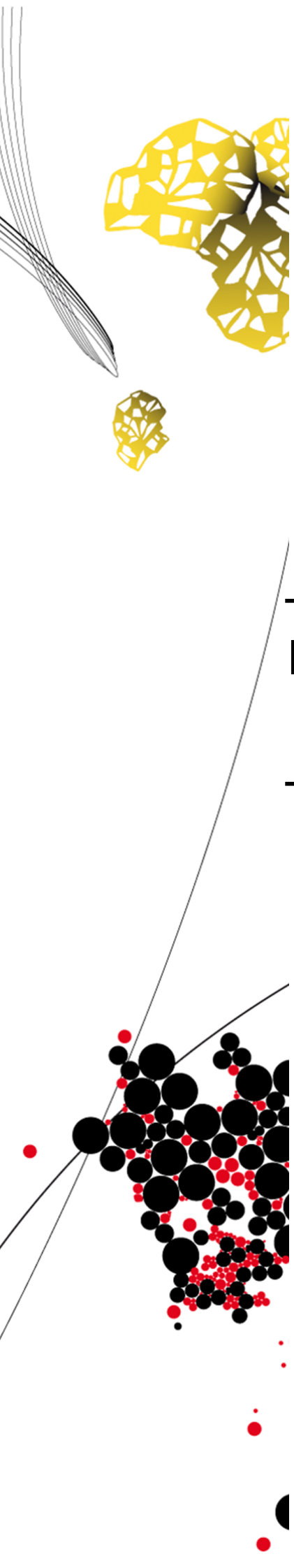

UNIVERSITY OF TWENTE.

Faculty of Electrical Engineering, Mathematics & Computer Science

Insights into deep learning methods with application to cancer imaging

Niek Huttinga

M.Sc. Thesis
May 2017

Assessment committee:
Prof. Dr. Stephan van Gils
Dr. Christoph Brune
Dr. Pranab Mandal

Daily supervisor:
Dr. Christoph Brune

Applied Analysis group
Department of Applied Mathematics
Faculty of Electrical Engineering,
Mathematics and Computer Science
University of Twente
P.O. Box 217
7500 AE Enschede
The Netherlands

Contents

Abstract

In recent years, deep learning methods have received a massively increasing amount of attention due to their almost human-level performance on high-level vision tasks like semantic segmentation, object classification and automatic image captioning. These methods have also shown to be competitive with current state-of-the-art methods for low-level vision tasks like denoising and inpainting. Despite their impressive performance, deep learning methods still lack a unified theoretical foundation and understanding. Low-level vision tasks can, on the other hand, also be addressed by methods that do have a very strong theoretical foundation. In this work we explore similarities between the two approaches for low-level vision task to gain more insight into deep learning methods in general. We explain the performance of convolutional neural networks (CNNs), and show that denoising auto-encoders effectively solve a bi-level optimization problem. On top of this, we apply the insights into CNNs to images of circulating tumor cells (CTCs). More specifically, a CNN is trained to perform classification of CTCs based on high-throughput fluorescence microscopy images obtained from the medical cell biophysics group (MCBP) at the University of Twente.

Keywords: deep learning, variational methods, convolutional neural networks, auto-encoders, skip-connections, bi-level optimization, regularization, denoising, circulating tumor cells.

Acknowledgements

This thesis is the result of my graduation project and I really enjoyed working on it. It was a combination of computer science and mathematics, which are two subjects that I really like. In this project, I was able to combine my interest in technological developments with my background in mathematics and it was very nice that this was made possible as a graduation project. I would like to express my graduate to a few people who have made my time working on this project enjoyable.

First of all, I would like to thank Dr. Christoph Brune for the great amount of effort he has put in my supervision during the project. His enthusiasm and open attitude made me always look forward to our meetings. We always had very fruitful and inspiring discussions that helped me to proceed with new ideas and directions afterwards. Next to this, I would like to thank him for making such an open project possible. He gave the right amount of guidance when needed, while still allowing creativity. Although it was sometimes hard to figure out in which direction the research was going, it is safe to say that without his guidance the result would not have been the same.

Next, I would like to express my gratitude to Prof. Dr. Stephan van Gils. He always had his door open for a friendly chat, and has always taken time to help if needed. Besides this, I would like to thank him, and the other members of the Applied Analysis reading club, for the discussions we had every Tuesday afternoon. I was always looking forward to it and have enjoyed the discussions.

I would like to thank Leonie Zeune for her effort regarding the tumor cell application and theoretical discussions. Leonie provided all the tumor cell datasets and knowledge to really understand the difficulties and get the best results out of it. We have also had several good discussion about the theory that helped me a lot.

Finally, I would like to express my gratitude to my family and friends for their support and interest; without this it would have been a lot harder to stay motivated. I would like to thank my parents for the support and freedom they give me in whatever I do. I thank Manu Kalia for the many fruitful theoretical discussions, good feedback and Indian jokes. I thank Yoeri Boink for the discussions, and Normen Oude Booijink for the discussions and, most importantly, distraction from my research with beers and music in the evenings. Especially I would like to thank Evelien for always cheering me up with her positive attitude when needed, her listening ears, and for always being there for me.

Chapter 1

Introduction

1.1 Motivation

Low-level vision tasks have been dominated by variational methods for the last few decades, but have recently been outperformed by deep learning methods on various tasks of this type. In contrast to variational methods, deep learning methods lack thorough understanding and a theoretical foundation. In this project we aim at exploring similarities between variational methods for low-level vision tasks and deep learning methods, in order to improve the former and gain more insight into the latter.

Ever since their introduction, neural networks (NNs) have received an increasing amount of attention. In short, an NN is a system built up of a composition of so-called *layers*. Each layer performs a linear transformation of its input, followed up by a nonlinearity called *the activation function*. The parallel uprise of computer vision lead to an extension of the neural net to visual pattern recognition in the form of the so-called *convolutional neural network* (CNN): a system designed specifically for visual information extraction by mimicking the processing behavior of the mammalian visual cortex.

In recent years the combination of GPU-computing, max-pooling, back-propagation and large CNNs has lead to several contest-winning, e.g. [1], [2], and superhuman-performing systems, e.g. [3]. We refer the interested reader to [4] for an extensive overview. A key ingredient in those success stories was the use of very *deep neural networks*. Here the concept of deep refers to the number of successive layers in the network architecture. The general view in literature is that deeper networks can perform more complex pattern recognition than their shallow counterparts. One of the reason for this, as argued in [5], is that those deep neural networks are able to learn more levels of abstraction. The shallow layers of a deep CNN trained for visual recognition learn low-level features like edges, whereas the deeper layers learn more semantical concepts like faces by combining lower-level features. In other words, deep CNNs perform hierarchical feature extraction: by increasing the depth one can increase the level of abstraction learned by the network. To get an idea of what is going on inside a deep CNN, we have visualized a CNN trained for speed sign recognition in Fig. 1.1. It consists of two main components: feature extraction and classification. Using the combination of filtering and subsampling, informative features are extracted and finally used for classification in the last part of the network. The

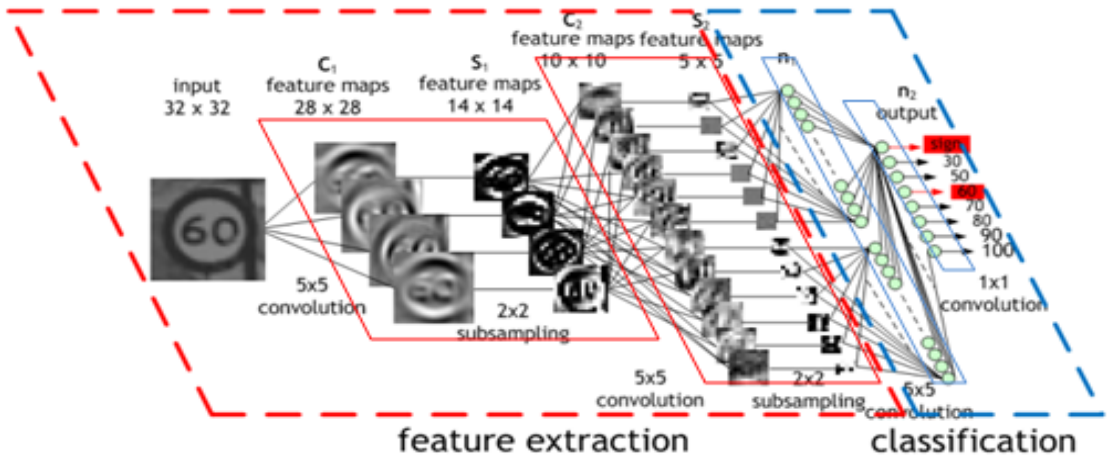

Figure 1.1: An example of a deep CNN trained for speed sign recognition. As input we have a 32×32 image of a traffic sign and as output the detected speed limit. The traffic sign image is filtered through a convolutional layer with 4 5×5 kernels, yielding 4 feature maps of size 28×28. The result is subsampled through a 2×2 max-pooling layer. After this the image is passed through a convolutional layer with 12 5×5 kernels, yielding 12 features maps of size 10×10. The result is again subsampled through a 2×2 max-pooling layer. The results after subsampling or filtering with a specific kernel are shown as images in the layers. After feature extraction, the network finally performs classification on the extracted features, yielding a predicted probability for the presence of the speed sign in the input image. Image by Maurice Peemen.

results after the different layers in the feature extraction clearly show that the abstraction of the extracted features increases with depth.

Despite the increased capacity, the deep structure of these networks brings about two serious problems: it vastly increases the amount of parameters, and it leads to so-called *vanishing gradients*. The first problem easily leads to overfitting, while the second often troubles the training in lower layers significantly. Due to these problems, training deep networks became so challenging, see e.g. [6], yet still very profitable, that it evolved into an active field of research on its own: *Deep learning* (DL). Next to those training issues, there is still a lack of understanding on what exactly happens inside the deep neural networks. For example, the following questions are still open in deep learning:

- *Which architecture is optimal for which task?*
- *How are deep networks able to generalize so well?*
- *How are deep networks related to other methods developed for similar tasks?*
- *What exactly do deep networks learn?*

As deep learning methods quickly approach (or, on some tasks, have already surpassed) human-level performance, it is indeed to be expected that they will soon be involved in making important decisions. Since this brings about increasing responsibility, it is of no surprise that the interest in understanding deep learning methods has also increased over the years.

Variational methods have been around for a long time and can be used to solve image restoration problems. These methods are model-based, meaning that they do not directly take the structure in the data into account but make assumptions on it through a model of the data. Deep learning methods are, on the other hand, data-driven and learn directly from the data, but lack theoretical background. Connecting variational methods with deep learning would therefore allow to take the best of both world, and improve understanding of the latter. For this reason, we investigate this connection between variational methods and deep learning.

1.2 Thesis outline

In this thesis we review and contribute to the current understanding of (deep) neural networks. First, we give a mathematical view on neural networks and give an intuitive explanation of their performance. Next, we consider variational denoising methods and denoising neural networks and explore the relations between them. Since the variational methods have a strong theoretical foundation we can use these relations to gain insights on neural networks. Finally, we train a deep convolutional neural network for the classification of circulating tumor cells to show the strength of deep learning. The thesis is organized as follows.

In Chapter 2 we present background information on variational methods for denoising. We discuss the transition from inverse problem to variational method and show how diffusion processes naturally arise in this setting. Next we discuss how popular (nonlocal) neigborhood filtering methods such as the Yaroslavsky's filter [7], the bilateral filter [8] and nonlocal means (NLM) [9] can be interpreted as a generalized diffusion process.

In Chapter 3 we develop the general interpretation of neural networks as a composition of a feature extractor and a task-specific module. We first introduce the general concept of a neural network and give an overview on the different aspects and difficulties of the training process. Subsequently we discuss supervised convolutional neural networks and unsupervised auto-encoders (AEs) in more detail. We show that in CNNs trained for classification the task-specific module is a simple classifier and that in AEs it is a reconstruction function. After this, we discuss the desired properties of the feature extractor in both cases, and explain the performance of neural networks by showing how those desired properties are realized with the different architectures.

Motivated by the learnable diffusion process introduced in [10] and the similarities with denoising auto-encoders, we explore in Chapter 4 the relation between auto-encoders and regularization methods. We follow the work in [11]–[13] to show that the reconstruction in a tied-weight denoising auto-encoder can be interpreted as one gradient descent step on a specific energy functional. Next, we extend this work to show another direct but more natural relation between similar gradient flows and tied-weight auto-encoders with a skip-connection. These results show that the forward pass of these types of auto-encoders effectively minimizes an energy. Since auto-encoders simultaneously solve another optimization problem over their parameters during the backward pass, this allows us to interpret auto-encoders as an alternating scheme to solve a bi-level optimization problem. We conclude that auto-encoders combine data-fitting with learning optimal regularization from structure in the data. We apply these results to show the connection between variational networks as introduced in [10] and convolutional denoising

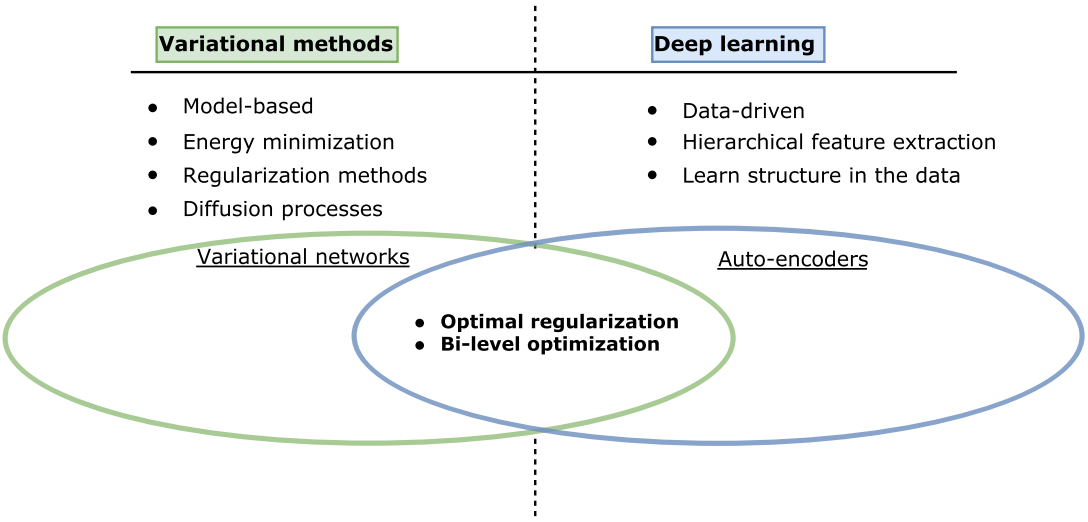

Figure 1.2: An overview of the work in this chapter. We look at the connections between the model-based variational methods and data-driven deep learning by linking variational networks to auto-encoders. Using denoising auto-encoders and skip-connection we show these methods are related through bi-level optimization and optimal regularization.

auto-encoders with a skip-connection. Finally, we apply this connection to review the VN and identify points of improvement. The work in this chapter is summarized in Fig. 1.2

In Chapter 5 we present a biomedical application of deep learning. We apply the tools from Chapter 3 to train a deep convolutional neural network capable of classifying circulating tumor cells in thumbnails obtained with fluorescence microscopy. This is a clear example that justifies the investigation towards the understanding of the networks; if we do not know what these networks do, then how can we trust the networks to decide whether or not cancer is present?

Chapter 2

Background on variational methods for denoising

In this chapter we present a theoretical background on variational methods. First, we will discuss the transition from an inverse problem to a variational method. Next, we focus on denoising and show how diffusion processes naturally arise from variational methods in this context. Finally, we generalize to nonlocal methods for denoising and show that those can be interpreted as a generalized diffusion process.

2.1 From inverse problems to variational methods

Let us consider the equation

$$A(u) = f \tag{2.1}$$

In this equation $A : X \to Y$ is an operator acting on vector spaces X and Y, usually called the *forward operator*. The forward operator maps the desired solution vector $u \in X$ to the data vector $f \in Y$. For ease of notation, we write (2.1) simply as $Au = f$. Note that this is purely for notational convience, the operator A may still be nonlinear.

In this work we will consider inverse problems arising in imaging, hence u and f are both images of some modality. We mathematically define an image by the function $u : \Omega \to [0, 255]$. Here $\Omega \subseteq \mathbb{R}^d$ is the image domain and d is the dimension (for now we take $d = 2$). Solving (2.1) for u is called the *inverse problem*; we try to find a way to 'invert' the actions of the forward operator A. If A is an invertible linear operator, we can simply obtain the solution with the matrix inverse as $\hat{u} = A^{-1}f$. In general, however, solving the inverse problem is much more challenging. Consider for instance the following cases: the inverse of A is not defined or f is not completely available or corrupted by noise. Instead of solving this problem directly we can try to approximate a solution by finding a vector $u \in X$ for which the difference between the left-hand side and right-hand side is as small as possible. For this we try to minimize the so-called data-fidelity functional $D : Y \times Y \to \mathbb{R}$. A common choice for the data-fidelity is the squared L^2-norm of the difference, i.e. $D(Au, f) := \|Au - f\|^2$. Minimizing just the data-fidelity will in general not give good results because with severe noise the solution procedure is likely to blow up. To get a good approximation of the solution we have to guide the optimization process by

making a-priori assumptions on the solution. One way to do this is to add a regularization functional $J : X \to \mathbb{R}$ to the objective function. This regularization term will favor solutions that fulfill the assumptions by penalizing solutions that do not. Hence, we approximate a solution by minimizing the energy

$$E(u) = \mu D\left(Au, f\right) + J(u), \tag{2.2}$$

for $\mu \geq 0$. Here $\mu \in \mathbb{R}$ is called the *regularization parameter* and determines the strength of the a-priori assumptions. In fact, it can be shown that with f corrupted by additive Gaussian noise, a Bayesian approach to calculate the MAP-estimate yields a similar optimization problem. From this it can be seen that the regularization term corresponds to the negative logarithm of the prior probability, and therefore models a-priori assumptions on the data.

Minimizing energies like (2.2) in order to solve (2.1) is what is called a variational method for an inverse problem. Since the type of regularization corresponds to the constraints on the solution, the choice for the regularization determines the type and quality of the solution. In case of an L^2-norm data-fidelity and a general convex regularization functional the resulting energy functional is convex. Therefore the optimality condition of (2.2) reads

$$0 = \partial_u E(u) = \mu A^*(Au - f) + \partial_u J(u). \tag{2.3}$$

We obtain the corresponding gradient flow equation as

$$\frac{\partial u}{\partial t} = -\mu A^*(Au - f) - \partial_u J(u), \quad u_0 = f, \tag{2.4}$$

which fulfills (2.3) in a limiting stationary solution.

2.2 Local diffusion equations for denoising

An example of an inverse problem in the form of (2.1) is *denoising*. Here we try to find the originally clean image from a noisy observation, see Fig. 2.1 for an example. The denoising problem can now be written formally in the form of (2.1) by taking $A = I$ and solving

$$u = f^\delta = f + \delta \tag{2.5}$$

for u. Here f^δ is the vector f corrupted by additive Gaussian noise, modeled by the Gaussian random variable δ. Because of the presence of noise, we do not want a solution that satisfies (2.5) exactly. Since we know that the clean image and its noisy counterpart should still be relatively close in norm, we follow the same approximation procedure as above: we minimize the energy

$$E(u) = \mu\|u - f\|^2 + J(u). \tag{2.6}$$

Since plain L^p spaces cannot separate signal from noise very well, special type of regularizations are added such that the solution will live in a different, more suitable space. A possible choice is to look for solutions in the Sobolev space, since these spaces involve the norm of the gradients. A regularization for denoising that will give a solution in the Sobolev space $W^{1,2}(\Omega)$ penalizes the squared norm of the gradient. In this space noisy signals have a significantly larger norm, hence it is a convenient space to use for denoising tasks. Intuitively this will result in a minimal amount of jumps in the intensity and therefore remove the noise. An example is the functional

$$J(u) = \frac{1}{2}\|\nabla u\|^2 = \frac{1}{2}\int_\Omega |\nabla u|^2 dx, \tag{2.7}$$

Figure 2.1: A visualization of the denoising task. Left we see a clean image and right the same image corrupted by additive Gaussian noise with $\mu = 0$ and $\sigma = 25$. The inverse problem of denoising now aims to recover the clean image in the left given the noisy observation on the right.

where $\nabla : X \to L^2(\Omega)^2$ is the gradient operator and $|\cdot|$ is the standard ℓ_2-norm. Note that this formally only makes sense for $u \in X = H^1(\Omega)$. Convexity is still preserved, hence the optimality condition for minimization of (2.6) with this functional reads

$$0 = \mu(u - f) + \nabla^* \nabla u.$$

Using the fact that $\nabla^* = -\nabla\cdot$, this becomes

$$\Delta u = \mu(u - f) \in L^2(\Omega).$$

Since we now have $\Delta u \in L^2(\Omega)$ this will yield solutions $u \in H^2(\Omega)$, hence the solution is oversmoothed. Another way to see this is to consider the corresponding gradient flow equation

$$\frac{\partial u}{\partial t} = \Delta u + \mu(f - u), \quad u_0 = f.$$

Since this is the heat equation with a reaction term, we expect the solution to be smoothed over time. Hence, while we minimize the energy (2.7) by evolving the solution through the gradient descent equation we more and more smooth the solution on the way. If we instead penalize the L^1-norm of the gradient, we get what is called the total variation (TV) regularizer [14]:

$$J(u) = TV(u) := \|\nabla u\|_{L^1}.$$

Formally for $\nabla u \neq 0$, the gradient becomes

$$\partial_u TV(u) = -\nabla \cdot \left(\frac{\nabla u}{|\nabla u|} \right).$$

Note that this definition of the TV only makes sense for $u \in W^{1,1}(\Omega)$. Since this space has some drawbacks, a more formal definition is used in practice, but we omit the details and simply assume $u \in W^{1,1}(\Omega)$. The gradient flow equation in case of the TV regularizer reads

$$\frac{\partial u}{\partial t} = \nabla \cdot \left(\frac{\nabla u}{|\nabla u|} \right) + \mu(f - u), \quad u_0 = f.$$

We see that with this choice of the regularization functional the diffusion coefficient is not constant but inversely proportional to the norm of the gradient, i.e. we have obtained a nonlinear diffusion equation. This type of nonlinear diffusion results in heavy smoothing in constant regions (low gradients), but limited smoothing over edges (large gradients). A generalization of this approach is to consider a regularization functional of the form

$$J(u) = \frac{1}{2}\int_\Omega g\left(|\nabla u|^2\right) dx, \tag{2.8}$$

for $g : \mathbb{R} \to \mathbb{R}$. Using calculus of variations we obtain the corresponding gradient descent equation as

$$\frac{\partial u}{\partial t} = \nabla \cdot \left(g'\left(|\nabla u|^2\right)\nabla u\right) + \mu(f-u), \quad u_0 = f. \tag{2.9}$$

Defining $c\left(|\nabla u|^2\right) := g'\left(|\nabla u|^2\right)$ we obtain exactly an anisotropic reaction-diffusion equation as proposed by Perona and Malik [15]. Note that since $g(\cdot)$ is not necessarily a convex function this possibly results in a non-convex energy (2.6). For this reason the gradient flow is not guaranteed to converge to the optimal solution.

