Lecture Notes
Institute for Machine Learning, Johannes Kepler University Linz

Deep Learning and Neural Networks I

Winter Semester 2021

by Günter Klambauer
with contributions by other authors

October 15, 2021

Institute for Machine Learning
Johannes Kepler University Linz
A-4040 Linz, Austria

Tel. +43 732 2468 4530
https://www.jku.at/iml

Contents

List of Figures

List of Tables

Chapter 1

A brief history of neural networks

GÜNTER KLAMBAUER

In this attempt to compact the history of neural networks to a single page, we try to give an overview of the most important discovery steps that led to the current state-of-the-art in deep learning. To this end, we rely on the work of Schmidhuber (2015). Once again, we remind the reader that all history is *biased*, *incomplete* and *inaccurate*.

Chain rule and linear neurons. Searching for the earliest works on neural networks, one could consider linear regression methods as single-layer, linear networks. These methods date back to the early 1800s (e.g., Legendre, 1805; Gauss, 1809, 1821). The most prominent way to train neural networks at the time of writing, is simple gradient descent. The minimisation of errors through *gradient descent* (Hadamard, 1908) also dates back a couple of hundred years (Leibniz, 1684; Euler, 1744). The chain rule that is at the core of the derivation of the *backpropagation* algorithm was also introduced by Leibniz (1676). Although these are the main mathematical tools necessary to train artificial neural networks, those had not been connected to neural networks.

Hebbian learning, Perceptron and first multilayer networks. In a sense, neural network research started in the 1940s (e.g., McCulloch and Pitts, 1943; Hebb, 1949) with Donald Hebb formulating a hypothesis how neural networks could learn and the Hebbian learning rule. The earliest neural network architectures (McCulloch and Pitts, 1943) did not learn. The so-called perceptron (Rosenblatt, 1958a, 1962) was implemented as a large machine with adaptive connections to learn image recognition tasks. Networks trained by the *Group Method of Data Handling* (GMDH) (Ivakhnenko and Lapa, 1965) were perhaps the first learning systems of the *Feedforward Multilayer Perceptron* type.

However, these early architectures were then abandoned and further work discouraged potentially because of the work of Minsky and Papert (1969) showing that linear perceptrons cannot learn XOR relations. This also coincides with the first *AI winter* of the 70s.

Multi-layer perceptron and backpropagation. Multi-layer perceptrons became popular again in the 80s with the works of (Rumelhart et al., 1986c) who used backpropagation to train multi-layer perceptrons and demonstrate the emergence of internal representations in hidden layers. Although the idea to use of backpropagation for neural networks had been earlier suggested by

Werbos (1974, 1981). Additionally, multi-layer perceptrons could not only solve the XOR problem, but it was shown that they are universal function approximators (Hornik et al., 1989).

Recurrent neural networks and the vanishing gradient problem. In 1991, the fundamental problem that comes with training neural networks with many layers was discovered by Sepp Hochreiter in his Diploma thesis (Hochreiter, 1990) and named as "vanishing gradient problem". The problem occurs especially pronounced for recurrent neural networks, which had been suggested earlier by (Hopfield, 1982) and (Rumelhart et al., 1986c). The problem is strongly ameliorated by long short-term memory networks (Hochreiter, 1991; Hochreiter and Schmidhuber, 1995).

Convolutional neural networks. A type of neural network that is highly performant at image recognition tasks, was originally introduced as Neocognitron (Fukushima, 1980), but later combined with backprop by (LeCun et al., 1989, 1990; LeCun and Bengio, 1995) and provided remarkable results at recognizing images.

Despite these promising results, there was not enough computational power at the time to train networks with many neurons or multiple layers. In the early 90s, a period started which is sometimes considered as the second *AI winter* came that lasted for almost two decades.

Re-emergence as Deep Learning. Finally, neural networks emerged again as Deep Learning (Schmidhuber, 2015; LeCun et al., 2015) methods. With the increased computational power by graphical processing units (GPUs), the availability of large data sets and a number of inventions ameliorating the vanishing gradient problem, *convolutional neural networks* and *recurrent neural networks*, such as LSTMs, and *deep feed-forward neural networks* demonstrated excelling predictive performance. As prominent enabling inventions, the discovery of unsupervised pre-training (Hinton et al., 2006), the rectified linear units (Nair and Hinton, 2009), deep CNNs (Krizhevsky et al., 2012a), and Dropout (Srivastava et al., 2014) can be named. Since about 2012, Deep Learning methods dominate many data-driven scientific and industrial areas.

A rough timeline of milestones for neural networks and Deep Learning.

1943: McCulloch and Pitts (1943) provide a mathematical description of the neuron.

1957: Rosenblatt (1957, 1958b) creates the *Perceptron*.

1960: Kelley (1960); Dreyfus (1962) lay the foundations for *backpropagation* with the chain rule.

1965: Ivakhnenko and Lapa (1965) propose the first *multi-layer perceptron* using the name Group Method of Data Handling (GMDH).

1969: Minsky and Papert (1969) demonstrate the perceptrons cannot solve the XOR problem which leads to a declined interest in neural networks.

1970: First computer-coded *backpropagation* algorithm by Linnainmaa (1970).

1971: Ivakhnenko (1971) creates an 8-layer *deep network* using the GMDH.

1974: Werbos (1974, 1981) suggest the use of backpropagation to train neural networks.

1980: Fukushima (1979, 1980) suggests the Neocognitron, the first *convolutional neural network*.

1982: John Hopfield creates a *Hopfield network*, which is the birth of *recurrent neural networks*.

1985: Ackley et al. (1985) create *Boltzmann Machine* that is a stochastic recurrent neural network. This neural network has only input layer and hidden layer but no output layer.

1986: Sejnowski and Rosenberg (1987) creates NeTalk, a neural network which learns to pronounce written English text by being shown text as input and matching phonetic transcriptions for comparison.

1986: Rumelhart et al. (1986b,c) in their paper "Learning Representations by back-propagating errors" show the successful implementation of backpropagation in the neural network.

1986: Smolensky (1986) suggests the *Restricted Boltzmann Machine*.

1989: LeCun (1989) trains a convolutional neural network using backpropagation that recognizes handwritten digits.

1989: The first version of the *universal function approximator theorem* is suggested by Cybenko (1989) and extended by Hornik et al. (1989).

1991: Sepp Hochreiter describes the *vanishing gradient problem* as fundamental problem for training deep neural networks in his Diploma thesis (Hochreiter, 1991).

1997: Sepp Hochreiter and Jürgen Schmidhuber publishes a milestone paper on "Long Short-Term Memory" (LSTM) (Hochreiter and Schmidhuber, 1997b). It is a type of recurrent neural network architecture which will go on to revolutionize deep learning in decades to come.

2004: GPU implementations of neural networks are suggested by Oh and Jung (2004).

2006: Hinton et al. (2006) stacked multiple RBMs together in layers and called them Deep Belief Networks. The training process is much more efficient for large amount of data.

2008: Andrew NG's group at Stanford advocates and promotes the use of GPUs to train Deep Neural Networks.

2009: In 2009 Fei-Fei Li, a professor at Stanford, launches ImageNet which is a database of 14 million labeled images (Deng et al., 2009). This datasets would serve as a benchmark for the deep learning researchers who would participate in ImageNet competitions (ILSVRC) every year.

2011: Yoshua Bengio and colleagues show that rectifiers can be used to counter the vanishing gradient problem (Glorot et al., 2011).

2011: Deep convolutional networks by Jürgen Schmidhuber's research group reach human-level performance at image recognition (Ciresan et al., 2011a,b).

2012: AlexNet (Krizhevsky et al., 2012a), a deep convolutional network for large-scale image recognition, wins the 2012 ImageNet challenge.

2014: Goodfellow et al. (2014) invents *Generative Adversarial Networks* that can produce realistic images.

2016: AlphaGo (Silver et al., 2016), a deep reinforcement learning model, beats a human champion at the Game of Go.

2019: Yoshua Bengio, Geoffrey Hinton, and Yann LeCun win the Turing Award 2018 for their contributions in advancements in area of deep learning and artificial intelligence.

Chapter 2

Biological neural networks

GÜNTER KLAMBAUER

In the following, we give a short overview of how biological neural networks are structured and are thought to function. As we will later see, artificial neural networks, or deep neural networks, have some similarities, but also some strong differences to those.

Neurons. A neuron is a biological cell specialized to process information. The discovery of neurons as the basic structural components of the brain, is addressed to Raman y Cajal and Camillo Golgi, who also won the Nobel Prize for their works on neurons. Roughly 90% of the neurons in humans are located in the cerebral cortex of our brain. Neurons (see Figure 2.1) are cells with several specialized parts, such as synapses, axons, and dendrites, which serve for information processing and whose morphology and function are detailed in the following.

Neurons are remarkable among the cells of the body in their ability to propagate signals rapidly over large distances (Dayan et al., 2001). They do this by generating characteristic electrical pulses called action potentials or, more simply, spikes that can travel down nerve fibers. Neurons represent and transmit information by firing sequences of spikes in various temporal patterns.

Synapses. Signals received from other neurons or other cells are transmitted to a neuron via special transition points, the *synapses*. Such a transition point is usually located at the *dendrites* of a neuron, sometimes also directly at the *soma*.

There are two types of synapses: electrical and chemical synapses. An electrical signal received at the synapse, i.e. from the pre-synaptic side, is directly transported into the post-synaptic cell nucleus. This means that there is a direct, strong, non-adjustable connection from signal transmitter to signal receiver, e.g. useful for escape reflexes, that have to be "hard-coded" in an organism. The second type is the chemical synapse. In these synapses, there is no direct electrical coupling of source and destination, but this coupling is interrupted by the synaptic cleft. This cleft separates the pre- and post-synaptic side electrically from each other. On the pre-synaptic side of the synaptic cleft, the electrical signal is converted into a chemical signal by releasing chemical signalling substances, so-called neurotransmitters. These neurotransmitters overcome the synaptic cleft and transfer the information into the cell nucleus, where the signal is converted back into electrical information. The neurotransmitters are degraded very quickly, so that also here exact information impulses are possible.

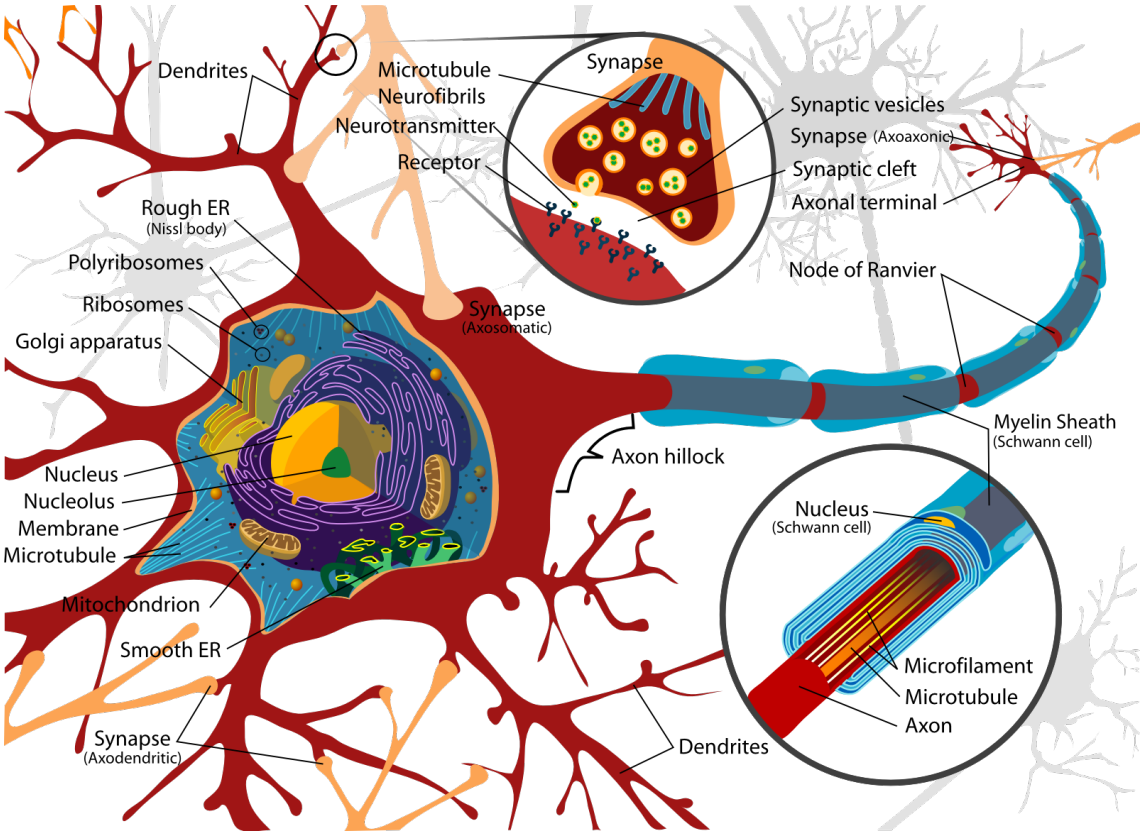


Figure 2.1: A schematic representation of a neuron. It is composed of a cell body, soma, and two types of branches called axon and dendrites. [Source:WikimediaCommons]

Despite the much more complicated functionality, the chemical synapse has striking differences compared to the electrical variant: First, the chemical synapse is a one-way transmission system. Since the pre- and post-synaptic parts are not directly electrically connected, electrical impulses in the post-synaptic part cannot pass over to the pre-synaptic part. Second, there are many different neurotransmitters that can be released in different amounts in a synaptic cleft. For example, there are neurotransmitters that have a excitatory effect on the post-synaptic cell nucleus, but there are also others that have and inhibitory effect. Some synapses transmit a strongly excitatory signal, some only weakly excitatory signals. The regulatory diversity is enormous, and the fact that the synapses are also variable, i.e. that they can form a stronger or weaker connection over time, is one of the central points when looking at the brain's ability to learn.

Dendrites. The elaborate branching structure of the dendritic tree allows a neuron to receive inputs from many other neurons through synaptic connections. Dendritic trees branch out from the neuron nucleus, called soma, and serve to pick up signals from many different sources, which are then transferred into the nucleus. The branching amount of dendrites is also called *dendritic arborization*. After an amount of excitatory and inhibiting signals has reached the cell nucleus (soma) via synapses and dendrites, the soma accumulates these signals. As soon as the accumulated signal exceeds a certain value (called threshold potential), the neuron nucleus in turn triggers an electrical impulse, which is then intended for transmission to the subsequent neurons to which the current neuron is connected.

Axon. The transmission of the impulse to other neurons is done by the axon. The axon is a thread-like extension of the soma. Axons from single neurons can traverse large fractions of the brain or, in some cases, of the entire body. In extreme cases, an axon can become about one meter long (e.g. in the spinal cord). The axon is electrically isolated to better conduct the electrical signal and flows into dendrites to pass on the information to other neurons.

Membrane potential. The electrical signal of relevance to the nervous system is the difference in electrical potential between the interior of a neuron and the surrounding extracellular medium. Neurons can have an electrical charge difference, a so-called potential compared to their environment. This charge difference is a central term needed to understand the processes in the neuron, we call it membrane potential. The *membrane potential*, i.e. the charge difference, is created by several types of charged atoms (ions), which are differently highly concentrated inside and outside the neuron. This difference is also called a concentration gradient.

Ion channels. Neurons have are a wide variety of membrane-spanning ion channels that allow ions, predominantly sodium (Na+), potassium (K+), calcium (Ca2+), and chloride (Cl-), to move into and out of the cell. Ion channels control the flow of ions across the cell membrane by opening and closing in response to voltage changes and to both internal and external signals.

Resting potential. Under resting conditions, the potential inside the cell membrane of a neuron is about -70 mV relative to that of the surrounding bath (which is conventionally defined to be 0 mV), and the cell is said to be polarized. In this case the membrane potential is 70mV.

Since this potential depends on concentration gradients of different ions, it is naturally a central question how these concentration gradients are maintained: Normally diffusion prevails everywhere, so all ions strive to reduce concentration gradients and distribute themselves evenly everywhere. Ion pumps located in the cell membrane maintain concentration gradients that support this membrane potential difference. In order to keep up the potential, several mechanisms act simultaneously to counter the tendency of the ions to be represented as evenly as possible. If the concentration of an ion inside the neuron is higher than outside, these ions diffuse to the outside and vice versa. The positively charged ion K+(potassium) is frequently found in the neuron and less frequently outside the neuron. Therefore it diffuses slowly through the membrane out of the neuron.

A "pump" actively moves ions against the direction they would naturally run. Sodium (Na+) is actively pumped out of the cell, although it wants to go along the concentration and electrical gradient into the cell. Potassium (K+) diffuses strongly out of the cell, but is actively pumped into it again. For this reason we call the pump also sodium-potassium pump. The pump maintains the concentration gradient for both sodium and potassium, so that a kind of flow equilibrium is created and the rest potential finally lands at the observed 70mV. In summary, the membrane potential is maintained by not allowing some ions to pass through the membrane and by actively pumping other ions against the concentration and electrical gradients. Now that we know that each neuron has a membrane potential, we want to look closely at how a neuron receives and transmits signals.

Action potential. Sodium and potassium can diffuse through the membrane, sodium slowly, potassium faster. This happens through channels contained in the membrane, sodium or potassium channels. In addition to these channels, which are always open and are responsible for the diffusion and are balanced by the sodium-potassium pump, there are also channels which are not always-opened, but only opened "as needed". These controllable channels are opened when an accumulated incoming stimulus exceeds a certain threshold value. Incoming stimuli can arise from other neurons or other causes: For example, there are specialized forms of neurons, sensory cells, for which the incidence of light can represent such a stimulus. If enough light falls in to exceed the threshold value, the controllable channels are opened. The threshold value (the threshold value potential) is about -55mV. As soon as this is reached by the incoming stimuli, the neuron is activated and an electrical signal, an action potential, is triggered. This signal is then transmitted to those cells which are connected to the neuron under consideration.

Action potential phases. We will take a closer look at the phases of the action potential:

- Resting state: Only the potassium and sodium channels are open, the membrane potential is at -70mV and is kept active by the neuron.
- Stimulus to threshold value: A stimulus opens channels so that sodium can flow in. The charge of the cell interior becomes more positive. As soon as the membrane potential exceeds the threshold of -55mV, the action potential is triggered by opening many sodium channels.
- Depolarisation: Sodium flows in. We remember that sodium wants to flow in both because there is considerably less of it in the cell than outside, and because there is a negative environment in the cell that attracts the positive sodium. Due to the strong inflow, the membrane

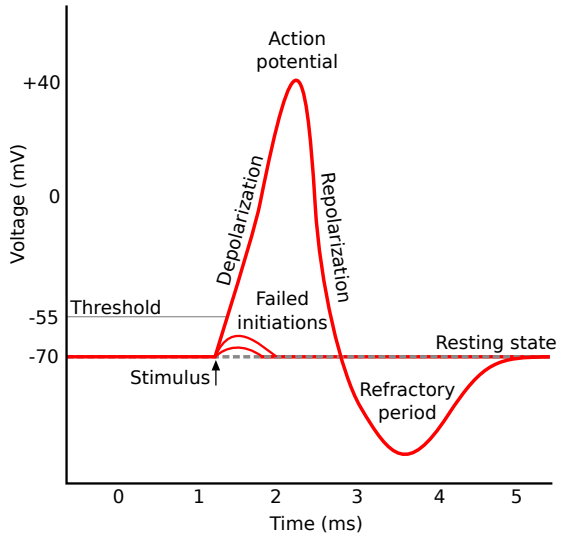


Figure 2.2: Action potential phases

potential rises drastically, up to approx. +30mV. This is then the electrical impulse, the action potential.

- Repolarisation: Sodium channels are now closed, but the potassium channels are opened. The positively charged potassium now wants to get out of the positive cell interior and is also highly concentrated in the cell interior than outside, which further accelerates the outflow.

- Refractory period: Both sodium and potassium channels are closed again. First, the membrane potential is now slightly more negative than the resting potential, which is due to the fact that the potassium channels close slower, which allows potassium (positively charged) to escape outside the cells due to its low concentration. After a refractory time of 1-2ms the resting state is restored, such that the neuron can react to new stimuli such as an action potential. The refractory time is a forced pause which a neuron must wait in order to regenerate. The shorter it is, the more often a neuron can fire per time fire.

Saltatory conduction. Many axons in vertebrates are covered with an insulating sheath of myelin except at gaps, called the nodes of Ranvier, where there is a high density of fast voltage-dependent Na+ channels. The myelin sheath consists of many layers of glial cell membrane wrapped around the axon. This insulation gives the myelinated region of the axon a very high membrane resistance and a small membrane capacitance and also results in what is called saltatory propagation, also called *saltatory conduction*. Saltatory propagation means that membrane potential depolarization is transferred passively down the myelin-covered sections of the axon, and action potentials are actively regenerated at the nodes of Ranvier.

Myelin sheaths consists of Schwann cells (in the PNS) or oligodendrocytes (in the CNS) which electrically isolate the axon. Between these cells there are gaps of 0.1-2mm, which are called Ranvier's nodes. These nodes occur where one auxiliary cell ends and the next begins. Logically, the axon is less isolated at such a nodes. At the nodes, an exchange of substances between intracellular

and extracellular space is possible, which does not function at dividing axons, which are located between two nodes (internodes) and thus isolated through the myelin sheath. This exchange of substances enables the generation of signals. The forwarding of an action potential works as follows: It does not run continuously along the axon, but jumps from one nodes to the next. So it runs a series of depolarizations along the Ranvier nodes . One action potential triggers the next one like a domino, usually several nodes are active at the same time. Both Schwann cells and oligodendrocytes are expressions of the glia cells, of which there are about 50 times more than neurons. Glia cells surround the neurons, isolate them from each other and supply them with energy.

It is obvious that the propagation is faster when the distance beween the nodes are larger. For example, axons with large internodes (2 mm) achieve a signal propagation speed of approx. 180m per second. The internodes cannot become arbitrarily large, however, since the action potential to be transmitted would otherwise fade too much until the next node. The nodes also have the task to amplify the signal regularly. At the end of the axon, there are other cells that receive the action potential.

Learning in biological neural networks Whereas the anatomy and morphology of biological is well known and understood, the actual process of how biological neural network learns is less well described. In the first attempt, in the year 1949, Donald Hebb postulated a learning rule (Hebb, 1949) that states that the connection between two neurons is strengthened if the neurons fire simultaneously (or within a time interval). This is sometimes called *Hebbian learning*. Hebbian learning is treated in more detail in a later chapter.

The Hebbian learning rule specifies how the strength of the connection between to neurons is changed in dependency of their activation.

There has been some evidence that this type of learning indeed exists in the brain (Bliss and Lømo, 1973; Paulsen and Sejnowski, 2000). However, many open problems in natural sciences revolve around the function of the brain, and learning is only one example. Another example of open scientific problems is how the brain stores information. Within machine learning, Hebbian learning has sparked the whole area of *biologically plausible learning*, which we will not further detail here. We will however comment on differences between how artificial neural networks learn and how biological neural networks are thought to learn in a later part of this work.

Comparison of biological and artificial neural networks Although we have not introduced artificial neural networks (ANNs) so far, we give a short overview over their differences to biological neural networks (BNNs).

- Learning: There are many differences when it comes to learning. As we will find out, ANNs typically learn with a gradient-based approach, where information from a distant output layer is propagated back through many layers of a network and all connections are changed by that information. For BNNs, as we have found out above, the current opinion is that rather local information is important for strengthening or weakening the connections. Furthermore, BNNs are much more flexible, areas can move or change function and new connections can grow. For ANNs, their structure in terms of numbers of neurons and connections, is typically fixed at the beginning of learning and cannot change.

- Size: The human brain is much larger than any current ANN with an estimated number of 100 billion neurons and ten times more glia cells and even magnitudes more connections. By comparison, the large Residual Networks (He et al., 2016) have about 50-100 millions of weights (connections) and even more recent ANNs are reported to have 8.3 billions of weights.
- ANNs usually perform computations sequentially layer-wise, meaning that the second layer of neurons has to wait until the first layer has finished his computations and passed information to the next layer. In BNNs, neurons can work on their own in an asynchronous fashion.
- Power consumption: One of the most surprising facts is how efficient our brain works with power consumption. The brain takes roughly 20% of the body's energy and operates at about 20 watts. To train a state-of-the-art neural network, it requires multiple GPUs that operate at, e.g., 250watts, and additionally generate a lot of heat. Overall, BNNs are much more efficient regarding power consumption, than ANNs. Note that power consumption is difficult to compare between these two types. ANNs can also process hundreds of images per second.
- Signals. Also the signal transport is much more efficient in BNNs, since only sparse information has to be transferred. In a biological neuron, an action potential is either triggered or not, such that overall only infrequently a piece of binary information has to be transferred. For an ANNs to make a decision, all input neurons have to transport their information to the next layer via continuous valued matrix-vector operations.

Chapter 3

Notation

GÜNTER KLAMBAUER

3.1 Scalars, Vectors, and Matrices

For *scalars*, we use simple lower-case, italic letters, such as x, y, or a. We use bold, lower-case symbols for denoting *vectors*, such as $\boldsymbol{x}$ and $\boldsymbol{w}$ — these represent *column vectors*. Finally, we use bold, capital letters to denote *matrices*, for example $\boldsymbol{X}$, $\boldsymbol{W}$ or $\boldsymbol{Y}$.

3.2 Sizes and dimensions

- N: number of training examples, i.e. objects or samples, in the *training data set*. Running index: n.
- D: number of input units which is also the number of features that a sample has. Running index: d
- M: number of test or validation examples, i.e. objects or samples, in the *test data set*. Running index: m with $N + 1 \leq m \leq N + M$.
- K: number of output units. Running index: k.
- L: number of layers in a network without counting the input layer. Running index: l.
- J: input dimension of a general neural network layer. Running index: j.
- I: output dimension of a general neural network layer. Running index: i.

3.3 Data objects

A superscript ^{n} will denote the n-th training example. On occasions, where a potential misunderstanding with exponentiation could occur, this can be replaced by $^{(n)}$.

- $\boldsymbol{x}^n \in \mathbb{R}^D$ or $\boldsymbol{x} \in \mathbb{R}^D$: the n-th input data point or a general data point, respectively. A column vector. Sometimes also used for the inputs of a particular neural network layer, then the dimensions could be $\boldsymbol{x} \in \mathbb{R}^J$. Alternatively, $\boldsymbol{x}^{(n)}$ could be used to avoid confusion with exponentiation.
- $y^n \in \mathbb{R}$ or $y \in \mathbb{R}$: a scalar label for the n-th data point or a general label y, respectively.
- $\boldsymbol{y}^n \in \mathbb{R}^K$ or $\boldsymbol{y} \in \mathbb{R}^K$: For multi-class of multi-task problems, this is a *label vector* for the n-th data point or a general label vector, respectively. A column vector.
- $\hat{y} \in \mathbb{R}$ or $\hat{\boldsymbol{y}} \in \mathbb{R}^K$: the predicted scalar label or – for a multi-class problem – the predicted label vector, respectively.
- $p \in \mathbb{R}$ or $\boldsymbol{p} \in \mathbb{R}^K$: same as above if outputs can be interpreted probabilistically, the predicted scalar label or – for a multi-class problem – the predicted label vector.
- $\boldsymbol{a} \in \mathbb{R}^I$: *activations* of a neural network. As an exception, $\boldsymbol{h}$ can be used.
- $\boldsymbol{s} \in \mathbb{R}^I$: *pre-activations*, also called *netI*, of a neural network.

Note that there is a difference between a superscript and a subscript:

- $\boldsymbol{x}^n$: the n-th data point. This represents a *column vector*.
- x_d: the d-th component of the vector $\boldsymbol{x} = (x_1, \ldots, x_d, \ldots, x_D)$. This represents a *scalar*.

In rare cases, to denote the d-th component of the n-th data point, we will use the notation x_{nd} (see also "Stacked data and labels" below).

3.4 Parameter objects

- $\boldsymbol{w} \in \mathbb{R}^D$: a *weight* or *parameter* vector of a simple machine learning method, such as linear regression. A column vector.
- $\boldsymbol{W} \in \mathbb{R}^{I \times J}$: a *weight* or *parameter* matrix of a learning method mapping from an input space with dimension J to an output space with dimension I.
- $\boldsymbol{b} \in \mathbb{R}^I$: a bias vector.
- $\boldsymbol{\theta}$: a set of parameters of a probabilistic model.

3.5 Multiple layers

For a neural network with multiple layers, the following notations are used:

- $n_h^{[l]}$: number of hidden units of the l-th layer. Thus, $D = n_h^{[0]}$ and $K = n_h^{[L+1]}$.
- $\boldsymbol{a}^{[l]} \in \mathbb{R}^{n_h^{[l]}}$: *activations* of a neural network in the l-th layer.

- $\boldsymbol{s}^{[l]} \in \mathbb{R}^{n_h^{[l]}}$: *pre-activations*, also called *netI*, of a neural network in the l-th layer.
- $\boldsymbol{W}^{[l]} \in \mathbb{R}^{n_h^{[l-1]} \times n_h^{[l]}}$: the weight matrix connecting the $(l-1)$-th layer with the l-th layer.
- $\boldsymbol{b}^{[l]} \in \mathbb{R}^{n_h^{[l]}}$: the bias vector in the l-th layer.

3.6 Stacked data and labels

At some points, the input data points $\boldsymbol{x}^1, \ldots, \boldsymbol{x}^n$ are stacked together to form a data matrix $\boldsymbol{X}$, which contains the data points $\boldsymbol{x}^n$ as *rows*. Scalar labels $y^1, \ldots, y^N$ are stacked together to form a vector $\boldsymbol{y}$, or vectorial labels $y^1, \ldots, y^N$ are stacked together to form a *label matrix* $\boldsymbol{Y}$, which contains the labels as *rows*.

- $\boldsymbol{X} \in \mathbb{R}^{N \times D}$: input data matrix. The rows represent objects, i.e. samples. We assume that objects, i.e. samples, are represented or described by *feature vectors* $\boldsymbol{x}^n$.
- $\boldsymbol{y} \in \mathbb{R}^N$: the scalar labels of all samples stacked to a column vector.
- $\boldsymbol{Y} \in \mathbb{R}^{N \times K}$: for multi-class or multi-task problems, stacked labels yield a *label matrix*.

3.7 Common transformations

The parameters of a single neuron, so-called weights, are written as a vector $\boldsymbol{w} = (w_1, \ldots, w_d)^T \in \mathbb{R}^d$ and we will repeatedly use *linear/affine transformations*:

$$\boldsymbol{w}^T \boldsymbol{x} + b = w_1 x_1 + \ldots w_D x_D + b. \tag{3.1}$$

In neural networks, linear or affine transformations into a multi-dimensional space are frequently used.

$$\boldsymbol{s} = \boldsymbol{W}\boldsymbol{x} + \boldsymbol{b}. \tag{3.2}$$

Furthermore, a non-linear function $f : \mathbb{R} \to \mathbb{R}$, a so-called *activation function*, is typically applied element-wise to the resulting vectors:

$$\boldsymbol{a} = f(\boldsymbol{W}\boldsymbol{x} + \boldsymbol{b}). \tag{3.3}$$

Occasionally, expressions such as

$$\boldsymbol{w}^T \boldsymbol{w} = \sum_{i=1}^{I} w_i^2 = \|\boldsymbol{w}\|^2, \tag{3.4}$$

where $\|\boldsymbol{w}\|^2$ is the squared 2-norm of $\boldsymbol{w}$, are used – typically in context of regularization techniques. Note that $\|\boldsymbol{w} + \boldsymbol{v}\|^2 = (\boldsymbol{w} + \boldsymbol{v})^T(\boldsymbol{w} + \boldsymbol{v}) = \|\boldsymbol{w}\|^2 + 2\boldsymbol{w}^T\boldsymbol{v} + \|\boldsymbol{v}\|^2$.

3.8 Functions

- $f : \mathbb{R} \to \mathbb{R}$: a non-linear *activation function* that is applied element-wise to a vector or matrix. Sometimes also denoted as ϕ.
- $g(\boldsymbol{x}; \boldsymbol{w})$: a machine learning model with input $\boldsymbol{x}$ and parameters $\boldsymbol{w}$.
- $p(\boldsymbol{x}; \boldsymbol{\theta})$: a probabilistic model with data point $\boldsymbol{x}$ and set of parameters $\boldsymbol{\theta}$.
- $L(y, g(\boldsymbol{x}; \boldsymbol{w}))$ or $L(y, \hat{y})$: a loss function L.
- $R_{\text{emp}}(\boldsymbol{y}, \boldsymbol{X}, \boldsymbol{w})$: an empirical error or risk function. Typically, the empirical error is an average of the loss function for a single training data point. For neural networks, this serves as a *cost function* that is minimized.
- $R_{\text{emp}}(\boldsymbol{w})$: an empirical error or risk function. The same as above only that occasionally the dependency on $\boldsymbol{X}$ and $\boldsymbol{y}$ is dropped to keep the notation uncluttered.

3.9 Derivatives and matrix layout

We use the so-called *numerator layout* of matrix calculus:

- $\frac{\partial R(\boldsymbol{w})}{\partial \boldsymbol{w}}$: a row vector of length W according to the definition of the Jacobian.
- $\frac{\partial \boldsymbol{a}}{\partial x}$: column vector
- $\frac{\partial a}{\partial \boldsymbol{x}}$: row vector
- $\frac{\partial \boldsymbol{a}}{\partial \boldsymbol{x}}$: matrix with as many rows as dimension of $\boldsymbol{a}$, as many columns as dimesion of $\boldsymbol{x}$.
- $\frac{\partial a}{\partial \boldsymbol{X}}$: dimension of transposed $\boldsymbol{X}$

With the *Nabla operator* ∇, we can switch to *denominator layout*:

- $\nabla R(\boldsymbol{w})$ or $\nabla_{\boldsymbol{w}} R(\boldsymbol{w})$: a column vector of length W. $\nabla_{\boldsymbol{w}} R(\boldsymbol{w}) = \left(\frac{\partial R(\boldsymbol{w})}{\partial \boldsymbol{w}}\right)^T$

3.10 Distributions

- $\mathcal{U}(a, b)$: a uniform distribution in the interval $[a, b]$. This distribution has mean $\frac{1}{2}(a+b)$ and variance $\frac{1}{12}(b-a)^2$.
- $\mathcal{N}(\mu, \sigma^2)$: a normal distribution with mean μ and variance σ^2. Also commonly referred to as the Gaussian distribution.
- $\mathcal{B}(n, p)$: a binomial distribution with size parameter n and probability parameter p. Sometimes used as $\mathcal{B}(1, p)$ to denote Bernoulli distributions.

Chapter 4

Basic neural network architectures

GÜNTER KLAMBAUER AND SEPP HOCHREITER

Artificial neural networks (ANNs) can be motivated by the computational principles of the human brain which is so far the best known pattern recognition and pattern association device.

The processing units of the human brain are the neurons which are interconnected. The connections are able to transfer information from one processing unit to others. The processing units can combine and modulate the incoming information. The incoming information changes the state of a neuron which represents the information at this neuron. The state serves to transfer the information to other neurons. The connections have different strength (synaptic weights) which give the coupling of the neurons and, therefore, the influence of one neuron onto the other.

Learning in the human brain is believed to be mainly determined by changing the synaptic weights, i.e. changing the strength of the connections between the processing units.

Artificial neural networks treated in our context ignore that the information in the human brain is transferred through spikes (short bursts of high voltage), although so-called *spiking neural networks* are also a special type of ANNs. However, the ANNs we will use can represent a rate coding, which means the information transferred is the firing rate of a neuron and the state of a neuron is its firing rate.

Neurons in artificial neural networks are represented by a variable a_j for the j-th neuron which gives the current state of the j-th neuron which is called *activation* of the neuron. Connections in ANNs are parameters w_{ij} giving the strength of the connection from unit j to unit i which are called *weights*. See Fig. 4.1 for the weight w_{ij} from unit j to unit i. These parameters are summarized in a vector (or weight vector) $\boldsymbol{w}$ which are the parameters of the neural network model.

Artificial neural networks possess special neurons. Neurons that are directly activated by the environment, are called *input units*. Their activation is typically the value of the feature vector $\boldsymbol{x}$, which is assumed to represent the sensory input to the neural network. Neurons from which the result of the processing is taken, are called *output units*. Typically the state of certain units is the output of the neural network. The remaining units are called *hidden units.* See Fig. 4.2 for a small architecture of a 3 layered net with only one hidden layer.

Artificial neural networks can in principle be processed in parallel, i.e. each unit can be updated independent of other units according to its current incoming signals. As we will see later, ANNs are universal function approximators and recurrent ANNs are as powerful as *Turing machines*. Thus, ANNs are powerful enough to model a part of the world to solve a problem.

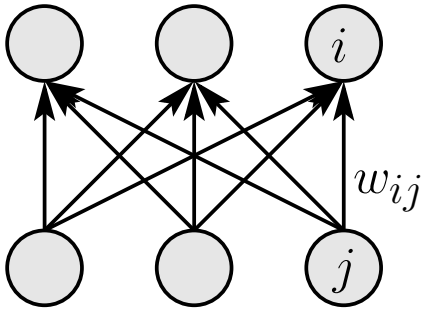


Figure 4.1: Artificial neural networks: units and weights. The weight w_{ij} gives the weight, connection strength, or synaptic weight from units j to unit i.

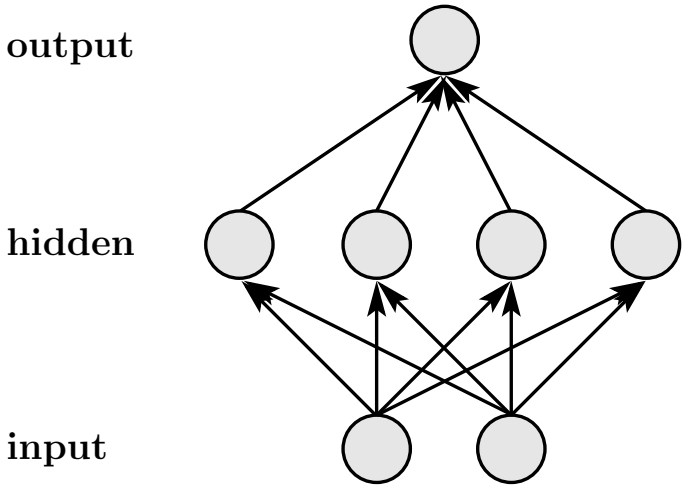


Figure 4.2: Artificial neural networks: a 3-layered net with an input, hidden, and output layer.

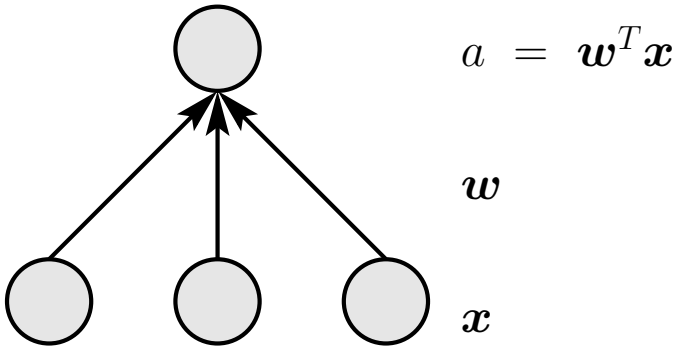


Figure 4.3: A linear network with one output unit.

4.1 Linear Model: Neuron

In this section we will focus on (artificial) neural networks (NNs) which represent linear discriminant functions. We express one variable y as a linear function of the other variable x:

$$y = w_0 + w_1 x. \tag{4.1}$$

If fitting a linear function (a line), that is, to find optimal parameters w_1 and w_0, the objective is the *sum of the squared deviations* between the y values and the regression line. The line that optimizes this criterion is the *least squares line*. The scalar variable y is called the *dependent variable*.

We now generalize this approach to the multivariate case. This means that instead of a scalar x, we now consider a vector of features $\boldsymbol{x}$ with components x_j, which are called *explanatory variables*, *independent variables*, *regressors*, or *features*.

The estimation of y from a vector of explanatory variables $\boldsymbol{x}$ is called *multiple linear regression*. If y is generalized to a vector $\boldsymbol{y}$, then this is called *multivariate linear regression*. We focus on multiple linear regression, that is, the case where multiple features are summarized in the vector $\boldsymbol{x}$.

4.1.1 Model

The linear neural network is

$$g(\boldsymbol{x}; \boldsymbol{w}) = a = \boldsymbol{w}^T\boldsymbol{x}. \tag{4.2}$$

That means we have D input neurons where the i-th input neuron has activation x_d. There is only one output neuron with activation a. Connections exist from each input unit to the output unit where the connection from the i-th input unit to the output is denoted by w_i (see Fig. 4.3).

The general form of a linear model with noise is

$$g(\boldsymbol{x}; \boldsymbol{w}) = a = \boldsymbol{w}^T\boldsymbol{x} + \epsilon. \tag{4.3}$$

ϵ is an additive *noise* or *error* term which accounts for the difference between the predicted value and the observed outcome y.

The *residual* ϵ for the n-th observation is

$$\epsilon^n = y^n - a^n = y^n - \tilde{\boldsymbol{w}}^T \boldsymbol{x}^n, \tag{4.4}$$

where $\tilde{\boldsymbol{w}}$ is a candidate for the parameter vector $\boldsymbol{w}$. The residual ϵ^n measures how well y^n is predicted by the linear model with parameters $\tilde{\boldsymbol{w}}$.

4.1.2 Loss function and empirical error

A *loss function* should quantify how far a prediction $\hat{y}$ is away from the true value (label) y. The *squared loss* is defined as

$$L(y, \hat{y}) = \frac{1}{2}(y - \hat{y})^2. \tag{4.5}$$

To assess how well all observations are fitted simultaneously by a linear model, the squared residuals of all observation are summed up to S, which is called the *residual sum of squares* (RSS) — also known as the sum of squared residuals (SSR) or the sum of squared errors of prediction (SSE). "ESS" is the *explained sum of squares* (also known as the model sum of squares or sum of squares due to regression (SSR)), which is the sum of squared deviations of the predicted values from the observed mean $\overline{y}$ of $\boldsymbol{y}$:

$$S(\tilde{\boldsymbol{w}}) = \frac{1}{2}\sum_{n=1}^{N}(\epsilon^n)^2 = \frac{1}{2}\sum_{n=1}^{N}\left(y^n - \tilde{\boldsymbol{w}}^T \boldsymbol{x}^n\right)^2 = \frac{1}{2}\left(\boldsymbol{y} - \boldsymbol{X}\tilde{\boldsymbol{w}}\right)^T \left(\boldsymbol{y} - \boldsymbol{X}\ \tilde{\boldsymbol{w}}\right). \tag{4.6}$$

This quantity S will later be introduced as *empirical error* R_{emp} that depends on the parameters $\boldsymbol{w}$, the data $\boldsymbol{X}$ and the labels $\boldsymbol{y}$:

$$R_{\text{emp}}(\tilde{\boldsymbol{w}}, \boldsymbol{X}, \boldsymbol{y}) = S(\tilde{\boldsymbol{w}}). \tag{4.7}$$

Probabilistic view of the loss function and empirical error. If we assume a centered Gaussian noise on the target values $\varepsilon \sim \mathcal{N}(0, \sigma^2)$ and that the data points are drawn independently. Our model $g(\boldsymbol{x}, \boldsymbol{w}) = \boldsymbol{w}^T \boldsymbol{x}$ should approximate the mean of the Gaussian $y = g(\boldsymbol{x}, \boldsymbol{w}) + \varepsilon$, therefore, we have the likelihood of the labels as follows:

$$p(\boldsymbol{y} \mid \boldsymbol{X}, \boldsymbol{w}, \sigma) = \prod_{n=1}^{N} \mathcal{N}(y^n \mid g(\boldsymbol{x}^n, \boldsymbol{w}), \sigma) \tag{4.8}$$

$$p(\boldsymbol{y} \mid \boldsymbol{X}, \boldsymbol{w}, \sigma) = \prod_{n=1}^{N} \mathcal{N}(y^n \mid \boldsymbol{w}^T \boldsymbol{x}^n, \sigma) \tag{4.9}$$

$$p(\boldsymbol{y} \mid \boldsymbol{X}, \boldsymbol{w}, \sigma) = \prod_{n=1}^{N} \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{(y^n - \boldsymbol{w}^T \boldsymbol{x}^n)^2}{2\sigma^2}} \tag{4.10}$$

The log-likelihood is:

$$\log p(\boldsymbol{y} \mid \boldsymbol{X}, \boldsymbol{w}, \sigma) = -\frac{N}{2}\log(2\pi\sigma^2) - \frac{1}{\sigma^2}S(\boldsymbol{w}, \boldsymbol{X}, \boldsymbol{y}), \tag{4.11}$$

where $S(\boldsymbol{w}, \boldsymbol{X}, \boldsymbol{y}) = \frac{1}{2}\sum_{n=1}^{N}(\underbrace{y^n - \boldsymbol{w}^T\boldsymbol{x}^n}_{:=\epsilon^n})^2$ is the residual sum of squares that we have defined above. Consequently, minimizing the residual sum of squares is equivalent to *maximizing the likelihood* of the data.

4.1.3 Learning linear models: closed form solution

The *least squares estimator* $\hat{\boldsymbol{w}}$ for $\boldsymbol{w}$ minimizes $S(\tilde{\boldsymbol{w}})$:

$$\hat{\boldsymbol{w}} = \arg\min_{\tilde{\boldsymbol{w}}} S(\tilde{\boldsymbol{w}}) = \left(\boldsymbol{X}^T\boldsymbol{X}\right)^{-1}\boldsymbol{X}^T\boldsymbol{y}. \tag{4.12}$$

The solution is obtained by setting the derivative of $S(\tilde{\boldsymbol{w}})$ with respect to the parameter vector $\tilde{\boldsymbol{w}}$ to zero:

$$\frac{\partial S(\tilde{\boldsymbol{w}})}{\partial \tilde{\boldsymbol{w}}} = (\boldsymbol{X}\tilde{\boldsymbol{w}} - \boldsymbol{y})^T\boldsymbol{X} = \boldsymbol{0}. \tag{4.13}$$

The matrix $\boldsymbol{X}^+ = \left(\boldsymbol{X}^T\boldsymbol{X}\right)^{-1}\boldsymbol{X}^T$ is called the *pseudo inverse* of the matrix $\boldsymbol{X}$ because $\boldsymbol{X}^+\boldsymbol{X} = \boldsymbol{I}_m$.

