub-branch of *artificial intelligence* (AI). *Deep learning* (DL) is a kind of *representation learning* (RL) which in turn is a part of *machine learning* (ML).

3

Multilayer Perceptron

Based on the idea of deep learning, complicated real-life problems such as image classification or segmentation can be tackled by AI approaches. However, models for this kind of tasks are complicated and not easy to understand without any prior knowledge of machine learning. Therefore, this chapter is devoted to summarizing the basic mechanisms. Machine learning is a huge and fast developing field of research and summarizing everything would exceed the framework of this report. Thus, the focus is only on either very fundamental strategies or those that will become relevant when dealing with image processing.

The core part of everything is a *learning algorithm*. This is an algorithm that is able to learn from data. Since learning is a process that is most often associated with humans or animals but rarely with machines, this needs further explanation. A concise definition is: "A computer program is said to learn from experience with respect to some class of tasks T and performance measure P, if its performance at tasks in T, as measured by P, improves with experience." [36]

Suppose for instance, the so-called *MNIST* data set is given. As can be seen in figure 3.1, it is a collection of images of handwritten digits. In a typical machine-learning task, one image is feed-forwarded to a learning algorithm. The latter is designed such that it recognizes which digit is encoded in the input image pixels. This is an instance of a *classification task* where an input is assigned to one of K classes. Referring to the MNIST data set, each digit forms one of the $K = 10$ possible classes.

From a mathematical point of view the classification is defined as a function that somehow maps an input vector $x \in \mathbb{R}^N$ to one out of K discrete class C_k where $k = \{1, \dots, K\}$. Most often, the classes are taken to be disjoint and each input uniquely belongs to one class. This can be imagined as separating the input space into *decision regions* whose boundaries are called *decision boundaries* or *decision surfaces*. Linear models for classification have decision boundaries which are linear functions of the input vector x and hence are defined by $(N-1)$-dimensional hyperplanes within the N dimensional input space. Data sets whose classes can be separated by linear decision boundaries are said to be *linearly separable*.

However, real-world classification problems are none of those and hence more complex models have to be derived. As will be shown, *Multilayer perceptrons* are a good choice for *non-linearly separable* data sets.

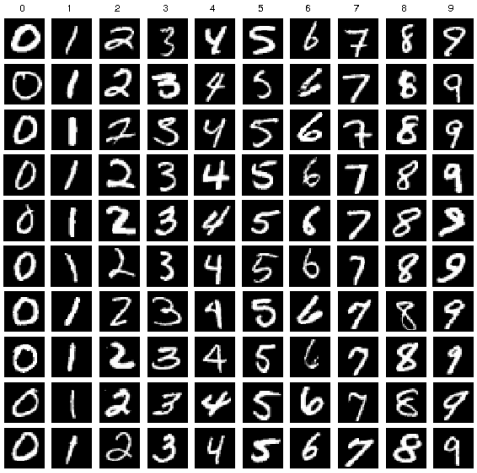

Figure 3.1: The MNIST data set contains 60 000 monochrome images. Each of them has a resolution of 28×28 and shows only one handwritten digit. [33]

However, classification is only one example of machine learning tasks. More generally, the machine learning system should be able to process a certain event or object like an image. Here, *examples* are defined as a collection of features which arose from measurements of the object like the pixel values of images. Then, the task is to describe how a learning algorithm should process with given examples [19].

A further essential part of learning is the *performance measure*. Often, it correlates to the learning task and indicates how well the learning algorithm solves it. Classification tasks can, e.g., be measured by the *accuracy* which gives the proportion of images that are assigned to the right class. At the same time, equivalent information can be obtained by the *error rate*. This performance measure depends on the amount of images for which the learning algorithm generates a wrong output. While only a few examples are used to design the learning algorithm, the final interest is in how well it performs in real-world application. Hence, a second set of unseen examples is necessary. Finally, this so-called *test set* is used to measure the performance and *validate* the learning algorithm [19].

Last but not least, the third pillar of the learning process is the *experience*. The kind of experience a learning algorithm has while learning separates the wide field of machine learning into two categories, the *unsupervised* and *supervised*. Note, that the examples which were introduced in terms of the learning tasks, are called *data set* or *data points*. For the MNIST data set in figure 3.1 the examples are simply the images. However, it might happen that each of the examples is associated with a *label* or *target*. In figure 3.1 these are denoted by the very first row which does not contain images but solely digits. Obviously, *labels* are the right solution to the learning task.

The existence of labels is the parting of the ways. While unsupervised learning algorithms do not have access to labels, supervised learning exploits them to design the algorithm. The remainder of the report focuses on supervised learning.

In the course of this report, the following frame is maintained. All machine learning algorithms discussed and derived are supervised. The final implementation of a supervised learning algorithm is separated into four steps. First of all, from a class of parameterised mathematical models the one that is associated with the learning task is chosen. As will be shown later, a well-known model class are so-called *neural networks* and a corresponding performance measure is defined. This is used in the next step where an optimization algorithm adjusts the model parameter. In terms of neural networks this phase is called the *training* step. Last but not least, the model is evaluated in a *validation step*. This phase can be used to further adjust special parameters, so-called *hyperparameters* that control the model performance but can not be derived adapted training.

These basic steps to design a machine learning algorithm are explained within the fol-

lowing chapter. All formulas that are derived during the following chapter are based on two books, *Pattern Recognition and Machine Learning* written by Bishop et al. [11] or *Deep Learning* by Goodfellow et al. [19]. It is exemplified by deriving a classification model for the MNIST data set.

3.1. Model

The most successful parametric model in the context of pattern recognition is the so-called *Multilayer Perceptron* (MLP). This is a representative of the class of *feedforward neural networks*. Initially appearing in 1943, McCulloch and Pitts aimed at finding a mathematical representation of information processing in biological systems. Nowadays, *neural networks* cover a wide range of different models including even models that do not resemble biological systems.

The goal of these networks is the approximation of a function $y(x^{(p)}) = f(x^{(p)}, w)$ with $w \in \Lambda$. Here, Λ is the space of all possible parameters. Consider the data set $\{x^{(p)}, y_{\text{true}}^{(p)}\}$ with $p = 1, \cdots, D$ where $x^{(p)} \in \mathbb{R}^N$ is some input data and $y_{\text{true}}^{(p)}$ the one out of K labels that corresponds to $x^{(p)}$. For instance, the input $x^{(p)}$ can be an image in pixel format that is flattened out to a $N \times 1$ dimensional column vector. The label $y_{\text{true}}^{(p)}$ denotes a class to which the image belongs to as, e.g., all images that show a hand-written four.

The function $f : \mathbb{R}^N \mapsto \mathbb{R}^K$ with $f(x^{(p)}, w)$ generates a K-dimensional output vector. A input vector $x^{(p)}$ is then assigned to class C_k if $y_k(x^{(p)}) > y_j(x^{(p)})$ for all $1 \leq j, k \leq K$ and $j \neq k$. For simplicity $y_k(x^{(p)})$ and $y_k^{(p)}$ will be used interchangeable. The parameters $w \in \Lambda$ are chosen such that $y_k(x^{(p)}) = f(x^{(p)}, w)$ predicts the true label. However, before the parameters $w \in \Lambda$ are set, the model $f(x^{(p)}, w)$ has to be defined.

A very first and naive attempt, is to model f as a linear combination of the input data

$$y_k(x^{(p)}) = w_{k0} + \sum_{i=1}^{N} w_{ki} x_i^{(p)}. \tag{3.1}$$

This is a simple but also very restrictive approach. First of all, it only exploits linear relations between input and output. In addition to that, it does not scale well with the input space dimension. This *curse of dimensionality* states that the amount of input data has to grow exponentially with the dimension of the input data to maintain statistical significance. Enough data has to be provided to learn the structure of the input space.

Linearly combining the input yields linear decision boundaries in the input space. Analogously, combining non-linear transformations $\phi(x^{(p)})$ of the input leads to linear decision boundaries in the space of the fixed, so-called *basis functions* ϕ. These correspond to non-linear decision boundaries in the space of the original input x. This can be formalized as

$$y_k(x^{(p)}) = w_{k0} + \sum_{j=1}^{M} w_{kj} \phi_j(x^{(p)}), \tag{3.2}$$

with M non-linear functions ϕ. For classification problems, a generalized version of equation 3.2 is

$$y_k(x^{(p)}) = g\left(\sum_{j=0}^{M} w_{kj} \phi_j(x^{(p)}) \right), \tag{3.3}$$

where $\phi(x_0^{(p)}) = 0$ and $g(\cdot)$ is a non-linear *activation function*. It is chosen such that the outputs are more interpretable for classifying the input. For instance, for the right choice of $g(\cdot)$ the output can be understood as the probability that the input belongs to a certain class.

Models that are defined by equation 3.3 are known as *Rosenblatt perceptrons* [11]. The corresponding *perceptron convergence theorem* guarantees that for any data set that is linearly separable, equation 3.3 finds the exact solution within a finite number of steps.

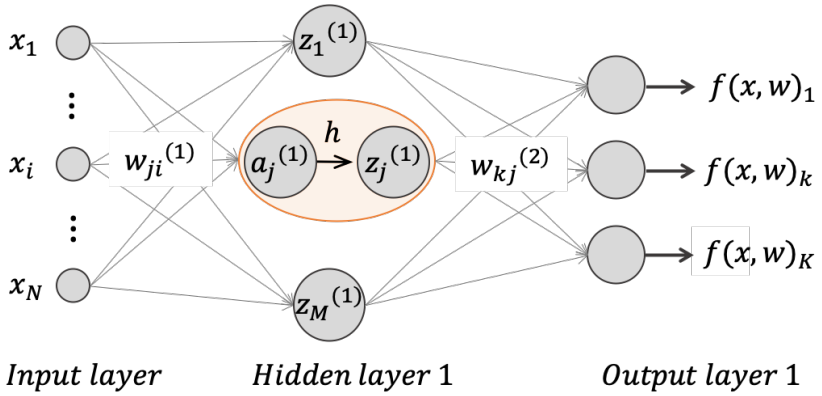

Figure 3.2: In general, feedforward neural networks are associated with directed computational graphs. In particular the multilayer perceptron is composed of several layers where each often them consist of so-called units, neurons or nodes. The number of layers is called the *depth* while the number of units in a layer determines its *width*.

Despite the fact that these functions have useful analytical and computational properties, their practical applicability is limited by the curse of dimensionality. MLPs remedy this shortcoming by choosing a particular kind of basis functions $\phi(x)$.

The fixed functions like in equation 3.3 are parameterized such that they adapt to the input data. Obviously, there exists a vast number of parametric nonlinear basis functions. The MLP model chooses in particular basis functions which are non-linear functions of a linear combination of the input set of

$$a_j = w_{j0}^{(1)} + \sum_{i=1}^{N} w_{ji}^{(1)} x_i, \tag{3.4}$$

where for simplicity $x_i^{(p)}$ is substituted by x_i. The parameters are the coefficients $w_{ji}^{(1)}$ of the linear combination which often are called *weights*. In case of $w_{j0}^{(1)}$ one often says *bias*. In terms of neural networks the quantities a_j are *activations*. To achieve nonlinearity with respect to the input, each of these coefficients are transferred to an *activation function* h.

The output $z_j = h(a_j)$, known as *hidden units* or *neurons* correspond to the output of the fixed basis functions ϕ. Following the definition of the Rosenblatt perceptron in equation 3.3, these hidden units are again linearly combined, yielding *output activations* of the form

$$a_k = w_{k0}^{(2)} + \sum_{j=1}^{M} w_{kj}^{(2)} z_j \tag{3.5}$$

where $k = \{1, \ldots, K\}$ is the total number of outputs. As before in equation 3.3, an appropriate non-linear activation function $g(\cdot)$ is applied and the final predictions y_k are

$$y_k = f(x,w)_k = g\left(\sum_{j=1}^{M} w_{kj}^{(2)} h\left(\sum_{i=1}^{N} w_{ji}^{(1)} x_i + w_{j0}^{(1)}\right) + w_{k0}^{(2)}\right) \tag{3.6}$$

All in all, this leads to a basic multilayer perceptron model which can be described as a composition of functional transformations. As illustrated in figure 3.2, this composition is associated with a directed, acyclic graph, where each layer corresponds to one functional transformation.

One of the striking advantages of MLP is the nonlinearity with respect to the input data, which is yield by nonlinear activation functions. The choice of these functions depends on the application purpose. While output activation functions mainly determine the appearance of the output units, activation functions in hidden layers strongly affect the behaviour of

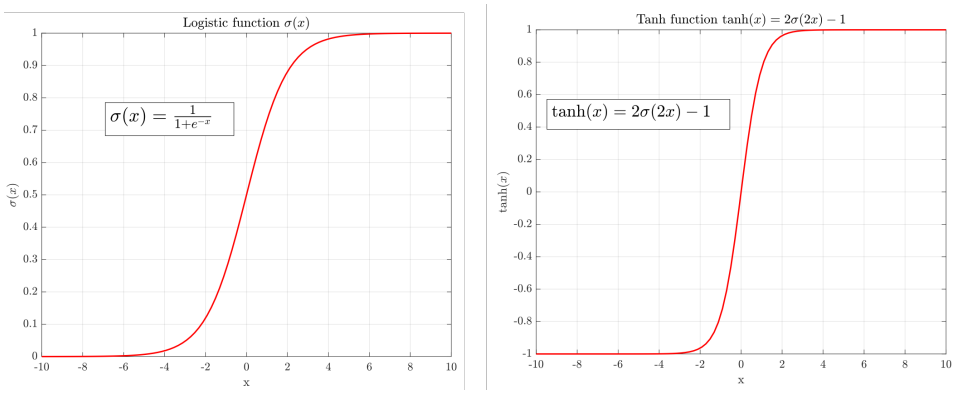

Figure 3.3: Two prominent representatives of saturating activation functions are the *(LEFT) logistic function* $\sigma(x) = \frac{1}{1+e^{-x}}$ and the *(RIGHT) tanh function* $tanh(x) = 2\sigma(2x) - 1$. In particular, the logistic function can easily be interpreted as a probability and hence is often used in the output layer. At the same time, saturating functions are hardly found in hidden layers. Their asymptotic behaviour might causes vanishing gradients what is harmful during backpropagation (see section 3.3.1).

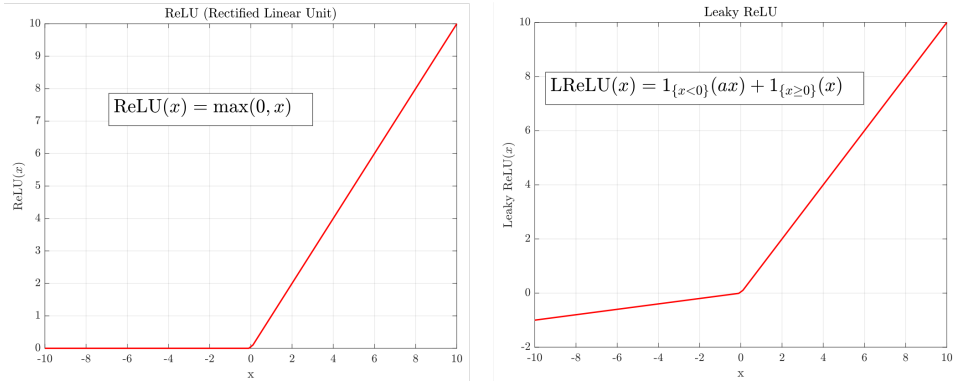

Figure 3.4: Non-saturating functions mainly include *(LEFT) Rectified Linear Unit* function which is nothing else than a threshold function at 0 as well as it's slightly modified version *(RIGHT)* the *Leaky ReLU*. Both of them are easy to implement and show fast convergence during optimization (see section 7.1.3). While *ReLU* suffers from dying units i.e. parameters are updated such that they will never become active again (see section 3.3.1) this issue is remedied in its leaky version.

the network while setting the parameters. Generally, two types of activation functions are distinguished, *saturating* or *non-saturating* ones. Figure 3.3 shows representatives of the first kind while figure 3.4 illustrates non-saturating functions.

Neural networks are *universal function approximators*. A multilayer perceptron with one *hidden layer* and a sufficiently large number of units can uniformly approximate any continuous function on a compact input space with arbitrary accuracy. However, the key issue is to find the right parameters $wji^{(n)}$ in the n-the hidden layer.

3.2. Performance measure

.

To evaluate how well a model solves the given task a *performance measure* is defined. A commonly chosen measure for classification tasks is the *accuracy*, which gives the portion of correctly predicted labels. This is quantified by the difference between predicted and true label and defined as the *loss function* $L^{(p)}(w) = L(y^{(p)}_{\text{true}}, f(x^{(p)}, w))$. Intuitively, a well designed model is one that yields a low loss and vice versa.

While the loss function itself can be defined arbitrarily, the procedure stays the same. In the following, this is exemplified by one of the most commonly used classifiers, namely the *softmax classifier*. It relies on the MLP model as defined in equation 3.6. The activations in the output layer, so-called *scores* can be understood to be the unnormalized *log*-probabilities

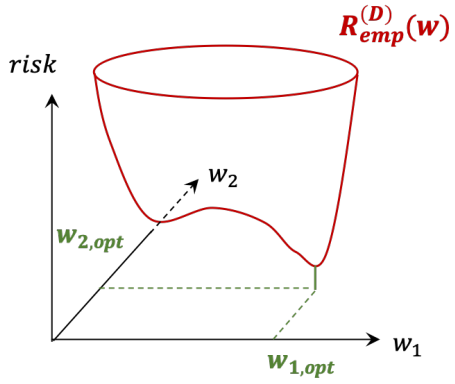

Figure 3.5: The empirical risk function can be imagined as a surface that is spanned in the space of the parameter. Adjusting the parameter of a neural network is equivalent to find the deepest point of this surface.

for each class. Hence, before calculating the loss a normalization is executed by applying the *softmax function*

$$\begin{aligned} \sigma : \mathbb{R}^K &\mapsto \{\sigma \in \mathbb{R}^K | \sigma_i > 0, \sum_{i=1}^{K} \sigma_i = 1\} \\ \sigma(f(x,w))_j &= \frac{e^{f(x,w)_j}}{\sum_{k=1}^{K} e^{f(x,w)_k}} \text{ for } j = 1, ..., K \end{aligned} \tag{3.7}$$

as the activation function to the scores. It takes a K-dimensional vector $f(x,w)$ of arbitrary real-valued scores and squashes it into a K-dimensional vector of values between 0 and 1 that sum to one. Finally, the loss in one data point $x^{(p)}$ is derived as

$$L^{(p)}(w) = -\ln \sigma(f(x,w))_l \tag{3.8}$$

where $\ln(\cdot)$ is the logarithm with natural basis. Moreover, $f(x,w))_l$ is the predicted value that corresponds to the true label. Often, loss 3.8 is called *cross-entropy loss*.

Summing over all data points gives the full data loss, defined as

$$R_{\text{emp}}^{(D)}(w) = \frac{1}{D} \sum_{p=1}^{D} L^{(p)}(w) \quad . \tag{3.9}$$

Next to data loss, this quantity has multiple names including *cost function*, *empirical risk function* or simply *training error*. In many cases an extended loss function is applied however the plain loss as in 3.9 is sufficient to obtain a basic understanding.

3.3. Optimization

The *softmax* classifier is an instance of a MLP whose performance is measured based on the *cross-entropy loss* as defined in equation 3.8. The higher the value, the more labels are wrongly predicted by the model. Although a perfect model would always predict the true label, this state of perfection is not feasible. Instead, an optimal model makes as few false predictions as possible what is equivalent to minimize the empirical loss 3.9. As depicted in figure 3.5, for a given model and the chosen loss function, the cost function depends on the number D of observations and the model parameters $w_{ij}^{(n)}$. Hence, adjusting the parameters is a method of controlling the performance. In terms of neural networks, this phase of parameter adaption is called *training* or *learning from examples*. One faces the optimization problem

$$w_{ij}^{(n)} = \arg\min_{w \in \Lambda} \left(R_{\text{emp}}^{(D)}(w) \right) \tag{3.10}$$

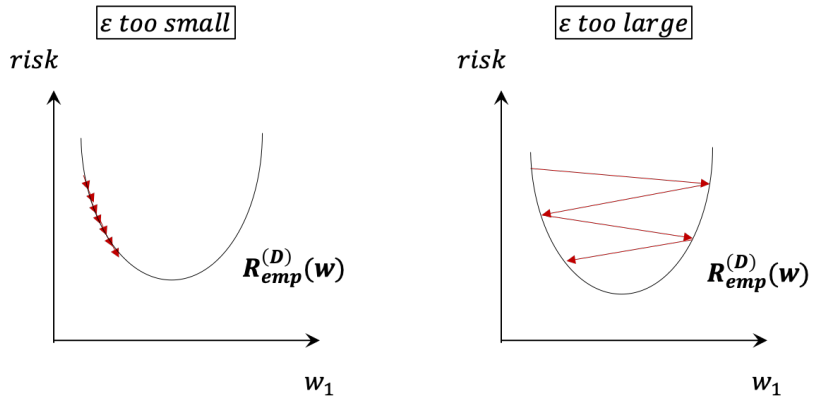

Figure 3.6: The choice of the hyperparameter ϵ is essential for the convergence behaviour of BGD. It is a hyperparameter and is interpreted as the step size in direction of steepest descent. *(LEFT)* Too small values slow convergence too strongly down while *(RIGHT)* too large values might lead to oscillations.

The branch of optimization is wide and this section is devoted to a very basic introduction of essential algorithms. However, in the course of this report more elaborated algorithms are applied whose short descriptions can be found on the Appendix **??**. Most often, neural networks are optimized iteratively.

Generally speaking, in every iteration step τ the weights are adjusted according to

$$w^{\tau+1} = w^{\tau} + \Delta w^{\tau} \tag{3.11}$$

where Δw^{τ} is the weight update. In particular, different algorithms such as *batch gradient descent* (BGD), *stochastic gradient descent* (SGD) or *momentum learning* (MOM) use different weight updates Δw^{τ}.

Batch Gradient Descent is the simplest algorithm. Here, the weights are chosen to be a step in the direction of the negative gradient of the empirical risk. The term *batch* indicates that the error is evaluated with respect to the whole data set. Iteration scheme 3.11 specifies to

$$w^{\tau+1} = w^{\tau} - \epsilon \nabla_w R^{(D)}_{\text{emp}}(w^{\tau}) \tag{3.12}$$

where the learning rate ϵ determines the size of the step in direction $-R^{(D)}_{\text{emp}}(w^{\tau})$. The learning rate is a so-called *hyperparameter*. Such parameters strongly affect the performance of the MLP but are neither learnable nor easy to estimate. However, they must be set before the learning process starts. For this reason, much time is invested in finding and adjusting the values of hyperparameters [49].

As becomes apparent in figure 3.6, the learning rate has a crucial influence on the convergence behaviour of 3.12. Hence, it has to be chosen wisely.

This additional fine-tuning effort is one characteristic disadvantage of BGD. A further disadvantage is the computation intensity, which is large since the whole data set has to be evaluated. As common for deterministic gradient-based optimization methods, BGD features the unfavorable property that it might get stuck in local minima.

This can be remedied by adjusting the number of data points that are taken into account while evaluating the risk. *Online Gradient Descent* is an extreme case and evaluates only one observation at a time. Although the term *online* is normally reserved for methods, where observations are drawn from a stream of continuously generated examples, it simply indicates that, here the error is evaluated with respect to only one data point. This introduces some noise to the parameter update. This is why Online Gradient Descent methods are more often known as *Stochastic Gradient Descent* (SGD) approaches. Opposed to that, BGD is a deterministic algorithm. As can be see in the iteration scheme 3.13, this randomness is controlled by the parameter ϵ_{τ}. Again, this hyperparameter is called *learning rate* but in contrast to BGD it now depends on the iteration step τ.

$$\begin{aligned} w^{\tau+1} = w^{\tau} - \epsilon_{\tau} \nabla_w R^{(m)}_{\text{emp}}(w^{\tau}) \quad \text{with} \quad & m \in [1, M], M << D \\ & m = 1 : \text{SGD} \\ & m > 1 : \text{Mini-batch GD} \end{aligned} \tag{3.13}$$

This is due to the fact, that convergence is only assured if the effect of noise is diminished during the iteration. Hence, a so-called *annealing scheme* takes care of this by gradually decreasing ϵ_τ.

Although a wide variety of learning rate decay methods exist, one mainly comes across the same three [49]:

1. *Step Decay* simply reduces the learning rate after a certain number of iterations, e.g. by halving it.
2. *Exponential Decay* with $\epsilon_\tau = \epsilon_0 \cdot e^{-k\cdot\tau}$ where k and ϵ_0 are two further hyperparameters.
3. $\frac{1}{\tau}$ *decay* has the mathematical formulation $\epsilon_\tau = \frac{\epsilon_0}{(1+k\cdot\tau)}$ with k and ϵ_0 again hyperparameters.

Finally, which of those is the right annealing schedule and how to adjust its hyperparameters depends on the given network and learning tasks and there is not one uniformly superior one. Hence, evaluating only one observation is boon and bane at once. Although it allows the algorithm 3.13 to escape from local minima and find the global one, it causes additional work due to the necessity to design an annealing schedule.

Both algorithms, BGD as well as SGD suffer from certain disadvantages which are diminished by creating a hybrid approach. The *Mini-Batch Gradient Descent* (Mini BG) method is based on the iteration scheme 3.13 and evaluates a small number m of observations at once. While SGD sets $m = 1$ and BDG uses $m = D$, the Mini BG choses an appropriate value of m that is somewhere in between.

3.3.1. Backpropagation

Obviously, there exist a vast number of different approaches how to specify the general iteration scheme 3.11. The previous section introduced instances of gradient descent methods which all rely on the same idea: the update of a particular parameter depends on how strongly it affects the performance, hence the loss of the MLP. This is measured by the gradient of the empirical risk function with respect to this parameter. The larger this gradient is, the more sensitive does the MLP react on changes of this parameter.

However, the graphical architecture of neural networks naturally hampers this approach. As the empirical risk function depends on the outputs $f(x,w)_k$, its gradient can easily be derived with respect to the parameters of the last layer. But how can the gradient information with respect to parameters of the first layer be derived?

Although their core ideas were already investigated in the early 1950, this question paralysed the development of neural networks for decades. The turning point was the appearance of *backpropagation*, introduced by various independent persons. It gives an applicable answer to the question on propagating gradient information [15].

Backpropagation (BP) is an efficient technique that applies the idea of the chain rule to evaluate the gradients of the empirical risk

$$\nabla_{w_{ij}^{(l)}} R_{\text{emp}}^{(m)}(w_{ij}^{(l),\tau}) \tag{3.14}$$

with respect to the parameter $w_{ij}^{(l),\tau}$ of the hidden layer l. Each iteration step τ contains one propagation which in turn consists of one *forward* and one *backward* sweep. As shown in figure 3.7, they differ in the direction in which information is propagated in the network. For simplicity, the iteration parameter τ will be omitted.

According to figure 3.8, the forward sweep processes information on the input by consecutive hidden layers. In the final layer, the empirical risk that m input observations yield with the current set of parameters of iteration step τ is calculated.

The backward sweep aims at deriving the gradient of this risk with respect to a particular parameter by backpropagating the error in the opposite direction. The implementation is

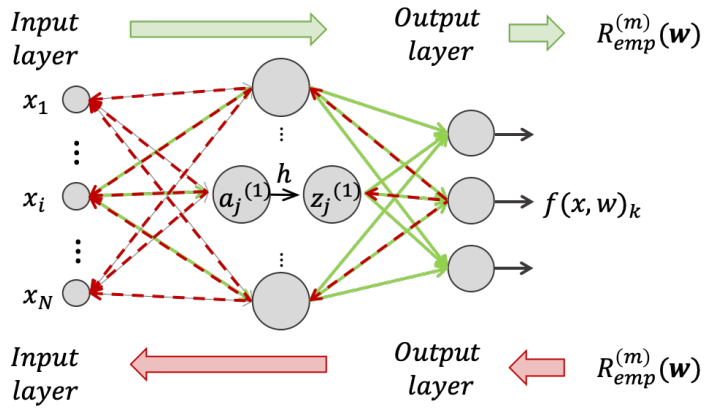

Figure 3.7: The green colored *forward sweep* denotes the direction of information that travels from the input to the output side. Intuitively, this is the "normal" working direction of neural networks since it is also used to derive the desired predictions $f(x,w)_k$. However, to adapt the parameters w_{ij} based on gradients of the empirical risk $R_{\text{emp}}^{(m)}(w)$, the direction of the information flow can be reversed. This *backward sweep* is denoted by the red arrows and points from output to input. Both sweeps together form one backpropagation step.