From the discussion above we see that the a-priori assumptions on the solution incorporated in the regularization directly relate to the diffusion coefficient in the corresponding diffusion process. We will later see how this observation can help to understand the functioning of a type of neural network called auto-encoders.

2.3 Nonlocal methods for denoising

Although the denoising procedures discussed in the previous section have shown good results, other methods have been proposed that outperform them significantly. Most of these methods exploit the *nonlocal self-similarity* property of natural images. For example, let us consider the denoising of an image containing a repetitive pattern. The local methods will not use the repetitive structure in the image, but it is not too hard to imagine that a denoising algorithm can benefit from this information. If it can identify nonlocal similarities between image patches, then those patches can be processed in a similar way.

Popular nonlocal approaches to denoising include methods based on patch-distances, which are consequently called neighborhood filters. Examples are Yaroslavsky's filter [7], bilateral filter [8] and nonlocal means (NLM) [9].

In this section we will introduce the basics of neighborhood filtering methods. In general these filters perform a filtering such that the denoised version u of the noisy image f at position $x \in \Omega$ is given by

$$u(x) = \frac{1}{C(x)}\int_\Omega K(x,y,f)f(y)dy, \quad \text{where } C(x) = \int_\Omega K(x,y,f)dy. \tag{2.10}$$

The kernel $K(x,y,f)$ used, will eventually determine the type of the resulting filter.

In [16], Kindermann-Osher-Jones showed how neighborhood filters can be related to variational methods. As we will see in the next section, an extension of this work allows us to formally

relate nonlocal methods such as NLM and the Yaroslavsky filter to diffusion equations on the image. The authors introduce the following general functional

$$J_{KOJ}(u) := \int_{\Omega\times\Omega} g\left(\frac{d(u(x),u(y))}{h^2}\right) w\left(|x-y|\right) dxdy, \tag{2.11}$$

where $d(u(x),u(y))$ is a distance measure. Moreover, $w(|x-y|)$ is a symmetric window and $g(\cdot)$ determines the type of regularization. The Yaroslavsky and NLM functional are obtained for $g(z) = 1-\exp(-z)$, and distance measures

$$d_{Yar}(u(x),u(y)) = |u(x)-u(y)|^2$$

and

$$d_{NLM}(u(x),u(y)) = \int_{\Omega} G_a(t)|u(x+t)-u(y+t)|^2 dt.$$

In the last equation G_a is a Gaussian with standard deviation a and mean 0. From these functionals it is easily concluded that NLM is a generalization of the Yaroslavsky filter. The problem with functionals of the form (2.10) is that they are in general not convex. To overcome this, Gilboa and Osher [17] changed the roles of the functions $g(\cdot)$ and $w(\cdot)$ slightly in order to obtain a similar but quadratic functional. Instead of J_{KOJ}, as introduced above, the authors proposed the following general nonlocal functional:

$$J_{GO}(u) := \frac{1}{4}\int\limits_{\Omega\times\Omega} (u(x)-u(y))^2 w(x,y) dxdy, \tag{2.12}$$

where the weight function $w(x,y) \in \Omega\times\Omega$ is non-negative ($w(x,y)\geq 0$) and symmetric ($w(x,y) = w(y,x)$). The corresponding Euler-Lagrange descent flow on this functional is

$$\frac{\partial u(x)}{\partial t} = -\partial_u J(u)(x) = -\int\limits_{\Omega} (u(x)-u(y)) w(x,y) dy, \quad u_{t=0}(x) = f(x). \tag{2.13}$$

The authors state that multiple iterations of the flow (2.13) can approximate several neighborhood filters - including NLM and Yaroslavsky - up to a normalization factor, by taking specific choices for $w(x,y)$.

2.4 Neighborhood filtering is nonlocal diffusion

In this section we will introduce some basic principles from spectral graph theory (see e.g. [18]) that will allow us to write (2.13) as a nonlocal diffusion equation. This provides a nice interpretation of denoising methods as diffusion processes on graphs embedded in Euclidean space. The local denoising methods use a graph defined by a local similarity measure (i.e. points nearby are similar), whereas nonlocal methods use more sophisticated similarity measures yielding nonlocal structures in the original image space.

A fundamental operator in spectral graph theory is the so-called *graph Laplacian*. We define $\mathcal{G} = (\mathcal{V},\mathcal{E},W)$ as an undirected weighted graph with vertices $\mathcal{V}$, edges $\mathcal{E}$ and corresponding weights W. We assume non-negative and symmetric weights, i.e. $W_{ij} = W_{ji}, W_{ij}\geq 0$ for all edges $e(i,j)\in\mathcal{E}$. In this framework a signal on the graph $\mathcal{G}$ is defined as the function $s:\mathcal{V}\to\mathbb{R}$.

Following [17], we define the non-normalized graph Laplacian on the graph $\mathcal{G}(\mathcal{V}, \mathcal{E}, W)$ as the difference operator

$$\Delta_w(u(i)) := \sum_{j:w_{ij}\neq 0} w_{ij}(s(j) - s(i)), \tag{2.14}$$

for the graph signal u and $i, j \in \mathcal{V}$. Note that this definition corresponds the negative graph Laplacian as usually defined in graph theory. Its natural continuous counterpart is the function

$$Lu(x) := \int_\Omega (u(y) - u(x))w(x, y)dy.$$

In this light, Gilboa and Osher argue that (2.13) can thus be interpreted as a nonlocal weighted linear diffusion equation:

$$\frac{\partial u(x)}{\partial t} = Lu(x), \quad u_{t=0}(x) = f(x). \tag{2.15}$$

In the successive paper [19] Gilboa and Osher extend this work and even show the relation with nonlocal nonlinear regularization functionals. For example they show that the minimization of the nonlocal counterpart of the nonlinear regularization functional (2.8), i.e.

$$J_{NL}(u) = \frac{1}{2}\int_\Omega \phi\left(|\nabla_w u|^2\right) dx,$$

will result in the gradient descent equation

$$\frac{\partial u(x)}{\partial t} = \nabla_w \cdot \left(\phi'\left(|\nabla_w u(x)|^2\right)\nabla_w u(x)\right), \quad u_{t=0}(x) = f(x) \tag{2.16}$$

similar to first term on the right-hand side of (2.9). Here $\phi(s)$ is a positive function convex in $\sqrt{s}$ with $\phi(0) = 0$. We refer the reader to [19] for exact definitions of the nonlocal gradient ∇_w and divergence ∇_w and other details.

We conclude from this that filtering with neighborhood filters as described in Section 2.3, can simply be interpreted as evolving the image with the nonlocal diffusion equation (2.16). Different filters such as Yaroslavsky's filter, the bilateral filter and nonlocal means (NLM) originate from different choices of the similarity graph. In fact, also the local diffusion processes as discussed in Section 2.2 arise from (2.16), if we take the similarity between two pixels inversely proportional to the Euclidean distance between them. We thus conclude that several popular methods for denoising can all be seen as specific choices of the similarity graph and diffusion function.

Which method to choose in what setting is still not very well understood: what similarity graph, and what diffusion function do we choose? It would therefore make sense to somehow *learn* from the data which method we should choose. In [10] a model has been proposed in this direction that can learn an optimal (local) diffusion model. We discuss this model in Chapter 4, and show how it is connected to neural networks. But first we will discuss the theory on neural networks in the next chapter.

Chapter 3

Representation learning with neural networks

Neural networks in general can be seen as a feature extraction followed by a task-specific module. We discuss two applications of neural networks: the supervised learning task of classification, and the unsupervised learning task of efficient data encoding. We show how these different tasks result naturally in different properties of the feature extraction. First of all we give a brief overview on the history of neural networks. Next we give some simple visual examples in order to justify and explain the computational structure in the most simple neural networks called multi-layer perceptrons. After this we make the transition towards convolutional neural networks and discuss their application to image classification. In the second part of the chapter we discuss the auto-encoder architecture for unsupervised learning, and explore similarities and differences with the supervised counterparts. Moreover, we try to develop an understanding of auto-encoders by making connections with methods for dimensionality reduction and filtering techniques. The overall goal of this chapter is to develop a deeper understanding of neural networks in general. In the next chapter we will use this knowledge to shed some light on the similarities between variational methods and neural networks.

3.1 An introduction to neural networks for classification

3.1.1 A brief history on neural networks

Since thousand years ago, researchers have been fascinated by the processing capability of the human brain. With the increasing popularity of electronic devices and computational power of computers, it came natural to try and capture this complex piece of nature with a machine: artificial intelligence was born.

Following the pioneering work on the modeling of neural networks in the brain by McCulloch, Pitts [20] and Hebb [21], the perceptron algorithm was introduced in 1957 by Frank Rosenblatt [22]. The perceptron algorithm is known as the first neural network, and is able to divide its input data in one or two classes via the calculation of a weighted sum followed by hard thresholding. Although the program seemed promising, it was proven by Minksy and Papert [23] to be limiting in the sense that this so-called single-layer perceptron (SLP) was not capable of

capturing the simple logical XOR operation. This publication discouraged many researchers to continue working on neural networks and, as a consequence, their popularity decreased.

In 1974 a resurgence occured through the work of amongst others Werbos [24], [25], that popularized the use of the backpropagation algorithm (BP) in NNs. Besides solving the XOR problem, this algorithm allowed quick training of multi-layer perceptrons (MLPs), the more complex counterpart of the SLP. One of the key advances that allowed the use of BP to train MLPs was the introduction of differentiable activation functions, instead of step functions as used in SLPs. This allowed for the calculation of the gradient of the loss function required for gradient descent, the backbone of the BP algorithm.

A few years earlier, the research by Hubel and Wiesel [26] was published that would form the inspiration for many NNs for visual pattern recognition. In their work it was shown that the visual perception system of a cat processed information hierachically through simple cells followed up by complex cells. These simple cells were shown to respond directly to the exact location of a stimulation pattern present in the receptive field, whereas the complex cells fired on patterns in the receptive field regardless of their location. Based on this work, Fukushima [27] developed a network termed neocognitron with similar properties by introducing the combination of convolution and pooling in NNs. Next to the translation invariance of these networks, they also required relatively few parameters due to the weight sharing naturally present in the convolution operation. The neocognitron forms the basis for what we now call the convolutional NN (CNN). This network received increasing interest ever since its first introduction because of its immense performance on visual recognition tasks.

A few years after the introduction of the neocognitron, LeCun et al. [28] raised the bar even more by introducing BP to CNNs, allowing the networks to learn better and more efficiently. Furthermore, they introduced the MNIST handwritten digits, one of the main benchmark datasets for visual recognition available today.

3.1.2 The classification problem

Before we focus on the theory on neural networks, let us first define the classification problem formally. Suppose we are given a set of n instances $x^{(i)} \in \Omega \subset \mathbb{R}^n$ with corresponding labels $y^{(i)} \in \mathbb{R}^l$. We model the label using a vector with one-hot encoding in $\mathbb{R}^l$, where l is the total number of possible labels. One-hot encoding in this context means we fill in 1 at the position corresponding to the correct label, and 0 elsewhere. We collect all samples in a soxtend-called *training set* $\mathcal{S}_{\text{train}}$, i.e.

$$\mathcal{S}_{\text{train}} := \{x^{(i)}, y^{(i)})\}_{i=1\ldots n}. \tag{3.1}$$

The classification problem can now be stated as approximating the function $f : \Omega \to \mathbb{R}$ for the whole domain Ω from the instances $f(x^{(i)}) = y^{(i)}$ in $\mathcal{S}_{\text{train}}$. A perfect classifier will give the right labels for any instance $x \in \Omega$. The performance of the classifier f is usually measured in terms of accuracy on a test set $\mathcal{S}_{\text{test}}$ that satisfies $\mathcal{S}_{\text{test}} \cap \mathcal{S}_{\text{train}} = \emptyset$.

Let us now consider a very simple classification problem, to see how we could approximate f. Suppose we have three classes of objects, triangles, squares and circles, and we would like to separate these three classes of objects. See Fig. 3.1a for a visualization of the problem. One

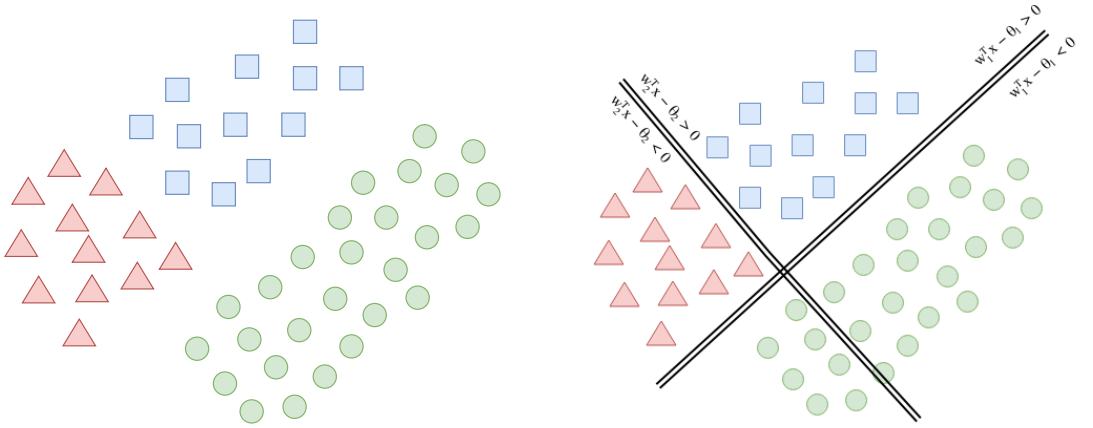

Figure 3.1: *Left:* An example of linearly separable data, as discussed in Section 3.1.2. *Right:* An example of two decision boundaries that could be used to seperate the three object classes.

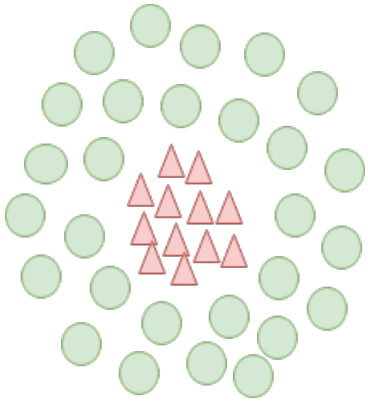

Figure 3.2: Circular data we wish to seperate into classes. As opposed to the data in Fig. 3.1a, this data is not directly linearly separable.

way to solve this classification problem is to first separate the circles from the triangles and squares. Since the data is linearly separable, this can be easily done by placing a hyperplane between the circles and the rest of the instances. The separating hyperplane in this case is $w_1^T x - b_1 = 0$ (see Fig. 3.1b). The function f that will correctly classify the circle instances is a simple orthogonal projection on the vector w_1 followed by a thresholding. Now that we have separated the circles, we can separate the squares and the triangles in a similar fashion with the hyperplane $w_2^T x - b_2 = 0$ (see Fig. 3.1b). By first removing all instances already classified as circles, we can now separate the squares from the triangles by an orthogonal projection on the vector w_2 followed by a thresholding. In this example the function f was very easy to construct, because the data was linearly separable. Very early approaches to tackle classification problems work similarly to what we described above, and will be subject of Section 3.1.3. In general, however, data is not linearly separable and therefore this approach will not work.

Suppose for example that we wish to classify the circular data in Fig. 3.2. In this setting, we cannot separate the data with just a few hyperplanes and orthogonal projections. However, we *can* use a similar approach if we first apply a polar coordinate transformation on the data.

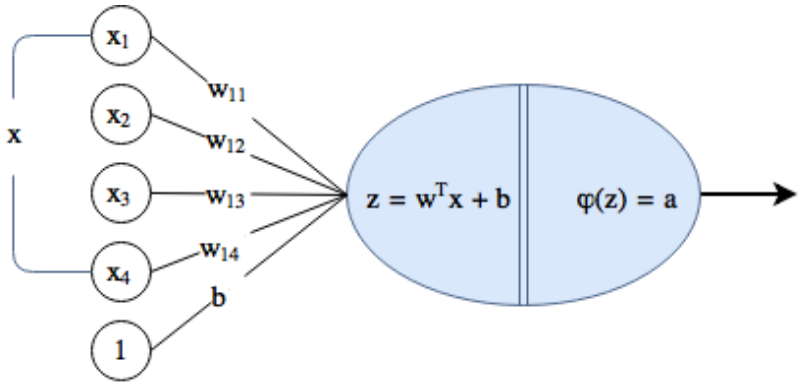

Figure 3.3: Illustration of a neuron with inputs $x \in \mathbb{R}^4$, bias b, weights w and activation function ϕ, as defined in (3.2).

After this transformation the data is easily linearly separable based on just the radial coordinate. So in this case performing a transformation to a feature space made the classification much easier, namely linear.

The approach above seems like a good strategy to tackle general nonlinear classification problem: first map the instances to a more informative space, and then separate them in this space instead of the original input space. We will see in Section 3.1.4 that neural networks can be interpreted as doing exactly this. Before we look into the details of neural networks as used nowadays, we will look at one of the predecessors: the perceptron.

3.1.3 The perceptron

The perceptron [22] is one of the first neural networks ever build and it was designed for classification of points in the input space. It is build of computational units called neurons, all of which calculate a weighted sum of the input followed by a nonlinear activation function.

Definition 3.1 (Artificial neuron)**.** A neuron with input $x \in \mathbb{R}^n$, activation function $\phi : \mathbb{R} \to \mathbb{R}$, weights $w \in \mathbb{R}^n$ and bias $b \in \mathbb{R}$ is defined as the function

$$\eta(x; \phi, w, b) = \phi(w^\top x + b). \tag{3.2}$$

We refer to $z = w^\top x + b$ as the scores, and its output $a = \phi(z)$ as the activations. ◊

In literature, neural networks are usually pictured as an a-cyclic computational graph wherein data flows from left to right. A neuron with properties as defined in Definition 3.1 can therefore be visualized as in Fig. 3.3. Here the circles denote the computational units; these units have inputs on the left and outputs on the right. The inputs $\tilde{x}$ flow through the connected lines towards the computational unit by multiplication with their corresponding non-zero weights w_{1j}. The unit then computes the *score* z of the neuron by summing all weighted inputs, effectively computing an inner product between w and x. This is in turn followed up by a nonlinear activation function, yielding the *activation* a of the neuron. The original perceptron as introduced in [22] uses neurons as defined in Definition 3.1 with a hard threshold as activation function, i.e.

$$\phi(z; \theta) = \begin{cases} 1, & z \geq 0 \\ 0, & z < 0. \end{cases}$$

Following the definition in (3.2) and taking the weighted sum as input, we see that this neuron simply separates the input space by means of the hyperplane $w^T x = -b$:

$$\phi(w^T x + b) = \begin{cases} 1, & w^T x + b \geq 0 \\ 0, & w^T x + b < 0 \end{cases} = \begin{cases} 1, & w^T x \geq -b \\ 0, & w^T x < -b. \end{cases} \tag{3.3}$$

We see that the biases b actually determine the threshold of the neuron; in this case the threshold is $-b$. As can be seen from this formula, a perceptron can be used to linearly separate its input data into two classes by learning optimal weights and bias. In fact, it learns exactly the separating hyperplanes as discussed in Section 3.1.2. We can combine multiple neurons in order to extend the classification to more classes, which is done in the perceptron. How the optimal weights and bias, and thus the optimally separating hyperplanes, can be learned will be discussed in Section 3.2.

3.1.4 Multi-layer perceptron

The general idea behind neural networks. In the previous subsection we saw that neurons can be used to linearly separate data in the input space. In general, however, data is likely *not* linearly separable and hence this approach will not work. In order to tackle a nonlinear classification problem, neural networks are designed to learn a mapping to a more representative feature space. Now instead of classifying points in the input space directly, the networks classify the inputs in the feature space. The idea is that the representation in the feature space will yield easier, separable data, and therefore makes the classification task easier.

In general, all neural networks start with an input layer, and end with an output layer. In between these two layers are the so-called *hidden layers*. We say that the *depth* of the network increases whenever more hidden layers are used, i.e. a deep neural network consists of more hidden layers than a shallow neural network. All layers together form a staged computational structure; the input flows through the network starting with computations in the first hidden layer, followed by the second, until the output layer is reached. In this process each layer passes its activations as inputs to the next layer. We refer to this computational process as the forward pass of the network. As we will see later, the backward pass is used to update the parameters in the network.

One of the most general type of neural networks is the so-called *multi-layer perceptron* (MLP). As the name suggests, it is a composition of units similar to perceptrons in multiple *layers*. To emphasize its properties, the type of layer used in an MLP is referred to as a *fully-connected* (FC) layer:

Definition 3.2 (Fully-connected layer)**.** Suppose we have inputs $x \in \mathbb{R}^{N_x}$ and N_y neurons with corresponding differentiable activation functions $\phi : \mathbb{R} \to \mathbb{R}$, weight vectors $w_i \in \mathbb{R}^{N_y}$ and biases $b_i \in \mathbb{R}$. We define a fully-connected layer as the function

$$g(x; W, b) := \phi(Wx + b). \tag{3.4}$$

Here $W := [w_1, w_2, \ldots, w_{N_x}] \in \mathbb{R}^{N_y \times N_x}$ is the weight matrix, $b \in \mathbb{R}^{N_y}$ is the vector of biases and the activation function $\phi(\cdot)$ is applied point-wise. We denote the inputs to the activation

functions as $z := Wx+b \in \mathbb{R}^{N_x}$ and the outputs - or activations - of the layer as $a := \phi(z) \in \mathbb{R}^{N_y}$. Then we can write (3.4) simply as

$$a = \phi(z).$$

$\Diamond$

The only difference between a perceptron and fully-connected layer is that the FC layers use differentiable activation functions. A simple and popular choice is the sigmoid function $\phi(z) = (1 + \exp(-z))^{-1}$. The sigmoid function has, however, some severe drawbacks with regard to the optimization of the network and is therefore not always used in practice. We discuss this in more detail in Section 3.2.5. If we would train an MLP as described above, we would learn a mapping into a space in which we can nicely perform classification. So in other words, we can view an MLP as a function from input space to label space, consisting of a feature extraction- followed up by a classification function:

Definition 3.3 (Multi-layer perceptron and neural network)**.** A neural network with L layers is a function $f : \mathbb{R}^n \to \mathcal{L}$ defined as

$$f(x) = \mathcal{C}(\mathcal{F}(x; \theta_F); \theta_C).$$

Here $\mathcal{L}$ is the label space, $\mathcal{F}(x; \theta_F)$ is the feature extraction function, parametrized by the weights θ_F, and $\mathcal{C}$ is a task-specific function, parametrized by the weights θ_C. In classification networks $\mathcal{C}$ is a classifier, but in auto-encoders (see Section 3.4) it is a reconstruction function. Both $\mathcal{F}$ and $\mathcal{C}$ are compositions of multiple layers $g_i(\cdot)$. Hence, for a total number of N_F layers in the feature extractor we have

$$\mathcal{F}(x) = (g_{N_F} \circ g_{N_F-1} \circ \cdots \circ g_1)(x)$$

and

$$\mathcal{C}(y) = (g_L \circ g_{N_{l-1}} \circ \cdots \circ g_{N_F+1})(y).$$

$\Diamond$

The MLP architecture arises from this more general definition by taking fully-connected layers g_i (see Definition 3.2) in the feature extractor $\mathcal{F}$. The classifier $\mathcal{C}$ is designed to map the outputs from the feature extractor to a score for each class. They are usually not very sophisticated, and only consist of a few fully-connected layers with a final output size equal to the total number of possible labels. As we will see in the next section, the network then adjusts its parameters based on a distance measure between the scores and the desired ground-truth output.