The least squares estimator is the best linear unbiased estimator. Under the normality assumption for the errors, the least squares estimator is the maximum likelihood estimator (MLE).

Concerning notation and the parameter vector, we have the true parameter vector $\boldsymbol{w}$, a candidate parameter vector or a variable $\tilde{\boldsymbol{w}}$, and an estimator $\hat{\boldsymbol{w}}$, which is in our case the least squares estimator.

4.1.4 Learning linear models: (stochastic) gradient descent

A solution of the optimization problem above may also be obtained by an iterative procedure called *gradient descent* (see Section "Learning methods") on the empirical error function. Because the error function is convex we can find its global minimum. We start with an initial guess for the parameters $\boldsymbol{w}^{\text{old}}$ and update it in the following way:

$$\boldsymbol{w}^{\text{new}} = \boldsymbol{w}^{\text{old}} - \eta\nabla_{\boldsymbol{w}}R_{\text{emp}}(\boldsymbol{w}, \boldsymbol{X}, \boldsymbol{y}) \mid_{\boldsymbol{w}^{\text{old}}} = \boldsymbol{w}^{\text{old}} - \eta\boldsymbol{X}^T(\boldsymbol{X}\boldsymbol{w}^{\text{old}} - \boldsymbol{y}), \tag{4.14}$$

where $\nabla_{\boldsymbol{w}}R_{\text{emp}}(\boldsymbol{w}, \boldsymbol{X}, \boldsymbol{y}) \mid_{\boldsymbol{w}^{\text{old}}}$ is the gradient of the empirical error with respect to $\boldsymbol{w}$ evaluated at $\boldsymbol{w}^{\text{old}}$ and η is the *learning rate*. The learning rate has to be adjusted carefully in order for the algorithm to converge.

For a single data point $\boldsymbol{x}$ with label y, the derivative of the empirical error is thus given by

$$\nabla_{\boldsymbol{w}}R_{\text{emp}}(\boldsymbol{w}, \boldsymbol{x}, y) = (\boldsymbol{w}^T\boldsymbol{x} - y)\boldsymbol{x}. \tag{4.15}$$

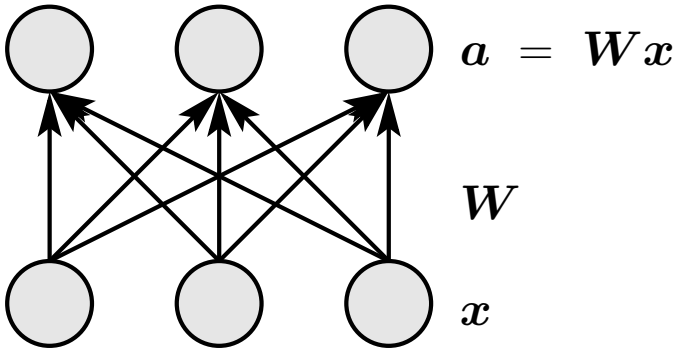


Figure 4.4: A linear network with three output units.

4.1.5 Interpretation of gradient-based learning of a linear model

We now investigate the gradient-based learning rule further. We find that the old parameter vector $\boldsymbol{w}^{\text{old}}$ is adjusted by the scaled gradient:

$$\nabla_{\boldsymbol{w}} R_{\text{emp}}(\boldsymbol{w}, \boldsymbol{X}, \boldsymbol{y}) = (\boldsymbol{X}\boldsymbol{w} - \boldsymbol{y})^T \boldsymbol{X} = \sum_{n=1}^{N} \underbrace{(\boldsymbol{w}^T \boldsymbol{x}^n - y^n)}_{\alpha^n} \boldsymbol{x}^n, \tag{4.16}$$

where we dropped the superscript $^{\text{old}}$ to keep the notation uncluttered. We observe that this is basically a weighted summation over data points $\sum_{n=1}^{N} \alpha^n \boldsymbol{x}^n$, where the weights α_n are the difference between the label y^n and the prediction $\boldsymbol{w}^T \boldsymbol{x}^n$ for the data point $\boldsymbol{x}^n$. Thus, a data point, for which the current prediction is far from the label, will have a large $|\alpha^n|$ and hence a strong influence on the parameter update.

4.1.6 Multi-task linear regression

The linear neuron can be extended to a linear net if the output contains more than one unit. The targets are now output vectors $\boldsymbol{Y} = \left(\boldsymbol{y}^1, \ldots, \boldsymbol{y}^N\right)^T$, where $\boldsymbol{Y} \in \mathbb{R}^{N \times K}$.

The linear neural network is

$$\boldsymbol{g}(\boldsymbol{x}; \boldsymbol{w}) = \boldsymbol{a} = \boldsymbol{W}\boldsymbol{x} + \boldsymbol{\epsilon}, \tag{4.17}$$

where $\boldsymbol{W}$ is the weight matrix $\boldsymbol{W} \in \mathbb{R}^{K \times D}$ and $\boldsymbol{\epsilon} \in \mathbb{R}^K$ the noise vector with K as the number of output units (see Fig. 4.4 with $K = 3$). For convenience the weight matrix is sometimes written as a weight vector $\boldsymbol{w}$, where the columns of $\boldsymbol{W}$ are stacked on top of each other. This network is a combination of linear neurons with

$$g_k(\boldsymbol{x}; \boldsymbol{w}) = a_k = \boldsymbol{w}_k^T \boldsymbol{x}, \tag{4.18}$$

where $\boldsymbol{w}_k$ is the k-th line of $\boldsymbol{W}$ written as column vector.

4.2 Perceptron

4.2.1 Model

A linear neuron with a threshold function is called *perceptron*. The output of the perceptron is binary. The *perceptron model* is

$$g(\boldsymbol{x}, \boldsymbol{w}) = a = \text{sign}\left(\boldsymbol{w}^T \boldsymbol{x}\right), \tag{4.19}$$

where sign is called *threshold function*. Note that the perceptron is a linear classifier without an offset b. Note, that without the offset b the learning problem is not translation invariant.

For the perceptron we assume a classification task, where $y \in \{-1, 1\}$.

Minsky and Papert showed in 1969 that many problems cannot be solved by the perceptron. For example the XOR problem (see Section 4.5) is a nonlinear problem which cannot be solved by the perceptron.

Because of the work of Minsky and Papert neural networks were not popular until the mid 80s, when the multi-layer perceptron with nonlinear units and back-propagation was reintroduced.

4.2.2 Loss and empirical error function

The loss function quantifies how far a prediction $\hat{y}$ is away from the true label y. We have already encountered the *squared loss* in the section on the linear neuron (above). The loss function for the perceptron, called *perceptron loss* is:

$$L(y, \hat{y}) = \begin{cases} 0, & \text{if } y \ \text{sign}(\hat{y}) \geq 0 \\ -y\ \hat{y} & \text{else} \end{cases} \tag{4.20}$$

Note that the first case covers both that the predicted and the actual label are positive and that they are negative. The "else"-case covers the cases $y = -1 \ \ \& \ \ \hat{y} = 1$ as well as $y = 1 \ \ \& \ \ \hat{y} = -1$.

Hence, the empirical error function for the perceptron is defined as

$$R_{\text{emp}}(\boldsymbol{w}, \boldsymbol{X}, \boldsymbol{y}) = - \sum_{n=1; a^n y^n = -1}^{N} y^n \boldsymbol{w}^T \boldsymbol{x}^n, \tag{4.21}$$

where the sum goes over all data points that are mis-classified. Scaling does not influence the perceptron output.

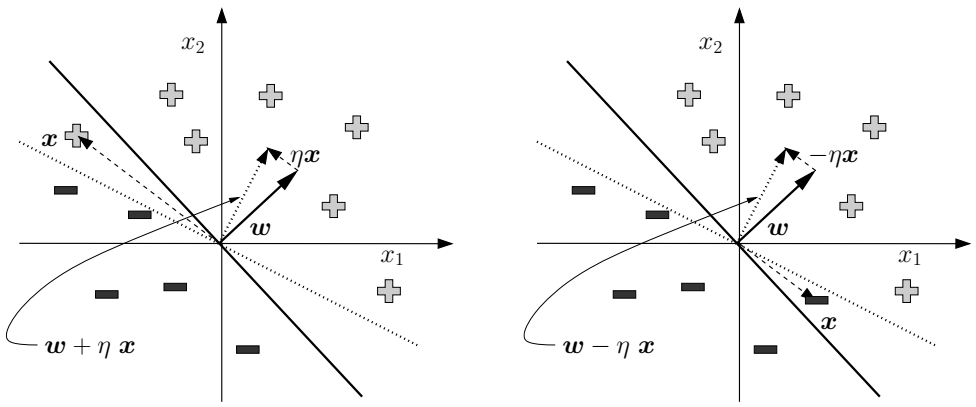


Figure 4.5: The perceptron learning rule. Left: a positive example is wrongly classified and correctly classified after the weight update. Right: a negative example is wrongly classified and correctly processed after weight update.

4.2.3 Perceptron learning rule

Deriving the above expression with respect to $\boldsymbol{w}$ and using a gradient update we obtain

$$\Delta \boldsymbol{w} = \eta \sum_{n=1;a^n y^n=-1}^{N} y^n \boldsymbol{x}^n \tag{4.22}$$

or in an on-line formulation, i.e., for a single sample:

$$\text{if} \quad ay = -1 : \Delta \boldsymbol{w} = \eta y \boldsymbol{x}. \tag{4.23}$$

Here we used the $\Delta \boldsymbol{w}$ notation which indicates the difference between to consecutive values of $\boldsymbol{w}$ or the change of the weights. Concretely $\Delta \boldsymbol{w} = \boldsymbol{w}^{t+1} - \boldsymbol{w}^t$, where $\boldsymbol{w}^t$ is the weight vector after t updates.

Interpretation of the update rule. The update rule is very simple: if a pattern is wrongly classified then the label multiplied by the input vector is added to the weight vector. Fig. 4.5 depicts the perceptron update rule. Assuming that the problem is linearly separable, the perceptron learning rule converges after finite many steps to a solution which classifies all training examples correctly. However an arbitrary solution out of all possible solutions is chosen. If the problem in not linearly separable then the algorithms does not stop.

4.2.4 Convergence: The Perceptron Convergence Theorem (PCT)

We consider a data set that is linearly separable. Note that the perceptron learning algorithm only converges if the data is *linearly separable*. This means we know that there exists at least one $\hat{\boldsymbol{w}}$ for which all training vectors are correctly classified:

$$\hat{\boldsymbol{w}}^T \boldsymbol{x}^n y^n > 0, \tag{4.24}$$

for all n.

The learning process starts with some arbitrary weight vector which, without loss of generality, we assume to be the zero vector. At each step of the algorithm, the weight vector is updated by the learning rule (see above):

$$\boldsymbol{w}^{\tau+1} = \boldsymbol{w}^{\tau} + \boldsymbol{x}^n y^n, \tag{4.25}$$

where $\boldsymbol{x}^n$ is a sample that is misclassified by the perceptron. Suppose that, after running the algorithm for some time, the number of times that each vector $\boldsymbol{x}^n$ has been presented and misclassified is τ^n. Then the weight vector at the current timepoint is:

$$\boldsymbol{w} = \sum_{n=1}^{N} \tau^n \boldsymbol{x}^n y^n. \tag{4.26}$$

We now take the scalar product of this expression with $\hat{\boldsymbol{w}}$ to obtain:

$$\begin{aligned} \hat{\boldsymbol{w}}^T \boldsymbol{w} &= \sum_{n=1}^{N} \tau^n \hat{\boldsymbol{w}}^T \boldsymbol{x}^n y^n && (4.27)\\ &\geq \tau \min_n \left(\hat{\boldsymbol{w}}^T \boldsymbol{x}^n y^n \right), && (4.28)\\ & && (4.29) \end{aligned}$$

where $\tau = \sum_{n=1}^{N} \tau^n$ is the number of total weight updates, and the equality results from replacing each update vector by the smallest of the update vectors.

From Eq. (4.28) it follows that $\hat{\boldsymbol{w}}^T \boldsymbol{w}$ is bounded from below by a function that grows linearly with the number of updates τ.

On the other hand, we have:

$$\begin{aligned} \|\boldsymbol{w}^{\tau+1}\|^2 &= \|\boldsymbol{w}^{\tau}\|^2 + \|\boldsymbol{x}^n\|^2 (y^n)^2 + 2\boldsymbol{w}^{\tau T} \boldsymbol{x}^n y^n && (4.30)\\ &\leq \|\boldsymbol{w}^{\tau}\|^2 + \|\boldsymbol{x}^n\|^2 (y^n)^2, && (4.31) \end{aligned}$$

where we dropped the expression $2\boldsymbol{w}^{\tau T} \boldsymbol{x}^n y^n$ to obtain the equality. The expression must be negative since the sample $\boldsymbol{x}^n$ must have been misclassified and thus $\boldsymbol{w}^{\tau T} \boldsymbol{x}^n y^n < 0$. We also have $(y^n)^2 = 1$ since the labels are from $\{-1, 1\}$. We can also bound $\|\boldsymbol{x}^n\|^2 \leq \|\boldsymbol{x}^{\max}\|^2$, where $\boldsymbol{x}^{\max}$ is the sample with the largest norm. Hence we have

$$\Delta \|\boldsymbol{w}\|^2 = \|\boldsymbol{w}^{\tau+}\|^2 - \|\boldsymbol{w}^{\tau}\|^2 \leq \|\boldsymbol{x}^{\max}\|^2. \tag{4.32}$$

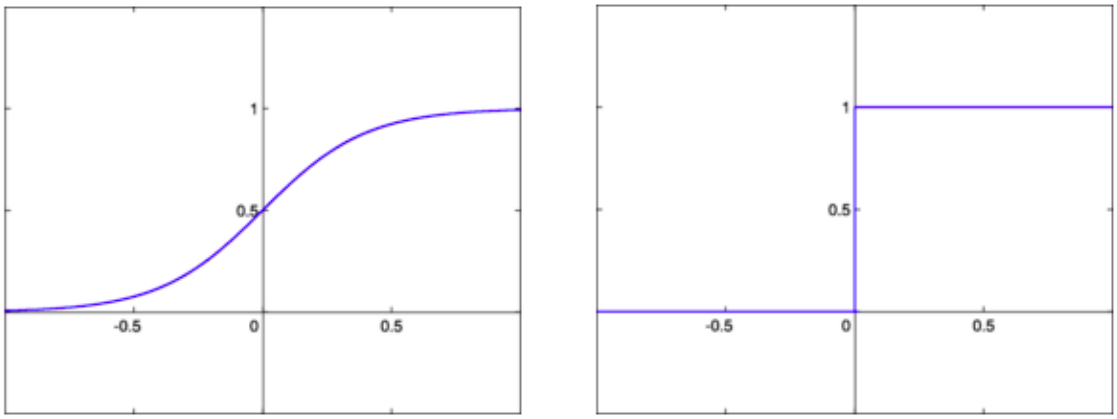


Figure 4.6: Graphs of the sigmoid function σ (left) and the step function (right).

After τ weight updates we have

$$\|\boldsymbol{w}\|^2 \leq \tau \|\boldsymbol{x}^{\max}\|^2, \tag{4.33}$$

which means that the length $\|\boldsymbol{w}\|$ of the weight vector increases no faster than with $\sqrt{\tau}$. We now remember that we also found that $\hat{\boldsymbol{w}}^T \boldsymbol{w}$ is bounded below by a function that grows linearly with τ. Since $\hat{\boldsymbol{w}}$ is fixed, we see that for sufficiently large τ, these two results would become incompatible (at some point the function with linear growth would overtake the square function). Thus τ, the number of updates and how often a data point is misclassified, cannot grow indefinitely and so the algorithm must converge in a finite number of time steps.

4.3 Linear neuron with sigmoid activation: logistic regression

The problem of non-convergence in the non-linearly separable case, is solved by logistic regression (Figure 4.7). The main idea of logistic regression is that the model $g(\boldsymbol{x}, \boldsymbol{w})$ aims to approximate the probability that the given sample $\boldsymbol{x}$ belongs to the positive class, thus $g(\boldsymbol{x}, \boldsymbol{w}) \approx p(y = 1|\boldsymbol{x})$. The logistic regression method employs a sigmoid function $\sigma(x)$ (Figure 4.6) to continuously approximate the sign or step function of the perceptron:

$$\sigma(x) = \frac{1}{1 + e^{-x}} \tag{4.34}$$

4.3.1 Model

Another difference between the Perceptron and logistic regression is that the logistic regression assumes classes coded as zero and one: $y \in \{0, 1\}$. Logistic regression learns a function of the following form:

$$g(\boldsymbol{x}, \boldsymbol{w}) = a = \sigma(\boldsymbol{w}^T \boldsymbol{x}) = \frac{1}{1 + e^{-\boldsymbol{w}^T \boldsymbol{x}}}, \tag{4.35}$$

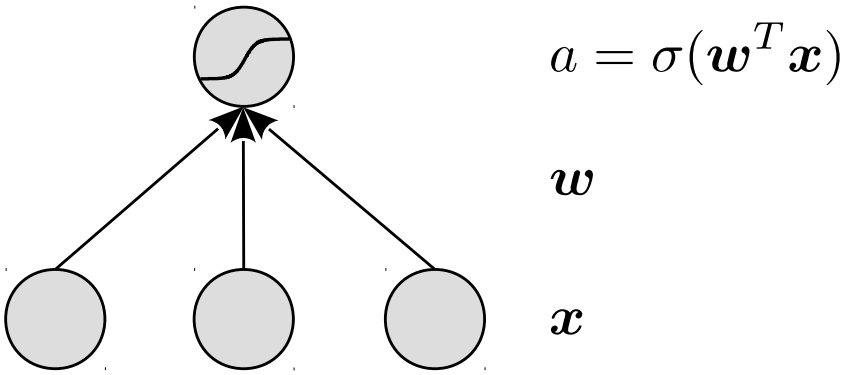


Figure 4.7: A linear network with one output unit with sigmoid activation: the model function used for logistic regression.

which provides values in the interval $(0, 1)$ that should provide the probability[1] $\hat{p}$ that a data point $\boldsymbol{x}$ belongs to class $\mathcal{C}_1$, i.e. $y = 1$:

$$\hat{p}(\mathcal{C}_1|\boldsymbol{x}) = \hat{p}(y = 1|\boldsymbol{x}) = a = \frac{1}{1 + e^{-\boldsymbol{w}^T \boldsymbol{x}}}, \tag{4.36}$$

where the notation $\hat{p}(\mathcal{C}_1|\boldsymbol{x})$ and $\hat{p}(y = 1|\boldsymbol{x})$ is equivalent. We can either code the binary label as C_0 and C_1 for the negative and the positive class, respectively, or as $y = 0$ and $y = 1$.

To keep the notation uncluttered, we will just use $p(y|\boldsymbol{x})$ instead of $\hat{p}(y|\boldsymbol{x})$ in the following.

4.3.2 Loss function and empirical error

The loss function is also motivated probabilistically, to maximize the likelihood of the given labels:

$$p(y \mid \boldsymbol{x}, \boldsymbol{w}) = p(\mathcal{C}_1|\boldsymbol{x})^y (1 - p(\mathcal{C}_1|\boldsymbol{x}))^{1-y} = a^y (1 - a)^{1-y} \tag{4.37}$$

$$-\log p(y \mid \boldsymbol{x}, \boldsymbol{w}) = -y \log(p(\mathcal{C}_1|\boldsymbol{x})) - (1 - y) \log(1 - p(\mathcal{C}_1|\boldsymbol{x})) \tag{4.38}$$

The function above (Eq.(4.38)) is called *binary cross-entropy* and is used frequently in machine learning and Deep Learning. The binary cross-entropy can be viewed as *negative log-likelihood* and can be used to formulate the empirical error $R_{\text{emp}}(\boldsymbol{w}, \boldsymbol{X}, \boldsymbol{y})$:

[1] Note that the this is not an actual probability, but a model fitted to provide values between zero and one.

$$
\begin{aligned}
R_{\text{emp}}(\boldsymbol{w}, \boldsymbol{x}, y) &= -\log\left[p(\boldsymbol{y} \mid \boldsymbol{x}, \boldsymbol{w})\right] \\
&= -\left(y \log(a) + (1-y)\log(1-a)\right) \\
&= -\left(y \log\left(\frac{1}{1+e^{-\boldsymbol{w}^T\boldsymbol{x}}}\right) + (1-y)\log\left(1 - \frac{1}{1+e^{-\boldsymbol{w}^T\boldsymbol{x}}}\right)\right)
\end{aligned} \tag{4.39}
$$

In case of multiple data points $\boldsymbol{X}$ with labels $\boldsymbol{y}$, the empirical error is just a sum over data points:

$$
R_{\text{emp}}(\boldsymbol{w}, \boldsymbol{X}, \boldsymbol{y}) = -\sum_{n=1}^{N}\left(y^n \log\left(\frac{1}{1+e^{-\boldsymbol{w}^T\boldsymbol{x}^n}}\right) + (1-y^n)\log\left(1 - \frac{1}{1+e^{-\boldsymbol{w}^T\boldsymbol{x}^n}}\right)\right). \tag{4.40}
$$

Note: Scaling of the objective function

Note that the location of the minima of the loss function does not change if the objective is scaled by a positive constant. Thus, oftentimes the mean over data points is used instead of the sum.

4.3.3 Learning logistic regression models: gradient descent

In order to be able to maximize the likelihood or – equivalently – minimizing the empirical error, we calculate the gradient of Eq.(4.39) with respect to the parameters for a single data point $\boldsymbol{x}$:

$$
\begin{aligned}
\nabla_{\boldsymbol{w}} R_{\text{emp}}(\boldsymbol{w}, \boldsymbol{x}, y) &= \left(\frac{1}{1+e^{-\boldsymbol{w}^T\boldsymbol{x}}} - y\right)\boldsymbol{x} \\
&= \left(\sigma(\boldsymbol{w}^T\boldsymbol{x}) - y\right)\boldsymbol{x} \\
&= (a-y)\,\boldsymbol{x}
\end{aligned} \tag{4.41}
$$

Unlike for linear models, no closed-form solution of setting this gradient to zero exists due to the non-linearity of the sigmoid. However, as we will see below, this is a *convex optimization* problem and thus an iterative procedure, in which we follow the gradient of the loss function downwards, leads to a unique global optimum. Thus, we will use the following rule to update the parameters $\boldsymbol{w}$:

$$
\boldsymbol{w}^{\text{new}} = \boldsymbol{w}^{\text{old}} - \eta \nabla_{\boldsymbol{w}} R_{\text{emp}}(\boldsymbol{w}, \boldsymbol{X}, \boldsymbol{y}) \mid_{\boldsymbol{w}^{\text{old}}} \tag{4.42}
$$

$$
= \boldsymbol{w}^{\text{old}} - \eta \sum_{n=1}^{N}\left(\frac{1}{1+e^{-\boldsymbol{w}^{\text{old}^T}\boldsymbol{x}^n}} - y^n\right)\boldsymbol{x}^n = \boldsymbol{w}^{\text{old}} - \eta(\boldsymbol{a}-\boldsymbol{y})^T\boldsymbol{X}. \tag{4.43}
$$

The learning rate η has to be chosen appropriately in order for the algorithm to converge. Typical learning rates are $\eta = 0.1$ or $\eta = 0.01$.

For a single data point $\boldsymbol{x}$ with label y, the derivative of the empirical error is thus given by

$$\nabla_{\boldsymbol{w}} R_{\text{emp}}(\boldsymbol{w}, \boldsymbol{x}, y) = (\sigma(\boldsymbol{w}^T \boldsymbol{x}) - y)\boldsymbol{x}. \tag{4.44}$$

4.3.4 Learning logistic regression models: Iterative reweighted least squares

We have used *gradient descent* before to learn linear regression models, perceptron or logistic regression models. The general rule was to update the current parameters $\boldsymbol{w}^{\text{old}}$ and move along the gradient to decrease the *empirical error*. The general approach was

$$\boldsymbol{w}^{\text{new}} = \boldsymbol{w}^{\text{old}} - \eta \nabla_{\boldsymbol{w}} R_{\text{emp}}(\boldsymbol{w}) \mid_{\boldsymbol{w}^{\text{old}}}, \tag{4.45}$$

where we dropped the dependency on $\boldsymbol{X}$ and $\boldsymbol{y}$ to keep the notation uncluttered. Gradient descent is a *first order* method since it considers only first-order derivatives. The second order derivative informs us whether the first derivative is increasing or decreasing which hints at the function's curvature. Thus, inclusion of the second derivative $\boldsymbol{H} = \nabla\nabla R_{\text{emp}}(\boldsymbol{w})$ can improve learning. However, the Hessian matrix $\boldsymbol{H}$ is often costly to compute and to store since it is quadratic in the number of parameters.

In the *Newton-Raphson* update, the minimization of $R_{\text{emp}}(\boldsymbol{w})$ takes the following form

$$\boldsymbol{w}^{\text{new}} = \boldsymbol{w}^{\text{old}} - \boldsymbol{H}^{-1} \nabla_{\boldsymbol{w}} R_{\text{emp}}(\boldsymbol{w}) \mid_{\boldsymbol{w}^{\text{old}}}, \tag{4.46}$$

where $\boldsymbol{H}$ is the Hessian matrix evaluated at $\boldsymbol{w}^{\text{old}}$. The Hessian matrix for the problem at hand can be calculated as follows:

$$\begin{aligned} \nabla_{\boldsymbol{w}} R_{\text{emp}}(\boldsymbol{w}) &= \left(\frac{1}{1 + e^{-\boldsymbol{w}^T \boldsymbol{x}}} - y \right) \boldsymbol{x} = (a - y)\, \boldsymbol{x} \\ \boldsymbol{H} = \nabla^2_{\boldsymbol{w}} R_{\text{emp}}(\boldsymbol{w}) &= a(1-a)\boldsymbol{x}\boldsymbol{x}^T, \end{aligned} \tag{4.47}$$

The Hessian for multiple data points $\boldsymbol{X}$ can therefore be written as $\boldsymbol{H} = \boldsymbol{X}^T \boldsymbol{R} \boldsymbol{X}$, where the matrix $\boldsymbol{R}$ is a diagonal matrix with entries $\boldsymbol{R}_{nn} = a_n(1 - a_n)$. First, we observe that for an arbitrary vector $\boldsymbol{\alpha}$ the expression $\boldsymbol{\alpha}^T \boldsymbol{H} \boldsymbol{\alpha} > 0$ and thus the Hessian is positive definite. If the calculation and inversion of the Hessian is feasible, the iterative procedure to learn the parameters $\boldsymbol{w}$ becomes:

$$\boldsymbol{w}^{\text{new}} = \boldsymbol{w}^{\text{old}} - (\underbrace{\boldsymbol{X}^T \boldsymbol{R} \boldsymbol{X}}_{\boldsymbol{H}})^{-1} \underbrace{\boldsymbol{X}^T (\boldsymbol{a} - \boldsymbol{y})}_{\nabla_{\boldsymbol{w}} R_{\text{emp}}(\boldsymbol{w})}. \tag{4.48}$$

The procedure is called *iterative reweighted least squares* because the expression $(\boldsymbol{X}^T \boldsymbol{R} \boldsymbol{X})^{-1} \boldsymbol{X}^T$ resembles the *pseudo-inverse* $(\boldsymbol{X}^T \boldsymbol{X})^{-1} \boldsymbol{X}^T$ that a appears in the solution of the linear regression problem. The difference is that the weighting matrix $\boldsymbol{R}$ changes after each parameter update.

4.4 Linear neuron with softmax: Softmax Regression

This method extends *logistic regression*, which solves two-class problems, to multiple classes $\mathcal{C}_1, \ldots, \mathcal{C}_K$, i.e., a so-called multi-class setting. For each input object $\boldsymbol{x}$, the method has to decide in which of those K categories the object belongs. Classical image recognition problems, in which each image has to be assigned into on of the categories "dog","cat", "ship", "car", etc., are an example of those *multi-class problems*. As we will later find out, softmax regression is not particularly suited for solving this image recognition tasks, but for the following section the idea of this problem setting can be kept.

4.4.1 One-hot encoding

In order to represent K classes or categories, a so-called *one-hot encoding* is typically used. The classes are encoded as follows: $\mathcal{C}_1 = (1, 0, \ldots, 0)^T$, $\mathcal{C}_k = (0, \ldots, 0, \underbrace{1}_{k\text{-th position}}, 0, \ldots, 0)^T$ and $\mathcal{C}_K = (0, 0, \ldots 0, 1)^T$. This has many advantages over encoding the categories as integers because it does not introduce an order of the categories.

4.4.2 Model

The softmax regression model maps the inputs $\boldsymbol{x}$ onto a probability vector of K categories:

$$g(\boldsymbol{x}, \boldsymbol{W}) = \boldsymbol{a} = \mathrm{softmax}(\boldsymbol{W}\boldsymbol{x}), \tag{4.49}$$

where we have a *weight matrix* $\boldsymbol{W} \in \mathbb{R}^{K \times D}$. The $\mathrm{softmax}$-function is defined as follows:

$$\mathrm{softmax}(\boldsymbol{s}) = \left(\frac{e^{s_1}}{\sum_{j=1}^{K} e^{s_j}}, \ldots, \frac{e^{s_k}}{\sum_{j=1}^{K} e^{s_j}}, \ldots, \frac{e^{s_K}}{\sum_{j=1}^{K} e^{s_j}} \right)^T. \tag{4.50}$$

From the definition of the softmax function it is clear that the resulting vector $\boldsymbol{a} = \mathrm{softmax}(\boldsymbol{s})$ is a probability vector, with non-negative entries that sum to one:

$$\sum_{k=1}^{K} \frac{e^{s_k}}{\sum_{j=1}^{K} e^{s_j}} = 1. \tag{4.51}$$

4.4.3 Loss function

For softmax regression we will employ the *categorical cross-entropy* as a loss function that is, again, motivated by a maximum likelihood criterion of the prediction $\boldsymbol{a}_n$ and the label $\boldsymbol{y}_n$ of a single object $\boldsymbol{x}$ (note the for each sample we now have a label *vector* instead of a scalar):

$$p(\boldsymbol{y}|\boldsymbol{a}) = \prod_{k=1}^{K} a_k^{y_k}, \tag{4.52}$$

from which we obtain the *categorical cross-entropy* by taking the negative log:

$$-\log p(\boldsymbol{y}|\boldsymbol{a}) = -\sum_{k=1}^{K} y_k \log(a_k). \tag{4.53}$$

Thus the empirical error on the training set $\boldsymbol{X}$ with label matrix $\boldsymbol{Y}$ is given by:

$$R_{\text{emp}}(\boldsymbol{w}, \boldsymbol{X}, \boldsymbol{Y}) = -\sum_{n=1}^{N}\sum_{k=1}^{K} y_{nk} \log(a_{nk}) = -\sum_{n=1}^{N}\sum_{k=1}^{K} (\boldsymbol{y}^n)_k \log((\boldsymbol{a}^n)_k), \tag{4.54}$$

where $(\boldsymbol{y}^n)_k$ and $(\boldsymbol{a}^n)_k$ are the k-th component of the n-th object's label vector or activation, respectively. This again gives rise to a convex optimization problem (see later) that has a unique global minimum.

4.4.4 Learning softmax regression: gradient descent

We can use gradient descent to find the minimum of the empirical error with an iterative procedure. Since the empirical error is a sum over data points, it is sufficient to derive the gradient for a single sample $\boldsymbol{x}$ with respect to a weight $w_{\xi d}$ in the data matrix $\boldsymbol{W}$:

$$\begin{aligned}
\frac{\partial}{\partial w_{\xi d}} \left(-\sum_{k=1}^{K} y_k \log(a_k) \right) &= -\frac{\partial}{\partial w_{\xi d}} \sum_{k=1}^{K} y_k \log \left(\frac{e^{\boldsymbol{w}_k^T \boldsymbol{x}}}{\sum_{j=1}^{K} e^{\boldsymbol{w}_j^T \boldsymbol{x}}} \right) \\
&= -\frac{\partial}{\partial s_k} \frac{\partial s_k}{\partial w_{\xi d}} \sum_{k=1}^{K} y_k \log \left(\frac{e^{s_k}}{\sum_{j=1}^{K} e^{s_j}} \right) \\
&= (a_\xi - y_\xi) x_d,
\end{aligned} \tag{4.55}$$

where $\boldsymbol{w}_k$ is the k-th row of the weight matrix $\boldsymbol{W}$, and we have used the *chain rule*. The variables $\boldsymbol{a} = (a_1, \ldots, a_k)$ and $\boldsymbol{s} = (s_1, \ldots, s_k)$ are defined as in Eq. (4.49) and Eq.(4.50). We also observe that the gradient with respect to the parameter matrix $\boldsymbol{W}$ can conveniently be written as an outer product: $\frac{\partial}{\partial \boldsymbol{W}} R_{\text{emp}}(\boldsymbol{w}, \boldsymbol{X}, \boldsymbol{Y}) = (\boldsymbol{a} - \boldsymbol{y})\boldsymbol{x}^T$.

4.5 Single-layer networks cannot solve XOR

We now consider a relatively simple problem that – despite its simplicity – cannot be solved by single-layer networks. Let us assume that obtain only four possible inputs $\boldsymbol{x}_1 = (0, 0)$, $\boldsymbol{x}_2 = (1, 0)$, $\boldsymbol{x}_3 = (0, 1)$, and $\boldsymbol{x}_4 = (1, 1)$ with the following labels $y_1 = 0$,$y_2 = 1$, $y_3 = 1$, and $y_4 = 0$, respectively. If we use a linear network

$$g_1(\boldsymbol{x}; \boldsymbol{w}) = x_1 w_1 + x_2 w_2, \tag{4.56}$$

it is impossible to find parameters w_1 and w_2 that solve this problem exactly because the equation system

$$\begin{aligned} 0 &= 0\, w_1 + 0\, w_2 \\ 1 &= 1\, w_1 + 0\, w_2 \\ 1 &= 0\, w_1 + 1\, w_2 \\ 0 &= 1\, w_1 + 1\, w_2. \end{aligned} \tag{4.57}$$

does not have a solution. Even by adding bias units or by adding a sigmoid, the problem cannot be solved. However, as soon as we use a two-layer network with a simple non-linear activation function:

$$g_2(\boldsymbol{x}; \boldsymbol{W}, \boldsymbol{w}, \boldsymbol{b}) = \boldsymbol{w}^T \max(0, \boldsymbol{W}\boldsymbol{x} + \boldsymbol{b}) = \boldsymbol{w}^T \boldsymbol{h}, \tag{4.58}$$

we can find parameters, that solve the problem. Concretely, $\boldsymbol{W} = \begin{pmatrix} 1 & 1 \\ 1 & 1 \end{pmatrix}$, $\boldsymbol{b} = \begin{pmatrix} 0 \\ -1 \end{pmatrix}$, and $\boldsymbol{w} = \begin{pmatrix} 1 \\ -2 \end{pmatrix}$ solve the problem in an exact way (Goodfellow et al., 2016). The transformation $\max(0, \boldsymbol{W}\boldsymbol{x} + \boldsymbol{b})$ has mapped the points $\boldsymbol{x}_1, \ldots, \boldsymbol{x}_4$ into a space, in which those data points a linearly separable.

We will find a more general results, concretely that neural networks with hidden layers are general function approximators (Hornik et al., 1989).

4.6 Multi-layer perceptron (MLP)

With the work of Rumelhart et al. (1986b,a) neural networks and especially the multi-layer perceptron trained with back-propagation became very popular. A multi-layer perceptron (MLP) consists of more than one perceptron in which the output of one perceptron is the input of the next perceptron. Further, the activation (state) of a neuron is computed through a nonlinear function.

In this section, we use the idea that we can enumerate all neurons in a network from $1, \ldots, Q$, where Q is the number of neurons, i.e. units, in a network.

4.6.1 Model and forward pass of an MLP

We define for the multi-layer perceptron (MLP), see Fig. 4.8:

- a_i: *activation* of the i-th unit
- $a_0 = 1$: activation of 1 of the bias unit
- w_{ij}: *weight* from unit j to unit i
- $w_{i0} = b_i$: *bias weight* of unit i

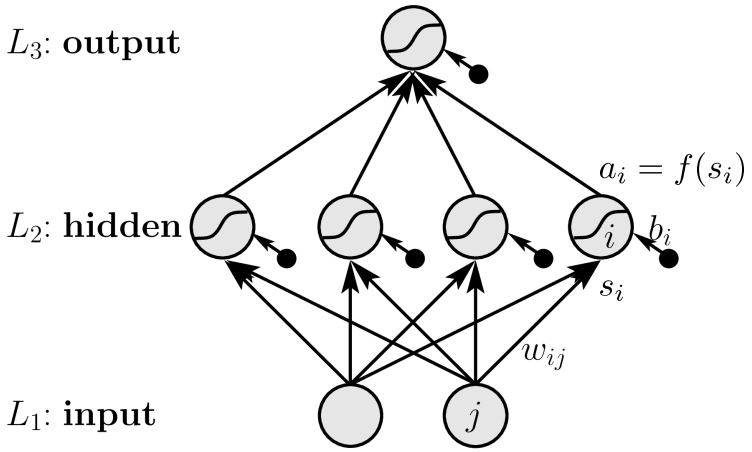


Figure 4.8: Figure of an MLP.

- W: number of weights
- Q: number of units
- D: number of inputs units ($1 \leq i \leq D$) located in the first layer called *input layer*.
- K: number of output units ($Q - K + 1 \leq i \leq Q$) located in the last layer called *output layer*.
- H: number of hidden units ($D < i \leq Q - K$) located in the hidden layers.
- L: number of layers, where L_ν is the index set of the ν-th layer; $L_1 = \{1, \ldots, D\}$ and $L_L = \{Q - K + 1, \ldots, Q\}$.
- s_i: *network input* to the i-th unit ($i > D$) computed as

$$s_i = \sum_{j=0}^{Q} w_{ij} a_j \tag{4.59}$$

- f: *activation function* with

$$a_i = f(s_i) \tag{4.60}$$

  It is possible to define different activation functions f_i for different units. The activation function is sometimes called *transfer function* (in more realistic networks one can distinguish between activation of a neuron and the signal which is transferred to other neurons).

- the *architecture* of a neural network is given through number of layers, units in the layers, and defined connections between units – the activations function may be accounted to the architecture.

A feed-forward MLP has only connections from units in lower layers to units in higher layers:

$$i \in L_\nu \text{ and } j \in L_{\nu'} \text{ and } \nu' \leq \nu \Rightarrow \boldsymbol{w}_{ij} = 0. \tag{4.61}$$

For a conventional multi-layer perceptron there are only connections or weights between consecutive layers. Other weights are fixed to zero. The network input is then for node i in hidden or output layer ν with $\nu > 1$

$$\forall_{i \in L_\nu} : s_i = \sum_{j \in L_{\nu-1}} w_{ij} a_j. \tag{4.62}$$

Connections between units in layers which are not adjacent are called *shortcut connections*.

The forward pass of a neural network is given in Alg. 4.1.

Algorithm 4.1 Forward Pass of an MLP

BEGIN initialization
  provide input $\boldsymbol{x}$
  for all $(i = 1; i \leq D;\ i++)$ **do**
    $a_i = x_i$
  end for
END initialization
BEGIN Forward Pass
  for $(\nu = 2; \nu \leq L; \nu++)$ **do**
    for all $i \in L_\nu$ **do**
      $s_i = \sum_{j=0; w_{ij} \text{ exists}}^{Q} w_{ij} a_j$
      $a_i = f(s_i)$
    end for
  end for
  provide output $\hat{y}_i = (g(\boldsymbol{x}; \boldsymbol{w}))_i = a_i$, for all $Q - K + 1 \leq i \leq Q$
END Forward Pass

4.6.2 Gradient descent and the backpropagation algorithm

Similar to before when we trained single-layer networks, such as linear regression, logistic regression and softmax regression, we aim at minimizing the empirical error by gradient descent.

The gradient of a neural network can be computed very efficiently by back-propagation (Rumelhart et al., 1986b,a) which was even earlier proposed by (Werbos, 1974). The back-propagation algorithm made artificial neural networks popular since the mid 80ies.

Gradient calculation. In the following we derive the back-propagation algorithm in order to compute the gradient of a neural network.

We define the *empirical error* as

$$R_{\text{emp}}(\boldsymbol{w}, \boldsymbol{X}, \boldsymbol{Y}) = \frac{1}{N} \sum_{n=1}^{N} L\left(\boldsymbol{y}^n, \boldsymbol{g}\left(\boldsymbol{x}^n; \boldsymbol{w}\right)\right), \tag{4.63}$$

where L is a loss function, e.g. mean squared error, the variable $\boldsymbol{w}$ now contains all adjustable weights of an MLP, that is all weight matrices $\boldsymbol{W}^{[1]}, \ldots, \boldsymbol{W}^{[L]}$ and potentially bias vectors $\boldsymbol{b}^{[1]}, \ldots, \boldsymbol{b}^{[L]}$ that connect layers. Note that we have also generalized to multiple outputs. Again, the loss function is just a mean over data points of the training set, thus, we can also consider the loss of a single sample $\boldsymbol{x}^n$.

The gradient descent update is given by

$$\boldsymbol{w}^{\text{new}} = \boldsymbol{w}^{\text{old}} - \eta \, \nabla_{\boldsymbol{w}} R_{\text{emp}}(\boldsymbol{w}, \boldsymbol{X}, \boldsymbol{Y}). \tag{4.64}$$

We select a weight w_{ij} in an arbitrary layer that connects from the unit j to unit i (see Figure 4.8).

$$\frac{\partial}{\partial w_{ij}} L\left(\boldsymbol{y}^n, \boldsymbol{g}\left(\boldsymbol{x}^n; \boldsymbol{w}\right)\right) \tag{4.65}$$

for all w_{ij}.

$$\begin{aligned} \frac{\partial}{\partial w_{ij}} L\left(\boldsymbol{y}^n, \boldsymbol{g}\left(\boldsymbol{x}^n; \boldsymbol{w}\right)\right) &= \frac{\partial}{\partial s_i} L\left(\boldsymbol{y}^n, \boldsymbol{g}\left(\boldsymbol{x}^n; \boldsymbol{w}\right)\right) \frac{\partial s_i}{\partial w_{ij}} \\ &= \frac{\partial}{\partial s_i} L\left(\boldsymbol{y}^n, \boldsymbol{g}\left(\boldsymbol{x}^n; \boldsymbol{w}\right)\right) a_j, \end{aligned} \tag{4.66}$$

where we have used that $\frac{\partial s_i}{\partial w_{ij}} = a_j$. We now define the δ-error at unit i as

$$\delta_i := \frac{\partial}{\partial s_i} L\left(\boldsymbol{y}^n, \boldsymbol{g}\left(\boldsymbol{x}^n; \boldsymbol{w}\right)\right) \tag{4.67}$$

and obtain

$$\frac{\partial}{\partial w_{ij}} L\left(\boldsymbol{y}^n, \boldsymbol{g}\left(\boldsymbol{x}^n; \boldsymbol{w}\right)\right) = \delta_i a_j. \tag{4.68}$$

It is advisable to study this result and its general form in more detail. We have found that the derivative of an MLP with respect to a weight going from neuron j to neuron i is equal to the "delta error" at neuron i multiplied by the activation of neuron j.

Comparing our results on the gradient of one layer networks, linear regression (Eq. (4.15)), logistic regression (Eq. (4.44)) and softmax regression (Eq. (4.55)), we find that all of those follow this principle form. We exemplify on the gradient of softmax regression:

$$\frac{\partial}{\partial w_{ij}} R_{\text{emp}}(\boldsymbol{w}, \boldsymbol{x}, y) = \underbrace{\left(\text{softmax}(\boldsymbol{W}\boldsymbol{x})_i - y_i\right)}_{=\delta_i} \underbrace{x_j}_{=a_j}, \tag{4.69}$$

because the input x_j to the softmax layer can be considered as an activation of unit d. Analogously, we find this structure in the gradients of linear regression and logistic regression models.

In the following, we investigate how the δ-quantities can be computed in different layers.

Delta errors at the output layers. The δ-error at the output units, denoted as δ_k with preactivation s_k and activation $a_k = g(\boldsymbol{x}^n; \boldsymbol{w})$ for $Q - K + 1 \leq k \leq Q$ is

$$\delta_k = \frac{\partial}{\partial a_k} L\left(\boldsymbol{y}^n, g\left(\boldsymbol{x}^n; \boldsymbol{w}\right)\right) f'(s_k) \tag{4.70}$$

where f' denotes the derivative of the activation function f of the output layer. This function f in the output layer is usually different from the activation functions in the hidden layers. The expression typically results in a difference between network outputs, i.e. predicted labels, and the labels, if the appropriate loss functions with their corresponding activations at the output units are used (see Table 4.1).

Loss function		output activation	delta-error
squared loss	$1/2(a-y)^2$	linear: $f(x) = x$	$\delta = (a - y)$
binary CE	$-y\log(a) - (1-y)\log(1-a)$	sigmoid: $f(x) = \frac{1}{1+e^{-x}}$	$\delta = (a - y)$
categorical CE	$-\sum_{k=1}^{K} y_k \log(a_k)$	softmax: $f(\boldsymbol{x}) = \mathrm{softmax}(\boldsymbol{x})$	$\boldsymbol{\delta} = (\boldsymbol{a} - \boldsymbol{y})$

Table 4.1: Frequently used loss functions for neural networks and their corresponding activation functions at the output units.

Delta errors at hidden layers. The δ-error at units not in the output layer, i.e. hidden layers, is

$$\begin{aligned}\delta_j &= \frac{\partial}{\partial s_j} L\left(\boldsymbol{y}^n, \boldsymbol{g}\left(\boldsymbol{x}^n; \boldsymbol{w}\right)\right) = \sum_i \frac{\partial}{\partial s_i} L\left(\boldsymbol{y}^n, \boldsymbol{g}\left(\boldsymbol{x}^n; \boldsymbol{w}\right)\right) \frac{\partial s_i}{\partial s_j} \\ &= f'(s_j) \sum_i \delta_i w_{ij},\end{aligned} \tag{4.71}$$

where the $\sum_i$ goes over all i for which w_{ij} exists. Typically i goes over all units in the layer above the layer where unit j is located.