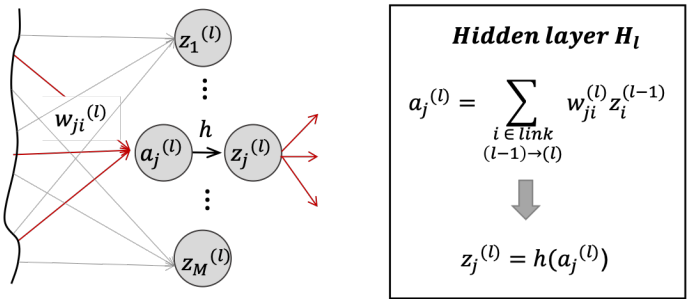

Figure 3.8: The hidden layer l receives its input $z_i^{(l-1)}$ from the units of the preceding layer $(l-1)$. Linearly combining the with its parameters $w_{ij}^{(l)}$ results in *activations* $a_j^{(l)}$. The final output $z_j^{(l)}$ of the layer is yield by applying the nonlinear activation function $h(\cdot)$. Often, $z_j^{(l)}$ is referred to as *hidden units*.

exemplified by a simple MLP with two hidden layers. It adapts parameters with SGD and since $m = 1$, $R_{\text{emp}}^{(m)}$ simplifies to $L^p(w) = L$.

In accordance with figure 3.8, the output layer is interpreted as a hidden layer $l = 2$. When evaluating the derivatives of L with respect to the parameter $w_{kj}^{(2)}$ it is important to see that the error only depends on $w_{kj}^{(2)}$ via the summed input $a_k^{(2)}$ to unit k. Therefore, the chain rule for partial derivatives gives

$$\frac{\partial L}{\partial w_{kj}^{(2)}} = \frac{\partial L}{\partial a_k^{(2)}} \frac{\partial a_k^{(2)}}{\partial w_{kj}^{(2)}}. \tag{3.15}$$

Using the definitions of *activations* $a_j^{(l)}$ as shown in figure 3.8 and introducing an abbreviation for the second derivative in the above equation allows the rewriting of the two factors as

$$\frac{\partial a_k^{(2)}}{\partial w_{kj}^{(2)}} = z_j^{(1)} \quad \text{and} \quad \delta_k^{(2)} := \frac{\partial L}{\partial a_k^{(2)}} \quad . \tag{3.16}$$

Hence, for parameter in the output layer it is

$$\frac{\partial L}{\partial w_{kj}^{(2)}} = z_j^{(1)} \delta_k^{(2)}. \tag{3.17}$$

Obviously, the partial derivative of the error with respect to $w_{kj}^{(2)}$ depends on two quantities, namely, the value of $z_j^{(1)}$ on the input side of the weight and the error derivative $\delta_k^{(2)}$ on its output end. The latter can easily be derived with the output of the network. Moreover, $\delta_k^{(2)}$ gives the relation between the error L and the inputs $a_k^{(2)}$ of the output layer. As will become relevant later, keep in mind that these inputs are linearly correlated to the outputs $z_j^{(1)}$ of the preceding hidden layer 1. Consequently, $\delta_k^{(2)}$ also reveals the relation between the loss and the values $z_j^{(1)}$ [11].

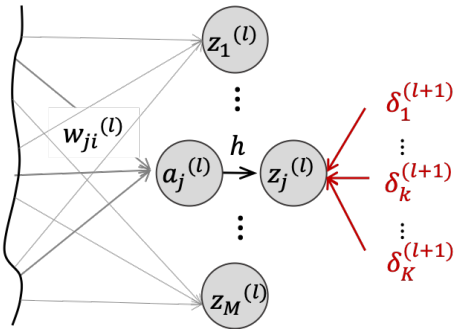

Figure 3.9: The figure shows the training of a deep neural network via gradient descent methods with BP. To derive the error derivative in the hidden layer l with respect to the parameter $w_{ij}^{(l)}$ the chain rule is repeatedly applied. This allows to express the desired derivative in terms of the outputs $z_j^{(l)}$. However, these are connected with the final loss L via the higher layers. This is indicated by the quantities $\delta_k^{(l+1)}$ with $k = 1, \dots, K$.

Computing the error derivative with respect to parameters $w_{ji}^{(1)}$ of the first hidden layer follows the same scheme. Again, it is

$$\frac{\partial L}{\partial w_{ji}^{(1)}} = \frac{\partial L}{\partial a_j^{(1)}} \frac{\partial a_j^{(1)}}{\partial w_{ji}^{(1)}} \tag{3.18}$$

with

$$\frac{\partial a_j^{(1)}}{\partial w_{ji}^{(1)}} = x_i \quad \text{and} \quad \delta_j^{(1)} := \frac{\partial L}{\partial a_j^{(1)}} \quad . \tag{3.19}$$

As before, the error derivative composes two factors, x_i on the input side of the weight $w_{ji}^{(1)}$ and the error derivative $\delta_j^{(1)}$ on the output side. However, computing $\delta_j^{(1)}$ for the inner layers is not that trivial anymore. The loss L depends on the activations $a_j^{(1)}$ via the output $z_j^{(1)}$ of the layer. Applying the chain rule, decomposes $\delta_j^{(1)}$ into two factors according to

$$\delta_j^{(1)} := \frac{\partial L}{\partial a_j^{(1)}} = \frac{\partial L}{\partial z_j^{(1)}} \frac{\partial z_j^{(1)}}{\partial a_j^{(1)}}. \tag{3.20}$$

While the second term exists for every differentiable activation function $h(\cdot)$, the first factor depends on information of higher layers. This clarifies by using the chain rule once more, since

$$\frac{\partial L}{\partial z_j^{(1)}} = \frac{\partial L}{\partial a_k^{(2)}} \frac{\partial a_k^{(2)}}{\partial z_j^{(1)}}. \tag{3.21}$$

Combining this with the preceding considerations, it becomes evident that the relation between L and $z_j^{(1)}$ can be formulated in terms of $\delta_k^{(2)}$. Figure 3.9 shows that the output $z_j^{(1)}$ of neuron j in layer 1 is in sum affected by all neurons of the higher level with which it is connected. This leads to

$$\frac{\partial L}{\partial z_j^{(1)}} = \frac{\partial L}{\partial a_k^{(2)}} \frac{\partial a_k^{(2)}}{\partial z_j^{(1)}} = \sum_{k \in \text{links layer } 2 \mapsto 1} \delta_k^{(2)} w_{kj}^{(2)}. \tag{3.22}$$

For a general deep neural network with more than two layers, this perceptions are merged in the following equations:

$$\frac{\partial L}{\partial w_{ji}^{(l)}} = \frac{\partial L}{\partial a_j^{(l)}} \frac{\partial a_j^{(l)}}{\partial w_{ji}^{(l)}} = \delta_j^{(l)} z_i^{l-1} \tag{3.23}$$

and

$$\delta_j^{(l)} = \frac{\partial z_j^{(l)}}{\partial a_j^{(l)}} \sum_{k \in \text{links layer } l+1 \mapsto l} \delta_k^{(l+1)} w_{kj}^{(l+1)}. \tag{3.24}$$

These BP formulas allow an easy calculation of the error derivative with respect to any parameter by using information from higher layers [11].

3.4. Validation

By introducing the BP, the last milestone of the MLP model is laid. Assuming again a classification model for the MNIST data set. Within the above framework, a handful of observations of the whole set is fed into the neural network and its parameters are iteratively adapted based on its performances. Any stopping criteria can be formulated to define, when this parameter updating should terminate. A fixed number of iterations, a maximal computing time or low error rate are only three of numerous examples [42].

A successful training, results into a network whose parameters are optimized for the examples by which is was trained. Hence, whenever one of the observations that were used during training is represented to the MLP, it will predict its label quite well.

A good MNIST classifier however, is one that performs well on any arbitrary input that is taken from the MNIST set. The *validation stage* evaluates how well this requirement is met by the designed MLP.

This phase is not only useful to check whether the trained model can be applied to the original classification task but also to fine-tune hyperparameters.

In general, the disjoint *training* and *test* or *validation data set* are used for each of the two phases of *training* and *validation*.

One very simple method for validation is the *Set Method*, where two distinct data sets are given. The training data $\{x^{(\alpha)}, y^{(\alpha)}_{\text{true}}\}$ with $\alpha = 1, \cdots, p$ is only used to select model parameters. The validation data $\{x^{(\beta)}, y^{(\beta)}_{\text{true}}\}$ with $\beta = 1, \cdots, q$ is uniquely used to evaluate the selected parameters [42].

A further approach is the *k-fold cross validation*. This is a special variation of *cross validation* where a given data set is partitioned into complementary subsets. One subset is used as training data while the other is used for validation.

k-fold cross validation is an iterative version, where all observations D are divided into k disjunct subsets D_j such that $\bigcup_{j=1}^{k} D_j$. Of the k subsamples, one is retained as validation set while $k-1$ others are merged and used for training. This is repeated until each subset was used for validation once. Finally, the model is evaluated k times and averaging these results gives one final estimation of the model performance.

This approach fully exploits the set of observations D since each observation is used for training as well as for validation. However, a noteworthy disadvantage is its high computation intensity [42, 49].

3.5. From Shallow to Deep Neural Networks

It seems as if the MLP is only a stone's throw away from the desired deep neural networks. All, that needs to be done is adding hidden layers. However for a long time, increasing the number of these layers was avoided and science mainly put effort in shallow neural networks as the presented MLP. For the start, Kolmogorov stated 1965 that a neural network with a single layer of enough hidden units can approximate any multivariate continuous function with arbitrary accuracy [6]. By the end of the 1980s, training based on gradients worked well for shallow problems. However, it seemed that in contempt of using BP, more than a few hidden layers do not offer empirical benefit. This insight in addition with Kolmogorov's theorem pushed the computer vision community even further away from deep neural nets.

1991, Hochreiter identified the major hurdles that impede training of typical deep neural networks by BP. Combined with standard, saturating activation functions, the cumulative backpropagated error signals either vanish or explode. Their evolution while backpropagated through deep hidden layers depends on the size of the network parameters. Large gradients may cause oscillating weights. In the case of gradients close to zero, the learning process becomes either prohibitively long or even unfeasible. During the following years, different ways to cure this problem of deep learning arose. Despite of discussing alternatives to gradient descent based training methods, the lately competition-winning deep neural networks stayed with this principle [22, 45]. The approach that finally settled down, combines *GPU-*

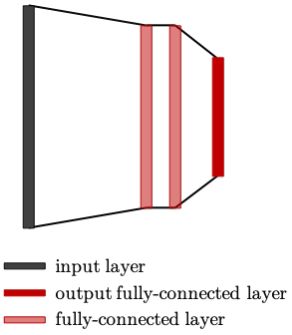

Figure 3.10: The schema illustrates the architecture of my implemented fully-connected neural network. The grey layers denote the input side, while shades of red mark fully-connected layers. I distinguish between fully-connected hidden layers and the fully-connected layer which is used for the final classification.

based computation (*GPU*: Graphics Processing Units) with adjusted activation functions and advanced optimization schemes.

The final breakthrough of deep neural networks became apparent in 2011 when Dan Ciresan et al. trained a neural network with a GPU for the first time [7]. From then on, improvements arose blow on blow. 2012, Krizhevsky et al. stated that in terms of training time with gradient descent methods, saturating nonlinearities perform worse than non-saturating. The proposed ReLU activation function, as shown in figure 3.4 *(LEFT)* can be found in most of the latest deep neural networks [27].

In addition to that, improved optimization schemes and parameter update stage were developed. These include momentum update or Adam, as introduced in the appendix A.

3.6. A MLP as MNIST Classifier

With the knowledge from the preceding section, I implemented a first prototype of an image processing MLP for the MNIST classification problem. The corresponding *Tensorflow* code can be found in the appendix B.

As shown in figure 3.10, the model is solely based on fully-connected layers. Each of the hidden layers comprise 2048 units with a ReLU activation function. This is followed by a third fully-connected layer, which applies the *softmax*-activation function to finally derive the ten class predictions. Summarizing all, the neural network has a total number of 5 824 522 parameters.

In accordance with the preceding section 3.2, the *cross-entropy* loss measures the performance. Moreover, I try to circumvent the difficult adjustment of the learning rate, by optimizing the parameter with the *Adams*-algorithm, as described in the appendix A.

In machine learning, an *epoch* is one complete presentation of the data set which is to be learned by a learning machine. Within one epoch, the MNIST classifier iterates over all 60 000 input images. Hence, training the MLP over 10 epochs is equivalent to feeding it 10 times with each training example. This results into a cross-entropy loss as shown in figure 3.11. Obviously, it shrinks and the final parameter setting predicts the correct digits with a probability of 99.72%.

This final accuracy seem to expose the set of parameters to be well-chosen. However, when evaluating the model over the same number of epochs for 10 000 test examples, this high performance can not be met. Only 97.15 % of all test images are recognized correctly.

In general, the undesired phenomena of an insufficient validation performance in contempt of a low training error is called *overfitting*. Before chapter 5 introduces efficient techniques to counteract it, the *statistical learning theory* provides a deeper insight into its entity.

In the remainder of this report, I use this neural network as the base model to examine the effect and strength of methods that are introduced to prevent overfitting.

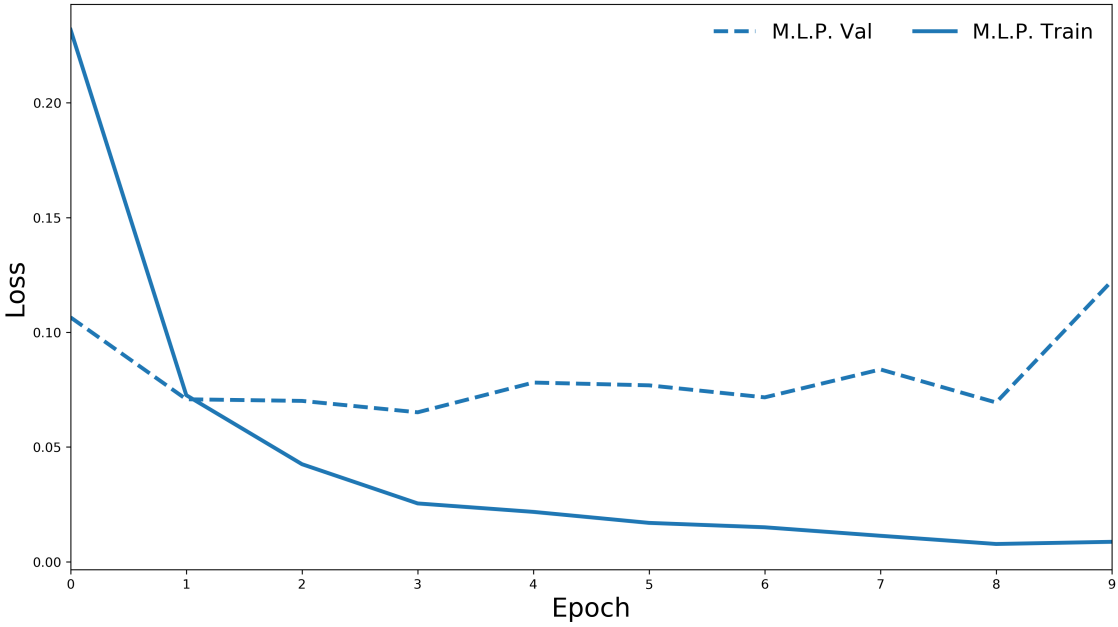

Figure 3.11: Training my neural network over 10 epochs with 60 000 examples yields the solid blue graph. Evaluating it over the same number of epochs for the 10 000 test examples leads a loss that is depicted by the dashed line.

4

Statistical Learning Theory

Two essential parts of designing a neural network are the *training* and *validation* phase. The first is necessary to adapt the model parameters. In principle, this is based on evaluating a performance measure of examples that are taken from the training set.

The validation stage takes a fixed set of trained parameters and examines whether the neural network successfully fulfills the learning task. This is done, by computing a performance measure yet, using a different set of observations.

At the first moment, a two-folded computation of an error might seems to be redundant and needless. However, training and evaluating the MNIST classifier in the previous chapter proved this assumption wrong. The graphs in figure 3.11 give the development of the loss during those two phases. While longer training increases the prediction accuracy, the performance during validation first saturates and then even starts to worsens.

Clearly, these observations impose several burning questions: How do the performance measures differ from each other? Why does the validation performance measure increase? Which of them really captures how well the network solves its task?

Examining *learning* from a statistical point of view gives the promising answers. The so-called *Statistical Learning Theory* (SLT) characterizes a machine learning model based on how well it *learns* and *generalizes* from data. In other words, its ability to use the knowledge that is retrieved from the training data to infer statements about any other point that originates from the same process. This is the true learning task of machine learning systems.

While the *learning ability* is analysed by asymptotic considerations of infinitely large training sets, *generalization* is related to its non-asymptotic behaviour. These characteristics are quantified by a *capacity* concept.

The framework of SLT allows the description of a model behaviour during learning based on its generalization ability. For instance, the model in figure 3.11 *overfits*. Both, *overfitting* as well as its counterpart *underfitting* are equally harmful for the generalization performance.

In the following section, the difference between training and validation loss is derived and a deeper insight into the origin of over- and underfitting is gained. Based on that, axioms that are useful for training and designing a machine learning system are formulated. All formulas are based on the paper *An overview of statistical learning theory* by V. Vapnik [54].

4.1. Learning Problem

Machine learning systems such as neural networks are so-called models of learning from examples. Their framework composes three elements:

1. a fixed but unknown *data-generating distribution* $P(x)$ from which *independently, identically distributed* (iid) random vectors x are drawn

2. a likewise fixed but unknown *conditional distribution* $P(y|x)$ that returns a respond y to every input x
3. the *learning algorithm* $f(x,w), w \in \Lambda$ described by set of functions that predict the response to an unknown input x_{test}

Within this terminology, the components of neural networks as introduced in section **??** are reformulated as follows.

The *training set* consists of D random observations

$$(x^{(1)}, y^{(1)}), \dots, (x^{(D)}, y^{(D)}) \tag{4.1}$$

that are iid drawn from the joint distribution $P(x,y) = P(y|x)P(x), P(x,y) \in \Pi$. Moreover, SLT assumes that the *test set* is drawn from the same distribution $P(x)$ as the training examples.

The general *loss function* $L(w) = L(y, f(x,w)), w \in \Lambda$ measures the discrepancy between the true response and the learning machine's prediction of one particular example x. Hence, it quantifies the error that the learning algorithm $f(x,w), w \in \Lambda$ makes when predicting the response. The expected value of this loss, given by the *risk functional* or short *risk*

$$R(w) = \int L(y, f(x,w)) dP(x,y) \tag{4.2}$$

determines the error for any arbitrary input pair $(x,y) \sim P(x,y)$.

Based on these definitions, the problem of learning is to find the function $f(x,w_0), w_0 \in \Lambda$ that minimizes the risk $R(w)$. This is equivalent to finding the function $f(x,w_0), w_0 \in \Lambda$ which approximates the unknown conditional distribution $P(y|x)$ the best.

However, this is hampered by the fact that the joint probability $P(x,y)$ is unknown and the only available information is contained in the empirical data 4.1 [54].

4.2. Empirical Risk Minimization

Since $P(x,y)$ is unknown, the expected value in 4.2 can not be computed exactly but only approximated. One approach to achieve this is the *Empirical Risk Minimization Induction Principle* (ERM) which retrieves the maximal available information by exploiting the training set 4.1. It is based on substituting the integral in 4.2 by a sum and defines an *empirical risk* as

$$R_{\text{emp}}(w) = \frac{1}{D} \sum_{i=1}^{D} L(y^{(i)}, f(x^{(i)}, w)) \quad . \tag{4.3}$$

As shown in figure 4.1, the idea of ERM is to approximate the final learning algorithm

$$f(x, w_0), w_0 \in \Lambda \quad \text{with} \quad w_0 = \arg\min_{w \in \Lambda}(R(w))$$

by a function

$$f(x, \tilde{w}) \quad \text{with} \quad \tilde{w} = \arg\min_{w \in \Lambda}(R_{\text{emp}}(w))$$

with $\tilde{w} \in \Lambda$, which minimizes the empirical risk.

Comparing equation 4.3 to the cost function 3.9 shows that the empirical risk has already appeared in section 3.2 in form of the *training error*. During training, a MLP minimizes it in order to adjust its parameter. In the following evaluation phase, the neural network is applied to a validation set and the test error is measured. In SLT, the previously unobserved data points of the validation set can be interpreted as any arbitrary sample that originates from the same distribution as the training examples. Hence, the validation error corresponds to the risk as defined in 4.2 and gives the ability of the neural net to learn from data. For this reason, it is referred to as the *generalization error*. A machine learning system is considered

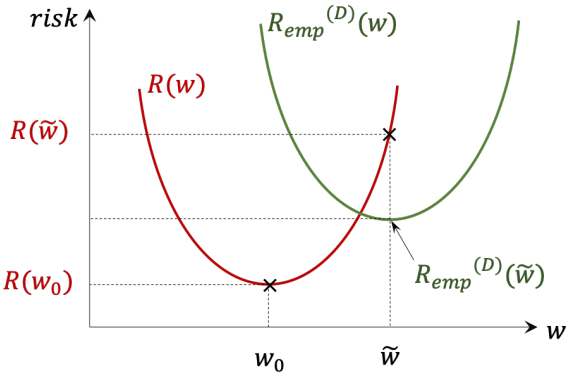

Figure 4.1: The ERM principle approximates the finally sought function $f(x, w_0)$ that minimizes the risk $R(w)$ by a further function $f(x, \tilde{w})$ that minimizes the empirical risk instead. This so-called learning from data is if ERM is consistent.

to be good if the set of parameter that is retrieved from data yields a minimal generalization error.

Obviously, MLPs are trained in accordance with the ERM. While chapter 3 examines its implementation step by step, this chapter asks for the justification of ERM based learning. Is it reasonable to assume that minimizing the empirical risk leads to the desired minimal generalization error?

The SLT states that learning from data is guaranteed if the ERM is *consistent* over the set of functions $f(x, w), w \in \Lambda$, i.e. if

$$\lim_{D\to\infty} P\left(|R(w_0) - R(\tilde{w})| > \eta\right) = 0 \quad \forall \eta > 0 \quad . \tag{4.4}$$

At the same time, the *Key Theorem of SLT* links this consistency with convergence and consequently condition 4.4 is satisfied if

$$\lim_{D\to\infty} P\left(\sup_{w\in\Lambda}(R(w) - R_{\text{emp}}(w)) > \epsilon\right) = 0 \quad \forall \epsilon \quad . \tag{4.5}$$

To quantify bounds on the convergence of the empirical to the actual risk so-called *capacity* concepts are introduced. These are essential when deriving necessary and sufficient conditions under which the ERM principle is applicable. The capacity of a given set of function $f(x, w), w \in \Lambda$ quantifies its ability to learn from data [54].

4.3. Capacity, Over- and Underfitting

A commonly used capacity measure are the so-called *Vapnik-Chervonenkis*-dimensions (VC) h. Despite their principle is valid for sets of general functions, they were originally defined for indicator functions [54].

Image 4.2 shows that such indicator functions $f(x, w)$ are half planes in $\mathbb{R}^d$ whose position in the d-dimensional space depends on the parameters $w \in \Lambda$. They are for instance used to separate a set of D data points into two classes. Naturally, there exist 2^D unique possibilities to separate a set of D points into two categories. The VC dimensions h capture the maximal number D_{max} of data points that these indicator functions can uniquely classify in all its $2^{D_{max}}$ possible ways. It is $h = D_{max}$.

Based on this definition and as further explained in figure 4.2, the classifier in image **??** has VC dimension $h = 3$. The situation in 4.2 shows that data sets which contain more than h examples restrict the choice of possible classifiers and hence enforce a selection. This is learning from data.

With this perception, a more intuitive idea of the necessary and sufficient conditions for learning from data i.e. for the ERM can be derived. This ability is given when the VC-dimensions of the set of functions $f(x, w), w \in \Lambda$ is *finite*.

Asymptotic considerations for samples of infinite size D clarify the circumstances under which learning is possible. However, to determine its quality it is interpretation as a problem

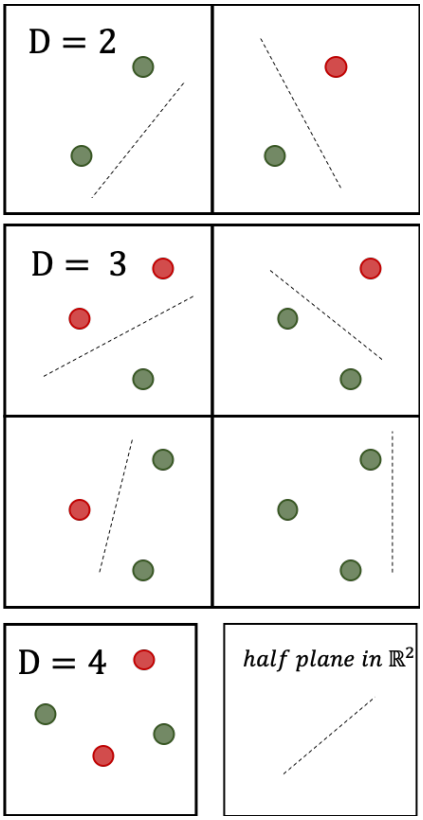

Figure 4.2: *(BOTTOM RIGHT)* Indicator functions $f(x,w)$ are half planes in $\mathbb{R}^d$. For $d = 2$, as denoted by the dashed line are planes they reduce to planes whose angle and position are adjusted by the parameters $w \in \Lambda$. They separate a set of D points into a green and a red colored class. The VC dimension h gives the maximal number D_{max} of data points that can be separated in its $2^{D_{max}}$ unique classifications by the given set of indicator functions. Hence, the sample of $h = D_{max}$ points is *shattered* by the functions. *(TOP)* A plane can uniquely separate $D = 2$ points in $2^D = 4$ ways. Note, that the picture only shows two of these classifications. The missing ones arise when inverting the color categories. *(MIDDLE)* Again, the half planes are capable of assigning the $D = 3$ points in $2^3 = 8$ unique ways two one of the two classes. For the same reasoning as before, only 4 of the 8 classifications are illustrated. *(BOTTOM LEFT)* For $D = 4$ points, constellations can appear which can not be uniquely classified. Hence, the VC dimension of indicator functions in $\mathbb{R}^2$ is $h = 3$.

of approximating the unknown conditional distribution $P(x|y)$ by $f(x,w)$ is reused. The capacity of the model directly determines the complexity of the approximation function $f(x,w)$. Ideally, a high-quality approximation is similarly complex as its solution. Models with low values of h are too simple to reproduce the true distribution and hence features a bad generalization ability. Opposed to that, machine learning systems of high capacity are modelled by a large number of parameters and can approximate more complex distributions. Small training sets however, do not provide enough information to perfectly adjust all w. Again the generalization error is large.

Obviously, the generalization ability does not only depend on h but also on the size D of the training set. A non-asymptotic analysis of $R(w)$ for finite D shows that, with probability $1-\eta$ the generalization error of a set of bounded functions $0 \leq L(y, f(x,w)) \leq 1$ is limited by

$$R(w) \leq R_{\text{emp}}^{(D)}(w) + \frac{\epsilon}{2}\left(1 + \sqrt{1 + \frac{4R_{\text{emp}}^{(D)}(w)}{\epsilon}}\right) \tag{4.6}$$

with

$$\epsilon = 4\frac{h\left(\ln\frac{2l}{h} + 1\right) - \ln\eta}{D} \quad . \tag{4.7}$$

Compromising the second summand in the function

$$C(\frac{h}{D}) = \frac{\epsilon}{2}\left(1 + \sqrt{1 + \frac{4R_{\text{emp}}^{(D)}(w)}{\epsilon}}\right) \tag{4.8}$$

allows the reformulation of the equation 4.6 as

$$R(w) \leq R_{\text{emp}}^{(D)}(w) + C\left(\frac{h}{D}\right) \quad . \tag{4.9}$$

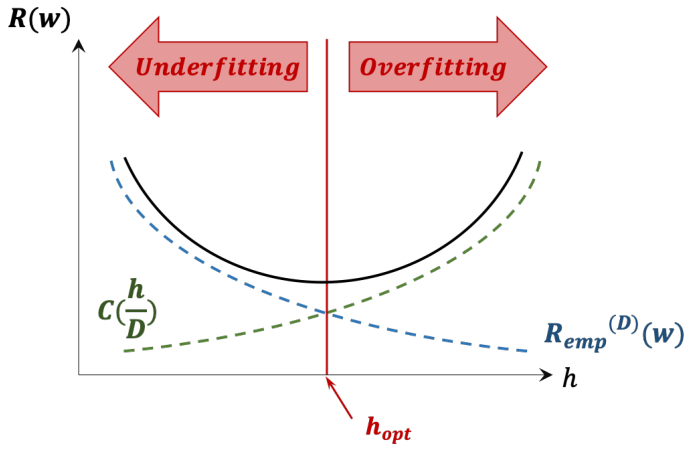

Figure 4.3: The relation $\frac{D}{h}$ heavily impacts the generalization error $R(w)$. The blacj graph denotes its development for fixed D and increasing capacity h, hence for decreasing $\frac{D}{h}$. Moreover, the dashed lines give the contributions of the complexity term and empirical risk. In situation in which the gap between those is big are even called *over-* or *underfitting*, depending on whether $C(\frac{h}{D})$ or $R_{\text{emp}}^{(D)}(w)$ dominate $R(w)$.

Consequently, the bound of the generalization error depend on the empirical risk $R_{\text{emp}}^{(D)}(w)$ as well as on a complexity function $C(\frac{h}{D})$.