How depth increases complexity. Since each layer is a linear transformation followed by a nonlinear activation function, we can conclude that a neural network is a type parametrization of a nonlinear mapping. Because of the nonlinear activation functions, an increase in the depth of the network will result in a more complex and more nonlinear mapping. It is now also clear why we should always use nonlinear activation functions: a composition of linear transformation is still linear, so stacking layers with linear activation function will not increase the complexity and explanation power of the network. Another way to see how depth in the network results in more complexity is to interpret neurons as decision units. Each layer makes a set of decisions based on the current value of the weights and the input of the layer. Since the input of one layer is the output of the previous, this means we make decisions based on the outcome of other decision. In

other words, our final complex decision (the output of the network) is built upon other, simpler decisions. By increasing the number of layers, the number of possible paths towards the final decision outcomes increases and therefore the complexity of the decision making is increased.

3.2 Deep learning

Until now we have seen that neural networks can be interpreted as systems that learn the most optimal representation for the input data, in order to present the most informative properties of the samples to the classifier. However, we have not yet looked at what may even be the most important and hardest part of working with neural networks: training them. For easier networks this can be achieved quite easily and hardly requires any background knowledge, but for deeper networks it turns out to be very challenging. In fact, this has become so important and popular in the past few years that it became an active research field on its own: deep learning. We will now discuss some important components of deep learning.

3.2.1 Cost-functions

In order to learn the optimal parameters for a certain task, we must first define what optimality means; when is the output of our network good, and when is it bad? Similarly to energies in variational methods, neural networks judge the output with a particular set of weights based on the value of a cost-function. A set of weights resulting in a low cost is better than a set of weights resulting in a high cost. If we have ground-truth data available, then the cost for one sample will usually be defined as some measure of distance between the output of the network and the desired ground-truth output for that particular sample.

As before, let us denote the output of the network as the function $f : \Omega \to \mathbb{R}^l$. For convenience, let us denote the output of the network with an input $x^{(i)} \in \Omega \subset \mathbb{R}^n$ and a specific set of parameters θ simply as $f^{(i)}$. Moreover, let us write the ground-truth corresponding to the sample $x^{(i)} \in \mathcal{S}_{\text{train}}$ as $y^{(i)}$. The optimization of a network parametrized by θ *over all* N *samples* $x^{(i)}$ can now mathematically be described as the minimization problem

$$\min_{\theta} \; C(\theta) = \frac{1}{N} \sum_{i=1}^{N} C^{(i)} \left(f^{(i)}, y^{(i)} \right) + \lambda J(\theta), \tag{3.5}$$

where $0 \leq \lambda \in \mathbb{R}$ is the regularization parameter. Here $C^{(i)}$ denotes the cost for the sample $x^{(i)}$, C denotes the overall cost and $J(\theta)$ denotes a regularizer. We will now discuss some different types of cost-functions, and in the next paragraph we will discuss regularization of neural networks.

Popular choices for the cost-function $C^{(i)}$ include the quadratic cost, cross-entropy cost. The quadratic cost is defined as

$$C_E^{(i)} \left(f^{(i)}, y^{(i)} \right) := \| f^{(i)} - y^{(i)} \|_2^2,$$

which is simply the squared Euclidean distance between the network output and the desired ground-truth. For several reasons this cost-function is not used much in practice. One of the reasons is that for a large number of possible labels, the vectors $f^{(i)}, y^{(i)}$ will be very high-dimensional. Euclidean distances in high-dimensional spaces are, however, not very informative;

it is easily possible that two instances are very similar while their Euclidean distance leads to believe otherwise. Because of this, the cost-function is very unstable and will make convergence to good solutions hard. In networks such as the unsupervised auto-encoder, however, the Euclidean cost-function is still used sometimes. Here the goal is to reconstruct the input using a latent variable representation, we will discuss this in Section 3.4. Mostly whenever the output after first forward pass of the network is already quite close to the desired reconstruction of the intput, this can still lead to good convergence. As we will see later, one way to achieve this is through the use of *skip-connections*.

The cross-entropy cost is defined as

$$C_{CE}\left(f^{(i)}, y^{(i)}\right) := -\sum_{j=1}^{l} y_j^{(i)} \log f_j^{(i)}. \tag{3.6}$$

Since the cross-entropy cost can be seen as the distance between the probability distributions $f^{(i)}$ and $y^{(i)}$, we must assure that our network outputs a probability distribution. That is, all activations in the last layer must be in the range $[0, 1]$ and must sum up to 1. In other words we would like to have that our outputs $f_j^{(i)}$ can be interpreted as the conditional probabilities $\mathbb{P}\left(y_j^{(i)} = 1 \mid x^{(i)}\right)$. One way to do this is to normalize the *scores* $z_j^{(i)}$ in the last layer of the network with the softmax function, such that

$$f_j^{(i)} = \frac{\exp\left(z_j^{(i)}\right)}{\sum_{j'=1}^{l} \exp\left(z_j^{(i)}\right)}. \tag{3.7}$$

Note that in this case the softmax function forms the activation function of the last layer. Whenever the softmax function is used, the cost-function is usually called the softmax cross-entropy. It is not too hard to see that this is a generalization of logistic regression, which corresponds to a network with just one output neuron with the sigmoid as activation function.

3.2.2 Regularization in neural networks

The term $J(\theta)$ in (3.5) is the regularization term of the cost-function in a neural net. A popular regularization is the L^1 regularization, enforcing sparse weight matrices. In this case the regularization parameter λ is called the weight-decay. Other popular choices for J include the L^2-norm and the Kullback-Leibler (KL) divergence [29]. The latter is defined as

$$C_{KL}^{(i)}\left(f^{(i)}, y^{(i)}\right) := D_{KL}(y^{(i)} || f^{(i)}) = \sum_j y_j^{(i)} \ln\left(\frac{y_j^{(i)}}{f_j^{(i)}}\right), \tag{3.8}$$

and can be seen as measure of how much information is lost whenever we approximate $y^{(i)}$ with $f^{(i)}$. This is particularly useful and popular in auto-encoders, since these neural networks are trained to compress the input data with as little information loss as possible.

Besides the regularization terms discussed above, another frequently used technique for regularization is the dropout technique [30], [31]. When a layer is equipped with dropout with a parameter p, this means that with probability p we drop the weights (i.e. set it to zero) of any neuron in that particular layer right before the forward pass. The idea is that this prevents complex co-adaptations of neurons in which feature detectors are highly dependent on feature

detectors in previous layers. This forces the network to use all of its capacity and to learn more general feature detectors, instead of relying on a small set of neurons. Hence, dropout prevents overfitting.

Finally, data-augmentation is used as regularization in neural networks. Instead of taking a fixed set with training instances, data-augmentation performs random transformations on each instance before processing it. Therefore with data-augmentations such as random rotation and translation, it will become significantly harder to overfit the training data. Data-augmentation can also be seen as a way to learn specific invariances to your network. Although we perform the random transformation on the training instances before we process them, we do not change any of the labels. For a classification network this means that in this way we can force insensitivity to the type of transformations used in the data-augmentation. For example, in Chapter 5 we train a network for the classification of cells. Since rotations of the cells will still return cells with exactly the same biological characterization, we *want* the classification to be invariant to rotation. Similarly, small translation of the cells should not lead to different labels. Therefore we add random rotations and small random translations as data-augmentation to the classification network. We will look at the invariances of neural networks in more detail in Section 3.3.4.

3.2.3 Optimization algorithms

In order to minimize (3.5) variants of an algorithm called *stochastic gradient descent* are used. This is a variant on the optimization algorithm gradient descent, which is particularly useful in case of non-convex optimization problems like (3.5). In gradient descent optimization, we iteratively perform the updates of the parameters based on the gradient of the cost-function with respect the parameters. The $k-th$ iteration of the algorithm computes

$$\theta_k = \theta_{k-1} - \eta \partial_\theta C(\theta_k). \tag{3.9}$$

Here η is called the *learning rate*. This means we move in the direction of steepest descent in the cost-function with step-size η. To see why stochasticity is helpful in case of non-convex optimization, let us consider the example in Fig. 3.4. Here we try to find the minimum of the non-convex function in the figure by means of gradient descent updates. Because the graph has many local minima, plain gradient descent will likely converge to one of the local minima instead of the global minimum. By adding randomness in the updates we hope to escape bad attractors. In stochastic gradient descent we do this by approximating the derivative $\partial_\theta C$. For ease of notation we have dropped the arguments. If we simple write the cost-function as the sum over the cost for all training samples x, i.e. $C = \frac{1}{N}\sum_x C_x$, then we see that the derivative is given by a similar sum $\partial_\theta C = \frac{1}{N}\sum_x \partial_\theta C_x$. This can be interpreted as the sample mean of the cost over N samples. From statistics we know that decreasing the number of samples will decrease the confidence of our approximation. By decreasing this confidence we can thus increase the randomness of our updates. This is exactly what happens with stochastic gradient descent, the derivatives $\partial_\theta C$ are approximated by the derivative of just one, randomly chosen, sample:

$$\partial_\theta C \approx \partial_\theta C_x.$$

In expectation this approximation is equal to the actual derivative used in gradient descent:

$$\mathbb{E}_x[\partial_\theta C_x] = \frac{1}{N}\sum_x \partial_\theta C_x.$$

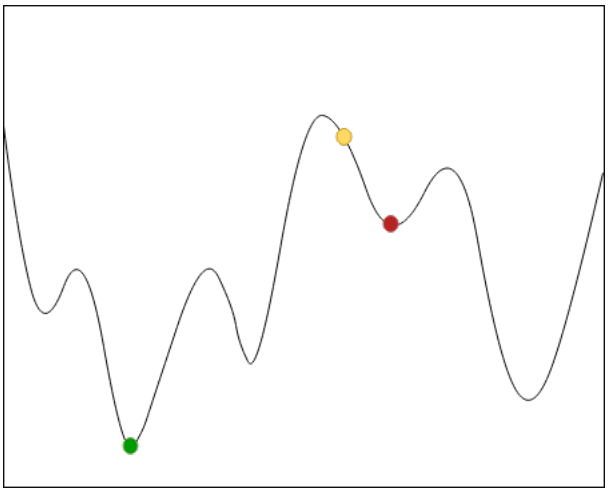

Figure 3.4: Non-convex optimization with gradient descent. If we start at the yellow dot on the graph and proceed in the direction of steepest descent, then the algorithm will likely converge to the local minimum indicated with the red dot instead of the global minimum indicated with the green dot.

In practice, a generalization of stochastic gradient descent is used, called mini-batch gradient descent. Instead of approximating the derivative with just one randomly chosen sample, the derivative is approximated by averaging over a small subset. This subset is randomly chosen from the training set and is called a mini-batch. The mini-batch size used in this case will determine the amount of randomness present in the updates. Mini-batch gradient descent with a batch-size of N will result in plain gradient descent, whereas a batch-size of 1 will result in stochastic gradient descent.

Besides the added benefit of stochasticity in the updates, mini-batch gradient descent and its variants have another big benefit for the optimization of neural networks. If we would use plain gradient descent, then the cost-function is a function of *all* samples in the training set. This means we must process the *whole* training set in order to get a value of the cost for a particular set of weights. Because of the increasing interest in training deep networks, training sets have grown accordingly towards more than a million samples in some cases. Calculating the cost-function over all samples in those cases is therefore far from optimal. With the mini-batch gradient descent we only need to calculate the gradients for a number of samples equal to the size of the mini-batch, and is therefore much more reasonable in practice.

3.2.4 Back-propagation

Now that we have looked at some algorithms for optimization, it is time to focus on the backbone of the neural networks: the back-propagation algorithm. In the previous paragraphs we saw that the gradient of the cost-function with respect to the parameters is required for the gradient descent updates, but we have not yet discussed how to calculate these gradients.

In short, the back-propagation algorithm is an implementation of gradient descent on the cost-

function in combination with the chain rule for differentiation. Recall that for gradient descent we need to compute the gradient of the cost-function with respect to all trainable parameters in the network. In order to calculate these derivatives, the back-propagation algorithm computes the so-called *backward pass* of the neural networks. Recall that the forward pass of a network corresponds to a left-to-right data flow. The backward pass consequently corresponds to a right-to-left data flow. Using the chain rule, we can express the gradients with respect to parameters in the $(l-1)$-th layer in terms of gradients with respect to parameters in the l-th layer. Exploiting this fact, the back-propagation algorithm first computes the gradients of the cost-function with respect to the parameters in the last layer L. Once these are computed, they are used to compute the derivatives with respect to the parameters in the $(L-1)$-th layer. The process is repeated until the gradients in the first layer of the network have been computed. The origin of the name back-propagation is therefore very clear: the algorithm back-propagates the gradients from later layers to earlier layers. When all gradients are computed it performs updates like (3.9) for all parameters in θ. Next, a forward pass is computed, yielding the value of the cost-function for the new set of weights. This process is repeated until convergence or manual termination.

3.2.5 Activation functions and vanishing gradients

Above we have outlined the back-propagation algorithm to compute the derivatives of the cost-function with respect to the network parameters recursivly. We have, however, omitted a very important detail: we require differentiability of the cost-function with respect to *all* network parameters. In case of a perceptron this is not the case, since we have an activation function that is not differentiable with respect to its weights. Therefore, in order to apply back-propagation we require the use of differentiable activation functions. Popular choices include the sigmoid function, tanh and rectified linear units (ReLUs) [32] defined as $\phi(x) = \max\{0, x\}$. As we will see next, not every activation function is suitable for a deep neural network because of the potential of causing either vanishing gradients or gradients that blow up.

Suppose we want to update a parameter w in the first layer of a network with L FC layers. Back-propagation calculates the derivative of the cost-function with respect to this parameter via the chain-rule in order to calculate the update. The derivative of one particular neuron with respect to its parameters involves the derivative of its activation function, so the derivative of the cost-function with respect to our parameter w will always involve L multiplications of derivatives of activation functions. Now this may not seem like a problem, but let us look closer at these derivatives. For traditional activation functions like the sigmoid and tanh, we will have derivatives in the range $[0, 1]$. Since we have to initialize with non-zero weights, most of these derivatives will in fact be smaller than 1. Therefore, the gradient of the cost-function with respect to our parameter in the first layer will decrease exponentially with the number of layers L. Hence, for a deep network we obtain a derivative very close to zero. The result is that the parameters in the early layers will hardly receive any updates and are therefore difficult to train, resulting in bad overall performance of the network. This phenomenon is what is called vanishing gradients.

On the other hand, if we would take activation functions with derivatives larger than one, then we can in a similar way obtain exploding gradients which will cause the whole gradient

descent scheme to blow up. Therefore, in present-day deep networks usually ReLU activation functions are used: they have a constant gradient of 1 for positive inputs, and a constant gradient of 0 for negative inputs. Usually the biases are initialized with small positive values in order to escape the regime of zero gradients in the beginning of the training progress. Other variants such as the leaky ReLU have been proposed [33] in order to completely evade zero gradients. Besides changing the activation functions to evade vanishing gradients and gradients that blow up, researchers have also proposed to use residual networks [34] that make use of skip-connections in order to guarantee the back-propagation of stable, non-zero gradients. We will look at this type of network in more detail in Chapter 4.

3.3 Convolutional neural networks

In the previous section we saw how multi-layer perceptrons can be used as classifiers. The applicability of MLPs to computer vision tasks such as image recognition is, however, limited. The reason for this is that images are in general high-dimensional objects, and therefore the number of parameters in the MLP becomes very large. For shallow networks optimization is still reasonable, but as we will see in the next section, this becomes troublesome for deeper networks. In this section we will consider an alternative type of networks called convolutional neural networks (CNNs). This network reduces the number of parameters significantly by exploiting the similarity structure in images. By means of the example of face recognition we will make a natural transition from MLPs to CNNs. After this, we discuss the properties of this type of neural networks in order to gain an understanding of their processing behavior. More specifically, we will follow the work by Mallat et al. [35]–[38] in order to mathematically describe the invariance properties of the feature extraction $\mathcal{F}$ in CNNs.

3.3.1 Face detection with MLPs

Let us suppose we want to design and train a neural network for the detection of faces. These faces are presented to the system in the form of $M \times M$ images $I \in \Omega \subset \mathbb{R}^{M \times M}$. Hence we would like to learn a function $f : \Omega \to \{0, 1\}$ that outputs 1 whenever a face is present and 0 otherwise. Mathematically speaking, we thus search for a function f such that for $I \in \Omega$

$$f(I) = \begin{cases} 1, & I \in \mathcal{A}, \\ 0, & \text{otherwise}, \end{cases} \tag{3.10}$$

where $\mathcal{A} \subset \Omega$ is the set of faces. In case of a neural network, the function f corresponds to the forward pass and is chosen as a composition of a feature extraction function $\mathcal{F}(\cdot)$ and a classification function $\mathcal{C}(\cdot)$. As discussed before, the classification functions are usually kept quite simple and only consist of up to three fully-connected layers followed by a softmax layer. In contrast, the feature extraction is usually sophisticated and mostly determines the performance of the overall network.

By looking at (3.5), we see that training of an MLP is effectively an optimization problem in a very high-dimensional parameter space, and the number of parameters increases with the dimension of the input and the depth of the network. Since images usually live in high-dimensional space, training a deep network for image classification is a difficult task.

To give an example, consider training an MLP with a total of 4 hidden layers and 200 neurons per layer. Even for small images the total number of parameters (denoted as θ) in this rather simple network is already quite large:

$$|\theta| = 200M^2 + 3(200 \cdot 200) + 200 \cdot 2 = 200M + 120400.$$

Now suppose we would like to classify images with a resolution of 150×150 pixels with this network. In this setting the total number of learnable parameters already increases to over 4.6 million. Optimization over this many parameters is a very hard problem, so addressing it may make training easier.

3.3.2 Convolutional layers

In Section 3.1 we considered MLPs in which fully-connected layers are used to extract features. As their name indicates, these layers are connected to *all* neurons in the previous layer. For the first hidden layer, this means that it is connected to *every* element (or pixel) of the input data. Effectively, this also means that the feature extraction function $\mathcal{F}$ searches for features that span the entire image. This seems rather inefficient, and addressing it could in fact reduce the solution space to a more reasonable size. Natural images satisfy the stationarity property, i.e. the statistics of an image patch are independent of its location. This means that features present in some part of the image, are likely also present in some other part. Therefore it makes more sense to process an image via patches extracted from the image rather than as a whole. This way of processing images is also found in the early stages in the mammalian visual processing system in which neurons have only localized receptive fields ([26], [39]), justifying the transition from full images to small patches even more.

A natural way to incorporate the ideas from the discussion above into a network is through the use of the convolution operation. This is done through the *convolutional layer*:

Definition 3.4 (Convolutional layer)**.** Suppose we have discrete grayscale images $I \in \mathbb{R}^{M \times M}$, and let us denote with $I[x, y]$ the intensity of the pixel at position $[x, y]$. Furthermore, suppose we have N_k kernels k_d of size $n \times n$, biases $b_d \in \mathbb{R}^{(M-n+1)\times(M-n+1)}$ and outputs $z \in \mathbb{R}^{(M-n+1)\times(M-n+1)\times N_k}$. If we collect all learnable parameters in $\theta = \{\theta_d\}_{d=1\ldots N_k}$, for $\theta_d = \{k_d, b_d\}$, then then the d-th resulting *feature map* of the convolutional layer is defined as

$$g(x; \theta_d) := z[x, y, d] = \sum_{i=0}^{n-1} \sum_{j=0}^{n-1} I[x - i, y - j] k_d[i, j] + b_d[x, y]. \tag{3.11}$$

Note from the size of the outputs y that this corresponds to the 'valid' convolution, i.e.

$$z[x, y, d] = (I * k_d)[x, y] + b_d[x, y].$$

The convolutional layer is usually followed up with a nonlinear activation function, eventually yielding the activations $a = \phi(z)$ again. ♢

The convolution operation slides a small template (usually maximally 9×9 pixels) over the image patch-wise and computes the weighted sum over the currently selected patch using weights from the template. The results of this computation is usually first passed through an activation function, after which they are passed on as inputs to the neurons in the next layer. In other

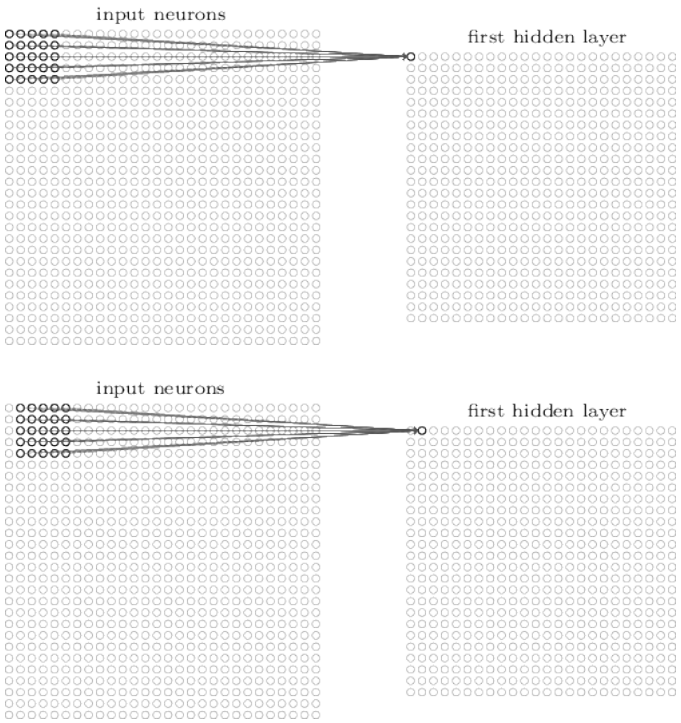

Figure 3.5: A visualization of the convolutional layer as defined in Definition 3.4. In this case a local receptive field of size 5×5 is slid over the grayscale input image of size 28×28. The lines correspond to non-zero weights, just as in the case of FC layers. [40]

words, it does exactly what we described above: process (or filter) an image by searching for local features at every location. From Fig. 3.5 we can see how (3.11) can be captured by a network by using locally connected neurons in combination with weight-sharing. All neurons at the input of a convolutional layer with kernels k_d of size $n \times n$ are connected via n^2 non-zero weights to neurons in the next layer. The pattern of neurons at the input with non-zero corresponding weights, forms the so-called receptive field of the convolutional layer. In CNNs it is usually chosen as a square. This receptive field is slid over the input until it has been covered completely. From Fig. 3.5 we can also see the connection with fully-connected layer: a convolutional layer can be seen as a FC layer with a sparse weight matrix. The sparsity structure in this matrix is determined by the parameters of the convolutional layer. One difference, however, is that the neurons in the convolutional layer use shared-weights instead of a different set of weights for every neuron. This enforces each neuron in the same feature map to search for the same pattern, but in a slightly different region. Recall that a feature map is the result of convolution with one kernel. So the convolutional layer in (3.11) produces N_k feature maps of size $\mathbb{R}^{(M-n+1)\times(M-n+1)}$.