Recap. We can now summarize the backpropagation algorithm in few steps:

- Select a sample $\boldsymbol{x}$ randomly from the training set and perform a forward-pass. Memorize activations and preactivations of all neurons.
- Calculate delta-errors at output units according to Eq. (4.70).
- Calculate delta-errors for hidden layers starting from the hidden layers closest to the output layer, thus backpropagating the error signal according to equation Eq. (4.71).
- Calculate gradients for weights according to Eq.(4.68).

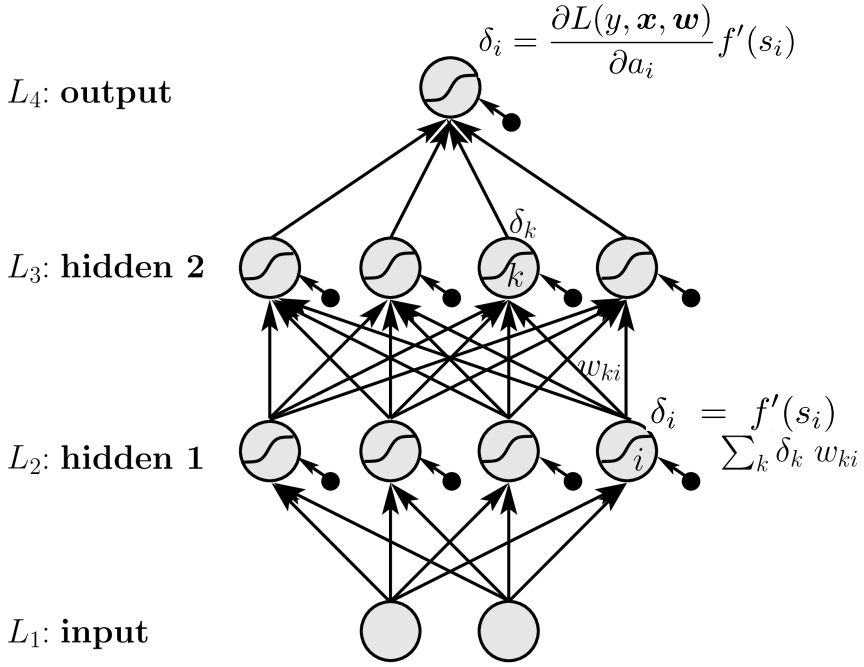


Figure 4.9: 4-layer MLP where the back-propagation algorithm is depicted. The $\delta_k = \frac{\partial}{\partial s_k} L\left(\boldsymbol{y}^n, \boldsymbol{g}\left(\boldsymbol{x}^n; \boldsymbol{w}\right)\right)$ are computed from the δ's of the higher layer. The back-propagation algorithm starts from the top layer and ends with the layer 2 where the weights to layer 2 are updated.

- Update the weights by performing a gradient descent step using the weight changes $\Delta w_{ij} = -\eta \delta_i a_j$. The learning rate η must be chosen appropriately for the algorithm to converge. Typical learning rates are $\eta = 0.1$ or $\eta = 0.01$.

Efficiency of backprop. This efficient computation of the δ-errors led to the term "back-propagation" because the δ-errors of one layer are used to compute the δ-errors of the layer below. The algorithm is sometimes also called "δ-propagation". Fig. 4.9 depicts the back-propagation algorithm for a 4-layer feed-forward network.

Remarks. Note that for the logistic function $f(a) = \frac{1}{1+\exp(-a)}$ the derivative is

$$f'(a) = f(a)\left(1 - f(a)\right). \tag{4.72}$$

This can be used to speed up the algorithm because the exponential function must only be evaluated once. Alg. 4.2 gives the backward pass for a single example where the weight update is accumulated in Δw_{ij}. Note that Alg. 4.2 has complexity of $O(W)$ which is very efficient.

4.6.3 Training MLPs.

Besides the many trainable parameters, i.e. weights, MLPs come with many hyperparameters, such as the number of layers, number of neurons, choice of activation function, initialization, etc.,

Algorithm 4.2 Backward Pass of an MLP

BEGIN initialization
provide activations a_i of the forward pass and the label y
for $(i = Q - K + 1; i \leq Q; i++)$ **do**
$\delta_i = \frac{\partial L(y, \boldsymbol{x}, \boldsymbol{w})}{\partial a_i} f'(s_i)$
for all $j \in L_{L-1}$ **do**
$\Delta w_{ij} = -\eta \delta_i a_j$
end for
end for
END initialization
BEGIN Backward Pass
for $(\nu = L - 1; \nu \geq 2; \nu --)$ **do**
for all $i \in L_\nu$ **do**
$\delta_i = f'(s_i) \sum_k \delta_k w_{ki}$
for all $j \in L_{\nu-1}$ **do**
$\Delta w_{ij} = -\eta \delta_i a_j$
end for
end for
end for
END Backward Pass

that have to be adjusted by a human expert[2]. Insights into this topic are given in the one of the following sections (Section 4.10). One of the early works dedicated to this topic is by LeCun et al. (1998b).

[2]Nowadays, also automated procedures and even neural networks are used to adjust those hyperparameters, but for simplicity we consider these choices to be taken by a human expert.

Table 4.2: Comparison of typical multi-layer perceptrons and deep feed-forward neural networks (informal guidelines).

	MLP	**DNN**
Number of hidden layers	1 or few	> 1 up to several hundreds
Number of neurons per layer	< 10	> 100
Activation function	tanh, sigmoid	ReLU, SELU

4.7 (Deep) Feed-forward neural networks

Deep feed-forward neural networks (DNNs) have the same principal structure as multi-layer perceptrons. However, DNNs comprise many layers and thousands of neurons in each layer, whereas MLPs typically have few neurons and a single hidden layer. The discrimination between these networks is historically grown and there are no clear definitions. The definition of *deep* is usually that a network has more than one hidden layer. In Table 4.2, we report typical differences between such networks.

Note: DNNs and MLPs

Deep feed-forward networks have the same principle structure as multiple layer perceptrons. However, DNNs typically contain many more neurons and layers, different activation functions, and some other differences. This section has the main purpose of introducing vectorial notation for the forward and backward pass. Note that multiple names for these types of networks are used and are usually equivalent: (deep) multi-layer perceptron (MLP), (deep) fully-connected network (FCN), and (deep) feed-forward neural network (FNN).

In the machine learning community, *matrix-vector notation* is typically used to denote the forward pass of an MLP. We denote the following quantities:

- $\boldsymbol{x}$: input for layer 0; equivalent to $\boldsymbol{a}^{[0]}$
- $\boldsymbol{W}^{[l]}$: weight matrix connecting layer $l-1$ and layer l
- $\boldsymbol{s}^{[l]}$: pre-activations of layer l
- $\boldsymbol{a}^{[l]}$: activations of layer l
- f: activation function that is applied element-wise to a vector.

Given the inputs $\boldsymbol{x}^{[l]}$ to a layer l, i.e. the activations $\boldsymbol{a}^{[l-1]}$ of the lower layer $l-1$, an MLP computes the activations of the layer in the following way (a schematic representation can be found in Figure 4.10):

$$\boldsymbol{a}^{[l]} = f(\boldsymbol{s}^{[l]}) = f(\boldsymbol{W}^{[l]}\boldsymbol{a}^{[l-1]}) \tag{4.73}$$

Thus the function from the input layer $\boldsymbol{x}$ to the output layer takes the following form:

$$\hat{\boldsymbol{y}} = g(\boldsymbol{x}; \boldsymbol{W}^{[1]}, \ldots, \boldsymbol{W}^{[L]}) = \sigma(\boldsymbol{W}^{[L]}(\ldots f(\boldsymbol{W}^{[2]} f(\boldsymbol{W}^{[1]}\boldsymbol{x}))\ldots), \tag{4.74}$$

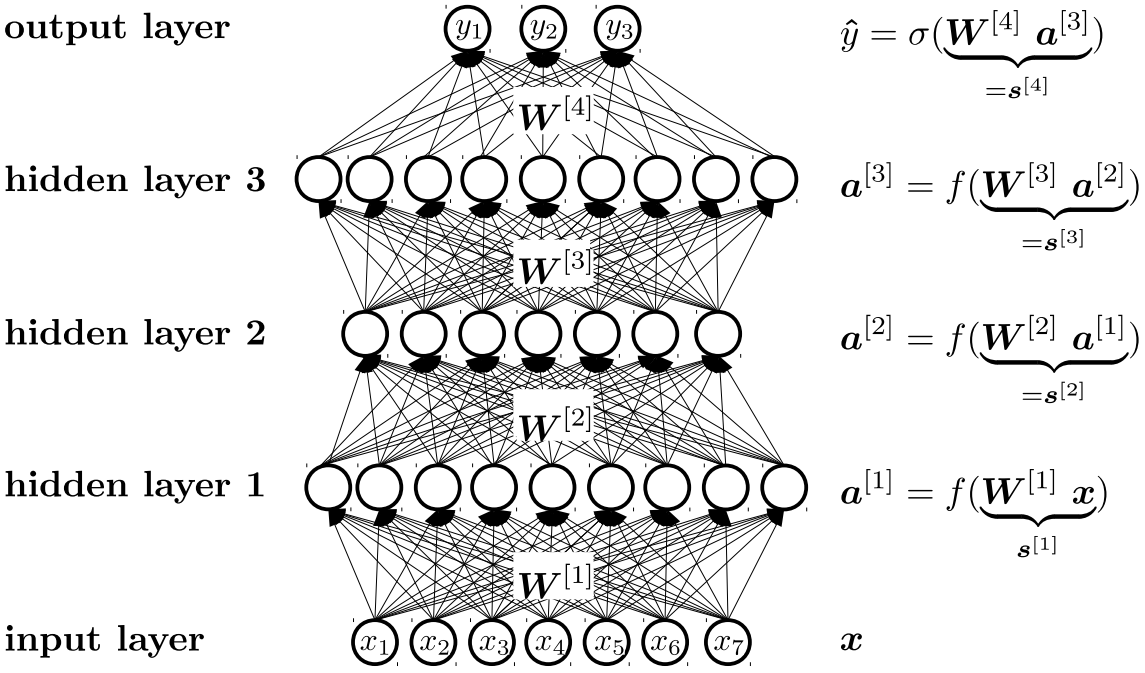


Figure 4.10: Schematic view of a deep feed-forward neural network. Bias units are omitted from the notation. As a simplification, only few neurons are displayed, whereas typical DNNs comprise thousands of neurons.

where $\hat{\boldsymbol{y}}$ are the activations of the output layer and $\boldsymbol{x}$ is the input of the network. Note that the outermost activation function σ is typically different from the activation functions in the hidden layer f.

Bias units and implicit biases. Each layer in a DNN can employ bias units $\boldsymbol{b}^{[1]}, \ldots, \boldsymbol{b}^{[L]}$, which means that the forward pass is given by:

$$\boldsymbol{a}^{[l]} = f(\boldsymbol{s}^{[l]}) = f(\boldsymbol{W}^{[l]}\boldsymbol{a}^{[l-1]} + \boldsymbol{b}^{[l]}). \tag{4.75}$$

4.7.1 Backpropagation in DNNs

The backpropagation algorithm is identical to the case of MLPs. Nevertheless, in this section, we provide the backprop algorithm in a different way, in which we use matrix-vector notation and derivatives.

Without loss of generality, we take the derivative of the loss function with respect to a weight of the first weight matrix. To this end, we use the chain rule in the following way:

$$\frac{\partial}{\partial w_{ij}^{[1]}} L(\boldsymbol{y}, g(\boldsymbol{x}; \boldsymbol{W}^{[1]}, \ldots, \boldsymbol{W}^{[L]})) = \underbrace{\frac{\partial L(\boldsymbol{y}, \hat{\boldsymbol{y}})}{\partial \hat{\boldsymbol{y}}} \frac{\partial \hat{\boldsymbol{y}}}{\partial \boldsymbol{s}^{[L]}}}_{=:A} \cdot \ldots \cdot \underbrace{\frac{\partial \boldsymbol{s}^{[l]}}{\partial \boldsymbol{s}^{[l-1]}}}_{=:B} \cdot \ldots \cdot \underbrace{\frac{\partial \boldsymbol{s}^{[1]}}{\partial w_{ij}^{[1]}}}_{=:C}, \tag{4.76}$$

where we have denoted three expressions A, B and C that we will now investigate in further detail. We define the delta-errors as the derivative of the loss function with respect to the pre-activations

in the respective layer:

$$\boldsymbol{\delta}^{[l]} = \left(\frac{\partial L(\boldsymbol{y}, \hat{\boldsymbol{y}})}{\partial \boldsymbol{s}^{[l]}} \right)^T. \tag{4.77}$$

Note that we typically use these deltas ${\boldsymbol{\delta}^{[l]}}^T$ as *row vectors*.

- (A) $\frac{\partial L(\boldsymbol{y},\hat{\boldsymbol{y}})}{\partial \hat{\boldsymbol{y}}} \frac{\partial \hat{\boldsymbol{y}}}{\partial \boldsymbol{s}^{[L]}}$: The derivative of the loss function with respect to the preactivations of the output layer. This is equivalent to the gradients of single-layer networks (see Sections 4.1,4.3 and 4.4). For some combinations of output activation function and loss, this expression results in a simple term, such as

$$\boldsymbol{\delta}^{[L]} = (\hat{\boldsymbol{y}} - \boldsymbol{y})^T. \tag{4.78}$$

- (B) $\frac{\partial \boldsymbol{s}^{[l]}}{\partial \boldsymbol{s}^{[l-1]}}$: The derivative of the pre-activations of layer l with respect to the pre-activations of layer $l-1$. This expression is used to propagate derivatives from layer to layer. In the forward pass we calculate,

$$\boldsymbol{s}^{[l]} = \boldsymbol{W}^{[l]} f(\boldsymbol{s}^{[l-1]}), \tag{4.79}$$

with an activation function f that is applied element-wise. The Jacobian of such an element-wise operation is a diagonal matrix with the dimensions of $\boldsymbol{s}^{[l-1]}$. For the backward pass we thus obtain,

$$\frac{\partial \boldsymbol{s}^{[l]}}{\partial \boldsymbol{s}^{[l-1]}} = \boldsymbol{W}^{[l]} \operatorname{diag}\left(f'(\boldsymbol{s}^{[l-1]}) \right), \tag{4.80}$$

where $\operatorname{diag}\left(f'(\boldsymbol{s}^{[l-1]}) \right)$ is a diagonal matrix that contains the values of $f'(\boldsymbol{s}^{[l-1]})$ on its diagonal. We see that the result is just the weight matrix multiplied with a diagonal matrix from the right, which amounts to multiplying each column in the weight matrix with a different scalar.

We can use this to deduce a *recursive formula* for the delta-errors

$${\boldsymbol{\delta}^{[l-1]}}^T = \frac{\partial L(\boldsymbol{y}, \hat{\boldsymbol{y}})}{\partial \boldsymbol{s}^{[l]}} \frac{\partial \boldsymbol{s}^{[l]}}{\partial \boldsymbol{s}^{[l-1]}} = {\boldsymbol{\delta}^{[l]}}^T \underbrace{\boldsymbol{W}^{[l]} \operatorname{diag}\left(f'(\boldsymbol{s}^{[l-1]}) \right)}_{\boldsymbol{J}^{[l]}}, \tag{4.81}$$

where all $\boldsymbol{\delta}$ are row vectors and we define the expression on the right hand side of $\boldsymbol{\delta}^{[l]}$ as single-layer Jacobian $\boldsymbol{J}^{[l]}$ (see also Section 4.8).

- (C) $\frac{\partial \boldsymbol{s}^{[1]}}{\partial w_{ij}^{[1]}}$: The derivative of the delta-errors with respect to the networks weights.

$$\frac{\partial \boldsymbol{s}^{[1]}}{\partial w_{ij}^{[1]}} = \frac{\partial}{\partial w_{ij}^{[1]}} \boldsymbol{W}^{[1]} \boldsymbol{a}^{[0]} = \begin{pmatrix} 0 \\ \vdots \\ a_j^{[0]} \\ \vdots \\ 0 \end{pmatrix}, \tag{4.82}$$

where the non-zero entry $a_j^{[0]}$ is at the i-th position. Thus we have

$$\frac{\partial L(\boldsymbol{y}, \hat{\boldsymbol{y}})}{\partial w_{ij}^{[l]}} = \frac{\partial L(\boldsymbol{y}, \hat{\boldsymbol{y}})}{\partial \boldsymbol{s}^{[l]}} \frac{\partial \boldsymbol{s}^{[l]}}{\partial w_{ij}^{[l]}} = (\boldsymbol{\delta}^{[l]})_i (\boldsymbol{a}^{[l-1]})_j \tag{4.83}$$

and we find that the derivative with respect to the full weight matrix is an outer product of the activations of the lower layer with the delta-errors of the higher layer:

$$\frac{\partial L(\boldsymbol{y}, \hat{\boldsymbol{y}})}{\partial \boldsymbol{W}^{[l]}} = \boldsymbol{a}^{[l-1]} {\boldsymbol{\delta}^{[l]}}^T. \tag{4.84}$$

Summary. In order to obtain gradients of the loss function, we first have to perform a forward pass through a network using formula Eq.(4.74) memorizing all pre-activations $\boldsymbol{s}$ and activations $\boldsymbol{a}$. Then, the delta-errors at the output units have to be calculated, typically using Eq.(4.78). The delta-errors have to be calculated recursively starting at the layer closest to the output layer using Eq.(4.81). Finally the gradients with respect to the weights can be calculated using Eq.(4.84).

Note: Vanishing and exploding gradients (Hochreiter, 1991)

From the recursion formula above (4.81)

$${\boldsymbol{\delta}^{[l-1]}}^T = {\boldsymbol{\delta}^{[l]}}^T \boldsymbol{W}^{[l]} \operatorname{diag}\left(f'(\boldsymbol{s}^{[l-1]})\right),$$

we find an important property of neural network that influences the ability of a network to learn: the size (norm) of the delta-errors that backpropagated through the network. Since for each layer, the delta errors are multiplied by the Jacobian they exhibit an exponential behaviour (growth or shrinkage) across layers. The following principal options exist:

- $\|\boldsymbol{\delta}^{[l-1]}\| < \|\boldsymbol{\delta}^{[l]}\|$: **Vanishing gradients.** This has been the typical case since the derivatives of the sigmoid activation function is at most $|f'(x)| = 0.25$, the norm $\|\operatorname{diag}\left(f'(\boldsymbol{s}^{[l-1]})\right)\| \leq 0.25$ and the norm of the delta-errors becomes smaller through each layer (if not compensated by the norm of $\boldsymbol{W}^{[l]}$).
- $\|\boldsymbol{\delta}^{[l-1]}\| \approx \|\boldsymbol{\delta}^{[l]}\|$: **Stable gradients.** This would be the ideal case that the delta errors have a similar norm in each layer, and in fact many recent algorithmic improvements aim at keeping this quantity close to one. For example, initialization strategies, see Chapter 9.
- $\|\boldsymbol{\delta}^{[l-1]}\| > \|\boldsymbol{\delta}^{[l]}\|$: **Exploding gradients.** If the norm of the weight matrix is large, this scenario can occur, in which the norm of the delta-errors grow through each layer. Large weight updates lead to unstable learning or even numeric overflows.

4.8 Jacobian

Although the term "the Jacobian" of a neural network is frequently used it is an ambiguous term should not be used without further explanation. There are at least three interpretations:

1. the usual derivative of the loss function with respect to the weights $\frac{\partial L(\boldsymbol{y}, g(\boldsymbol{x};\boldsymbol{w}))}{\partial \boldsymbol{w}}$.
2. The derivative of the network outputs with respect to the inputs $\frac{\partial g(\boldsymbol{x};\boldsymbol{w})}{\partial \boldsymbol{x}}$,
3. the derivative of the outputs of a particular layer with respect to the layer inputs $\frac{\partial}{\partial \boldsymbol{x}} f(\boldsymbol{W}\boldsymbol{x})$ (see below).

Thus, to be precise with formulation, we suggest to specify the type of Jacobian that is meant: *parameter Jacobian*, *input-output Jacobian* or the *single-layer Jacobian*.

In the sections above, we were concerned with the derivative of the empirical error, or a loss function, with respect to the *weights*. The backpropagation technique can however also be applied to the calculation of the neural network outputs with respect to the inputs or the outputs of a layer with respect to the layer inputs.

Input-output Jacobian Here we consider the evaluation of the input-output Jacobian, whose elements are given by the derivatives of the network outputs with respect to the inputs

$$\mathcal{J} = \frac{\partial \hat{\boldsymbol{y}}}{\partial \boldsymbol{x}}, \tag{4.85}$$

where $\hat{\boldsymbol{y}} = g(\boldsymbol{x};\boldsymbol{w})$ are the predictions of a neural network and $\boldsymbol{x}$ are the inputs and $\mathcal{J} \in \mathbb{R}^{K\times D}$.[3] This Jacobian provides a measure of the local sensitivity of the outputs to changes in each of the input variables. This can be used in the context of interpreting a neural networks predictions. The calculation of this derivatives is analogous to backpropagation.

Example: Deep Dream The input-output Jacobian can be used to optimize the input $\boldsymbol{x}$ to maximize a particular output unit y_k. This has been utilized – with some necessary adaptions (Mordvintsev et al., 2015) – to change input images towards an output class. The resulting images can be considered as visualization of features that are learned by a neural networks (see Figure 4.11).

Single-layer Jacobian Another type of Jacobian is the single-layer Jacobian that we obtain from the function from one layer to the next

$$\boldsymbol{a} = f(\boldsymbol{W}\boldsymbol{x}), \tag{4.86}$$

where $\boldsymbol{x}$ is the input to an arbitrary layer (not necessarily the input layer). Then

$$\boldsymbol{J} = \frac{\partial \boldsymbol{a}}{\partial \boldsymbol{x}} = \operatorname{diag}\left(f'(\boldsymbol{W}\boldsymbol{x})\right)\boldsymbol{W}, \tag{4.87}$$

with $\boldsymbol{J} \in \mathbb{R}^{I\times J}$, is the single-layer Jacobian of the transformation of one fully-connected layer. We will use this expression in the context of the *vanishing gradient problem*.

[3] We will use $\mathcal{J}$ to denote this Jacobian and $\boldsymbol{J}$ to denote the single-layer Jacobian

Figure 4.11: Examples of input images that were maximized for a particular output unit using information of the input-output Jacobian. A pre-trained image recognition neural network was used as $g(\boldsymbol{x}; \boldsymbol{w})$. Note that straight-forward optimization of $\boldsymbol{x}$ is not sufficient, but adaptions are necessary (Mordvintsev et al., 2015).

4.9 Hessian

Up to now, we have considered the first-order derivatives of the empirical error or loss function with respect to the parameters $\boldsymbol{w}$ of a neural network. The second-order derivatives form the so-called *Hessian matrix* $\boldsymbol{H}$:

$$\boldsymbol{H}_{ij} = \frac{\partial^2 L(\boldsymbol{y}, g(\boldsymbol{x}; \boldsymbol{w}))}{\partial w_i \, \partial w_j}, \tag{4.88}$$

where all parameters of the neural network are written as vector $\boldsymbol{w}$. The Hessian matrix can be quite large for Deep Neural Networks with millions of parameters since it is of size $Q \times Q$, where Q is the number of parameters. This vast size often prohibits or limits the use of second-order optimization methods. The Hessian can also be calculated using back-propagation.

The Hessian matrix can in principle be used for non-linear optimization algorithms, pruning via the inverse Hessian (see later), assigning error bars to predictions, and several other methods that will be discussed later.

Furthermore, we will sometimes use the second order Taylor approximation of the error function $R(\boldsymbol{w})$ at $\boldsymbol{w}_0$ that involves the Hessian

$$R(\boldsymbol{w}) = R(\boldsymbol{w}_0) + (\boldsymbol{w} - \boldsymbol{w}_0)^T \boldsymbol{g} + \frac{1}{2}(\boldsymbol{w} - \boldsymbol{w}_0)^T \boldsymbol{H} (\boldsymbol{w} - \boldsymbol{w}_0) + \mathcal{O}\left((\boldsymbol{w} - \boldsymbol{w}_0)^3\right), \tag{4.89}$$

where $\boldsymbol{g}$ is the gradient of R at $\boldsymbol{w}_0$ and $\boldsymbol{H}$ is the Hessian of R at $\boldsymbol{w}_0$.

4.10 Efficient Training of DNNs and MLPs: Basic tricks of the trade

Although at this point, it might seem easy to implement a neural network and the backpropagation algorithm, there can be considerable problems when it comes to training a neural network and optimizing its performance. There are many pitfalls that can hamper a practitioner from successfully training a neural network model and use it in prediction or production mode. One early work that considers this problem and provides some solutions and ameliorations is LeCun et al. (1998b).

Remark. Note that statements in this section are often only backed empirically or treated later theoretically or practically at full length, for example "initialization". Therefore, this part should for now be seen as a help to run and train neural networks for exercise.

4.10.1 Online, stochastic and batch learning

For standard backpropagation, at each iteration, we would be required to perform a complete pass through the training set in order to obtain all the quantities necessary for Eq.(4.64). This is called *full-batch learning* because the entire batch of training data has to be considered before the weights are updated. However, noting that the main expression in Eq.(4.64) is an average over all training

data points, we can use stochastic learning, in which we calculate the average by a random subset of trainng data points:

$$\nabla_{\boldsymbol{w}} R_{\text{emp}}(\boldsymbol{w}, \boldsymbol{X}, \boldsymbol{y}) = \frac{1}{N} \sum_{n=1}^{N} \nabla_{\boldsymbol{w}} L(y^n, g(\boldsymbol{w}, \boldsymbol{x}^n)) \approx \frac{1}{B} \sum_{b=1}^{B} \nabla_{\boldsymbol{w}} L(y^{n_b}, g(\boldsymbol{w}, \boldsymbol{x}^{n_b})), \quad (4.90)$$

where $\mathcal{B} = \{\boldsymbol{x}_{n_1}, \ldots, \boldsymbol{x}_{n_B}\} \subset \{\boldsymbol{x}_1, \ldots, \boldsymbol{x}_N\}$ (with $|\mathcal{B}| = B$) is a small subset[4] of the training set $\boldsymbol{X}$. This variant is called *stochastic gradient descent (SGD)* and $\mathcal{B}$ is called *mini-batch*. When the batch size B is one, we speak of *on-line learning*: a single sample is propagated through the network, causes an error and thus a gradient and subsequently the weights are slightly adjusted.

SGD is typically much faster than full-batch gradient descent, particularly in large redundant data sets, and also introduces some amount of *regularization* to the learning process.

4.10.2 Sampling from training data, batch size, epochs and learning curves

For SGD, we have to randomly draw a mini-batch from the training set. To this end, typically *sampling without replacement* is performed by uniformly sampling a mini-batch from the training set and then removing these samples from the training set. When the training set is "depleted", a so-called *epoch* has ended and another training epoch starts with a full training set. An *epoch* is a set of subsequent updates to the neural network, in which each sample is propagated once through the network. Typical numbers of epochs required to train an neural networks are 50, 100 or 300, but this depends strongly on the size of the training set in relation to the complexity of the task.

In order to monitor the training process of the neural network, experienced practitioners monitor the loss on the training set of each *mini-batch update*, and after each epoch the loss on a validation set. This results in graphs of updates (x-axis) versus loss on the training/validation set on the y-axis. Additionally, the performance measure of interest, e.g. accuracy should be traced on both the training and the validation set.

4.10.3 Normalizing inputs and outputs

The convergence of neural networks is usually faster if the mean of the input variables is close to zero. If this is not the case, we speak of a so-called *bias shift*. As a simple, but intuitive example, assume that for an input $\boldsymbol{x}$ all elements are positive ($x_j > 0 \quad \forall j$). Now, consider a neuron in the next layer that has a negative (positive) δ_i, then all weights will increase (decrease) $\Delta w_{ij} \propto \delta_i x_j$ together. In order to decouple this behaviour, a weight vector has to "zig-zag" over updates[5]. Therefore, input variables should be shifted to zero mean.

Similar to the shifting the inputs to zero mean, the inputs should also have unit variance. Here, the intuitive explanation is the following: A high variance of the inputs, leads also to a high variance in the first hidden layer, which means we will encounter many very high and low values as network inputs s_i. In the backward pass, we will evaluate the quantities $f'(s_i)$ for the

[4] Practically the size of such a mini-batch is often 32 or 64.

[5] Note that there are more explanations why a bias shift hampers learning.

derivative of the sigmoid f at those values. However, for values larger than 3 and smaller than -3, the derivative is very flat (see Figure A.1) and we will hence obtain very small delta-errors. This means that weights are hardly updated. Therefore, input variables should be scaled to unit variance. For more details, see chapter 9.

4.10.4 Initialization

With similar lines of thought as before, weights should be initialized with zero mean and variance $1/J$, where J is the number of input units of the particular layer. For layers using ReLU-activations, the variance should be set to $2/J$. For more details, see chapter 9 and chapter 10.

4.10.5 Learning rates

The learning rate η is one of the most important hyperparameters and is crucial to allow the neural network training to converge. Therefore, the learning rate has to be adjusted using a validation set. A valid approach is to start with high learning rates and decrease the learning rate on a logarithmic scale until the network converges. Typical values for η are 0.1 and 0.01 for standard stochastic gradient descent. For more information, see chapter 7

4.10.6 Number of neurons and hidden layers

The number of neurons and number of hidden layers that are appropriate for a proficient neural network, depend predominantly on the complexity of the problem at hand. For simple problems, linear networks or single-layer networks are sufficient and will provide the best predictive performance. More complex problem, require more neurons and more layers, which also increase the capacity of the neural network. As a rule-of-thumb the network should be designed such that it can *overfit* to the training set when it is used without regularization techniques. Subsequently, regularization should be added to equip the network with the ability to generalize. Single-layer networks should serve as a baseline.

Chapter 5

Generalization error, test set and cross-validation

SEPP HOCHREITER AND GÜNTER KLAMBAUER

In this chapter, we define the performance of a model on future data for the supervised case. The performance of a model on future data is called *generalization error*. For the supervised case an error for each example can be defined and then averaged over all possible examples. The error on one example is called *loss* but also *error*. The expected loss is called *risk*.

5.1 Definition of the Generalization Error / Risk

We assume that *objects* $x \in \mathcal{X}$ from an object set $\mathcal{X}$ are represented or described by *feature vectors* $\boldsymbol{x} \in \mathbb{R}^d$.

The *training set* consists of N objects $X = \left\{\boldsymbol{x}^1, \ldots, \boldsymbol{x}^n, \ldots, \boldsymbol{x}^N\right\}$ with a characterization $y^n \in \mathbb{R}$ like a label or an associated value which must be predicted for future objects. For simplicity we assume that y^n is a scalar, the so-called *target*. For simplicity we will write $\boldsymbol{z} = (\boldsymbol{x}, y)$ and $Z = X \times \mathbb{R}$.

The *training data* is $\left\{\boldsymbol{z}^1, \ldots, \boldsymbol{z}^N\right\}$ with a sample $\boldsymbol{z}^n = (\boldsymbol{x}^n, y^n)$, where we will later use the *matrix of feature vectors* $\boldsymbol{X} = \left(\boldsymbol{x}^1, \ldots, \boldsymbol{x}^N\right)^T$, the *vector of labels* $\boldsymbol{y} = \left(y^1, \ldots, y^N\right)^T$, and the *training data matrix* $\boldsymbol{Z} = \left(\boldsymbol{z}^1, \ldots, \boldsymbol{z}^N\right)$ ("^{T}" means the transposed of a matrix and here it makes a column vector out of a row vector).

In order to compute the performance on the future data we need to know the future data and need a quality measure for the deviation of the prediction from the true value, i.e. a *loss function*.

The future data is not known, therefore, we need at least the probability that a certain data point is observed in the future. The data generation process has a density $p\left(\boldsymbol{z}\right)$ at $\boldsymbol{z}$ over its data space. For finite discrete data $p\left(\boldsymbol{z}\right)$ is the probability of the data generating process to produce $\boldsymbol{z}$. $p\left(\boldsymbol{z}\right)$ is the *data probability*.

The loss function is a function of the target and the model prediction. The model prediction is given by a function $g(\boldsymbol{x})$ and if the models are parameterized by a parameter vector $\boldsymbol{w}$ the model prediction is a parameterized function $g(\boldsymbol{x}; \boldsymbol{w})$. Therefore the loss function is $\mathcal{L}(y, g(\boldsymbol{x}; \boldsymbol{w}))$.

Typical loss functions are the *quadratic loss* $\mathcal{L}(y, g(\boldsymbol{x}; \boldsymbol{w})) = (y - g(\boldsymbol{x}; \boldsymbol{w}))^2$ or the zero-one loss function

$$\mathcal{L}(y, g(\boldsymbol{x}; \boldsymbol{w})) = \begin{cases} 0 & \text{for} \quad y = g(\boldsymbol{x}; \boldsymbol{w}) \\ 1 & \text{for} \quad y \neq g(\boldsymbol{x}; \boldsymbol{w}) \end{cases} \tag{5.1}$$

Now we can define the *generalization error* which is the expected loss on future data, also called *risk* R_{gen} (a functional, i.e. a operator which maps functions to scalars):

$$R_{\text{gen}}(g(.; \boldsymbol{w})) = \mathbb{E}_{\boldsymbol{z}} \left[\mathcal{L}(y, g(\boldsymbol{x}; \boldsymbol{w})) \right]. \tag{5.2}$$

The risk for the quadratic loss is called *mean squared error*.

$$R_{\text{gen}}(g(.; \boldsymbol{w})) = \int_Z \mathcal{L}(y, g(\boldsymbol{x}; \boldsymbol{w})) p(\boldsymbol{z}) \, d\boldsymbol{z}. \tag{5.3}$$

In many cases we assume that y is a function of $\boldsymbol{x}$, the *target function* $f(\boldsymbol{x})$, which is disturbed by noise

$$y = f(\boldsymbol{x}) + \epsilon, \tag{5.4}$$

where ϵ is a noise term drawn from a certain distribution $p^n(\epsilon)$, thus

$$p(y \mid \boldsymbol{x}) = p^n(y - f(\boldsymbol{x})). \tag{5.5}$$

Here the probabilities can be rewritten as

$$p(\boldsymbol{z}) = p(\boldsymbol{x}) \, p(y \mid \boldsymbol{x}) = p(\boldsymbol{x}) \, p^n(y - f(\boldsymbol{x})). \tag{5.6}$$

Now the risk can be computed as

$$\begin{aligned} R_{\text{gen}}(g(.; \boldsymbol{w})) &= \int_Z \mathcal{L}(y, g(\boldsymbol{x}; \boldsymbol{w})) p(\boldsymbol{x}) \, p^n(y - f(\boldsymbol{x})) d\boldsymbol{z} \\ &= \int_X p(\boldsymbol{x}) \int_{\mathbb{R}} \mathcal{L}(y, g(\boldsymbol{x}; \boldsymbol{w})) p^n(y - f(\boldsymbol{x})) dy d\boldsymbol{x}, \end{aligned} \tag{5.7}$$

where

$$\begin{aligned} R_{\text{gen}}(g(\boldsymbol{x}; \boldsymbol{w})) &= \mathbb{E}_{y|\boldsymbol{x}} \left[\mathcal{L}(y, g(\boldsymbol{x}; \boldsymbol{w})) \right] \\ &= \int_{\mathbb{R}} \mathcal{L}(y, g(\boldsymbol{x}; \boldsymbol{w})) p^n(y - f(\boldsymbol{x})) dy. \end{aligned} \tag{5.8}$$

The noise-free case is $y = f(\boldsymbol{x})$, where $p^n = \delta$ can be viewed as a Dirac delta-distribution:

$$\int_{\mathbb{R}} h(\boldsymbol{x}) \delta(\boldsymbol{x}) d\boldsymbol{x} = h(\boldsymbol{0}) \tag{5.9}$$

therefore

$$R_{\text{gen}}(g(\boldsymbol{x}; \boldsymbol{w})) = \mathcal{L}(f(\boldsymbol{x}), g(\boldsymbol{x}; \boldsymbol{w})) = \mathcal{L}(y, g(\boldsymbol{x}; \boldsymbol{w})) \tag{5.10}$$

and eq. (5.3) simplifies to

$$R_{\text{gen}}(g(.; \boldsymbol{w})) = \int_X p(\boldsymbol{x}) \, \mathcal{L}(f(\boldsymbol{x}), g(\boldsymbol{x}; \boldsymbol{w})) d\boldsymbol{x}. \tag{5.11}$$

Because we do not know $p(\boldsymbol{z})$ the risk cannot be computed; especially we do not know $p(y \mid \boldsymbol{x})$. In practical applications we have to approximate the risk.

To be more precise, the parameters depend on the training set, hence $\boldsymbol{w} = \boldsymbol{w}(\boldsymbol{Z})$.

5.2 Empirical Estimation of the Generalization Error

Here we describe some methods how to estimate the risk (generalization error) for a certain model.

5.2.1 Test Set

We assume that data points $\boldsymbol{z} = (\boldsymbol{x}, y)$ are iid (independent identical distributed) and, therefore also $\mathcal{L}(y, g(\boldsymbol{x}; \boldsymbol{w}))$, and $\mathbb{E}_{\boldsymbol{z}} \left[\left| \mathcal{L}(y, g(\boldsymbol{x}; \boldsymbol{w})) \right| \right] < \infty$.

The risk is an expectation of the loss function:

$$R_{\text{gen}}(g(.; \boldsymbol{w})) = \mathbb{E}_{\boldsymbol{z}} \left[\mathcal{L}(y, g(\boldsymbol{x}; \boldsymbol{w})) \right], \tag{5.12}$$

therefore this expectation can be approximated using the (strong) law of large numbers:

$$R_{\text{gen}}(g(.; \boldsymbol{w})) \approx \frac{1}{M} \sum_{n=N+1}^{N+M} L\left(y^n, g(\boldsymbol{x}^n; \boldsymbol{w})\right), \tag{5.13}$$

where the set of m elements $\left\{ \boldsymbol{z}^{M+1}, \ldots, \boldsymbol{z}^{N+M} \right\}$ is called *test set*.

Disadvantage of the test set method is, that the test set cannot be used for learning because $\boldsymbol{w}$ is selected using the training set and, therefore, $\mathcal{L}(y, g(\boldsymbol{x}; \boldsymbol{w}))$ is not iid for training data points. Intuitively, if the loss is low for some training data points then we will expect that the loss will also be low for the following training data points.

5.2.2 Cross-validation

If we have only few data points available we want to use them all for learning and not for estimating the performance via a test set. But we want to estimate the performance for our final model.

We can divide the available data multiple times into training data and test data and average over the result. Problem here is that the test data is overlapping and we estimate with dependent test data points.

To avoid overlapping test data points we divide the training set into L parts[1] (see Fig. 5.1.). Then we make L runs where for the l-th run part no. l is used for testing and the remaining parts for training (see Fig. 5.2). That procedure is called *L-fold cross-validation*. The *cross-validation risk* $R_{L-\text{cv}}(\boldsymbol{Z})$ is the cumulative loss over all folds used for testing.

A special case of cross-validation is *leave-one-out cross-validation* (LOO CV) where $N = L$ and a fold contains only one element.

The cross-validation risk is a nearly (almost) unbiased estimate for the risk.

Unbiased means that the expected cross-validation error is equal the expected risk, where the expectation is over training sets with l elements.

[1] Note that in this chapter, the index $1 \leq l \leq L$ is used for cross-validation folds and not for layers in a neural network

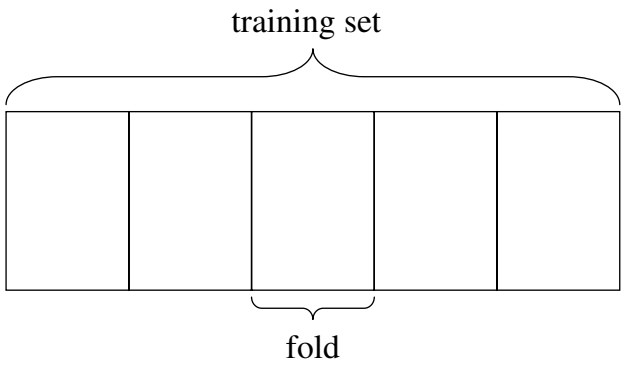


Figure 5.1: Cross-validation: The data set is divided into 5 parts for 5-fold cross-validation — each part is called fold.

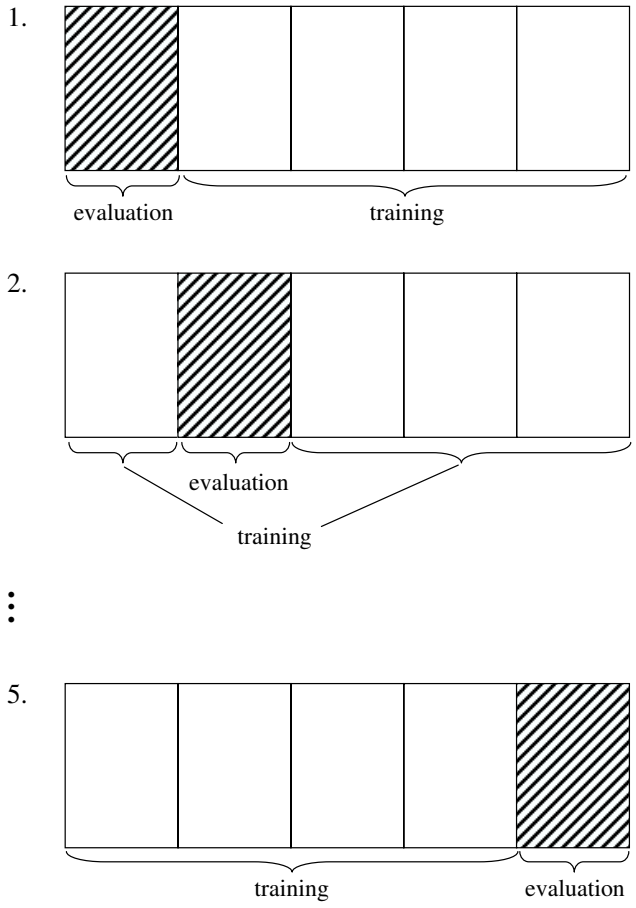


Figure 5.2: Cross-validation: For 5-fold cross-validation there are 5 iterations and in each iteration a different fold is omitted and the remaining folds form the training set. After training the model is tested on the omitted fold. The cumulative error on all folds is the cross-validation error.

We will write $\boldsymbol{Z}_N := \boldsymbol{Z}$ as a variable for training sets with N elements. The l-th fold of an L-fold cross-validation is denoted by $\boldsymbol{Z}^l$ or $\boldsymbol{Z}^l_{N/L}$ to include the number N/L of elements of the fold. The L-fold cross-validation risk is

$$R_{L-\mathrm{cv}}(\boldsymbol{Z}_N) = \frac{1}{L}\sum_{l=1}^{L}\frac{L}{N}\sum_{\boldsymbol{z}\in\boldsymbol{Z}^l_{N/L}} L\left(y, g\left(\boldsymbol{x}; \boldsymbol{w}_l\left(\boldsymbol{Z}_N \setminus \boldsymbol{Z}^l_{N/L}\right)\right)\right), \tag{5.14}$$

where $\boldsymbol{w}_l$ is the model selected when removing the l-th fold and

$$R_{\mathrm{L-cv},l}(\boldsymbol{Z}_l) = \frac{L}{N}\sum_{\boldsymbol{z}\in\boldsymbol{Z}^l_{N/L}} L\left(y, g\left(\boldsymbol{x}; \boldsymbol{w}_l\left(\boldsymbol{Z}_N \setminus \boldsymbol{Z}^l_{N/L}\right)\right)\right) \tag{5.15}$$

is the risk for the l-th fold.

The statement that the "cross-validation estimate for the risk is almost unbiased" (Luntz and Brailovsky) means

$$\mathbb{E}_{\boldsymbol{Z}_{N(1-1/L)}}\left[R_{\mathrm{gen}}\left(g\left(.; \boldsymbol{w}\left(\boldsymbol{Z}_{N(1-1/L)}\right)\right)\right)\right] = \mathbb{E}_{\boldsymbol{Z}_N}\left[R_{L-\mathrm{cv}}\left(\boldsymbol{Z}_N\right)\right]. \tag{5.16}$$

The generalization error on training size N without one fold N/L, namely $N - N/L = N(1-1/L)$ can be estimated by cross-validation on training data of size N by L-fold cross-validation. For large N the training size N or $N(1-1/L)$ should lead similar results, that is the estimate is almost unbiased.

The following two equations will prove eq. (5.16). The left hand side of eq. (5.16) can be rewritten as

$$\begin{aligned}
&\mathbb{E}_{\boldsymbol{Z}_{N(1-1/L)}}\left[R_{\mathrm{gen}}\left(g\left(.; \boldsymbol{w}\left(\boldsymbol{Z}_{N(1-1/L)}\right)\right)\right)\right] \\
&\quad = \mathbb{E}_{\boldsymbol{Z}_{N(1-1/L)}\cup\boldsymbol{z}}\left[L\left(y, g\left(\boldsymbol{x}; \boldsymbol{w}\left(\boldsymbol{Z}_{N(1-1/L)}\right)\right)\right)\right] \\
&\quad = \mathbb{E}_{\boldsymbol{Z}_{N(1-1/L)}}\mathbb{E}_{\boldsymbol{Z}_{N/L}}\left[\frac{L}{N}\sum_{\boldsymbol{z}\in\boldsymbol{Z}_{N/L}}\left(L\left(y, g\left(\boldsymbol{x}; \boldsymbol{w}\left(\boldsymbol{Z}_{N(1-1/L)}\right)\right)\right)\right)\right].
\end{aligned} \tag{5.17}$$

The second equation arises from the fact that the data points are iid, therefore

$$\mathbb{E}_{\boldsymbol{z}}\left[f(\boldsymbol{z})\right] = \frac{1}{k}\sum_{i=1}^{k}\mathbb{E}_{\boldsymbol{z}}\left[f(\boldsymbol{z}^i)\right] = \mathbb{E}_{\boldsymbol{Z}_k}\left[\frac{1}{k}\sum_{i=1}^{k} f(\boldsymbol{z}^i)\right]$$

.