Figure 4.3 shows the schematic development of the terms for a fixed sample size D and growing h. First of all, the two extreme cases in which h is either very low or high are analyzed. For both, the generalization error is large and considering the contributions of $R_{\text{emp}}^{(D)}(w)$ and $C(\frac{h}{D})$ reveals that in both situations the gap between them is huge.

For small h the complexity term is very low and the generalization error mainly consists of the empirical risk. This situation is know as *underfitting* and can easily be tackled by the ERM. Thus, for high values of $\frac{D}{h}$ vanishing training errors $R_{\text{emp}}^{(D)}(w)$ cause minimal generalization error $R(w)$.

In the reverse situation for high h, i.e. small $\frac{D}{h}$ the gap between $R_{\text{emp}}^{(D)}(w)$ and $C(\frac{h}{D})$ arises from large complexity terms. Hence, the parameters of the model fit nearly perfectly but exclusively to the given training set. Such an adaption of the parameter to the training examples is that accurate that the model even captures singularities of the training set, which are not representative for general data point (x, y). This so-called *overfitting* leads to high generalization errors $R(w)$. Here, the distance between the two error terms can not be bridged by ERM. Given this situation, reducing $R_{\text{emp}}^{(D)}(w)$ is not sufficient to minimize $R(w)$. Even for $R_{\text{emp}}^{(D)}(w) \mapsto 0$, the complexity term $C(\frac{h}{D})$ sets a bound on the generalization error.

This shortcoming of the ERM principle motivates the development of a more elaborate approach to minimize the generalization error, namely the *Structural Risk Minimization* (SRM).

4.4. Structural Risk Minimization

The SRM can be understood as an extension of the ERM, which does not solely minimize the empirical risk but also the capacity of the model and, hence, it reduces the undesirable gap between $C(\frac{h}{D})$ and $R_{\text{emp}}^{(D)}(w)$. To accomplish this, the model complexity is treated by imposing a structure on the set of functions $f(x, w), w \in \Lambda$.

Assume, for instance, that a learning machine has to select a model for a binary classifier for the observations as given in figure 4.4. In this case, a structured set S_k of all possible models are polygonal chains with k line segments. Figure 4.4 shows that k can be utilized to control the complexity of the model.

The simplest model, for $k = 1$ is described by one line. Its parameters can be adjusted such that at least 9 observations are assigned incorrectly. The simple model, with a high training error generalizes badly.

This is tackled by further increasing the complexity and adding a second line segment. The optimal model in this set S_2 reduces the missclassification rate to 5. Hence, a relatively low training error while maintaining simplicity is achieved and the model generalizes sufficiently.

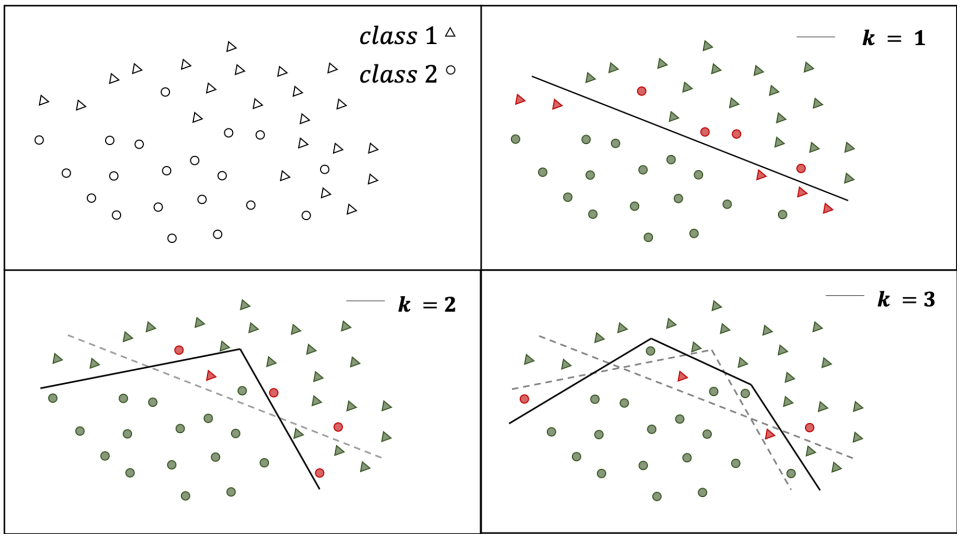

Figure 4.4: Depicted are the applications of three classifiers of the set S_k of k polygonal chains whose complexity is stepwise increased. *(TOP LEFT)* The learning task is to separate the given oberservations into its two calsses, a triangular and a circular one. *(TOP RIGHT)* For $k = 1$ the classifier is very simple and the training error with 9 missclassifications is high. This is insufficient for generalization. *(BOTTOM LEFT)* Increasing k to 2 reduces the training error to 5 missclassifications while keeping the model simple, hence that generalization error suffice. *(BOTTOM RIGHT)* Trying to further decrease the trainging error to 4 missclassifications by increasing $k = 3$ results in a too high complexity and generalization is again insufficient.

The training error can be further decreased by applying more complex models which are composed of three lines. Indeed, the training error lowers to only 4 wrong classification. However, due to the increase in complexity the generalization error starts to increase again.

Consequently, the optimal generalization is achieved with a complexity of $k = 2$.

SRM is based on the same stepwise approach of first setting the complexity and then minimizing the risk as shown in the above example. In general, it defines a set S of functions with an admissible structure[1] such that

$$\begin{aligned} &\mathrm{S}_k = \{L(y, f(x,w)), w \in \Lambda_k\} \quad \text{and} \\ &\mathrm{S}_1 \subset \mathrm{S}_2 \subset \cdots \subset \mathrm{S}_n \cdots \end{aligned} \tag{4.10}$$

and $\mathrm{S}^* = \cup_k \mathrm{S}_k$. Then, for a given set of D observations the complexity is set by choosing an element S_n of S with $n = n(D)$. For this set S_n, the ERM is applied. Thus, the function in S_n for which $R(w)$ is minimized by decreasing $R_{\mathrm{emp}}^{(D)}(w)$ is selected.

Often, SRM is interpreted as an approach to balance the trade-off between over- and underfitting.

[1] An admissible structure of set S of functions features: S^* everywhere dense in S, VC-dimension h_k of each set S_k is finite, any element of S_k is totally bounded $0 \leq L(y, f(x,w)) \leq B_k, w \in \Lambda$.

5

Regularization Theory for Deep Learning

In 1996, Wolpert et al. unveiled in their *No Free Lunch Theorem* (NLT) for machine learning algorithms that the error rate that is yield by applying every classification algorithm to previously unobserved data points $x \sim P(x)$ is averaged over all possible data-generating processes $P(x)$ always the same. Universally, no learning algorithm is better than any other [56].

Although this seems to be a discouraging result, it only holds if taking all possible $P(x)$ into account. However, not all of them are equally relevant for the given learning task. Only taking the meaningful ones into account allows the design of learning algorithms that perform well on these distributions. To restrict the focus to only a few distributions particular preference are built into the learning algorithm.

One example of such a preference is the adaption of the model complexity by adjusting k, as shown in the figure 4.4. Modifying the capacity affects the set S_k of functions from which the learning algorithm can choose the optimal one. This has a direct impact on the generalization ability. However, the performance of learning machines is not only affected by the size of S_k but also by the kind of functions that it comprises. While polygonal chains are useful to categorize observations that are separated by linear boundaries, splines perform better in classifying data points that are non-linearly separated [19].

Instead of excluding or including particular functions from the set of possible models, a more general approach is to modify the learning algorithm such that it prefers one over the other. Such preferences can be expressed in numerous ways and are summarized in the term of *regularization*. To be precise, any modification to a learning machine to reduce its generalization error but not the training error is called regularization. The latter might even increase, and finally regularization aims at closing the gap between the two errors [19]. While ERM combats underfitting, regularization counteracts overfitting.

The field of regularization techniques is huge and which of them improves the performance the most, finally depends on the learning task. Some of them restrict the learning algorithm while others modify the cost function and hence put a soft constraint on the values of the parameter. Still others, improve generalization by encoding prior knowledge. Further, some regularization methods are designed to prefer simpler models over complex ones as advocated by *Occam's razor*. In terms of neural networks, "simple" can be interpreted in different ways. One straightforward meaning is simple in the sense of containing fewer parameter. As depicted in figure 5.1, I applied this idea to the MNIST classification model in section 3. Reducing the number of parameters from 2048 units per hidden layer to only 512, indeed boosts the generalization performance. The price to pay is an increase in the training error [53].

Presenting all existing regularization techniques is beyond the scope of the present report. In lieu thereof a small selection of different methods, mainly applied for neural networks is examined. The choices are either motivated by their commonness of occurrence or their role

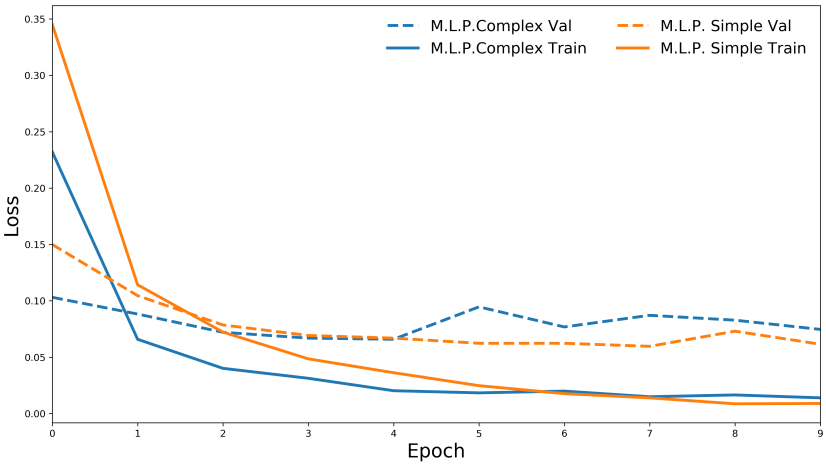

Figure 5.1: The graphs show the training and validation loss of two versions of the MNIST classifier. A *complex* model, corresponding to the blue graphs is similar to the original but has 2 hidden layers with 4096 units. Summing all trainable parameters leads to a total of 20 037 642. Opposed to that, the 2 layers of the *simple* model have only 512 units. This reduces the number parameters that need to be trained to 669 706. Although, the simple model performs worse during training, it outperforms the complex model during validation. Moreover, the gap between training and test error reduces with decreasing model size.

in the field of image processing. I investigate their regularization effect and power by modifying the MNIST classifier from section 3.6. All corresponding *Tensorflow* implementations can be found in the appendix B.

5.1. Parameter Norm Penalties

A particular solution can by achieved, by excluding any other. Whether to discard a certain parameter constellation or not, depends on the loss that it yields. Hence, alerting the loss function to the disadvantage of any solution that is is to be excluded, steers the system in the direction of more favorable ones.

Parameter Norm Penalties are one instance of this regularization approach. As implied by its name, solutions are punished in dependency of a parameter norm. This is formulated in terms of an adapted, *regularized* loss function, which takes the form

$$L_{\text{total}}(y, f(x,w)) = L(y, f(x,w)) + \lambda L_{\text{Reg}}(w), \tag{5.1}$$

where $L(y, f(x,w))$ is the general or *data loss* since it is induced by observations. Opposed to that, the additional term $L_{\text{Reg}}(w)$ is the *regularization loss*, which solely depends on the weights. Its contribution to $L_{\text{total}}(y, f(x,w))$ is controlled by the *regularization parameter* $\lambda \in [0, \infty)$.

Different choices of $L_{\text{Reg}}(w)$ have a different influence on the weights and, hence, result in different solutions. Two common techniques to define the regularization loss are the L^2- and L^1-*Regularization*.

5.1.1. L^2-Regularization

The L^2-parameter penalty norm, often referred to as *weight decay* defines the regularization loss as

$$L_{\text{Reg}}(w) = \frac{1}{2} w^T \cdot w \quad . \tag{5.2}$$

Models that contain many parameters or a few of high value are punished by increasing their loss.

Its mechanism becomes more evident by analysing its behaviour during the training phase. For instance, a SGD optimization that minimizing the newly defined regularized loss updates parameters in iteration τ according to

$$w^{\tau+1} = w^{\tau} - \epsilon_{\tau} \nabla_w L_{\text{total}}(y, f(x,w)) \tag{5.3}$$

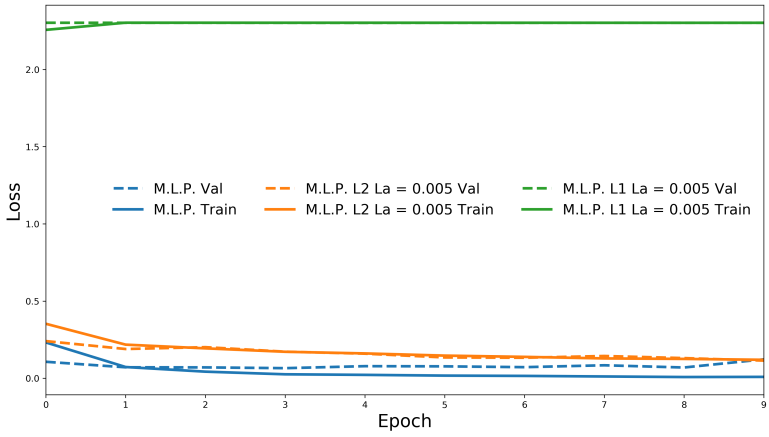

Figure 5.2: The original MNIST classifier is marked by the blue graphs and its regularized versions by orange and green. While the first applies the L^2-regularization with $\lambda = 0.005$, the second uses the same regularization parameter for a L^1-regularization.

with learning rate ϵ_τ. As the regularized loss, its gradient

$$-\frac{\partial L_{\text{total}}}{\partial w_{ij}} = -\frac{\partial L}{\partial w_{ij}} - \lambda w_{ij} \tag{5.4}$$

as well contains two parts. The first displays the dependency of parameter w_{ij} on the data, represented by the data loss $L(y, f(x, w))$. Small values indicate that w_{ij} are weakly supported by the data. Hence, they are less reliably estimable and one runs the risk of distorting the solution when taking them into account. This is were regularization becomes evident, embodied by the second part of the gradient 5.4. While the data loss is too small to do so, this *weight decay* reduces parameter w_{ij}. For the right choice of λ, the parameter w_{ij} gradually decays to zero [42].

While examined only for a particular parameter w_{ij}, this advance is applied independently to all of them. However, each parameter is estimated more or less reliable from the data and hence is more or less strongly affected by the weight decay.

5.1.2. L^1-Regularization

The L^1-Regularization is closely related to the previous approach, yet it defines the regularization loss as

$$L_{\text{Reg}}(w) = \lambda \|w\|_1 \quad . \tag{5.5}$$

As stated earlier for the L^2-regularization, this loss function is plugged into any gradient descent scheme of desire, now applying

$$-\frac{\delta L_{\text{total}}}{\delta w_{ij}} = -\frac{\delta L}{\delta w_{ij}} - \lambda \ \text{sign}(w_{ij}) \tag{5.6}$$

to update the parameters. Again, regularization comes into effect when updating parameters that rely only weakly on the data. Depending on the adjustment of λ the decay of w_{ij} is more or less fast. For sufficiently large values, $\lambda \text{sign}(w_{ij})$ can overrule the data induced gradient and set the parameter directly to 0.

Both, the L^2- as well as the L^1-regularization steer the values of parameters that can not be derived from data towards zero. The latter, however, is more aggressive in doing so and sets more parameters to 0.

The impact of such solutions that feature a high level of sparsity becomes evident when considering for instance feature learning. The feature which corresponds to the parameters that are set zero is safely discarded [19].

The schematic illustration in figure 5.4 shows the adapted architecture of my base model. The regularization strength of the L^1-approach, however, becomes already apparent when I incorporate it into the MNIST classifier. In order to compare it with the L^2-regularization,

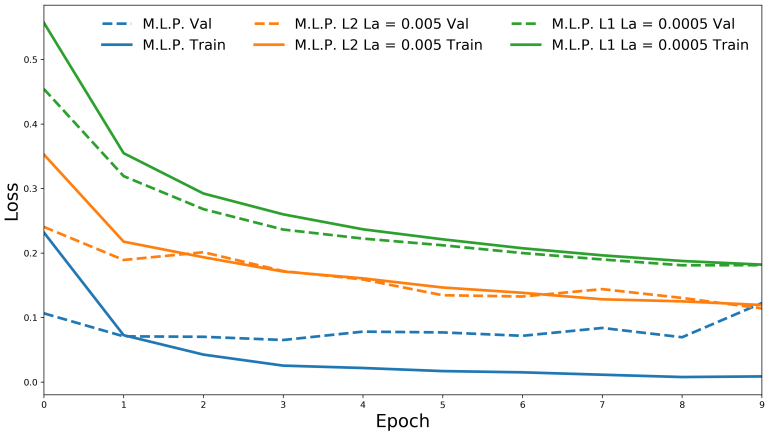

Figure 5.3: Choosing the right regularization parameter λ is crucial for the effect of parameter norm penalties. While $\lambda = 0.005$ seems to be a reasonable value for the L^2-regularization, it is $\lambda_{L^1} = 0.0005$.

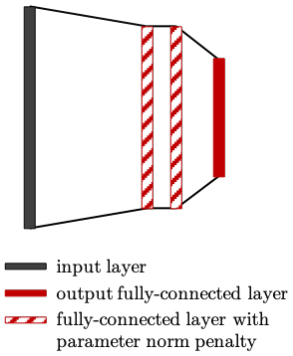

Figure 5.4: The scheme shows the structure of my MNIST classifier after incorporating parameter norm penalties. Obviously, the architecture does not change. As denote by the color (shade of red) the hidden layers are still fully connected. The stripped pattern marks the additional regularization.

both methods are include into the base model. To ensures that the regularization terms are included into the loss at the same extent, I set the regularization parameter in both cases to $\lambda = 0.005$. As shown in figure 5.2, the L^2-method shifts the errors closer to each other while increasing both. Opposed to that, regularizing with L^1 sets the gap to zero and pushes the errors to a high level. Although the setting prevented overfitting for both approaches, over-regularizing can be damaging as well. Figure 5.3 reveals that finding the right regularization parameter is a dedicated issue. Apparently, reducing λ by one order of magnitude leads to a less aggressive L^1-regularization.

5.2. Invariance Learning

Parameter norm penalties regularize by exploiting the relation between parameters and data. This approach can be applied without knowing the final purpose of the learning machine. Opposed to this *invariance learning* aims at improving the generalization performance by incorporating prior knowledge about the learning task. A system is called invariant towards a specific feature, if changes in the input with respect to this feature do not alert the output.

However, the nature of this invariance strongly depends on the application purpose of the learning system. Assuming for instance a neural network that is designed to classify the MNIST data set. For a human, it is obvious that a five in the bottom of an image still remains a five if it is shifted to the top. On the pixel level of the input data though, this translation causes a significant change. Nevertheless, a neural network that generalizes well has to be capable of correctly recognize the five. It has to be invariant with respect to translation.

In principle, an adaptive model such as the neural network can learn this invariance if it is trained with a sufficiently large number of input images which show fours in either of the lower corners. This approach is hampered if not infeasible if too few training examples are given or several invariances have to be trained.

The following presented techniques offer alternative approaches how to learn invariances

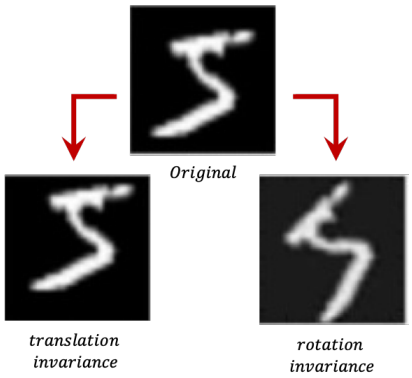

Figure 5.5: To train the MNIST classifier with respect to invariance towards translation or rotation can be yield by modifying the input in accordance with the invariance.

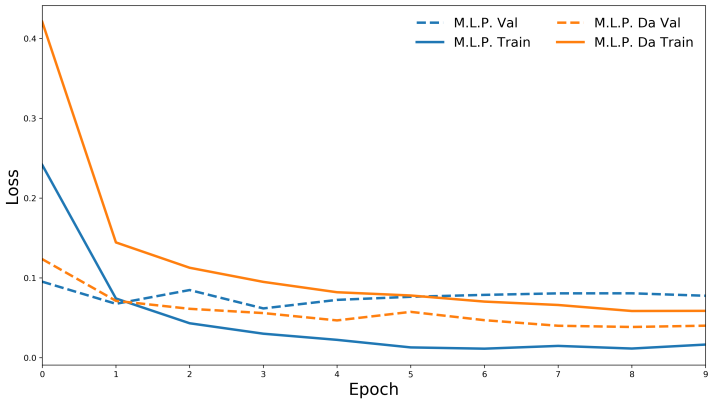

Figure 5.6: Feeding the basic MNIST classifier with augmented input data leads to less overfitting. Blue denotes the basic model while the orange graphs depict the bahivour of the same model, but trained with augmented input data. The following input modifications were used: random rotation of $+/-8°$, random horizontal and vertical shifts of $+/-8\%$ of pixel width respectively height, random shearing with a maximal angle of 0.3% in counter-clockwise direction as well as random zoom in a range of 8%.

and hence improve the generalization ability [11].

5.2.1. Data augmentation

Small data sets restrict the most straightforward way to improve the generalization ability of a model, namely training with more data. One approach to get around this issue is to artificially extent the given data set by exploiting it to create faked inputs. However, not every learning task allows the augmentation of data.

For a density estimation task, fake data can only be produced with the final solution. Opposed to that, classification is one of the learning tasks for which it is easiest. Here, a high-dimensional and complicated input x is mapped to a single category y. It is inherent in this learning task, that the classifier features invariances towards a wide range of transformation. With this knowledge about classification, new training data can be generated by augmenting the input data x [19].

How to finally augment the data depends again on the learning tasks and the kind of invariances that the system needs to learn. To examine that, I trained the base model for the MNIST classification with respect to translation by spatially shifting the original image. figure 5.5 shows, that turning it leads to invariance towards rotation. Figure 5.6 depicts that my classifier as earlier tends to less overfitting when training it with slightly rotated, spatially shifted, zoomed or sheared input images.

A further possibility to benefit from data augmentation arises with the finding of Tang and Eliasmith that neural networks are not very robust towards noise [51]. This characteristic can be diminished by training them with data that is augmented by applied random noise.

5.2.2. Parameter sharing

While data augmentation trains the learning machine to produce the same output for two slightly different inputs, invariances can also manifest themselves within the same input.

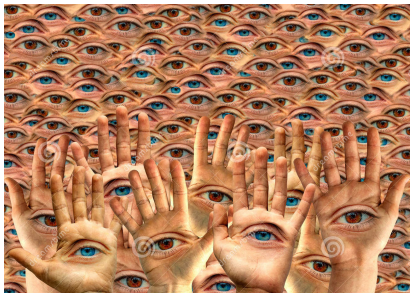

Figure 5.7: Invariances can also be necessary to find the same object within one input data. Image: [1]

Assuming for instance that a classifier tries to recognize all blue eyes in figure 5.7. Obviously, the object of desire is spread all over the image and might even differ in its angle. Hence, the system has to behave invariant towards translation and small rotations within the input. In spite of the different locations, the object of the blue eye is still the same and all of its instances share the same pixelwise description. From a neural network point of view the same feature can be detected by the same parameters. Hence, instead of using independent parameters for each eye, the different components of the model share a unique set of parameters. This so-called *parameter sharing* leads not only to invariance towards translation and distortion but it also reduces the number of parameters.

A special instance of neural networks, namely *Convolutional Neural Networks* makes extensive use of this regularization technique. They mainly profit from the significantly lower amount of parameters. This allows the network not only to save memory but also to dramatically grow in size without requiring a corresponding increase in training data [19].

In general, parameter sharing is not only applicable to components of the same model but also to several models that are instructed with the same learning task. An even more general approach is *parameter tying* or *soft parameter sharing*. This approach encourages groups of parameters or parameters of multiple models for the same learning task to have similar values. Assuming for instance, one model uses parameter $w^{(1)}$ while a second model uses $w^{(2)}$. The dependency between them can be expressed by adding a regularizing parameter norm penalty of the form $L_{\text{Reg}} = \|w^{(1)} - w^{(2)}\|_2^2$ to the loss function. Opposed to parameter sharing, where the $w^{(1)} = w^{(2)}$ parameter tying misses the remarkable advantage of a reduced number of parameters [11, 19].

5.3. Dropout

The approach of *dropout* has its origin far away from artificial intelligence, in genetics. Naturally, a set of genes is an offspring of combining each time half of the parental sets and adding a small pinch of random mutation. In a laboratory, this process is emulated by using slightly alerted copies of the original genes. Intuitively, one might assume that individual fitness is optimized by combining such set of genes, which have proven to work well together. However, in nature such connections are broken up but still the highest developed organisms evolve.

Srivastava et al.[48] try to explain that by fathoming the paths of natural selection. According to them, nature does not generate gene sets to optimize individual fitness but maximize their ability to mix with others. Indeed, genes that are able to work well together with another random set of genes make them more robust. Such genes can not rely on the presence of a large sets of partner. Rather, they have to adapt to situations in which they are on their own or with only a small set of others.

Interpreting units of a neural network as genes, leads to the idea to increase their robustness and ability to work independently from other neurons by combining them with a randomly chosen sample of units.

Dropout achieves this random combination of units by temporally removing a number of units from the network. Figure 5.8 illustrates, that this removal includes all incoming and outcoming connections. Discarded units neither participate in the forward nor in the backward propagation when training the network with gradient descent methods and BP. As with

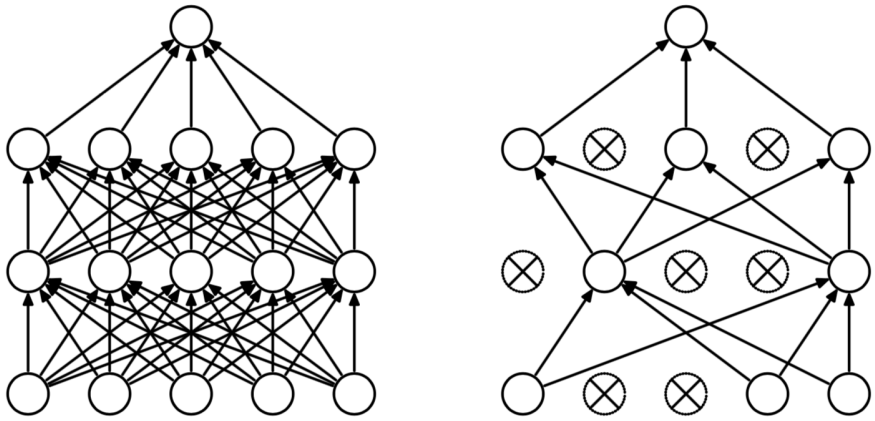

Figure 5.8: The scheme visualizes the drop out model. *(LEFT)* An original two-layered standard neural network.*(RIGHT)* After *Dropout* is applied, the crossed units as well as their in- and outcoming connections, are temporally removed from the network. Comparing it to the original architecture shows how *Dropout* alerts the architecture [48].

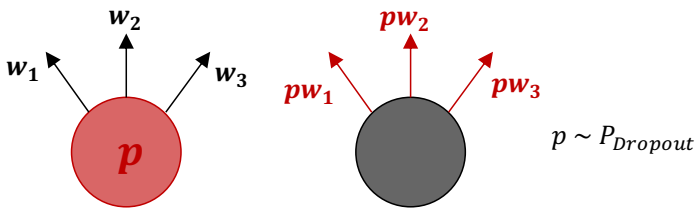

Figure 5.9: Models that are trained with dropout have to be treated differently during training and test phase. *(LEFT)* While training the model each of the units is only present with probability p. *(RIGHT)* During testing, this is represented by keeping all units but down-scaling their outputs with a factor of p.

genes, the dropped units are set randomly. For instance, each unit is retained with a fixed probability of p.

Dropout can be interpreted as a version of *ensemble learning*. In general, such methods combine several base models to derive a final, optimal prediction model. Given a neural network of fixed size, base models are all possible different parameter settings which are combined by averaging their weighted predictions. This works well, since different models suffer from different shortcomings. It works even better for combined models that strongly differ from each other i.e. feature varying architectures. For a sufficiently large model, however training all of its instances or several different architectures is fatally expensive [19].

Dropout addresses this issue by providing a way to approximately combine exponentially many different neural networks architectures. In figure 5.8, the dropped out network appears to be a "thinned out" samples of its version. Assuming n parameters, then dropping out units leads to 2^n different instances of the network. Each time an input is presented to the network, a different architecture is sampled by randomly dropping out units. Yet, all of them share the same parameters. All in all, training with dropout is equivalent to combining 2^n different models which all share the same weights and are rarely trained [27, 48].

Finally, the neural network is trained with exponentially many smaller samples of itself by retaining each unit with a probability of p. During the test phase, however, this is not reproducible. As shown in figure 5.9, it is instead approximated by one single neural network without dropout. It is a scaled-down version of the original model whose units outputs are scaled with a factor p. This allows to merge the 2^n training networks, which share weights into a single one during test phase.