3.3.3 Pooling layers

Next to the convolutional layers, CNNs usually also contain *pooling layers*. Pooling layers were introduced to summarize the outputs of convolutional layers by means of subsampling. The idea is that only the important activations are passed forward to the layers after the pooling, and

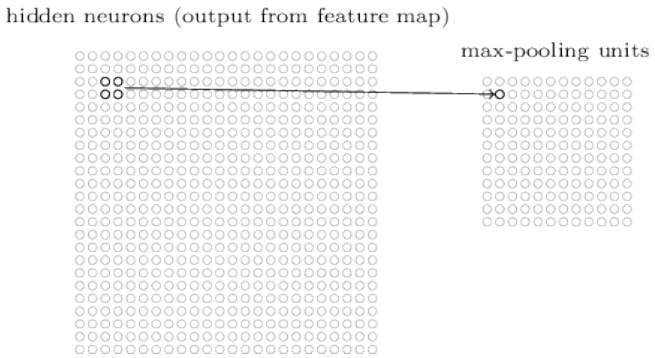

Figure 3.6: A visualization of a max-pooling layer with receptive field of size 2×2. [40]

that this will result in more robust features. Formally the pooling layer is defined as follows:

Definition 3.5 (Pooling layer)**.** Let us denote with $\mathcal{R}_n^I[x, y]$ a window of size $n \times n$ from the image $I \in \mathbb{R}^{M \times M}$, with bottom-left coordinate $[x, y]$. Then the forward pass of a pooling layer with receptive field of size n computes the outputs

$$a[x, y] = \phi(\mathcal{R}_n^I[nx, ny]),$$

for $0 \leq x, y \leq M - 2$. In case of even M we have $a \in \mathbb{R}^{M/2 \times M/2}$, and in case of odd M we add padding to the input to get $a \in \mathbb{R}^{M/2 \times M/2}$. Finally, $\phi : \mathbb{R}^{n \times n} \to \mathbb{R}$ is the pooling function that determines the type of pooling. ◊

A frequently used pooling function is the $\max(\cdot)$ function, resulting in so-called *max-pooling*. Max-pooling layers compute the maximum value over the receptive windows $\mathcal{R}_n^I$. A visualization of a pooling layer is shown in Fig. 3.6.

3.3.4 Understanding the exceptional performance of CNNs

Now that we have defined the typical layers in a CNN, we will look at them in more detail. More specifically, we will see how they influence the feature extraction $\mathcal{F}$ of the CNNs. Usually CNNs are constructed by a feature extractor consisting of a stack of multiple convolution and pooling layers followed up by a classifier at the very end of the network. The convolutional layers perform valid convolution, so with an $n \times n$ sized kernel and input $I \in \mathbb{R}^{M \times M}$, the resulting image is of size $(M - n + 1) \times (M - n + 1)$. Pooling layers with a window of size p result in downsampling of the input by a fraction of p. Let us now consider the simple network visualized in Fig. 3.7 and explore what happens to the effective receptive field as we pass forward through the network.

Hierarchical feature extraction. To see what happens with the receptive field, consider we somehow obtain an image of size 3×3 as output of the first layer of our network. Let us denote the actual input with I_0 and the output after the k-th layer with I_k. We want to explore what the effective receptive field is on I_0, if we apply a convolution on I_1. If we apply a 3×3 convolution on I_1, the kernels will cover the whole image in one step. Hence, if we calculate what the size of the actual input I_0 is, then we have our answer. Since we use convolutional layers with kernels of size 3×3, these layers downsize an $N \times N$ sized input to $(N-3+1) \times (N-3+1)$. If we obtain a 3×3 sized image as I_1, then we can easily solve $N-3+1 = 3$ to obtain $N = 5$. Hence the original

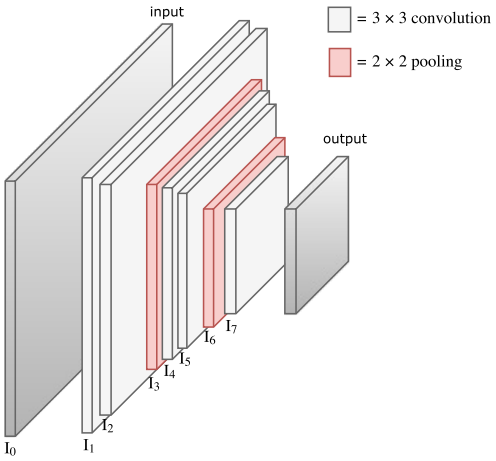

Figure 3.7: Visualization of a simple CNN, as discussed in Section 3.3.4.

input I_0 was of size 5×5. From this we see that if we apply a convolution with 3×3 receptive field on the output of the first layer, we effectively use a receptive field of size 5×5 on the input image.

We now look at the effective receptive field of the last convolutional layer in the network in Fig. 3.7. Suppose the output of the last pooling layer is of size 3×3, i.e. $I_6 \in \mathbb{R}^{3\times3}$. Using a similar analysis as above we derive the effective receptive field of this layer:

$$
\begin{aligned}
I_6 &\in \mathbb{R}^{3\times3} \\
I_5 &\in \mathbb{R}^{6\times6} \\
I_4 &\in \mathbb{R}^{8\times8} \\
I_3 &\in \mathbb{R}^{10\times10} \\
I_2 &\in \mathbb{R}^{20\times20} \\
I_1 &\in \mathbb{R}^{22\times22} \\
I_0 &\in \mathbb{R}^{24\times24}.
\end{aligned}
$$

The increase of the effective receptive field by stacking convolutional layers is solely due to the fact that the valid convolutions also downsize the image. Through the use of pooling layers in the architecture, we can directly target this downsizing and therefore directly increase the effective receptive field of the subsequent layers. In the previous paragraph we saw that the last convolutional layer in the network in Fig. 3.7 has an effective receptive field of 24×24. This is not a coincidence; in general the effective receptive field increases with the depth of the convolutional layer. In addition, we know from earlier discussion that convolutional layers extract patterns from their input. From our discussion in the last paragraph of Section 3.1.4 we also know that the complexity of the extracted features in layers increases with its depth. Combining these three observations, we can conclude the following: *the effective size and complexity of the extracted features in convolutional layers increases with the depth of the layer in the network.* In other words, CNNs learn to hierarchically extract features from the input images on different scales. The complexity level and number of scales in the extracted features is basically determined by

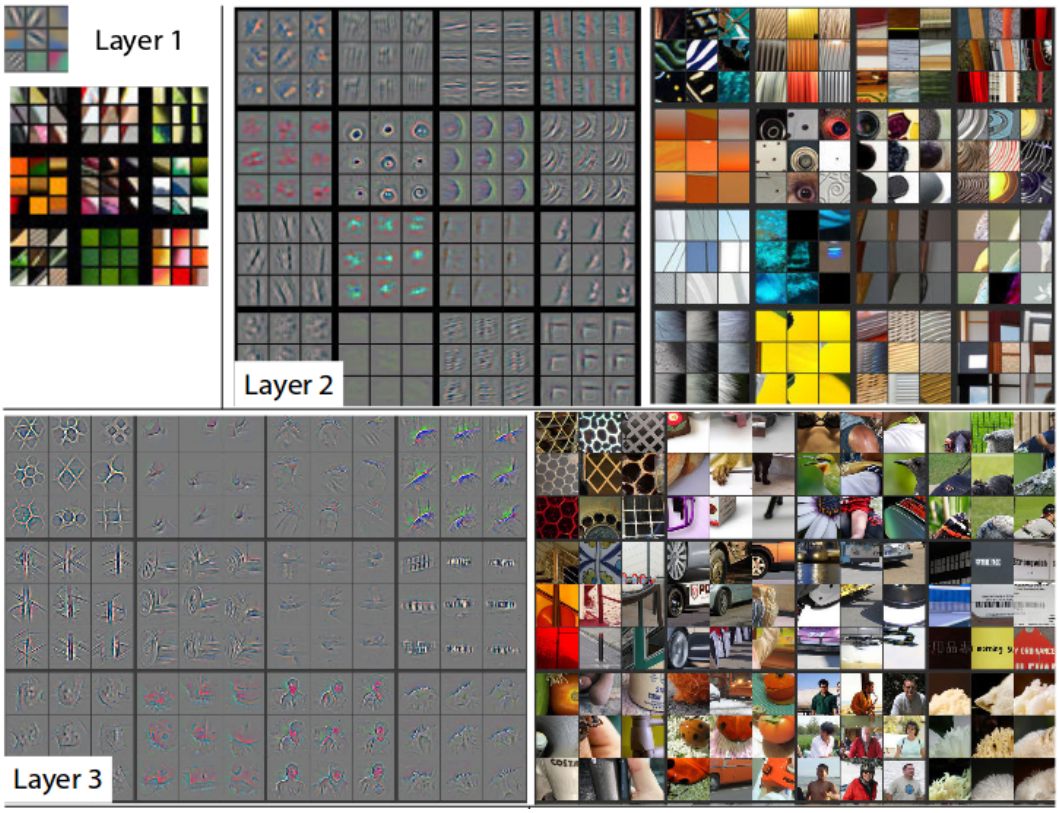

Figure 3.8: A visualization of the learned features in a deep CNN. For each layer images are shown of the original image, and corresponding images that caused most prominent activations in the layers. [5]

the number of layers, i.e. deep CNNs can extract more complex features than shallow CNNs. A visualization of patterns learned by a deep CNN is shown in Fig. 3.8 [5]. Here the most prominent activation in each layer are mapped back to image space using a deconvolutional neural network. In this way we can visualize the learned patterns by a deep CNN. From this visualization we can also see that the extracted features in deeper layers are more complex than in shallow layers.

Desired invariances of $\mathcal{F}$. In most cases a convolutional layer is followed up with sub-sampling through a pooling layer. In order to explain why this makes sense we will jump back to the face recognition problem, i.e. the problem of finding the function f in (3.10). Using this example we can derive some properties of the overall function f that we would like our system to have as well.

To start off, we would like to have a classification *independent* of absolute location; a face in the top-right corner of an image should be detected equally well as a face in the bottom-left corner. Mathematically speaking, we thus require that f is translation invariant for translations up to a certain size. Besides translations, we would like f to be invariant to small rotations and deformations.

It is not too hard to see that the convolution operation discussed above is *locally equivari-*

ant to translation, i.e. small translation of the input results in similar translation in the output. Take any integers $x, y \in [0, M-1]$, then

$$\begin{aligned}(\tau_{p,q} I * k)[x,y] &= \sum_{i=1}^{n}\sum_{j=1}^{n} I[x-p-i, y-q-j]k[i,j] \\ &= (I * k)[x-p, y-q] \\ &= (\tau_{p,q}(I * k))[x,y].\end{aligned}$$

Hence $(\tau_{p,q} I) * k = \tau_{p,q}(I * k)$, i.e. convolution is locally equivariant to translation. By pooling right after the convolution, we reduce the effect of the translation in the input. By repeating the convolution and pooling combination multiple times we can even obatin a feature extraction that is completely invariant to small translations. Similarly, we can obtain a feature extraction that is invariant to small rotations and deformations of the input.

In [38], statements on invariances in CNNs are formalized. All so-called local-symmetries are assumed to be contained in a group G. All small translations, deformations and rotations are then elements of this group. For any element $g \in G$ its action on x is defined as $g.x$. In [38], it is now argued that CNNs learn local invariants to the actions of this group, i.e. functions f such that

$$\forall x \in \Omega,\ \exists C_x > 0,\ \forall g \in G \quad \text{with } |g|_G < C_x, \quad f(g.x) = f(x).$$

Here $|g|_G$ is a metric that measures the distance between the identity and $g \in G$, and C_x a constant that determines the range of symmetries that preserve f. In CNNs these local invariants are learned as a combination of convolution, pooling and nonlinearities. The combination of convolution and pooling realizes the local invariance, whereas nonlinearities realize a contractive mapping. In every layer a contraction is applied, each killing undesired variances in the signal. The overall feature extractor $\mathcal{F}$ is therefore a contractive mapping, i.e. for every $x, y \in \mathbb{R}2$ there is a $c \in [0,1]$ such that

$$\|\mathcal{F}(x) - \mathcal{F}(y)\| \leq c\|x - y\|.$$

In conclusion, we see that the computational structure of CNNs realizes stability in the feature extraction and desired local invariances to translation, rotation and deformation. On top of this, the network perform an efficient hierarchical feature extraction to extract the most informative patterns from the data. All these properties are very beneficial to a classifier, and it is therefore not surprising that these networks perform so well on classification.

3.4 Unsupervised learning with auto-encoders

3.4.1 Introduction to auto-encoders

In the previous sections we have discussed the use of neural networks for the supervised classification task. Supervised in this context means that we had supervision from the ground-truth labels; recall the training set in (3.1) as $\mathcal{S}_{\text{train}} = \{x^{(i)}, y^{(i)}\}_{i=1\ldots N}$. In this section we will focus on an unsupervised learning algorithm called *auto-encoder* (AE). Auto-encoders are trained with back-propagation to reconstruct their input from a latent variable representation. So although no ground-truth is available AEs design their own targets as $y^{(i)} = x^{(i)}$. Instead of simply learning the identity mapping, an auto-encoder is forced to first *encode* the data in a compressed

representation, and subsequently *decode* from this compressed representation. Because of this compression, it must make a selection of the most important features for reconstruction, and will therefore learn structure about the data. Note that this is very similar to the idea of (sparse) dictionary learning [41].

Auto-encoders are defined as a neural network (see Definition 3.3) with encoder $\mathcal{F} : \mathbb{R}^n \to \mathbb{R}^h$, parametrized by θ_F, and decoder $\mathcal{C} : \mathbb{R}^h \to \mathbb{R}^n$, parametrized by θ_C. The forward pass is again given by $f(x) = \mathcal{C}(\mathcal{F}(x))$. Typically we have either $h \ll n$, or $h > n$ and $\mathcal{F}$ is forced to learn a sparse representation in $\mathbb{R}^h$. Auto-encoders usually minimize the squared reconstruction error with a regularization:

$$\min_{\theta=\{\theta_F,\theta_C\}} \frac{1}{N} \sum_{i=1}^{N} \left\| \mathcal{C}\left(\mathcal{F}\left(x^{(i)};\theta_F\right);\theta_C\right) - x^{(i)} \right\|_2^2 + \lambda J(\theta).$$

Possibilities for the regularization $J(\theta)$ include the Kullback-Leibler divergence as defined in (3.8), or a sparsity constraint such as the L^1-norm of the representation in $\mathbb{R}^h$. Especially the latter enforces sparsity, and is particularly useful when an over-complete latent representation is used ($h > n$).

Many types of auto-encoders have been proposed over the years, of which the simplest one is a neural network with just one hidden layer and so-called *tied-weights*, see Fig. 3.9 for a visualization. Taking tied-weights means that we take the transpose of the weight matrix from the encoder as weight matrix for the decoder. In this way the decoder function becomes the adjoint of the encoder function. Since this restricts the solution space of the auto-encoder it can be seen as a form of regularization. From now on we will assume tied-weights, unless stated otherwise. By omitting the biases and assuming tied-weights we can define the forward pass of the auto-encoder as simply

$$f(x; W) = W^\top \phi(Wx). \tag{3.12}$$

The figure on the right will form the basis for our interpretation of auto-encoders. The encoding in the hidden layer corresponds to downsampling and thus an increase of abstraction in the representation. The decoding can be seen as an upsampling and decreasing abstraction again. We will see in the next chapter that this interpretation proves useful to explain the performance of certain architectures.

Other auto-encoders that have been proposed including the contractive auto-encoder [42], convolutional auto-encoder [43], variational auto-encoder [44] and denoising auto-encoders [45]. All of these variants mainly originate from different regularization strategies to prevent AEs from learning the identity mapping. In the next subsection we will discuss the denoising variant of an auto-encoder.

3.4.2 Denoising auto-encoders

Since the reconstruction criterion alone is not enough by itself to extract useful features from the data, several variants of the auto-encoder have been proposed. Here we discuss the *denoising auto-encoders* (DAEs) [45]. The idea behind these DAEs is that a good representation is one that can be extracted from a noisy input and is sufficient for recovering the original, clean, input. A DAE is trained to reconstruct the clean input from its corrupted version; i.e. it is

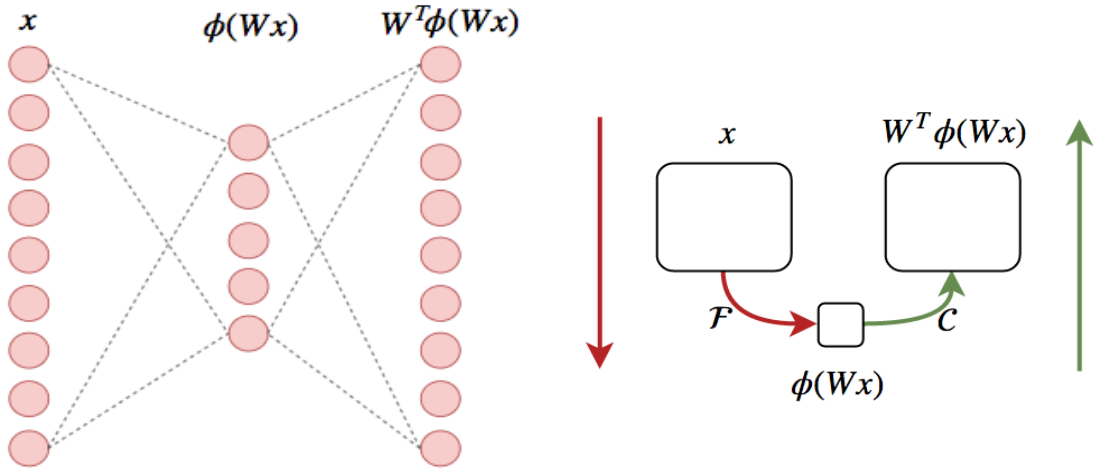

Figure 3.9: *Left:* Visualization of a simple auto-encoder with a fully-connected layer as both encoder and decoder, for which the biases are left out for convenience. It is trained with back-propagation to approximate $W^T\phi(Wx) \approx x$. *Right:* Alternative schematic representation of the auto-encoder on the left. The red arrows correspond to downsampling and increase of abstraction, and the green arrows correspond to upsampling and decrease of abstraction. This general view will form the basis of our interpretation of auto-encoders and other work in next chapter.

trained to denoise its inputs. In order to achieve the noisy inputs, the original (clean) samples are corrupted by a stochastic process. In general the architectures of DAEs are similar to AEs, but their learned representations and extracted features are usually more useful because they are robust and stable under corruptions of the input. See Fig. 3.10 for a nice visualization of the processes in a DAE. To emphasize the nature of the forward pass of this type of auto-encoder, we will refer to it as the reconstruction function and denote it with $r(x;\theta)$ instead of $f(x;\theta)$.

Because of the apparent similarities with the denoising processes in Chapter 2, we will discuss this type of auto-encoders in more detail in the next chapter. First let us look at some similarities with other dimensionality reduction techniques and filtering methods to get an intuition of what auto-encoders learn.

3.4.3 Auto-encoders combine data compression with filtering

There are many methods available for dimensionality reduction and filtering of signals, most of which decompose the signal in orthogonal basis functions. Examples include principal component analysis (PCA) and Fourier-related methods using Fourier transforms, wavelet transforms or discrete cosine transforms. Filtering coefficients in these basis representations amounts to amplifying or attenuating specific properties or patterns of the input signal, whereas sparsifying the representation amounts to a form of dimension reduction.

Most of the discrete versions of these methods can be written as an orthogonal projection into a spectral domain, followed by a filtering in the spectral domain, and finally followed by a projection back into the data domain. For example, let us consider the filtering of a signal x by means of a convolution with a kernel k. Since the basis of Fourier vectors diagonalizes the

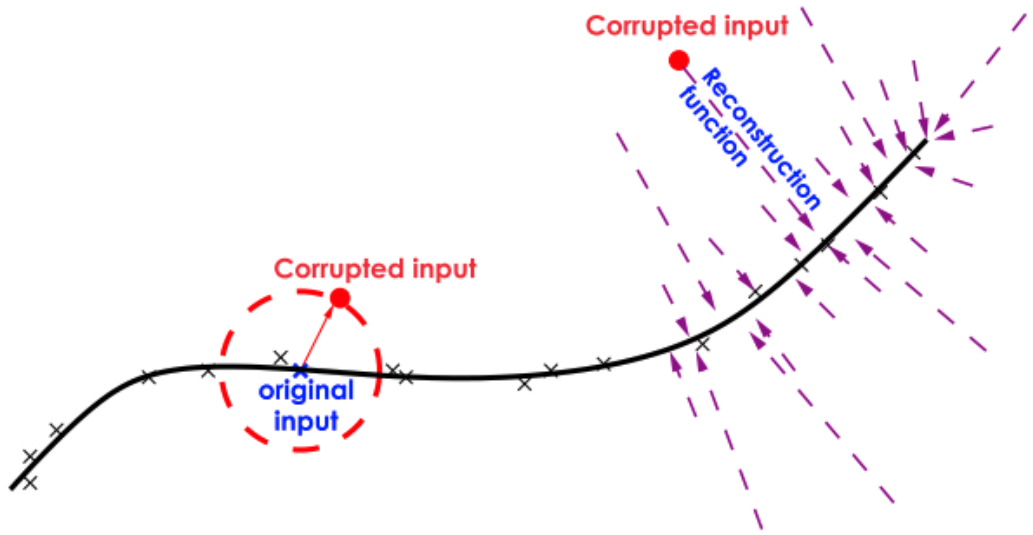

Figure 3.10: Visualization of the processes in a DAE. Suppose the the training data lies in close proximity to a lower-dimensional manifold. The stochastic corruption process maps the training data away from the manifold, while the reconstruction process is optimized to map it back towards the manifold. In other words, DAEs learn the lower-dimensional manifold structure in the input data. The hidden-layer representation can be interpreted as the coordinates on this manifold. [45]

operation of convolution, we can write this filtering process as follows:

$$(x * k) = U_F \hat{K}_F {U_F}^\dagger x.$$

Here U_F is the matrix with the Fourier vectors as columns, and $\hat{K}$ is the diagonal matrix with the filter coefficients in the spectral domain, $\hat{k}[\omega_k]$, along its diagonal. Generalization of convolution from the regular Euclidean grid to irregular grids in the form of undirected graphs [18] shows that convolution can be written even more generally as

$$(x * k)_\mathcal{G} = U_\mathcal{G} \hat{K}_\mathcal{G} {U_\mathcal{G}}^\dagger x. \tag{3.13}$$

Here $U_\mathcal{G}$ is the matrix with Fourier basis vectors on the graph $\mathcal{G}$ as its columns.