The right hand side of eq.(5.16) can be rewritten as

$$\begin{aligned}
\mathbb{E}_{\boldsymbol{Z}_N}\left[R_{L-\mathrm{cv}}\left(\boldsymbol{Z}_N\right)\right] &= \mathbb{E}_{\boldsymbol{Z}_N}\left[\frac{1}{L}\sum_{l=1}^{L}\frac{L}{N}\sum_{\boldsymbol{z}\in\boldsymbol{Z}^l_{N/L}} L\left(y, g\left(\boldsymbol{x}; \boldsymbol{w}_l\left(\boldsymbol{Z}_N \setminus \boldsymbol{Z}^l_{N/L}\right)\right)\right)\right] \\
&= \frac{1}{L}\sum_{l=1}^{L}\mathbb{E}_{\boldsymbol{Z}_N}\left[\frac{L}{N}\sum_{\boldsymbol{z}\in\boldsymbol{Z}^l_{N/L}} L\left(y, g\left(\boldsymbol{x}; \boldsymbol{w}_l\left(\boldsymbol{Z}_N \setminus \boldsymbol{Z}^l_{N/L}\right)\right)\right)\right] \\
&= \mathbb{E}_{\boldsymbol{Z}_{N(1-1/L)}}\mathbb{E}_{\boldsymbol{Z}_{N/L}}\left[\frac{L}{N}\sum_{\boldsymbol{z}\in\boldsymbol{Z}_{N/L}} L\left(y, g\left(\boldsymbol{x}; \boldsymbol{w}\left(\boldsymbol{Z}_{N(1-1/L)}\right)\right)\right)\right].
\end{aligned} \tag{5.18}$$

The first equality comes from the fact that sum and integral are interchangeable. Therefore it does not matter whether first the data is drawn and then the different folds are treated or the data is drawn again for treating each fold. The second equality comes from the fact that $\mathbb{E}\left[\boldsymbol{Z}_{N/L}^{l}\right] = \mathbb{E}\left[\boldsymbol{Z}_{N/L}\right]$.

Therefore both sides of eq. (5.16) are equal.

The term “almost” addresses the fact that the estimation is made with $N(1 - 1/L)$ training data using the risk and with N training data using L-fold cross-validation.

However the cross-validation estimate has high variance. The high variance stems from the fact that the training data is overlapping. Also test and training data are overlapping. Intuitively speaking, if data points are drawn which make the task very complicated, then these data points appear in many training sets and at least in one test set. These data points strongly increase the estimate of the risk. The opposite is true for data points which make learning more easy. That means single data points may strongly influence the estimate of the risk.

Note: On the sloppy use of the term "complexity"

The term **complexity** is often used in a sloppy way together with the term **model**. A model is a fixed function whose parameters might have been found by gradient descent, but a fixed function does not have a complexity. However, a *model class*, i.e. a set of functions of the same class but with different parameters, has a *complexity*. In the sense of the VC-dimension, the *complexity of a model class* is the cardinality of the largest set of points that the algorithm can shatter. If you encounter formulations, such as "the complexity of the neural network", usually the complexity of a model class defined by a neural network architecture is meant.

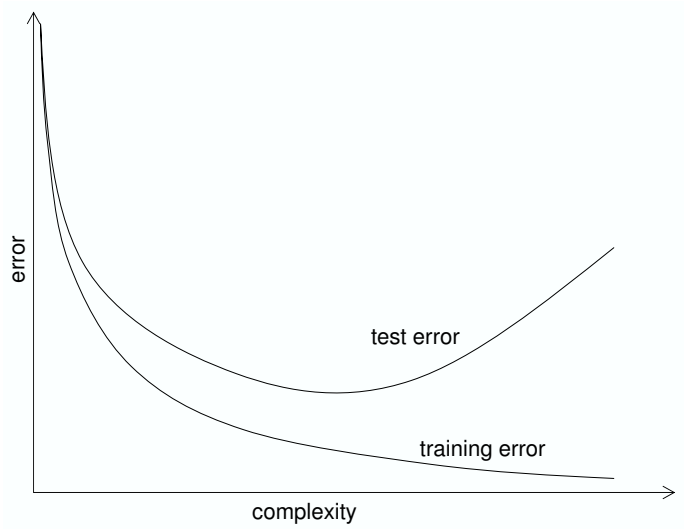


Figure 5.3: Typical example where the test error first decreases and then increases with increasing complexity. The training error decreases with increasing complexity of the *model class*. The test error, the risk, is the sum of training error and a complexity term. At some complexity point the training error decreases slower than the complexity term increases – this is the point of the optimal test error.

Chapter 6

Convolutional neural networks: Basic architectures

GÜNTER KLAMBAUER AND PHILIPP RENZ

When images (or sequences) are input objects, we have some information about the structure of the object, for example, which pixels are next to each other. However, feed-forward neural networks do not use this information. A *convolutional neural network* (CNN) is a neural network that uses the so-called *convolution* operation instead of matrix multiplication. This operation utilizes the local structure of the object, as we will see. CNNs also use another operation called *pooling*, which we will treat later.

ImageNet. Convolutional neural networks – although already known in the 90s – became prominent again after being used to win the so-called ImageNet Large Scale Visual Recognition Competition (ILSVRC) Challenge in 2012 (Krizhevsky et al., 2012a). The data set consisted of 1.2M images belonging to 1,000 different classes, ranging from dog breeds over cars to household items. The challenge was executed yearly to determine the "state of the art" in image recognition. When in 2012, the challenge was won by a CNN-based solution by a landslide victory, the attention was drawn to this type of method again. From then on CNNs started to dominate not only the ImageNet Challenge, but also almost all image recognition and object detection areas. For further details about ImageNet see Section D.3.

6.1 Image data and their characteristics

Images are an extremely high-dimensional data type, for example a relatively small image with 250x250 pixels and 3 color channels represents an input object with a dimension of 200,000. A feed-forward neural network with 1,000 neurons in the first hidden layer, would have 200M adjustable weights (see the MNIST example in Fig. 6.1). Upon reflection, it is actually unreasonable to use networks with feed-forward neural networks to classify images, because such a network architecture does not take into account the spatial structure of the images. For instance, it treats input pixels which are far apart and close together on exactly the same footing. Such concepts of spatial structure must instead be inferred from the training data.

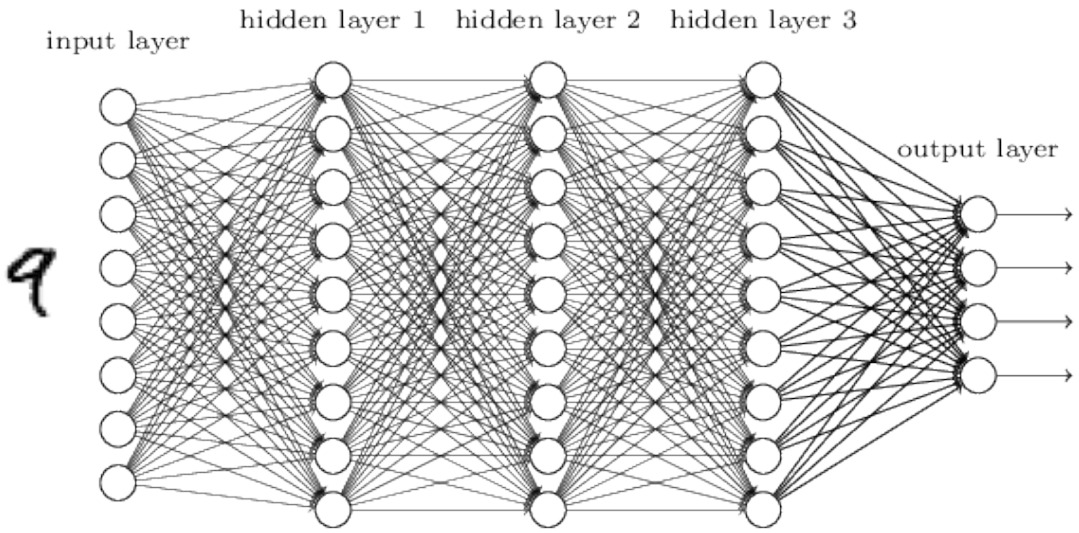


Figure 6.1: Schematic visualization of a for feed-forward or fully-connected neural network for classification of handwritten digits (MNIST, see Section D.1).

MNIST example. Let us go back to the problem of handwritten digit recognition. In figure 6.1 a MLP, that is unaware of the input structure is sketched.

Instead of starting with an architecture that is agnostic to the spatial structure, we aim for one which tries to take advantage of it. This is exactly what convolutional neural networks are designed for, as they use a special architecture which is particularly well-adapted to classify images. In the following, we will investigate how these work in detail. Today, deep convolutional networks are the standard for many image recognition tasks.

The main idea of CNNs is to use local receptive fields which are applied on all positions of the image by sharing parameters across neurons. Subsequently, they are given some invariance to the exact location of the feature by *pooling operations*. Pooling also allows to detect features at different levels. Convolutional neural networks use the following basic ideas:

1. **Local receptive fields via the convolution operation**: see Subsection 6.2 and 6.4.
2. **Shared weights**: see Subsection 6.5.
3. **Non-linearities**: see Subsection 6.6.
4. **Pooling**: see Subsection 6.7.

6.2 The convolution operation

The *discrete convolution* operation uses two real-valued functions h and k defined on $\mathbb{Z}$, $h : \mathbb{Z} \mapsto \mathbb{R}$ and $k : \mathbb{Z} \mapsto \mathbb{R}$, and is given by

$$(h * k)(a) = \sum_{i=-\infty}^{\infty} h(a-i)\, k(i), \tag{6.1}$$

where we use $*$ to denote the convolution operation and h is the function to be convolved with the kernel k, where the different roles of these functions will become evident soon.

For images, convolutions over two axes at the same time are used:

$$(h * k)(a,b) = \sum_{i=-\infty}^{\infty} \sum_{j=-\infty}^{\infty} h(a-i, b-j)\, k(i,j), \tag{6.2}$$

where the h and k are two-place functions, but otherwise defined as above.

However, the operation that is used in *convolutional neural networks*, is the following:

$$(h * k)(a,b) = \sum_{i=-\infty}^{\infty} \sum_{j=-\infty}^{\infty} h(a+i, b+j)\, k(i,j). \tag{6.3}$$

The above formula is actually what is known as *cross-correlation*, but still termed "convolution" in machine learning.

We identify the function $h(a,b)$ with the image $\boldsymbol{x} \in \mathbb{R}^{H \times W}$, that is our input object with height H and width W. $k(i,j)$ is viewed as a weight matrix $\boldsymbol{W} \in \mathbb{R}^{R \times R}$ with entries $w_{i,j}$, where R is the size of the local receptive field[1]. The weight matrix, also called kernel, is shifted across the image to obtain a *feature map* $s_{a,b}$:

$$s_{a,b} = \sum_{i=0}^{R-1} \sum_{j=0}^{R-1} w_{i,j}\, x_{a+i,b+j} = (\boldsymbol{W} * \boldsymbol{x})_{a,b}. \tag{6.4}$$

This operation is depicted in figure 6.2. As can be seen in the mentioned image the size of the feature map depends on both the size of the input (H, W) and the kernel size R. The size of the feature map (A, B) turns out to be:

$$(A, B) = (H - R + 1, W - R + 1) \tag{6.5}$$

Equation 6.4 is at the core of the *forward-pass* of a convolutional layer. $s_{a,b}$ are called pre-activations and are usually followed by a non-linear activation function.

Note: Indexing in this chapter

Note that – unlike in other chapters – indices start with zeros rather then ones, which is the typical notation for image data.

Note that when we considered fully-connected networks, the input $\boldsymbol{x}$ was a one-dimensional vector $\boldsymbol{x} = (x_1, \ldots, x_D)$. Now $\boldsymbol{x}$ is an image, which is a two-dimensional object, that has a height and a width and each entry represents the intensity of a pixel.

[1] We only consider square receptive fields here for simplicity, while in general they could also be rectangular

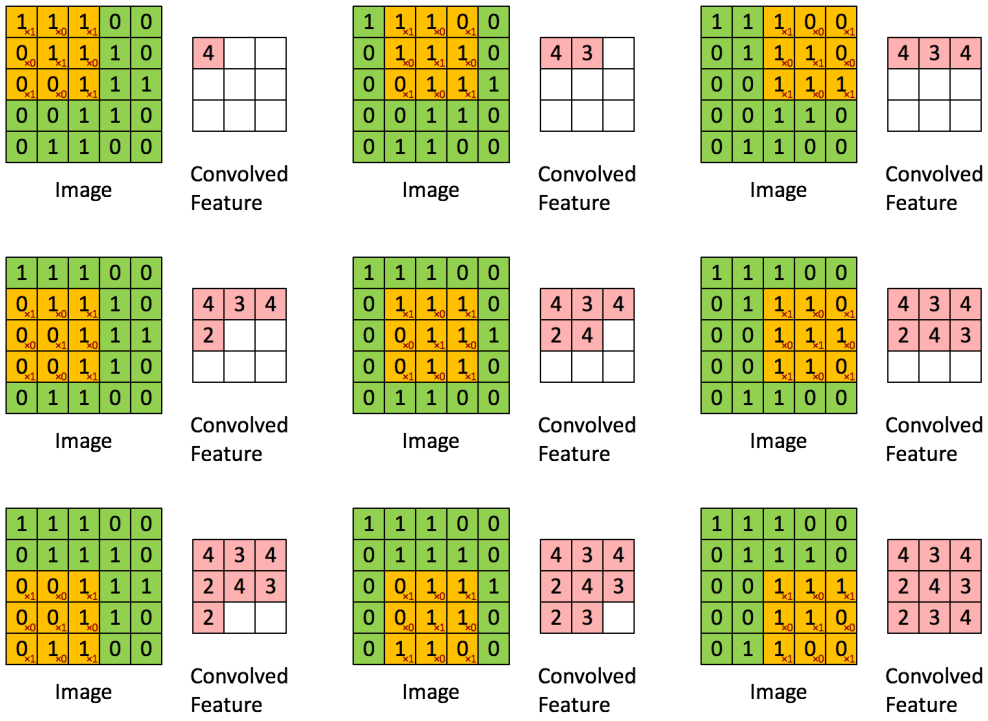


Figure 6.2: An example for a convolution and its shifting across image locations. The kernel (yellow box), $\boldsymbol{w} = ((1,0,1),(0,1,0),(1,0,1))$, is shifted across the image (green box). The result of this operation is shown beside the image (red box entries). There are nine neurons in the upper layer that have a local receptive field of 3x3. All these nine neurons share the same weights.

A note on images with depth. While in the above formulations, we considered the case of an object $x_{a,b}$ with two dimensions, such as an image with height A and width B. Oftentimes objects also have a distinct third dimension, such as RGB channels or feature maps, along which no convolution should be performed. Thus, the following variant of the convolution operation is employed for such data:

$$s_{a,b} = \sum_{c=0}^{C-1} \sum_{i=0}^{R-1} \sum_{j=0}^{R-1} w_{i,j,c}\, x_{a+i,b+j,c} = (\boldsymbol{W} * \boldsymbol{x})_{a,b}, \tag{6.6}$$

where the index c runs across the C different channels (or feature maps) of the image $x_{a,b,c}$. The weight matrix $w_{i,j,c}$ can be considered as a stack of kernels.

6.3 Backpropagation through the convolution operation

In order to train a network containing a convolutional layer we need to be able to obtain the derivative of the loss function with respect to its weights and its inputs. We need the former to update the weights of the convolution operation itself and the latter to propagate the gradients to preceding layers.

We assume that we have error function $L(y, \hat{y})$ with predictions $\hat{y}$ from a neural network that includes the convolution operation from Eq. (6.4) in its forward pass:

$$s_{a,b} = \sum_{i=0}^{R-1} \sum_{j=0}^{R-1} w_{i,j}\, x_{a+i,b+j}, \tag{6.7}$$

where $x_{a,b}$ can be the pixels of the input image for the network, but also the input to a higher level layer.

6.3.1 Weight update

The updates for weights are given by:

$$\frac{\partial L}{\partial w_{i,j}} = \sum_{a=0}^{A-1} \sum_{b=0}^{B-1} \frac{\partial L}{\partial s_{a,b}} \frac{\partial s_{a,b}}{\partial w_{i,j}} \tag{6.8}$$

$$= \sum_{a=0}^{A-1} \sum_{b=0}^{B-1} \underbrace{\frac{\partial L}{\partial s_{a,b}}}_{:=\delta_{a,b}} x_{a+i,b+j} \tag{6.9}$$

$$= (\boldsymbol{\delta} * \boldsymbol{x})_{i,j} \tag{6.10}$$

where A and B can be inferred from (6.5). Note that in order to update $w_{i,j}$, the delta errors are summed over different positions a, b of the feature map and are stabilized by this summation.

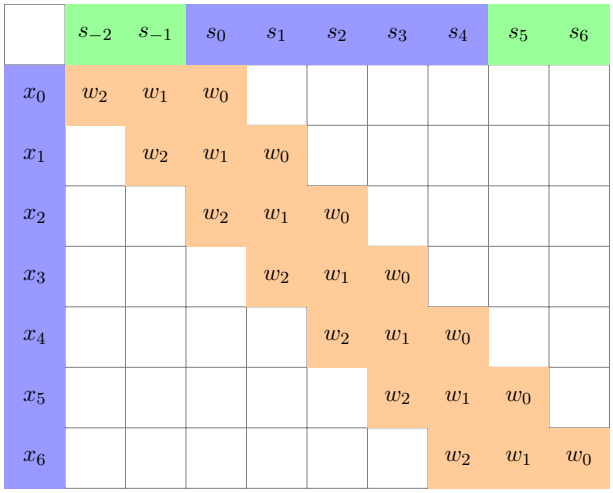


Figure 6.3: Visualization of a transposed 1-D convolution with kernel size three. When looking at the filters vertically we can see which elements of the output $\boldsymbol{s}$ depend on which input elements via which weight. When doing a transposed convolution or deconvolution we convolve the inverted kernel with with the padded (green) output. For understanding the backward pass one can replace s_i by $\frac{\partial L}{\partial s_i} = \delta_i$

.

6.3.2 Deltas for lower layer

We can write the derivative of a loss function w.r.t. the inputs of a convolution operation which shows us how to propagate gradients through it.

$$
\begin{aligned}
\frac{\partial L}{\partial x_{h,w}} &= \sum_{a=0}^{A-1}\sum_{b=0}^{B-1} \frac{\partial L}{\partial s_{a,b}} \frac{\partial s_{a,b}}{\partial x_{h,w}} \\
&= \sum_{a=0}^{A-1}\sum_{b=0}^{B-1} \underbrace{\frac{\partial L}{\partial s_{a,b}}}_{:=\delta_{a,b}} \sum_{u=0}^{R-1}\sum_{v=0}^{R-1} w_{u,v} \frac{\partial x_{a+u,b+v}}{\partial x_{h,w}} \\
&= \sum_{a=0}^{A-1}\sum_{b=0}^{B-1} \delta_{a,b} \sum_{u=0}^{R-1}\sum_{v=0}^{R-1} w_{u,v} \mathbf{1}_{(a+u=h,b+v=w)} \\
&= \sum_{a=h-R+1}^{h}\sum_{b=w-R+1}^{w} \delta_{a,b}\, w_{h-a,w-b} \\
&= \sum_{a'=0}^{R-1}\sum_{b'=0}^{R-1} w_{R-1-a',R-1-b'}\, \delta_{h+a'-R+1,w+b'-R+1}, \\
&= \sum_{a'=0}^{R-1}\sum_{b'=0}^{R-1} \hat{w}_{a',b'}\, \hat{\delta}_{h+a',w+b'} \\
&= (\hat{\boldsymbol{w}} * \hat{\boldsymbol{\delta}})_{h,w},
\end{aligned} \tag{6.11}
$$

where $\hat{\boldsymbol{\delta}}$ the matrix of delta errors padded with $R-1$ zeroes, $\hat{\boldsymbol{w}}$ is the matrix $\boldsymbol{w}$ with the inverse order of rows and columns and $\mathbf{1}$ is the indicator function, which equals one if the conditions in the subscript hold and zero else.

The first two lines of (6.11) follow from the chain rule and the definition of the convolution. Then the sums over u and v collapse to only one term as the different elements of $\boldsymbol{x}$ do not depend on each other. In line four we change the sum limits such that a and b run through all indices of the weight matrix. This makes it possible that the indices $\delta_{a,b}$ are out of range, which is not a problem if we set $\delta_{a,b} = 0$ if $(a,b) \notin \{0,\dots,A-1\} \times \{0,\dots,B-1\}$. When looking at the indices of $\delta_{a,b}$ now, we can see that a takes the minimum value of $-R+1$ and a maximum value of $A-1+R-1$ as the shape of $\boldsymbol{x}$ is $(A+R-1, B+R-1)$. This will lead us to pad $\boldsymbol{\delta}$ with $R-1$ pixels on each side later on. In line five we replace the sum indices using $a' = a-h+R-1$ and $b' = b-w+R-1$. In the last but one line we pad $\boldsymbol{\delta}$ with $R-1$ zeros on each side to obtain $\boldsymbol{\delta}'$. Now we can see that the last but one line is a convolution again. The process of padding an image and convolving it with an inverted kernel is a common operation known as a transpose convolution or deconvolution. This is analogous to backpropagation in fully connected networks where we need the transposed weight matrix to pass back the gradients to preceding layers.

We further investigate the term $\frac{\partial s_{a,b}}{\partial x_{h,w}}$ below noting that because of the local connectivity, the expression will be zero for many combinations of a, b and h, w.

$$\frac{\partial s_{a,b}}{\partial x_{h,w}} = \sum_{k=0}^{R-1}\sum_{l=0}^{R-1} w_{k,l}\,\frac{\partial x_{a+k,b+l}}{\partial x_{h,w}} \tag{6.12}$$

$$= \sum_{k=0}^{R-1}\sum_{l=0}^{R-1} w_{h-a,w-b}\,\frac{\partial x_{a+k,b+l}}{\partial x_{h,w}} \tag{6.13}$$

$$= \begin{cases} w_{h-a,w-b}, & 0 \le h-a \le R-1 \wedge 0 \le w-b \le R-1 \\ 0, & \text{else} \end{cases} \tag{6.14}$$

The second line follows from the fact

$$\frac{\partial x_{a+k,b+l}}{\partial x_{h,w}} = \begin{cases} 1, & \text{if } a+k=h \wedge b+l=w \\ 0, & \text{else} \end{cases} \tag{6.15}$$

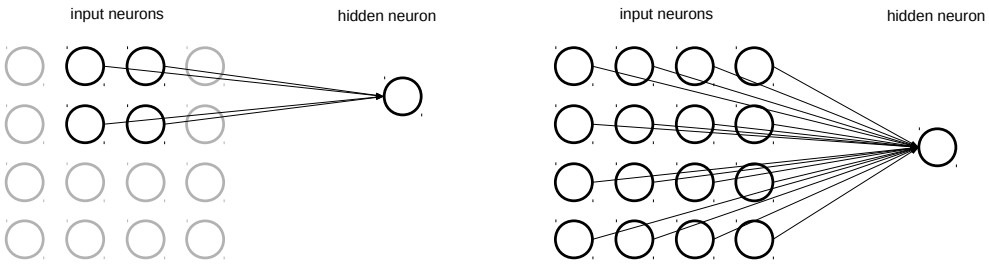
The third line can be better understood if one is aware that the double sum collapses to just one term where the condition of (6.15) is fulfilled. One pixel of the feature map is only influenced by one pixel of the input via one weight. Further we look at fixed h, w, a, b. We know that an output with index a cannot depend on an input with index h if there is no k for which $a+k=h$.
Now we show why the condition in (6.14) holds. To show:

$$\text{For given } a,h \nexists k \in \{0,\dots,R-1\} \text{ s.t. } a+k=h \implies h-a<0 \text{ or } h-a>R-1 \tag{6.16}$$

We consider the two cases where $a+k$ is smaller/bigger than h for all k.
Case 1:

$$a+k<h \quad \forall k \implies \max(a+k) = a+R-1 < h \tag{6.17}$$

$$\implies R-1 < h-a \tag{6.18}$$



a) b)

Figure 6.4: **a)** Local receptive fields of convolution. A hidden neuron in the next layer (right hand side) only has connection to a few neurons in the former layer (left hand side). The neurons marked by black circles are the local receptive field of the neuron on the right hand side. **b)** In contrast, in feed-forward neural network each hidden neuron has connections to all neurons of the previous layer.

Case 2:

$$a + k > h \quad \forall k \implies \min(a + k) = a < h \tag{6.19}$$
$$\implies h - a < 0 \tag{6.20}$$

This proves the condition in (6.14). If there is a k such that $a + k = h$, it is trivial to see that the index on w is not out of range. The proof for the second dimension is the same as for the first. These results simplify our result from before as we can just write

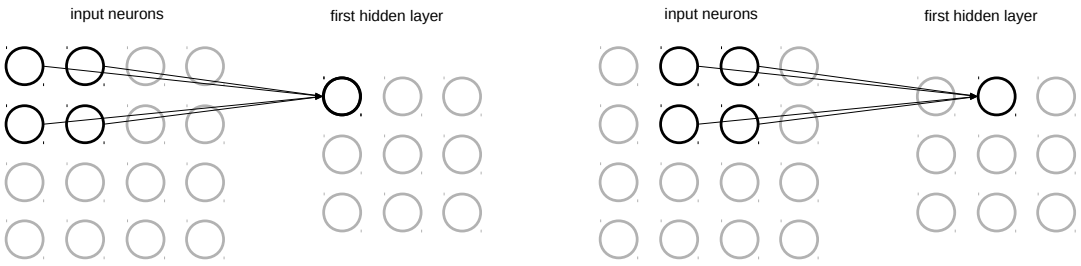
$$\frac{\partial s_{a,b}}{\partial x_{h,w}} = w_{a-h,b-w}, \tag{6.21}$$

when we consider that the term on the right hand side should be interpreted as zero if at least one of the indices is out of range.

6.4 Local receptive fields

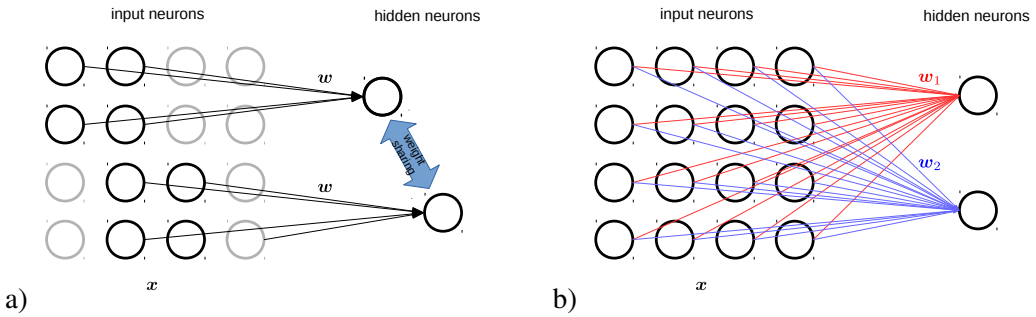
Comparing the current convolution operation with operations in a feed-forward network, we find that input pixels are not connected to every hidden neuron. Instead, there are only connections in small, localized regions of the input image.

This region in the input image is called the *local receptive field* for the hidden neuron. As in FNNs, each connection has an adaptive weight, and potentially also a bias unit. A particular hidden neuron learns to analyze its particular local receptive field. For each local receptive field, there is a different hidden neuron in the first hidden layer. However, by sharing the same weights across different locations, the local receptive field can potentially learn across the entire input image. This is illustrated in (Fig. 6.5)



a) b)

Figure 6.5: Visualization of weight sharing and shifting in a convolution layer. a) Each neuron in the first hidden layer (right hand side) is connected to particular input neurons in its local receptive fields (left hand side). b) Both the receptive field and the neuron in the first hidden layer are shifted by one pixel to the right. The weights associated by the connecting lines are the same in figure a) and b), which is called weight sharing.



a) b)

Figure 6.6: **a)** Schematic representation of weight sharing in a convolutional layer. **b)** In contrast, in fully-connected networks neurons do not share weights.

Instead of saying that one neuron learns parameters for its local receptive field and shares these parameters with all other neurons in this layer, we could also say that learned filter is shifted by one pixel [2] to the right (i.e., by one neuron), to connect to a second hidden neuron [3] (see Fig. 6.5)

6.5 Shared weights and biases

Sharing weights means that all the neurons in the first hidden layer detect exactly the same feature just at different locations in the input image (see Fig. 6.5). This means that by sliding the kernel across locations, the same feature can be detected at different locations in the image.

[2] Sometimes a different *stride* length (e.g, 2) is used.

[3] Note that if we have a 28×28 input image, and 5×5 local receptive fields, then there will be 24×24 neurons in the hidden layer.

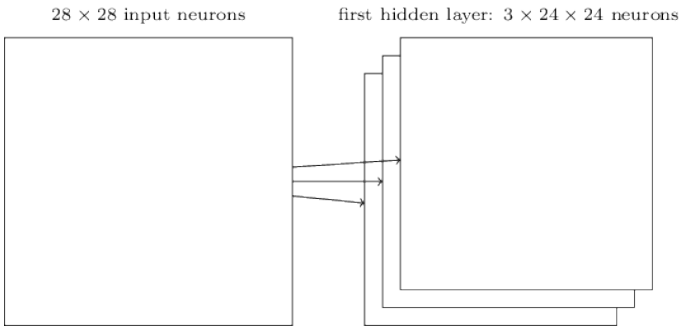


Figure 6.7: Schematic representation of three convolutions applied to an input image. This results in a hidden layer with an additional dimension containing the three feature maps.

The network structure we have described so far can detect just a single kind of localized feature. However, for image classification and object recognition tasks many features can play a role. Thus, a complete convolutional layer consists of multiple kernels that each give rise to several *feature maps* (see Fig. 6.7).

6.6 Non-linearities for CNNs

In analogy to feed-forward neural networks, the convolution operation is typically followed by a non-linear transformation, such as ReLU (Nair and Hinton, 2009), that is applied element-wise. We refer to the chapter A that is dedicated to activation functions.

6.7 Pooling

In addition to the convolutional layers just described, convolutional neural networks also contain pooling layers. Pooling layers are usually used immediately after convolutional layers. What the *pooling layers* do is simplify the information in the output from the convolutional layer.

In detail, a pooling layer takes each feature map output from the convolutional layer and prepares a condensed feature map. For instance, each unit in the pooling layer may summarize a region of, for example, 2x2 neurons in the previous layer. As a concrete example, one common procedure for pooling is known as *max-pooling*. In max-pooling, a pooling unit simply outputs the maximum activation in the 2x2 input region, as illustrated in Figure 6.8.

As mentioned above, the convolutional layer usually involves more than a single feature map. We apply max-pooling to each feature map separately. So if there were three feature maps, the combined convolutional and max-pooling layers would look like figure 6.9.

We can think of max-pooling as a way for the network to ask whether a given feature is found anywhere in a region of the image. It then throws away the exact positional information. The intuition is that once a feature has been found, its exact location is not as important as its rough location relative to other features. A big benefit is that there are many fewer pooled features, and so this helps reduce the number of parameters needed in later layers.

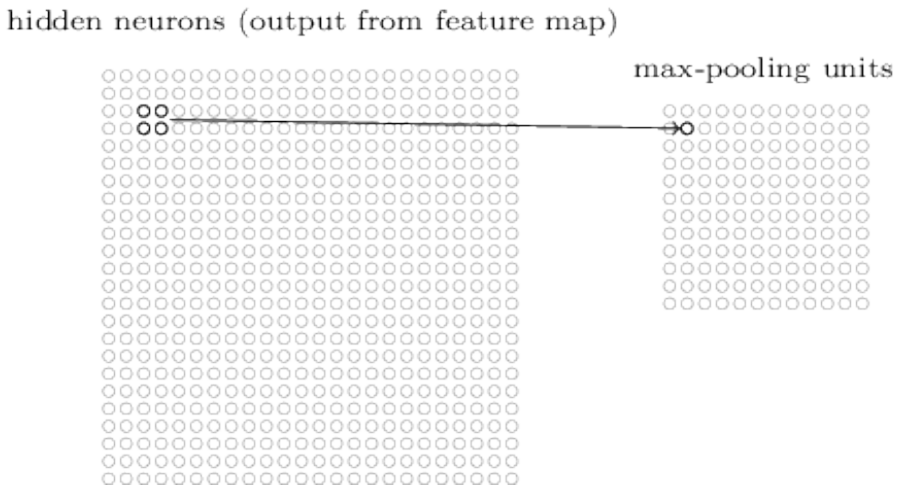


Figure 6.8: Pooling Layer. Source.

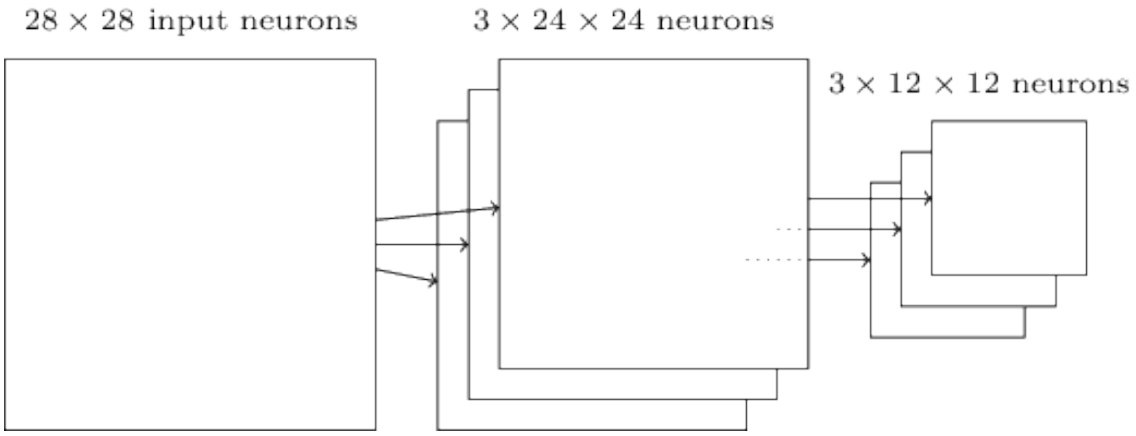


Figure 6.9: Convolution and pooling layer. Source.

6.8 Recap, definitions, input and output sizes

In the section above, we have defined different terms, such as *kernel*, *filter*, *feature map*, which we now collect and for which we provide intuitive definitions.

- **Kernel**: The kernels can be considered as the weights of the neurons of a convolutional layer. In the case of 2D convolutions, the kernel is a fixed rectangle (typically a square) with dimensions $R \times R$.
- **Filter**: Typically used synonymously with kernel.
- **Kernel size**: If the dimension of the kernel is $R \times R$, the kernel size is said to be R. Typical values are $R = 3$ giving 3x3 filters or $R = 5$ giving 5x5 filters.
- **Shared weights**: For 2D convolutions, each neuron in the output has R^2 C weights, where C is the number of feature maps (or channels) in the input. These weights are shared by the

neurons in the output.

- **Stride**: A filter need not be shifted pixel by pixel but can also move across the input in bigger steps. The number of pixels that a filter is moved at each step is referred to as stride or S.
- **Padding**: Depending on the image size, the kernel size and the stride, problems can occur at the borders of the image as the filter might not use all elements of the input (figure 6.10).

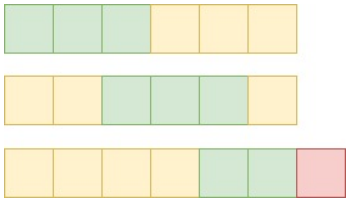


Figure 6.10: A 1-D convolution with a with input size of 6, kernel size of 3 and a stride of 2. The filter is moved in steps of size 2. The last element of the input will not be processed as the filter would protrude over the input.

In this case, zero padding adds additional columns or rows to the input image to avoid these problems. In other words, padding defines how the border of a sample is handled. Padding can be used to keep the spatial output dimensions equal to those of the input, unpadded convolutions will crop away some of the borders if the kernel dimension is larger than 1.

- **Feature map**: A feature map is the output of a filter applied to the input to a particular layer. A given kernel is moved across the entire previous layer, typically shifted by one pixel at a time. Each position results in an activation of the neuron and the output is collected in the feature map (see Eq.(6.4) and (6.6)). The dimensions of a feature map are

$$A^{[l+1]} \times B^{[l+1]} \times C^{[l+1]}, \tag{6.22}$$

with $A^{[l+1]} = (A^{[l]} - R + 2P)/S + 1$ and $B^{[l+1]} = (B^{[l]} - R + 2P)/S + 1$, where

  - $A^{[l]}$ is the height of the feature maps or the input image in the previous layer
  - $B^{[l]}$ is the width of the feature maps or the input image in the previous layer
  - P is the amount of (symmetric) zero padding,
  - R is the kernel size,
  - S is the stride, and
  - $C^{[l+1]}$ is the number of output feature maps of the convolutional layer.

- **Convolutional layer**: A layer in a convolutional network that applies one or more convolution operations to a structured input. A convolutional layer typically uses several kernels to produce feature maps as outputs. The parameters of a convolutional layer l usually are the kernel size R, the number of output feature maps $C^{[l+1]}$, the stride S (typically $S = 1$), and the amount of padding P. Note that convolutional layers produce structured outputs again. Occasionally, the non-linearity is considered as a part of the convolutional layer.

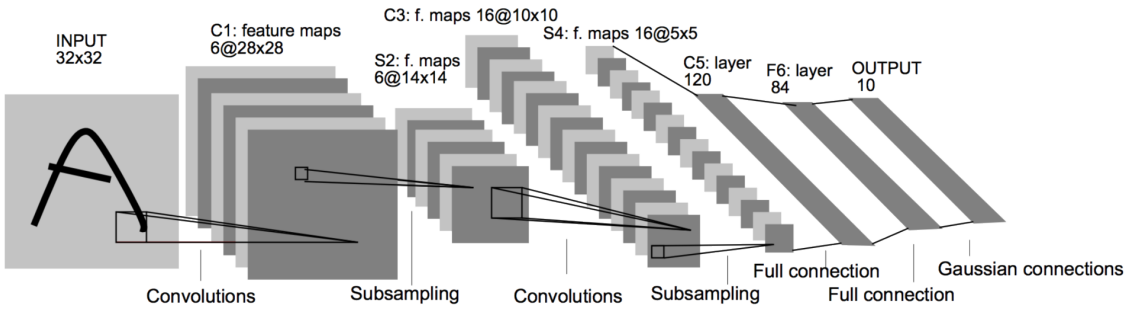


Figure 6.11: LeNet architecture.

6.9 Example: LeNet

In this section we investigate a simple convolutional neural network architecture in more detail. This basic structure of this CNN has been developed already in 1998 (LeCun et al., 1998a). This is the *architecture* of LeNet:

Investigating each of the component layers of LeNet: [4]

- **Input**: Gray scale image of size 32x32.
- **C1**: Convolutional layer of 6 feature maps, kernel size (5, 5) and stride 1. Output size therefore is 28x28x6. Number of trainable parameters is $(5 * 5 + 1) * 6 = 156$.
- **S2**: Pooling/subsampling layer with kernel size (2, 2) and stride 2. Output size is 14x14x6. Number of trainable parameters = 0.
- **C3**: Convolutional layer of 16 feature maps. Each feature map is connected to all the 6 feature maps from the previous layer. Kernel size and stride are same as before. Output size is 10x10x16. Number of trainable parameters is $(6 * 5 * 5 + 1) * 16 = 2416$, where the "+1" arises from a bias unit.
- **S4**: Pooling layer with same *hyperparameters* as above. The output size = 5x5x16.
- **C5**: Convolutional layer of 120 feature maps and kernel size (5, 5). This amounts to *full connection* with outputs of previous layer. Number of parameters are $(16 * 5 * 5 + 1) * 120 = 48120$.
- **F6**: *Fully connected layer*[5] of 84 units. i.e, All units in this layer are connected to previous layer's outputs. Number of parameters is $(120 + 1) * 84 = 10164$
- **Output**: Fully connected layer of 10 units with softmax activation[6].

The dataset used was MNIST (see Section D.1). It has 50,000 training images and 10,000 testing examples.

[4] Actually a slightly modified version of LeNet is described.
[5] This is same as layers in MLP we've seen before.
[6] Ignore 'Gaussian connections'. It is for a older loss function no longer in use.

A big advantage of sharing weights and biases is that it greatly reduces the number of parameters involved in a convolutional network. Coming back to the MNIST example, for each kernel we need 25=5x5 shared weights, plus a single shared bias. So each kernel requires 26 parameters. If we have 20 kernels that's a total of 20x26=520 parameters defining the convolutional layer. By comparison, suppose we had a fully-connected first layer, with 784=28x28 input neurons, and a relatively modest 30 hidden neurons. That is a total of 784x30 weights, plus an extra 30 biases, for a total of 23,550 parameters. Thus, convolutional neural networks typically have fewer adaptive weights than feed-forward neural network.

6.10 Convolutional blocks

A basic element of a convolutional neural network is a *convolutional block* also sometimes simply referred to as *convolutional layer*. Note that the term *convolutional layer* can have two meanings in literature: 1) a layer in a neural network that applies the convolution operation defined in Eq.(6.4) or 2) the combination of several convolutional operations, non-linearity and pooling. In order to avoid an unclear terminology, we use the term *convolutional block*.

A convolutional block typically starts with a convolution operation that generates feature maps (see Fig. 6.12, bottom right), followed by non-linear activation functions (see Fig. 6.12, mid right). This is sometimes called the *detector stage* of a CNN. Finally, max pooling, or other pooling functions, replaces the output of the network at a particular location with a summary statistics of nearby outputs. The pooling operations (see Fig. 6.12, top right) makes the the representation approximately invariant to small translations in the feature map (Goodfellow et al., 2016).

6.11 Visualizing and understanding CNNs

Convolutional neural networks have been shown to learn increasingly abstract features or hierarchical features (Lee et al., 2011). The hierarchical organization of neurons and their receptive fields even have some resemblances to biological neurons in the visual cortex (Kuzovkin et al., 2018). Through multiple layers, CNNs have the ability to extract low-level, mid-level, and high-level features, where features in higher levels are typically a combination of lower-level and mid-level features. Thus, CNNs are thought to have an automatic feature extraction ability and through that learn an informative representation from the low level representation as raw pixels.

The features that are typically learned in the lowest layer of CNNs are shown in Figure 6.13. Features in higher layers are more difficult to visualize since they do not act on pixels anymore. Zeiler and Fergus (2014) develop a method to visualize features in higher layers. These visualizations are shown in Fig.6.14, where it becomes evident that increasingly abstract features are learned in higher level layers.

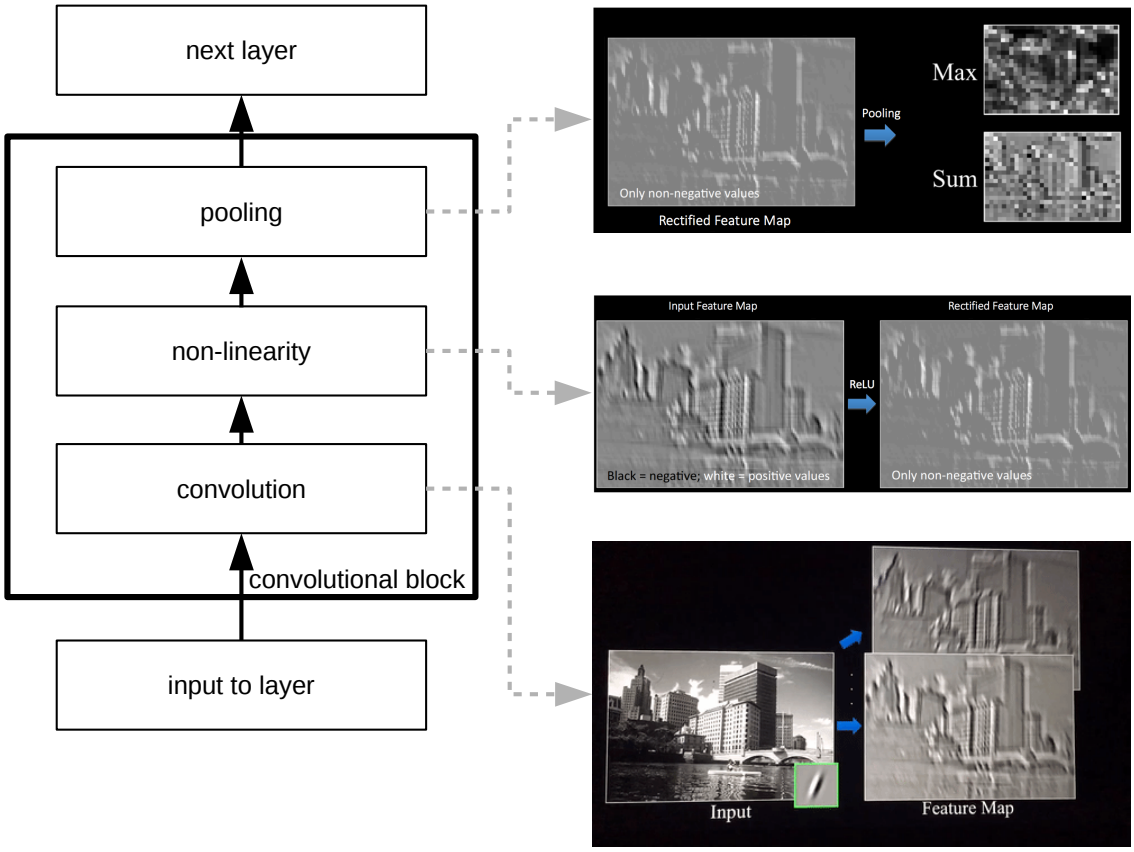


Figure 6.12: The main components of a convolutional block. A convolutional block is the basic element of a convolutional network and consists of one or more convolution operations, a non-linearity, such as ReLU, and subsequent pooling, such as max-pooling (Fergus, 2015).