Again to examine the effect of *dropout*, I build it into the MNIST classifier. Opposed to parameter norm penalties, I have to alert the network architecture to include dropout regularization. This is visualized in figure 5.11. The training and generalization performances of the base model as introduced in section 3.6 and its adapted version with dropout are depicted in figure 5.10. Setting the keep probability to $p = 0.5$ regularizes well in the sense that the gap between the errors is narrowed while even improving the generalization performance. For this achievement, an increasing training error is accepted.

As mentioned before in context of the parameter norm penalties, the choice of the hyperparameter p is essential for the regularization performance. To get an idea of its impact,

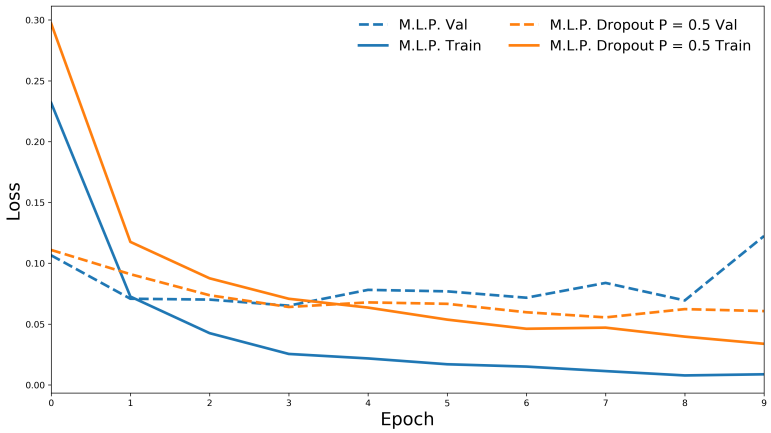

Figure 5.10: Modifying the basic MNIST classifier by incorporating *Dropout* regularizes well. However, overfitting is not completely prevented but reduced.

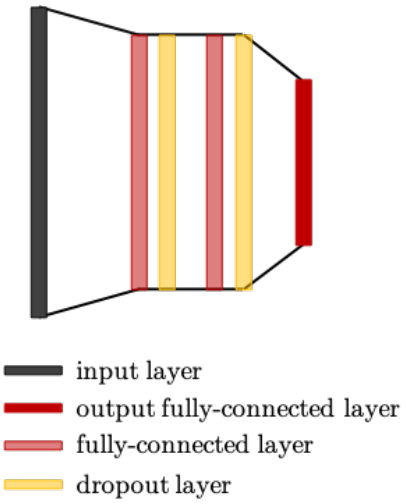

Figure 5.11: The scheme illustrates again the architecture of my MNIST classifier. To include dropout, I add a further stage after each fully-connected layer. This dropout layer is colored in yellow.

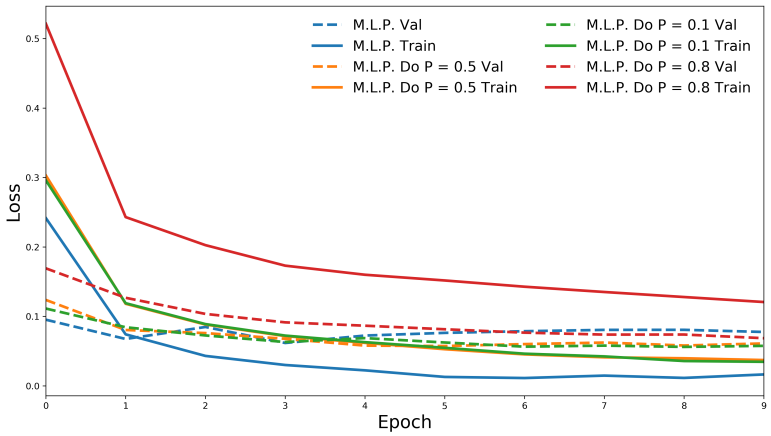

Figure 5.12: Modifying the basic MNIST classifier by incorporating *Dropout* regularizes well. However, overfitting is not completely prevented but reduced.

I train and evaluate the MNIST classification model with three different keep probabilities $p \in \{0.1, 0.5, 0.8\}$. The error courses are depicted in figure 5.12. Although three values are by far not enough to make final predictions, the graphs indicate a non-linear behaviour of the loss function on p. While increasing p from 0.1 to 0.5 nearly does not cause any change in performance, further increasing p finally leads to radical modifications in training and validation performance. While the validation error stays on the same level as with the examined lower values of p, the training error strongly increases. Hence, the gap between the empirical and actual risk is extremely high and it is questionable whether that is necessary.

6

Basics of Convolutional Networks

With their 1943 released mathematical description of nervous activities, neural nets and their relation among each other, Warren McCulloch and Walter Pitts laid the cornerstone of neural networks [35]. One outstanding characteristic of them is their ability to learn i.e. generalize from examples. While aiming to optimize this power, researchers draw the conclusion that generalization performance is improved by reducing the number of free parameters in the model. At the same time the network has to maintain a certain level of complexity to deal with the learning task.

LeCun et al. found that a success-promising remedy for this trade-off is the incorporation of prior knowledge. Obviously, it is utopistic to uniformly specify such knowledge. However, considering a particular learning task makes it feasible.

A prime example for this idea is visual pattern recognition. Three basic perceptions from classical works in this field are adopted. First, higher-level features can be split up into basic, more abstract ones. Hence, extracting local features and using them to rebuilt the final pattern is advantageous. Second, distinctive features of an object can appear in various locations of the input pattern. And last but not least, to eventually recognize the object, the relative position of such features towards each other matters whereas information on the precise location is needless [31].

In terms of neural networks, these principles formulate as follows. The detection of abstract features correlates to hidden layers that only combine local information. Hence, a full connection of units in a network is obsolete and they can be substitute by sparse links instead. Since similar parameters extract similar features, parameter sharing allows to scan the entire input pattern for the sane feature. Finally, reducing the outputs resolution, from now on called *downsampling* leads to approximated feature locations instead of retain exact positions [31].

Fully-connected, regular neural networks are based on common matrix multiplication. Opposed to that, the above considered requirements for visual pattern recognition are met by networks which are built on alternating layers of discrete convolution and downsampling. In terms of LeCun's idea, this is equivalent to exploit prior knowledge by tailoring the network's architecture.

This chapter is devoted to the discrete convolution whereas downsampling is thematised in chapter 7. Starting with a discussion on the fundamental operation of convolution, the remainder of the chapter is concerned with the detailed explanations of the impact of discrete convolution if applied in neural networks.

6.1. The Discrete Convolution Operation

The operation of *convolution* has its origin in functional analysis. For the two functions $f, g : \mathbb{R}^N \times \mathbb{R}^M \mapsto \mathbb{C}$, it is defined as

$$(f * g)(x, y) = \int_{\mathbb{R}^N} \int_{\mathbb{R}^M} f(s,t) g(x - s, y - t) ds dt \tag{6.1}$$

with

$$(f * g)(x, y) = (g * f)(x, y). \tag{6.2}$$

To understand the effect of convolution, consider the following situation. The function f describes a series of data e.g. supplied by a sensor that scans the surface of a particular material. However, the sensor is noisy and one tries to diminish its impact by averaging several data values.

For a measurement in location x, the values of the closest surrounding are the most relevant. Hence, a weighted average with emphasis on spatially neighboring values leads to a more precise estimate of f[19, 41].

In the convolution operation as defined in 6.1, the function $g(s,t)$ takes the part of the weights of certain values. For the smoothed estimate of f in location (x, y), values that are shifted by s respectively t to the left or right respectively up or down are enhanced.

Equation 6.1 describes a continuous convolution. However, the principle can easily be extended to discrete situation as appearing when the sensor measures the surface in equidistant steps of size x and y. Analogously to 6.1, the discrete convolution formulates as

$$(f * g)(x, y) = \sum_{s=-\infty}^{\infty} \sum_{t=-\infty}^{\infty} f(s,t) g(x - s, y - t) \tag{6.3}$$

where (s,t) take only integer values. For functions f, g that are non-zero for a finite subset of s and t, the summation reduces to the sum over the values of non-zero overlaps. Hence,

$$(f * g)(x, y) = \sum_{m} \sum_{n} f(s,t) g(x - s, y - t). \tag{6.4}$$

If the functions f, g are interpreted as matrices, the 2D discrete convolution can also be regarded as a matrix multiplication. For that, represent $f(x, y)$ by the matrix $I \in \mathbb{R}^{n \times m}$ and g by $Q \in \mathbb{R}^{k \times k}$. The convolution $(f * g)(x, y)$ is then a matrix F whose dimension depends on the final implementation of the convolution [19]. Taking the commutative property 6.2 of the convolution into account, the matrix-multiplication reads as follows:

$$F_{xy} = \sum_{m'=0}^{k} \sum_{n'=0}^{k} I_{x-n', y-m'} \cdot Q_{n', m'} \tag{6.5}$$

Figure 6.1 illustrates how a 2D discerte convolution is computed.

In real life, there is a wide range of different interpretations of the matrices in 6.5. Assuming for example, that the matrix I represents a black-white image of resolution $n \times m$. The matrix Q takes the role of a so-called *filter* or *kernel*. Such kernels, applied to images according to equation 6.5 extract one specific property of the image as for instance a vertical edge. In each pixel of I, a numerical value is associated with this feature. These are stored in the convolution matrix F, which often is referred to as *feature map* of kernel Q.

If the regular matrix multiplication in a fully-connected neural network is substituted by a convolution operation as defined in equation 6.5 the network turns into a so-called *Convolution Neural Network* (CNN) or *deep Convolution Neural Network* (ConvNets) [19]. In the further course of this report, the terminology concerning I, Q and F as used in the above example is retained.

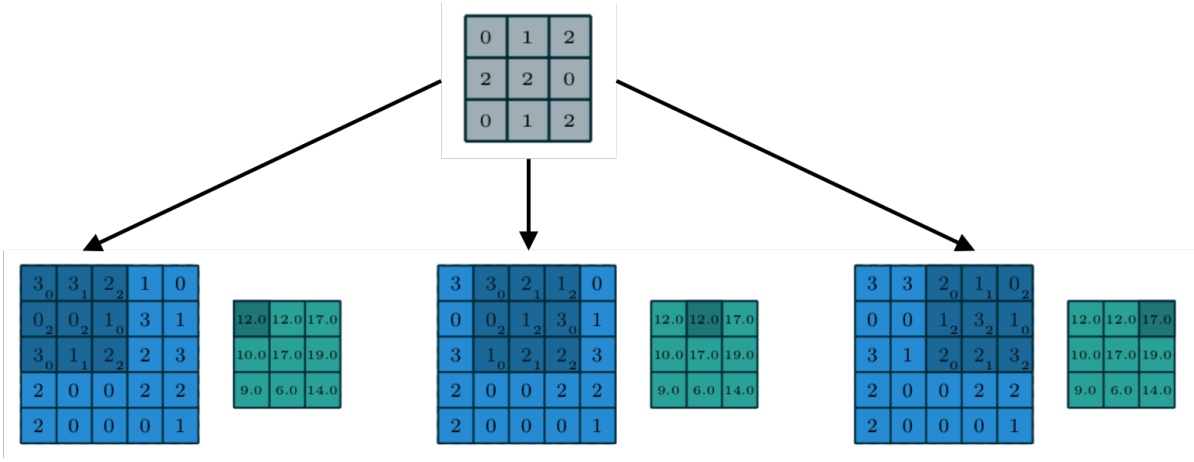

Figure 6.1: The image illustrates one example for a 2D discrete convolution with a filter of values as shown in the gray matrix in top. Applying this to the blue 5×5 input matrix generates an output of size 3×3. The image exemplifies the calculation on the first row of the input matrix. Discrete convolution is illustrated as overlapping the input matrix with the kernel. The resulting outputs are yield by first multiplying the overlapping input and kernel entries and then summing over all these products [**?**].

6.2. Characteristics of Discrete Convolution in Neural Networks

The motivation to substitute an ordinary matrix multiplication by a convolution operation and thus change the network's architecture, is rooted in the need for an increase in generalization performance. In fact, convolution comes with a few distinctive and advantageous features. It corresponds to a *sparsely* connected, *shared* parameter matrix that is associated with an *equivariant* network. A further distinctive feature of the discrete convolution is its ability to proceed *tensors*. Before examining the properties and their effect, the term of tensors is briefly clarified.

6.2.1. Tensor

The heart piece of common neural networks is affine transformations. In each layer, an input vector is received and multiplied with a matrix to produce an output vector. This procedure is independent of the input's dimensionality. Sound clips, images or even a not ordered collection of data, can be flattened into an one-dimensional vector before being fed into the transformation. However, a wide range of input data have an intrinsic structure which features distinctive properties [16].

Such data is represented by so-called *tensors*. For the given purpose, it is sufficient to think of tensors as a multi-dimensional array. They are characterized by the number of dimensions or axes, the *rank* of the tensor. A tensor of rank 0 is simply a scalar. Rank 1 tensors denote a column vector of size $1 \times n$ and analogously, rank 2 a $m \times n$-matrix.
Although, tensors seem to be abstract mathematical structures, they can be encountered in many daily situations. As shown in figure 6.2 images for example are simply tensors of either rank 2 or rank 3.

A further important feature of structured data is the fact that the ordering of the dimension matters. An image has particular width and height axes, while sounds clips possess one time axis. In addition to that, one of the axes, called *channel axis* is used to access different views of the data. For images it is three-dimensional and represents the three colors red, blue and green whereas the channel axis of a stereo audio track has only two dimensions. Each of them representing the left and right channel.

If applying an affine transformation as in regular neural networks, these properties of the structured data are lost while flattening. All the axes are treated the same, assuming that none of them carry additional knowledge. However, when dealing with the complex task of object recognition one should be thankful for every bit of data that can be retrieved. Hence,

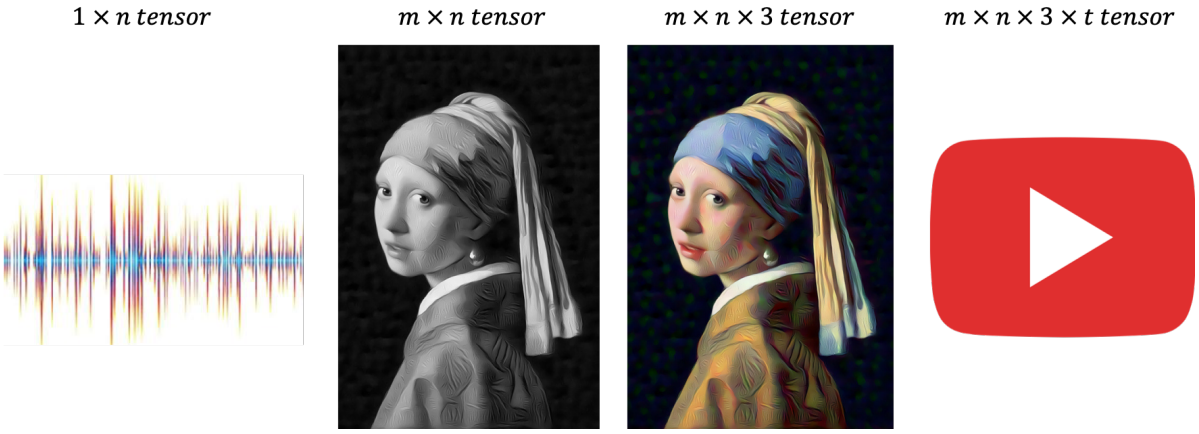

Figure 6.2: Daily life examples such as audio files, images or video clips are tensors of different dimensions.

preserving the intrinsic structure would be of striking advantage [16].

That is where discrete convolution comes into game. Opposed to common matrix-vector multiplication, it is able to retain the topology and its containing information. Incorporating prior knowledge about the data representation of images, layers in convolutional networks are designed such that they expect a structured input.
In CNNs tensors do not only appear when dealing with input data. Furthermore, the tensor representation is also applied to transfer data between the internal layers of the network. Here again, knowledge is stored in the data structure. Working with tensors can be understood as an increase in the information density of the data that travels within the network.

However, to efficiently deal with large input, CNNs take advantage of further properties of discrete convolution [19] .

6.2.2. Sparse connectivity

Imagine one wants to detect a feature in an image that occupies 100 pixels. Even if the input image has thousands or millions of pixels, it still suffices to scan a small, 100 pixels containing area to find the feature. Such a local restriction can be accomplished by constraining the kernel to the spatial extent of the feature. The so-called *receptive field* of Q describes this spatial size. It determines the ”window of the image” that is visible for the kernel. Choosing kernels Q of remarkable smaller size than the input I results in networks with *sparsely connected* layers [19].

Thus, there exist two kinds of connectivity in neural networks. Regular neural networks are built up by fully-connected layers where every input interacts with every output. Each of these connections is uniquely described by one parameter. Opposed to that, convolutional neural networks make use of sparse interactions where an input is only connected to a few outputs. Figure 6.3 illustrates the effect of an input unit on the output for both kinds of connectivity. Figure 6.4 describes the opposed point of view.

Sparse connectivity features a distinctive advantage: it leads to a remarkably reduced number of parameters. This in turn, does not only alleviate overfitting and save memory. In addition to that, the computational costs decrease in the same extent in which the connectivity is reduced. Hence, sparsity yields an improvement in efficiency without affecting the performance [19].

However, it seems as if locally connected networks suffer from a serious drawback. The number of inputs that directly interact with each other is limited by the number of inputs that is covered by the kernel. Thus, more complex connections i.e. interactions that involves more inputs are represented indirectly and the inputs are connected in deeper layer of the network. Hence, this issue is tackled by describing complicated connections by constructing

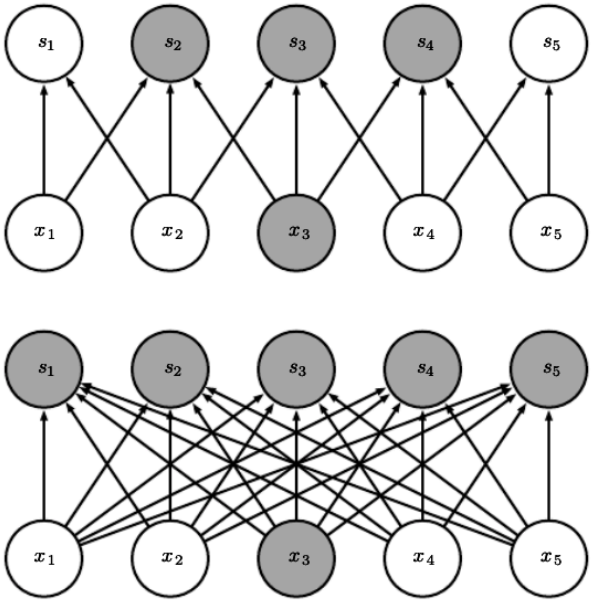

Figure 6.3: Two versions of connectivity, sparse and fully. Input units are denoted with x, output units with s. Grey colored units are connected. *(TOP)* The output s is generated by convolution with a kernel of size 3×3 and input x_3 only affects 3 outputs. *(BOTTOM)* The output is formed by a matrix multiplication and no longer spares. x_3 affects each of the output units [19]

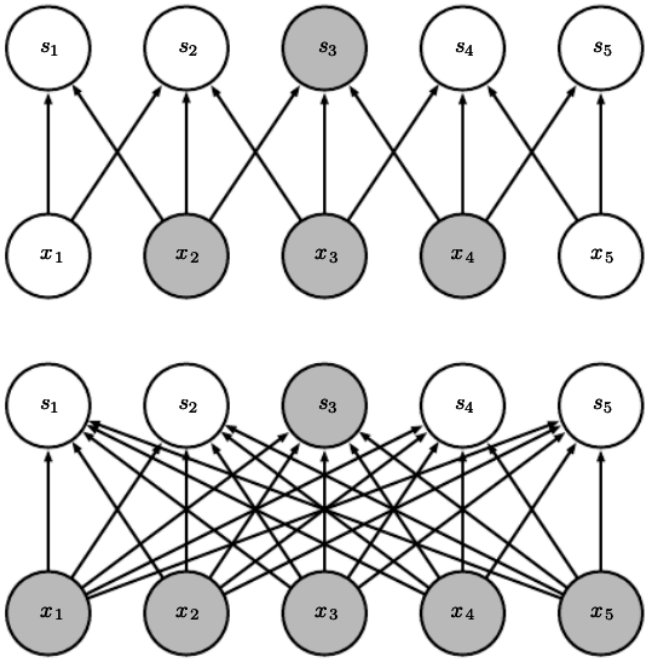

Figure 6.4: Two versions of connectivity, sparse and fully. Input units are denoted with x, output units with s. Grey colored units are connected. *(TOP)* The output s is generated by convolution with a kernel with receptive field of size 3×3. Output s is affected by 3 inputs. *(BOTTOM)* The output is formed by a matrix multiplication and no longer spares. s is affected by each of the input units [19]

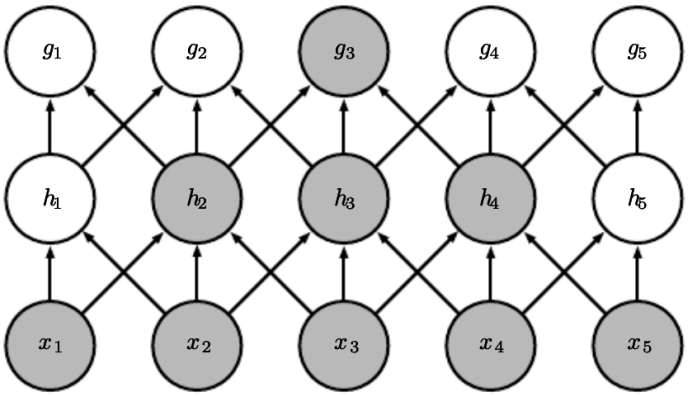

Figure 6.5: In the above illustration, x_i denotes the inputs and h_j and k_l following hidden units. Assuming that a kernel of size 3×3 operates on each of this layers. Hence, the first hidden layer is not able to combine the information derived from input x_1 and x_5. This can be tackled by adding the second hidden layer g which merges the units h_2, h_3 and h_4 that suffice to represent the whole range of inputs x_i with $i = 1, \ldots, 5$. [19]

them from simpler ones, as presented in figure .

6.2.3. Parameter sharing

Each kernel is associated with a particular learning task, namely to extract a specific feature. Assuming that the same feature can be detected in several areas of the input, the parameters of the correlated kernel Q are used in every location of the input I. Consequently, instead of training a set of parameter for each location, the same parameters are shared in each location [19].

Parameter sharing is a technique to integrate prior knowledge. As mentioned in section **??** it regularizes the network and hence improves its generalization performance.

6.2.4. Equivariance

Despite their outstanding performance in image representation, a comprehensive theoretical understanding of deep convolutional neural networks has not been reached so far. Attempting to extent the limited knowledge about them, the mathematical properties of representation are examined. These are equivalence and equivariance with the special form of invariance [32]. Equivalence is concerned with the question whether two CNNs capture the same visual information or not. Equivariance investigates how transformations of the input impact the output [1]. Broadly speaking, a representation is equivariant with respect to a transformation in the input, if the output changes in the same way. Invariance occurs if transformations of the input have no effect at all.

The success of CNNs is generally attributed to their equivariance to translation [25]. If e.g. the input image is shifted by one pixle to the left its representation in the feature map shifts in the same extent [19]. This property is directly inherited from parameter sharing which, in turn is intrinsic to discrete convolution [25]. However, the convolution operation leads only to representations that are equivariant with respect to translation. To deal with differently transformed inputs such as rotated images, further mechanism are necessary. With data augmentation, one such alternative approach was presented in **??**.

Concluding, neural networks generalize better if prior knowledge about their learning task is exploited. For instance, discrete convolution forces the network to behave in accordance with principles of object recognition. The price to pay is a design overhead since the network architecture needs to be restructured.

[1] Let the function ϕ that maps an image $\mathbf{x} \in X$ to $\phi(\mathbf{x}) \in Z$ describe a CNN. This CNN is equivariant with respect to a transformation $g : X \mapsto X$ if

$$\exists M_g : Z \mapsto Z, \forall \mathbf{x} \in X : \phi(g\mathbf{x}) \approx M_g \phi(\mathbf{x}) \quad .$$

A sufficient condition for the existence of M_g is that ϕ is invertible, since in this case $M_g = \phi \circ g \circ \phi^{-1}$. The nature of g is in principle arbitrary [32].

7

Image Classification with Convolutional Networks

The task of image classification consists of two sub problems: finding objects in the image and classifying them. This division can be recovered in the architecture of classification networks. One substructure detects features, the second categorizes them. While the latter can easily be implemented as a fully-connected multi-layer network, the feature extraction is less straightforward [29]. Since the late 1980s, it is known that regular neural networks with a few hidden layers do not suffice for real-life object recognition tasks [31].

This is justified by the fact that fully-connected layers do not scale well with images. To clarify, consider a colored image of resolution $m \times n$. Since tensors can not be proceeded with fully-connected layers, a transformation of the 3D original data into a 1D vector of size $m \cdot n \cdot 3$ is required. Consequently, not only the topology of the input is ignored but also the model capacity increased. A first hidden layer of size k has $n \cdot m \cdot 3 \cdot k$ parameters. At the same time, sufficiently many and large hidden layers are necessary to cope with the complex task of image recognition. However, this further increases the number of parameters and the model becomes more and more prone to overfit. Finally, not even extremely large data sets can deliver enough information for the network to generalize well [27]. The redeeming idea, as examined in chapter 6 is the incorporation of prior knowledge by redesigning the network as implemented in convolutional neural networks.

A further deficit of fully-connected layers manifests when taking early studies on cat's visual cortex[1] into account. Accoring to Hubel and Wiesel [23] *simple* and *complex* cells for visual processing can be distinguished. Both primarily respond to edges and gratings. In addition to that, complex cells pool from the simple and, hence, feature a certain degree of spatial invariance.

Equivariance is a generalization of invariance. Pooling from simple cells is emulated by downsampling stages. This exploits the equivariance property of discrete convolution and turns it into invariance towards translation.

The following chapter aims at giving an overview on convolutional neural networks, which are designed to classify images. In their outlines, all of them share the same substructures which are presented in the first two sections. Applying convolutional neural networks to real life problems comes with further implementation issues. The type of problems and ways to cope with them is exemplified in the remainder of the chapter.

[1] Visual cortex is the part of the brain that is responsible to process visual information.

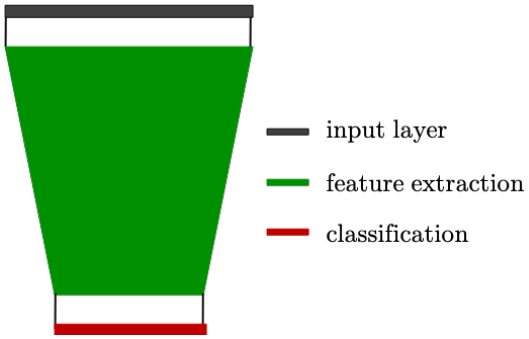

Figure 7.1: The two-folded architecture of a conventional convolutional network is emphasis by coloring: green denotes the feature extraction. In this first part, the properties of the input which are relevant for classification are obtained. They are forwarded to the subsequent classification part.

7.1. Conventional Convolutional Networks

Conventional Convolutional Networks for images classification share a common structure. Figure 7.1 shows that first features are extracted which are then used for classification.

7.1.1. Feature Extraction

Since 1979, when Fukushima et al. introduced a neural network for pattern recognition based on convolution and translation invariance[2] for the first time, numerous improvements have been achieved. However, feature detectors still appear as a stack of stages [18]. A general overview of functionality and design parameters of the layers will be given in the following section.

Convolution Layer

As implied by its name, the core part of the convolution stage is the discrete convolution operation itself. Briefly recapping, the convolution consists of an input I and a kernel Q that maps to a feature map F. This corresponds to a sparsely connected and shared parameter matrix. Entries in feature maps will be referred to as units, activations or pixels.

One stage of the network consists of multiple such kernels K that are applied in parallel. Each of them extract a particular feature and hence results in a distinctive feature map. This imprints a three dimensional shape on the output. Often, the number of maps and hence the third dimension is referred to as the *depth* of the layer. Note that this is an unfavorable terminology, since the here described *depth* should not be confused with the *depth* of deep neural networks which gives the number of hidden layers. Sparse connectivity is asymmetric and only valid for spatial extent. Kernels are always full in depth and thus their third dimension equals the number of maps in the prior layer. In the first convolutional layer, this depends on the input, as already pointed out in 6.2.2. Besides, some kernels even have a fourth dimension which treats the batch size [49].

In addition to the number of kernels, their implementation play a crucial role for the appearance of the output. While depth sets the third dimension, the spatial extent of the output is controlled by the *stride* and *zero-padding*.

The discrete convolution as defined in 6.5 is applied to each pixel of an input image. However, taking only each s-th pixel into account, yield a more compact but less fine representation as illustrated in figure 7.2. Note that this reduction in the output size is interpreted as a downsampling of the full convolution. It does not only lower the computational cost but also exploits spatial invariance. The number s of skipped values is the *stride* of the kernel [19].

Zero-padding describes the possibility to pad the input with zeros. This is motivated by the fact, that each entry of the kernel needs to correspond to an element of the input. To illustrated this, the extreme situations are taken into account.

[2] Fukushima et al. developed the so-called *Neocognitron* is a convolutional network whose parameters are fixed. The first CNN that applied backpropagation to learn from examples was *LeNet* [29].