Besides convolution, we can also write a PCA approximation in a similar way. If we denote with U the matrix with eigenvectors of the covariance matrix, then we can write the PCA approximation with m principal components as

$$\text{PCA}_m(x) = U_m U_m^\top x. \tag{3.14}$$

Here U_m is the matrix with the first m principal components of U. Since the columns of the matrix U_m have the same dimension as the data x, we can visualize them in the image domain. For a data-driven technique like PCA, this results in interesting images of generic samples in the dataset. For PCA used on a set of faces, this gives the so-called *eigenfaces* as visualized in Fig. 3.11

Just like PCA, auto-encoders are a data-driven technique as well. They are trained without any supervision to compress the input data in a lower-dimensional space from which they can

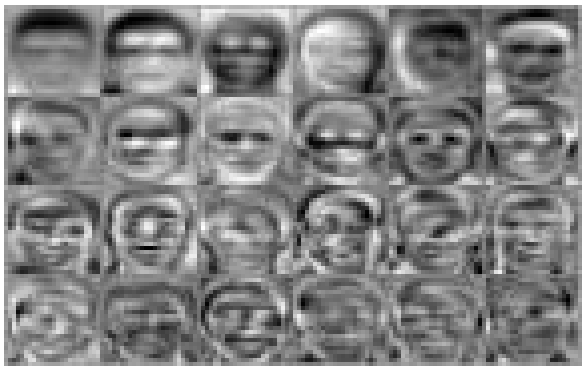

Figure 3.11: The eigenvectors of the covariance matrix of a set of images containing faces: the so-called eigenfaces.

still reconstruct the original input. The forward pass of the auto-encoder as defined in (3.12) shows a lot similarity with both the filtering in (3.13) and dimension reduction in (3.14). In fact, we can cover both techniques with specific choices of weights and activation functions. In case of the convolution we can take as weights $W := \hat{K}^{\frac{1}{2}} U_F^\top$ and as activation function we take the identity. In case of PCA we also take for the activation functions the identity and for the weights we take $W := U_m^\top$. Note that this means we reduce the number of neurons in the hidden layer to m.

In conclusion, we saw that on the one hand auto-encoders can be related to dimensionality reduction techniques like PCA and on the other hand they can be related to Fourier-related filtering techniques such as convolution. In both cases, the weights represented a certain structure in the input data in the form of an orthogonal basis. This basis was then used in order to either compress the input data or filter it. We therefore conjecture that denoising auto-encoders trained for reconstruction of the clean input will learn eigenvector-like elements from the input data, in order to simultaneously compress and filter the input data. The type of activation functions used (or learned) will determine the structure of the coefficients used in this representation. For example, sigmoid activation functions will try to enforce either 0's or 1's as coefficients, while the ReLU activation function will enforce only positive or 0's as coefficients. In this view, activation function can be seen as a type of regularization, forcing the representation and reconstruction to be of a specific form.

Chapter 4

A variational view on auto-encoders

4.1 Introduction

In the last part of the previous chapter we saw how auto-encoders show a lot of similarities with filtering- and dimensionality reduction methods. Especially denoising auto-encoders that aim to simultaneously reconstruct and denoise the input seem to process their data very similarly to diffusion processes. By also showing the similarity with PCA we made a conjecture that auto-encoders learn the most informative structures in the data in order to reconstruct the input as good as possible, and in the case of DAEs even improve the input.

In Chapter 2 we discussed several types of variational methods for denoising. We saw that variational methods are related to diffusion processes, and that most popular denoising methods arise from special choices of the regularization functional. In Chapter 2 we discussed so-called model-based methods, that use an a-priori model of the data in order to process it. Since the performance of these models heavily depends on the type of input data, many researchers have proposed to incorporate a learning process to decide which model to use. An example of this is a paper by Chen, Wei and Pock [10], where the authors propose a diffusion process for denoising with learnable parameters. They also show that their model corresponds to the minimization of a variational energy consisting of a regularization term and data-fidelity. As discussed in Chapter 2, regularization terms correspond to assumptions on the input data that guide the optimization process. If this regularization functional is learned in [10], then we can conclude that their model learns structure from the data and uses this to obtain an optimized diffusion process for denoising.

As already evident from the reasoning above, there again are many similarities between the optimized diffusion process from [10] and the auto-encoders as introduced in the previous chapter. In this chapter we will show that the mathematical formulations of both models fortify the arguments above, and that we can in fact make a direct relation between the two. To this extent, we will first introduce the general ideas behind the model in [10]. We show how this model can related to regularization methods. Next, we will show how auto-encoders can be related to regularization and show that they effectively solve a bi-level optimization problem. Finally, we

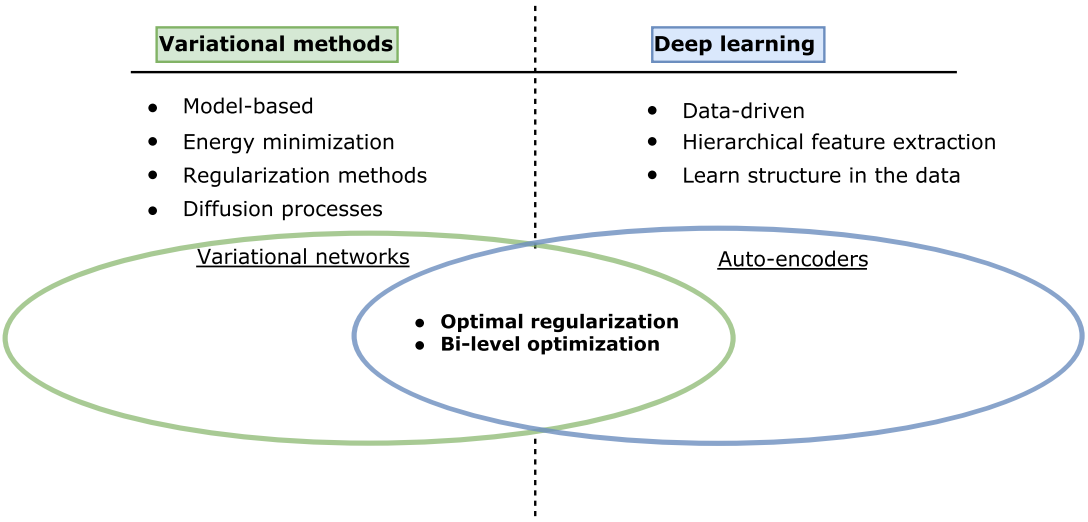

Figure 4.1: An overview of the work in this chapter. We look at the connections between the model-based variational methods and data-driven deep learning by linking variational networks to auto-encoders. Using denoising auto-encoders and skip-connection we show these methods are related through bi-level optimization and optimal regularization.

show how variational networks are related to auto-encoders by intepretating the model in [10] as a special type of an auto-encoder. Comparison with other AE-based models for denoising allows us to point out downsides of their model and propose directions for future research. To guide the reader, we have summarized the work in this chapter in Fig. 4.1.

4.2 The variational network

4.2.1 Basic principles

In recent papers, see e.g. [46], models similar to the one proposed in [10] are referred to as *variational networks* (VNs), emphasizing their connection with both variational methods and neural networks. Consequently, we will also refer to these models as variational networks in the rest of this work. In this section we will mainly follow [10] to present the basics of these variational networks and emphasize the relation with the minimization of an energy in the last part. We will start off with the motivation behind the proposed VNs.

In Chapter 2 we saw that diffusion processes arise naturally from the minimization procedure for a variational energy. Recall that a general local model we discussed was the Perona-Malik [15] anisotropic diffusion process, given as the time-dependent PDE

$$\frac{\partial u}{\partial t} = \boldsymbol{\nabla} \cdot \left(g\left(|\boldsymbol{\nabla} u| \right) \boldsymbol{\nabla} u \right). \tag{4.1}$$

Here the function $g : \mathbb{R} \to \mathbb{R}$ acts as the diffusion coefficient and determines the type of diffusion. In [10] an even more general version of this model is proposed, where most parameters in fact can be tuned. To see what motivated the proposed model, let us first discretize (4.1). For convenience we reshape images into vectors, such that $u \in \mathrm{R}^{N^2}$. Using this vector notation we

now have for the finite difference (FD) approximations of the derivatives $K_x u, K_x u \in \mathrm{R}^{N^2}$. Here the matrices $K_x, K_y \in \mathbb{R}^{N^2 \times N^2}$ contain the FD schemes for the derivatives. Moreover, we define $(\boldsymbol{\nabla} u)_p := [(K_x u)_p, (K_y u)_p] \in \mathbb{R}^2$ as the FD approximations of the gradient at the coordinate corresponding to the p-th element of the reshaped image vector u. The discretization now becomes

$$\frac{u_{j+1} - u_j}{\Delta t} = -K_x^\top (g\left(|\boldsymbol{\nabla} u|\right) \odot K_x u) - K_y^\top (g\left(|\boldsymbol{\nabla} u|\right) \odot K_y u). \tag{4.2}$$

Here $\odot$ denotes component-wise multiplication, u_j denotes the solution at the j-th time-step, and Δt denotes the size of the time-step. We now define $\phi_i(\boldsymbol{\nabla} u) := g(|\boldsymbol{\nabla} u|) \odot K_i u$, for $i \in \{x, y\}$. For simplicity, the authors in [10] ignore the coupled relationship between the two partial derivatives in the diffusion coefficient by assuming $\phi_i(\boldsymbol{\nabla} u) = \phi_i(K_i u)$ for $i \in \{x, y\}$. Note that this is an anisotropy assumption, resulting in a diffusion coefficient that is different for the x- and y-directions and depends *only* on the magnitude of the derivatives in both directions separately. With this assumptions we can decouple the diffusion processes completely and write (4.2) as

$$\frac{u_{j+1} - u_j}{\Delta t} = -\sum_{i \in \{x,y\}} {K_i^j}^\top \phi_i(K_i^j u_j). \tag{4.3}$$

Here the left multiplication by the matrices K_i^j and ${K_i^j}^\top$ corresponds respectively to the convolutions with kernels k_i^j and $\overline{k}_i^j$, the same kernel rotated by 180 degrees. Now a generalization of this diffusion process is made by using not just two, but a total of N_k kernels. On top of this, they introduce a different diffusion function ϕ_i for *every* one of these kernels. Both the kernels and the diffusion functions are even chosen time-dependent yielding an even more general model. Finally, a reaction term $A^*(Au - f)$ is added with corresponding coefficient $\mu \in \mathbb{R}$. The resulting diffusion process is as follows:

$$\frac{u_{j+1} - u_j}{\Delta t} = -\sum_{i=1}^{N_k} {K_i^j}^\top \phi_i^j(K_i^j u_j) - \mu A^*(Au_j - f). \tag{4.4}$$

Note the striking equivalence between (4.4) and the previously derived gradient flow (2.4). Not suprisingly, it is shown that (4.4) is indeed one gradient descent step of the time-dependent energy functional

$$E^j = \sum_{i=1}^{N_k} J_i^j(u_k) + D(Au_j, f). \tag{4.5}$$

Here J_i^j denote the regularization components defined as $J_i^j(u) := \sum_{p=1}^{N^2} \psi_i^j \left(\left(K_i^j u_j \right)_p \right)$, the data-fidelity term as $D(Au_j, f) := \|Au_j - f\|_2^2$ and $\phi(\cdot) \equiv \psi'(\cdot)$.

Just like in Chapter 2 we take $A = I$ for denoising purposes. By comparing (4.5) with (2.6) we can conclude that $J := \sum_i J_i$ is the regularizer in this energy. Now, in [10] all kernels K_i^j and diffusion functions ϕ_i^j and even the parameter μ are trainable; they are optimized based on the outcome of several stages of the diffusion process (4.4). In conclusion, this means that by optimizing all parameters in the diffusion model (4.4), it actually *learns an optimal regularizer*.

Recall from the discussion in Section 2.2 that regularizers incorporate assumptions on the structure of the data in the variational models. Since the trainable diffusion process (4.4) learns the optimal regularization energy (4.5), it must be able to learn structure in the data. Since this was exactly the conclusion on auto-encoders, we raise the question: *how are auto-encoders related to*

regularization? Before investigating this question, we look how the variational network relates to the popular TV regularizer. More specifically, we show that is in principle possible that the diffusion process learns exactly this regularizer, justifying the statement that variational networks learn optimal regularizers. As we will see in the next section, this will also give us insight on the role of the diffusion functions ϕ_i.

4.2.2 Variational networks learn optimal regularizers

Let us consider the well-known total variation (TV) regularization functional and see how it relates to (4.5). The generalized definition for the TV is given by

$$\mathrm{TV}_\gamma(u) := \int_\Omega \gamma(\boldsymbol{\nabla} u) dx, \tag{4.6}$$

where $\gamma : \mathbb{R}^d \to \mathbb{R}$ is a convex, positively 1-homogeneous functional, $\gamma(x) > 0$ for $x \neq 0$, and $u \in W^{1,1}(\Omega)$. Discretizing in space gives the approximation

$$\mathrm{TV}_\gamma^d(u) = \sum_{p=1}^{N^2} \gamma((\boldsymbol{\nabla} u)_p). \tag{4.7}$$

We can see the similarity between the first term on the right-hand side of (4.5) and (4.7). To be able to make this more concrete we will have to write $\gamma(\boldsymbol{\nabla} u)$ as a point-wise function, like the functions $\psi_i(\cdot)$ in (4.5). Hence let us assume γ is a norm-like function, and we can write

$$\gamma(z) = \tilde{\gamma}(z_1) + \cdots + \tilde{\gamma}(z_d).$$

Note that this is the same anisotropicity assumption as made in the previous section to derive the variational network. In case of the discrete approximation (4.7), and $d = 2$ we get

$$TV_\gamma^d(u) = \sum_{i \in \{x.y\}} \sum_{p=1}^{N^2} \tilde{\gamma}((K_i u)_p).$$

In the last equation we again defined K_x, K_y as the finite difference approximations of the gradient operator in the x- and y-direction, respectively. We have now arrived at the discrete approximation of the anisotropic TV functional exactly in the form of the regularizer in (4.5).

We see that the functions $\tilde{\gamma}'(\cdot)$ that will be learned in this setting, correspond directly to the norm $\tilde{\gamma}(\cdot)$ in the TV functional. Therefore we conjecture that learning the function ϕ_i^j in (4.4) amounts to learning the type of norm being used in the corresponding regularizer in (4.5). In Fig. 4.2 we can see four learned diffusion functions of a variational network trained for denoising (left graphs) and their corresponding anti-derivatives (right graph). The left graphs correspond to $\gamma'(\cdot)$ in this example, and the right graphs corresponds to $\gamma(\cdot)$. These examples show that a variational network can indeed learn norm-like, convex functions. Next to this, it also shows that it could learn concave functions and a combination of both.

The kernels K_i act as filters, emphasizing certain structures in the image. Hence we conjecture that learning the kernels in (4.4) amounts to learning what structures in the image are important to preserve. In case of model-based approaches such as variational methods, both kernels and activation functions are chosen fixed and incorporate a-priori information on the solution. For

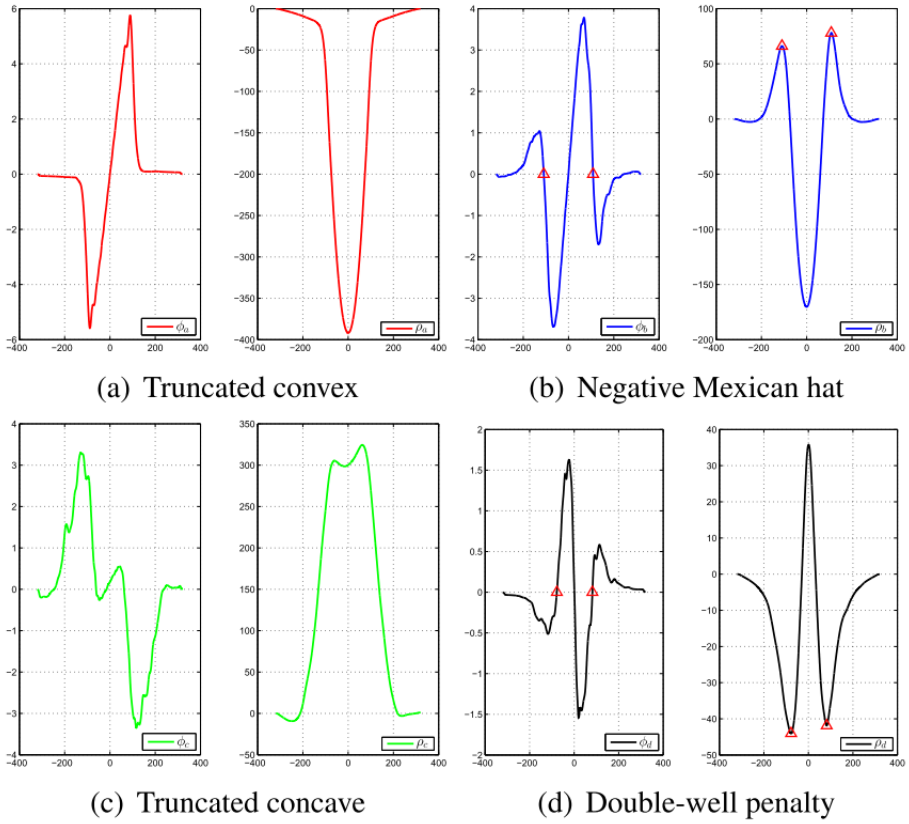

Figure 4.2: Examples of functions learned by a variational network. Graphs on the left correspond to the diffusion functions and graphs on the right correspond to the corresponding anti-derivatives. From the graphs on the right we see that norm-like, convex functions, concave functions and a combination of both are learned by this network. This image is copied from [10].

data-driven methods such as this variational network these can be learned, allowing for much more flexibility. Since this seems very similar to what auto-encoders do, we will investigate the relation between auto-encoders and regularizer in the next section.

4.3 Auto-encoders as regularizers

In this section we will explore how auto-encoders are related to regularizers. We will make use of several ideas presented in [11]–[13]. Besides the relation with the anisotropic TV functional, (4.4) shows strong resemblance with the forward pass of an auto-encoder. Since (4.4) can simply be rewritten as a gradient flow on the energy E by taking $\Delta t = 1$,

$$u_{j+1} = u_j - \partial_u E, \quad u_0 = \tilde{u}, \tag{4.8}$$

the question is actually how auto-encoders are related to gradient flows in general. Note that we can take any step-size we want since we can freely scale the right-hand side of (4.4). From the answer to this question it will then follow naturally how auto-encoders are related to regularization in variational methods.

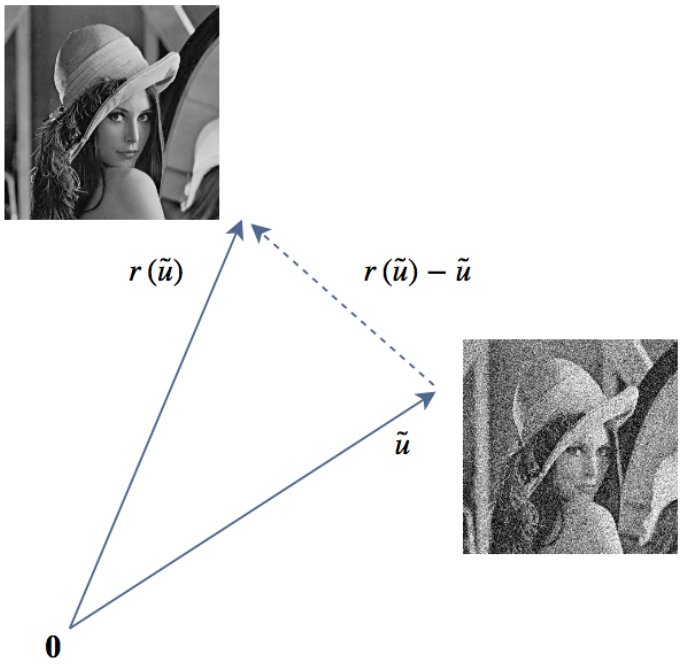

Figure 4.3: A DAE learns the vector field $r(u) - u$ (dotted line). Here 0 is the reference vector, $\tilde{u}$ is the noisy input image and the vector $r(\tilde{u})$ is the (clean) reconstruction by the DAE.

4.3.1 Gradient flows and auto-encoders

Let us suppose we have trained a denoising auto-encoder and the result is the reconstruction function $r(u;\theta)$ as the forward pass of the auto-encoder (see Section 3.4.2). For notational convenience we will omit the parameters in the reconstruction function, and simply write $r(u;\theta) = r(u)$. We can derive a corresponding vector field $G(u)$ that points from the input vector u towards the reconstruction $r(u)$ as follows:

$$G(u) = r(u) - u. \tag{4.9}$$

Hence, the vector field $G(u)$ represents the linear transformation applied to the input u to obtain the reconstruction $r(u)$ [11]–[13] (see Fig. 4.3 for a visualization). The vector field G is the gradient field of some energy functional whenever it satisfies the Poincaré integrability criterion. Loosely speaking, this requires symmetry in the partial derivatives. As shown in [47], an auto-encoder with tied-weights and linear decoder satisfies this criterion. Recall that this type of auto-encoder is defined by the reconstruction formula

$$r(u) = W^\top \phi_e(Wu). \tag{4.10}$$

Since we use the convention that we *minimize* the energy whenever we reconstruct a clean image from a noisy input, we will have that G points in the direction of the negative gradient of the energy E, that is,

$$G(u) \sim -\partial_u E.$$

From (4.9) it now follows that for a general tied-weight auto-encoder defined as in (4.10), there is an energy for which we have the connection

$$r(u) - u \sim -\partial_u E.$$

Rewriting this, we see that for a corrupted sample $\tilde{u}$ we have that there is an energy E for which one step of the gradient flow corresponds exactly to the reconstruction in a tied-weight auto-encoder:

$$W^\top \phi_e(W\tilde{u}) = \tilde{u} - \partial_u E(\tilde{u}; W). \tag{4.11}$$

Note that the reconstruction function of the auto-encoder and the corresponding energy are both parametrized by the *same* parameters W. Now that we know there exists an energy for which (4.11) holds, the questions that remains is what exactly this energy E looks like.