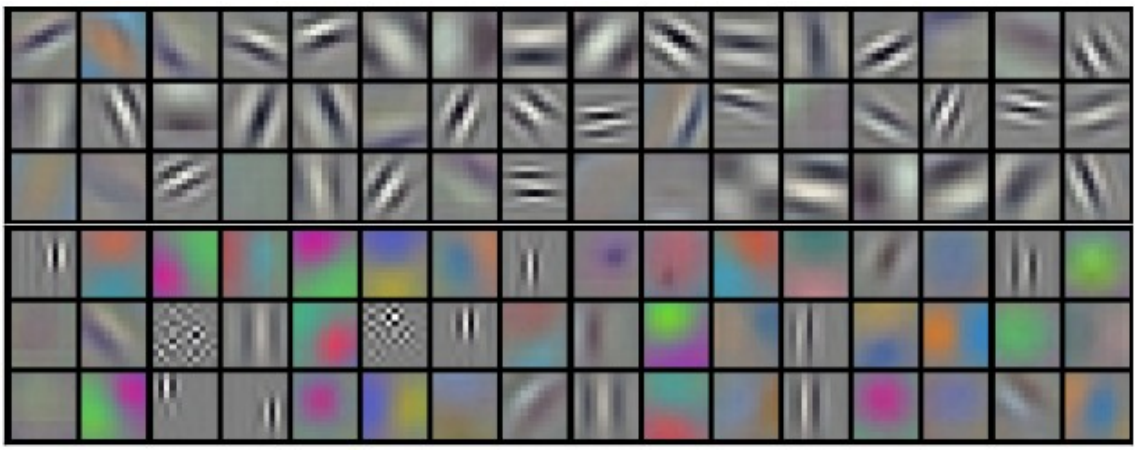


Figure 6.13: Learned receptive fields of a large CNN
ImageNet Classification (see Section D.3) with Deep Convolutional Neural Networks, Kriezhevsky & Sutskever & Hinton, NIPS 2012

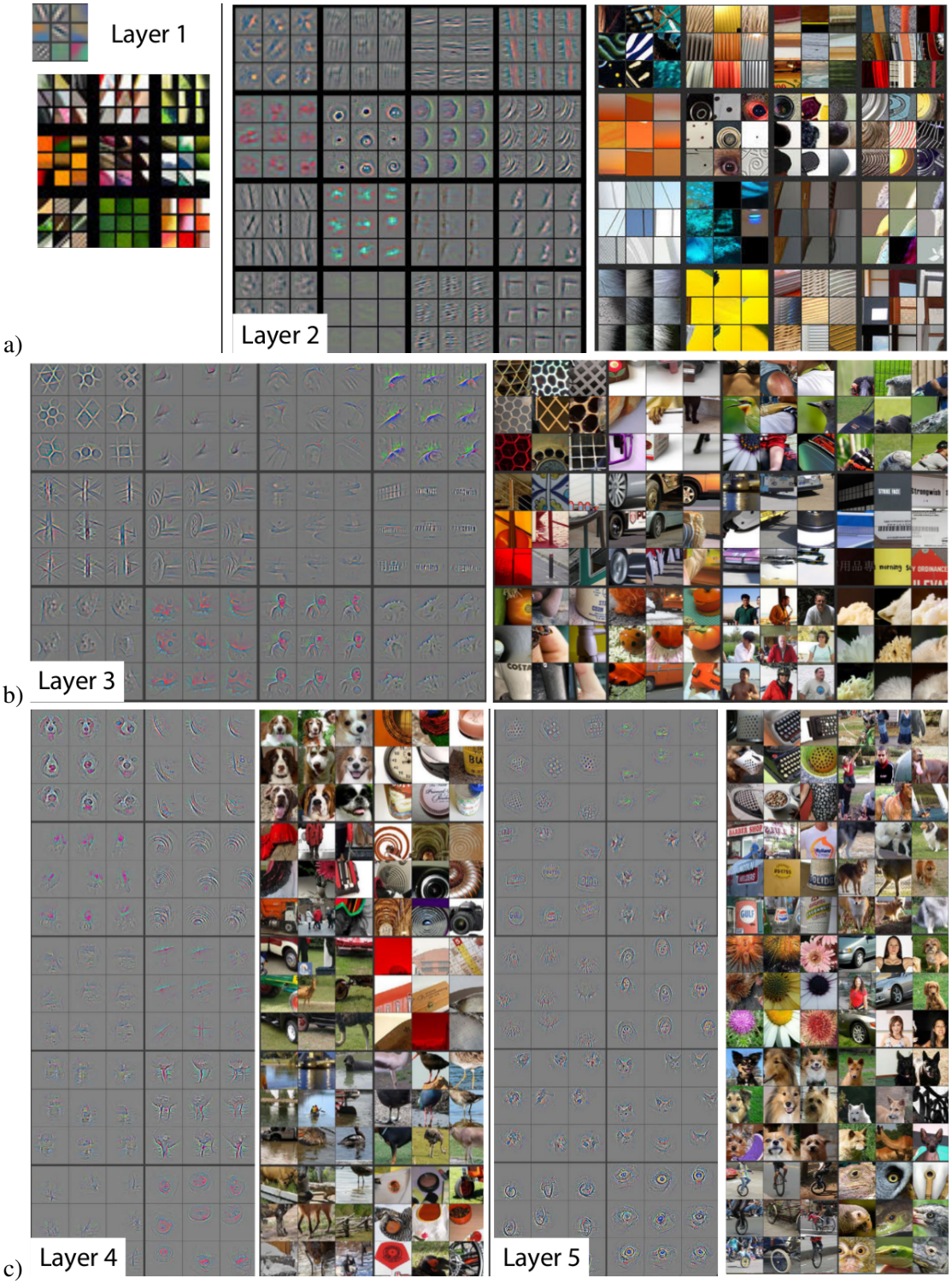


Figure 6.14: Visualizations of features learned in increasingly higher layers of a convolutional neural network.

Chapter 7

Learning methods: Basic algorithms

PHILIPP RENZ, PIETER-JAN HOEDT AND GÜNTER KLAMBAUER

In this chapter, we consider the problem of *learning*: a term which is used in machine learning, when parameters $\boldsymbol{w}$ of a model are adjusted in order to optimize some objective function or *cost function* R that depends on those parameters $R(\boldsymbol{w})$[1]. In previous chapters, we have used the *empirical error* as objective function. The objective function often consists of a data-driven error plus a regularization error. Instead of randomly searching for parameters with a low (or high) cost, we can use gradient information $\nabla_{\boldsymbol{w}} R(\boldsymbol{w})$ and start with randomly drawn parameters and move a small step following the gradient or negative gradient in order to increase or decrease the objective function. This simple approach has proven very effective for deep neural networks and we will get to know different variants of this approach, called *gradient descent*.

The process of iteratively adapting the parameters and gradually improving the objective function can very naturally be understood as "learning": whereas a neural network with the initial random parameters produces random outputs, the output values should gradually improve with increasing number of update steps. There is also a difference between "learning" and pure optimization: In machine learning, we are often interested in a *performance measure*, such as accuracy, that is defined with respect to the test set and could be intractable to optimize, and we use use a "surrogate" for that: an error function that is tractable and we hope that by optimizing this error function, we will improve the performance measure.

Currently, the most popular algorithm to train neural networks is gradient descent. Almost all Deep Learning libraries comprise implementations of algorithms to perform gradient descent. Gradient descent is a way to minimize an objective function $\nabla_{\boldsymbol{w}} R(\boldsymbol{w})$ parameterized by a model's parameters $\boldsymbol{w}$ by updating the parameters in the negative direction of the gradient of $R(\boldsymbol{w})$. A learning rate η scales the gradient and thereby determines the size of the steps in this direction. Thus, we follow the direction of the slope of the surface created by the objective function downhill.

Stochastic gradient-based optimization is of core practical importance in many fields of science and engineering (Kingma and Ba, 2014). Many problems in these fields can be cast as the optimization of some scalar parameterized objective function requiring maximization or minimization with respect to its parameters. If the function is differentiable w.r.t. its parameters, gradient descent is a relatively efficient optimization method, since the computation of first-order partial derivatives w.r.t. all the parameters is of the same computational complexity as just evaluating

[1] Note that in current machine learning literature, the cost function and parameters are sometimes also denoted by $J(\boldsymbol{\theta})$.

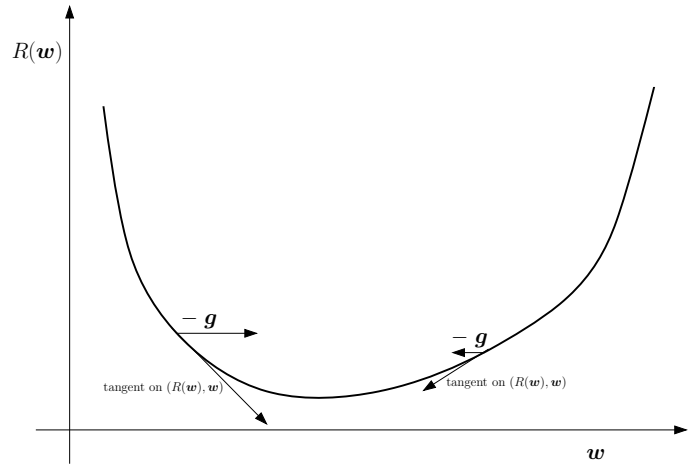


Figure 7.1: The negative gradient $-\boldsymbol{g}$ gives the direction of the steepest decent depicted by the tangent on $(R(\boldsymbol{w}), \boldsymbol{w})$, the error surface.

the function. Often, objective functions are stochastic. For example, many objective functions are composed of a sum of sub-functions evaluated at different sub-samples of data; in this case optimization can be made more efficient by taking gradient steps w.r.t. individual sub-functions, i.e. stochastic gradient descent (SGD) or ascent. SGD proved itself as an efficient and effective optimization method that was central in many machine learning success stories, such as recent advances in deep learning (Krizhevsky et al., 2012b). Objectives may also have other sources of noise than data subsampling, such as dropout (Srivastava et al., 2014) regularization.

7.1 Gradient Descent

Assume that the objective function $R(g(.; \boldsymbol{w}))$ is a differentiable function with respect to the parameter vector $\boldsymbol{w}$. The function g is the model. For simplicity, we will write $R(\boldsymbol{w}) = R(g(.; \boldsymbol{w}))$.

The gradient is

$$\frac{\partial R(\boldsymbol{w})}{\partial \boldsymbol{w}} = (\nabla_{\boldsymbol{w}} R(\boldsymbol{w}))^T = \left(\frac{\partial R(\boldsymbol{w})}{\partial w_1}, \ldots, \frac{\partial R(\boldsymbol{w})}{\partial w_W} \right), \tag{7.1}$$

for a W-dimensional parameter vector $\boldsymbol{w}$. Note that the partial derivative results in a row vector, whereas the Nabla operator results in a column vector.

For the gradient of $R(\boldsymbol{w})$ we use the column vector

$$\boldsymbol{g} = \nabla_{\boldsymbol{w}} R(\boldsymbol{w}). \tag{7.2}$$

The negative gradient is the direction of the steepest descent of the objective. The negative gradient is depicted in Fig. 7.1 for a one-dimensional error surface and in Fig. 7.2 for a two-dimensional error surface.

Because the gradient is valid only locally (for nonlinear objectives) we only go a small step in the direction of the negative gradient if we want to minimize the objective function. The step-size is controlled by $\eta > 0$, the *learning rate*.

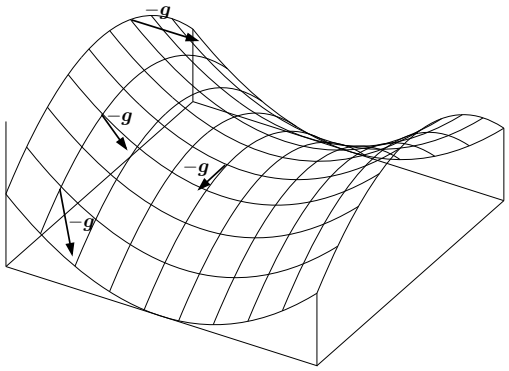


Figure 7.2: The negative gradient $-\boldsymbol{g}$ attached at different positions on a two-dimensional error surface $(R(\boldsymbol{w}), \boldsymbol{w})$. Again the negative gradient gives the direction of the steepest descent.

The gradient descent update is

$$\begin{aligned} \Delta \boldsymbol{w}^{(t)} &= -\eta \nabla_{\boldsymbol{w}} R(\boldsymbol{w})|_{\boldsymbol{w}^{(t)}} \\ \boldsymbol{w}^{(t+1)} &= \boldsymbol{w}^{(t)} + \Delta \boldsymbol{w}^{(t)}, \end{aligned} \tag{7.3}$$

where $\nabla_{\boldsymbol{w}} R(\boldsymbol{w})|_{\boldsymbol{w}^{(t)}}$ is the gradient of the objective function with respect to the parameter $\boldsymbol{w}$ evaluated at $\boldsymbol{w}^{(t)}$. We remember that gradient descent is an iterative procedure, in which we start with some parameters $\boldsymbol{w}^{\text{init}} = \boldsymbol{w}^{(0)}$ that we gradually change by update steps:

$$\boldsymbol{w}^{(0)} \rightarrow \boldsymbol{w}^{(1)} \rightarrow \ldots \rightarrow \boldsymbol{w}^{(t)} \tag{7.4}$$

The change at each time step is given by $\Delta \boldsymbol{w}^{(t)}$. We will denote $\boldsymbol{g}^{(t)} = \nabla_{\boldsymbol{w}} R(\boldsymbol{w})|_{\boldsymbol{w}^{\text{t}}}$ the gradient at update step t. The iterative procedure needs some initial parameters $\boldsymbol{w}^{\text{init}} = \boldsymbol{w}^{(0)}$ that are typically drawn from a random distribution (see also Chapter 9). Then we calculate the gradient at this location to obtain $\boldsymbol{g}^{(0)}$, by which we calculate $\Delta \boldsymbol{w}^{(0)}$ to obtain $\boldsymbol{w}^{(1)}$ and so on.

We can show that in case of a convex and differentiable R, gradient descent converges to the optimal solution. Intuitively, this means that in this case gradient descent is guaranteed to converge and that it converges with rate $O(1/t)$ The exact formulation is the following:

Theorem 7.1. *Suppose the function $R : \mathbb{R}^W \mapsto \mathbb{R}$ is convex and differentiable, and that its gradient is Lipschitz continuous with constant $L > 0$, i.e. $\|\nabla R(\boldsymbol{w}) - \nabla R(\boldsymbol{v})\|_2 \leq L\|\boldsymbol{w} - \boldsymbol{v}\|_2$ for any $\boldsymbol{w}, \boldsymbol{v}$. Then gradient descent with a fixed step size $\eta \leq 1/L$ will yield a solution $\boldsymbol{w}^{(t)}$ which satisfies*

$$R(\boldsymbol{w}^{(t)}) - R(\boldsymbol{w}^*) \leq \frac{\|\boldsymbol{w}^{(0)} - \boldsymbol{w}^*\|_2^2}{2\eta t}, \tag{7.5}$$

where $R(\boldsymbol{w}^*)$ is the optimal value.

Proof. We assumed that ∇R is globally Lipschitz continuous, therefore $\nabla^2 R(\boldsymbol{w})$ is bounded for all $\boldsymbol{w}$. Next we perform a Taylor expansion[2] up to order 2 and afterward apply the fact that $\|\boldsymbol{X}\boldsymbol{v}\| \leq \|\boldsymbol{X}\|\|\boldsymbol{v}\|$ for any matrix $\boldsymbol{X}$ and vector $\boldsymbol{v}$, so that we end up with:

$$\begin{aligned} R(\boldsymbol{v}) &= R(\boldsymbol{w}) + (\boldsymbol{v}-\boldsymbol{w})^T \nabla R(\boldsymbol{w}) + \frac{1}{2}(\boldsymbol{v}-\boldsymbol{w})^T \nabla^2 R(\boldsymbol{m})(\boldsymbol{v}-\boldsymbol{w}) \\ &\leq R(\boldsymbol{w}) + (\boldsymbol{v}-\boldsymbol{w})^T \nabla R(\boldsymbol{w}) + \frac{1}{2} L \|(\boldsymbol{v}-\boldsymbol{w})\|_2^2, \end{aligned} \tag{7.6}$$

where $\boldsymbol{m}$ is a vector that lies on the straight line connecting $\boldsymbol{v}$ and $\boldsymbol{w}$.

Now, we use this statement for the point $\boldsymbol{v} = \boldsymbol{w}^{\text{new}} = \boldsymbol{w} - \eta \nabla R(\boldsymbol{w})$. We obtain:

$$\begin{aligned} R(\boldsymbol{w}^{\text{new}}) &\leq R(\boldsymbol{w}) + (\boldsymbol{w}^{\text{new}} - \boldsymbol{w})^T \nabla R(\boldsymbol{w}) + \frac{1}{2} L \|(\boldsymbol{w}^{\text{new}} - \boldsymbol{w})\|_2^2 \\ &= R(\boldsymbol{w}) + (\boldsymbol{w} - \eta \nabla R(\boldsymbol{w}) - \boldsymbol{w})^T \nabla R(\boldsymbol{w}) + \frac{1}{2} L \|(\boldsymbol{w} - \eta \nabla R(\boldsymbol{w}) - \boldsymbol{w})\|_2^2 \\ &= R(\boldsymbol{w}) - \eta \nabla R(\boldsymbol{w})^T \nabla R(\boldsymbol{w}) + \frac{1}{2} L \|(\eta \nabla R(\boldsymbol{w}))\|_2^2 \\ &= R(\boldsymbol{w}) - \eta \|\nabla R(\boldsymbol{w})\|_2^2 + \frac{1}{2} L \eta^2 \|(\nabla R(\boldsymbol{w}))\|_2^2 \\ &= R(\boldsymbol{w}) - (1 - \frac{1}{2} L\eta) \eta \|\nabla R(\boldsymbol{w})\|_2^2. \end{aligned} \tag{7.7}$$

We now use $\eta \leq 1/L$ and find that the term $-(1 - \frac{1}{2}L\eta) \leq -\frac{1}{2}$. This result together with (7.7) provides.

$$R(\boldsymbol{w}^{\text{new}}) \leq R(\boldsymbol{w}) - \frac{1}{2}\eta \|\nabla R(\boldsymbol{w})\|_2^2. \tag{7.8}$$

Since the expression $\frac{1}{2}\eta\|\nabla R(\boldsymbol{w})\|_2^2$ is always positive except for $\nabla R(\boldsymbol{w}) = 0$, this inequality implies that the objective function value decreases with each iteration of gradient descent until it reaches the optimal value $R(\boldsymbol{w}) = R(\boldsymbol{w}^*)$. Note that this convergence result only holds when we choose η to be small enough, i.e. $\eta \leq \frac{1}{L}$. This explains why we observe in practice that gradient descent diverges when the step size is too large.

Next, we can bound $R(\boldsymbol{w}^{\text{new}})$, the objective value at the next iteration, in terms of $R(\boldsymbol{w}^*)$, the optimal objective value. Since R is convex, we can write

$$\begin{aligned} R(\boldsymbol{w}^*) &\geq R(\boldsymbol{w}) + \nabla R(\boldsymbol{w})^T (\boldsymbol{w}^* - \boldsymbol{w}) \\ R(\boldsymbol{w}) &\leq R(\boldsymbol{w}^*) + \nabla R(\boldsymbol{w})^T (\boldsymbol{w} - \boldsymbol{w}^*), \end{aligned} \tag{7.9}$$

where the first inequality[3] yields the second through rearrangement of terms. Combining this

[2] actually we are using Taylor's theorem here

[3] This inequality stems from the fact that for convex functions the graph lies above all tangents.

with Eq. (7.8), we obtain

$$
\begin{aligned}
R(\boldsymbol{w}^{\text{new}}) &\leq R(\boldsymbol{w}^*) + \nabla R(\boldsymbol{w})^T(\boldsymbol{w} - \boldsymbol{w}^*) - \frac{\eta}{2}\|\nabla R(\boldsymbol{w})\|_2^2 \\
R(\boldsymbol{w}^{\text{new}}) - R(\boldsymbol{w}^*) &\leq \frac{1}{2\eta}\left(2\eta\nabla R(\boldsymbol{w})^T(\boldsymbol{w} - \boldsymbol{w}^*) - \eta^2\|\nabla R(\boldsymbol{w})\|_2^2\right) \\
R(\boldsymbol{w}^{\text{new}}) - R(\boldsymbol{w}^*) &\leq \frac{1}{2\eta}\left(2\eta\nabla R(\boldsymbol{w})^T(\boldsymbol{w} - \boldsymbol{w}^*) - \eta^2\|\nabla R(\boldsymbol{w})\|_2^2 - \|\boldsymbol{w} - \boldsymbol{w}^*\|_2^2 + \|\boldsymbol{w} - \boldsymbol{w}^*\|_2^2\right) \\
R(\boldsymbol{w}^{\text{new}}) - R(\boldsymbol{w}^*) &\leq \frac{1}{2\eta}\left(\|\boldsymbol{w} - \boldsymbol{w}^*\|_2^2 - \|\boldsymbol{w} - \eta\nabla R(\boldsymbol{w}) - \boldsymbol{w}^*\|_2^2\right), \qquad (7.10)
\end{aligned}
$$

where the final inequality is obtained by observing that expanding the square of $\|\boldsymbol{w} - \eta\nabla R(\boldsymbol{w}) - \boldsymbol{w}^*\|_2^2$ yields $\|\boldsymbol{w} - \boldsymbol{w}^*\|_2^2 - 2\eta\nabla R(\boldsymbol{w})^T(\boldsymbol{w} - \boldsymbol{w}^*) + \eta^2\|\nabla R(\boldsymbol{w})\|_2^2$. Notice that by definition we had $\boldsymbol{w}^{\text{new}} = \boldsymbol{w} - \eta\nabla R(\boldsymbol{w})$. Inserting this into the last term of Eq. (7.10), we get:

$$
R(\boldsymbol{w}^{\text{new}}) - R(\boldsymbol{w}^*) \leq \frac{1}{2\eta}\left(\|\boldsymbol{w} - \boldsymbol{w}^*\|_2^2 - \|\boldsymbol{w}^{\text{new}} - \boldsymbol{w}^*\|_2^2\right). \qquad (7.11)
$$

The inequality holds for $\boldsymbol{w}^{\text{new}}$ on every iteration of gradient descent. Summing over iterations, we get:

$$
\begin{aligned}
\sum_{t=0}^{T} R(\boldsymbol{w}^{(t)}) - R(\boldsymbol{w}^*) &\leq \sum_{t=0}^{T} \frac{1}{2\eta}\left(\|\boldsymbol{w}^{(t-1)} - \boldsymbol{w}^*\|_2^2 - \|\boldsymbol{w}^{(t)} - \boldsymbol{w}^*\|_2^2\right) \\
&= \frac{1}{2\eta}\left(\|\boldsymbol{w}^{(0)} - \boldsymbol{w}^*\|_2^2 - \|\boldsymbol{w}^{(T)} - \boldsymbol{w}^*\|_2^2\right) \\
&\leq \frac{1}{2\eta}\left(\|\boldsymbol{w}^{(0)} - \boldsymbol{w}^*\|_2^2\right), \qquad (7.12)
\end{aligned}
$$

where the summation on the right-hand side disappears because it is a telescoping sum. Finally, using the fact the R is decreasing on every iteration, we can conclude that

$$
\begin{aligned}
R(\boldsymbol{w}^{(T)}) - R(\boldsymbol{w}^*) &\leq \frac{1}{T}\sum_{t=1}^{T} R(\boldsymbol{w}^{(t)}) - R(\boldsymbol{w}^*) \\
&\leq \frac{\|\boldsymbol{w}^{(0)} - \boldsymbol{w}^*\|_2^2}{2\eta T}, \qquad (7.13)
\end{aligned}
$$

where we used Eq. (7.12) in the final step to obtain our result.

□

7.2 Gradient descent with Momentum Term

Sometimes gradient descent oscillates or slows down because the error surface (the objective function) has a flat plateau. To avoid oscillation and to speed up gradient descent a momentum term $\boldsymbol{m}^{(t)}$ can be used. The oscillation of the gradient is depicted in Fig. 7.3 and the reduction of

oscillation through the momentum term in Fig. 7.4. This is achieved by using an exponentially decaying average over the past gradients. In this way the old gradient is memorized in $\boldsymbol{m}^{(t)}$ and altered towards the new direction.

Another effect of the momentum term is that in flat regions the gradients pointing in the same directions are accumulated and the learning rate is implicitly increased. The problem of flat regions is depicted in Fig. 7.5 and the speed up of the convergence in flat regions in Fig. 7.6.

The gradient descent update with momentum term is

$$\begin{aligned} \boldsymbol{m}^{(t)} &= \mu \boldsymbol{m}^{(t-1)} + (1-\mu)\boldsymbol{g}^{(t)} \\ \boldsymbol{w}^{(t+1)} &= \boldsymbol{w}^{(t)} - \eta \boldsymbol{m}^{(t)} \end{aligned} \tag{7.14}$$

where $0 \leq \mu \leq 1$ is the *momentum parameter* or *momentum factor* which is usually close to one.

7.2.1 Physical view of momentum

From the point of physics, learning with momentum term has a particular interpretation as the velocity m of a physical object, as we will see in the following.

We can write $m^{(t)}$ explicitly as:

$$m^{(t)} = \mu m^{(t-1)} + (1-\mu)g^{(t)} = \tag{7.15}$$

$$= \mu^2 m^{(t-2)} + \mu(1-\mu)g^{(t-1)} + (1-\mu)g^{(t)} = \tag{7.16}$$

$$= \mu^3 m^{(t-3)} + \mu^2(1-\mu)g^{(t-2)} + \mu(1-\mu)g^{(t-1)} + (1-\mu)g^{(t)} = \tag{7.17}$$

$$= \mu^t m^{(0)} + (1-\mu)\sum_{i=1}^{t} \mu^{t-i} g^{(i)} \tag{7.18}$$

$$\tag{7.19}$$

From this and $m^{(0)} = 0$ it follows that

$$\Delta w^{(t)} = -\sum_{i=1}^{t} \mu^{t-i} \left(\eta(1-\mu) g^{(i)} \right) \tag{7.20}$$

$$\tag{7.21}$$

which means that with we could also write the algorithm as:

$$\boldsymbol{m}^{(t)} = \mu \boldsymbol{m}^{(t-1)} - \eta' \boldsymbol{g} \tag{7.22}$$

$$\Delta \boldsymbol{w}^{(t)} = \boldsymbol{m}^{(t)} \tag{7.23}$$

with $\eta' = -\eta(1-\mu)$. This makes an analogy with a physical system easier. We can view $\boldsymbol{m}^{(t)}$ as a velocity and look at its rate of change and if we assume a mass of one, the left hand side below is equal to the change of momentum from $t-1$ to t. This interpretation means that we can interpret the right hand side as a force times a time interval Δt (which is equal to one so we can insert it).

$$\boldsymbol{m}^{(t)} - \boldsymbol{m}^{(t-1)} = \left(-(1-\mu)\boldsymbol{m}^{(t-1)} - \eta' \boldsymbol{g}^{(t)} \right) \Delta t \tag{7.24}$$

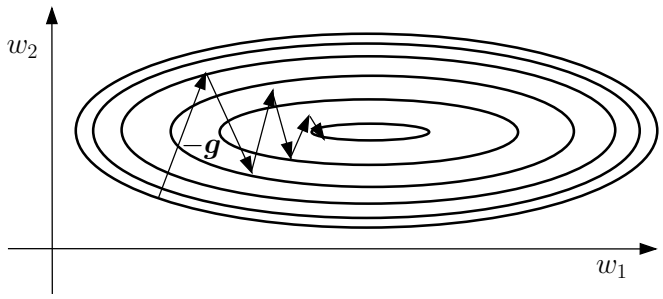


Figure 7.3: The negative gradient $-\boldsymbol{g}$ oscillates as it converges to the minimum.

Taking the limit as $\Delta t \to 0$ we get the following differential equation:

$$\frac{d\boldsymbol{m}(t)}{dt} = \underbrace{-(1-\mu)\boldsymbol{m}(t)}_{\text{Friction}} \underbrace{-\eta'\boldsymbol{g}(t)}_{\text{Lossforce}}, \tag{7.25}$$

where $m(t)$ is a function of t now, because we moved from discrete time steps to continuous ones.

First we look at the case when $\boldsymbol{g}(t) = 0$ and see that

$$\boldsymbol{m}(t) = \boldsymbol{m}(t_0)\, e^{-(1-\mu)t}, \tag{7.26}$$

and because $0 < 1 - \mu < 1$ this corresponds to an exponential decay in the velocity. What we have modelled here is a body that slows down due to friction. We can also see that if $\mu = 1$ the velocity is constant. When looking at the whole of (7.25) we now see that it consists of a friction term and a force stemming from the loss that is increasing the velocity in the negative direction of the gradient.

Now we take a look at the parameter update in continuous time:

$$\boldsymbol{w}^{(t)} - \boldsymbol{w}^{(t-1)} = \boldsymbol{m}^{(t)}\Delta t \tag{7.27}$$

$$\frac{\boldsymbol{w}^{(t)} - \boldsymbol{w}^{(t-1)}}{\Delta t} = \boldsymbol{m}^{(t)}. \tag{7.28}$$

In the limit $\Delta t \to 0$ we get:

$$\frac{d\boldsymbol{w}(t)}{dt} = \boldsymbol{m}(t), \tag{7.29}$$

which is again easy to interpet. If $\boldsymbol{w}(t)$ is a position in space its rate of change w.r.t. to time is the velocity. By differentation of (7.29) and insertion of (7.25) it follows that

$$\frac{d^2\boldsymbol{w}(t)}{dt^2} = -(1-\mu)\frac{d\boldsymbol{w}(t)}{dt} - \eta'\boldsymbol{g}(t). \tag{7.30}$$

which is a differential equation describing the motion of $\boldsymbol{w}(t)$ in parameter space.

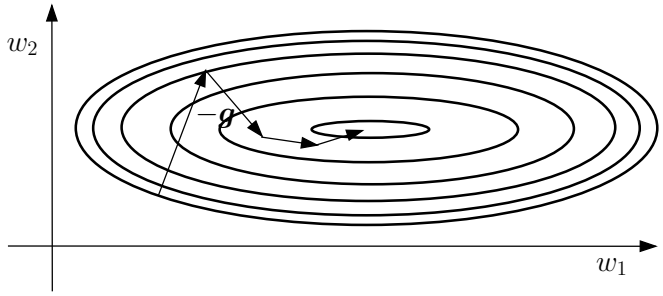


Figure 7.4: Using the momentum term the oscillation of the negative gradient $-\boldsymbol{g}$ is reduced because consecutive gradients which point in opposite directions are superimposed. The minimum is found faster.

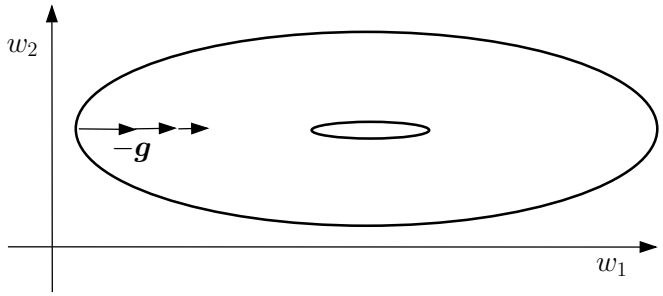


Figure 7.5: The negative gradient $-\boldsymbol{g}$ lets the weight vector converge very slowly to the minimum if the region around the minimum is flat.

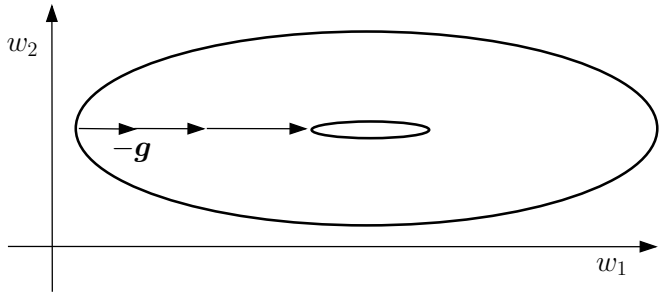


Figure 7.6: The negative gradient $-\boldsymbol{g}$ is accumulated through the momentum term because consecutive gradients point into the same directions. The minimum is found faster.

7.3 Stochastic Gradient Descent (SGD)

For neural networks, the objective function that is minimized is typically an average over the loss of individual data points

$$R_{\text{emp}}(\boldsymbol{w}, \boldsymbol{X}, \boldsymbol{y}) = \frac{1}{N} \sum_{n=1}^{N} L\left(y^n, \boldsymbol{g}\left(\boldsymbol{x}^n; \boldsymbol{w}\right)\right), \tag{7.31}$$

where $\boldsymbol{w}$ is the vector of parameters and L is the loss function for a single data point $\boldsymbol{x}^n$ with label y^n. *Full-batch* gradient descent requires to compute the gradient for each sample and then take the average over it

$$\nabla_{\boldsymbol{w}} R_{\text{emp}}(\boldsymbol{w}, \boldsymbol{X}, \boldsymbol{y}) = \frac{1}{N} \sum_{n=1}^{N} \nabla_{\boldsymbol{w}} L\left(y^n, \boldsymbol{g}\left(\boldsymbol{x}^n; \boldsymbol{w}\right)\right). \tag{7.32}$$

Once, this gradient is calculated, a step in the negative direction is taken to update the parameters (see 7.1). However, if the data set is large, this is a quite computationally costly procedure because it requires evaluating the model on every example in the data set. Thus we could also aim at estimating the gradient on a random subset of the training set, that we call *mini-batch* $\mathcal{B} = \{j_1, \ldots, j_B\} \subset \{1, \ldots, N\}$ with batch size B.

$$\nabla_{\boldsymbol{w}} R_{\text{emp}}(\boldsymbol{w}, \boldsymbol{X}, \boldsymbol{y}) \approx \frac{1}{B} \sum_{b=1}^{B} \nabla_{\boldsymbol{w}} L\left(y^{j_b}, \boldsymbol{g}\left(\boldsymbol{x}^{j_b}; \boldsymbol{w}\right)\right). \tag{7.33}$$

We sometimes use the symbol $\tilde{\nabla}_{\boldsymbol{w}}$ for the expression on the right hand side. Gradient descent using these estimates of the gradient based on mini-batches is called *stochastic gradient descent (SGD)*. An extreme case of stochastic gradient descent is when $B = 1$, single samples are used. This is also called *on-line learning*, a setting in which samples are presented sequentially to the neural network.

Efficiency of stochastic gradient descent. We note that the expression for the empirical error is actually an expectation of the loss of the training set (Eq. (7.32)) and that the gradient similarly:

$$\nabla_{\boldsymbol{w}} R_{\text{emp}}(\boldsymbol{w}, \boldsymbol{X}, \boldsymbol{y}) = \mathbb{E}_{\boldsymbol{x}, y \sim \tilde{p}_{\text{data}}} \left[\nabla_{\boldsymbol{w}} L\left(y, \boldsymbol{g}\left(\boldsymbol{x}; \boldsymbol{w}\right)\right)\right], \tag{7.34}$$

where $\tilde{p}_{\text{data}}$ is the empirical distribution. When we use stochastic gradient descent, we actually approximate this expectation by randomly sampling a small number of examples from the training set. Note that the improvement of the estimate does not scale linearly with batch size B, since the standard error of the mean of B samples is given by $\frac{\sigma}{\sqrt{B}}$, where σ is the true standard deviation in the samples (Goodfellow et al., 2016). Thus, assuming that the full data set has $N = 100,000$ samples and we calculate the gradient based on a subset of $B = 1000$ samples, the standard error of our estimate only increases by a factor of 10, whereas the computational costs are decreased by a factor of 100.

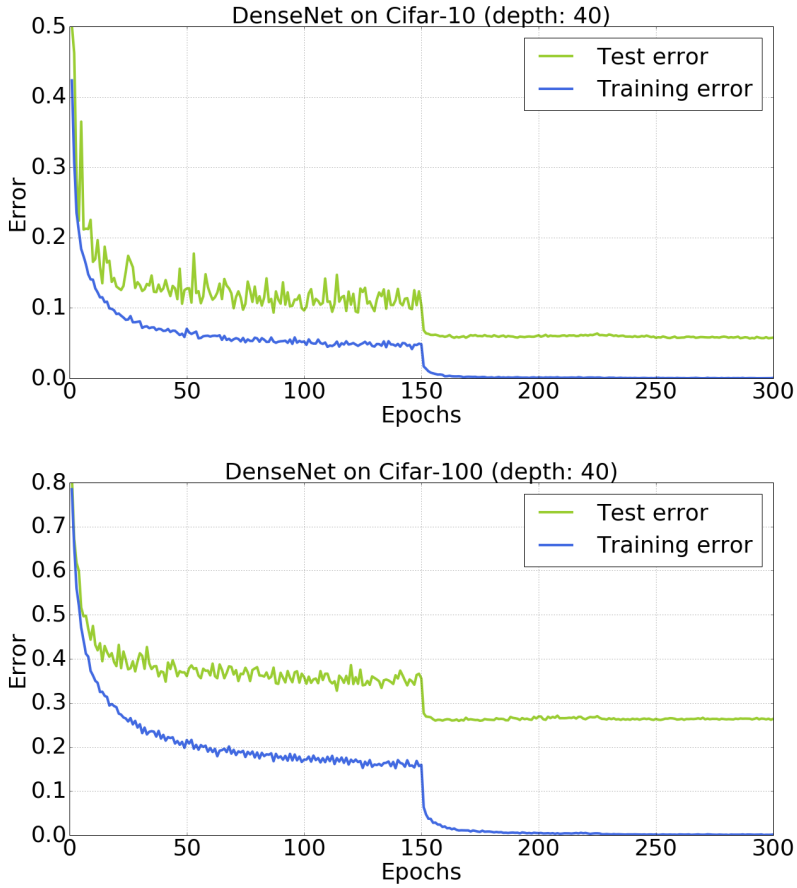


Figure 7.7: Learning curves for training of a neural network (DenseNet) on CIFAR-10 (left) and CIFAR-100 (right). The x-axis displays training epochs and the y-axis displays the loss on the training set (blue) and the loss on the test set (green). After each training epoch, the loss on both the training and the test set was calculated. Note that the learning rate was decreased by a factor of 10 after 150 epochs of training, which causes the drop in both learning curves.

Mini-batches, sampling without replacement, epochs and learning curves How stochastic gradient descent is practically employed is the following: A mini-batch of a certain size, e.g. 32, is drawn from the training set and removed from this set by sampling without replacement. The gradient on this mini-batch is calculated and the parameters are updated. This is counted as an update step. When the training set is depleted, where potentially the last batch is an incomplete batch, an *epoch* has ended. Thus a *training epoch* means that each sample in the training set has been presented once to the network. The number of updates per epoch is $\lceil N/B \rceil$. During training, practitioners monitor the loss and performance metric on the training set, but frequently also on a validation set – for example after an epoch is completed. This produces so-called *learning curves*, of which an example is given here:

Properties of stochastic gradient descent SGD even introduces beneficial effects for the learning dynamics of neural networks via regularization (Wilson and Martinez, 2003; Chaudhari and

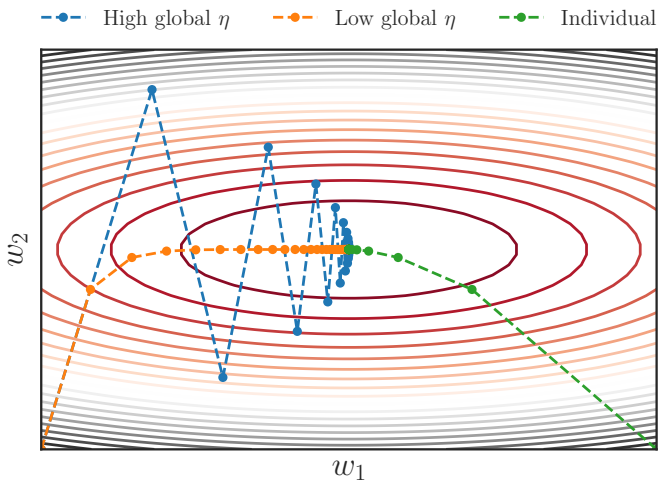


Figure 7.8: This plot shows gradient descent in a two dimensional optimization problem. The contour lines represent the loss function. The curvature difference between the two directions causes problems in finding a single good learning rate for all parameters.

Soatto, 2018).

7.4 Adaptive Learning Rate Optimizers

Choosing the learning rate for gradient descent is critical for their success. If it is too high the parameter update may "overshoot" its goal, while progress may be slow if it is too low. A second problem is that each parameter might benefit from different learning rates. If we use a specific learning rate η_i for each parameter w_i the parameter update becomes:

$$\Delta w_i = -\eta_i g_i \quad \text{or} \quad \Delta \boldsymbol{w} = -\boldsymbol{\eta} \odot \boldsymbol{g}. \tag{7.35}$$

where $g_i = \frac{\partial R(\boldsymbol{w})}{\partial w_i}$ is the i-th component of the gradient $\boldsymbol{g}$. The following example should show why using such an update rule can be beneficial.

Example In this example we want to find the optimum of the function $f(w_1, w_2) = w_1^2 + 5w_2^2$ using gradient descent. The contour lines of that function and gradient descent paths using different learning rates are shown in Figure 7.8. First we can see in the blue curve that a high global learning rate causes oscillations because the curvature in direction of w_2 is large. Looking at the orange curve we see that a low global learning rate is not satisfying either because it takes many update steps to reach the optimum in the direction of w_1. In the green curve, however, we chose a small η_2 to avoid oscillations and a larger η_1 to speed up convergence in direction of w_1. The green path is mirrored across the middle vertical line for the sake of visualization.

Up till now it is not clear that $\Delta \boldsymbol{w}$ will lead us to a point with lower error. While the direction of steepest descent is parallel to the gradient we can also show that for positive η_i we still get

improvements. The error function can be written as:

$$R(\boldsymbol{w} + \boldsymbol{\Delta w}) = R(\boldsymbol{w}) + \boldsymbol{g}^T \Delta \boldsymbol{w} + \mathcal{O}(\Delta \boldsymbol{w}^2) \tag{7.36}$$

Using the update rule (7.35) we can show that the update yields an improvement in the linear approximation by looking at the linear term above.

$$-\boldsymbol{g}^T (\boldsymbol{\eta} \odot \boldsymbol{g}) = -\sum_i g_i g_i \eta_i = \tag{7.37}$$

$$\leq -\sum_i g_i g_i \eta_{\text{min}} \tag{7.38}$$

$$= -\eta_{\text{min}} \sum_i g_i g_i \tag{7.39}$$

$$= -\eta_{\text{min}}\, \boldsymbol{g}^T \boldsymbol{g} \leq 0 \tag{7.40}$$

Thus we get

$$R(\boldsymbol{w} + \boldsymbol{\Delta w}) \leq R(\boldsymbol{w}), \tag{7.41}$$

by combining (7.35) and (7.36) and assuming that $\Delta \boldsymbol{w}$ is small enough to ignore higher order terms.

A third issue of choosing a fixed learning rate, is that this assumes that the loss surface looks similar everywhere. In order to ensure rapid convergence, it is useful to adapt the learning rate to the topology of the loss surface at the current point (Jacobs, 1988). Concretely, this means that the learning rate should be allowed to vary as learning proceeds. For update rules with adaptive learning rates, one can write

$$\boldsymbol{w}^{(t+1)} = \boldsymbol{w}^{(t)} - \boldsymbol{\eta}^{(t)} \odot \boldsymbol{g}^{(t)}. \tag{7.42}$$

The problems of finding good learning rates for each parameter that also adapt as optimization continues led to the development of numerous optimization algorithms that use varying and distinct learning rates for each parameter. Most of these methods approximate or incorporate second order information in some way and commonly allow for faster convergence with less hyperparameter tuning. Next we show different methods of how to determine the η_i in practice.

7.4.1 Delta-Delta Rule

The *delta-delta rule* is a typical example of an adaptive learning rate schedule (Barto and Sutton, 1981; Jacobs, 1988). In this algorithm one does not only update the model parameters but also the learning rate for each parameter. At step t the learning rates is given by:

$$\eta_i^{(t)} = \eta_i^{(t-1)} + \gamma g_i^{(t)} g_i^{(t-1)}, \tag{7.43}$$

where γ would be the step size for the gradient descent in learning rate space.

Intuitively, the delta-delta rule will increase the learning rate if the sign of two consecutive updates is the same. If the gradients point in opposite directions, however, the learning rate will decrease. Since the magnitude of the in-/decrease depends on the magnitude of the gradients, this rule might not be satisfactory when the loss surface consists of large plateaus, where the learning rate can only be updated slowly due to small gradients. The issues are even worse for regions with

high curvature, where large gradients will likely cause oscillating gradients, which can result in negative learning rates. (Jacobs, 1988).

The *delta-bar-delta rule* was designed to alleviate these issues by ignoring the magnitude of the gradients in the actual learning rate updates (Jacobs, 1988). The learning rate is updated according to:

$$\begin{aligned} \eta_i^{(t)} &= \eta_i^{(t-1)} + \Delta\eta_i^{(t)} \\ \Delta\eta_i^{(t)} &= \begin{cases} \kappa & \bar{g}_i^{(t)} g_i^{(t)} > 0 \\ -\phi\eta_i^{(t-1)} & \bar{g}_i^{(t)} g_i^{(t)} < 0 \\ 0 & \text{otherwise} \end{cases}, \end{aligned} \tag{7.44}$$

where

$$\bar{g}_i^{(t)} = \theta\bar{g}_i^{(t-1)} + (1-\theta)g_i^{(t)}.$$

Whereas the *delta-delta rule* in (7.43) uses the previous gradient to detect direction changes, the *delta-bar-delta rule* uses an exponentially weighted average, $\bar{g}_i$, of all previous gradients base θ. The learning rate for dimension i is increased by κ if the sign of the current gradient equals the one of the moving average in dimension i and is exponentially decreased with a factor of $1-\phi$ if not. This effectively prevents η_i from becoming negative and allows the learning rate to keep growing on large plateaus.

Note that the delta-bar-delta rule requires quite a bit of hyper-parameters to be chosen, compared to standard gradient descent methods. First of all, a vector of initial learning rates, $\boldsymbol{\eta}^{(0)}$, must be chosen, which is often done by setting all learning rates to some reasonable, small positive value. Furthermore, the increment, $\kappa \in \mathbb{R}^+$, the downscaling factor, $0 < \phi < 1$, and the base for the exponential moving average of gradients, $0 < \theta < 1$ must be set on top of possible other hyperparameters of the learning algorithm, e.g. momentum μ.