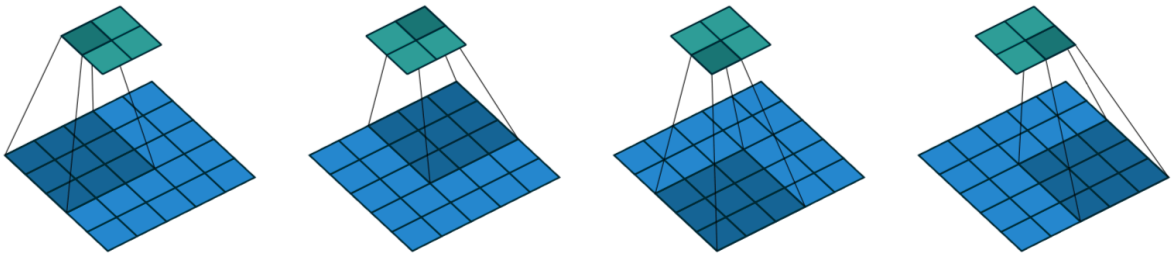

Figure 7.2: In figure 6.1 it was shown that convolving a 3×3 kernel over a 5×5 input matrix generates an output of size 3×3. This is equivalent to use a unit stride $s = 1$. However, increasing the stride to $s = 2$ in both dimensions of the input alerts not only the output's values but also its size. It reduces to 2×2. The larger the stride, the smaller the output. This example also illustrates that the stride can be imagined as the step size with which the filter slides trough the image [16].

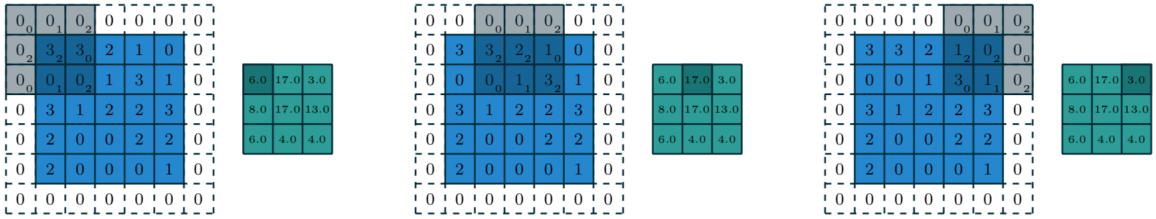

Figure 7.3: Shown is the same situation as in figure 6.1 but this time, the input is padded with one layer of zeros and the filter is convolved with a stride of $s = 2$. Comparing the output entries of the first row shows that zero padding affects the results. Assuming a zero-padding with unit stride $s = 1$ would increase the output map. However, this effect is counteracted by adapting the stride to $s = 2$. That shows how both, stride and zero-padding can be used to control the size of the output [16].

First, consider a convolution without zero-padding. Convolution is applied such that the kernel only visits position where the complete kernel is contained in the image. This approach ensures that the output only contains entries that arise from the same number of input pixels. However, it necessarily leads to a reduced output size. Thus, this rate of shrinkage restricts the number of convolutional layers. As soon as the feature map reaches a spatial extent of 1×1, convolution is not possible anymore. This effect is indicated in figures 6.1 and 7.2. In both cases the convolution is executed without zero-padding and indeed the output size decreases compared to the input tensor.

Opposed to that, the input can be padded such that a convolutional layer does not modify the size of the input. It stays constant through the entire ConvNet and does not constraint the networks architecture. This freedom is excepted at the expense of the representation of the bordering pixels. Due to the padded zeros, these pixels influence fewer output units than those in the center of the input [19].

Summarizing, the output of the convolutional layer is a volume whose third dimension is described by the depth. The spatial size is a function of the several hyperparameters such as input size, stride, zero-padding and the receptive field of the kernel.

The presented convolutional layer is the simplest of its kind. In reality, a widely used approach is the multi-channel convolution opposed to the here assumed single channel convolution. This is based on adding multiple feature maps of different kernels.

In addition to that, in some cases it is necessary to adapt the connectivity of input and output. By sharing weights throughout the complete input, it detects the same feature in the whole image. However, if it is known that a specific feature only appears in certain areas of the input, shared weights are unnecessary. For instance, eyebrows only appear in the upper part of faces. Instead of convolution, locally connected layers can be used. Misleadingly, this is sometimes called *unshared convolution* [19].

In section 6.1 discrete convolution is introduced as a specialized form of linear operations.

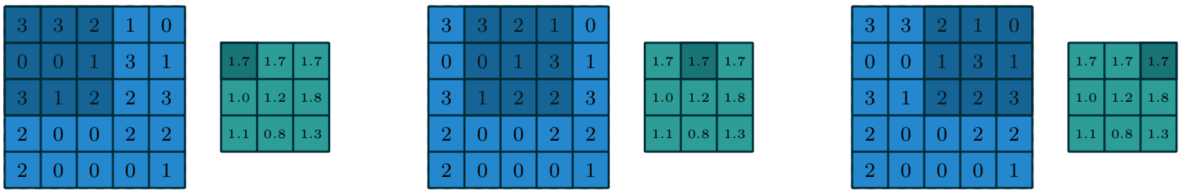

Figure 7.4: The image applies *average pooling* to the same input tensor as introduced in figure 6.1. Assuming a pooling kernel of size 3×3 means that all 9 input elements that are covered by the kernel are averaged [16].

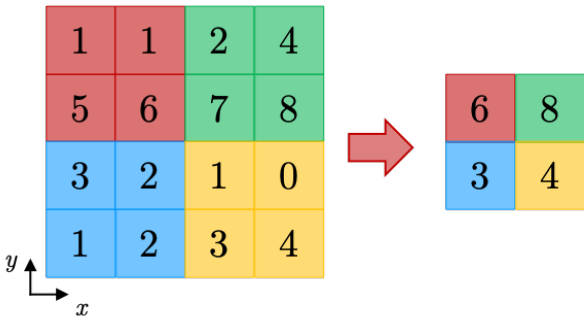

Figure 7.5: MAX pooling with a pooling kernel of size 2×2 can be imagined as splitting the input in several grids. The pooling is then conducted independently on each of these. If the stride is chosen such that it is equivalent to the kernel dimension leads to non-overlapping grids. In the above example, $s = 2$ and each separate grid region is marked with a different color. Since each grid generates only one output, namely the one with the largest value, the output size is remarkable smaller [49].

Applying it to any kind of input yields linear combinations, which are then forwarded to a *nonlinear activation function*. According to Krizhevsky et al. [27], deep neural networks with ReLU activation functions train several times faster than networks with hyperbolic tangent activation functions. Hence, choosing an appropriate activation function is crucial for the feasibility of deep neural nets.

Downsampling

The nonlinear activations of convolutional layers are fed into a donwsampling stage. This reduction in resolution yields spatial invariance.

In its earliest beginnings, downsampling was performed by applying a non-linearity on the weighted average of the input units. Further possibilities are averaging as shown in figure 7.4 or taking the L2-norm within a small spatial window of the feature maps [49]. In 1992, the so-called *MAX-pooling* method was introduced [55]. Since Scherer et al. showed in 2010 that MAX-pooling generalizes better and converges faster than any others, it has established as the downsampling approach of choice [44]. Hence, the idea and properties of general downsampling are examined based on MAX-pooling.

MAX-pooling computes the maximum of a local patch of units in one feature map. Typically, neighboring pooling units do not overlap. As depicted in figure 7.5, pooling layers can be imagined as multiple pooling grids, each of them operating on $z \times z$ units. At the same time, the center of adjacent grids are spaced s pixels apart.

Figure 7.5 shows that non-overlapping pooling is obtained for $s = z$. Here, each unit is represented in only one pooling grid. Opposed to that, for $s < z$ the region of impact of two neighboring pooling grids overlap. Slightly better results in the top-1 and top-5 error rate can be attributed to overlapping pooling. Moreover, Krizhevsky et al. claim that networks with overlapping are more robust towards overfitting [27].

The effect of pooling can be enhanced by even using coarser pooling grids. Again, figure 7.5 shows that fewer pooling units than input units down-sample stronger and lead to layers that are invariant to larger spatial translations. Additionally, this leads to reduced computational time and less memory demand.

Changing the dimension in which the pooling grids are extended, effects the invariance learning. While spatial pooling correlates to translation invariance, pooling over feature maps is associated with an invaraince towards other transformations in the input [19].

A more descriptive perspective on pooling and invarince to translation is obtain by considering the processes of pooling at the feature level. For that, recap that the convolutional layer detects features. When generating patterns from such basic features their relative position towards each other can vary slightly [28]. Assume for instance, a CNN that classifies the MNIST data set. If an input image contains the endpoint of a roughly horizontal edge in its upper left and right corners as well as the endpoint of the a vertical edge in its lower right input pixels, that is enough information to deduce a 7 [29].

Pooling exploits this freedom by coarse-graining the position of the features and finally merges semantically similar features into one [30]. Obviously, invariance to local translation is in particular useful if the pure existence of several features is more important than their exact position towards each other. Thus, pooling happens at the expense of loosing position information.

This motivates a global interpretation of down-sampling. Correlating spatial dimension to image content, shrinking feature map dimensions are equivalent to compressed content. Transferring these maps to a subsequent convolutional stage means that a kernel of the same size has a more global view on the features and their relation as before. Hence, down-sampling results in a large input image context and allows the capturing of structures on a larger scale [58], [9].

Both, its natural invariance towards translation as well as the reduction of parameters make pooling an efficient prevention of overfitting. Moreover, it compresses the semantics of the input step-wise until global objects can be recognized [46]. In each passing through a whole convolution layer, higher level features are extracted by discarding locality information, a process comparable to a contradiction or compression of the image [43]. Lining up multiple of these layers, generates a Convolutional Neural Network. They transform highly resolved inputs with high-level features such as edges into coarsely resolved outputs with low-level features, e.g., a car.

7.1.2. Classification

For classification, off-the-shell classifiers such as fully-connected layers are convenient. To bridge the convolutional structure with traditional neural network the feature maps of the last convolutional layer are vectorized. After feeding them into fully-connected layers, a final softmax logistic regression layer computes the scores for each class [20, 27].

7.1.3. Implementation

Neural networks for pattern recognition are built up by a stack of alternating convolution and downsampling layers, followed by a fully-connected classifier, exist in their simplest form since the early 1980s [18]. While these precursors were handcrafted, LeCun et al. applied backpropagation to let the network learn its parameters in 1989 [28]. Despite the fact that none of the former state-of-art approaches classified the MNIST data set more accurately than his *LeNet* neural network, no remarkable advancement in the field of convolutional neural networks was recorded within the following two decades [15]. The remedy was long in coming, but finally the breakthrough of deep convolutional neural networks was marked in 2012. During the annual ImageNet Large Scale Visual Recognition Challenge (ILSVRC), one of the most well-known object classification task challenges in computer vision community, Alex Krizhevsky, Ilya Sutskever and Geoff Hinton presented their deep convolutional neural network *AlexNet*. While the runner-up network reached a top-5-error of 26% when classifying images, *AlexNet* significantly outperformed it with an error-rate of only 15.8% [27].

The years of silence are owed to a bilateral constraint of deep neural networks. One inherently algorithmic, the other of pragmatic nature.

As discussed in section 3.3.1, the key issue of deep learning is the fading or amplification of feedback error signals as the number of layers increase. Although not solving the fundamental problem in its heart, Krizhevsky et al. combined GPU-based training with non-saturating nonlinearities to alleviate its influence. Further, they found that locally nor-

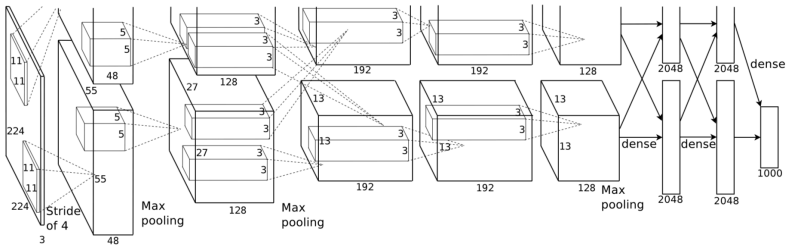

Figure 7.6: As indicated by thin lines, training is distributed on two GPUs that communicates only at certain layers. The applied training scheme is the momentum update [27].

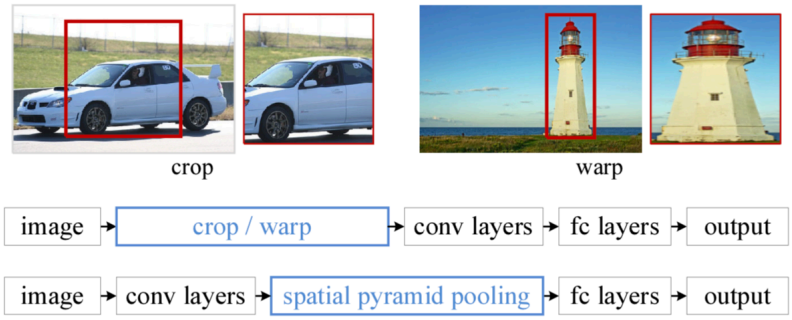

Figure 7.7: (*(TOP)*) cropping or wrapping to a fixed size. (*(MIDDLE)*) Process in conventional CNN. (*(BOTTOM)*) Process with Spatial Pyramid Pooling. [21]

malizing the nonlinear activations improves generalization even more. In addition to these modifications, *AlexNet* incorporates overlapping pooling as mentioned in section 7.1.1. The resulting network is shown in figure 7.6.

The fully-connected layers that are used for classification suffer from two serious drawbacks. First of all, they are prone to overfit and thus hamper the generalization ability. Heavy use of data augmentation and dropout are relatively easy ways to alleviate this [27].

In addition to that, fully-connected layers need to have a fixed-size input by definition. Despite the fact that convolution layers are not constraint to inputs of a particular size, CNNs require a fixed input image to allow classification. Since most data sets comprise images of arbitrary sizes, these are either cropped or wrapped to obtain the requested resolution. As shown in figure 7.7 cropped region might suffer from content loss, whereas wrapping leads to distortion of the image. Both can compromise recognition accuracy.

In 2015, He et al. suggested an additional *spatial pyramid pooling* (SPP) on top of the last convolution layer, as displayed in figure 7.7 [21]. This allows the removal of the fixed-size constraint without drastically modifying the network architecture. The SPP layer contains M so-called spatial bins, each applying *MAX*-pooling to the response of all filters [21]. The generated output is a kM dimensional vector where k denotes the number of filters in the last conventional layer. These vectors are the fixed-size inputs of the fully-connected classifier. Figure 7.8 illustrates the method.

7.2. Network in Network Approach

In 2013, Bengio et al. released a paper on representation learning, answering the essential question of the ingredients for a good representation [10]. Among others, hierarchically organized explanatory factors is one of them. Deep learning exploits this assumption by reusing features, i.e., constructing multiple levels of representations with increasing abstraction. More abstract concepts are composed of easier ones and are more invariant to local modifi-

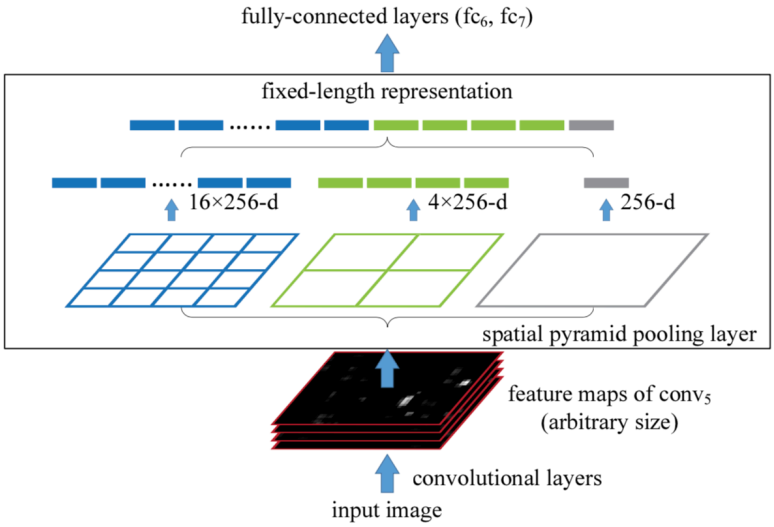

Figure 7.8: Network structure with a spatial pyramid pooling layer. After the last convolution layer with 256 output feature maps, SPP is applied. The vectorized output is passed to fully-connected layers denoted with fc_6 and fc_7 [21].

cation of the input. Bengio et al. conclude that representations that are capable of capturing such concepts are generally nonlinear functions of the raw input [10]. This reveals a systematical deficit of conventional CNNs since the convolutional filters are linear models. One suggested approach to compensate the linearity is an over-complete set of filters. Multiple filters are learned to detect different variations of the same concept. A *MAXOUT* stage reduces the number of filters by *MAX*-pooling over the direct outputs from linear convolution. Networks that apply this strategy are convex function approximators and thus not capable to capture nonlinearities.

However, following Kolmogorov's theorem [6], MLPs are known to be universal function approximators. Lin et al. exploit this idea by substituting convolutional filters by micro neural networks, creating the idea of *Network in Network* (NIN) [34].

7.2.1. Feature Extraction

In their eponymous paper *Network in Network* [34], the authors design a convolution-like layer yet, using a MLP consisting of multiple fully-connected layers with non-linear activation functions instead of convolution filters. As before in convolution, this MLP is shared among all local receptive fields and feature maps are generated by sliding the MLP over the input.

This sequence of steps is the same as cascaded *cross channel parametric pooling*. Cross channel parametric pooling describes the weighted linear recombination of input feature maps and applying a ReLU nonlinearity. Cascading implies an over and over repetition. According to [34], this is equivalent to a 1×1 convolution filter.

7.2.2. Classification

Besides, the classification structure is fundamentally adapted. Due to numerous disadvantages of fully-connected layers, a *global averaging pooling layer* is favored. Averaging over each feature map and feeding this directly into a softmax function yields a more native classification as with fully-connected layers. Here, more native means that averaging pooling layers can be interpreted more intuitively as category confidence maps.

A further noteworthy advantage of global averaging is its behaviour as structural regularizer. It is not only free of parameters and, hence, not fragile in overfitting but also more robust to spatial translation of the input [34].

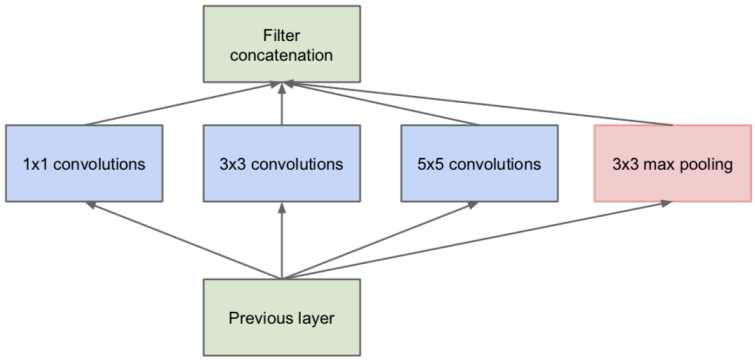

Figure 7.9: Naive Inception layer that consists of convolution filters with different receptive fields. This empowers the layer to find correlated units with varying spatially distance [50].

7.2.3. Implementation

A well-known application of the *NIN*-principle are so-called *Inception*-layers. Since initially introduced during ILSVRC14 in the CNN *GoogLeNet* by Szegedy et al. [50], it has been the new state-of-the-art for classification and detection.

Its driving key idea is founded on the theoretical analysis of Arora et al. [8]. The main point of their work is a scheme to layer-wise construct an optimal network topology for datasets that feature probability distributions which are representable by a large, very sparse deep neural network. New layers are constructed by analyzing the statistical correlation of the preceding layer activations and cluster such neurons whose outputs are highly correlated [3] [8].

Based on that, Szegedy et al. approximate the optimal local sparse structure of convolution layers [50]. Note that building blocks have to consist of convolutional layers in order to maintain spatial invariance. However, their filter sizes depend on the spatial freedom of the features, which in turn depend on the depth level of the network. For instance, in input-near layers a majority of correlated units are rather concentrated in local regions. The few correlations that are spatially set wider apart are covered by convolution over larger patches. The deeper one goes, the less patches of increased size will appear. For convenience, Szegedy et al. concentrate on filter of three varying receptive fields, namely 1×1, 3×3 and 5×5. Finally, the overall network is a combination of those layers. Since pooling proved to be beneficial, a fourth pooling option is added. As depicted in figure 7.9 their outputs are concatenated in a joint output vector which becomes the input of the following layer [50].

The ratio of 3×3 and 5×5 should increase with increasing depth and hence increasing abstraction of the features that are captured. However, this is hampered by the large computational effort, needed for convolution with large receptive fields, in particular if applied to numerous filters. To prevent a computation blow-up, Szegedy et al. introduce additional dimension reductions, accomplished by 1×1 convolution. Besides, these reductions are equipped with a ReLU nonlinearity [50]. The final structure of the *Inception*-layer is shown in figure 7.10.

As shown in figure 7.11, *GoogLeNet* starts with a conventional network to obtain a certain level of feature abstraction before stacking *Inception* layers above each other. Despite its 22 layers it uses 12 times less parameters as *AlexNet* while pushing the top-5 error rate from 15.8 % to 6.67 %.

Introducing the alternative Image Classification approach is motivated by the importance of these networks for further image processing. *GoogLeNet* and its successors are often found as building blocks for image segmentation as will be depicted in the following chapter.

[3] Within a network whose egde weights are in [-1,1], Arora et al. define related nodes of a hidden layers as node with common neighbour to which they are attached with a +1 edge [8].

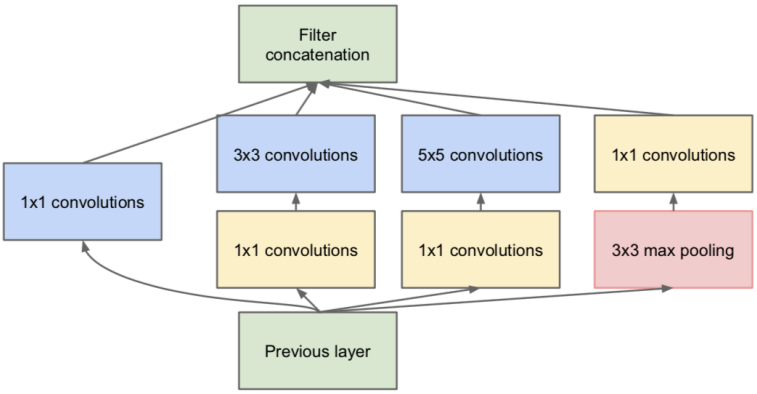

Figure 7.10: A low dimensional embedding might compress a lot of information about a large input image and hence a dimension reduction does not lead to a loss of information. To prevent unbearable computational cost, Szegedy et al. add 1×1 convolution before convolution with large receptive fields and after the pooling path [50].

7.3. A CNN as MNIST Classifier

Within the course of the report, the MNIST data set proved itself to be a grateful guinea pig. Section **??**, I examined the applicability of shallow neural networks. Although the MLP I used is rather small, it is prone to overfit and counteracting this behaviour by regularization turns out to be a delicate issue that demands a lot of fine-tuning.

This gives rise to classify the MNIST data set with a CNN. However, in this section I rather aim at exploring the behaviour of CNNs than at outperforming classifiers that are based on common MLPs.

Figure 7.12 shows the schematic construction of the CNN model I designed. While red denotes the part for classification, the layers in shades of green are responsible for the feature extraction and yellow marks additional dropout regularization layers. The kernel sizes and further implementation details of the two alternating *convolution* and *MAX-pooling* layers are summarized in table 1.1. The corresponding *Tensorflow* code is attached in the appendix B.

Again, I decided to use the *Adams* optimizer, which is an advanced instance of gradient descent to train the model and its performance is measured with the cross-entropy loss. Opposed to my earlier MLP-based attempt in section 3.6, I train the CNN classifier for 1000 epochs. Figure 7.13 visualizes some randomly chosen kernels.

Although, the trained filters do not seem to exhibit any predictable patter, their outputs partly do.

Assuming for instance, that the first convolution layer receives a handwritten three. In the generated output, its shape can still be recognized although the image is alerted by the filters. However, the relation between in- and output is less distinct in the second convolutional layer. After going through a MAX-pooling stage and applying further kernels modifies the image that strongly, that a human can not realize any digit. CNNs though, seem to be capable. Classifying the images with a fully-connected layer and the cross-entropy loss leads to a final training accuracy of 96.88%.

Since only validation uncovers the relevant usability of the model, the trained CNN is applied to the test set. The graphs in figure 7.14 shows that, although the model performs slightly worse during evaluation and correctly predicts the digits with a probability of 96.15%, evaluation and training accuracy are of the same order of magnitude. Hence, opposed to my first MLP classifier the more elaborated CNN model does not tend to overfit. My implementation confirm that an adaption of the neutal networks architecture, combined with further regularization prevent overfitting and allows a precise classification of the MNIST data set.

Figure 7.11: Training deep networks with gradient descent methods and BP is hampered by vanishing gradients. Szegedy et al. tackle this issue by adding auxiliary classifiers. While adding their loss to the total loss of the network with a weight decay during training, they are discarded while inferring [50].

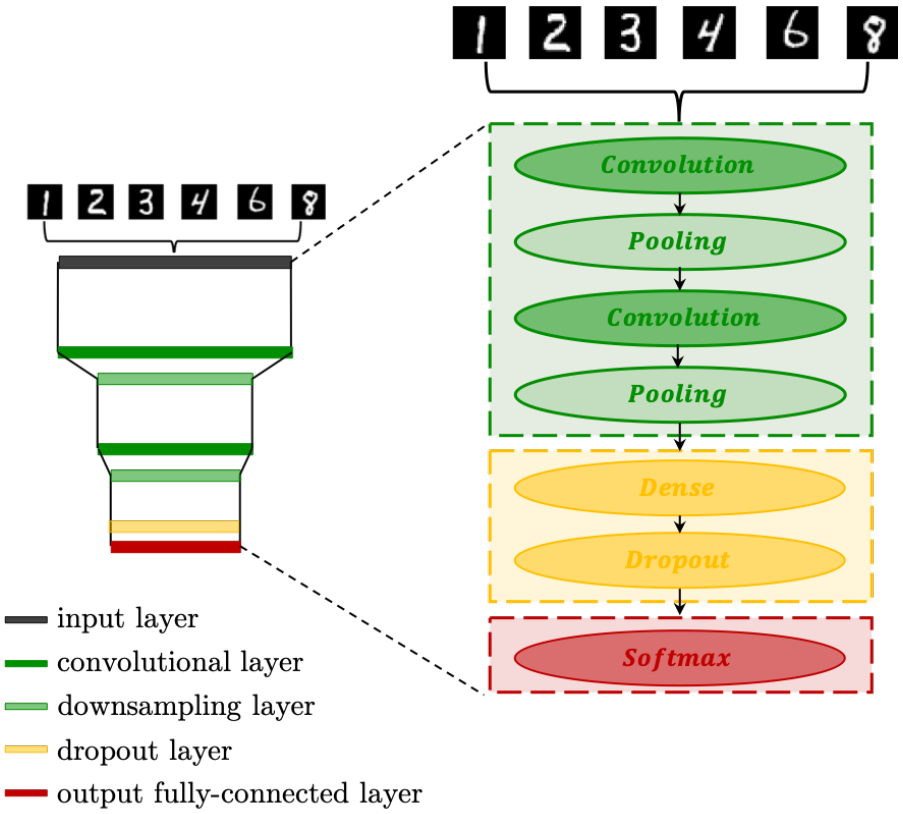

Figure 7.12: The illustration shows the schema of the CNN that I used to classify the MNIST data set. Each oval form denotes one different layer. The first four are responsible for feature extraction as denoted by the green color. Opposed to that, red denotes the classification layers. Between those block I added a regularization intermediate stage. *Dense* relates to a fully-connected layer and *pooling* to *MAX*-pooling. Furthermore, a flattening is interposed between the feature extraction and dropout block.