4.3.2 The energy functional and skip-connections

The energy functional. In what follows next, we will see that there is a general energy for which (4.11) is satisfied. Next to this, we see that - besides a normal tied-weight auto-encoder - there is in fact another type of auto-encoder that captures (4.11) as its forward pass more naturally.

We will first consider a normal tied-weight auto-encoder. Let us define the energy

$$E(u) = \sum_{p=1}^{N^2} \psi((Wu)_p) + \frac{1}{2} \|u\|_2^2 . \tag{4.12}$$

Here ψ is applied point-wise and N^2 denotes the number of pixels. Defining the point-wise function $\phi(\cdot) := -\psi'(\cdot)$ then yields the gradient of (4.12) with respect to u evaluated at $\tilde{u}$ as

$$\partial_u E(\tilde{u}) = -W^\top \phi(W\tilde{u}) + \tilde{u}.$$

Plugging this into the right-hand side of (4.11) gives

$$r(\tilde{u}) = W^\top \phi(W\tilde{u}). \tag{4.13}$$

We see that we recover exactly the reconstruction formula (4.10) of a tied-weight auto-encoder with linear decoder.

Skip-connection auto-encoders. If we remove the second term from the right-hand side in (4.12) we get the energy

$$E(u) = \sum_{p=1}^{N^2} \psi((Wu)_p). \tag{4.14}$$

If we perform gradient descent on this functional we recover an auto-encoder with so-called *skip-connections*, defined by the reconstruction formula

$$r(u) = u + W^\top \phi(Wu).$$

Here we have again defined the activation functions as $\phi(\cdot) = -\psi(\cdot)$. Skip-connections were popularized by researchers from Microsoft through the introduction of their revolutionary deep

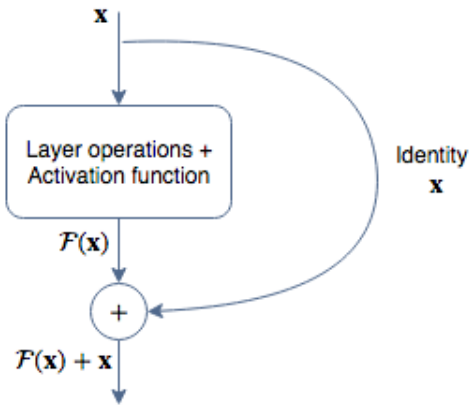

Figure 4.4: A visualization of a layer with a skip-connection. Image edited from [34].

residual network (ResNet) [34]. In short, skip-connections add identity mappings parallel to usual layers via shortcuts. This allows the networks to learn residual functions with reference to the input, rather than unreferenced functions (see Fig. 4.4). Different arguments are given in literature why the introduction of skip-connections makes sense and improves the results. For this application the main arguments are that it naturally arises in the gradient descent scheme (4.8) we try to capture, and that the residual mapping is easier to learn than the full mapping without skip-connections. The latter originates from the fact that the noisy input is already quite a good estimate of the ground-truth solution; most of the structure is already present in the input, it is only corrupted by noise. Therefore it seems more reasonable to learn a reconstruction with the noisy input as the starting reference point than to reconstruct the clean image from scratch.

For this reason, skip-connections seem like a natural way to go if our task is reconstruction. Without explicitly mentioning it, the network proposed in [10] also uses skip-connections by enrolling gradient descent on the energy (4.5) as a network. From the discussion above, we see that one iteration of (4.8) with the energy (4.14) can be modeled by a skip-connection auto-encoder with tied-weights W and activation functions $\phi(\cdot) := -\psi'(\cdot)$. We will look at this in more detail in Section 4.4.

4.3.3 Auto-encoders learn optimal regularization

In summary, we have now shown how special types of auto-encoders are related to gradient flows. Especially for denoising purposes it seems reasonable to use auto-encoders in combination with skip-connections. Now that we know the exact form of the energy for which the gradient descent scheme can be seen as the forward pass of a tied-weight auto-encoder, we will explore what it means to train the auto-encoder in this context.

As usual with auto-encoders we will try to minimize the reconstruction error with respect to the parameters θ:

$$\min_{\theta} \ \|r(\tilde{u};\theta) - u^*\|_2^2 \,. \tag{4.15}$$

Now, from (4.11) we have that by calculating the reconstruction $r(\cdot)$ we minimize an energy E as well. So in other words, minimizing (4.15) is equivalent to minimizing the error between the result after one gradient descent step on the energy E and the clean image u^*, i.e.

$$\min_{\theta} \ \|\tilde{u} - \partial_u E(\tilde{u}; \theta) - u^*\|_2^2 . \tag{4.16}$$

Since the objective function value in (4.16) is recalculated after each gradient descent step on θ through a forward pass of the AE, **we can see this process as the minimization scheme for the bi-level optimization problem**

$$\boxed{\begin{aligned} &\min_{\theta} \quad \|y_\theta - u^*\|_2^2 \\ &\text{s.t.} \quad y_\theta \in \operatorname{argmin}_u E(u; \theta). \end{aligned}} \tag{4.17}$$

Here E is of the form (4.12) and $\theta = W$. So in case of training the AE we alternatingly solve the inner and outer minimization problems in (4.17), by respectively doing a forward pass of the AE and updating its parameters via back-propagation. The inner-level of this optimization problem corresponds to the denoising process based on a variational energy, i.e. it is model-driven. The outer-level corresponds to the minimization of the model parameters based on the reconstruction error, hence it is data-driven.

Another interpretation is that by solving (4.15) we learn the energy, or regularizer, (4.12) that is optimal whenever for all training instances the direction of steepest descent points directly towards the clean images. In dynamical system terminology, this means that all clean images should be attractors of the dynamical system defined by the energy $E(u; \theta)$ at the corresponding corrupted images. We optimize over θ in order to achieve this goal.

4.4 Variational networks and auto-encoders

4.4.1 Convolutional auto-encoders

Now that we have seen what the relation is between gradient flows and auto-encoders, we can proceed to couple the minimization of (4.5) and (4.12) directly. We will use the skip-connection auto-encoders as introduced in Section 4.3.2. As we compare them, we see that the biggest difference is in the weights; (4.5) uses sparse matrices corresponding to convolution, whereas (4.12) uses normal matrices. To make the direct analogy we need to introduce so-called *convolutional auto-encoders* (CAEs); an architecture that combines the efficient feature extraction of a CNN with the unsupervised learning of an AE. CAEs encode the input with a number of convolutions followed up with an activation function. Afterwards they decode using deconvolution. Here deconvolution of convolution with a filter k is defined as a convolution with the same filter, rotated 180 degrees. A CAE in combination with a skip-connection is visualized in Fig. 4.5. From Fig. 4.5 we can also easily see the connection with one iteration of (4.4).

4.4.2 The lack of scale in variational networks

Now that we have interpreted the variational network as introduced in [10] as a convolutional denoising auto-encoder, we can also see a downside of the method. Recall from Chapter 3 that in deep (C)NNs the level of abstraction and the size of the extracted features increases while we

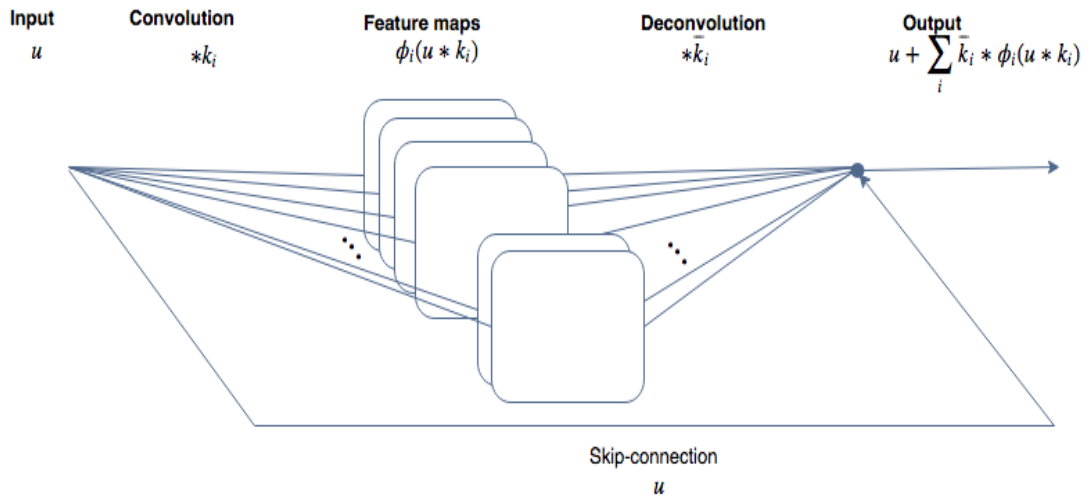

Figure 4.5: A convolutional auto-encoder in combination with a skip-connection, as introduced in Section 4.4.

go deeper into the network. In the VN, there is only one layer for both encoding and decoding, hence the features extracted in the variational network can only be of the same size as the kernels. As a result, the filtering performed by the activation functions can only filter patterns up to a certain size and level of abstraction. In our opinion, a denoising procedure could benefit from adding more scale in the feature extraction, because it will then be able to filter at more than just one level subsequently.

Besides the added benefits of scale, one could also argue that it is natural for an AE to consist of just one layer for both encoding and decoding; they should not encode in too complex features because their goal is to reconstruct from them. For example, suppose we were given the statement "a cat is present in this image" as the very abstract, semantic, representation of an image of a cat. Although this may be enough for a classifier to label the image correctly, it is by no means enough for exact reconstruction. The reason for this is that CNNs are designed specifically to get rid of redundant information such as exact spatial locations; these are simply not necessary for the classification task and therefore can only make performance worse. For AEs, however, it is crucial for good reconstruction quality to have the exact spatial information at their disposal.

In conclusion, adding scale to the feature extraction in the VN may be beneficial, but care has to be taken since it will also remove important information for the reconstruction.

4.4.3 Architectures for semantic segmentation and image restoration

To see how we could do this nicely we shortly jump to the task of semantic segmentation. Semantic segmentation is the task of jointly classifying and segmenting objects in images. A good quality segmentation map is required to have both a high spatial resolution and good localization of the object. The classification task that will be performed jointly, will require a much higher level of abstraction as output in the form of labels. To perform semantic segmentation is therefore a challenging task. Several architectures have been proposed by the deep learning

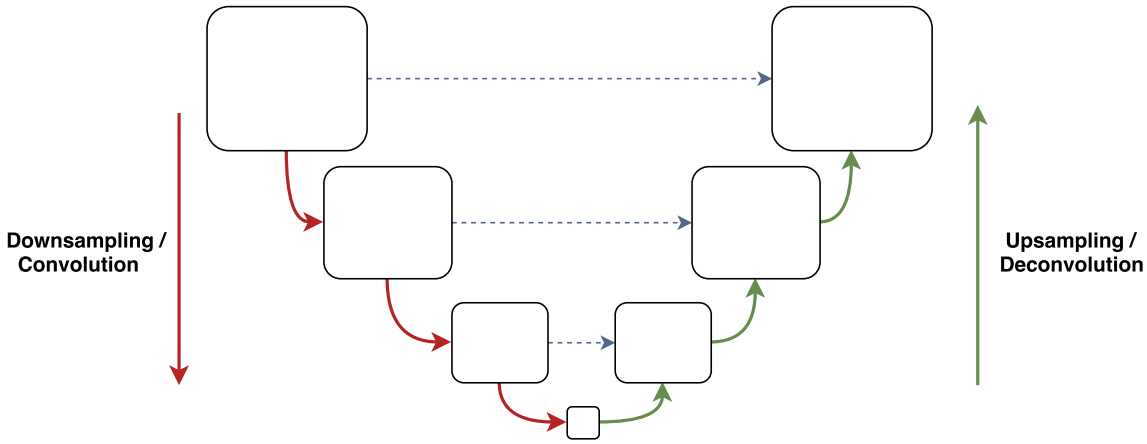

Figure 4.6: Schematic overview of the U-Net and RED-Net architectures. The blue dotted lines represent skip-connections, the solid lines represent the forward passes of (multiple subsequent) layers and the squares represent feature maps.

community to perform this task. Since segmentation can be seen as a special type of reconstruction it comes as no surprise that most proposed architectures show strong resemblance with auto-encoders. In fact, many authors propose to combine CNNs with AE to perform the joint classification and segmentation. As discussed above, bluntly combining these two architecture is unnatural and will probably lead to bad results. Hence, measures have to be taken to make the combination work.

In [48], classic CNNs are *convolutionized* and followed up with upsampling layers in order to obtain pixel-level outputs in the form of semantic segmentation masks. The convolutionalization is obtained by transforming the fully connected (FC) layers to convolutional layers. The authors recognize the same challenge as discussed above: CNNs output coarse images with a high-level of abstraction and therefore require some upsampling procedure in order to make them useful for tasks like (semantic) segmentation. In [48], it is empirically shown that the upsampling quality is improved by using information from more than just the last layer through the use of skip-connections. Other examples, where skip-connections are used to ease the reconstruction process, are the U-Net architecture, as introduced in [49], and RED-Net, a very deep Residual Encoder-Decoder network as introduced in [50]. All authors argue that the use of skip-connections has two big benefits: 1) from an image restoration perspective skip-connections transfer image details from lower layers to higher layers, easing the reconstruction, and 2) from a deep learning perspective skip-connections allow a much easier back-propagation of the gradients, improving the quality of the training significantly. Both U-Net and RED-Net use multiple convolution layers for encoding and decoding combined with symmetric skip-connections, see Fig. 4.6 for a schematic visualization.

4.4.4 Understanding the difficulty and importance of upsampling

In the previous paragraphs we saw intuitively why upsampling from a low-dimensional space is hard. Mathematically speaking, this is due to the fact that the reconstruction problem is very ill-posed; there is too little data available to perform a good reconstruction. To see why, let us define the encoding of the input $x \in \mathbb{R}^n$ as the function $\mathcal{F}(x; W_e)$, parametrized by the weights

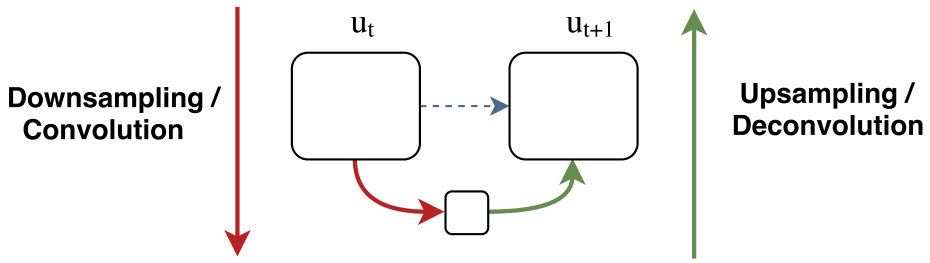

Figure 4.7: A variational network visualized as a residual encoder-decoder architecture deployed in among others the U-Net and RED-Net.

W_e. Defining the codes $\mathcal{F}(x)$ as y, the reconstruction problem can be stated as finding $\tilde{x}$ such that

$$\mathcal{F}(\tilde{x}) = y. \tag{4.18}$$

Recall that auto-encoders also solve a similar problem, in which case $\tilde{x}$ is the result of the forward pass. If $\mathcal{F}$ downsamples the input similar to a CNN, then we will have that $\mathcal{F}(x) \in \mathbb{R}^h$ for $h \ll n$. Hence, even in the simple case of a linear $\mathcal{F}$ solving (4.18) amounts to solving an ill-posed system: we search for n unknowns with only h equations available ($h \ll n$). If our goal is to reconstruct an input from codes produced by a CNN, then $\mathcal{F}$ is a highly nonlinear mapping, making matters even worse.

There has been a lot research into the construction of models to invert codes in deep learning networks. One of the most influential papers on this subject by Matthew Zeiler and Rob Fergus [5] introduced the idea of using a deconvolutional network to tackle the inversion problem directly. They also show the importance and necessity of good inversion methods: if we can invert activation from the layers in a CNN then we can interpret these activations in image space. This allowed them to analyze and improve layers, and even full architectures, resulting in the 1$^{\text{st}}$ place in the ImageNet competition in 2013.

In this light, CAEs can be interpreted as end-to-end learnable architectures that perform high-level feature extraction followed by an inversion. To make the inversion tractable, skip-connections are used to inject more fine-structured data. Mathematically speaking they add a reference point to the optimization problem (4.18), as discussed in Section 4.3 as well. Since we assume that the noisy input image is already rather close to its clean counterpart in terms of their norm, this adds stability to the gradients used for the training process and makes the sought-after inversion mapping less nonlinear. The U-Net and RED-Net architectures as visualized in Fig. 4.6 consist of multiple stages of encoding and decoding. While the stages in the encoding can be seen as feature extraction on different levels, it can be seen as multi-staged reconstruction procedure in the decoding. At every stage a reconstruction process is solved starting from a reference point provided by the skip-connection to the corresponding encoding stages. By performing subsequent reconstruction at the different levels, the network eventually arrives at a reconstruction of the input in image space.

4.4.5 Relation with variational networks

From the discussion above, our statement regarding the lack of scale in variational networks becomes even clearer. From Fig. 4.7 we can easily see that residual encoder-decoder architectures used in U-Net and RED-Net are a generalized version of one iteration of the variational network. From this we can also conclude that the VN lacks scale; it only uses an auto-encoder with 1 layer for encoding and 1 layer for decoding. Because the kernels are of fixed size, this means that the extracted features will be of fixed size. Therefore a VN can never learn to take the global context of an image into account, and thus lacks scale.

Looking back at (4.17), we can now also see that the difference between variational networks and U-Net and RED-Net-like architectures is that the former perform multiple iterations to solve the inner problem, after which it switches to the outer problem. The iterations corresponds to the several diffusion steps in the model. On the other hand, the U-Net and RED-Net only perform one iteration on the inner problem and then switch to the outer problem.

4.5 Summary of results

In this chapter we have explored the similarities between variational networks, regularization functionals and auto-encoders.

First, we have shown that variational networks can be interpreted as models that learn optimal regularization. Here we saw that the diffusion functions are related to the choice of norm in the regularization and that the weights determine what type of structure we penalize for the denoising process. We also saw that the VN can in principle learn classical regularization methods such as TV. The model is, however, limited to anisotropic diffusion because of the assumptions made during the derivation.

Next, we have shown that auto-encoders are related to regularization methods as well under the condition of tied-weights. More specifically, the reconstruction functions of a (skip-connection) auto-encoder can be interpreted as one gradient descent step on an energy. This allowed us to interpret the training process in an auto-encoder as an alternating minimization scheme for a bi-level optimization problem. The inner-level of this optimization problem corresponds to the denoising process based on a variational energy, i.e. it is model-driven. The outer-level corresponds to the minimization of the model parameters based on the reconstruction error, hence it is data-driven. We conclude that auto-encoders are very suitable to combine model-driven and data-driven approaches for image restoration.

Finally, we show the connection between auto-encoders and variational networks by interpreting one iteration of the latter as a special case of the former: a convolutional denoising auto-encoder with one hidden layer and a skip-connection. Using this interpretation we compared the variational network with other AE-based models such as the U-Net and RED-net, from which we concluded that the VN lacks scale. Although there are several iterations in the VN to complete the denoising, all diffusion processes in the iterations are started on the same resolution and can therefore not take complex features into account.

Chapter 5

Classificating circulating tumor cells with a deep CNN

5.1 Introduction

Why it is important. Tumor metastasis is the spread of cancer from one organ to another by circulating tumor cells (CTCs) detached from primary tumors traveling in the peripheral blood of the patient. It is generally accepted as the main cause of death in cancer patients, hence it is essential to develop good and accurate techniques to determine the level of metastasis [51]. Detection and enumeration of CTCs has been shown to be an important diagnostic tool for this task, since their presence has been associated with a reduced probability of survival and has a direct relation with metastasis [52]–[57]. However, it is extremely labor-intensive and time-consuming due to the relatively low fraction of only 1-10 CTCs per mL blood compared to several million white blood cells (WBCs) [58]. This naturally resulted in an increase in the request for systems designed specifically for automated extraction and detection of circulating tumor cells from blood samples. The heterogeneity in the morphology of CTC candidates renders the classification task extremely difficult for humans, let alone automated systems. Recently, several researchers have proposed to use machine learning techniques to tackle the classification problem (see e.g. [59]–[66]) because of the demonstrated success of these techniques on similar tasks.

Our approach. Our approach to design a system for automated CTC classification involves a deep neural network, more specifically a deep convolutional neural network. We have trained the CNN on a set of thumbnail images containing either a CTC or some other cell-like object. These thumbnails were extracted with ACCEPT [67], a tool similar to CellSearch [68]. The extracted CTC-candidates were labeled by an expert reviewer. As opposed to some other methods, the training will allow our system to directly learn from the data what features are important for classification and should thus be extracted from the images.

5.2 Materials and methods

Dataset and preprocessing. The data used in this chapter is obtained from the medical cell biophysics (MCBP) group at the University of Twente. Here ferrofluids with so-called fluorescent

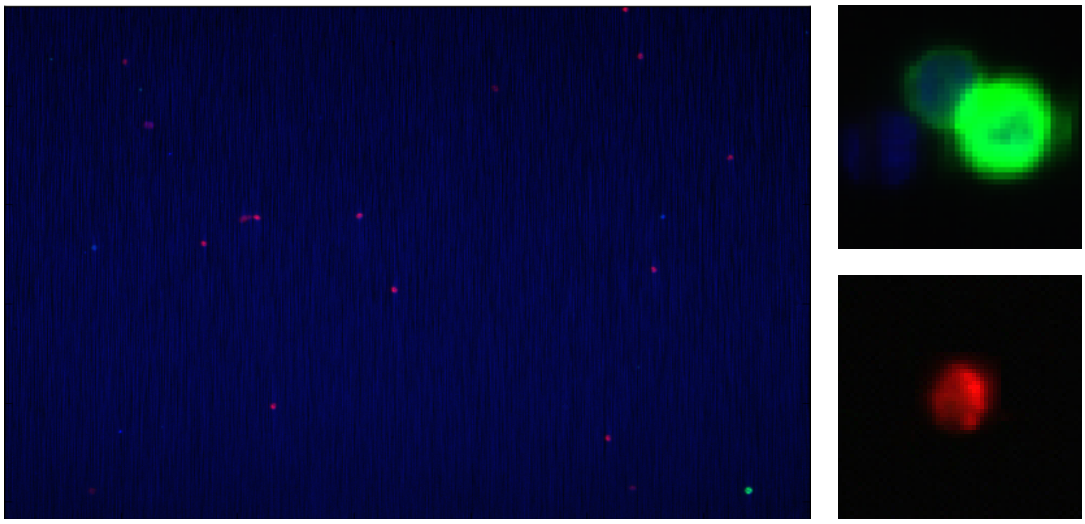

Figure 5.1: Example of a cartridge (left), CTC (top-right) and WBC (bottom-right) obtained with high-throughput fluorescence microscopy as explained in Section 5.2.

biomarkers were used to highlight cells of specific type or origin. The different markers result in different fluorescent colors in the final image. In this case four markers were used: an epithelial cell adhesion molecule (EpCAM), a DNA marker (DAPI-DNA), a cytokeratin/phycoerythrin (PE-CK) marker and CD45-APC. Roughly speaking, the DNA marker will highlight the nucleus of a cell, the cytokeratin marker will highlight cell bodies other than those of white blood cells, and CD45-APC will highlight white blood cells. The image obtained using fluorescence microscopy has four channels and many cells in it. Each of the channels corresponds to the activity of a different biomarker. ACCEPT then extracted thumbnails by cropping a square region around each cell-like object in the cartridge. Samples from this process are shown in Fig. 5.1.