7.4.2 AdaGrad

The Adagrad optimizer (Duchi et al., 2011) adapts step sizes by remembering past gradients. The parameter wise learning rates are scaled down for parameters with large past gradients and up for ones with small gradients. This has the nice property, as in second order methods, that the progress along each dimension evens out over time (Zeiler, 2012). For each parameter we store the sum of squared past gradients:

$$v_i^{(t)} = \sum_{s=1}^{t} (g_i^{(s)})^2, \tag{7.45}$$

where $g_i^{(s)}$ is the i-th component of the gradient at step s. Now we use this history of gradients to scale the learning rate in the following way:

$$w_i^{(t+1)} = w_i^{(t)} - \eta \frac{1}{\sqrt{v_i^{(t)} + \epsilon}} g_i^{(t)}, \qquad \boldsymbol{w}^{(t+1)} = \boldsymbol{w}^{(t)} - \eta \frac{1}{\sqrt{\boldsymbol{v}^{(t)} + \epsilon}} \odot \boldsymbol{g}^{(t)}, \tag{7.46}$$

where the right hand side is the update rule in vectorial notation, in which $\odot$ means element-wise multiplication and division through a vector is also to be understood elementwise. Adding a scalar

to a vector is to be interpreted as adding the scalar to each element of the vector. The variable ϵ is a parameter that accounts for potential numerical problems with the division and is typically set to values in the magnitude of $\epsilon = 1e - 8$.

Since each parameter is divided by the sum of squared past gradients, parameters with large gradients have smaller learning rates and vice versa. Additionally, the learning rates decrease over time as the $v_i^{(t)}$ grow over time. While this can help training in general it might also lead to problems as learning rates could shrink too much and hinder further learning.

This update rule is often useful for deep neural networks as the scale of the gradients in each layer is often different by several orders of magnitude and optimization can benefit from scaled gradients Zeiler (2012).

7.4.3 Adadelta and RMSprop

As we already mentioned Adagrad sums up the past squared gradients by $v_i^{(t)} = \sum_{s=1}^{t} {g_i^{(s)}}^2$ and parameters that have already received large updates are bound to small learning rates. Adadelta (Zeiler, 2012) and RMSprop (Tieleman and Hinton, 2012) tackle this problem by using an exponential running average over previous gradients,

$$\boldsymbol{v}^{(t)} = \gamma \boldsymbol{v}^{(t-1)} + (1-\gamma)(\boldsymbol{g}^{(t)})^2 \tag{7.47}$$

instead of the sum in (7.45). The square of a vector is to be understood as elementwise here. The parameter $\gamma \in [0, 1)$ can be set to similar values as a momentum term, e.g. $\gamma = 0.9$. The update for RMSprop is identical to (7.46) except that we use $\boldsymbol{v}^{(t)}$ as defined in (7.47).

The update rule for Adadelta adds the moving average over past parameter updates as an additional scaling factor. The complete algorithm looks as follows:

$$\boldsymbol{v}^{(t)} = \gamma \boldsymbol{v}^{(t-1)} + (1-\gamma)(\boldsymbol{g}^{(t)})^2 \tag{7.48}$$

$$\boldsymbol{d}^{(t-1)} = \gamma \boldsymbol{d}^{(t-2)} + (1-\gamma)(\Delta \boldsymbol{w}^{(t-1)})^2 \tag{7.49}$$

$$\Delta \boldsymbol{w}^{(t)} = \frac{\sqrt{\boldsymbol{d}^{(t-1)} + \epsilon}}{\sqrt{\boldsymbol{v}^{(t)} + \epsilon}} \odot \boldsymbol{g}^{(t)}, \tag{7.50}$$

where the squares, and divisions are again to be understood elementwise. This update can be considered a scaled version of RMSprop that does not require an explicit learning rate. The motivation for introducing the extra scaling factor stems from the idea that the update should have the same "unit" as the original parameters (Zeiler, 2012).

7.4.4 Adam

Adaptive moment estimation (Adam) (Kingma and Ba, 2014) combines the idea of momentum term (see Section 7.2) and RMSProp (see Section 7.4.3) by memorizing both exponentially decaying averages of *past gradients* $\boldsymbol{g}^{(t)}$ and *past squared gradients* $(\boldsymbol{g}^{(t-1)})^2$ (the exponent refers to element-wise exponentiation). We define the following variables:

$$\boldsymbol{m}^{(t)} = \beta_1 \boldsymbol{m}^{(t-1)} + (1-\beta_1)\boldsymbol{g}^{(t)}$$
$$\boldsymbol{v}^{(t)} = \beta_2 \boldsymbol{v}^{(t-1)} + (1-\beta_2)(\boldsymbol{g}^{(t)})^2,$$

where $\boldsymbol{m}^{(t)}$ contains the exponentially averaged past gradients (the first moment or the mean) and $\boldsymbol{v}^{(t)}$ the exponentially averaged past squared gradients (the second moment or the uncentered variance). β_1 and β_2 are again decay rates that are set to values close to 1, e.g. $\beta_1 = 0.9$ and $\beta_2 = 0.999$. In line with the update rule of AdaGrad, the parameter update for Adam is:

$$\boldsymbol{w}^{(t+1)} = \boldsymbol{w}^{(t)} - \eta \frac{1}{\sqrt{\boldsymbol{v}^{(t)}} + \epsilon} \odot \boldsymbol{m}^{(t)}, \tag{7.51}$$

where ϵ is again a small value for numeric stability, and the division is to be understood element-wise. Typical learning rates for deep neural networks trained with the Adam optimizer range from $1e-3$ to $1e-5$.

7.5 First Order Gradient Alternatives

Although back-propagation has become the standard learning algorithm for neural networks, there are some alternatives to using the exact gradient for updating the network weights. A recurring theme in the naming for these update rules is the "prop" suffix, which refers to "backprop", the commonly used abbreviation for the back-propagation algorithm.

7.5.1 Quickprop

In *quickprop*, the main idea is to assume that the loss surface is locally quadratic (Fahlman, 1991). Given the last two gradients, it is then possible to jump directly to the minimum of the assumed parabola. The resulting update rule is:

$$\Delta^{\text{new}} w_i = \frac{g_i^{\text{new}}}{g_i^{\text{old}} - g_i^{\text{new}}} \Delta^{\text{old}} w_i. \tag{7.52}$$

One or more gradient descent steps can be used to bootstrap this process.

To understand this update rule, let us assume that $R(\boldsymbol{w})$ is a piece-wise function so that $R(\boldsymbol{w})_i = R(w_i)$, then the Taylor expansion for each element is

$$R(w_i + \Delta w_i) = R(w_i) + g_i \Delta w_i + \frac{1}{2} g_i' (\Delta w_i)^2 + \mathcal{O}\big(\big((\Delta w_i)^3\big)\big),$$

where g_i' is the second order derivative of the objective w.r.t. w_i. This is a quadratic approximation of the function. See Fig. 7.9 for an example of this approximation.

To maximize the change in the objective due to the update, one can minimize

$$R(w_i + \Delta w_i) - R(w_i) = g_i \Delta w_i + \frac{1}{2} g_i' (\Delta w_i)^2.$$

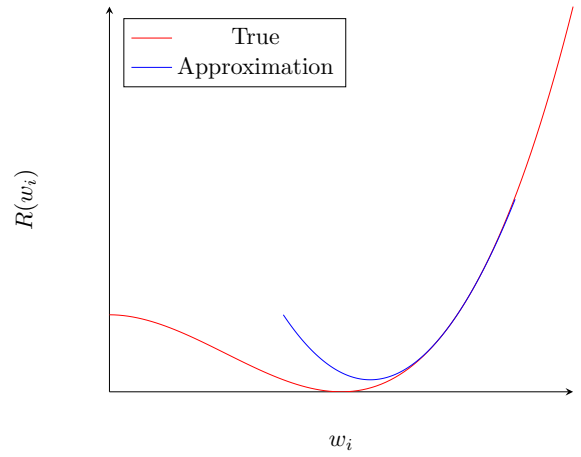


Figure 7.9: The error surface (the solid curve) is locally approximated by a quadratic function (the dashed curve).

We set the derivative with respect to Δw_i to zero and obtain

$$\Delta w_i = -\frac{g_i}{g_i'}. \tag{7.53}$$

Now, approximate the second order derivative, g_i', in the current point, w_i by the second order derivative in the previous point, $w_i - \Delta^{\text{old}} w_i$, using a difference quotient:

$$g_i' = \frac{g_i^{\text{new}} - g_i^{\text{old}}}{\Delta^{\text{old}} w_i}, \tag{7.54}$$

where g_i^{old} is the gradient of the network with the old weights, $w_i - \Delta^{\text{old}} w_i$. Inserting this approximation into Eq. (7.53) and solving for Δw_i results in the quickprop update rule, Eq. (7.52).

7.5.2 RProp

The *delta-bar-delta rule* ignores the gradient magnitude for updating the learning rate from step to step. The *Rprop* algorithm takes this one step further and ignores the gradient magnitude altogether (Riedmiller and Braun, 1993). Concretely, this is done by replacing the gradient by its sign in the gradient descent update rule:

$$\Delta w_i = -\eta_i^{\text{new}} \text{sign}(g_i^{\text{new}}). \tag{7.55}$$

The magnitude of the weight update therefore solely depends on the learning rate.

$\eta_i^{(t)}$ is an adaptive learning rate that is updated in a similar way as the delta-bar-delta rule, with the difference that changes in both directions are exponential:

$$\eta_i^{\text{new}} = \begin{cases} \eta^+ \eta_i^{\text{old}} & g_i^{\text{old}} g_i^{\text{new}} > 0 \\ \eta^- \eta_i^{\text{old}} & g_i^{\text{old}} g_i^{\text{new}} < 0 \\ \eta_i^{\text{old}} & \text{otherwise} \end{cases}, \tag{7.56}$$

where $0 < \eta^- < 1 < \eta^+$ are the hyperparameters for Rprop.

There is one special rule in Rprop, however, to prevent overshooting the (local) minimum. When the gradient flips sign, i.e. when $g_i^{\text{old}} g_i^{\text{new}} < 0$, the minimum must have been between the weights before the last update and the current weights. In the original formulation of Rprop, this overshooting is simply undone by reverting the last update if the gradients flip signs, i.e.

$$\Delta w_i^{\text{new}} = -\Delta w_i^{\text{old}} \qquad \text{if } g_i^{\text{old}} g_i^{\text{new}} < 0.$$

Note, however, that this additional rule is not necessary, since overshooting only means that the minimum is approached from the opposite side.

7.6 Second-order methods

Second-order methods are treated in Part II of this lecture.

Chapter 8

Regularization

SEPP HOCHREITER AND GÜNTER KLAMBAUER

In chapter 5, we have investigated the relation between the complexity of the model class and the training error, i.e. the empirical error, and the test error. One of the main challenges in machine learning is to chose a model class with the appropriate complexity. This problem is especially pronounced in Deep Learning, because neural network model classes have a high complexity and can approximate functions arbitrarily closely (Hornik et al., 1989).

In Fig. 8.1 the relation between the test error R and the training error as a function of the complexity is depicted. The test error R first decreases and then increases with increasing complexity. The training error decreases with increasing complexity. The test error R is the sum of training error and a complexity term. At some complexity point the training error decreases slower than the complexity term increases – this is the point of the optimal test error.

$$R \leq R_{\text{emp}} + \text{complexity} . \tag{8.1}$$

Increasing complexity leads to smaller training error but the test error, the risk, increases at some point. For low test error, i.e. high generalization, we have to control the complexity of the neural network.

To avoid *overfitting* and allow for generalization, neural networks have to be appropriately regularized and a large number of regularization techniques are currently in use. In this chapter, we review different regularization strategies for (deep) neural networks.

Principle regularization approaches. Typical regularization terms are

- smoothing terms, which control the curvature and higher order derivatives of the function represented by the network
- complexity terms which control number of units and weights or their precision.

If the network extracts characteristics which are unique for training data only (the data points which have been drawn), arise from noise, or are due to outliers in the training data then overfitting is present. In general all regularization terms which avoid that the network can extract such characteristics from the training data can regularize. These terms allow the network only to extract the most prominent characteristics which are not observed at one example but for many examples.

The regularization can be done a) during or b) after training the network.

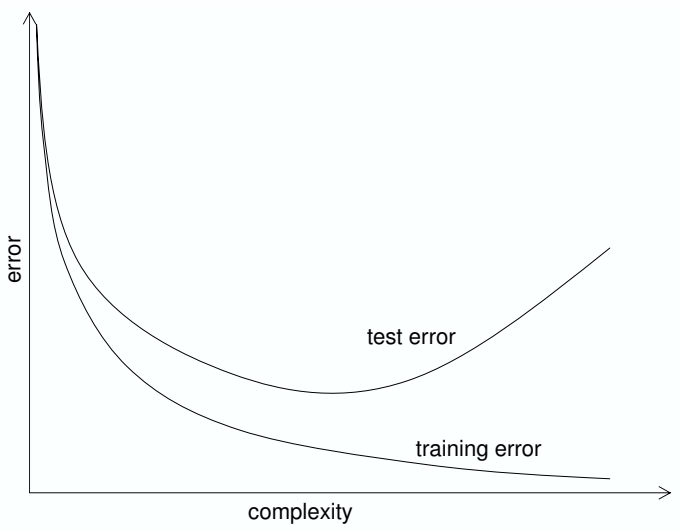


Figure 8.1: Typical example where the test error first decreases and then increases with increasing complexity. The training error decreases with increasing complexity. The test error, the risk, is the sum of training error and a complexity term. At some complexity point the training error decreases slower than the complexity term increases – this is the point of the optimal test error.

Regularization during training. This approach has the problem that there is a trade-off between decreasing the empirical risk and the complexity of the model class. This trade-off is difficult to regulate during learning. Advantage is that a constant pressure is on the learning to efficiently extract the structures.

Regularization after training. This type has the problem that the network can be spoiled by the training procedure. Important structures may distributed over the whole network so that pruning weights or nodes is impossible without destroying the important structure. E.g. a structure which can be represented by a small sub-network may be in a large architecture containing copies of this small sub-network. If now one of these copies is deleted the error increases because it contributes to the whole output. On the other hand the sub-network share the task of representing the important structure therefore have free capacity to represent more characteristics of the training data including noise. Advantage of the regularization after training is that one can better control the complexity and the trade-off between empirical error and complexity.

Because all regularization must be expressed through weight values, a problem appears for all of these methods. The network function can be implemented through different weight values and some weight values may indicate lower complexity even if they implement the same function.

8.1 Weight decay

In this approach, the empirical error function R_{emp} is extended by a complexity term Ω which expresses the network complexity as a function of the weights:

$$R_{\text{reg}}(\boldsymbol{w}) \;=\; R_{\text{emp}}(\boldsymbol{w}) \;+\; \lambda\, \Omega(\boldsymbol{w})\,, \tag{8.2}$$

where we dropped the dependency of $R_{\text{emp}}(\boldsymbol{w})$ on $\boldsymbol{X}$ and $\boldsymbol{y}$ to keep the notation uncluttered. This gives rise to a new objective function $R_{\text{reg}}(\boldsymbol{w})$ that we aim to minimize. Gradient descent is now performed on this regularized error function R_{reg}:

$$\nabla_{\boldsymbol{w}} R_{\text{reg}}(\boldsymbol{w}) \;=\; \nabla_{\boldsymbol{w}} R_{\text{emp}}(\boldsymbol{w}) \;+\; \lambda\, \nabla_{\boldsymbol{w}} \Omega(\boldsymbol{w})\,. \tag{8.3}$$

The term $\Omega(\boldsymbol{w})$ can be viewed as a penalty term that influences the form of the solution. The hyperparameter λ controls the extent to which $\Omega(\boldsymbol{w})$ can influence the solution. The best known term for $\Omega(\boldsymbol{w})$ is *weight decay*, where small absolute weights values are preferred over large absolute weight values.

L2 weight decay. A natural choice for the complexity term is the 2-norm of the weights (Krogh and Hertz, 1992):

$$R_{\text{reg}}(\boldsymbol{w}) \;=\; R_{\text{emp}}(\boldsymbol{w}) \;+\; \frac{\lambda}{2}\,\|\boldsymbol{w}\|_2^2\,, \tag{8.4}$$

which implicitly assumes a Gaussian distribution of the weights in a Bayesian setting.

The motivation for this term as a complexity term is similar to early stopping (see later) with small weights. If a sigmoid activation function is used then small weights keep the activation in the linear range. Therefore small absolute weight values prefer linear solutions.

Interpretation of L2 weight decay If we neglect the first data-driven term in Eq.(8.4) and only consider the complexity term, the gradient would be $\lambda\boldsymbol{w}$ and with gradient descent the weights would change by $\boldsymbol{w}^{(t+1)} = \boldsymbol{w}_{(t)} - \eta\lambda\boldsymbol{w}_t$. This is a recursive definition of the weight update that also has an explicit definition:

$$\boldsymbol{w}^{(t)} = (1-\eta\lambda)^t\, \boldsymbol{w}^{(0)}, \tag{8.5}$$

where $\boldsymbol{w}^{(t)}$ are the weights at update step t and $\boldsymbol{w}^{(0)}$ are the initial weights. We observe that in this case, in which we have neglected the data-driven error term, the weights decay to zero over time if $0 < \eta\lambda < 1$.

We aim at getting some further insights on the dynamics introduced by the weight decay term by assuming a quadratic approximation of the error around a local minimum $\boldsymbol{w}_0$:

$$R(\boldsymbol{w}) = R(\boldsymbol{w}_0) + \frac{1}{2}(\boldsymbol{w}-\boldsymbol{w}_0)^T \boldsymbol{H}(\boldsymbol{w}-\boldsymbol{w}_0) \tag{8.6}$$

in which $\boldsymbol{H}$ is the Hessian of R at $\boldsymbol{w}_0$. The Hessian is positive-semidefinite and the linear term is missing as we are in a local minimum. The minimum of this approximation obviously occurs at $\boldsymbol{w} = \boldsymbol{w}_0$. If we now add an L2 regularization term, the minimum moves to a point $\tilde{\boldsymbol{w}}$, where:

$$\lambda\tilde{\boldsymbol{w}} + \boldsymbol{H}(\tilde{\boldsymbol{w}} - \boldsymbol{w}_0) = \boldsymbol{0}. \tag{8.7}$$

Solving for $\tilde{\boldsymbol{w}}$ gives

$$\tilde{\boldsymbol{w}} = (\boldsymbol{H} + \lambda \boldsymbol{I})^{-1} \boldsymbol{H} \boldsymbol{w}_0, \tag{8.8}$$

from which we see that as λ approaches zero, the regularized solution $\tilde{\boldsymbol{w}}$ approaches the unregularized solution $\boldsymbol{w}_0$.

In order to obtain more insight (Goodfellow et al., 2016; Bishop, 1995), we make use of the eigendecomposition of the Hessian $\boldsymbol{H} = \boldsymbol{Q} \boldsymbol{\Lambda} \boldsymbol{Q}^T$, in which $\boldsymbol{\Lambda}$ is a diagonal matrix containing the eigenvalues of $\boldsymbol{H}$ and $\boldsymbol{Q}$ containing its eigenvectors. Combining the eigen-decomposition with the equation above, we obtain:

$$\tilde{\boldsymbol{w}} = \boldsymbol{Q} (\boldsymbol{\Lambda} + \lambda \boldsymbol{I})^{-1} \boldsymbol{\Lambda} \boldsymbol{Q}^T \boldsymbol{w}_0, \tag{8.9}$$

from which we see that the effect of weight decay is to rescale along the directions of the eigenvectors. First $\boldsymbol{Q}^T$ projects $\boldsymbol{w}_0$ onto the eigenvectors. Then $(\boldsymbol{\Lambda} + \lambda \boldsymbol{I})^{-1} \boldsymbol{\Lambda}$, which is a diagonal matrix with $\frac{\lambda_i}{\lambda_i + \lambda}$ as entries[1] at index (i, i), scales the entries of $\boldsymbol{w}_0$ along the directions of the eigenvectors. Finally $\boldsymbol{Q}$ transforms this scaled vector back into the original parameter space.

L1 weight decay. Different weight decay terms were suggested like a term based on the 1-norm $\|\boldsymbol{w}\|_1$ (Hanson and Pratt, 1989). This can be motivated in a Bayesian setting, by a Laplace distribution of weights:

$$R_{\text{reg}}(\boldsymbol{w}) \;=\; R_{\text{emp}}(\boldsymbol{w}) \;+\; \lambda \, \|\boldsymbol{w}\|_1 \; . \tag{8.10}$$

Further variants. Williams (1994) suggest a term based on $\log \left(1 \;+\; \boldsymbol{w}^T \boldsymbol{w}\right)$, which arises from a Cauchy distribution of weights in the Bayes treatment.

All these mentioned complexity terms tend to replace large absolute weights through a couple of small absolute weights. For large networks also small absolute weights can lead to high non-linearities if the small weights accumulate to transfer a signal. Therefore, the complexity term was further developed and a threshold for large weights was introduced (Weigend et al., 1991). Only changes of weights near the threshold have large impact on the complexity term. All weights below the threshold are pushed towards zero and the weights beyond the threshold are not further penalized and can be even made larger in order to approximate the training data. The weight decay term according to (Weigend et al., 1991) is

$$\Omega(\boldsymbol{w}) \;=\; \sum_i \frac{w_i^2 / w_0^2}{1 \;+\; w_i^2 / w_0^2} \; , \tag{8.11}$$

where w_0 is the threshold value. For example $w_0 = 0.2$.

To have more control and to better adjust the hyper-parameter λ the gradient of $\Omega(\boldsymbol{w})$ can be normalized and multiplied by the length of the gradient $\nabla_{\boldsymbol{w}} R_{\text{emp}}(\boldsymbol{w})$ which amounts to a variable weighting of the complexity term.

[1] Note that the notation is slightly overloaded here: $\lambda_1, \ldots, \lambda_W$ are the eigenvalues of the Hessian, whereas λ is the hyperparameter that determines the strength of regularization.

Practical hints. For an MLP or a DNN, the weight decay terms can be separated for each layer of weights and individual hyper-parameters given for each layer. For example the input layer can be treated separately e.g. also for selecting or ranking input variables.

8.2 Neuron noise and weight noise

To regularize and to avoid that the network extracts individual characteristics of few training examples either noise can be added to the training examples or to networks components (Minai and Williams, 1994; Murray and Edwards, 1993; Neti et al., 1992; Matsuoka, 1992; Bishop, 1993; Kerlirzin and Vallet, 1993; Carter et al., 1990; Flower and Jabri, 1993).

For example Murray and Edwards (1993) inject white noise into the weights which results in a new complexity term. The complexity term consists of squared weight values and second order derivatives. However due to the second order derivatives the algorithm is very complex. It can be shown that noise injection is for small weight values equivalent to Tikhonov regularization. Tikhonov regularization penalizes large absolute higher order derivatives of the output with respect to the inputs and therefore highly nonlinear functions.

A recent regularization technique that can also be considered as regularization by noise is *Dropout*, which is treated in more detail below.

8.3 Weight Sharing

Nowlan and Hinton (Nowlan and Hinton, 1992) propose that the weights have special preferred values and should be grouped. Each group of weights should have similar values. Each group has a preferred value and a certain range which says what weights are considered as similar. Therefore each group j is modeled by a Gaussian

$$\mathcal{N}(w; \sigma_j, \mu_j) = \frac{1}{(2\pi)^{1/2}\,\sigma_j} \exp\left(-\frac{1}{2\sigma_j^2}(w-\mu_j)^2\right). \tag{8.12}$$

Each group j has a prior size $p(j) = \alpha_j$ which estimates what percentage of weights will belong to this group.

The likelihood of a weight value w under this mixture model is:

$$p(w) = \sum_j \alpha_j \mathcal{N}(w; \sigma_j, \mu_j). \tag{8.13}$$

We obtain for the conditional probability that a particular weight w was generated from a particular Gaussian j:

$$p(w \mid j) = \frac{\alpha_j \mathcal{N}(w; \sigma_j, \mu_j)}{\sum_k \alpha_k \mathcal{N}(w; \sigma_k, \mu_k)} \tag{8.14}$$

The value $p(w)$ should be maximized and we add its logarithm as complexity term:

$$\begin{aligned} R(\boldsymbol{w}) &= R_{\text{emp}}(\boldsymbol{w}) - \lambda \sum_i \log p(w_i) \\ &= R_{\text{emp}}(\boldsymbol{w}) - \lambda \sum_i \log \left(\sum_j \alpha_j \mathcal{N}(w_i; \sigma_j, \mu_j) \right) \end{aligned} \tag{8.15}$$

We obtain for the derivative of $R(\boldsymbol{w})$ with respect to a single weight

$$\frac{\partial R(\boldsymbol{w})}{\partial w_i} = \frac{\partial R_{\text{emp}}(\boldsymbol{w})}{\partial w_i} + \lambda \sum_j p(w_i \mid j) \frac{(w_i - \mu_j)}{\sigma_j^2}. \tag{8.16}$$

Also the centers μ_j and the variances σ_j^2 can be adaptively adjusted during learning.

Weight sharing is a central element of *Convolutional Neural Networks*, in which weights are shared spatially (see Chapter 6 for details), and *Recurrent Neural Networks*, in which weights are shared across time-steps.

8.4 Dropout

Dropout (Srivastava et al., 2014) is a technique similar to *neuron noise* or *weight noise*, but with some particularly interesting properties and that empirically strongly improves the generalization performance of a neural network. The main idea is to randomly set neuron activations to zero during training, which means that a large part of the information that the next layer expects is removed. This fits particularly well with rectified linear units (ReLU activations) because there is a default off-state of zero that is induced by the lower tail of the ReLU activations or by dropping a unit to zero. Randomly setting activations to zero is also equivalent to using a sub-network of the full network. The network without dropout can be then viewed as an ensemble of sub-networks. Furthermore, Gal and Ghahramani (2016) have shown that learning with dropout minimises the Kullback–Leibler divergence between an approximate distribution and the posterior of a deep Gaussian process.

A standard neural network performs the following transformation during training and inference:

$$\boldsymbol{a} = f(\boldsymbol{W}\boldsymbol{x}), \tag{8.17}$$

where $\boldsymbol{a}$ are the activations, $\boldsymbol{x}$ are the inputs to this layer and f is an activation function.

A neural network with dropout performs the following transformation during training:

$$\boldsymbol{a} = \boldsymbol{d} \odot f(\boldsymbol{W}\boldsymbol{x}), \tag{8.18}$$

where $\odot$ is element-wise multiplication. $\boldsymbol{d}$ is a binary vector called *dropout mask* and a realization of drawing from an I-dimensional Bernoulli random variable with probability p: $d_i \sim \mathcal{B}(1, p)$ for all i. The entries of the vector $\boldsymbol{d}$ are sampled independently from each other. p, the probability to keep the activation, can be considered as a hyperparameter of the training and is typically set to 0.5 for hidden units and 0.9 for input units.

One of the main problems that comes with Dropout is not the application of the backpropagation algorithm, which can be applied in its standard form noting that some activations are zero, but the form of the neural network function during inference. Dropout should only be applied during training, but at "test time" or for inference, the randomness in the network should be removed to obtain a deterministic forward pass undisturbed by noise. However, because a large part of the activations are dropped, the network input of the next layer $s^{[l+1]}$ are on average different than if they are not dropped:

$$\tilde{s} = \boldsymbol{w}^T(\boldsymbol{d} \odot \boldsymbol{a}) \qquad\qquad s = \boldsymbol{w}^T\boldsymbol{a}, \tag{8.19}$$

where $\tilde{s}$ is the preactivation of the next layer with dropout and s is the preactivation without dropout. In the equation above, we dropped the superscript indicating the layer which would be $s^{[l+1]}$ and $\boldsymbol{a}^{[l]}$.

We easily see that

$$\mathbb{E}_{p_{\text{data}}}(\tilde{s}) = \mathbb{E}_{p_{\text{data}}}\left(\sum_i w_i d_i a_i\right) = p\sum_i w_i\,\mathbb{E}_{p_{\text{data}}}(a_i). \tag{8.20}$$

which is in general not equal to $\mathbb{E}_{p_{\text{data}}}(s) = \sum_i w_i\,\mathbb{E}_{p_{\text{data}}}(a_i)$. Thus, there are at least three strategies to counter this bias:

- "Weight scaling": After training, we re-scale the weights by p, which removes the bias.
- "State scaling": During training, we re-scale the activations by $1/p$, which also counters the bias. The advantage is that the same network can be used during training and only the dropout function has two operation modes: rescaling and dropping during training and returning the identity during inference.
- "Monte-Carlo-Dropout": Applying standard dropout during training and performing multiple forward passes, typically hundreds or thousands, with dropout with subsequent averaging for inference. This still renders the inference non-deterministic, but has the advantage that it is a Bayesian approximation (Gal and Ghahramani, 2016).

For "weight scaling" and "state scaling", the aim is to keep the expected total input to a unit at test time to be roughly the same as the expected total input to that unit at train time.

Remarks. Note that in our derivations above, we have only considered the linear transformation and countered its effect in connection with dropout. However, for non-linearities, the approximation is inexact, but a nevertheless often works well empirically (Goodfellow et al., 2013).

Note that we also did not consider the second-moment nor the variance of the activations. By rescaling the activations, we only aimed at countering the bias of the expected value. This strategy would however increase the variance.

Dropout is a computationally cheap regularization technique. It only requires the additional time of sampling from a binary distribution and multiplying the activations with this vector. Depending on the implementation, it may also require to store the dropout mask in memory for

backpropagation. For inference, the same computational costs as a normal neural network are required, except for Monte-Carlo-Dropout, in which the computational costs are multiplied by the number of forward passes across which is averaged. Dropout is also quite flexible and can be included in almost all neural network architectures, such that practitioners simply speak of "adding a dropout layer".

8.5 Multi-task learning

The main idea of *multi-task learning* (Caruana, 1993) as regularization technique is to solve multiple prediction tasks using a single network with multiple output units.

> Multitask Learning is an approach to inductive transfer that improves generalization by using the domain information contained in the training signals of related tasks as an inductive bias. It does this by learning tasks in parallel while using a shared representation; what is learned for each task can help other tasks be learned better. (Caruana, 1997)

Multi-task learning requires that for each input object, multiple types of labels are available or can be acquired. The inductive transfer improves learning for one task by using the information contained in the training signals of other related tasks. This is done by learning tasks in parallel while using a shared representation of the data. What is learned for each task can help other tasks be learned better.

In a multi-task setting, each object $\boldsymbol{x}$ has multiple associated labels y^t for each task t instead of a single label y. A classical example is object recognition from images, because in an image multiple objects such as cars, persons, cats, landscape elements, etc, can occur at the same time. Another example could be in the healthcare context, where the disease status has to be predicted from an image, but it might be also helpful to predict other patient parameters, such as blood-pressure, from the image. The second, auxiliary task, could be helpful to learn a representation that is beneficial for the first task on disease status.

In a multi-task setting, we consider a parametric hypothesis class per task as $g(\boldsymbol{x}, \boldsymbol{w}^{\mathrm{sh}}, \boldsymbol{w}^{\mathrm{t}}) : \mathcal{X} \mapsto \mathcal{Y}^t$, where $\boldsymbol{w}^{\mathrm{sh}}$ are the parameters are shared between tasks and and $\boldsymbol{w}^{\mathrm{t}}$ are task-specific parameters. For each task, there can be a task-specific loss function $L^t(\cdot, \cdot)$. With the usual empirical risk minimization principle, the multi-task objective typically has the following form:

$$\min_{\boldsymbol{w}^{\mathrm{sh}}, \boldsymbol{w}^{1}, \ldots, \boldsymbol{w}^{\mathrm{t}}, \ldots, \boldsymbol{w}^{\mathrm{T}}} \quad \sum_{t=1}^{T} \alpha^t R^t_{\mathrm{emp}}(\boldsymbol{w}^{\mathrm{sh}}, \boldsymbol{w}^{\mathrm{t}}) \tag{8.21}$$

where we did not explicitly write the dependency of the task-specific empirical errors $R^t_{\mathrm{emp}}(\boldsymbol{w}^{\mathrm{sh}}, \boldsymbol{w}^{\mathrm{t}})$ on $\boldsymbol{X}$ and $\boldsymbol{y}^t$. The weights α^t can either be static or also dynamically adjusted.

Assuming that we are mainly interested in task $t = 1$, one can use all other tasks $t \geq 2$ as auxiliary tasks. These auxiliary tasks, act as a regularization for task $t = 1$ via their influence on $\boldsymbol{w}^{\mathrm{sh}}$. This effect has, for example, be used in the context of drug discovery by Mayr et al. (2016).

8.6 Growing

Growing algorithms start with small networks and add new units in a step-wise way or new weights which amount to increasing the complexity. The complexity of the neural network[2] is determined by the architecture, the number of units and weights. Thus, growing algorithms allow to dynamically change the complexity of the model class.

Growing algorithms are classified as regularization techniques that are applied after training because during training no regularization is done. Regularization is obtained through selecting a network from a hierarchy of networks with increasing complexity similar to early stopping.

Well known algorithms for classification are the *pocket* and the *tiling* algorithm. We will have a close look at a particular growing algorithm, called *Cascade correlation*.

8.6.1 Cascade correlation

The cascade correlation algorithm starts with a single layer network and then sequentially adds new hidden neurons. These new hidden neurons have incoming connections from all existing input neurons and hidden neurons. The new connections are adjusted so that the correlation between the new unit's activation and the output errors in maximal. Units that correlate with the error signal can potentially allow the whole network to decrease the overall error further.

In more detail, cascade-correlation works as follows: At the beginning, only direct connections from the input units to the output units exists and are adapted by any methods that are suited for the single layer problem (see Chapter 4). It is not necessary to assume a single layer network in principle, but this assumption is helpful for the further explanation. After having trained this initial network, single hidden neurons are added sequentially that have connections to all input and hidden units. If such a new hidden unit ξ is added then first its incoming weights are trained to maximize the correlation between the residual error and the unit's activation. Then these incoming weights are fixed and the output layer connections are retrained.

The objective to maximize is

$$C_\xi \;=\; \sum_{k=1}^{K} \left| \sum_{n=1}^{N} \left(a_\xi^n \;-\; \bar{a_\xi}\right) \;\left(\epsilon_k^n \;-\; \bar{\epsilon}_k\right) \right| \;, \tag{8.22}$$

where $\epsilon_k^n = (g(\boldsymbol{x}^n; \boldsymbol{w}) \;-\; y_k^n)$ is the whole network's error for sample $\boldsymbol{x}_n$ at the output unit k ($\bar{\epsilon}_k$ its mean value) and a_ξ^n is the activation of the new hidden unit ξ for sample n ($\bar{a_\xi}$ its mean value).

The new hidden unit ξ gets inputs from all input neurons and hidden neurons that currently exits:

$$s_\xi = \sum_{j=1}^{J} w_{\xi j} a_j, \tag{8.23}$$

where s_ξ is the pre-activation of the new unit ξ, a_j is the activation of unit j and $w_{\xi j}$ is the weight from unit j to ξ. Note that this sum runs over all input neurons and hidden neurons.

[2] With the formulation "complexity of the neural network" we mean "the complexity of the model class defined by the neural network"

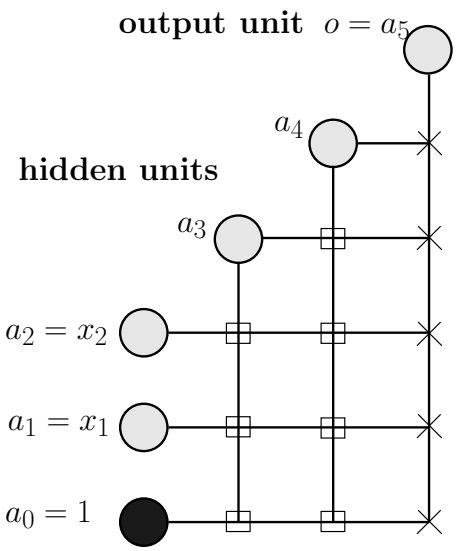


Figure 8.2: Cascade-correlation: architecture of the network. Squares are weights which are trained whereas crosses represent weights which retrained only after the addition of a hidden unit. First the hidden unit 3 with activation a_3 was added and then hidden unit 4 with activation a_4.

The derivative of C_ξ with respect to the weights $w_{\xi i}$ connecting to it, is just

$$\frac{\partial C_\xi}{\partial w_{\xi j}} = \pm \sum_{k=1}^{K} \sum_{n=1}^{N} (\epsilon_k^n - \bar{\epsilon}_k) \; f'(s_\xi) \, a_j^n \; , \tag{8.24}$$

where a_j^n is the activation of sample $\boldsymbol{x}^n$ at unit j and the sign is given by the sign of the correlation in eq. (8.22).

The training of cascade-correlation networks is very fast because only the incoming connections to the new hidden units have to be trained. The hidden to output weights are then found by a linear least-square estimate if the output unit(s) is (are) linear. Otherwise a single layer network has to be trained, which we have treated in chapter 4. Fig. 8.2 shows the architecture of the cascade-correlation network.

A disadvantage of growing or "constructive" algorithms is that units or weights are added only in small groups. Therefore, the combined effect of units cannot be used. If for example a reduction of the error can be done only through a couple of units then this would not be detected. It is unclear when the error starts to decrease therefore it is hard to decide when to stop or add a new unit.

8.7 Pruning methods

The opposite approach of growing is pruning. Pruning methods are typical methods for regularization after training. With *pruning* first a neural network is trained until a local minimum of the training error (empirical risk) is found. Now the complexity of the model class is reduced by removing weights (setting them to zero) from the trained network.

Since state-of-the-art neural networks, for example, for image recognition can have millions of weights, the computational effort to perform a forward pass is large. Pruning away weights, can

reduce both the size of a network and the required computations for inference. Therefore, this is relevant when neural networks are deployed on mobile devices or other devices with low storage or compute capacity. At the same time, a loss of predictive quality should be avoided.

Pruning methods differ in their way how and which weights are removed. Simple approaches start with removing small absolute weights. Those weights are removed which do not dramatically increase the training error to ensure that the major information extracted from the training data is still coded in the network.

Methods like in (White, 1989; Mozer and Smolensky, 1989; Levin et al., 1994) remove units and in (Moody, 1992; Refenes et al., 1994) even input units are removed. Before removing a unit or a weight its importance must be measured. The importance is determined by the influence or contribution of the unit to producing the output.

8.7.1 Pruning low-influence weights

To determine the increase of the error if a weight is deleted, a Taylor expansion of the empirical error $R(\boldsymbol{w})$ around the local minimum $\boldsymbol{w}^*$ is made. The gradient vanishes in the minimum $\nabla_{\boldsymbol{w}} R(\boldsymbol{w}^*) \;=\; \mathbf{0}$, therefore we obtain with $\Delta\boldsymbol{w} \;=\; (\boldsymbol{w} \;-\; \boldsymbol{w}^*)$

$$R(\boldsymbol{w}) \;=\; R(\boldsymbol{w}^*) \;+\; \frac{1}{2}\Delta\boldsymbol{w}^T \boldsymbol{H}(\boldsymbol{w}^*)\, \Delta\boldsymbol{w} \;+\; O\left((\Delta\boldsymbol{w})^3\right)\;, \tag{8.25}$$

where $\boldsymbol{H}$ is the Hessian. For small $|\Delta\boldsymbol{w}|$ the error increase $R(\boldsymbol{w}) \;-\; R(\boldsymbol{w}^*)$ for $\boldsymbol{w} \;=\; \boldsymbol{w}^* + \;\Delta\boldsymbol{w}$ is determined by $\Delta\boldsymbol{w}^T \boldsymbol{H}(\boldsymbol{w}^*)\, \Delta\boldsymbol{w}$. To delete the weight w_i we have to ensure

$$\boldsymbol{e}_i^T\, \Delta\boldsymbol{w} \;+\; w_i \;=\; 0\,, \tag{8.26}$$

where $\boldsymbol{e}_i$ is the unit vector with a one at the i-th position and otherwise zeros.

8.7.2 "Optimal Brain Damage" (OBD)

Optimal Brain Damage (LeCun et al., 1990) uses only the main diagonal of the Hessian which can be computed in $O(W)$ time as we will see later. The error increase is determined by

$$\sum_i H_{ii}\; (\Delta w_i)^2\;. \tag{8.27}$$

If we want to remove a weight then this term should be as small as possible. At the minimum the Hessian is positive definite, which means that for all $\boldsymbol{v}$

$$\boldsymbol{v}^T \boldsymbol{H}\; \boldsymbol{v} \;\geq\; 0\,, \tag{8.28}$$

therefore

$$\boldsymbol{e}_i^T \boldsymbol{H}\; \boldsymbol{e}_i \;=\; H_{ii} \;\geq\; 0\,. \tag{8.29}$$

Therefore the vector $\Delta\boldsymbol{w}$ is chosen as

$$\Delta\boldsymbol{w} \;=\; -\, w_i\, \boldsymbol{e}_i \tag{8.30}$$

so that all components except the i-th are zero. The minimal error increase is given by

$$\min_i H_{ii}\, w_i^2 \tag{8.31}$$

and the weight w_k with

$$k \;=\; \arg\min_i H_{ii}\, w_i^2 \tag{8.32}$$

is removed.

If the error increases after removal of some weights the network is retrained again until a local minimum is found.

OBD is not able to recognize redundant weights. For example if two weights perform the same tasks which can be done by one weight alone: assume each of the two weights has a value of 0.5 then this may be equivalent to set one of the weights to 1 and delete the other.

8.7.3 "Optimal Brain Surgeon" (OBS)

A method called "Optimal Brain Surgeon" (OBS) Hassibi and Stork (1993) which uses the complete Hessian was introduced by Hassibi & Stork. The full Hessian allows for correcting other weights which also allows to detect redundant weights and remove redundancies. Also retraining can be avoided or at least be reduced.

The Taylor expansion and the constraint of removing weight i can be stated as a quadratic optimization problem

$$\begin{aligned} \min_{\Delta \boldsymbol{w}} \quad & \frac{1}{2} \Delta \boldsymbol{w}^T \boldsymbol{H}\, \Delta \boldsymbol{w} \\ \text{s.t.} \quad & \boldsymbol{e}_i^T\, \Delta \boldsymbol{w} \;+\; w_i \;=\; 0 \end{aligned} \tag{8.33}$$

The Lagrangian [3]is

$$L \;=\; \frac{1}{2}\, \Delta \boldsymbol{w}^T \boldsymbol{H}\, \Delta \boldsymbol{w} \;+\; \alpha \left(\boldsymbol{e}_i^T\, \Delta \boldsymbol{w} \;+\; w_i \right) \;. \tag{8.34}$$

The derivative has to be zero

$$\frac{\partial L}{\partial \Delta \boldsymbol{w}} \;=\; \boldsymbol{H}\, \Delta \boldsymbol{w} \;+\; \alpha\, \boldsymbol{e}_i \;= \boldsymbol{0} \tag{8.35}$$

which is

$$\Delta \boldsymbol{w} \;=\; -\, \alpha\, \boldsymbol{H}^{-1}\, \boldsymbol{e}_i \tag{8.36}$$

Further we have

$$w_i \;=\; -\, \boldsymbol{e}_i^T\, \Delta \boldsymbol{w} \;=\; \alpha\, \boldsymbol{e}_i^T \boldsymbol{H}^{-1}\, \boldsymbol{e}_i \;=\; \alpha\, \boldsymbol{H}_{ii}^{-1} \tag{8.37}$$

[3]Note that in this case the letter L is used for the Lagragian and not for the loss function

which gives

$$\alpha \;=\; \frac{w_i}{\boldsymbol{H}_{ii}^{-1}} \tag{8.38}$$

and results in

$$\Delta \boldsymbol{w} \;=\; -\,\frac{w_i}{\boldsymbol{H}_{ii}^{-1}}\,\boldsymbol{H}^{-1}\,\boldsymbol{e}_i\;. \tag{8.39}$$

The objective is

$$\frac{1}{2}\,\Delta \boldsymbol{w}^T \boldsymbol{H}\,\Delta \boldsymbol{w} \;=\; \frac{1}{2}\frac{w_i^2}{\boldsymbol{H}_{ii}^{-1}}\;, \tag{8.40}$$

where one $\boldsymbol{H}^{-1}$ vanishes with $\boldsymbol{H}$ and the other has the form $\boldsymbol{e}_i^T \boldsymbol{H}^{-1}\,\boldsymbol{e}_i \;=\; \boldsymbol{H}_{ii}^{-1}$.

The criterion for selecting a weight to remove is the

$$\frac{1}{2}\frac{w_i^2}{\boldsymbol{H}_{ii}^{-1}} \tag{8.41}$$

and the correction is done by

$$\Delta \boldsymbol{w} \;=\; -\,\frac{w_i}{\boldsymbol{H}_{ii}^{-1}}\,\boldsymbol{H}^{-1}\,\boldsymbol{e}_i\;. \tag{8.42}$$

The expression $\frac{w_i^2}{\boldsymbol{H}_{ii}^{-1}}$ always larger or equal to zero (the proof is left as exercise). An approximation technique for the inverse Hessian will be given later.

The problem with OBS is that most weights may be larger than 1.0 and the Taylor expansion is not valid because higher terms in $\Delta \boldsymbol{w}$ do not vanish. However in practice OBS does also work in these cases because higher order derivatives are small.

Heuristics to improve OBS are

- checking for the first order derivative $\nabla_{\boldsymbol{w}} R(\boldsymbol{w}^*) \;=\; \boldsymbol{0}$; if the gradient is not zero then also the linear term can be used to determine the increase of the error
- retraining after weight deletion
- checking for $\left\|\boldsymbol{I} \;-\; \boldsymbol{H}^{-1}\,\boldsymbol{H}\right\| \;>\; \beta$ to see the quality of the approximation of the inverse Hessian.
- check for weight changes larger than γ (e.g. $\gamma = 0.5$), if changes appear which are larger than γ then the weight correction should be scaled.