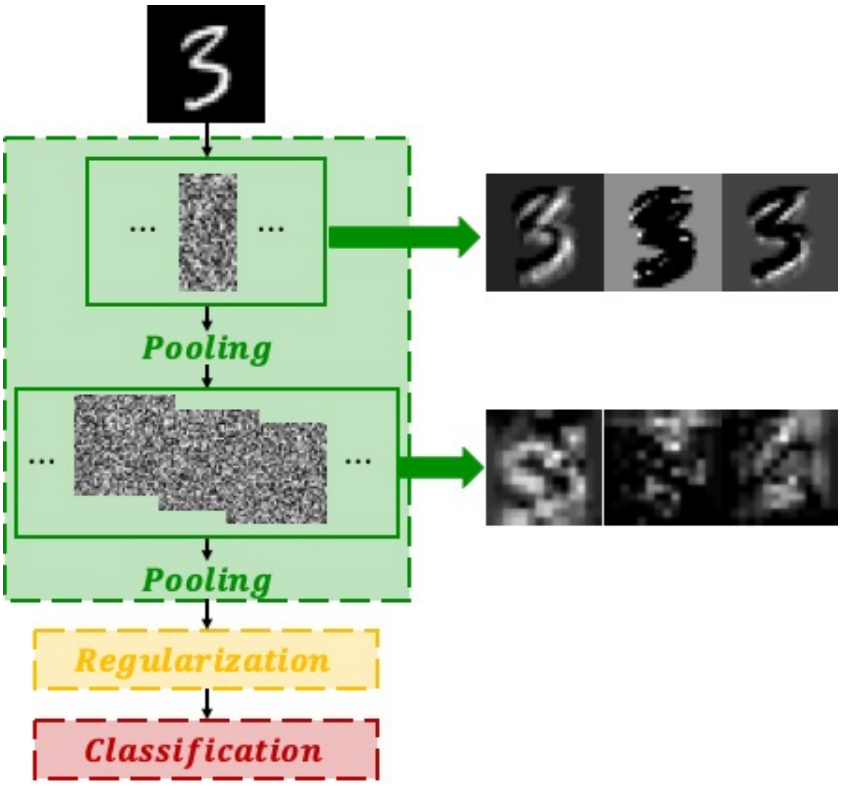

Figure 7.13: In this schematic architecture of the CNN the convolutional stages are substituted by some of the trained filters from which they built up. The figure illustrats the path of an input image through the classification part of the model. However, for reasons of visualization the sizes of kernels and intermediate images are adapted.

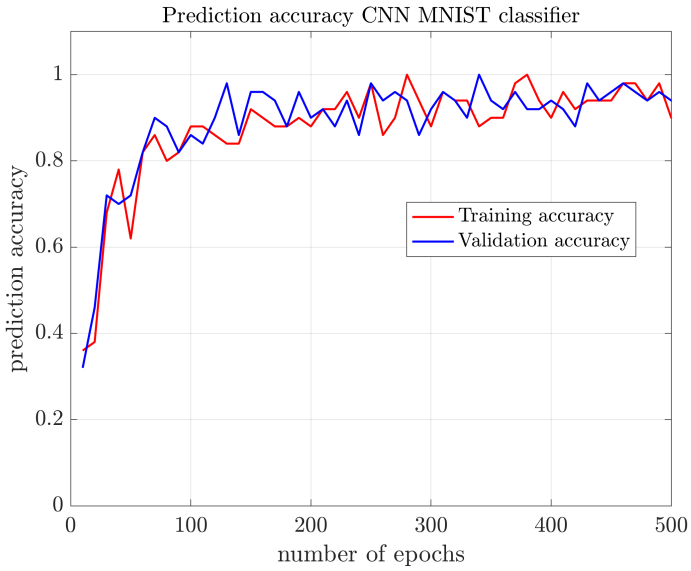

Figure 7.14: Training the CNN MNIST data set with *Adams* for 1000 training epochs and finally evaluating lead to a prediction accuracy of 96.88% respectively 96.15%.

	input size	kernel size	output size	padding	stride	activation	other
Input	–	–	[128,28,28,1]	–	–	–	–
Conv1	[128,28,28,1]	5×5	[128,28,28,32]	*same*	(1,1)	ReLU	32 filters
Pool1	[128,28,28,32]	2×2	[128,14,14,32]	–	(2,2)	–	–
Conv2	[128,14,14,32]	5×5	[128,14,14,64]	*same*	(1,1)	ReLU	64 filters
Pool2	[128,14,14,64]	2×2	[128,7,7,64]	–	(2,2)	–	–
Dense	[128,7*7*64]	–	[128,1024]	–	–	ReLU	1024 units
Dropout	[128,1024]	–	[128,1024]	–	–	–	p_{train} = 0.9

Table 7.1: The table summarizes information on the layers. The first dimension of the input and output size denotes the number of elements in one batch, whereas the last denotes either the channel or the number of kernels in the preceding layer. Second and third give the width and height dimensions of the feature map. Moreover, padding in mode *same* refers to a padding such that the input size is maintained. Opposed to that, *valid* means no zero padding. Furthermore, the stride is defined as a tuple. Each entry denotes to the stride in one of the dimensions of the width and height dimension of the input tensor. Note, that dropout in *Tensorflow* is only applied during training.

8

Image Segmentation with Convolutional Networks

Classification results in denoting the entire input image to one unique class. In contrast, semantic segmentation detects instances of multiple objects and yields dense predictions. This is equivalent to fine-grained classification where labels are inferred for each pixel separately. While classification tasks lead to a single vector that contains the probability of each class, semantic segmentation leads to a score map that gives class predictions for each pixel.

The task of semantic segmentation is considerable more complex than the previously introduced classification problem, since it consists of two prior challenges. First, segmentation has to deal with classification since a detected object has to be assigned to a semantic concept correctly. Besides, the classification label for a pixel must be aligned to the appropriate coordinates of the output score map. That is a localization problem.

Recently, deep convolutional neural networks are driving advances in whole image classification and state of the art ConvNets cover the semantic issue of segmentation quite well [27, 50]. Among others, new concepts such as Spatial Pyramid Pooling empower them to perform well in tasks with structured output such as bounding box object detection [21]. These improvements attract attention of science to further employ the concept of ConvNets for image segmentation tasks.

The crux of the matter is the bilateral and inherently contradictory requirements of image segmentation. The classification performance of ConvNets is based in invariance towards translation and other transformations. Opposed to that, localization results depend on the position of inputs i.e. it is a transformation-sensitive problem. In terms of deep learning, different requirements lead to different models and there is a natural trade-off between classification and localization accuracy. As depicted in figure 8.1 classification models such as *AlexNet* or *GoogLeNet* are cone-like shaped, compressing the input to high-level features. Small and coarse hidden layers are responsible to extract these features, at the expense of location information. Classifiers are normally densely connected to the final, low resolved feature map to assign the feature to a semantic object.

Localization favours a relatively large, highly resolved feature map that contains a lot of spatial information. Models rather take the shape as shown in figure 8.1. Finally, pixel-wise labels are generated with locally connected classifiers [39].

A well-designed ConvNet for image segmentation has to cover both sides simultaneously. In 2014, Long et al. faced this challenge by taking advantage of special ConvNets, so-called *fully-convolutional networks* (FCN). Semantic segmentation becomes feasible by adapting these ConvNets to enable pixel-wise predictions [46]. While maintaining the structure and idea of fully-convolutional neural nets, several lately released works try to further improve the ansatz. Focus of research are three main aspects, namely *context embedding*, *resolution*

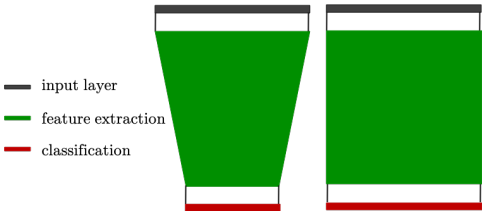

Figure 8.1: Different learning tasks result in different network architecture. *(LEFT)* Classification networks reduce the input image to one single prediction by discarding spatial information. This becomes apparent in the cone-like shaping of these nets. *(LEFT)* Opposed to that, networks that aim on localizing objects in an image keep spatial information and hence do not reduce the input dimension. Again, this is depicted in the networks architecture. Input and output are of the same dimension .

enlargening and *bounadry alignment*.

The following chapter is a review of the active fields of research. For each, the dominating difficulties, possible solutions and their implementation are summarized. Beforehand, the basic concept of fully-convolutional networks is introduced.

8.1. Fully Convolution Networks for Image Segmentation

Standard deep convolutional neural networks as initially introduced for classification tasks, share a common structure. As presented in chapter 7, high-level features are extracted by alternating convolutional and downsampling layers and finally classified by fully-connected or global pooling ones [27, 50].

Despite performing well in classification tasks, two design properties of ConvNets hamper semantic segmentation. One problem is induced by the fully-connected or global pooling layers, the second by downsampling and the accompanying loss of spatial information. While the latter is less easy to solve and until today a hot topic in research, handling the first is relatively straightforward.

Semantic segmentation aims to achieve a dense classification, hence an optimal output has the same size as the input and each of its pixels is assigned to one class. However, as examined in section 7.1.3, fully-connected layers have stipulated dimensions. Given a fixed-sized input, they generate an output vector of prescribed dimension $K \times 1$, with K the number of classification classes. Although, as demonstrates in section 7.2.3, global average pooling layers allows flexible input sizes, still they generate an output tensor of dimension $1 \times 1 \times d$, d denoting the number of feature maps in the last convolution layer. Consequently, both these layers do not only modify the resolution of feature map but alert the spatial information between the pixels by changing the data structure. Opposed to that convolution conserves data structures. Thus, to yield an output score map of input image size, fully-connected and global average layers have to be substituted. However, fully-connected layers are equivalent to convolutions with kernels that cover the entire input image. Hence, discarding the last layers of standard ConvNets and substituting them by convolutions with a kernel of size $1 \times 1 \times d \times K$ allows a prediction of class labels at each pixel of the last feature map. This casts ConvNet in *deep fully-convolutional neural networks* (FCN). Here, the attribute fully-convolution is related to the fact that every layer involves convolution [46]. A FCN derived from a ConvNet generates a score map and hence is capable of performing semantic segmentation. However, Long et al. found resulting segmentation dissatisfying coarse [46].

This outcome reveals the second issue when alienating standard classification networks for semantic segmentation tasks. While the modification of the standard ConvNets to FCN in principle allows segmentation, the output dimensions are typically alerted by downsampling and hence smaller than the input image size. However, upsampling strategies increase the resolution of the score map and connect coarse outputs to dense pixels [46]. Long et al. apply *fractionally strided convolution*, also called *deconvolution* to perform upsampling [22].

Although might associated with the term ”deconvolution”, this procedure is not equivalent to the inverse of convolution. Rather than reproducing the original input, it only recovers the input’s shape. Not only for this reason but also since it is more closely related to the underlying mathematical operation, it is advocated to use the more descriptive expression *transpose convolution*. For further explanation, the convolution has to be rewritten as a linear

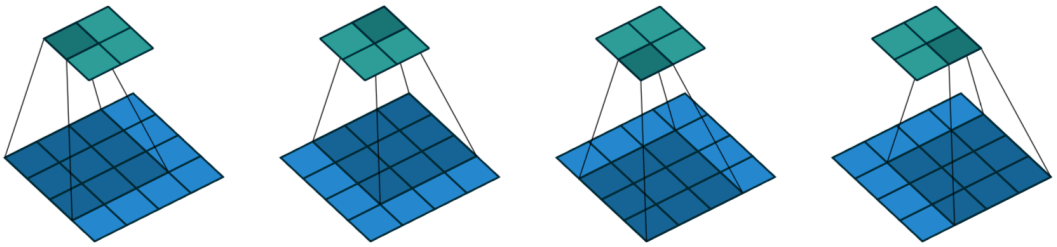

Figure 8.2: Convolving a 3×3 kernel over a 4×4 input image using no padding and unit stride yields a 2×2 output map [16].

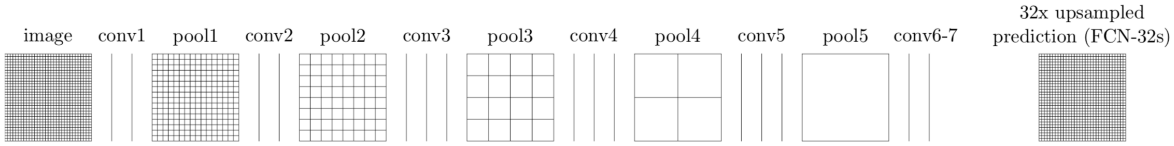

Figure 8.3: Long et al. extract features with the help of 5 alternating convolution and downsampling layers. This part of the neural net is responsible for extracting semantic objects. conv6-7 are additional 1×1 convolutions which predict scores at each coarse output pixel before upsampling the feature map in one step by factor 32 [46].

matrix operation. Taking for instance the common convolution from chapter 6 as represented in figure 8.2 and assuming the weights w of the kernel are learned with backpropagation. Unrolling the input from left to right and from top to the bottom yields a vector of size 16. The weights of the kernel define a unique sparse matrix $C \in \mathbb{R}^{4\times16}$, with a nonzero pattern as shown in equation 8.1.

$$\begin{bmatrix} x & x & x & 0 & x & x & x & 0 & x & x & x & 0 & 0 & 0 & 0 & 0 \\ 0 & x & x & x & 0 & x & x & x & 0 & x & x & x & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & x & x & x & 0 & x & x & x & 0 & x & x & x & 0 \\ 0 & 0 & 0 & 0 & 0 & x & x & x & 0 & x & x & x & 0 & x & x & x \end{bmatrix} \tag{8.1}$$

In the forward pass, a matrix multiplication generates a 4 dimensional outputs. Reshaping it reproduces the 2×2 output as shown in figure 8.2. In the backward pass, the error is backpropagated by multiplying the loss with C^T. This maps a 4-dimensional vector in a 16-dimensional space. Both, the forward and backward pass use the same matrix C which in turn only depends on the kernels weights.

The *transposed convolution* is then simply swapping forward and backward pass. Consequently, the same kernel defines both, common and transposed convolution. Whether the convolution is transposed or not only depends on whether C is assigned to the forward or to the backward pass through the network.

Despite the parameters of the *transpose convolution* can be learned by pixel-wise training, Long et al. initialize them correspond to bilinear interpolation [46].

As shown in figure 8.3, a first trial includes upsampling in one stage. However, the last feature map contains only little spatial information and a simple reshaping of it results in insufficiently coarse segmentation. Deconvolution as applied here, only recovers the input image shape but it does not reconstruct the locality information that was discarded during pooling. To embed this lost context, Long et al. equip the FCN with additional skips. The lower the layers i.e. the closer to the input the more contextual information are contained. Hence, fusing copies of these layers with the upsampled complement by element-wise addition improves segmentation remarkably. A detailed description is illustrated in figure 8.4.

FCNs are instances of *auto-encoder*, neural networks that are trained to reproduce a copy of their input. Their general structure is illustrated in figure 8.5. Obviously, they consist of an encoder- and decoder-stage. The encoding part of FCNs, generates a coarse prediction

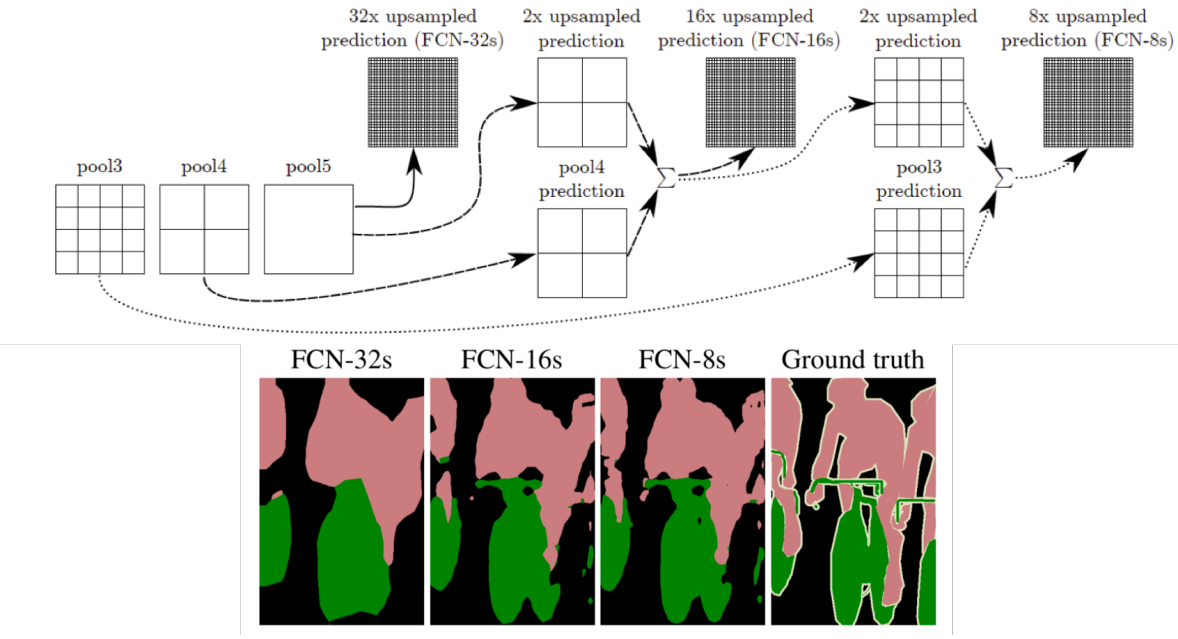

Figure 8.4: Simply upsampling the last feature map by a large factor limits the scale of detail in the output. This is addressed by adding a corresponding feature map from the feature extraction part. *(TOP)* Note that for reasons of depiction, convolution layers are left out. Instead of upsampling by factor 32, predictions are derived from twice upsampling the output feature map with factor 16. At the same time an additional class prediction is yield by convolving the pool4 layer with a 1×1 kernel. Element-wise adding them yields a more detailed segmentation as in the first trail. This can even be extended by reusing this prediction and fusing it with lower level feature maps. *(BOTTOM)* The more fusion stages are involved, the more precise the segmentation is [46].

while extracting latent features by compressing the input. The decoding block, reverses the encoding by enlarging the resolution. However, due to the loss of spatial information while compressing, decoding is not the exact inverse of encoding but only an approximation of it [58]. To refine the prediction and hence improve its quality, as much context as possible has to be embedded by recovering spatial information from the encoder block.

Encoding is mainly based on the principles of classification and hence established architectures can be exploited. Opposed to that, upsampling and embedding context during decoding are the hottest topics of image segmentation. Since the first success of Long et al. numerous improvements were suggested [22]. While most of them maintain the encoding structure of FCN some alerting them to achieve further advances while decoding.

The remainder of the chapter is devoted to the most important fields of research, including methods for upsampling and retrieving spatial information. Although boundaries between the research fields are blurry and modifications in one alerts the network behaviour with respect to another, all three topics are investigated separately.

8.2. Methods of Upsampling

Essential part of encoding based on ConvNets is the downsampling. Hence, to reconstruct a copy of the input the resolution of the feature maps has to be enlarged.

Initially, FCNs proposed *transposed convolution* with fixed kernels to enlarge the resolution of the output. Missing spatial information is added by copying feature maps from the encoder block.

Zeiler et al. primarily introduced *unpooling* as a reverse operation of *MAX*-pooling [58]. Approaches such as *Deconv-Nets* and *SegNet* employ this scheme to enlarge the resolution and add spatial information at once. As shown in figure 8.6, location information that is

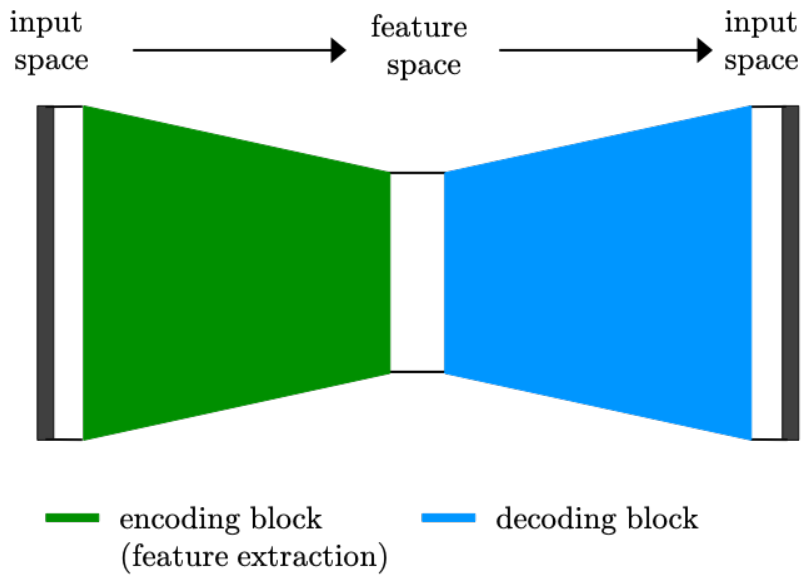

Figure 8.5: In the encoder block the input image X is mapped from the input to the latent feature space. This yields an encoded and compressed representation Z which in turn is mapped back to input space in the decoder block. Here, the output X' is generated such that the reconstruction error between X and X' is minimal [19].

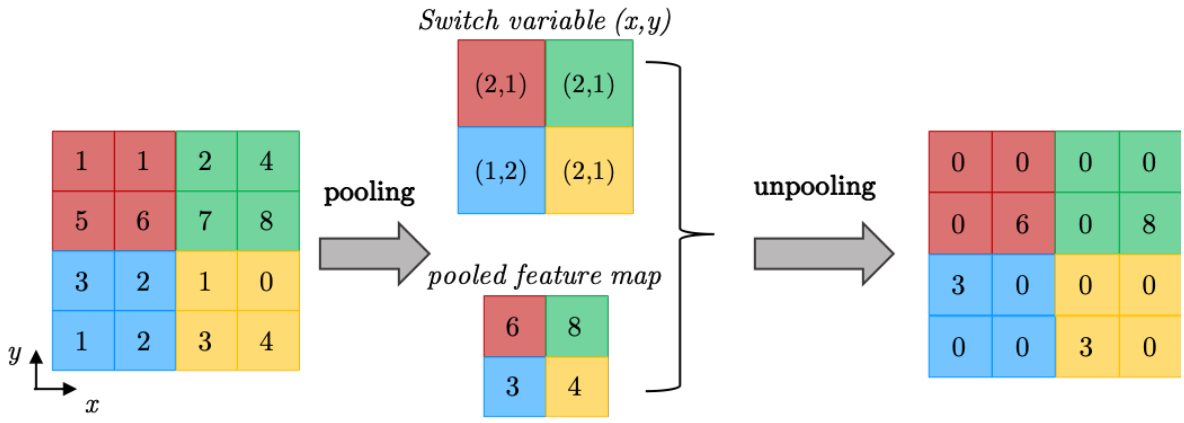

Figure 8.6: The scheme visualizes, how *MAX*-pooling with filter size 2×2 has to be adapted for later unpooling. Pooling is applied to feature maps during the encoding phase. While downsampling, switchs record the location of the chosen maximal activation. The reverse operation, the upsampling, uses these in combination with the pooled map to enlarge the resolution.

discarded during downsampling is captured in separate variables. A variable called *switches* stores the indices of the activations with maximal values. Together with the pooled map, the positions stored in switches is used to place each variable that is stored in the pooled map back to its original position. This preserves the original structure and hence no further contextual information is necessary.

Despite unpooling generates an enlarged output, most of the entries are zero elements. Therefore, multiple filters are learned and applied in a transposed convolution layer to densify the sparse activation map [38].

Summarizing, comparing *Deconv-Nets* [38] and *SegNet* [9] to the basic FCN [46], they differ concerning three main characteristics. Deconvolution filters are learned by data and only used for densification. The employed unpooling strategy covers both requirements: it enlarges resolution and recovers spatial relation in unpooled outputs. Last but not least, instead of entire feature maps only switch variables and unpooled feature maps are copied to the decoding path. This fact yields a remarkable reduction in memory demand.

A more radical ansatz to address the issue of reversal downsampling, is based on *dilated convolution*. Although it is based on alternatively embedding context, it is discussed with respect to its effect on upsampling. It is driven by the question after structural differences in

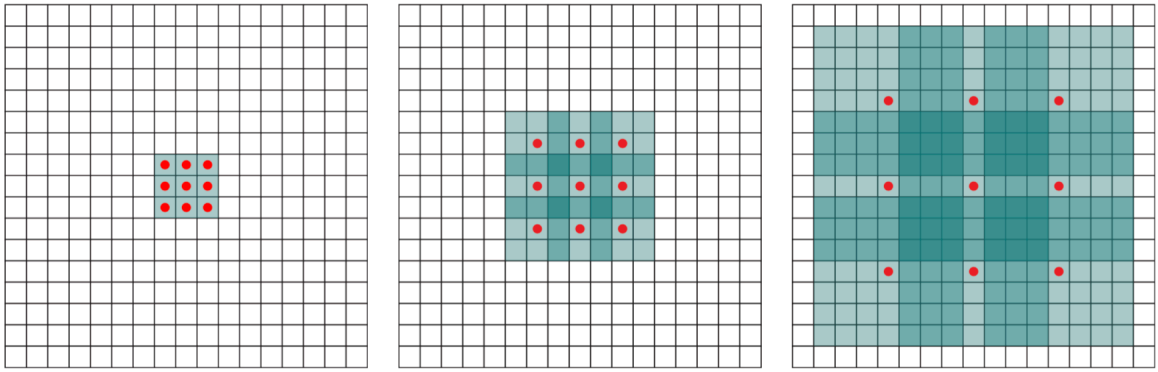

Figure 8.7: *(Left)* Standard discrete convolution has dilation rate $r = 1$ *(Middle)* Increasing rate r increases receptive field to 7×7*(Right)* With $r = 4$ the same parameters as before as used in an enlarged receptive field [57].

classification and dense prediction. While classification involves downsampling to aggregate multi-scale contextual information, this hampers the dense prediction in segmentation task since it needs multi-scale reasoning with full-resolution outputs. With the initially by Yu et al. [57] introduced module of *dilated convolution*, context can be implied without losing resolution. By omitting downsampling stages, upsampling in turns is obsolete while decoding.

Dilated convolution used for image segmentation tasks is defines as

$$F_{xy,r} = \sum_{m'=0}^{k} \sum_{n'=0}^{k} I_{x-r\cdot n', y-r\cdot m'} \cdot Q_{n',m'} \tag{8.2}$$

where r is the *dilation factor*. Setting $r = 1$ recovers the already known equation 6.5 and hence standard discrete convolution a special instance dilated convolution. As visible in figure 8.7, dilated convolution supports exponentially expanding receptive fields without losing resolution.

Dilating a convolution does not modify the parameter of the filter but the manner to use them. Indeed, the same filter is applied but using different ranges and different dilation. Despite an increase of the effective receptive field the number of parameters and operations stay constant since only non-zero elements need to be taken into account.

The basic so-called *context module*, developed by Yu et al. is designed such that it takes d input maps and outputs d feature maps of the same form. It consists of 7 layer that apply 3×3 convolution with different dilation factors r. In principle, this can be plugged into existing ConvNets. However, Yu et al. suggested attaching to a front-end module which is equivalent to the semantic encoder of FCNs. Compared to classical FCN as introduced by Long et al. [46], dilated convolution based networks yield more accurate segmentation results [57].

Summarizing, two main approaches to deal with the fateful downsampling established. FCNs directly reverse it by adding stages of upsampling while dilated convolution tries to avoid it and rather maintain the resolution.

8.3. Methods of Context Embedding

Context embedding is motivated by the fact that pixel-wise classification is hampered if only information that is retrieved from an isolated local surrounding is considered. Accuracy is improved and segmentation facilitated if the classifier exploits and analyzes contextual information about the whole image. Among the first, Mostajabi et al. introduced a hand-crafted hierarchical *Zoom-out* approach. The core idea is to compute representation at multiple levels with increasing spatial extend around the pixel to be classified and to combine all the features before classification [37]. The effect of context information is shown in figure 8.8.

Lately, different concepts to exploit contextual information has established. Four of them,

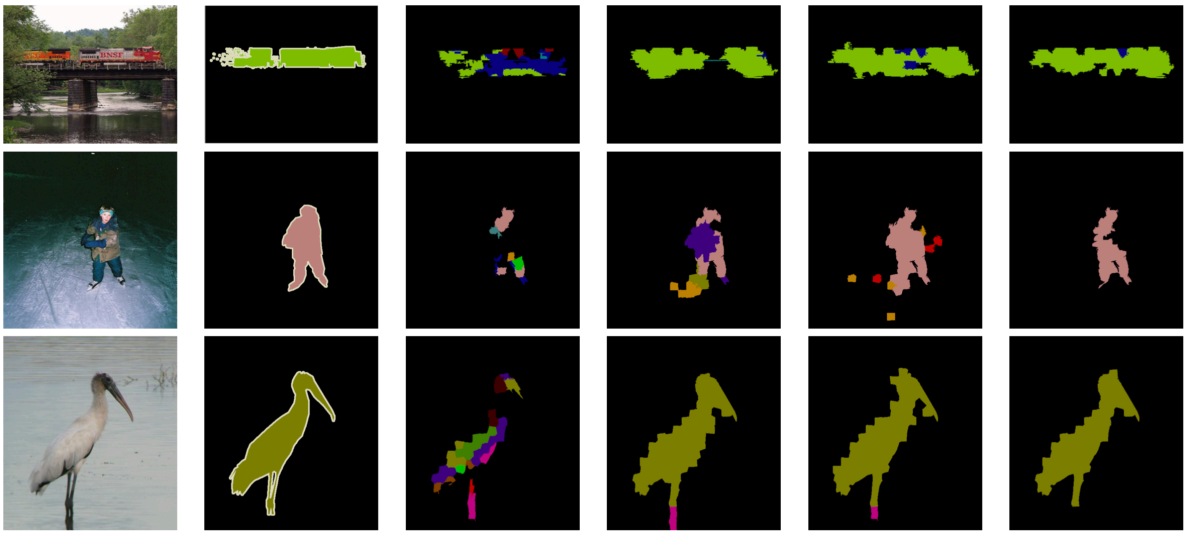

Figure 8.8: Three examples that visualize the effect of inclusion of context information. They are taken from VOC database and segmented based on the *Zoom-out* ansatz. Different colours denote different VOC classes whereby the background is black. Images from left to the right show the following: Original image, Ground truth [37].