Because the original fluorescence microscopy images are stored in a format not suitable for direct integration with the deep learning toolbox Caffe, we have to perform some preprocessing steps first. First of all we apply symmetrical padding and centered cropping with Matlab in order to obtain images of fixed size 60×60. Simultaneously, we scale the images from 12-bit to 8-bit, resulting in intensity values in the range of 0 to 255. Finally, we apply a soft filtering on the whole dataset, keeping only the thumbnails with activity likely corresponding to cells. Especially this last step significantly increases performance. We attribute this performance gain to the presence of a large number of both noisy images and segmentations of background artifacts in the original raw dataset. The filtering is based on the *activity* in the thumbnails, defined as the mean intensity over pixels exceeding an intensity value of 60. We remove all thumbnails with an activity lower than or equal to 75 from the raw dataset. The final results are saved as TIFF images, making them readable by the ImageData layer in Caffe. The resulting cleaned dataset contains a total of 15143 thumbnails, of which 7571 are labeled as CTC (1) and 7572 as non-CTC (0). The labeling was done by an expert reviewer. The non-CTC thumbnails consist of an evenly distributed set of white blood cells and tumor-cell lookalikes that are hard to separate from real CTCs. These cells were obtained from the CellSearch system as the cells that were manually rejected by the user after being presented by the system as possible CTC candidates. We partition the final dataset in three subsets: a training-, test- and validation set containing

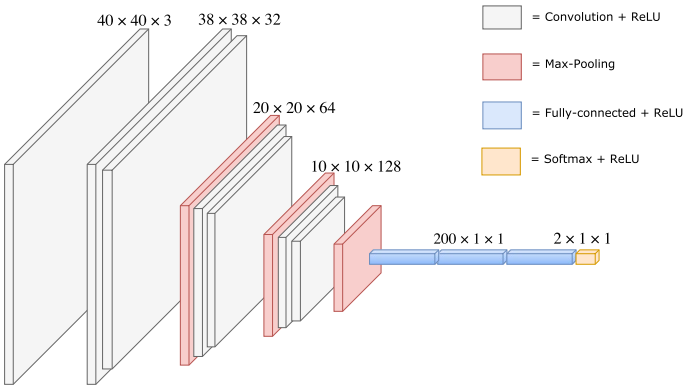

Figure 5.2: A visualization of the architecture used for CTC classification. The input to the network are thumbnails of size 60×60, with three channels. These thumbnails are then randomly cropped to 40×40-sized thumbnails, and then passed as input to the first convolutional layer. Each convolution and fully connected layer is followed up with a ReLU nonlinearity. The result after the convolution layers is sketched in gray, after the max-pooling layers in red, after the fully-connected layer in blue, and after the softmax layer in yellow. The size of the result after the first layer of the blocks is written on top.

respectively a total number of 12116 ($\approx 80\%$), 1511 ($\approx 10\%$) and 1516 ($\approx 10\%$) cells in the same ratio as the original dataset.

Network architecture. The architecture of the classification network is inspired by that of the VGG-net [69]. We used the same network as in [70] for single-cell phenotype classification, see Fig. 5.2 for a visualization. The architecture consists of a total number of 13 layers: 3 blocks of Conv-ReLU-Conv-ReLU-MaxPool, followed up by 3 FC-ReLU blocks and finally 1 softmax layer. All convolutions use filters of size 3×3 and stride 1, the max-pooling uses a receptive field of 2×2 with a stride of 2, and all fully-connected layers consist of 200 neurons.

Implementation. We implemented the network as visualized in Fig. 5.2 in the Caffe framework for deep learning [71]. For this task we favor Caffe over toolboxes like Tensorflow and Theano because it is user-friendly, highly optimized for vision tasks and widely accepted by the computer vision community. On top of this, most state-of-the-art networks have been implemented in Caffe and are available through their model-zoo.

The initial scaling, padding and cropping described above is implemented in Matlab. We also used Matlab to create the text files necessary for the ImageData layer. The filtering applied as preprocessing is implemented in Python. Finally, we use Pycaffe as front-end since it offers most flexibility and allows for easy analysis of trained network parameters by following the tutorials on the Caffe website.

Data augmentation. To artificially enlarge the dataset, we used data augmentation in the form of random rotations, mirroring and random cropping. Since the classification should be rotation invariant, we applied random rotation over the full range of $[-\pi, \pi]$ radians. The

mirroring is applied with a probability of 0.5, and the random cropping operation randomly crops a 40×40 window from the 60×60 input images. The random rotation was implemented using the "CreateDeformation" layer from [72], with linear interpolation and mirroring extrapolation. This layer also comes with the option of random deformation, but since these deformations results in too much artifacts because of the small size of our images we choose not to incorporate this.

Optimizing the network parameters. We use standard momentum stochastic gradient descent in combination with the back-propagation algorithm - as available in Caffe - in order to minimize the cross-entropy classification loss function (see (3.6)) with respect to the network parameters. Here the predicted probabilities are obtained in the last layer of the network using the softmax function (see (3.7)). The network weights are all initialized using the Xavier initialization [6] with a standard deviation of 0.01 and mean 0. The biases in the convolution layers are initialized at a constant value of 0.1. The small positive initialization of the bias is used to force the network to produce non-zero gradients in the first phase of training. This is sometimes helpful to reduce the effect of bad initializations and improve the speed of the training in the very beginning. For the fully-connected layers we set the initial bias values to zero, because these are much less likely to produce zero gradients due to the weighted sum over *all* activations in the previous layer. Hence, even with small initial weights we will get a significant activation and back-propagating gradient.

We start the training with a learning rate of 0.005 and reduce it with a factor of 0.8 every 1000 iterations. In order to obtain stable gradients, we take a relatively large batch size of 512 with a momentum parameter of 0.9. As regularization we use the dropout technique [30] on all fully-connected- and softmax layers with a ratio of 0.5. This results in randomly setting 50% of the weights in these layers to zero after each gradient descent step during training. Furthermore, we add the L_1-norm of the network weights as regularization term to the loss function with a regularization coefficient of 10^{-5}. This parameter is specified as 'weight-decay' in Caffe. Both regularization techniques can be seen as sparsity promoting, as they both force the network to learn a small set of really significant, non-zero weights while setting the rest to zero.

5.3 Results

5.3.1 Thumbnail classification

Performance evaluation. The network was trained on a cluster running on Linux with 32 Intel Xeon E5-2650 CPUs @ 2.60GHz and a Quadro K4000 GPU. We trained for a total of 600 epochs which took about 14 hours. The training progress is visualized in Fig. 5.3. The performance was evaluated with the following metrics: precision ($P = TP/(TP + FP)$), recall ($R = TP/(TP + FN)$) and the F1-score ($F_1 = 2PR/(P + R)$). The plot of the precision-recall curve for the resulting trained network is shown in Fig. 5.4. To generate this plot we varied the bias of the last layer of the network, and calculated the precision and recall at every value. The original bias resulting from the training is 0.079, and the corresponding F_1-score 0.979 is marked as the red cross in the plot Fig. 5.4. We see that the trained network learned a bias very close to highest possible F_1-score 0.981. This value of the F_1-score corresponds to an accuracy of 97.88% correctly classified cells from the test set and 97.67% from the training set.

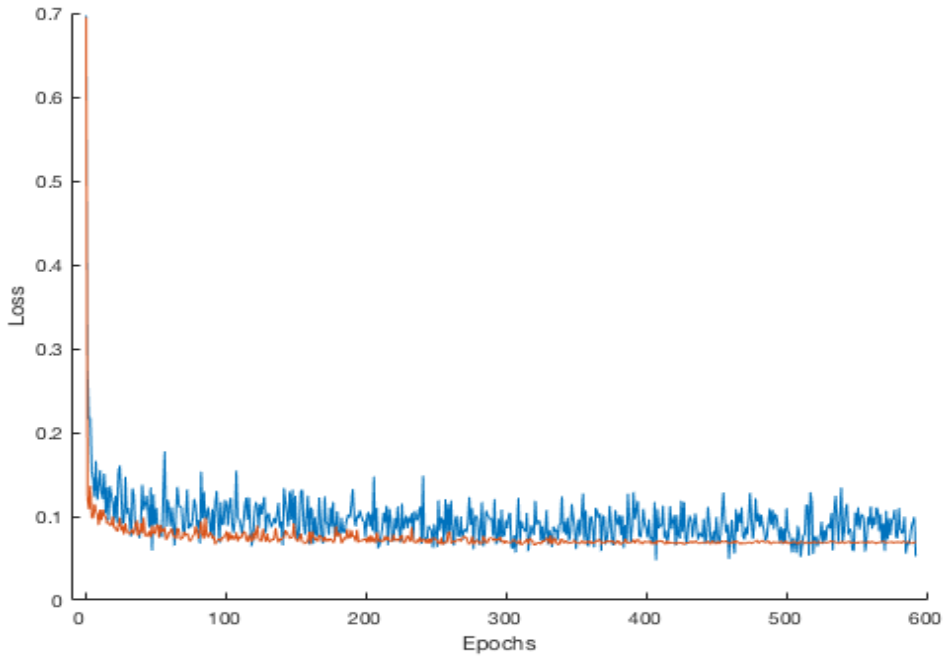

Figure 5.3: The training progress of the CNN, as discussed in Section 5.3.1. The yellow graph is the loss on the validation set and the blue one is the loss on the training set.

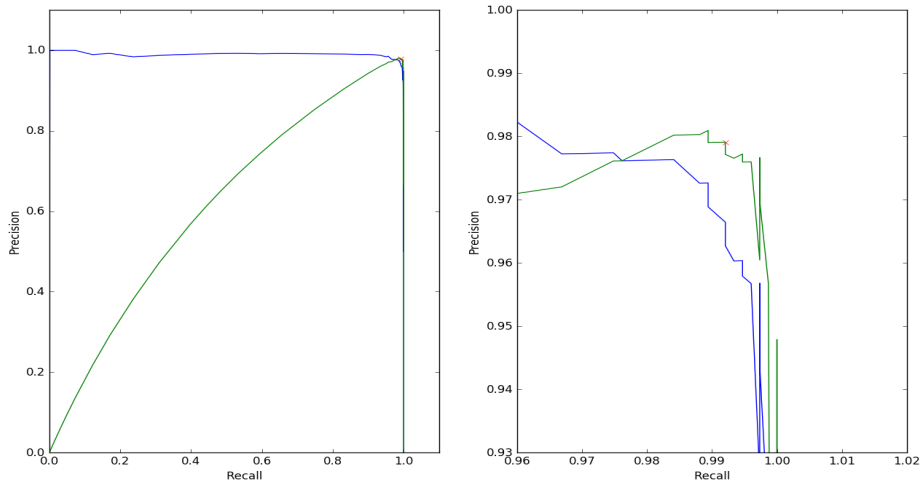

Figure 5.4: The precision-recall (blue) and F_1 (green) curves for the trained network, obtained by varying the bias of the last layer and calculating the three metrics at every value. The original bias resulting from the training is 0.079, and the corresponding F_1-score 0.979 is marked as the red cross in the plot. We see that the trained network learned a bias very close to the bias resulting in the highest possible F_1-score 0.981. The plot on the right is the zoomed-in version of the one on the left.

	Linear SVM	RBF SVM	CNN
Accuracy	0.8830 ± 0.1060 (0.9214)	0.6809 ± 0.0141 (0.6869)	**0.9788**
F_1-score	0.8442 ± 0.2731 (0.9288)	0.7581 ± 0.0060 (0.7560)	**0.9810**

Table 5.1: Classification results on test set. The results for the SVM classifiers are obtained using cross validation with $k = 5$, which resulted in the 95% confidence intervals in the table. The values in the brackets are the maximum values over all splits.

Optimal precision-recall ratio for CTC classification. In the previous paragraph we saw that results after training a network for CTC classification. Since we did not specify any bias towards either CTC or non-CTC and used an evenly distributed dataset, the network has learned to predict equally well the label of a true CTC and a true non-CTC. However, in case of CTC detection this may not reflect the behavior we would like. Recall from Section 5.1 that on average only 1-10 out of a million cells in the blood is truly a CTC. Our system is equally likely to predict the right label for both CTC and non-CTC, hence it is also equally likely to make a wrong prediction on both labels. But since only 1-10 out of a million cells is truly a CTC, we would rather not miss any of them. In machine learning jargon this means we would like our classification system to be *more* sensitive to CTCs than to non-CTCs; we would like to reduce the number of false-negatives (i.e. true CTCs classified as non-CTC) and thus have a recall as close to 1 as possible. Increasing the recall of our system will classify more cells as CTCs (even when they are truly non-CTC), and it will therefore be less likely that we miss any true CTCs. Experts tend to disagree on the definition of a "true CTC" and therefore on the classification, but by increasing the recall we can ensure we are on the safe side in any case. Of course care has to be taken when increasing the recall manually, since this will always be accompanied with a decrease in precision.

Comparison with other methods. To place the results of this chapter into context, we now compare them with results from other methods on the same dataset. For this comparison we take a support vector machine (SVM) classifier with linear kernels and radial basis function (RBF) kernels. We have trained both classifiers on manually extracted cell features. For every cell we have computed the eccentricity, perimeter, mean- and max intensity, size, standard deviation, mass and perimeter to area ratio of segmented objects for all three channels. Moreover, we calculate the overlay between the objects in channels 1 and 2, and channels 3 and 2. The segmentation was done by Leonie Zeune using a nonlinear spectral analysis technique [67] and the features were extracted using the Matlab `regionprops` from the Image Processing toolbox. The performance of the classifiers is measured in the accuracy and F_1-score, and the results are presented in Table 5.1.

5.3.2 Extension to pixel-wise cartridge classification

Now that we have a network that is able to classify thumbnails extracted from a larger cartridge as either CTC or non-CTC, it is natural to try and extend this to classification on the cartridge directly. Our approach to this is to slide a receptive window over the whole cartridge, classifying each thumbnail under the receptive window. See Fig. 5.5 for an illustration. For a cartridge of size $w_c \times h_c$ and receptive window of size $w_{rw} \times h_{rw}$ this will result in a classification mask of size $w_c - w_{rw} + 1 \times h_c - h_{rw} + 1$. Each pixel in the classification mask will be the result of the

classification of the thumbnail centered around that particular pixel in the original cartridge.

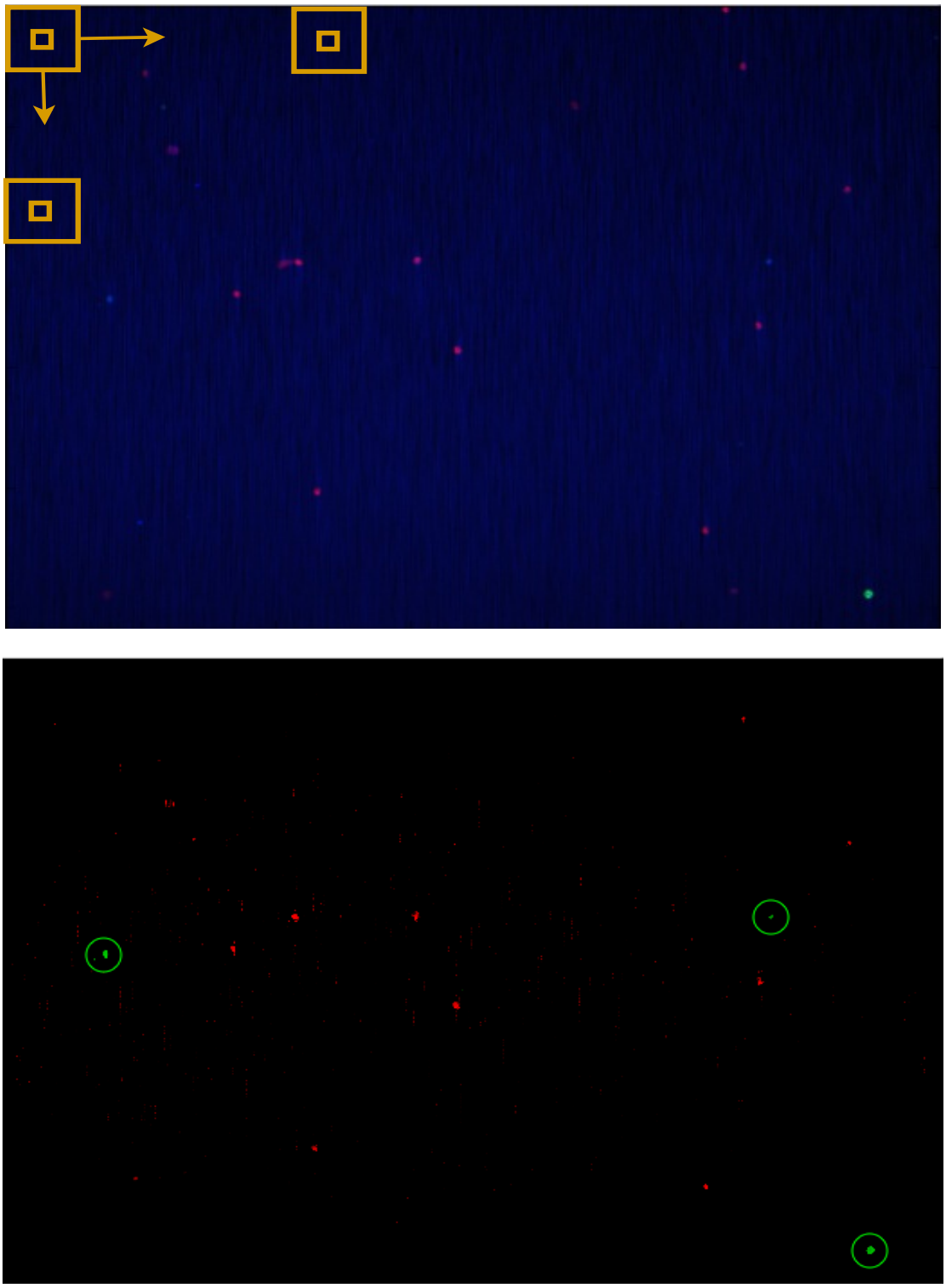

Figure 5.5: Pixel-wise classification on the full cartridge. The sliding window (orange) slides over the whole cartridge, classifying each thumbnail on its way. *Top:* Original cartridge with the sliding receptive window in orange. The small square in the middle of the receptive field denotes the center pixel on which the classification result will be projected to form the classification mask. Colors are as follows: green = CTC marker, blue = CK marker, red = WBC marker. *Bottom:* Result after processing the cartridge with the system. We used an initial intensity threshold of 0.45 and a probability threshold of 0.6 for visualization. Pixels are colored red whenever the predicted probability for non-CTC is highest and green whenever the predicted probability for CTC is highest. The intensities correspond to the highest predicted probability.

Chapter 6

Summary and outlook

6.1 Summary

In this thesis we have developed a deep understanding on neural networks by giving a mathematical definition of neural networks and its variants, analyzing the properties of convolutional neural networks and auto-encoders, and by showing direct connections between variational methods for denoising and (convolutional) denoising auto-encoders. One highlight of the latter is that denoising auto-encoders can be interpreted as a solution procedure to a bi-level optimization problem; they alternatingly denoise an image by minimizing its energy and optimize the energy based on the denoising result. Next to this we have shown the strength of deep learning methods on an application of automatic circulating tumor cell classification, achieving nearly 98% accuracy.

We started with the motivation for this research. Deep learning methods have received an increasing amount of attention in the last 5 years. The development of toolboxes such as Caffe, Theano and TensorFlow made deep learning tools available to the general audience. As a result, the number of applications grew rapidly while the understanding of these methods was lacking behind. After motivating this work, we presented the mathematical background on variational methods for denoising. Here we showed that popular denoising methods can be interpreted as generalizations of diffusion processes.

After this we focused on the theory of neural networks and deep learning. We first gave a brief overview on the history of neural networks. After this we formally described the problem of classification, and explained some difficulties through visual examples. We concluded that neural networks tackle the classification problem by first extracting features from the data with a feature extractor, and next perform classification on the extracted features. We developed the general interpretation of neural networks as the composition of feature extraction and a task-specific module. We showed intuitively that the level of complexity and abstraction of the extracted features increases with the depth of the network. Next we discussed important components of deep learning including the actual optimization process and algorithms, but also difficulties that arise during the training of deep networks. After this we introduced convolutional neural networks through an example of face detection. We gave the formal definitions of convolutional and pooling layers and showed how their combination realizes hierarchical

feature extraction and the desired invariance to small translations and rotations. After the supervised convolutional neural nets we discussed the unsupervised auto-encoders. We started with an introduction to auto-encoders and a variant called the denoising auto-encoder. Next we showed similarities between Fourier-related filtering techniques such as convolution and the dimensionality method called principal component analysis. We showed that both methods can be captured by an auto-encoder with specific choices for the weights and activation functions. We conjectured that denoising auto-encoders learn structure from the input data in the form of eigenvector-like elements in order to simultaneously compress and filter the input data. Next to this, we discussed how activation functions can be seen as a type of regularization, forcing the representation and reconstruction to be of a specific form.

In the next part we considered an optimized diffusion process called the variational network. We first introduced the basic principles and discussed the relation with the minimization of an energy. From this we concluded that this model learns an optimal regularizer, and thus important structures in the data. We justify this statement with an example showing how this model can in general capture an anisotropic TV regularization functional. We next explored how auto-encoders are related to regularization methods. We divided this into two steps: 1) we showed the relationship between gradient flows and auto-encoders and 2) we showed what energy corresponds to this type of gradient flow. Using these two steps we concluded that auto-encoders learn an optimal regularizer and effectively solve a bi-level optimization problem; they alternatingly minimize an energy and optimize the energy based on the quality of the result after one gradient descent step. We proceeded with showing that the variational networks can be interpreted as convolutional auto-encoders with a skip-connection. Using this interpretation, and by comparing with other similar architectures such as U-Net and RED-Net, we were able to show that the variational network lacks scale. More specifically, the variational network can only filter patterns up to a certain size and level of abstraction.