In general, OBS will not find the smallest network because the gradient descent training was not forced to use the smallest architecture to solve the problem. Therefore some structures are distributed in a larger network which avoids to prune the network to an optimal size.

8.8 Early stopping

To control the complexity of the network [4] it is possible to stop learning before the minimum is reached. This regularization technique is called "*early stopping*" and belongs to regularization after learning. Early stopping can be seen as a training procedure which produces a sequences of networks with increasing complexity. Complexity is controlled by selecting one of these networks after learning.

During learning the network will first extract the most dominant rules, that is the most important structures in the data. Here "important" or "dominant" means that the rule can be applied to or structure is present at many training examples. These rules or structures can decrease the training error most efficiently because the loss decreases for many examples. Later in the learning procedure characteristics are found which apply only to a few or just one example. If the network is initialized with small weights and sigmoid activation functions are used, then at the beginning of learning the network implements an almost linear function. The initial network function is made more and more nonlinear during learning. For a highly nonlinear network function the weights in the network must be large. This means the earlier the learning is stopped the more linear the network function will be and the lower is the complexity. In Fig. 8.1 the relation between the test error and the training error is shown. If the complexity grows with learning time, then the test error first decreases and then increases again. Optimal would be to stop where the test error is minimal. To analytically determine the best stopping time is complicated. Only rough guesses can be given and do not help in practice. A practical way is to use a validation set besides the training set to determine the best stopping time. However the validation examples are lost as training examples. A disadvantage of early stopping is that there is no pressure on the learning algorithm to focus on the most prominent structure and efficiently use the resources. Also dominant but complicated structures are not found because to extract them would take more time.

8.9 Flat Minimum Search

Another algorithm for regularization during training is the "Flat Minimum Search" (FMS) algorithms (Hochreiter and Schmidhuber, 1997a). It searches for large regions in the weight space where the network function does not change but the empirical risk is small. Each parameter vector from this region leads to the same empirical risk. Such a region is called "flat minimum" (see Fig. 8.3).

A steep minimum corresponds to a weight vector which has to be given with high precision in order to ensure low empirical error. In contrast a flat minimum corresponds to a weight vector which can be given with low precision without influencing the empirical risk. Precision is here defined as how exact the weight vector is given in terms of intervals. For example 1.5 means the interval $[1.45, 1.55]$ of length 0.1 and 1.233 means the interval $[1.2325, 1.2335]$ of length 0.001, where the later is given more precisely than the former but needs more digits to describe. The FMS algorithm removes weights and units and reduces the sensitivity of the output with respect to the weights and other units. Therefore it can be viewed as an algorithm which enforces robustness.

From the point of view of "Minimum Message Length" (Wallace and Boulton, 1968) and "Minimum Description Length" (Rissanen, 1978) fewer bits are needed to describe a flat mini-

[4] More precisely: "the complexity of the model class defined by the network

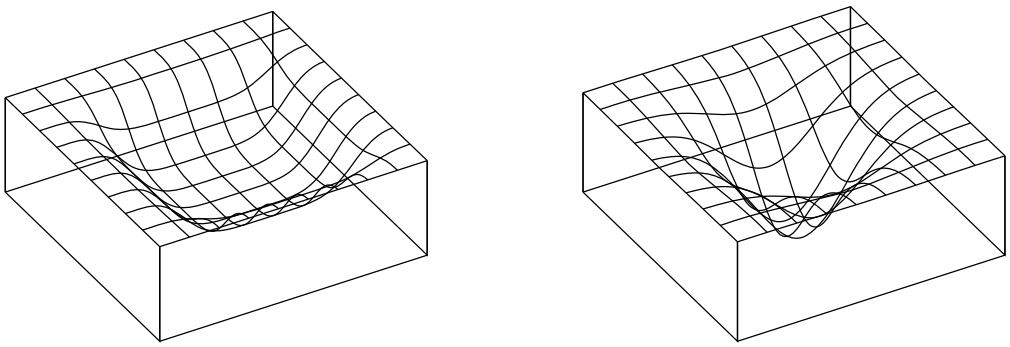


Figure 8.3: Left: example of a flat minimum. Right: example of a sharp minimum.

mum. That means a flat minimum corresponds to restricting the neural network model class to low complexity.

Flat minimum search adds an additional error term $\Omega(\boldsymbol{w})$ to the objective function $R(\boldsymbol{w})$. This error term $\Omega(\boldsymbol{w})$ describes the local flatness and has to be minimized. This error term consists of first order derivatives and can be minimized by gradient methods

The FMS error term is

$$\Omega(\boldsymbol{w}) \;= \frac{1}{2}\left(-W\log\epsilon + \sum_{i,j,l}\log\sum_{k=1}^{K}\left(\frac{\partial a_k}{\partial w_{ij}^{[l]}}\right)^2 + W\log\sum_{k=1}^{K}\left(\sum_{i,j,l}\frac{\left|\frac{\partial a_k}{\partial w_{ij}^{[l]}}\right|}{\sqrt{\sum_{k=1}^{K}(\frac{\partial a_k}{\partial w_{ij}^{[l]}})^2}}\right)^2\right), \tag{8.43}$$

where a_k are the activations of the output layer (the superscript $[L]$ was omitted), W is the number of weights, the sum $\sum_{i,j,l}$ runs across all weights in the network, and ϵ gives the tolerable output change as Euclidian distance (output changes below ϵ are considered to be equal).

The derivative of this error term with respect to a particular weight w_{uv} is

$$\frac{\partial\Omega(\boldsymbol{w})}{\partial w_{uv}} \;= \sum_{k=1}^{K}\sum_{i,j,l}\frac{\partial\Omega(\boldsymbol{w})}{\partial\left(\frac{\partial a_k}{\partial w_{ij}^{[l]}}\right)}\frac{\partial^2 a_k}{\partial w_{ij}^{[l]}\,\partial w_{uv}} \tag{8.44}$$

which is with $\frac{\partial a_k}{\partial w_{ij}^{[l]}}$ as new variables

$$\nabla_{\boldsymbol{w}}\Omega(\boldsymbol{w}) \;= \sum_{k=1}^{K}\boldsymbol{H}^k\left(\nabla_{\frac{\partial a_k}{\partial w_{ij}^{[l]}}}\Omega(\boldsymbol{w})\right), \tag{8.45}$$

where $\boldsymbol{H}^k$ is the Hessian of the output unit a_k.

The algorithm has complexity of $K(K\,W)$ (remember that K is the number of output units).

Similar to weight decay FMS prefers weights close to zero, especially outgoing weights from units which are not required (then the incoming weights can be given with low precision). In

contrast to weight decay, FMS also prefers large weight values. Especially large negative bias weights to push the activation of units either to zero (outgoing weights can be given with low precision.). Other large weights push activation into regions of saturation (incoming weights can be given with low precision).

8.10 Data Augmentation

Another strategy to prevent overfitting is to increase the size of the training set. This strategy is called *data augmentation* and is widely used across different application areas.

For data augmentation, domain-specific transformations are applied to the training set (Krizhevsky et al., 2012b; Perez and Wang, 2017). For imaging data, for example, such transformations include random cropping, mirroring, random perturbation of brightness, saturation, hue and contrast. In this work, we focus on data augmentation techniques for imaging data, however, some of the presented strategies or transformations could easily be transferred to other application domains.

The first question that arises is why data augmentation increases the generalization ability of the neural network model $g(\boldsymbol{x}; \boldsymbol{w})$. To investigate this we remind ourselves, that the parameters $\boldsymbol{w}$ are adjusted to minimize the empirical error

$$R_{\text{emp}}(\boldsymbol{w}) = \frac{1}{N} \sum_{n=1}^{N} L(y^n, g(\boldsymbol{x}^n; \boldsymbol{w})). \tag{8.46}$$

If we augment the data set, generating L additional examples with labels from our data set, we can adjust the parameters on more data:

$$R_{\text{reg}}(\boldsymbol{w}) = \frac{1}{N} \sum_{n=1}^{N} L(y^n, g(\boldsymbol{x}^n; \boldsymbol{w})) + \underbrace{\frac{1}{L} \sum_{l=1}^{L} L(y^l, g(\boldsymbol{x}^l; \boldsymbol{w}))}_{\text{error on augmented data}}. \tag{8.47}$$

However, despite these additional L data points, it is unclear whether the generalization performance will improve since $\frac{1}{L} \sum_{l=1}^{L} L(y^l, g(\boldsymbol{x}^l; \boldsymbol{w}))$ could also increase the noise that lead to parameters and models that generalize worse. Unfortunately, it has turned out that there is no analytical way to determine whether the augmentation strategy will improve the generalization ability. However, we can procure that the noise induced by $\frac{1}{L} \sum_{l=1}^{L} L(y^l, g(\boldsymbol{x}^l; \boldsymbol{w}))$ is small by using domain knowledge to ensure that the label of the augmented data point $\boldsymbol{x}^l$ is still correct. For standard object recognition tasks, this means that if we rotate or flip the image, the label should still be correct, e.g., a rotated object 'dog' should still be recognized as 'dog'. However, for object recognition tasks, in which letters should be identified, flipping could lead to incorrect labels, for example when a flipped letter 'b' could lead to a 'd' sign. In summary, data augmentation techniques have the potential to increase generalization ability of a model by adding to the empirical error, however, domain knowledge has to be used to design the augmentation strategy [5].

[5] Of course, one could also try to learn how to augment the data set in a way that increases generalization ability, see Cubuk et al. (2018)

The following data augmentation strategies are usually used for image recognition tasks (Krizhevsky et al., 2012b):

- Rotation: The image is rotated by a random angle.
- Translation: The image is shifted by a random amount of pixels.
- Cropping: The image is cropped, typically by a small amount
- Brightness noise: The brightness of each channel is changed by a random amount
- Shearing: the image is sheared by a random amount.

See Figure 8.4 for different transformations applied to the CIFAR-10 data set (see Section D.2).

8.11 Self-regularization by stochastic gradient descent

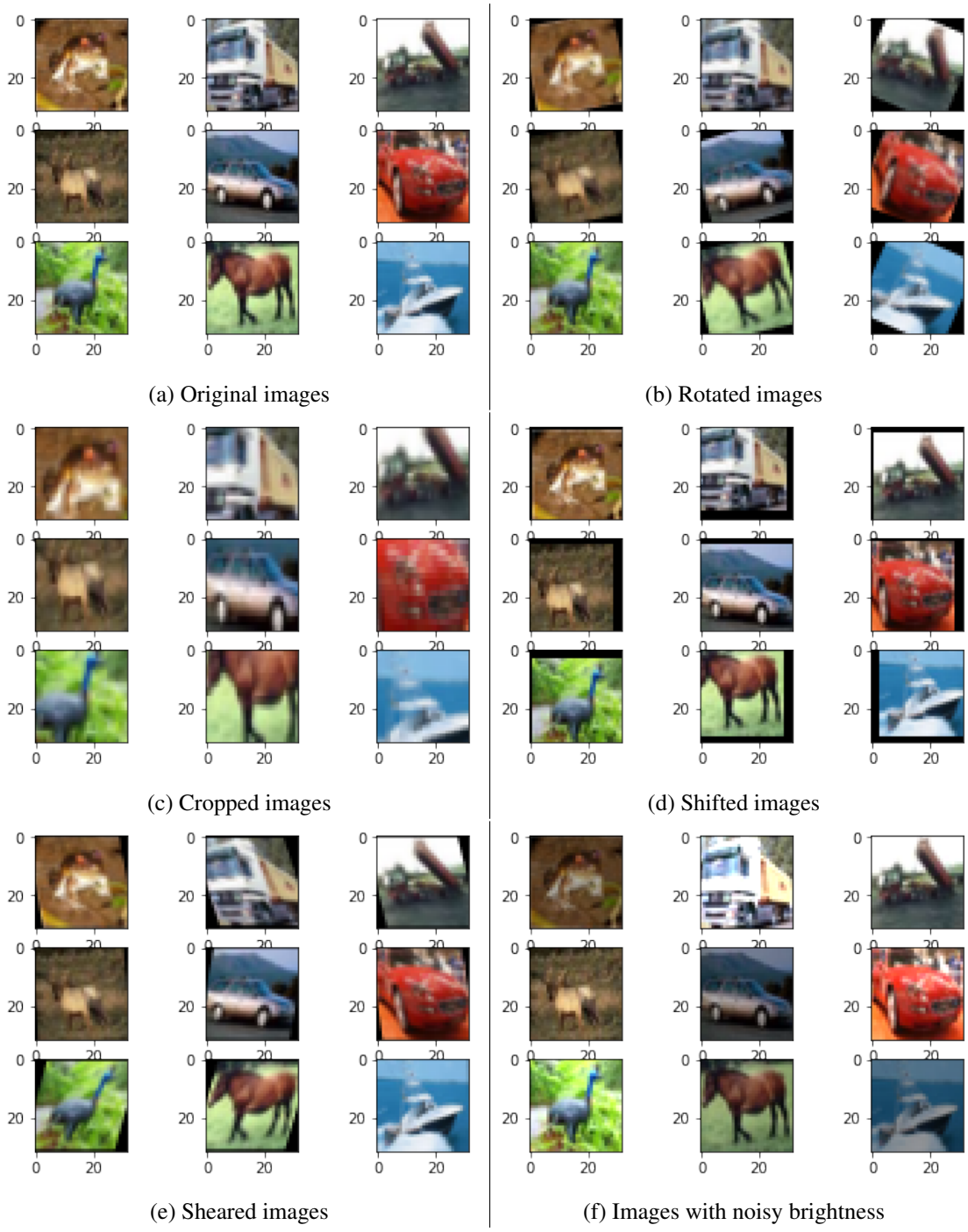
For many years, it had been observed empirically that *stochastic gradient descent (SGD)* (detailed in Section 7.3) works well for training neural networks. Despite the highly non-convex surface of the loss function, and the strong randomness of the optimization direction, SGD excels at finding weights that have a low empirical error and generalize well.

The manner in which the regularization happens has been investigated by (Chaudhari and Soatto, 2018). They prove that SGD minimizes an average potential over the posterior distribution of weights along with an entropic regularization term. To this end, SGD was considered as a continuous-time problem and as the behaviour of a particle under influence of drag forces and random forces, such as Brownian motion. The so-called *Fokker–Planck equations* describe the evolution of this system and can be used to provide an expression of the steady-state distribution. This steady-state distribution ρ^{ss} over the weights estimated by SGD is given by (assuming it exists):

$$\rho^{\text{ss}} = \underset{\rho}{\operatorname{argmin}}\, \mathbb{E}_{\boldsymbol{w}\sim\rho}\left(\Phi(\boldsymbol{w})\right) - \frac{\eta}{2b} H(\rho), \tag{8.48}$$

where $H(\rho)$ is the entropy of the distribution ρ, η is the learning rate and b is the batch size. The potential $\Phi(\boldsymbol{w})$ can be thought of as the original empirical error function $R(\boldsymbol{w})$ to be minimized, but is only equal to it iff the noise in the mini-batch gradients is isotropic. For details on the proof, see Chaudhari and Soatto (2018).

From this result, we can deduce several consequences for training neural networks with SGD. First and quite surprisingly, SGD does not find minima of the original objective function $R(w)$ via $\Phi(\boldsymbol{w})$, but locations that deviate from those points. Second, the deviation scales linearly with $\beta^{-1} = \eta/(2b)$, which can be large in practice. Third, large learning rates and small batch sizes have a regularization effect on learning. The quantity $\beta^{-1} = \frac{\eta}{2b}$ that determines the strength of the entropic regularization, increases with learning rate and decreases with batch size. Fourth, the learning rate should scale linearly with batch-size to generalize well. To maintain the entropic regularization effect, the learning rate should be increased when increasing batch size. Interestingly,



(a) Original images

(b) Rotated images

(c) Cropped images

(d) Shifted images

(e) Sheared images

(f) Images with noisy brightness

Figure 8.4: Images from the CIFAR-10 data set (see Section D.2). **Top left:** Original images. **Other cells:** Images with different transformations used for data augmentation. Source: `https://memoex.github.io/note/tech/image/augmentation/`

and in contrast to the typical practice of *sampling mini-batches without replacement*, this theoretical results imply that sampling with replacement should be preferred. For *sampling without replacement*, the parameter β^{-1} changes to $\beta'^{-1} = \frac{\eta}{2b}(1 - \frac{b}{N})$ which slightly decreases entropic regularization.

Chapter 9

Initialization

PIETER-JAN HOEDT AND GÜNTER KLAMBAUER

Most learning methods presented in Chapter 7 work by moving step-wise towards some point where the loss is (locally) minimal. One important choice that needs to be made in this iterative approach, is where to start the walk through parameter space. In an ideal case, the starting point $\boldsymbol{w}^{(0)}$ is (close to) the optimal point, so that only few steps are necessary to learn. On the other hand, obtaining the starting point should not be more complex than running the chosen optimisation method.

Since the parameters of deep neural networks consist of its weights, finding a good starting point comes down to finding matrices that make up good initial weights. There are several approaches to tackle the problem of finding these initial matrices. The most important of these initialization techniques together with the guiding principles that underlie these methods, will be discussed in this chapter.

9.1 Breaking Symmetry

One of the easiest ways to initialize weights would be to set all weights to some constant value. This approach might be very efficient computationally, but it also prevents the network from using its full capacity. The main problem with constant initialization is that this makes all neurons in one layer equivalent to each other, effectively making the network behave as if it had only one neuron in each layer. This equivalence of neurons is commonly referred to as *symmetry*.

9.1.1 Constant initialization

Obviously, all neurons in a layer will be symmetric when initializing all weights with some constant c. It might be less obvious, however, that this symmetry persists during learning when using back-propagation or similar gradient-based optimisation algorithms. In what follows, we will provide some insight as to why constant initialization should not be used for weight matrices and why it is still a reasonable choice for initializing biases.

Consider a fully connected layer in a network so that the expressions

$$s_i^{[l+1]} = \sum_{j=1}^{J} w_{ij} f(s_j^{[l]}) \qquad\qquad \delta_j^{[l]} = f'(s_j^{[l]}) \sum_{i=1}^{I} \delta_i^{[l+1]} w_{ij}$$

describe the forward and backward pass through this layer, respectively. With constant weights, $w_{ij} = c$, the pre-activations would be the same in every neuron:

$$s^{[l+1]} = \sum_{j=1}^{J} c\, f(s_j^{[l]}).$$

As a consequence, also the back-propagated deltas will be the same for each neuron, assuming constant weights in the previous layer:

$$\delta^{[l]} = f'(s^{[l]}) \sum_{i=1}^{I} c\delta_i^{[l]}.$$

Therefore, the gradients w.r.t. weights will be the same for all i and j:

$$\frac{\partial L}{\partial w_{ij}} = \delta_i^{[l+1]} f(s_j^{[l]}).$$

Note that in the first layer, the x_j's will generally not be constant. However, the gradient will still be the same for every single output neuron in the first layer. Similarly, it is unlikely that the δ_i's in the last layer would be the same — assuming that the network has more than one output unit. In this case, constant initialization of the final layer is actually possible, as long as the activations from the previous layer are not symmetric, i.e. if the weight matrix in the previous layer does not consist of constant entries.

9.1.2 Random initialization

Since it is not useful to use the same value for initializing weight matrices — see section 9.1, an alternative strategy is necessary to have neurons capture distinct patterns. An easy and relatively cheap way to break this symmetry is to randomly sample values from some reasonable distribution. What distribution to sample from is essentially just an extra hyperparameter that needs to be chosen. Most of this chapter will be dedicated to finding proper distributions to sample weights from.

Since most activation functions are centred around zero, weights are commonly initialized with values close to zero. Historically, this was even necessary to assure gradient flow when using sigmoidal functions. An easy way to get random values in a small range around zero, is to sample from a uniform distribution with proper bounds, i.e.

$$W_{ij} \sim \mathcal{U}(-\epsilon, \epsilon), \tag{9.1}$$

where ϵ is some small value that specifies the range of possible values. e.g. $\epsilon = 0.01$ (Hochreiter and Schmidhuber, 1997c). An alternative approach to get values close to zero, is to sample from a Gaussian or normal distribution with zero mean and some small variance, i.e.

$$W_{ij} \sim \mathcal{N}(0, \sigma^2). \tag{9.2}$$

With a Gaussian distribution, more values will be closer to zero than would be the case with the uniform distribution. On the other hand, it is also possible to get some arbitrarily large values

when sampling from a normal distribution. Since it is sometimes not desired to have large weights upon initialization, it is therefore not uncommon to cut off the tails to limit support. For sampling weights, the normal distribution is commonly truncated at two standard deviations from zero on both sides, i.e. values are in the range $[-2\sigma, 2\sigma]$. Therefore, we can write this truncated normal distribution in terms of the original variance of the normal distribution:

$$W_{ij} \sim \mathcal{N}_{\text{tr}}(0, \sigma^2). \tag{9.3}$$

Note that the variance of the truncated distribution is not the same as for the normal distribution, i.e. $\sigma^2 > \text{Var}[W_{ij}] \approx 0.8796\sigma^2$.

Not only does random initialization break symmetry, it also allows to explore larger parts of the loss surface. After all, each initialization will (very likely) result in a different starting point, making it possible to end up in a different local minimum for each run. This is a double-edged sword, because this also means that for some starting points, the performance might be worse than for others. In practice, this is not so much of a problem, however, since the combination of deep neural networks with big data is powerful enough to overcome these suboptimal starting positions. In some special cases, a bad starting point can effectively hinder learning, but when using pseudo-random generators, this problem can be alleviated by excluding bad seeds[1].

9.1.3 Bias weights

Although they are often ignored in analyses, *biases* can still be important for learning. Most noticeably, bias weights make it much easier to learn the mean signal. This allows the weights to focus on capturing more interesting information. Since most activation functions are centred around zero, it makes sense to initialize this bias to zero. In some cases, it can also be useful to set the initial biases directly to some value that corresponds to some known mean value. E.g. in a regression task, the targets often do not have a mean of zero and the biases could immediately be set to this non-zero mean to accelerate learning. This is constant initialization, but does not cause problems as long as symmetry is broken in some other way, e.g. by using random weight matrices.

Just as the constant initialization of biases relies on the symmetry to be broken by the weights, it is also possible to use constant initialization on the weights if the biases break symmetry. This means that it is possible to start learning something useful from a zero-initialized weight matrix as long as the biases are initialized randomly. This does not work so well in practice, however, since the biases will learn much faster and cause the network to ignore the input signal that has to pass through small weights.

9.2 Mean Field Theory

It is often interesting to study how a signal propagates through the network. In order for most analyses on this propagation to work, we need to assume that the inputs can be modelled by i.i.d. random variables. In practice, inputs will not be i.i.d., e.g. pixels in an image are generally highly correlated with neighbouring pixels. Nevertheless, it can be useful to study the averaged effect

[1] Also the pseudo-random number generator needs to be initialized with a seed. It can — although it should not — happen that some seeds make learning harder than other seeds.

of individual components in the often high dimensional inputs. This technique is known as *mean field theory* and has already proven useful in various branches of statistics.

9.2.1 Variance Propagation

In a fully connected network, each layer consists of a dot product of its weight matrix with the inputs to that layer. To analyse how the mean of the inputs propagates through the network, we consider the expectation of this dot product. Modelling the inputs as i.i.d. random variables, X_j, and assuming that the entries of the weight matrix, W_{ij}, were randomly sampled from some distribution with mean μ_w and variance σ_w^2, the expectation is given by:

$$\mathbb{E}_{W_{ij},X_j}\left[\sum_j W_{ij}X_j\right] = J\mu_w\,\mathbb{E}_{X_j}[X_j],$$

where J is the number of inputs or *fan-in*. Since weights are generally initialized from a distribution with $\mu_w = 0$, the result of these dot products will have zero mean.

More interesting results can be obtained when applying a similar analysis to the variance. Since the mean of the dot product is zero for $\mu_w = 0$, its variance is given by the second moment:

$$\mathbb{E}_{W_{ij},X_j}\left[\left(\sum_j W_{ij}X_j\right)^2\right] = J\sigma_w^2\,\mathbb{E}_{X_j}\left[X_j^2\right].$$

Unlike the mean, the variance of the input signal does propagate through the dot product. [2] The dot product does alter the variance, however, scaling it by a factor $J\sigma_w^2$.

This scaling factor can become a problem in deep neural networks, since it might lead to an exponential increase or decrease of the propagated variance. Concretely, if $J\sigma_w^2 < 1$ in each layer, the variance in the last layers will have been reduced to zero, whereas if $J\sigma_w^2 > 1$, variance in deep layers can grow extremely large. To assure a more stable propagation of variance, the easiest solution is to set the variance of the weights so that $J\sigma_w^2 = 1$. This can be done by setting the variance for the weight distribution so that:

$$\sigma_w^2 = \frac{1}{J}. \tag{9.4}$$

LeCun initialization For the purpose of initialization, this means that the weights in each layer, $\boldsymbol{W}$, should be sampled from a distribution with mean zero and a variance that corrects for the number of input neurons to that layer, J, e.g.

$$W_{ij} \sim \mathcal{N}(0, \frac{1}{J}) \qquad\qquad W_{ij} \sim \mathcal{U}(-\sqrt{\frac{3}{J}}, \sqrt{\frac{3}{J}}). \tag{9.5}$$

This initialization strategy is often referred to as *LeCun initialization* (LeCun et al., 1998b).

[2] To be exact, it is actually the second raw moment, $\mathbb{E}_X\left[X\right] = \sigma_x^2 + \mu_x^2$ of the inputs.

9.2.2 Error Propagation

The analysis of the variance propagation (section 9.2.1) can also be applied to the error that propagates backwards through the network. To make this work, we again use mean field theory to model the deltas, $\boldsymbol{\delta} = \frac{\partial L(\hat{\boldsymbol{y}}, \boldsymbol{y})}{\partial \boldsymbol{s}}$, as i.i.d. random variables D_i. Using the backprop recurrence from equation (4.71), the back-propagation of the first two moments through a (linear) layer is specified by

$$\mathbb{E}_{W_{ij}, D_i}\left[\sum_i D_i W_{ij}\right] = I\mu_w \,\mathbb{E}\left[D_i\right] = 0$$
$$\mathbb{E}_{W_{ij}, D_i}\left[\left(\sum_i D_i W_{ij}\right)^2\right] = I\sigma_w^2 \,\mathbb{E}_{D_i}\left[D_i^2\right],$$

where we again take $\mu_w = 0$ and I is the number of outputs or *fan-out* of the layer.

Note that these expressions imply that the deltas are independent from the weights. When using random weights for the back-propagation (Lillicrap et al., 2016), the deltas are indeed completely independent from the weights. For regular networks, however, this is a rather strong assumption, since $\boldsymbol{x} \times \boldsymbol{\delta}$ are the derivatives of the loss function with respect to these weights. By using the mean field approximation on the deltas, however, this direct dependency is practically modelled away because we assume the deltas to be completely random. Although this is theoretically not entirely satisfying, it turns out to work well enough in practice.

Similar to analysis of the forward propagation, zero-mean weights will keep the mean from propagating, whereas the variance is scaled by a factor $I\sigma_w^2$, which can lead to vanishing or exploding gradients. To counter this behaviour, the weights can be initialized so that the scale factor is one, i.e.

$$\sigma_w^2 = \frac{1}{I}. \tag{9.6}$$

When the number of input and output neurons are the same in a layer, i.e. when $J = I$, this would be equivalent to what we found for the forward propagation (compare to equation (9.6)). Otherwise, equations (9.4) and (9.6) are mutually exclusive. This means that we have to choose between stable forward with unstable error propagation or vice versa.

Glorot initialization In order not to have to sacrifice either one of the forward or backward propagation, it is possible to combine equations (9.4) and (9.6) to get the best of both worlds. This is why Glorot and Bengio (2010) proposed to take the average number of neurons in each layer for initialization. This means that initial weights for a layer are sampled from distributions like

$$W_{ij} \sim \mathcal{N}(0, \frac{2}{J+I}) \qquad\qquad W_{ij} \sim \mathcal{U}(-\sqrt{\frac{6}{J+I}}, \sqrt{\frac{6}{J+I}}), \tag{9.7}$$

where J and I are respectively the number of input and output units to that layer. This initialization strategy is commonly referred to as *Glorot initialization* or *Xavier initialization* (Glorot and Bengio, 2010).

9.3 Nonlinearities

By considering the variance propagation through dot products, it is possible to initialize networks consisting of linear layers in a reasonably justified manner. Unfortunately, linear neural networks are practically irrelevant. For practically useful analyses, it is obligatory to also consider how the activation function affects the propagation of variance through the network.

9.3.1 Propagation Function

When introducing activation functions to a layer, there are essentially two places where the propagation could be studied: before or after applying the nonlinearity. Because the expressions for mean and variance of the activations are rather clumsy, we will focus on how the activation functions affect the distributions of the pre-activations. This means that we will be studying the expectations of the pre-activations, given the pre-activations in the previous layer:

$$\mathbb{E}_{W_{ij},S_j\sim\mathcal{N}(\mu_s,\sigma_s^2)}\left[\sum_j W_{ij}f(S_j)\right] = J\mu_w\,\mathbb{E}_{S_j\sim\mathcal{N}(\mu_s,\sigma_s^2)\sim}\left[f(S_j)\right] = 0 \tag{9.8}$$

$$\mathbb{E}_{W_{ij},S_j\sim\mathcal{N}(\mu_s,\sigma_s^2)}\left[\left(\sum_j W_{ij}f(S_j)\right)^2\right] = J\sigma_w^2\,\mathbb{E}_{S_j\sim\mathcal{N}(\mu_s,\sigma_s^2)}\left[f(S_j)^2\right], \tag{9.9}$$

where we model the pre-activations by random variables S_j and assume $\mu_w = 0$.

Pre-activations are essentially weighted sums of the inputs. Given that the number of inputs is large enough, i.e. the layers in the network are wide enough, the central limit theorem applies and the pre-activations will be normally distributed. Moreover, we are able to write down the mean and variance of this distribution (equations (9.8) and (9.9)), so that the pre-activations in layer l are distributed according to $S_i \sim \mathcal{N}(0, \sigma_s^2)$, where

$$\sigma_s^2 = J\sigma_w^2 \begin{cases} \mathbb{E}_{X_j}\left[X_j^2\right] & l = 1 \\ \mathbb{E}_{S_j\sim\mathcal{N}(0,\sigma_s^2)}\left[f(S_j)^2\right] & l > 1 \end{cases}. \tag{9.10}$$

Knowing that the pre-activations are (approximately) normally distributed, is especially interesting because it allows to write down more precise expressions for the moments:

$$\mathbb{E}_{S_j\sim\mathcal{N}(0,\sigma_s^2)}[f(S_j)^n] = \mathbb{E}_{Z\sim\mathcal{N}(0,1)}[f(\sigma_s Z)^n] = \frac{1}{\sqrt{2\pi\sigma_s^2}}\int_{\mathbb{R}} f(x)^n e^{-\frac{x}{2\sigma_s^2}}\,\mathrm{d}x,$$

where $Z \sim \mathcal{N}(0,1)$. Assuming that this integral converges, this expression specifies how to get the second moment of the activations, given the variance of the pre-activations for some activation function f. There are a few nonlinearities for which this integral can be computed analytically, but being able to compute it numerically, will also prove useful.

In order to ease the discussion on propagation analysis, it is useful to introduce a function that gives the variance of the pre-activations as a function of the variance of the pre-activations in the previous layer. We will call this function the *propagation function*, which can be defined as follows,

$$F_f : \mathbb{R}^+ \to \mathbb{R}^+ : q \mapsto F_f(q) = J\sigma_w^2\,\mathbb{E}_{Z\sim\mathcal{N}(0,1)}\left[f(\sqrt{q}Z)^2\right]. \tag{9.11}$$

9.3.2 Gain Factor

Second to the identity function, the ReLU activation function is often the easiest activation function to analyse analytically. Given $S \sim \mathcal{N}(0, \sigma_s^2)$, we find for the second moment

$$\mathbb{E}[\mathrm{ReLU}(S)^2] = \mathbb{E}_{S<0}[0] + \mathbb{E}_{S\geq 0}[S^2] = \frac{1}{2}\sigma_s^2.$$

This means that each layer with ReLU activations scales the variance with a factor $\frac{1}{2}$, so that

$$F_{\mathrm{ReLU}}(q) = J\sigma_w^2 \frac{1}{2} q.$$

He initialization In order to keep the variance propagation from fading out or exploding in deep networks due to the effects of the ReLU activation function, we need to account for this scaling. This can easily be done by rescaling the variance of the weights, so that $J\sigma_w^2 \frac{1}{2} = 1$, i.e.

$$\sigma_w^2 = \frac{2}{J}.$$

The resulting initialization for a layer with J input neurons,

$$W_{ij} \sim \mathcal{N}(0, \frac{2}{J}) \qquad\qquad W_{ij} \sim \mathcal{U}(-\sqrt{\frac{6}{J}}, \sqrt{\frac{6}{J}}), \tag{9.12}$$

is known as *He initialization*, *Kaiming initialization* and is sometimes even referred to as *Microsoft initialization* (He et al., 2015). Note that this initialization strategy ignores the fact that the first layer should actually be initialized differently, cf. equation (9.10).

Following Glorot and Bengio (2010), $\tanh$ can be approximated by a linear function in a narrow interval around zero. Since we generally take weights with zero mean and small variance in (wide) linear networks (see section 9.2.1), this approximation might indeed be good enough. Using this approximation, we find

$$\mathbb{E}_{S\sim\mathcal{N}(0,\sigma_s^2)}[f(S)^2] \approx \mathbb{E}_{S\sim\mathcal{N}(0,\sigma_s^2)}[S^2] = \sigma_s^2.$$

This initialization strategy for $\tanh$ networks comes down to *LeCun initialization*, cf. Eq. (9.5) — or *Glorot initialization*, cf. Eq. (9.7), when backward dynamics are considered as well.

It turns out that for quite some activation functions, the effects on the variance propagation can be modelled by a positive scaling factor, $g_f \in \mathbb{R}^+$, so that

$$F_f(\sigma_s^2) = J\sigma_w^2\, \mathbb{E}[f(S)^2] \approx J\sigma_w^2 \frac{1}{g_f}\sigma_s^2.$$

As a result, the initial weights are sampled from a distribution with a variance

$$\sigma_w^2 = \frac{g_f}{J} \qquad \text{or} \qquad \sigma_w^2 = g_f \frac{2}{I+J}. \tag{9.13}$$

The scalar factor, g_f, that counters the effects of the activation function in the variance for the initial weights, will be referred to as the *gain factor* for the initialization (Saxe et al., 2014), or *gain* for short. E.g. He initialization uses $g_{\mathrm{ReLU}} = 2$ and Glorot initialization has a gain of $g_{\tanh} = 1$.

Chapter 10

Normalization

GÜNTER KLAMBAUER

Training Deep Neural Networks is complicated by the fact that the distribution of each layer's inputs changes during training, as the parameters of the previous layers change (see Fig. 10.1). This effect can become even more pronounced with an increasing number of layers. The absolute value of the activations $\boldsymbol{a}$ impacts the training of a neural network because the weight update is

$$\Delta w_{ij} = -\eta a_i \delta_j, \tag{10.1}$$

in which we see that the activation a_i determines the size (and direction) of the update. If the distribution of activations in the network's layers is strongly different, this can lead to unstable learning (high absolute values of updates) or almost no learning (small absolute values of updates). This phenomenon is linked to the so-called *vanishing gradient problem*, which is treated in more detail in a later chapter.

To counter the effect of vanishing and exploding activations and to robustly train very deep CNNs, different so-called *normalization* methods have been suggested that aim at keeping the distribution of the activations stable, for example at zero mean and unit variance. One method that has become a standard is batch normalization (Ioffe and Szegedy, 2015) which learns to standardize each unit.Furthermore, layer normalization (Ba et al., 2016) also ensures zero mean and unit variance in a layer, while weight normalization (Salimans and Kingma, 2016) ensures zero mean and unit variance if in the previous layer the activations have zero mean and unit variance. A method that avoids to externally interfere with the network, but rather provides a network with a stable fixed point at zero mean and unit variance is *self-normalization*. These four normalization methods are discussed in more detail in the following.

10.1 Batch normalization

The batch normalization (Ioffe and Szegedy, 2015) strategy is to learn the average activation and variance of a neuron in the training set and to use this information to standardize the neuron activations.

Assume we have a certain neuron whose activation is a. Batch normalization will transform this activation by

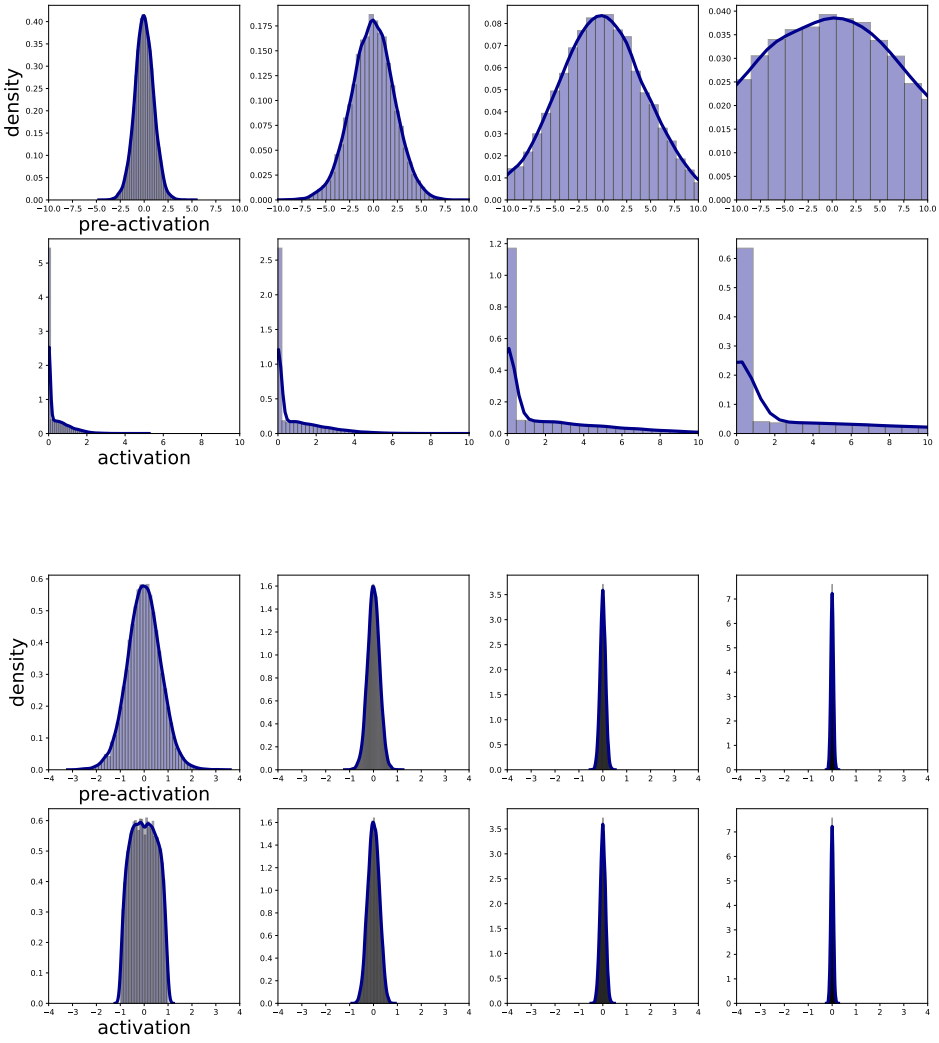


Figure 10.1: Vanishing and exploding activations through multiple layers of a neural network. The distribution of activations and pre-activations in a five layer feed-forward neural network trained on MNIST (see Section D.1) are shown. Each layer has 512 hidden units. The output-layer activations are not shown. **Top:** This network uses ReLU activations. From left to right layer 1 to layer 4 are displayed. Both the absolute values of the pre-activations and the activations increase over layers. **Bottom:** This network uses tanh activations. From left to right layer 1 to layer 4 are displayed. Both the absolute values of the pre-activations and the activations decrease over layers.

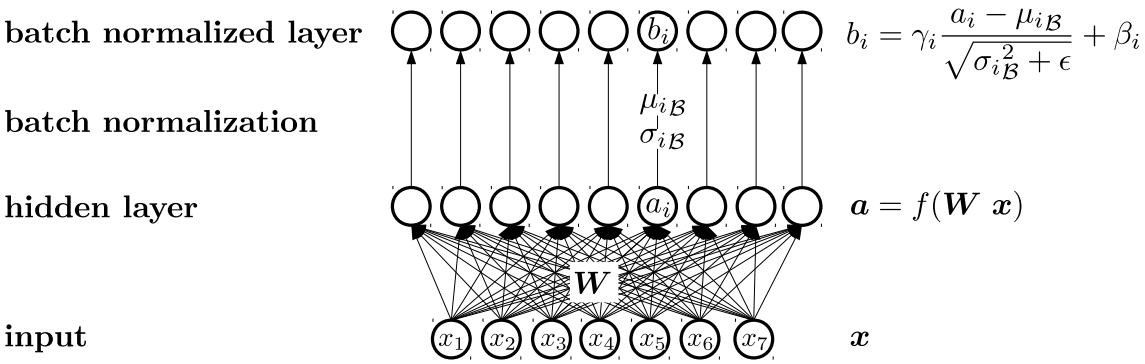


Figure 10.2: Schematic representation of batch normalization. Each unit a_i learns a batch mean $\mu_{i\mathcal{B}}$ and standard deviation $\sigma_{i\mathcal{B}}$ by which the inputs to the batch normalization layer are standardized.

$$\hat{a} = \frac{a - \mathbb{E}[a]}{\sqrt{\mathrm{Var}[a]}}, \tag{10.2}$$

where the expectation and the variance are computed over the training data set. This might seem a simple standardization, but the expectation and the variance of the activations of this particular neuron will change during learning, such that estimators of these two expressions have to be continuously updated.

Batch normalization also introduces two learnable parameters β and γ for each neuron, which can scale and shift the normalized activation

$$b = \gamma \hat{a} + \beta. \tag{10.3}$$

These parameters could also learn to keep the original activations ($\gamma = \sqrt{\mathrm{Var}[a]}$ and $\beta = \mathbb{E}[a]$). In case of full-batch training the mean and the variance across the whole training set could be used to standardize the activations. However, this is impractical when used together with stochastic gradient descent, i.e. mini-batch training. Thus, batch normalization uses an estimate of $\mathbb{E}[a]$ and $\mathrm{Var}[a]$ based on a mini-batch $\mathcal{B}$ with $|\mathcal{B}| = b$. We denote the activations of our particular neuron in the minibatch [1] as $a^{(1)}, \ldots, a^{(B)}$ and formulate the *batch normalizing transformation*:

[1] Here we use the additional brackets in the superscript to avoid confusion with the power operation. This is consistent with the notation of the n-th sample $\boldsymbol{x}^n = \boldsymbol{x}^{(n)}$

$$\mu_{\mathcal{B}} = \frac{1}{B}\sum_{k=1}^{B} a^{(k)} \tag{10.4}$$

$$\sigma^2_{\mathcal{B}} = \frac{1}{B}\sum_{k=1}^{B} (a^{(k)} - \mu_{\mathcal{B}})^2 \tag{10.5}$$

$$\hat{a}^{(\xi)} = \frac{a^{(\xi)} - \mu_{\mathcal{B}}}{\sqrt{\sigma^2_{\mathcal{B}} + \epsilon}} \tag{10.6}$$

$$b^{(\xi)} = \gamma \hat{a}^{(\xi)} + \beta, \tag{10.7}$$

for any $1 \leq \xi \leq B$ and where ϵ is a constant that added to the mini-batch variance to provide numerical stability. We denote the whole batch normalizing transform as

$$b = \mathrm{BN}_{\beta,\gamma}(a). \tag{10.8}$$

A fully-connected neural network layer with subsequent batch normalization is depicted in Figure 10.2.

Properties: invariance to weight scaling in linear and ReLU layers. One interesting property of batch normalization is that it renders a network invariant to the scaling of the weights of a feed-forward layer with linear or ReLU activations.

Let us assume we have a neuron $a^{(n)} = \mathrm{ReLU}(\boldsymbol{w}^T \boldsymbol{x}^n)$ where $\boldsymbol{x}^n$ can either be the n-th input object or the activations for the n-th sample from the lower layer. We now find that the mean $\frac{1}{n}\sum_{n=1}^{N} a^{(n)}$ scales linearly with the scaling the weights by a constant $c > 0$:

$$\frac{1}{n}\sum_{n=1}^{N} \mathrm{ReLU}(c\boldsymbol{w}^T \boldsymbol{x}^n) = c\frac{1}{n}\sum_{n=1}^{N} \mathrm{ReLU}(\boldsymbol{w}^T \boldsymbol{x}^n), \tag{10.9}$$

and the second moment scales with c^2:

$$\frac{1}{n}\sum_{n=1}^{N} \mathrm{ReLU}(c\boldsymbol{w}^T \boldsymbol{x}^n)^2 = c^2\frac{1}{n}\sum_{n=1}^{N} \mathrm{ReLU}(\boldsymbol{w}^T \boldsymbol{x}^n)^2. \tag{10.10}$$

We now use that to show that the batchnorm transform Eq. (10.4) does not change if the weights are scaled (and assuming $\epsilon = 0$). We use the variables $\mu^s_{\mathcal{B}}$, ${\sigma^s_{\mathcal{B}}}^2$, and $\hat{a}^s$ with a superscript s to denote the quantities with scaled weights. Thus, our aim is to show that $\hat{a}^s = \hat{a}$, which means that the batch normalized activation after scaling the weights is equal to the normalized activation with non-scaled weights:

$$\begin{aligned}
\mu_{\mathcal{B}}^s &= \frac{1}{B}\sum_{k=1}^{B}\mathrm{ReLU}(c\boldsymbol{w}^T\boldsymbol{x}^k) = c\,\mu_{\mathcal{B}} \\
{\sigma_{\mathcal{B}}^s}^2 &= \frac{1}{B}\sum_{k=1}^{B}(\mathrm{ReLU}(c\boldsymbol{w}^T\boldsymbol{x}^k) - \mu_{\mathcal{B}}^s)^2 = c^2\,\sigma_{\mathcal{B}}^2 \\
\hat{a}^s &= \frac{\mathrm{ReLU}(c\boldsymbol{w}^T\boldsymbol{x}) - c\mu_{\mathcal{B}}}{\sqrt{c^2\sigma_{\mathcal{B}}^2}} = \frac{\mathrm{ReLU}(\boldsymbol{w}^T\boldsymbol{x}) - \mu_{\mathcal{B}}}{\sqrt{\sigma_{\mathcal{B}}^2}} = \hat{a},
\end{aligned} \tag{10.11}$$

where the last line shows the invariance property. Thus batch normalization renders a feed-forward layer with ReLU activations invariant to weight scaling. The case of linear activations can be shown in a similar and easier manner.