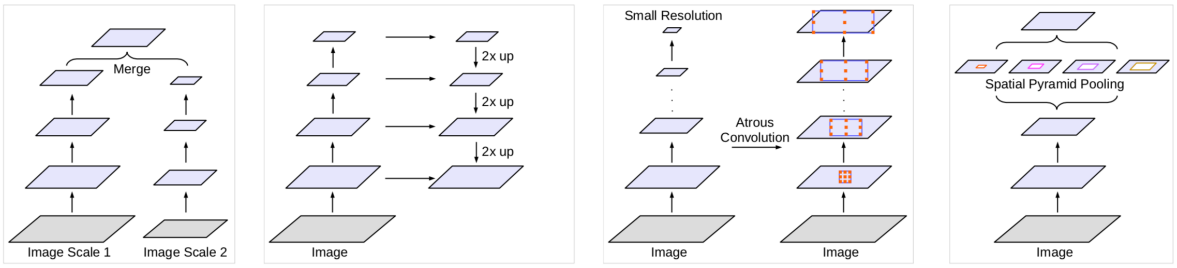

Figure 8.9: Alternative architectures to capture multi-scale context. Going from left to right it is *Image Pyramid*, *Encoder-Decoder*, *Dilated Convolution* and *Spatial Pyramid Pooling* [13].

namely *Image Pyramid*, *Encoder-Decoder*, *Dilated Convolution* and *Spatial Pyramid Pooling* are shown in figure 8.9. Due to the various but only slightly differing implementations of each branch, only the main idea and one descriptive example are given in the following.

Image Pyramids apply the same model to multi-scale inputs. The larger the scale the more detailed the extracted features are. Thus, feature responses from small scale inputs encode global context. For instance, Farabet et al. [17] generate inputs of multiple scales by transforming the image through a Laplacian pyramid representation. Each differently scaled image is fed into a 3-stage convolutional network and after aligning the spatial dimensions all outputs are merged. However, models of this class do not scale well for very deep convolutional neural networks [13].

With FCNs [46] such as *SegNet* [9] an important representative of *Encoder-Decoder* models is already introduced in section 8.4. Encoder parts gradually reduce the spatial dimension of feature map and capture longer range information in the deeper encoder output. From this output, the decoder recovers spatial information and image details [13].

Further, context can be captured stage-wise by employing *spatial pyramid pooling* as presented in section 7.1.3. As depicted in figure 8.10, the *Pyramid Scene Parsing Net* (PSP) performs spatial pyramid pooling at different grid scales.

Last but not least, *atrous convolution* is taken into account. Atrous convolution is equivalent to the previously introduced dilated convolution. The core idea is to convolve the output of a common ConvNet with multiple dilated filters of different rates. The more dilated a filter is, the more global context it aggregates. As shown in figure 8.11, a module based in atrous

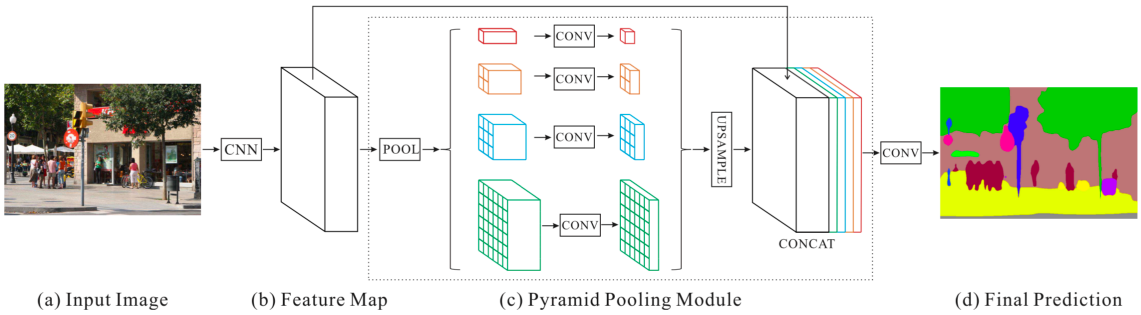

Figure 8.10: Structure of *PSPNet* which employs spatial pyramid pooling to capture long range information.*(a)* Shows the input image which is to be segmented. *(b)* A regular Convnets is applied. The output feature map is transfered to a spatial pooling module in *(c)*. There, different sub-region representation are captured, upsampled and concatenated. In this stage, local and global information is fused. *(d)* finally shows the pixel-wise prediction [59].

convolution can either be implemented sequentially or in parallel [13].

In general, it is worth mentioning that upsampling and context embedding are strongly connected. Indeed, a clear distinction is nearly impossible since, e.g., some upsampling strategies such as dilated convolution naturally come with context embedding.

8.4. Methods of Boundary Alignment

In turn, boundary alignment is strongly related to context embedding. It tries to refine the predictions near the object boundaries.

Among other, one often used approach is an additional post processing stage where *Conditional Random Fields* (CRF) improve the segmentation results, yield by ConvNets. *DeepLab* [12], as shown in figure 8.12 appends a fully connected CRF stage. Further examination and explanation of CRF would exceed the framework of the present report. However, details can be found in [12] which in turn refers to Krähenbühl and Koltun [26].

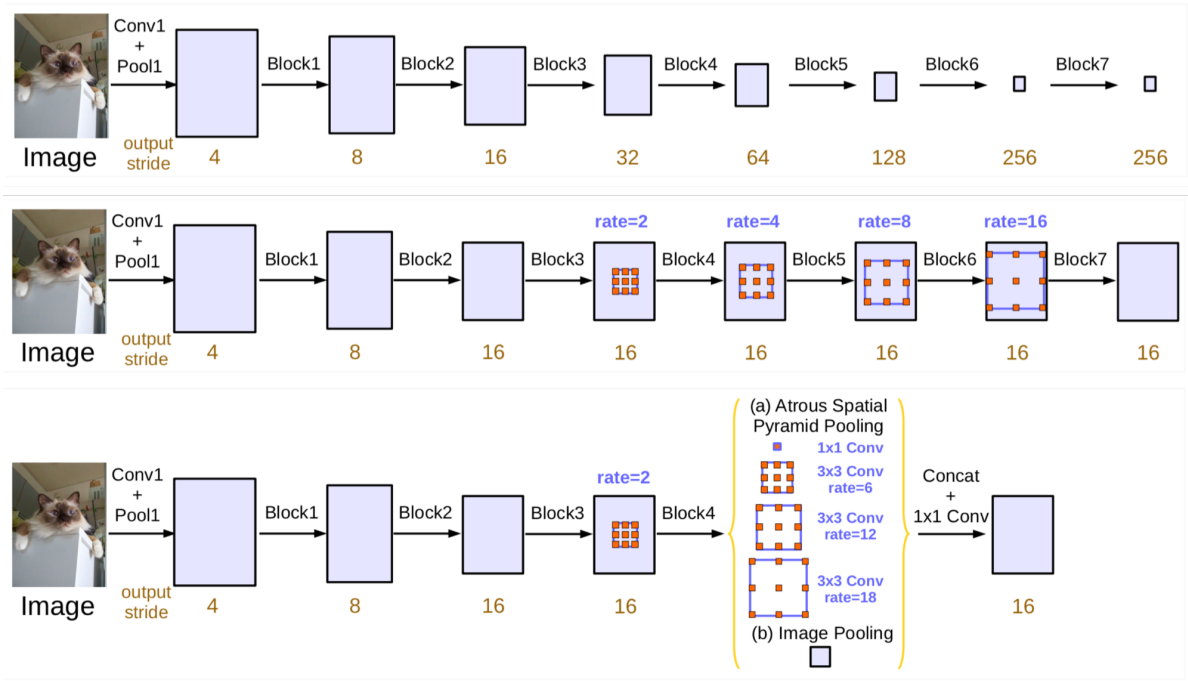

Figure 8.11: *(TOP)* Starting with a regular Convnet, cascaded modules based on atrous convolution can be yield duplicating several blocks of the original Convnet. *(MIDDLE)* Feature map size is maintained by convolving with dilated filters with increasing rates. This structure rather resembles an encoder-decoder structure like FCNs as presented in section 8.1. *(BOTTOM)* The idea of atrous convolution can also be used in parallel. This structure can be recovered in segmentation nets like *DeepLabV2* [14] and is understood as an adopted version of *spatial pyramid pooling*. [13]

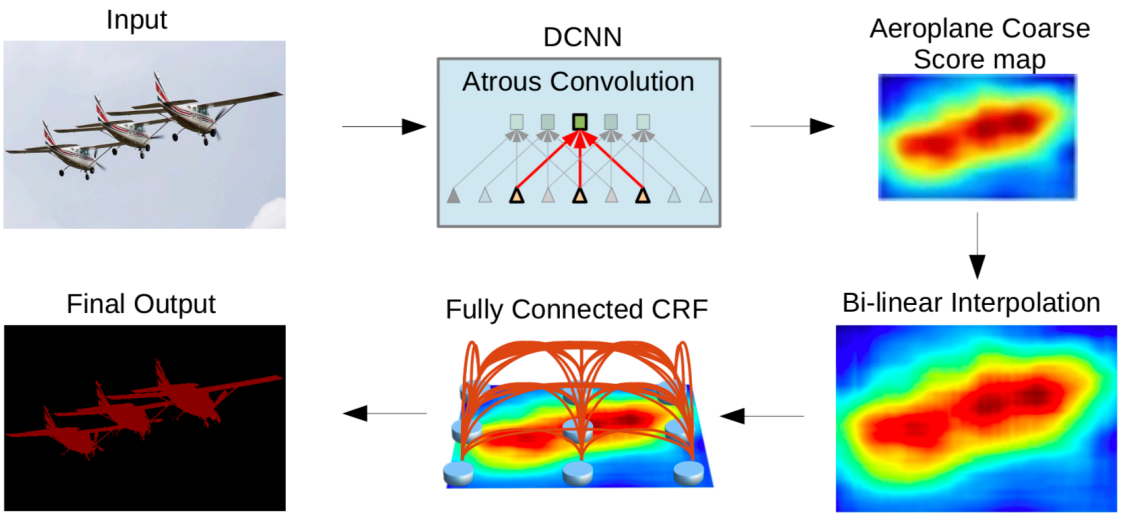

Figure 8.12: The *DeepLab* model gradually improves the segmentation performance. A coarse score map, generated by a Convnet is upsampled by bi-linear interpolation. A fully connected CRF stage is appended to further refine the segmentation [12]

9

Conclusion and Research Strategy

An intrinsic challenge of DIP is the complexity of images. The vast amount of information which they contain, hampers an efficient processing based on machine learning approaches. For image classification problems, e.g., a big fraction of this data is not even relevant and simple neural networks are not powerful enough to extract the latent features. Instead of inferring general rules, they overfit the training data set and perform badly during validation. Merely presenting more images during the optimization to the network does not provide remedy. This behaviour is explained by the statistical learning theory, which characterizes a model class by a capacity measure. It turns out, that for large ratios of this capacity and the size of the given training set, further training does not improve the generalization ability of the model. Instead, regularization methods try to decrease the capacity of the model, in order to boost the network performance.

One of such strategy, namely the incorporation of prior knowledge about the learning task is the fundament of deep convolutional networks. In ConvNets for image classification and segmentation this idea is heavily exploited and appears for instance in form of adapting the network architecture. Image classification networks, e.g., take their learning task into account by subdividing the structure into two main blocks: one feature extraction part, which is followed by the classification. Among other concepts, the idea of combining the extraction of features with an invariance towards translation leads to a high performance of these networks.

The consideration of the learning task of image segmentation however, is less straightforward. This is due to the fact that segmentation suffers from a natural tension. It does not only have to extract latent features themselves but also their position. This loacilty information however, has to be discarded for a high-quality feature extraction. Figure 9.1 shows one possible solution how this problem can be tackled by inserting a third block between feature extraction and classification. This upsampling stage recovers the necessary location information. It also illustrates how different learning tasks result in different network architectures.

The chapters 7 and 8 revealed, that the design and implementation of ConvNets does not simply depend on the differentiation between image classification and segmentation but rather on the very specific learning task itself. For instance, even within the field of image segmentation, there exist numerous instances of ConvNets and the final choice depends on the particular requirements of the segmentation task.

All in all, extremely tailoring a neural network to the learning task leads to a remarkable boost in performance. Based on this idea in combination with the basic principles of machine learning, I am going to approach the research task iteratively. For this, I subdivide the development of the segmentation model into five main phases. In the first four steps of the initial iteration, a base model is designed and trained. Its quality is evaluated in the fifth

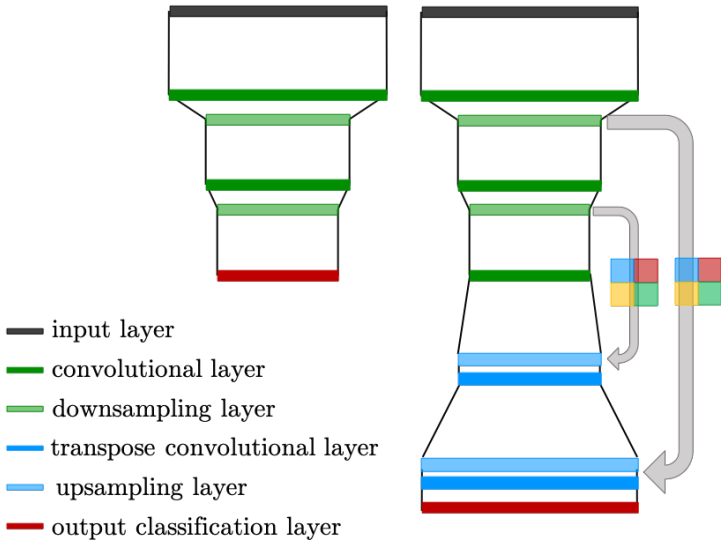

Figure 9.1: The scheme compares the architecture of a conventional convolutional network for image classification to the architecture of a fully-convolutional network with unpooling and transpose convolution stages as it can be used for image segmentation problems. The structures exhibit fundamental variations due to different demands of the learning problem.

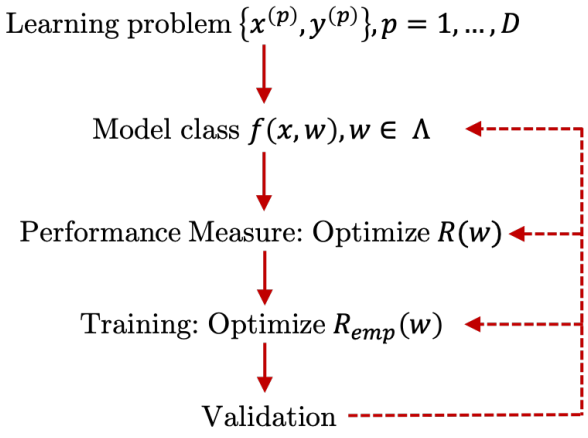

Figure 9.2: The scheme illustrates the my workflow how to approach the research question.

phase and depending on this, one of the previous steps are revisited and the quality of the new model is evaluated again. My strategy is illustrated in figure 9.2 and each of the phases with corresponding central questions, is explained in the following listing.

1. The learning problem
First of all, the learning problem has to be defined and specified. The data set is investigated and the framework of the classification and segmentation is defined. The characteristics and difficulties of the γ'-phase as stated in chapter 1 can be transferred to the context of image classification and segmentation as follows:

1. the classification needs to be sensitive towards object size
2. at the same time it can be invariant towards deformation
3. due to quality issues the data might has to be preprocessed
4. the context embedding plays a crucial role.

This already imposes requirements for the segmentation, however they have to be formulated more precisely in collaboration with material scientists. All in all, this first phase is concerned with the following questions:

- *Which classes are to be detected and which are the classification and segmentation criteria? For instance, what is the magnitude of acceptable deformations?*
- *What input images are given and do the corresponding ground truth segmentation maps exist? Is preprocessing or preparation of the data necessary? If so, what kind of preprocessing is reasonable? Is data augmentation possible and can it be exploited to meet any of the classification criteria (such as invariance towards distortion)?*
- *Which validation method is applied, i.e. how should the data set be divided into training and validation set?*

2. The model class
Based on the outcomes of phase 1, an appropriate, basic networks architecture is chosen. Chapter 8 discusses the three different research fields of image segmentation and how modifications in the network architecture can shift the focus from one of those fields to another. Knowing this might affect the decision and the reply to the question

- *Which network architecture meets the requirements of step 1 the best and can be chosen as a starting point?*

3. The performance measure
The quality of this model is quantified by the performance measure. A natural choice for classification problems is the prediction accuracy where the true label is compared with the predicted class. However, in image segmentation multiple classes appear within the same image and instead of prediction vectors, entire maps have to be compared. Hence, an adapted performance measure has to be defined:

- *How can the accuracy of a prediction map be measured?*

One possible solution which I am going to examine, is to measure the overlapping segmentation ares of the prediction map and the ground truth segmentation map. The larger the overlapping, the more accurate the segmentation.

4. The training
First thing to do in the training phase is to set the iteration method and initially guesses of the hyperparameters if necessary. Since this is an extremely working intense step, the following questions have to be considered:

- *Which possibilities to reduce the computation time do exist? Is running a parallelized algorithm on a CPU-based cluster an option? Which speed-up can be achieved by using a hardware accelerator such as a GPU? Does the latter request any adaption of the model or optimization algorithm?*

Based in section 3.5, a further interesting point to examine, is the development of the gradients during optimization:

- *How do the error gradients behave during the optimization? Is saturating an issue? If so, which effect does it have and are there possibilities to prevent it?*

5. The validation
From a computational point of view, the validation step faces the same challenges as the training. In addition to that, it shows the quality of the defined model and reveals whether overfitting occures or not:

- *Does the model overfit? If so, how to prevent this?*
- *Is a regular pattern in the missclassification or inaccurate segmentation recognizable (e.g. do errors occur in particular classes or does the model struggle with the segmentation of certain areas)? Can this be correlated to certain structural parts of the network or to steps in the optimization algorithm?*

Based on this, different measures can be taken and the previous steps are revisited, adjusted and the performance is measured again.

⇒ *Revisiting phase 1*: Possibilities to enhance regularization are
  - further data augmentation without alerting the classification criteria of the model
  - other techniques to enlarge the data set

⇒ *Revisiting phase 2*: The performance might be improve by modifying the network architecture
  - if regularization is necessary, structural regularizer can be incorporate such as adding dropout layers, adjust the number and kind of layers etc
  - if a certain error pattern in the segmentation maps occurs the basic network has to adapted more dramatically; depending on this patter, possibilities are e.g. to alert the architecture such that the focus is larger on boundary alignment or context embedding

⇒ *Revisiting phase 3*: Possibilities to enhance regularization are
  - adjust the empirical risk function by adding a regularization term as examined in section 5.1

⇒ *Revisiting phase 4*: Stronger regularization can be achieved by
  - early stopping (a regularization method that is based on steady comparison of validation and training error)

Furthermore, the validation phase is crucial to optimize the value of hyperparameters. It also reveals how strongly the performance is affected by a particular parameter.

- *How strong and in which way (e.g. convergence rate, convergence accuracy etc) does each hyperparameter affect the performance of the model? What is an efficient approach to examine this?*

Based on this stepwise stragety

Appendices

A. Optimization

The step size of the parameter update is crucial for the performance of the optimization algorithm. While SGD defines it as the product of the learning rate ϵ_τ and the norm of the gradients, the following algorithms choose more elaborate techniques.

A.1. Momentum Algorithm

Although the SGD as introduced in section 3.3 does its job and remains a popular optimization strategy [19], it is sometimes found to be too slow. Updating the parameters according to

$$\begin{aligned} v^{(\tau+1)} &= \alpha v^{(\tau)} - \epsilon \nabla_w R_{emp}^{(m)}(w^\tau) \\ w^{(\tau+1)} &= w^{(\tau)} + v^{(\tau+1)} \end{aligned} \tag{1}$$

nearly always enjoys better convergence rates on deep neural networks [49].

In this so-called *momentum* update an exponentially decaying moving average of the past m gradients is accumulated. How strongly the past gradients contribute to the updated is controlled by the hyperparameter α. A basic SGD algorithm derives the step size of the parameter update depending on the learning rate ϵ and the gradient. Opposed to that, the *momentum* approach takes into account how large and how aligned a sequence of gradients are. Consequently, the step in a particular direction the parameter space is larger if many successiv gradients point in this direction [19].

A further detailed description of the method can be found in the 1964 released paper by Polyak [40].

A.2. Adam

Although, the *momentum* algorithm is a possibility to tackle the issue of step size by enlarging it in relevant directions, it introduces a new hyperparameter α. More advanced methods adaptively tune the learning rates, and even do so per parameter. The 2015 developed ”adaptive moment” or short *Adam* algorithm reads as follows [24]

Algorithm 1 *Adam* optimization

input: step size ϵ
input: the exponential decay rates for moment estimates β_1 and β_2
input: constant α used for numerical stabilization
Initializes: $m_0 = 0, v_0 = 0, \tau = 0$

while convergence criterion not met **do**

$g_{\tau+1} = \nabla_w R_{emp}^{(m)}(w^\tau)$
$m_{\tau+1} = \beta_1 \cdot m_\tau + (1-\beta_1) \cdot g_{\tau+1}$
$v_{\tau+1} = \beta_2 \cdot v_\tau + (1-\beta_2) \cdot g_{\tau+1}^2$
$\widehat{m_{\tau+1}} = \frac{m_{\tau+1}}{1-\beta_1^{\tau+1}}$
$\widehat{v_{\tau+1}} = \frac{v_{\tau+1}}{1-\beta_2^{\tau+1}}$
$w_{\tau+1} = w_\tau - \epsilon \cdot \frac{\widehat{m_{\tau+1}}}{\sqrt{\widehat{v_{\tau+1}}}+\alpha}$

B. *Tensorflow* Implementation

B.1. The MLP MNIST Classifier

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""
Created on Thu Jan 17 01:06:46 2019

@author: franziskariegger
"""

import tensorflow as tf
from tensorflow import keras

import numpy as np
import matplotlib.pyplot as plt


from tensorflow.keras.preprocessing.image import ImageDataGenerator

print(tf.__version__)


#%%

mnist = tf.keras.datasets.mnist

(x_train, y_train), (x_test, y_test) = mnist.load_data()
x_train, x_test = x_train / 255.0, x_test / 255.0

#%%

(X_train_gen, Y_train_gen), (X_test_gen, Y_test_gen) = mnist.load_data()

X_train_gen = X_train_gen.reshape(X_train_gen.shape[0], 28, 28, 1)
X_test_gen = X_test_gen.reshape(x_test.shape[0], 28, 28, 1)

```

```
X_train_gen = X_train_gen.astype('float32')
X_test_gen = X_test_gen.astype('float32')

X_train_gen/=255
X_test_gen/=255

#%% BASE 2048 with Data augmentation

model_2048_DA = tf.keras.models.Sequential([
              tf.keras.layers.Flatten(input_shape=(28,28,1)),
              tf.keras.layers.Dense(2048, activation=tf.nn.relu),
              tf.keras.layers.Dense(2048, activation=tf.nn.relu),
              tf.keras.layers.Dense(10, activation=tf.nn.softmax)
            ])

model_2048_DA.compile(optimizer='adam',
                  loss='sparse_categorical_crossentropy',
                  metrics=['accuracy','sparse_categorical_crossentropy'])

gen = ImageDataGenerator(rotation_range=8, width_shift_range=0.08,
  ↪ shear_range=0.3,
                         height_shift_range=0.08, zoom_range=0.08)

test_gen = ImageDataGenerator()

train_generator = gen.flow(X_train_gen, Y_train_gen, batch_size=512)
test_generator = test_gen.flow(X_test_gen, Y_test_gen, batch_size=512)

model_2048_DA_history = model_2048_DA.fit_generator(train_generator,
                                                    epochs=10,
                                               validation_data=test_generator,
                                                    verbose=2)

model_2048_DA.summary()

#%% BASE 1024

model_1024 = tf.keras.models.Sequential([
  tf.keras.layers.Flatten(),
  tf.keras.layers.Dense(1024, activation=tf.nn.relu),
  tf.keras.layers.Dense(1024, activation=tf.nn.relu),
  tf.keras.layers.Dense(10, activation=tf.nn.softmax)
])

model_1024.compile(optimizer='adam',
              loss='sparse_categorical_crossentropy',
              metrics=['accuracy','sparse_categorical_crossentropy'])

model_1024_history = model_1024.fit(x_train,
                         y_train,
                         epochs=10,
                         batch_size=512,
                         validation_data=(x_test, y_test),
                         verbose=2)

model_1024.summary()
```

```
90
91  #%% BASE 2048
92
93  model_2048 = tf.keras.models.Sequential([
94    tf.keras.layers.Flatten(),
95    tf.keras.layers.Dense(2048, activation=tf.nn.relu),
96    tf.keras.layers.Dense(2048, activation=tf.nn.relu),
97    tf.keras.layers.Dense(10, activation=tf.nn.softmax)
98  ])
99
100 model_2048.compile(optimizer='adam',
101               loss='sparse_categorical_crossentropy',
102               metrics=['accuracy','sparse_categorical_crossentropy'])
103
104 model_2048_history = model_2048.fit(x_train,
105                           y_train,
106                           epochs=10,
107                           batch_size=512,
108                           validation_data=(x_test, y_test),
109                           verbose=2)
110
111 model_2048.summary()
112
113 #%% BASE 4096
114
115 model_4096 = tf.keras.models.Sequential([
116   tf.keras.layers.Flatten(),
117   tf.keras.layers.Dense(4096, activation=tf.nn.relu),
118   tf.keras.layers.Dense(4096, activation=tf.nn.relu),
119   tf.keras.layers.Dense(10, activation=tf.nn.softmax)
120 ])
121
122 model_4096.compile(optimizer='adam',
123               loss='sparse_categorical_crossentropy',
124               metrics=['accuracy','sparse_categorical_crossentropy'])
125
126 model_4096_history = model_4096.fit(x_train,
127                           y_train,
128                           epochs=10,
129                           batch_size=512,
130                           validation_data=(x_test, y_test),
131                           verbose=2)
132
133 model_4096.summary()
134
135 #%% L2-REGULARIZATION
136
137 model_l2 = tf.keras.models.Sequential([
138   tf.keras.layers.Flatten(),
139
    ↪ tf.keras.layers.Dense(2048,kernel_regularizer=keras.regularizers.l2(0.005),
    ↪ activation=tf.nn.relu),
140
    ↪ tf.keras.layers.Dense(2048,kernel_regularizer=keras.regularizers.l2(0.005),
    ↪ activation=tf.nn.relu),
141   tf.keras.layers.Dense(10, activation=tf.nn.softmax)
```

```
142 ])
143
144 model_l2.compile(optimizer='adam',
145               loss='sparse_categorical_crossentropy',
146               metrics=['accuracy','sparse_categorical_crossentropy'])
147
148 model_l2_history = model_l2.fit(x_train,
149                           y_train,
150                           epochs=10,
151                           batch_size=512,
152                           validation_data=(x_test, y_test),
153                           verbose=2)
154
155 model_l2.summary()
156
157
158 #%% L1-REGULARIZATION
159
160 model_l1 = tf.keras.models.Sequential([
161   tf.keras.layers.Flatten(),
162
      ↪ tf.keras.layers.Dense(2048,kernel_regularizer=keras.regularizers.l1(0.0005),
      ↪ activation=tf.nn.relu),
163
      ↪ tf.keras.layers.Dense(2048,kernel_regularizer=keras.regularizers.l1(0.0005),
      ↪ activation=tf.nn.relu),
164   tf.keras.layers.Dense(10, activation=tf.nn.softmax)
165 ])
166
167 model_l1.compile(optimizer='adam',
168               loss='sparse_categorical_crossentropy',
169               metrics=['accuracy','sparse_categorical_crossentropy'])
170
171 model_l1_history = model_l1.fit(x_train,
172                           y_train,
173                           epochs=10,
174                           batch_size=512,
175                           validation_data=(x_test, y_test),
176                           verbose=2)
177
178 model_l1.summary()
179
180 #%% L1-REGULARIZATION
181
182 model_l1_0_005 = tf.keras.models.Sequential([
183                    tf.keras.layers.Flatten(),
184
                       ↪ tf.keras.layers.Dense(2048,kernel_regularizer=keras.regularizers.l1(
                       ↪ activation=tf.nn.relu),
185
                       ↪ tf.keras.layers.Dense(2048,kernel_regularizer=keras.regularizers.l1(
                       ↪ activation=tf.nn.relu),
186                    tf.keras.layers.Dense(10, activation=tf.nn.softmax)
187                  ])
188
189 model_l1_0_005.compile(optimizer='adam',
```

```
              loss='sparse_categorical_crossentropy',
              metrics=['accuracy','sparse_categorical_crossentropy'])

model_l1_0_005_history = model_l1_0_005.fit(x_train,
                                            y_train,
                                            epochs=10,
                                            batch_size=512,
                                            validation_data=(x_test, y_test),
                                            verbose=2)

model_l1_0_005.summary()