Finally, we presented an application of a deep convolution neural network on the automatic classification of circulating tumor cells from small thumbnails. We first explained the importance of these automatic classification systems due to the low fraction of positive samples, and showed some visual samples from the dataset. Next we explained our approach, what dataset we used and what preprocessing we performed. We discussed our VGG-based architecture and the implementation in Caffe. In order to make the method more robust to small perturbations in the input we have added data augmentations in the form of random rotations over the full range of $[-\pi, \pi]$ radians, mirroring and small random translations. Random rotations over the full range were chosen because the classification of cells should be invariant to rotations. Next we have discussed how we optimized our network; we initialized the weights with a Xavier initialization and used momentum stochastic gradient descent in combination with back-propagation to minimize the cross-entropy classification loss. Furthermore, we have initialized the biases with a small positive value to force the network to produce non-zero gradients in the first phase of training, and applied regularization in the form of an L_1-type regularization of the weights in the energy, and dropout on the fully-connected and softmax layers. We discussed the results obtained with this network in terms of accuracy and $F1$-score, and showed how we can adapt its sensitivity to suit the preferences of the user. Finally, we compared the performance of our network with that of a linear- and radial basis function support vector machine. This showed that our network significantly outperforms both other methods. Finally we discussed a possible

extension of our thumbnail-wise classification to full cartridge classification.

The final conclusion of this work fourfold. First, we conclude that the performance of neural networks can be explained intuitively by looking at the properties of the combinations of layers in the architecture. Second, we conclude that auto-encoders with skip-connection show direct connections with energy-based minimization methods such as variational methods and can be interpreted as solving a bi-level optimization problem. Third, we conclude that variational networks are related to architectures in deep learning such as the U-Net and RED-Net. And fourth, we conclude that deep learning methods are indeed very powerful methods and can be very useful in the automatic detection of cancer cells.

6.2 Outlook

6.2.1 Variational networks

Adding scale. As discussed in Section 4.4.2, the variational network lacks scale when compared to other similar architecture; the kernels used are of fixed size, and since no consecutive convolutions are used, the detected features will be of fixed size. Adding scale to the diffusion iterations, by using U-Net-like architectures with several convolutional encoding and decoding layers, could be an interesting direction of future work. This will allow the detected features to be of larger size, allowing the system to take a more global perspective of the image into account. Moreover, since VNs are directly coupled to learning optimal regularization, it could be interesting to explore how the deeper convolutional auto-encoder architectures such as U-Net and RED-Net are related to energies.

From local- to nonlocal convolutions. Another possibly interesting direction of future work could involve changing the local diffusion in the variational network, U-Net and RED-Net to non-local diffusion as discussed in Section 2.3 and Section 2.4. One work in this direction is [73], where a BM3D-like approach is followed to incorporate non-local dependencies in a denoising network. Lately there has, however, also been an increasing amount of research on graph convolutional networks [74]–[77], in which non-local convolutions are directly implemented. In most works they are implemented as a multiplication by a matrix $UHU^\dagger$. Here the columns of U are chosen as the Fourier eigenvectors of a specific graph and H is a diagonal matrix. The non-local convolution kernels are learned in this setting by learning the diagonal of H. Note that in this way the non-local convolution is just an adaptation of the weight matrices. In contrast to the suggestions in the previous paragraph, this extension will directly be related to (non-local) regularization functionals.

Scale-space analysis. Although not presented in this thesis we have conducted several experiments on the variational networks from [10]. One of these experiments was designed to discover the scale-space structure behind the learned diffusion process. The scale-space of a diffusion process can be seen as the result of the process for different values of the regularization parameter λ, which in this sense is the scale. Another way to obtain similar results is to completely remove the data-fidelity term (i.e. set $\lambda = 0$) and iterate the pure diffusion equation several times until the required scale is obtained. Exactly this last experiment was conducted on a trained VN by adapting the original code from the authors, resulting in the images in Fig. 6.1.

From this it is clear that we indeed obtained a scale-space, and that somehow the system prefers rectangular shapes; it transforms gradually transforms round shapes to rectangular shapes and even turns some humps of noise into squares. We assume that this is due to the anisotropicity assumptions made in the derivation of the networks. In [10] networks are trained on specific noise-levels because transferring gave bad results. Our experiments hint, however, that it could be possible to use the same network for different noise-levels, and only adapt the scale through this scale-space approach by determining a suitable termination point. The first experiments show slightly worse results compared to fully retraining a network, and therefore extending these results could be another interesting direction of future research.

6.2.2 Deep net for the classification of CTCs

Robustness of the classifier. We have trained a deep CNN to classify CTCs. This system classifies small thumbnails based on the highest predicted probability for the different possible classes. In some cases, however, it is the case that the network is unsure about both classes. In these cases, it classifies the cells as CTC to be on the safe side. This worked good for a dataset similar to the training set, but performed worse on a slightly different dataset of different cancer cells. We were able to improve the performance by manually setting a treshold on the probability, i.e. every cell that has a predicted probability higher than p to be a CTC will be classified as CTC and everything below will be classified as non-CTC. Improving this workflow may improve the robustness of the system, and is therefore an interesting direction for future research.

Extension to cartridge classification or semantic segmentation. In Chapter 5 we showed that classification can be extended to pixel-wise cartridge classification by using a sliding-window to process the cartridge in terms of small thumbnails. We have, however, used a very simple thresholding to reduce the number of pixels to process. Because of this, some background pixels were included and some actual cell pixels were excluded. In order to improve the results, future research should focus on improved thresholding and possibly direct semantic segmentation with architectures like U-Net and RED-Net. The latter, however, requires a training set of segmented and annotated cartridges and for the specific application of CTC classification, this training set is not yet available.

Figure 6.1: Results from the scale-space experiments on the variational network. We have added severe noise with $\sigma = 90$ to the original input and added several basic structures such as such as small squares and disks of different intensities in order to see how the system processes them. From top to bottom we show the results after more and more iterations of the scale-space.

Bibliography

[1] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Advances in neural information processing systems*, 2012, pp. 1097–1105.

[2] D. C. Cireşan, A. Giusti, L. M. Gambardella, and J. Schmidhuber, "Mitosis detection in breast cancer histology images with deep neural networks," in *International Conference on Medical Image Computing and Computer-assisted Intervention*. Springer, 2013, pp. 411–418.

[3] D. Cireşan, U. Meier, J. Masci, and J. Schmidhuber, "A committee of neural networks for traffic sign classification," in *Neural Networks (IJCNN), The 2011 International Joint Conference on*. IEEE, 2011, pp. 1918–1921.

[4] J. Schmidhuber, "Deep learning in neural networks: An overview," *Neural networks*, vol. 61, pp. 85–117, 2015.

[5] M. D. Zeiler and R. Fergus, "Visualizing and understanding convolutional networks," in *European conference on computer vision*. Springer, 2014, pp. 818–833.

[6] X. Glorot and Y. Bengio, "Understanding the difficulty of training deep feedforward neural networks." in *Aistats*, vol. 9, 2010, pp. 249–256.

[7] L. Yaroslavsky, *Digital picture processing: an introduction*. Springer Science & Business Media, 2012, vol. 9.

[8] C. Tomasi and R. Manduchi, "Bilateral filtering for gray and color images," in *Sixth International Conference on Computer Vision, 1998*. IEEE, 1998, pp. 839–846.

[9] A. Buades, B. Coll, and J.-M. Morel, "A review of image denoising algorithms, with a new one," *Multiscale Modeling & Simulation*, vol. 4, no. 2, pp. 490–530, 2005.

[10] Y. Chen, W. Yu, and T. Pock, "On learning optimized reaction diffusion processes for effective image restoration," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2015, pp. 5261–5269.

[11] H. S. Seung, "Learning continuous attractors in recurrent networks." in *NIPS*, vol. 97, 1997, pp. 654–660.

[12] Y. Bengio, L. Yao, G. Alain, and P. Vincent, "Generalized denoising auto-encoders as generative models," in *Advances in Neural Information Processing Systems*, 2013, pp. 899–907.

[13] G. Alain and Y. Bengio, "What regularized auto-encoders learn from the data-generating distribution." *Journal of Machine Learning Research*, vol. 15, no. 1, pp. 3563–3593, 2014.

[14] L. I. Rudin, S. Osher, and E. Fatemi, "Nonlinear total variation based noise removal algorithms," *Physica D: Nonlinear Phenomena*, vol. 60, no. 1-4, pp. 259–268, 1992.

[15] P. Perona and J. Malik, "Scale-space and edge detection using anisotropic diffusion," *IEEE Transactions on pattern analysis and machine intelligence*, vol. 12, no. 7, pp. 629–639, 1990.

[16] S. Kindermann, S. Osher, and P. W. Jones, "Deblurring and denoising of images by nonlocal functionals," *Multiscale Modeling & Simulation*, vol. 4, no. 4, pp. 1091–1115, 2005.

[17] G. Gilboa and S. Osher, "Nonlocal linear image regularization and supervised segmentation," *Multiscale Modeling & Simulation*, vol. 6, no. 2, pp. 595–630, 2007.

[18] D. I. Shuman, S. K. Narang, P. Frossard, A. Ortega, and P. Vandergheynst, "The emerging field of signal processing on graphs: Extending high-dimensional data analysis to networks and other irregular domains," *IEEE Signal Processing Magazine*, vol. 30, no. 3, pp. 83–98, 2013.

[19] G. Gilboa and S. Osher, "Nonlocal operators with applications to image processing," *Multiscale Modeling & Simulation*, vol. 7, no. 3, pp. 1005–1028, 2008.

[20] W. S. McCulloch and W. Pitts, "A logical calculus of the ideas immanent in nervous activity," *The bulletin of mathematical biophysics*, vol. 5, no. 4, pp. 115–133, 1943.

[21] D. O. Hebb, *The organization of behavior: A neuropsychological theory*. Psychology Press, 2005.

[22] F. Rosenblatt, *The perceptron, a perceiving and recognizing automaton Project Para*. Cornell Aeronautical Laboratory, 1957.

[23] M. Minsky and S. Papert, *Perceptrons*. MIT press, 1988.

[24] P. Werbos, "Beyond regression: New tools for prediction and analysis in the behavioral sciences," Ph.D. dissertation, 1975.

[25] P. J. Werbos, "Applications of advances in nonlinear sensitivity analysis," in *System modeling and optimization*. Springer, 1982, pp. 762–770.

[26] D. H. Hubel and T. N. Wiesel, "Receptive fields, binocular interaction and functional architecture in the cat's visual cortex," *The Journal of physiology*, vol. 160, no. 1, pp. 106–154, 1962.

[27] K. Fukushima and S. Miyake, "Neocognitron: A self-organizing neural network model for a mechanism of visual pattern recognition," in *Competition and cooperation in neural nets*. Springer, 1982, pp. 267–285.

[28] Y. LeCun, B. Boser, J. S. Denker, D. Henderson, R. E. Howard, W. Hubbard, and L. D. Jackel, "Backpropagation applied to handwritten zip code recognition," *Neural computation*, vol. 1, no. 4, pp. 541–551, 1989.

[29] S. Kullback and R. A. Leibler, "On information and sufficiency," *The annals of mathematical statistics*, vol. 22, no. 1, pp. 79–86, 1951.

[30] N. Srivastava, G. E. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, "Dropout: a simple way to prevent neural networks from overfitting." *Journal of Machine Learning Research*, vol. 15, no. 1, pp. 1929–1958, 2014.

[31] G. E. Hinton, N. Srivastava, A. Krizhevsky, I. Sutskever, and R. R. Salakhutdinov, "Improving neural networks by preventing co-adaptation of feature detectors," *arXiv preprint arXiv:1207.0580*, 2012.

[32] V. Nair and G. E. Hinton, "Rectified linear units improve restricted boltzmann machines," in *Proceedings of the 27th international conference on machine learning (ICML-10)*, 2010, pp. 807–814.

[33] A. L. Maas, A. Y. Hannun, and A. Y. Ng, "Rectifier nonlinearities improve neural network acoustic models," in *Proc. ICML*, vol. 30, no. 1, 2013.

[34] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2016, pp. 770–778.

[35] S. Mallat, "Group invariant scattering," *Communications on Pure and Applied Mathematics*, vol. 65, no. 10, pp. 1331–1398, 2012.

[36] L. Sifre and S. Mallat, "Rotation, scaling and deformation invariant scattering for texture discrimination," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2013, pp. 1233–1240.

[37] J. Bruna and S. Mallat, "Invariant scattering convolution networks," *IEEE transactions on pattern analysis and machine intelligence*, vol. 35, no. 8, pp. 1872–1886, 2013.

[38] S. Mallat, "Understanding deep convolutional networks," *Phil. Trans. R. Soc. A*, vol. 374, no. 2065, p. 20150203, 2016.

[39] D. H. Hubel and T. N. Wiesel, "Receptive fields and functional architecture in two nonstriate visual areas (18 and 19) of the cat," *Journal of neurophysiology*, vol. 28, no. 2, pp. 229–289, 1965.

[40] M. A. Nielsen, *Neural Networks and Deep Learning*. Determination Press, 2015.

[41] J. Mairal, F. Bach, J. Ponce, and G. Sapiro, "Online dictionary learning for sparse coding," in *Proceedings of the 26th annual international conference on machine learning*. ACM, 2009, pp. 689–696.

[42] S. Rifai, P. Vincent, X. Muller, X. Glorot, and Y. Bengio, "Contractive auto-encoders: Explicit invariance during feature extraction," in *Proceedings of the 28th international conference on machine learning (ICML-11)*, 2011, pp. 833–840.

[43] J. Masci, U. Meier, D. Cireşan, and J. Schmidhuber, "Stacked convolutional auto-encoders for hierarchical feature extraction," *Artificial Neural Networks and Machine Learning–ICANN 2011*, pp. 52–59, 2011.

[44] D. P. Kingma and M. Welling, "Auto-encoding variational bayes," *arXiv preprint arXiv:1312.6114*, 2013.

[45] P. Vincent, H. Larochelle, I. Lajoie, Y. Bengio, and P.-A. Manzagol, "Stacked denoising autoencoders: Learning useful representations in a deep network with a local denoising criterion," *Journal of Machine Learning Research*, vol. 11, no. Dec, pp. 3371–3408, 2010.

[46] K. Hammernik, T. Klatzer, E. Kobler, M. P. Recht, D. K. Sodickson, T. Pock, and F. Knoll, "Learning a variational network for reconstruction of accelerated mri data," *arXiv preprint arXiv:1704.00447*, 2017.

[47] H. Kamyshanska and R. Memisevic, "On autoencoder scoring." in *ICML (3)*, 2013, pp. 720–728.

[48] J. Long, E. Shelhamer, and T. Darrell, "Fully convolutional networks for semantic segmentation," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2015, pp. 3431–3440.

[49] O. Ronneberger, P. Fischer, and T. Brox, "U-net: Convolutional networks for biomedical image segmentation," in *International Conference on Medical Image Computing and Computer-Assisted Intervention*. Springer, 2015, pp. 234–241.

[50] X. Mao, C. Shen, and Y.-B. Yang, "Image restoration using very deep convolutional encoder-decoder networks with symmetric skip connections," in *Advances in Neural Information Processing Systems*, 2016, pp. 2802–2810.

[51] D. C. Danila, K. Pantel, M. Fleisher, and H. I. Scher, "Circulating tumors cells as biomarkers: progress toward biomarker qualification," *Cancer journal (Sudbury, Mass.)*, vol. 17, no. 6, p. 438, 2011.

[52] M. Cristofanilli, G. T. Budd, M. J. Ellis, A. Stopeck, J. Matera, M. C. Miller, J. M. Reuben, G. V. Doyle, W. J. Allard, L. W. Terstappen *et al.*, "Circulating tumor cells, disease progression, and survival in metastatic breast cancer," *New England Journal of Medicine*, vol. 351, no. 8, pp. 781–791, 2004.

[53] M. C. Liu, P. G. Shields, R. D. Warren, P. Cohen, M. Wilkinson, Y. L. Ottaviano, S. B. Rao, J. Eng-Wong, F. Seillier-Moiseiwitsch, A.-M. Noone *et al.*, "Circulating tumor cells: a useful predictor of treatment efficacy in metastatic breast cancer," *Journal of Clinical Oncology*, vol. 27, no. 31, pp. 5153–5159, 2009.

[54] S. J. Cohen, C. J. Punt, N. Iannotti, B. H. Saidman, K. D. Sabbath, N. Y. Gabrail, J. Picus, M. Morse, E. Mitchell, M. C. Miller *et al.*, "Relationship of circulating tumor cells to tumor response, progression-free survival, and overall survival in patients with metastatic colorectal cancer," *Journal of clinical oncology*, vol. 26, no. 19, pp. 3213–3221, 2008.

[55] J. Tol, M. Koopman, M. Miller, A. Tibbe, A. Cats, G. Creemers, A. Vos, I. Nagtegaal, L. Terstappen, and C. Punt, "Circulating tumour cells early predict progression-free and overall survival in advanced colorectal cancer patients treated with chemotherapy and targeted agents," *Annals of Oncology*, vol. 21, no. 5, pp. 1006–1012, 2010.

[56] J. S. De Bono, H. I. Scher, R. B. Montgomery, C. Parker, M. C. Miller, H. Tissing, G. V. Doyle, L. W. Terstappen, K. J. Pienta, and D. Raghavan, "Circulating tumor cells predict survival benefit from treatment in metastatic castration-resistant prostate cancer," *Clinical Cancer Research*, vol. 14, no. 19, pp. 6302–6309, 2008.

[57] D. Olmos, H.-T. Arkenau, J. Ang, I. Ledaki, G. Attard, C. Carden, A. Reid, R. A'Hern, P. Fong, N. Oomen *et al.*, "Circulating tumour cell (ctc) counts as intermediate end points in castration-resistant prostate cancer (crpc): a single-centre experience," *Annals of Oncology*, vol. 20, no. 1, pp. 27–33, 2009.

[58] M. Yu, S. Stott, M. Toner, S. Maheswaran, and D. A. Haber, "Circulating tumor cells: approaches to isolation and characterization," *The Journal of cell biology*, vol. 192, no. 3, pp. 373–382, 2011.

[59] J. Khan, J. S. Wei, M. Ringner, L. H. Saal, M. Ladanyi, F. Westermann, F. Berthold, M. Schwab, C. R. Antonescu, C. Peterson *et al.*, "Classification and diagnostic prediction of cancers using gene expression profiling and artificial neural networks," *Nature medicine*, vol. 7, no. 6, pp. 673–679, 2001.

[60] C.-M. Svensson, S. Krusekopf, J. Lücke, and M. Thilo Figge, "Automated detection of circulating tumor cells with naive bayesian classifiers," *Cytometry Part A*, vol. 85, no. 6, pp. 501–511, 2014.

[61] Y. Mao, Z. Yin, and J. Schober, "A deep convolutional neural network trained on representative samples for circulating tumor cell detection," in *Applications of Computer Vision (WACV), 2016 IEEE Winter Conference on*. IEEE, 2016, pp. 1–6.

[62] Y. Yuan, Y. Shi, C. Li, J. Kim, W. Cai, Z. Han, and D. D. Feng, "Deepgene: an advanced cancer type classifier based on deep learning and somatic point mutations," *BMC Bioinformatics*, vol. 17, no. 17, p. 243, 2016.

[63] S. Riethdorf, H. Fritsche, V. Müller, T. Rau, C. Schindlbeck, B. Rack, W. Janni, C. Coith, K. Beck, F. Jänicke *et al.*, "Detection of circulating tumor cells in peripheral blood of patients with metastatic breast cancer: a validation study of the cellsearch system," *Clinical cancer research*, vol. 13, no. 3, pp. 920–928, 2007.

[64] C. L. Chen, A. Mahjoubfar, L.-C. Tai, I. K. Blaby, A. Huang, K. R. Niazi, and B. Jalali, "Deep learning in label-free cell classification," *Scientific reports*, vol. 6, 2016.

[65] D. Hopkins, "A computer vision approach to classification of circulating tumor cells," Ph.D. dissertation, Colorado State University, 2013.

[66] T. M. Scholtens, F. Schreuder, S. T. Ligthart, J. F. Swennenhuis, J. Greve, and L. W. Terstappen, "Automated identification of circulating tumor cells by image cytometry," *Cytometry Part A*, vol. 81, no. 2, pp. 138–148, 2012.

[67] L. Zeune, G. van Dalum, L. W. Terstappen, S. A. van Gils, and C. Brune, "Multiscale segmentation via bregman distances and nonlinear spectral analysis," *SIAM journal on imaging sciences*, vol. 10, no. 1, pp. 111–146, 2017.

[68] W. J. Allard, J. Matera, M. C. Miller, M. Repollet, M. C. Connelly, C. Rao, A. G. Tibbe, J. W. Uhr, and L. W. Terstappen, "Tumor cells circulate in the peripheral blood of all major carcinomas but not in healthy subjects or patients with nonmalignant diseases," *Clinical Cancer Research*, vol. 10, no. 20, pp. 6897–6904, 2004.

[69] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," *arXiv preprint arXiv:1409.1556*, 2014.

[70] O. Dürr and B. Sick, "Single-cell phenotype classification using deep convolutional neural networks," *Journal of Biomolecular Screening*, vol. 21, no. 9, pp. 998–1003, 2016.

[71] Y. Jia, E. Shelhamer, J. Donahue, S. Karayev, J. Long, R. Girshick, S. Guadarrama, and T. Darrell, "Caffe: Convolutional architecture for fast feature embedding," *arXiv preprint arXiv:1408.5093*, 2014.

[72] Ö. Çiçek, A. Abdulkadir, S. S. Lienkamp, T. Brox, and O. Ronneberger, "3d u-net: learning dense volumetric segmentation from sparse annotation," in *International Conference on Medical Image Computing and Computer-Assisted Intervention*. Springer, 2016, pp. 424–432.

[73] S. Lefkimmiatis, "Non-local color image denoising with convolutional neural networks," *arXiv preprint arXiv:1611.06757*, 2016.

[74] M. Defferrard, X. Bresson, and P. Vandergheynst, "Convolutional neural networks on graphs with fast localized spectral filtering," in *Advances in Neural Information Processing Systems*, 2016, pp. 3837–3845.

[75] M. Henaff, J. Bruna, and Y. LeCun, "Deep convolutional networks on graph-structured data," *arXiv preprint arXiv:1506.05163*, 2015.

[76] T. N. Kipf and M. Welling, "Semi-supervised classification with graph convolutional networks," *arXiv preprint arXiv:1609.02907*, 2016.

[77] M. M. Bronstein, J. Bruna, Y. LeCun, A. Szlam, and P. Vandergheynst, "Geometric deep learning: going beyond euclidean data," *arXiv preprint arXiv:1611.08097*, 2016.