#%% DROPOUT p = 0.5

model_DO = tf.keras.models.Sequential([
  tf.keras.layers.Flatten(),
  tf.keras.layers.Dense(2048, activation=tf.nn.relu),
  tf.keras.layers.Dropout(0.5),
  tf.keras.layers.Dense(2048, activation=tf.nn.relu),
  tf.keras.layers.Dropout(0.5),
  tf.keras.layers.Dense(10, activation=tf.nn.softmax)
])

model_DO.compile(optimizer='adam',
              loss='sparse_categorical_crossentropy',
              metrics=['accuracy','sparse_categorical_crossentropy'])

model_DO_history = model_DO.fit(x_train,
                          y_train,
                          epochs=10,
                          batch_size=512,
                          validation_data=(x_test, y_test),
                          verbose=2)

model_DO.summary()

#%% DROPOUT p = 0.8

model_DO_0_8 = tf.keras.models.Sequential([
  tf.keras.layers.Flatten(),
  tf.keras.layers.Dense(2048, activation=tf.nn.relu),
  tf.keras.layers.Dropout(0.8),
  tf.keras.layers.Dense(2048, activation=tf.nn.relu),
  tf.keras.layers.Dropout(0.8),
  tf.keras.layers.Dense(10, activation=tf.nn.softmax)
])

model_DO_0_8.compile(optimizer='adam',
              loss='sparse_categorical_crossentropy',
              metrics=['accuracy','sparse_categorical_crossentropy'])

model_DO_0_8_history = model_DO_0_8.fit(x_train,
                          y_train,
                          epochs=10,
                          batch_size=512,
                          validation_data=(x_test, y_test),
```

```
                          verbose=2)

model_DO_0_8.summary()

#%% DROPOUT p = 0.1

model_DO_0_1 = tf.keras.models.Sequential([
  tf.keras.layers.Flatten(),
  tf.keras.layers.Dense(2048, activation=tf.nn.relu),
  tf.keras.layers.Dropout(0.5),
  tf.keras.layers.Dense(2048, activation=tf.nn.relu),
  tf.keras.layers.Dropout(0.5),
  tf.keras.layers.Dense(10, activation=tf.nn.softmax)
])

model_DO_0_1.compile(optimizer='adam',
              loss='sparse_categorical_crossentropy',
              metrics=['accuracy','sparse_categorical_crossentropy'])

model_DO_0_1_history = model_DO_0_1.fit(x_train,
                          y_train,
                          epochs=10,
                          batch_size=512,
                          validation_data=(x_test, y_test),
                          verbose=2)

model_DO_0_1.summary()
```

B.2. The CNN MNIST Classifier

```
import tensorflow as tf
import os
import shutil
from tensorflow.contrib.learn.python.learn.datasets.mnist import
  ↪ read_data_sets
#-------------------------------------------------------------------------
#-------------------------------------------------------------------------
# ###READ DATA###
#-------------------------------------------------------------------------
#-------------------------------------------------------------------------
print ("\nImporting the MNIST data")
mnist = read_data_sets("/tmp/data/", one_hot=True)
#-------------------------------------------------------------------------
#-------------------------------------------------------------------------
# ###SET THE CNN OPTIONS###
#-------------------------------------------------------------------------
#-------------------------------------------------------------------------
n_outputs= 10
image_x  = 28
image_y  = 28
display_step = 10
training_epochs = 1000
image_shape = [-1, image_x, image_y, 1]
batch_size = 50
learning_rate = 1e-4
output_directory = 'mnist_TB_logs'
```

```
#-------------------------------------------------------------------------
#-------------------------------------------------------------------------
# ###BUILD THE CNN###
#-------------------------------------------------------------------------
#-------------------------------------------------------------------------
print('\nBuilding the CNN.')
# set placeholders
with tf.name_scope('input'):
    x = tf.placeholder(tf.float32, [None, 784], name='x-input')
    y_ = tf.placeholder(tf.float32, [None, 10], name='y-input')
#-------------------------------------------------------------------------
with tf.name_scope('input_reshape'):
    x_reshaped = tf.reshape(x, image_shape)
    tf.summary.image('input', x_reshaped, 10)
#-------------------------------------------------------------------------
with tf.name_scope('dropout'):
    keep_prob = tf.placeholder(tf.float32)
    tf.summary.scalar('dropout_keep_probability', keep_prob)
#-------------------------------------------------------------------------
# First conv+pool layer
#-------------------------------------------------------------------------
with tf.name_scope('conv1'):
    with tf.name_scope('weights'):
        W_conv1 = tf.Variable(tf.truncated_normal([5, 5, 1, 32],
          ↪ stddev=0.1))
        with tf.name_scope('summaries'):
            mean = tf.reduce_mean(W_conv1)
            tf.summary.scalar('mean', mean)
            with tf.name_scope('stddev'):
                stddev = tf.sqrt(tf.reduce_mean(tf.square(W_conv1 - mean)))
            tf.summary.scalar('stddev', stddev)
            tf.summary.scalar('max', tf.reduce_max(W_conv1))
            tf.summary.scalar('min', tf.reduce_min(W_conv1))
            tf.summary.histogram('histogram', W_conv1)

    with tf.name_scope('biases'):
        b_conv1 = tf.Variable(tf.constant(0.1, shape=[32]))
        with tf.name_scope('summaries'):
            mean = tf.reduce_mean(b_conv1)
            tf.summary.scalar('mean', mean)
            with tf.name_scope('stddev'):
                stddev = tf.sqrt(tf.reduce_mean(tf.square(b_conv1 - mean)))
            tf.summary.scalar('stddev', stddev)
            tf.summary.scalar('max', tf.reduce_max(b_conv1))
            tf.summary.scalar('min', tf.reduce_min(b_conv1))
            tf.summary.histogram('histogram', b_conv1)
    with tf.name_scope('Wx_plus_b'):
        preactivated1 = tf.nn.conv2d(x_reshaped, W_conv1,
                                     strides=[1, 1, 1, 1],
                                     padding='SAME') + b_conv1
        h_conv1 = tf.nn.relu(preactivated1)
        tf.summary.histogram('pre_activations', preactivated1)
        tf.summary.histogram('activations', h_conv1)
    with tf.name_scope('max_pool'):
        h_pool1 =  tf.nn.max_pool(h_conv1,
                                  ksize=[1, 2, 2, 1],
```

```
                                    strides=[1, 2, 2, 1],
                                    padding='SAME')
    # save output of conv layer to TensorBoard - first 16 filters
    with tf.name_scope('Image_output_conv1'):
        image = h_conv1[0:1, :, :, 0:16]
        image = tf.transpose(image, perm=[3,1,2,0])
        tf.summary.image('Image_output_conv1', image)
    # save a visual representation of weights to TensorBoard
with tf.name_scope('Visualise_weights_conv1'):
    # We concatenate the filters into one image of row size 8 images
    W_a = W_conv1                        # i.e. [5, 5, 1, 32]
    W_b = tf.split(W_a, 32, 3)           # i.e. [32, 5, 5, 1, 1]
    rows = []
    for i in range(int(32/8)):
        x1 = i*8
        x2 = (i+1)*8
        row = tf.concat(W_b[x1:x2],0)
        rows.append(row)
    W_c = tf.concat(rows, 1)
    c_shape = W_c.get_shape().as_list()
    W_d = tf.reshape(W_c, [c_shape[2], c_shape[0], c_shape[1], 1])
    tf.summary.image("Visualize_kernels_conv1", W_d, 1024)
#-------------------------------------------------------------------------
# Second conv+pool layer
#-------------------------------------------------------------------------
with tf.name_scope('conv2'):
    with tf.name_scope('weights'):
        W_conv2 = tf.Variable(tf.truncated_normal([5, 5, 32, 64],
          ↪ stddev=0.1))
        with tf.name_scope('summaries'):
            mean = tf.reduce_mean(W_conv2)
            tf.summary.scalar('mean', mean)
            with tf.name_scope('stddev'):
                stddev = tf.sqrt(tf.reduce_mean(tf.square(W_conv2 - mean)))
            tf.summary.scalar('stddev', stddev)
            tf.summary.scalar('max', tf.reduce_max(W_conv2))
            tf.summary.scalar('min', tf.reduce_min(W_conv2))
            tf.summary.histogram('histogram', W_conv2)

    with tf.name_scope('biases'):
        b_conv2 = tf.Variable(tf.constant(0.1, shape=[64]))
        with tf.name_scope('summaries'):
            mean = tf.reduce_mean(b_conv2)
            tf.summary.scalar('mean', mean)
            with tf.name_scope('stddev'):
                stddev = tf.sqrt(tf.reduce_mean(tf.square(b_conv2 - mean)))
            tf.summary.scalar('stddev', stddev)
            tf.summary.scalar('max', tf.reduce_max(b_conv2))
            tf.summary.scalar('min', tf.reduce_min(b_conv2))
            tf.summary.histogram('histogram', b_conv2)
    with tf.name_scope('Wx_plus_b'):
        preactivated2 = tf.nn.conv2d(h_pool1, W_conv2,
                                     strides=[1, 1, 1, 1],
                                     padding='SAME') + b_conv2
        h_conv2 = tf.nn.relu(preactivated2)
        tf.summary.histogram('pre_activations', preactivated2)
```

```
136             tf.summary.histogram('activations', h_conv2)
137         with tf.name_scope('max_pool'):
138             h_pool2 =  tf.nn.max_pool(h_conv2,
139                                       ksize=[1, 2, 2, 1],
140                                       strides=[1, 2, 2, 1],
141                                       padding='SAME')
142         # save output of conv layer to TensorBoard - first 16 filters
143         with tf.name_scope('Image_output_conv2'):
144             image = h_conv2[0:1, :, :, 0:16]
145             image = tf.transpose(image, perm=[3,1,2,0])
146             tf.summary.image('Image_output_conv2', image)
147         # save a visual representation of weights to TensorBoard
148     with tf.name_scope('Visualise_weights_conv2'):
149         # We concatenate the filters into one image of row size 8 images
150         W_a = W_conv2
151         W_b = tf.split(W_a, 64, 3)
152         rows = []
153         for i in range(int(64/8)):
154             x1 = i*8
155             x2 = (i+1)*8
156             row = tf.concat(W_b[x1:x2],0)
157             rows.append(row)
158         W_c = tf.concat(rows, 1)
159         c_shape = W_c.get_shape().as_list()
160         W_d = tf.reshape(W_c, [c_shape[2], c_shape[0], c_shape[1], 1])
161         tf.summary.image("Visualize_kernels_conv2", W_d, 1024)
162     #-------------------------------------------------------------------
163     # Fully connected layer
164     #-------------------------------------------------------------------
165     with tf.name_scope('Fully_Connected'):
166         W_fc1 = tf.Variable(tf.truncated_normal([7*7*64, 1024], stddev=0.1))
            ↪ #28/(2*2) = 7
167         b_fc1 = tf.Variable(tf.constant(0.1, shape=[1024]))
168         # Flatten the output of the second pool layer
169         h_pool2_flat = tf.reshape(h_pool2, [-1, 7*7*64])
170         h_fc1 = tf.nn.relu(tf.matmul(h_pool2_flat, W_fc1) + b_fc1)
171         # Dropout
172         #keep_prob = tf.placeholder(tf.float32)
173         h_fc1_drop = tf.nn.dropout(h_fc1, keep_prob=1.0)
174     #-------------------------------------------------------------------
175     # Readout layer
176     #-------------------------------------------------------------------
177     with tf.name_scope('Readout_Layer'):
178         W_fc2 = tf.Variable(tf.truncated_normal([1024, n_outputs], stddev=0.1))
179         b_fc2 = tf.Variable(tf.constant(0.1, shape=[n_outputs]))
180     # CNN output
181     with tf.name_scope('Final_matmul'):
182         y_conv = tf.matmul(h_fc1_drop, W_fc2) + b_fc2
183     #-------------------------------------------------------------------
184     # Cross entropy functions
185     #-------------------------------------------------------------------
186     with tf.name_scope('cross_entropy'):
187         diff = tf.nn.softmax_cross_entropy_with_logits(labels=y_,
188                                                        logits=y_conv)
189         with tf.name_scope('total'):
190             cross_entropy = tf.reduce_mean(diff)
```

```
191 tf.summary.scalar('cross_entropy', cross_entropy)
192 #-----------------------------------------------------------------------
193 # Optimiser
194 #-----------------------------------------------------------------------
195 with tf.name_scope('train'):
196     train_step =
      ↪ tf.train.AdamOptimizer(learning_rate).minimize(cross_entropy)
197 #-----------------------------------------------------------------------
198 # Accuracy checks
199 #-----------------------------------------------------------------------
200 with tf.name_scope('accuracy'):
201     with tf.name_scope('correct_prediction'):
202         correct_prediction = tf.equal(tf.argmax(y_conv,1),
203                                       tf.argmax(y_,1))
204     with tf.name_scope('accuracy'):
205         accuracy = tf.reduce_mean(tf.cast(correct_prediction, tf.float32))
206 tf.summary.scalar('accuracy', accuracy)
207 print('CNN successfully built.')
208 #-----------------------------------------------------------------------
209 #-----------------------------------------------------------------------
210 # ###START SESSSION AND COMMENCE TRAINING###
211 #-----------------------------------------------------------------------
212 #-----------------------------------------------------------------------
213 # create session
214 sess = tf.Session()
215 # initalise variables
216 sess.run(tf.global_variables_initializer())
217
218 # Merge all the summaries and write them out to "mnist_logs"
219 merged = tf.summary.merge_all()
220 if not os.path.exists(output_directory):
221     print('\nOutput directory does not exist - creating...')
222     os.makedirs(output_directory)
223     os.makedirs(output_directory + '/train')
224     os.makedirs(output_directory + '/test')
225     print('Output directory created.')
226 else:
227     print('\nOutput directory already exists - overwriting...')
228     shutil.rmtree(output_directory, ignore_errors=True)
229     os.makedirs(output_directory)
230     os.makedirs(output_directory + '/train')
231     os.makedirs(output_directory + '/test')
232     print('Output directory overwitten.')
233 # prepare log writers
234 train_writer = tf.summary.FileWriter(output_directory + '/train',
      ↪ sess.graph)
235 test_writer = tf.summary.FileWriter(output_directory + '/test')
236 roc_writer = tf.summary.FileWriter(output_directory)
237 # prepare checkpoint writer
238 saver = tf.train.Saver()
239 #-----------------------------------------------------------------------
240 # Train
241 #-----------------------------------------------------------------------
242 print('\nTraining phase initiated.\n')
243 for i in range(1,training_epochs+1):
244     batch_img, batch_lbl = mnist.train.next_batch(batch_size)
```

```
245     testbatch = mnist.test.next_batch(batch_size)
246 
247     # run training step
248     sess.run(train_step, feed_dict={x: batch_img,
249                                     y_: batch_lbl,
250                                     keep_prob: 1.0})
251 
252     # output the data into TensorBoard summaries every 10 steps
253     if (i)%display_step == 0:
254         train_summary, train_accuracy = sess.run([merged, accuracy],
              ↪ feed_dict={x:batch_img,
255                                                                 y_: batch_lbl,
256                                                                keep_prob: 1.0})
257         train_writer.add_summary(train_summary, i)
258         print("step %d, training accuracy %g"%(i, train_accuracy))
259 
260         test_summary, test_accuracy = sess.run([merged, accuracy],
              ↪ feed_dict={x: testbatch[0],
261                                                               y_: testbatch[1],
262                                                                keep_prob: 0.9})
263         test_writer.add_summary(test_summary, i)
264         print("test accuracy %g"%test_accuracy)
265     # output metadata every 100 epochs
266     if i % 100 == 0 or i == training_epochs:
267         print('\nAdding run metadata for epoch ' + str(i) + '\n')
268         run_options = tf.RunOptions(trace_level=tf.RunOptions.FULL_TRACE)
269         run_metadata = tf.RunMetadata()
270         summary, _ = sess.run([merged, train_step],
271                               feed_dict={x:batch_img, y_: batch_lbl,
                                    ↪ keep_prob: 1.0},
272                               options=run_options,
273                               run_metadata=run_metadata)
274         train_writer.add_run_metadata(run_metadata, 'step%03d' % i)
275         train_writer.add_summary(summary, i)
276     # save checkpoint at the end
277     if i == training_epochs:
278         print('\nSaving model at ' + str(i) + ' epochs.')
279         saver.save(sess, output_directory + "/model_at_" + str(i) +
              ↪ "_epochs.ckpt",
280                    global_step=i)
281 
282 
283 
284 # close writers
285 train_writer.close()
286 test_writer.close()
287 #-----------------------------------------------------------------------
288 # Evaluate model
289 #-----------------------------------------------------------------------
290 print('\nEvaluating final accuracy of the model (1/3)')
291 train_accuracy = sess.run(accuracy, feed_dict={x: mnist.test.images,
292                                                y_: mnist.test.labels,
293                                                keep_prob: 1.0})
294 print('Evaluating final accuracy of the model (2/3)')
295 test_accuracy = sess.run(accuracy, feed_dict={x: mnist.train.images,
296                                               y_: mnist.train.labels,
```

```
                                                keep_prob: 1.0})
print('Evaluating final accuracy of the model (3/3)')
val_accuracy  = sess.run(accuracy, feed_dict={x: mnist.validation.images,
                                                y_: mnist.validation.labels,
                                                keep_prob: 1.0})
#-----------------------------------------------------------------------
#-----------------------------------------------------------------------
# Output results
#-----------------------------------------------------------------------
#-----------------------------------------------------------------------
print ("\nTraining phase finished")
print ("\tAccuracy for the train set examples      = " , train_accuracy)
print ("\tAccuracy for the test set examples       = " , test_accuracy)
print ("\tAccuracy for the validation set examples = " , val_accuracy)

#print a statement to indicate where to find TensorBoard logs
print('\nRun "tensorboard --logdir=' + output_directory + '" to see results
  ↪ on localhost:6006')
#-----------------------------------------------------------------------
#-----------------------------------------------------------------------
#-----------------------------------------------------------------------
```

Bibliography

[1] dreamstimes. https://www.dreamstime.com/photos-images/abstract-composition-hands-eyes.html. Accessed: 2018-12-13.

[2] Convolutional neural networks for image and video processing. https://wiki.tum.de/display/lfdv, . Accessed: 2018-01-22.

[3] Nestor, artificial intelligence. https://nestor-ai.com/#, . Accessed: 2018-01-22.

[4] Satellite imaging corporation. https://www.satimagingcorp.com, . Accessed: 2018-01-22.

[5] Listerine - feel every smile. https://jwt.co.uk/work/feel-every-smile, . Accessed: 2018-01-22.

[6] N Andrey. Kolmogorov. on the representation of continuous functions of many variables by superposition of continuous functions of one variable and addition. *Amer. Math. Soc. Transl*, 28:55–59, 1963.

[7] Ignacio Arganda-Carreras, Srinivas C Turaga, Daniel R Berger, Dan Ciresan, Alessandro Giusti, Luca Maria Gambardella, Jürgen Schmidhuber, Dmitry Laptev, Sarvesh Dwivedi, Joachim M Buhmann, Ting Liu, Mojtaba Seyedhosseini, Tolga Tasdizen, Lee Kamentsky, Radim Burget, Vaclav Uher, Xiao Tan, Cangming Sun, Tuan Pham, Erhan Bas, Mustafa Gokhan Uzunbas, Albert Cardona, Johannes Schindelin, and H. Sebastian Seung. Crowdsourcing the creation of image segmentation algorithms for connectomics. *Frontiers in Neuroanatomy*, 9(142), 2015. ISSN 1662-5129. doi: 10.3389/fnana.2015.00142. URL http://www.frontiersin.org/neuroanatomy/10.3389/fnana.2015.00142/abstract.

[8] Sanjeev Arora, Aditya Bhaskara, Rong Ge, and Tengyu Ma. Provable bounds for learning some deep representations. In *International Conference on Machine Learning*, pages 584–592, 2014.

[9] Vijay Badrinarayanan, Alex Kendall, and Roberto Cipolla. Segnet: A deep convolutional encoder-decoder architecture for image segmentation. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 39(12):2481–2495, Dec 2017. ISSN 2160-9292. doi: 10.1109/tpami.2016.2644615. URL http://dx.doi.org/10.1109/TPAMI.2016.2644615.

[10] Yoshua Bengio, Aaron Courville, and Pascal Vincent. Representation learning: A review and new perspectives. *IEEE transactions on pattern analysis and machine intelligence*, 35(8):1798–1828, 2013.

[11] C.M. Bishop. *Pattern Recognition and Machine Learning*. Information Science and Statistics. Springer New York, 2016. ISBN 9781493938438.

[12] Liang-Chieh Chen, George Papandreou, Iasonas Kokkinos, Kevin Murphy, and Alan L. Yuille. Semantic image segmentation with deep convolutional nets and fully connected crfs, 2014.

[13] Liang-Chieh Chen, George Papandreou, Florian Schroff, and Hartwig Adam. Rethinking atrous convolution for semantic image segmentation. *arXiv preprint arXiv:1706.05587*, 2017.

[14] Liang-Chieh Chen, George Papandreou, Iasonas Kokkinos, Kevin Murphy, and Alan L. Yuille. Deeplab: Semantic image segmentation with deep convolutional nets, atrous convolution, and fully connected crfs. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 40(4):834–848, Apr 2018. ISSN 2160-9292. doi: 10.1109/tpami.2017.2699184. URL http://dx.doi.org/10.1109/TPAMI.2017.2699184.

[15] Francois Chollet. *Deep Learning with Python*. Manning Publications Co., Greenwich, CT, USA, 1st edition, 2017. ISBN 1617294438, 9781617294433.

[16] Vincent Dumoulin and Francesco Visin. A guide to convolution arithmetic for deep learning. *CoRR*, abs/1603.07285, 2016. URL http://dblp.uni-trier.de/db/journals/corr/corr1603.html#DumoulinV16.

[17] Clement Farabet, Camille Couprie, Laurent Najman, and Yann LeCun. Learning hierarchical features for scene labeling. *IEEE transactions on pattern analysis and machine intelligence*, 35(8):1915–1929, 2013.

[18] Kunihiko Fukushima. Neocognitron: A self-organizing neural network model for a mechanism of pattern recognition unaffected by shift in position. *Biological Cybernetics*, 36(4):193–202, Apr 1980. ISSN 1432-0770. doi: 10.1007/BF00344251. URL https://doi.org/10.1007/BF00344251.

[19] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. http://www.deeplearningbook.org.

[20] Ian J Goodfellow, David Warde-Farley, Mehdi Mirza, Aaron Courville, and Yoshua Bengio. Maxout networks. *arXiv preprint arXiv:1302.4389*, 2013.

[21] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Spatial pyramid pooling in deep convolutional networks for visual recognition. In *European conference on computer vision*, pages 346–361. Springer, 2014.

[22] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.

[23] David H Hubel and Torsten N Wiesel. Receptive fields, binocular interaction and functional architecture in the cat's visual cortex. *The Journal of physiology*, 160(1):106–154, 1962.

[24] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization, 2014.

[25] Risi Kondor and Shubhendu Trivedi. On the Generalization of Equivariance and Convolution in Neural Networks to the Action of Compact Groups. *arXiv e-prints*, art. arXiv:1802.03690, February 2018.

[26] Philipp Krähenbühl and Vladlen Koltun. Efficient inference in fully connected crfs with gaussian edge potentials. In *Advances in neural information processing systems*, pages 109–117, 2011.

[27] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems 25*, pages 1097–1105. Curran Associates, Inc., 2012. URL http://papers.nips.cc/paper/4824-imagenet-classification-with-deep-convolutional-neural-networks.pdf.

[28] Yann LeCun, Bernhard Boser, John S Denker, Donnie Henderson, Richard E Howard, Wayne Hubbard, and Lawrence D Jackel. Backpropagation applied to handwritten zip code recognition. *Neural computation*, 1(4):541–551, 1989.

[29] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

[30] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. Deep learning. *nature*, 521(7553): 436, 2015.

[31] Yann LeCun et al. Generalization and network design strategies. *Connectionism in perspective*, pages 143–155, 1989.

[32] Karel Lenc and Andrea Vedaldi. Understanding image representations by measuring their equivariance and equivalence. *CoRR*, abs/1411.5908, 2014. URL http://arxiv.org/abs/1411.5908.

[33] Seung-Hwan Lim, Steven R Young, and Robert M Patton. An analysis of image storage systems for scalable training of deep neural networks. *system*, 5(7):11, 2016.

[34] Min Lin, Qiang Chen, and Shuicheng Yan. Network in network. *arXiv preprint arXiv:1312.4400*, 2013.

[35] Warren S McCulloch and Walter Pitts. A logical calculus of the ideas immanent in nervous activity. *The bulletin of mathematical biophysics*, 5(4):115–133, 1943.

[36] Tom M Mitchell. Does machine learning really work? *AI magazine*, 18(3):11, 1997.

[37] Mohammadreza Mostajabi, Payman Yadollahpour, and Gregory Shakhnarovich. Feedforward semantic segmentation with zoom-out features. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 3376–3385, 2015.

[38] Hyeonwoo Noh, Seunghoon Hong, and Bohyung Han. Learning deconvolution network for semantic segmentation. In *Proceedings of the IEEE international conference on computer vision*, pages 1520–1528, 2015.

[39] Chao Peng, Xiangyu Zhang, Gang Yu, Guiming Luo, and Jian Sun. Large kernel matters — improve semantic segmentation by global convolutional network. *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, Jul 2017. doi: 10.1109/cvpr.2017.189. URL http://dx.doi.org/10.1109/CVPR.2017.189.

[40] Boris T Polyak. Some methods of speeding up the convergence of iteration methods. *USSR Computational Mathematics and Mathematical Physics*, 4(5):1–17, 1964.

[41] William H. Press, Saul A. Teukolsky, William T. Vetterling, and Brian P. Flannery. *Numerical Recipes 3rd Edition: The Art of Scientific Computing*. Cambridge University Press, New York, NY, USA, 3 edition, 2007. ISBN 0521880688, 9780521880688.

[42] Klaus Prof. Dr. Obermayer. Machine intelligence 1. *TU Berlin, Institute of Software Engineering and Theoretical Computer Science*, March, 2018.

[43] Olaf Ronneberger, Philipp Fischer, and Thomas Brox. U-net: Convolutional networks for biomedical image segmentation. In *International Conference on Medical image computing and computer-assisted intervention*, pages 234–241. Springer, 2015.

[44] Dominik Scherer, Andreas Müller, and Sven Behnke. Evaluation of pooling operations in convolutional architectures for object recognition. In *Artificial Neural Networks–ICANN 2010*, pages 92–101. Springer, 2010.

[45] Jürgen Schmidhuber. Deep learning in neural networks: An overview. *Neural networks*, 61:85–117, 2015.

[46] Evan Shelhamer, Jonathan Long, and Trevor Darrell. Fully convolutional networks for semantic segmentation. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 39(4):640–651, Apr 2017. ISSN 2160-9292. doi: 10.1109/tpami.2016.2572683. URL http://dx.doi.org/10.1109/TPAMI.2016.2572683.

[47] Lakshmanaprabu S.K., Sachi Nandan Mohanty, Shankar K., Arunkumar N., and Gustavo Ramirez. Optimal deep learning model for classification of lung cancer on ct images. *Future Generation Computer Systems*, 92:374 – 382, 2019. ISSN 0167-739X. URL http://www.sciencedirect.com/science/article/pii/S0167739X18317011.

[48] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: a simple way to prevent neural networks from overfitting. *The Journal of Machine Learning Research*, 15(1):1929–1958, 2014.

[49] Standford. Convolutional neural networks for visual recognition. http://cs231n.stanford.edu/. Accessed: 2018-10-13.

[50] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. Going deeper with convolutions. In *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2015.

[51] Yichuan Tang and Chris Eliasmith. Deep networks for robust visual recognition. In *Proceedings of the 27th International Conference on Machine Learning (ICML-10)*, pages 1055–1062. Citeseer, 2010.

[52] Tensorflow. Graphs and sessions. https://www.tensorflow.org/tutorials/keras/overfit_and_underfit, . Accessed: 2019-01-16.

[53] Tensorflow. Explore overfitting and underfitting. https://www.tensorflow.org/guide/graphs, . Accessed: 2019-01-17.

[54] Vladimir Naumovich Vapnik. An overview of statistical learning theory. *IEEE transactions on neural networks*, 10(5):988–999, 1999.

[55] Juyang Weng, Narendra Ahuja, and Thomas S Huang. Cresceptron: a self-organizing neural network which grows adaptively. In *Neural Networks, 1992. IJCNN., International Joint Conference on*, volume 1, pages 576–581. IEEE, 1992.

[56] David H Wolpert. The lack of a priori distinctions between learning algorithms. *Neural computation*, 8(7):1341–1390, 1996.

[57] Fisher Yu and Vladlen Koltun. Multi-scale context aggregation by dilated convolutions. *arXiv preprint arXiv:1511.07122*, 2015.

[58] Matthew D Zeiler and Rob Fergus. Visualizing and Understanding Convolutional Networks. *arXiv e-prints*, art. arXiv:1311.2901, November 2013.

[59] Hengshuang Zhao, Jianping Shi, Xiaojuan Qi, Xiaogang Wang, and Jiaya Jia. Pyramid scene parsing network. *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, Jul 2017. doi: 10.1109/cvpr.2017.660. URL http://dx.doi.org/10.1109/CVPR.2017.660.