Backpropagation for batch normalization. Similar to before, we assume that a neural network $g(\boldsymbol{x};\boldsymbol{w})$ with parameters $\boldsymbol{w}$ uses the batch normalization transform in its forward pass, i.e., the function g includes $\mathrm{BN}_{\beta,\gamma}$. Furthermore, we assume to have differentiable loss function $L(y, g(\boldsymbol{x};\boldsymbol{w}))$, then we have

$$\begin{aligned}
\frac{\partial L}{\partial \hat{a}^{(k)}} &= \underbrace{\frac{\partial L}{\partial b^{(k)}}}_{:=\delta_{b^{(k)}}}\gamma \\
\frac{\partial L}{\partial \sigma_{\mathcal{B}}^2} &= \sum_{k=1}^{B}\frac{\partial L}{\partial \hat{a}^{(k)}}(a^{(k)} - \mu_{\mathcal{B}})\frac{-1}{2}(\sigma_{\mathcal{B}}^2 + \epsilon)^{-3/2} \\
\frac{\partial L}{\partial \mu_{\mathcal{B}}} &= \left(\sum_{k=1}^{B}\frac{\partial L}{\partial \hat{a}^{(k)}}\frac{-1}{\sqrt{\sigma_{\mathcal{B}}^2 + \epsilon}}\right) + \frac{\partial L}{\partial \sigma_{\mathcal{B}}^2}\frac{\sum_{k=1}^{B} -2(a^{(k)} - \mu_{\mathcal{B}})}{B} \\
\frac{\partial L}{\partial a^{(k)}} &= \frac{\partial L}{\partial \hat{a}^{(k)}}\frac{1}{\sqrt{\sigma_{\mathcal{B}}^2 + \epsilon}} + \frac{\partial L}{\partial \sigma_{\mathcal{B}}^2}\frac{2(a^{(k)} - \mu_{\mathcal{B}})}{B} + \frac{\partial L}{\partial \mu_{\mathcal{B}}}\frac{1}{B} \\
\frac{\partial L}{\partial \gamma} &= \sum_{k=1}^{B}\frac{\partial L}{\partial b^{(k)}}\hat{a}^{(k)} \\
\frac{\partial L}{\partial \beta} &= \sum_{k=1}^{B}\frac{\partial L}{\partial b^{(k)}},
\end{aligned} \tag{10.12}$$

where $\delta_{b^{(k)}}$ is the delta error at the unit $b^{(k)}$ received from the upper layer, $a^{(k)}$ is the activation of the k-the sample in the mini-batch before batch normalization, and $\hat{a}^{(k)}$ after standardization an $\hat{b}^{(k)}$ is the batch normalized activation. Note that during training, only the parameters β and γ are updated with a gradient descent step. The expression $\frac{\partial L}{\partial a^{(k)}}$ is needed to provide delta-errors for the lower layer. The parameters $\mu_{\mathcal{B}}$ and $\sigma_{\mathcal{B}}^2$ are updated on-the-fly for each minibatch. For inference, where a single sample has to be propagated through, an exponentially decaying average over the last estimates of $\mu_{\mathcal{B}}$ and $\sigma_{\mathcal{B}}^2$ is used.

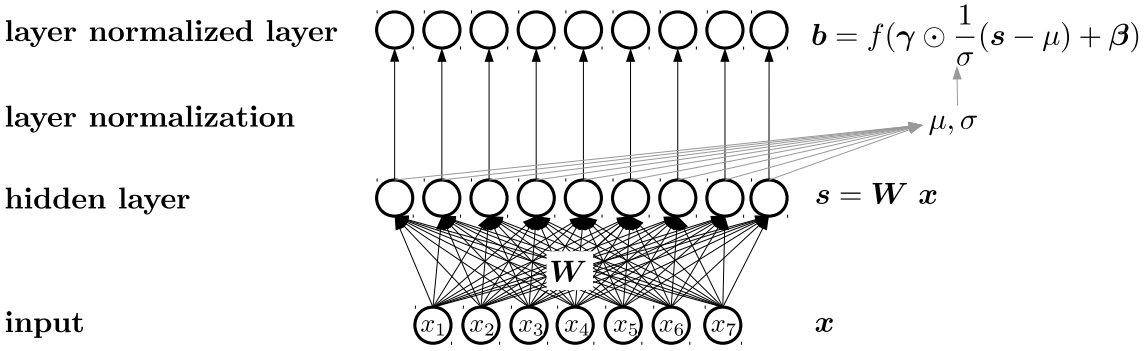


Figure 10.3: Schematic representation of layer normalization. Each layer learns a batch mean activation μ and standard deviation σ by which the inputs to the layer normalization layer are standardized.

10.2 Layer normalization

Batch normalization has the drawback that for each neuron a mean μ and a standard deviation σ have to be estimated using a mini-batch. This means that a batch size of $b = 1$ cannot be used and small batches could introduce high variance of the estimators. Therefore Ba et al. (2016) suggested layer normalization, in which all neurons in one layer share the same parameters μ and σ. Furthermore, layer normalization works on the output of a linear transformation (the pre-activations), and applies a non-linear activation after normalization. We assume that the inputs to the layer normalization layer are pre-activations $\boldsymbol{s}$, a vector of length J. Then the *layer normalization* works on those pre-activations in a particular layer as follows:

$$\mu = \frac{1}{J} \sum_{j=1}^{J} s_j \tag{10.13}$$

$$\sigma^2 = \frac{1}{J} \sum_{j=1}^{J} (s_j - \mu)^2 \tag{10.14}$$

$$\boldsymbol{b} = f(\boldsymbol{\gamma} \odot \frac{1}{\sigma}(\boldsymbol{s} - \mu) + \boldsymbol{\beta}), \tag{10.15}$$

where μ and σ are the mean and standard deviation of the respective layer, $\boldsymbol{\gamma}$ and $\boldsymbol{\beta}$ are learnable parameters equivalent to those in batch normalization.

Layer normalization also hat the property of invariance to scaling the weight matrix and – in contrast to batch normalization – is invariant to shifting the weight matrix.

10.3 Weight normalization

Weight normalization (Salimans and Kingma, 2016) tackles the problem with the distribution shifts in different way. Again, we assume that a neuron in our network has the activation $a =$

$\mathrm{ReLU}(\boldsymbol{w}^T\boldsymbol{x})$. The main idea is to reparameterize the weights in terms of a new weight vector $\boldsymbol{v}$ and a parameter g, which is called *weight normalization*:

$$\boldsymbol{w} = \frac{g}{\|\boldsymbol{v}\|}\boldsymbol{v}, \tag{10.16}$$

where the vector $\boldsymbol{v}$ has the same dimension as $\boldsymbol{w}$ and is divided by its Euclidean norm. We further assume that there is a loss function $L(\boldsymbol{y}, g(\boldsymbol{x}, \boldsymbol{w}))$ which depends on $\boldsymbol{w}$, such that we can have a gradient with respect to the parameters $\boldsymbol{w}$: $\nabla_{\boldsymbol{w}} L$, then:

$$\begin{aligned}\nabla_g L &= \frac{(\nabla_{\boldsymbol{w}} L)\cdot \boldsymbol{v}}{\|\boldsymbol{v}\|} \\ \nabla_{v_i} L &= \frac{g}{\|\boldsymbol{v}\|}\left(1 - \frac{v_i^2}{\|\boldsymbol{v}\|^2}\right)\nabla_{w_i} L,\end{aligned} \tag{10.17}$$

where $\nabla_{\boldsymbol{w}} L$ is a row vector and we have used the quotient rule for the second term.

Empirically, the ability to grow the norm $\|\boldsymbol{v}\|$ renders learning with weight normalization robust to the choice of the learning rate: With large learning rates, the norm of the unnormalized weights increases quickly. When the norm of the weights has become large compared to the norm of the updates, the effective learning rate is stabilized. Thus, weight normalized neural networks can be trained with a larger range of learning rates. For batch normalization a similar explanation holds. One particular difference to batch normalization is that neural networks with weight normalization are not invariant to dataset re-scaling, while batch normalized neural networks are (if batch normalization is used on the pre-activations).

10.4 Self-normalization

Normalization techniques, such as batch-, layer- and weight normalization represented external perturbations to the learning process since external mechanisms control the distribution of the neuron activations or preactivations. Thus, training with normalization techniques is perturbed by stochastic gradient descent (SGD), stochastic regularization (like dropout), and the estimation of the normalization parameters. Both RNNs and CNNs can stabilize learning via weight sharing, therefore they are less prone to these perturbations. In contrast, feed-forward neural networks (FNNs) trained with normalization techniques suffer from these perturbations and have high variance in the training error (see Figure 10.4). This high variance hinders learning and slows it down. Furthermore, strong regularization, such as dropout, is not possible as it would further increase the variance which in turn would lead to divergence of the learning process.

Normalization and SNNs. For a neural network with activation function f, we consider two consecutive layers that are connected by a weight matrix $\boldsymbol{W}$. Since the input to a neural network is a random variable, the activations $\boldsymbol{x}$ in the lower layer, the network inputs $\boldsymbol{s} = \boldsymbol{W}\boldsymbol{x}$, and the activations $\boldsymbol{a} = f(\boldsymbol{s})$ in the higher layer are random variables as well. We assume that all activations x_i of the lower layer have mean $\mu := \mathbb{E}(x_i)$ and variance $\nu := \mathrm{Var}(x_i)$. An activation

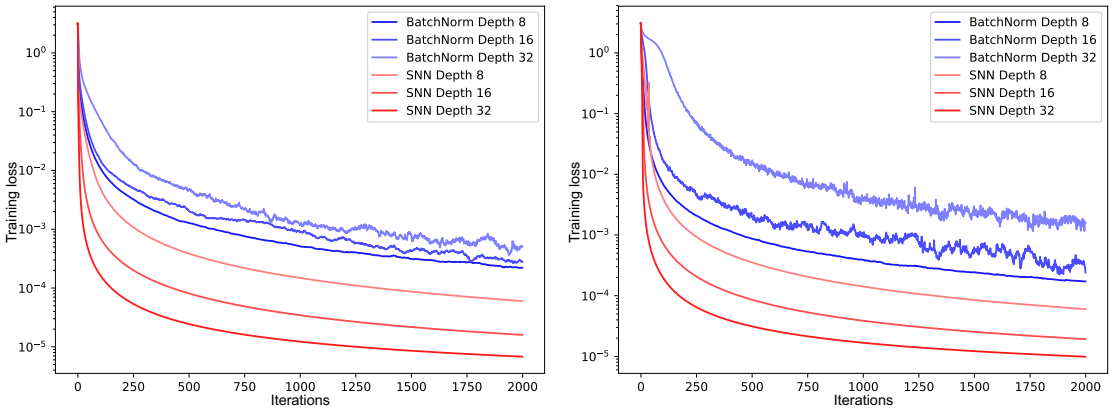


Figure 10.4: The left panel and the right panel show the training error (y-axis) for feed-forward neural networks (FNNs) with batch normalization (BatchNorm) and self-normalizing networks (SNN) across update steps (x-axis) on the MNIST dataset and the CIFAR-10 dataset (see Chapter D), respectively. Networks with 8, 16, and 32 layers and learning rate 1e-5 were tested. FNNs with batch normalization exhibit high variance due to perturbations. In contrast, SNNs do not suffer from high variance as they are more robust to perturbations and learn faster.

y in the higher layer has mean $\tilde{\mu} := \mathbb{E}(a)$ and variance $\tilde{\nu} := \mathrm{Var}(a)$. Here $\mathbb{E}(.)$ denotes the expectation and $\mathrm{Var}(.)$ the variance of a random variable. A single activation $a = f(s)$ has net input $s = \boldsymbol{w}^T \boldsymbol{x}$. For J units with activation $x_j, 1 \leq j \leq J$ in the lower layer, we define J times the mean of the weight vector $\boldsymbol{w} \in \mathbb{R}^J$ as $\omega := \sum_{j=1}^{J} w_j$ and n times the second moment as $\tau := \sum_{j=1}^{J} w_j^2$.

We consider the mapping h that maps mean and variance of the activations from one layer to mean and variance of the activations in the next layer mapping h

$$\begin{pmatrix} \mu \\ \nu \end{pmatrix} \mapsto \begin{pmatrix} \tilde{\mu} \\ \tilde{\nu} \end{pmatrix} : \quad \begin{pmatrix} \tilde{\mu} \\ \tilde{\nu} \end{pmatrix} = h \begin{pmatrix} \mu \\ \nu \end{pmatrix} . \tag{10.18}$$

Normalization techniques such as batch, layer, or weight normalization ensure a mapping h that keeps (μ, ν) and $(\tilde{\mu}, \tilde{\nu})$ close to predefined values, typically $(0, 1)$.

Definition 1 (Self-normalizing neural net)**.** *A neural network is self-normalizing if it possesses a mapping $h : \Omega \mapsto \Omega$ for each activation a that maps mean and variance from one layer to the next and has a stable and attracting fixed point depending on (ω, τ) in Ω. Furthermore, the mean and the variance remain in the domain Ω, that is $h(\Omega) \subseteq \Omega$, where $\Omega = \{(\mu, \nu) \mid \mu \in [\mu_{\min}, \mu_{\max}], \nu \in [\nu_{\min}, \nu_{\max}]\}$. When iteratively applying the mapping h, each point within Ω converges to this fixed point.*

Therefore, we consider activations of a neural network to be normalized, if both their mean and their variance across samples are within predefined intervals. If mean and variance of $\boldsymbol{x}$ are already within these intervals, then also mean and variance of $\boldsymbol{a}$ remain in these intervals, i.e., the normalization is transitive across layers. Within these intervals, the mean and variance both converge to a fixed point if the mapping h is applied iteratively.

Therefore, SNNs keep normalization of activations when propagating them through layers of the network. The normalization effect is observed across layers of a network: in each layer the activations are getting closer to the fixed point. The normalization effect can also observed be for two fixed layers across learning steps: perturbations of lower layer activations or weights are damped in the higher layer by drawing the activations towards the fixed point. If for all a in the higher layer, ω and τ of the corresponding weight vector are the same, then the fixed points are also the same. In this case we have a unique fixed point for all activations a. Otherwise, in the more general case, ω and τ differ for different a but the mean activations are drawn into $[\mu_{\min}, \mu_{\max}]$ and the variances are drawn into $[\nu_{\min}, \nu_{\max}]$.

Constructing Self-Normalizing Neural Networks. We now aim at constructing self-normalizing neural networks by adjusting the properties of the function h. Only two design choices are available for the function h: (1) the activation function and (2) the initialization of the weights.

For the activation function, "scaled exponential linear units" (SELUs) can be used to render a FNN as self-normalizing. The SELU activation function is given by

$$\mathrm{SELU}(x) \;=\; \lambda \begin{cases} x & \text{if } x > 0 \\ \alpha e^x - \alpha & \text{if } x \leq 0 \end{cases} . \tag{10.19}$$

SELUs allow to construct a mapping h with properties that lead to SNNs. SNNs cannot be derived with (scaled) rectified linear units (ReLUs), sigmoid units, $\tanh$ units, and leaky ReLUs. The activation function is required to have (1) negative and positive values for controlling the mean, (2) saturation regions (derivatives approaching zero) to dampen the variance if it is too large in the lower layer, (3) a slope larger than one to increase the variance if it is too small in the lower layer, (4) a continuous curve. The latter ensures a fixed point, where variance damping is equalized by variance increasing. These properties of the activation function are met by multiplying the exponential linear unit (ELU) with $\lambda > 1$ to ensure a slope larger than one for positive net inputs.

For the weight initialization, $\omega = 0$ and $\tau = 1$ for all units in the higher layer have to be used. The next paragraphs will show the advantages of this initialization. Of course, during learning these assumptions on the weight vector will be violated. However, the self-normalizing property can be proved even for weight vectors that are not normalized, therefore, the self-normalizing property can be kept during learning and weight changes.

Deriving the Mean and Variance Mapping Function h. We assume that the x_j are independent from each other but share the same mean μ and variance ν. Of course, the independence assumptions is not fulfilled in general. We will elaborate on the independence assumption below. The network input s in the higher layer is $s = \boldsymbol{w}^T \boldsymbol{x}$ for which we can infer the following moments $\mathbb{E}(s) \;=\; \sum_{j=1}^{J} w_j \; \mathbb{E}(x_j) \;=\; \mu \, \omega$ and $\mathrm{Var}(s) \;=\; \mathrm{Var}(\sum_{j=1}^{J} w_j \; x_j) \;=\; \nu \, \tau$, where we used the independence of the x_j. The net input s is a weighted sum of independent, but not necessarily identically distributed variables x_j, for which the central limit theorem (CLT) states that s approaches a normal distribution: $s \sim \mathcal{N}(\mu\omega, \sqrt{\nu\tau})$ with density $p_{\mathcal{N}}(z; \mu\omega, \sqrt{\nu\tau})$. According to the CLT, the larger J, the closer is s to a normal distribution. For Deep Learning, broad layers with hundreds of neurons x_j are common. Therefore the assumption that s is normally distributed is met well for most currently used neural networks (see Figure 10.5).

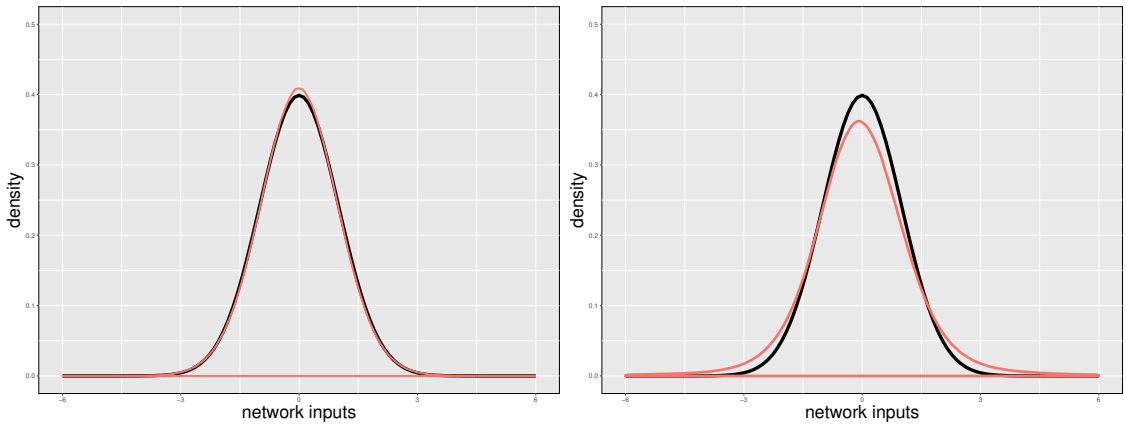


Figure 10.5: Distribution of network inputs of an SNN for the Tox21 data set. The plots show the distribution of network inputs z of the second layer of a typical Tox21 network. The red curves display a kernel density estimator of the network inputs and the black curve is the density of a standard normal distribution. **Left panel:** At initialization time before learning. The distribution of network inputs is close to a standard normal distribution. **Right panel:** After 40 epochs of learning. The distributions of network inputs is close to a normal distribution.

The function h maps the mean and variance of activations in the lower layer to the mean $\tilde{\mu} = \mathbb{E}(a)$ and variance $\tilde{\nu} = \mathrm{Var}(a)$ of the activations a in the next layer:

$$h : \begin{pmatrix} \mu \\ \nu \end{pmatrix} \mapsto \begin{pmatrix} \tilde{\mu} \\ \tilde{\nu} \end{pmatrix} : \quad \begin{aligned} \tilde{\mu}(\mu, \omega, \nu, \tau) &= \int_{-\infty}^{\infty} \mathrm{SELU}(s)\, p_{\mathcal{N}}(z; \mu\omega, \sqrt{\nu\tau})\, \mathrm{d}s \\ \tilde{\nu}(\mu, \omega, \nu, \tau) &= \int_{-\infty}^{\infty} \mathrm{SELU}(s)^2\, p_{\mathcal{N}}(s; \mu\omega, \sqrt{\nu\tau})\, \mathrm{d}s \; - \; (\tilde{\mu})^2 \,. \end{aligned} \tag{10.20}$$

These integrals can be analytically computed and lead to following mappings of the moments:

$$\begin{aligned} \tilde{\mu} \;=\; & \frac{1}{2}\lambda \left((\mu\omega)\, \mathrm{erf}\left(\frac{\mu\omega}{\sqrt{2}\sqrt{\nu\tau}} \right) + \right. \\ & \left. \alpha\, e^{\mu\omega + \frac{\nu\tau}{2}}\, \mathrm{erfc}\left(\frac{\mu\omega + \nu\tau}{\sqrt{2}\sqrt{\nu\tau}} \right) - \alpha\, \mathrm{erfc}\left(\frac{\mu\omega}{\sqrt{2}\sqrt{\nu\tau}} \right) + \sqrt{\frac{2}{\pi}}\sqrt{\nu\tau} e^{-\frac{(\mu\omega)^2}{2(\nu\tau)}} + \mu\omega \right) \end{aligned} \tag{10.21}$$

$$\begin{aligned} \tilde{\nu} \;=\; & \frac{1}{2}\lambda^2 \left(((\mu\omega)^2 + \nu\tau) \left(2 - \mathrm{erfc}\left(\frac{\mu\omega}{\sqrt{2}\sqrt{\nu\tau}} \right) \right) + \alpha^2 \left(-2 e^{\mu\omega + \frac{\nu\tau}{2}}\, \mathrm{erfc}\left(\frac{\mu\omega + \nu\tau}{\sqrt{2}\sqrt{\nu\tau}} \right) \right. \right. \\ & \left. \left. + e^{2(\mu\omega + \nu\tau)}\, \mathrm{erfc}\left(\frac{\mu\omega + 2\nu\tau}{\sqrt{2}\sqrt{\nu\tau}} \right) + \mathrm{erfc}\left(\frac{\mu\omega}{\sqrt{2}\sqrt{\nu\tau}} \right) \right) + \sqrt{\frac{2}{\pi}} (\mu\omega) \sqrt{\nu\tau} e^{-\frac{(\mu\omega)^2}{2(\nu\tau)}} \right) - (\tilde{\mu})^2 \end{aligned} \tag{10.22}$$

Stable and Attracting Fixed Point $(0, 1)$ for Normalized Weights. We assume a normalized weight vector $\boldsymbol{w}$ with $\omega = 0$ and $\tau = 1$. Given a fixed point (μ, ν), we can solve equations Eq. (10.21) and Eq. (10.22) for α and λ. We chose the fixed point $(\mu, \nu) = (0, 1)$, which is typical for activation normalization. We obtain the fixed point equations $\tilde{\mu} = \mu = 0$ and $\tilde{\nu} = \nu = 1$

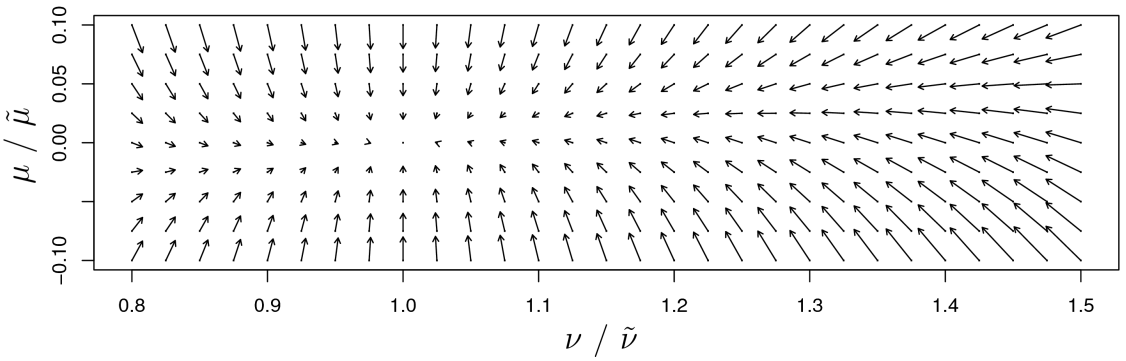


Figure 10.6: For $\omega = 0$ and $\tau = 1$, the mapping g of mean μ (x-axis) and variance ν (y-axis) to the next layer's mean $\tilde{\mu}$ and variance $\tilde{\nu}$ is depicted. Arrows show in which direction (μ, ν) is mapped by $g : (\mu, \nu) \mapsto (\tilde{\mu}, \tilde{\nu})$. The fixed point of the mapping g is $(0, 1)$.

that we solve for α and λ and obtain the solutions $\alpha_{01} \approx 1.6733$ and $\lambda_{01} \approx 1.0507$, where the subscript 01 indicates that these are the parameters for fixed point $(0, 1)$. There are exact analytical expressions for α_{01} and λ_{01}. We are interested whether the fixed point $(\mu, \nu) = (0, 1)$ is stable and attracting. If the Jacobian of h has a norm smaller than 1 at the fixed point, then h is a contraction mapping and the fixed point is stable. The (2x2)-Jacobian $\mathcal{J}(\mu, \nu)$ of $h : (\mu, \nu) \mapsto (\tilde{\mu}, \tilde{\nu})$ evaluated at the fixed point $(0, 1)$ with α_{01} and λ_{01} is

$$\mathcal{J}(\mu, \nu) = \begin{pmatrix} \partial \frac{\mu^{\text{new}}(\mu,\nu)}{\partial \mu} & \partial \frac{\mu^{\text{new}}(\mu,\nu)}{\partial \nu} \\ \partial \frac{\nu^{\text{new}}(\mu,\nu)}{\partial \mu} & \partial \frac{\nu^{\text{new}}(\mu,\nu)}{\partial \nu} \end{pmatrix}, \qquad \mathcal{J}(0, 1) = \begin{pmatrix} 0.0 & 0.088834 \\ 0.0 & 0.782648 \end{pmatrix}. \tag{10.23}$$

The spectral norm of $\mathcal{J}(0, 1)$ (its largest singular value) is $0.7877 < 1$. That means g is a contraction mapping around the fixed point $(0, 1)$ (the mapping is depicted in Figure 10.6). Therefore, $(0, 1)$ is a stable fixed point of the mapping g.

It can be shown that a fixed point close to $(0, 1)$ is maintained also during learning when weights are changed (see Klambauer et al. (2017)).

Consequently, feed-forward neural networks with many units in each layer and with the SELU activation function are self-normalizing. To give an intuition, the main property of SELUs is that they damp the variance for negative net inputs and increase the variance for positive net inputs. The variance damping is stronger if net inputs are further away from zero while the variance increase is stronger if net inputs are close to zero. Thus, for large variance of the activations in the lower layer the damping effect is dominant and the variance decreases in the higher layer. Vice versa, for small variance the variance increase is dominant and the variance increases in the higher layer.

However, we cannot guarantee that mean and variance remain in the domain Ω. Therefore, the cases where (μ, ν) are outside Ω should also be considered. It is especially crucial to consider ν because this variable has much stronger influence than μ. Mapping ν across layers to a high value corresponds to an exploding gradient, since the Jacobian of the activation of high layers with respect to activations in lower layers has large singular values. Analogously, mapping ν across layers to a low value corresponds to an vanishing gradient. Bounding the mapping of ν from above and below would avoid both exploding and vanishing gradients. Both bounding the variance from above and below, can be shown theoretically.

Initialization. Since SNNs have a fixed point at zero mean and unit variance for normalized weights $\omega = \sum_{j=1}^{J} w_j = 0$ and $\tau = \sum_{j=1}^{J} w_j^2 = 1$ (see above), SNNs are initialized such that these constraints are fulfilled in expectation. The weights are drawn from a Gaussian distribution with $\mathbb{E}(w_j) = 0$ and variance $\mathrm{Var}(w_j) = 1/n$. Uniform and truncated Gaussian distributions with these moments led to networks with similar behavior. This is exactly the so-called LeCun initialization (see also Section 9.3.2). The "MSRA initialization" is similar since it uses zero mean and variance $2/n$ to initialize the weights (He et al., 2015). The additional factor 2 counters the effect of rectified linear units.

Alpha Dropout. Standard dropout randomly sets an activation x to zero with probability $1 - q$ for $0 < q \leq 1$. In order to preserve the mean, the activations are scaled by $1/q$ during training. If x has mean $\mathbb{E}(x) = \mu$ and variance $\mathrm{Var}(x) = \nu$, and the dropout variable d follows a binomial distribution $B(1, q)$, then the mean $\mathbb{E}(1/qdx) = \mu$ is kept. Dropout fits well to rectified linear units, since zero is in the low variance region and corresponds to the default value. For scaled exponential linear units, the default and low variance value is $\lim_{x \to -\infty} \mathrm{SELU}(x) = -\lambda\alpha = \alpha'$. Therefore, "alpha dropout" is proposed, that randomly sets inputs to α'. The new mean and new variance is $\mathbb{E}(xd + \alpha'(1 - d)) = q\mu + (1 - q)\alpha'$, and $\mathrm{Var}(xd + \alpha'(1 - d)) = q((1 - q)(\alpha' - \mu)^2 + \nu)$. We aim at keeping mean and variance to their original values after "alpha dropout", in order to ensure the self-normalizing property even for "alpha dropout". The affine transformation $a(xd+\alpha'(1-d))+b$ allows to determine parameters a and b such that mean and variance are kept to their values: $\mathbb{E}(a(xd+\alpha'(1-d))+b) = \mu$ and $\mathrm{Var}(a(xd+\alpha'(1-d))+b) = \nu$. In contrast to dropout, a and b will depend on μ and ν, however our SNNs converge to activations with zero mean and unit variance. With $\mu = 0$ and $\nu = 1$, we obtain $a = \left(q + \alpha'^2 q(1 - q)\right)^{-1/2}$ and $b = -\left(q + \alpha'^2 q(1 - q)\right)^{-1/2} ((1 - q)\alpha')$. The parameters a and b only depend on the dropout rate $1-q$ and the most negative activation α'. Empirically, we found that dropout rates $1-q = 0.05$ or 0.10 lead to models with good performance. "Alpha-dropout" fits well to scaled exponential linear units by randomly setting activations to the negative saturation value.

Applicability of the central limit theorem and independence assumption. In the derivative of the mapping (Eq. (10.20)), we used the central limit theorem (CLT) to approximate the network inputs $z = \sum_{j=1}^{J} w_j x_j$ with a normal distribution. We justified normality because network inputs represent a weighted sum of the inputs x_j, where for Deep Learning J is typically large. The Berry-Esseen theorem states that the convergence rate to normality is $J^{-1/2}$ (Korolev and Shevtsova, 2012). In the classical version of the CLT, the random variables have to be independent and identically distributed, which typically does not hold for neural networks. However, the Lyapunov CLT does not require the variable to be identically distributed anymore. Furthermore, even under weak dependence, sums of random variables converge in distribution to a Gaussian distribution Bradley (1981).

Vanishing and exploding activations in self-normalizing neural networks. At the beginning of this chapter, we introduced the problem of vanishing and exploding activations (see Fig. 10.1) that describes the distribution shifts of preactivations and activations through layers of a neural network. We now investigate these distributions for a self-normalizing neural networks and find that the distributions remain stable through layers, see Fig.10.7.

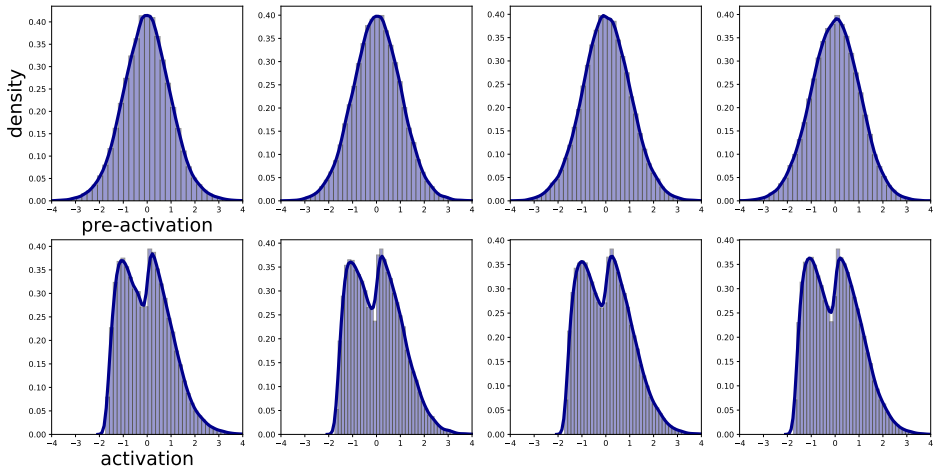


Figure 10.7: Distribution of activations through multiple layers of a self-normalizing neural network. The distribution of activations and pre-activations in a five layer self-normalizing neural network trained on MNIST (see Section D.1) are shown. Each layer has 512 hidden units. The output-layer activations are not shown. The distribution of the pre-activation remains close to a standard Gaussian distribution and the distribution of the activations maintains zero mean and unit variance.

10.5 Summary of invariance properties of normalization techniques

We have seen in Eq. (10.11), normalization techniques can equip neural network layers with some invariance properties. The invariance properties reported in 10.1 only hold if the normalization technique is applied on the pre-activations of full-connected layer (linear transformation). Invariance to some transformations need not necessarily be beneficial for a neural network model. In many cases, this can also depend on the application area.

	weight matrix re-scaling	weight matrix re-centering	weight vector re-scaling	data set re-scaling	data set re-centering	single training case re-scaling
batch norm	invariant	no	invariant	invariant	invariant	no
weight norm	invariant	no	invariant	no	no	no
layer norm	invariant	invariant	no	invariant	no	invariant
self norm	no	no	no	no	no	no

Table 10.1: Invariance properties of normalization methods according to (Ba et al., 2016). The invariance properties only hold if the normalization technique is applied on the pre-activations of full-connected layer (linear transformation).

Appendix A

Activation functions

GÜNTER KLAMBAUER

A.1 Activation functions for hidden units

A.1.1 Sigmoid units: numeric control, soft step function

The commonly used activation functions are sigmoid functions:

The *logistic sigmoid* function is

$$f(x) \;=\; \frac{1}{1 \;+\; \exp(-\,x)} \tag{A.1}$$

and the *hyperbolic tangent* activation function ($\tanh$) is

$$f(x) \;=\; \tanh(x) \;=\; \frac{\exp(x) \;-\; \exp(-\,x)}{\exp(x) \;+\; \exp(-\,x)}\,. \tag{A.2}$$

Both functions are equivalent because

$$\frac{1}{2}\,(\tanh(x/2) \;+\; 1) \;=\; \frac{1}{1 \;+\; \exp(-\,x)}\,. \tag{A.3}$$

That means through weight scaling and through adjusting the bias weights the networks with logistic function or $\tanh$ can be transformed into each other.

The derivatives of the logistic sigmoid function is the following:

$$f'(x) = f(x)(1 - f(x)), \tag{A.4}$$

and the derivative of the $\tanh$ activation function is:

$$f'(x) = 1 - f(x)^2. \tag{A.5}$$

Also quadratic or other activation functions can be used but the fact that the sigmoid functions are squashed into an interval makes learning robust. Because the activation is bounded the derivatives are bounded as we see later. Networks with sigmoid activation are even more robust against

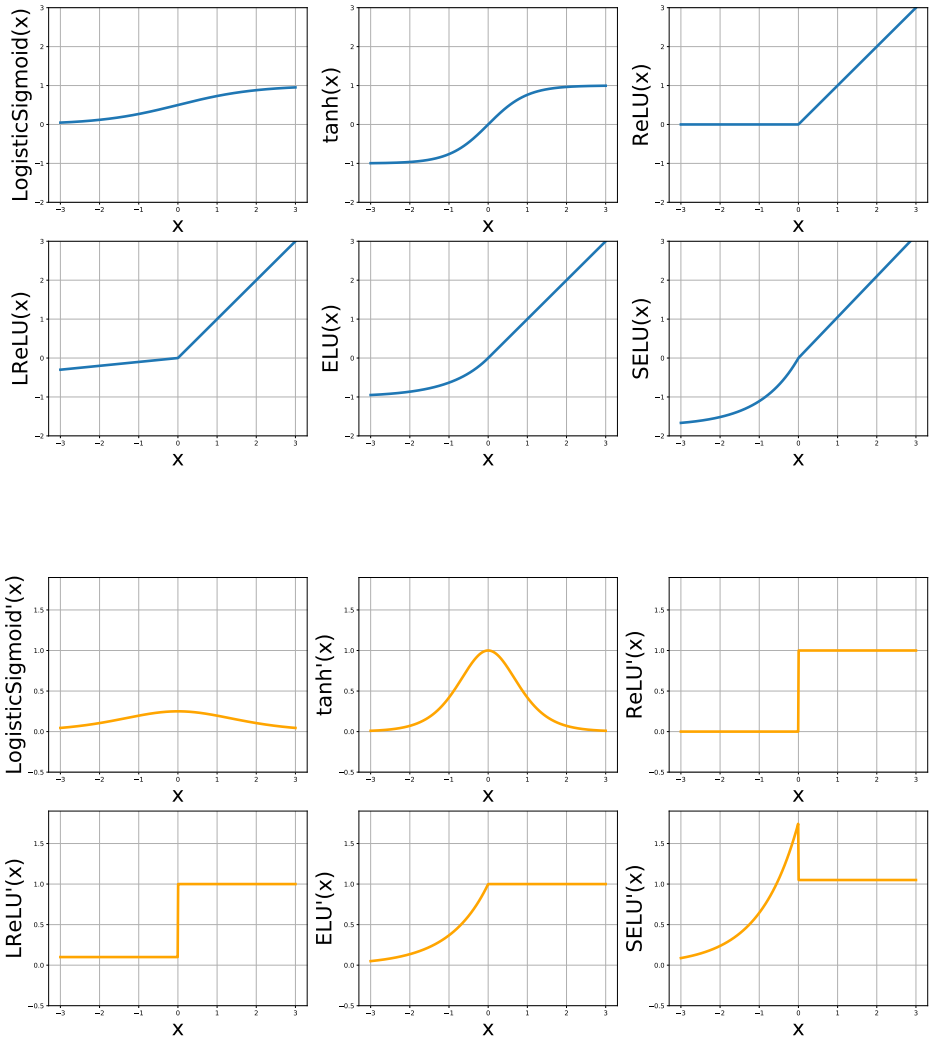


Figure A.1: Graphs of commonly used activation functions for hidden layers. **Top:** Graph of the function. **Bottom:** Graph of the derivative.

unknown input which can drive some activations to regions which are not explored in the training phase.

Symmetric Networks. Symmetric networks can be produced with the $\tanh$ function if the signs of input weights and output weights are changed because $\tanh(-x) = -\tanh(x)$, therefore, $w_2 \tanh(w_1\, x) = (-w_2)\, \tanh((-w_1)\, x)$.

Permutations of the hidden units in one layer leads to equivalent networks. That means the same function can be represented through different network parameters.

Properties:

- Boundedness: Sigmoid units are upper and lower bounded, thus activations stay within the interval $f(x) \in [0, 1]$ for the logistic sigmoid or $f(x) \in [-1, 1]$ for the tanh sigmoid.
- Monotonicity: Both sigmoid functions are strictly monotonically increasing.
- Behavior around origin: The hyperbolic tangent approximates identity near the origin.
- Maximum of absolute derivative: 1 for tanh and 0.25 for logistic sigmoid.

A.1.2 Rectified linear units (ReLU): efficient non-linearities

Rectified linear units have the following activation functions:

$$f(x) = \begin{cases} 0 & \text{for } x < 0 \\ x & \text{for } x \geq 0 \end{cases} \tag{A.6}$$

and relatively simple derivative:

$$f'(x) = \begin{cases} 0 & \text{for } x < 0 \\ 1 & \text{for } x \geq 0 \end{cases}. \tag{A.7}$$

Properties:

- Boundedness: ReLU activations are lower bounded, but do not have an upper bound: $f(x) \in [0, \infty)$
- Monotonicity: The ReLU activation function is monotonically increasing.
- Behavior around origin: does not approximate identity around zero.
- Maximum of absolute derivative: $\max_x f'(x) = 1$

A.1.3 Leaky rectified linear units (Leaky-ReLU, LReLU)

Leaky-ReLUs are given by:

$$f(x) = \begin{cases} 0.01x & \text{for } x < 0 \\ x & \text{for } x \geq 0 \end{cases} \tag{A.8}$$

$$f'(x) = \begin{cases} 0.01 & \text{for } x < 0 \\ 1 & \text{for } x \geq 0 \end{cases} \tag{A.9}$$

Properties:

- Boundedness: LReLU activations are not bounded: $f(x) \in (-\infty, \infty)$
- Monotonicity: The LReLU activation function is strictly monotonically increasing.
- Behavior around origin: does not approximate identity around zero.
- Maximum of absolute derivative: $\max_x f'(x) = 1$

A.1.4 (Scaled) exponential linear units (ELU and SELU): countering bias shift and self-normalization

Scaled exponential linear units have the following activation functions:

$$f(x) = \lambda \begin{cases} \alpha(e^x - 1) & \text{for } x < 0 \\ x & \text{for } x \geq 0 \end{cases} \tag{A.10}$$

with $\lambda \approx 1.0507$ and $\alpha \approx 1.67326$. Note that exponential linear units (ELU) have $\lambda = 1$ and $\alpha = 1$

$$f'(x) = \lambda \begin{cases} \alpha e^x & \text{for } x < 0 \\ 1 & \text{for } x \geq 0 \end{cases} \tag{A.11}$$

Properties:

- Boundedness: SELU activations have a lower bound: $f(x) \in (-\lambda\alpha, \infty)$
- Monotonicity: The SELU activation function is strictly monotonically increasing.
- Behavior around origin: does not approximate identity around zero.
- Maximum of absolute derivative: $\max_x f'(x) = \lambda\alpha$.
- Provide a sufficiently broad network with the ability to keep activations around zero mean and unit variance (self-normalization). See Section on Self-Normalizing Neural Networks.

A.1.5 Higher order units.

Higher order units do not use a linear s_i. For example second order units have the form

$$s_i \;=\; \sum_{(j_1,j_2)=(0,0)}^{N} w_{ij_1j_2}\; a_{j_1}\; a_{j_2}\;. \tag{A.12}$$

Note that linear and constant terms are considered because of the bias unit $a_0 \;=\; 1$.

A.2 Activation functions for the output layer

A.2.1 Linear units: regression tasks

For regression tasks, the output units usually are linearly activated by

$$f(x) = x \tag{A.13}$$

with the derivative

$$f'(x) = 1. \tag{A.14}$$

A.2.2 Logistic sigmoid units: binary classification tasks

For binary classification tasks, usually logistic sigmoid units are used for output tasks:

$$f(x) = \sigma(x) = \frac{1}{1+e^{-x}} \tag{A.15}$$

with the derivative

$$f'(x) = \frac{e^x}{1+(e^x)^2} = \sigma(x)(1-\sigma(x)). \tag{A.16}$$

A.2.3 Softmax: categorical classification tasks

For multi-class classification tasks, usually softmax units are used for output tasks:

$$\mathrm{softmax}(\boldsymbol{s}) = \left(\frac{e^{s_1}}{\sum_{j=1}^{K} e^{s_j}}, \ldots, \frac{e^{s_k}}{\sum_{j=1}^{K} e^{s_j}}, \ldots, \frac{e^{s_K}}{\sum_{j=1}^{K} e^{s_j}}\right)^T. \tag{A.17}$$

with the derivative

$$\frac{\partial\,\mathrm{softmax}(\boldsymbol{s})_i}{\partial s_k} = \mathrm{softmax}(\boldsymbol{s})_i(\delta_{ik} - \mathrm{softmax}(\boldsymbol{s})_k), \tag{A.18}$$

where δ_{ik} is the Kronecker-delta.

Appendix B

Matrix Derivatives

PHILIPP RENZ AND GÜNTER KLAMBAUER

When working with neural networks we often have to deal with derivatives of scalars with respect to vectors or matrices. Here we establish some of the basics that are often needed in practice and try to give some tools to work things out for yourself. Let's consider the vector $\boldsymbol{a} \in \mathbb{R}^n$:

$$\boldsymbol{a} = \begin{pmatrix} a_1 \\ \vdots \\ a_n \end{pmatrix}. \tag{B.1}$$

It often makes sense in practice not to write this as "vector" but when calculating something to just write a_i. That means if we have an single index in a formula we can view it as a vector. Similarly if we have some quantity A_{ij} that depends on two indices we can write it as a matrix:

$$\boldsymbol{A} = \begin{pmatrix} A_{11} & A_{12} & \dots & A_{1n} \\ A_{21} & A_{22} & \dots & A_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ A_{m1} & A_{m2} & \dots & A_{mn}. \end{pmatrix} \tag{B.2}$$

Often we also have quantities that depend on more than two indices, for example T_{ijk} for which common matrix notation fails. These are often called tensors although they are not in the original sense of the word.

Let's get into something more practical. When differentiating a vector with respect to a scalar we get:

$$\left(\frac{\partial \boldsymbol{a}}{\partial b}\right)_i = \frac{\partial a_i}{\partial b}, \tag{B.3}$$

where the result is again a vector.

Similarly when we differentiate a vector w.r.t. a vector what we get as a result is called the Jacobian:

$$\boldsymbol{J} = \frac{\partial \boldsymbol{a}}{\partial \boldsymbol{b}} = \begin{pmatrix} \frac{\partial a_1}{\partial b_1} & \frac{\partial a_1}{\partial b_2} & \dots & \frac{\partial a_1}{\partial b_n} \\ \frac{\partial a_2}{\partial b_1} & \frac{\partial a_2}{\partial b_2} & \dots & \frac{\partial a_2}{\partial b_n} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial a_m}{\partial b_1} & \frac{\partial a_m}{\partial b_2} & \dots & \frac{\partial a_m}{\partial b_n}. \end{pmatrix}, \tag{B.4}$$

where by convention the rows indices correspond to the entries of the vector on being derived and the column indices those w.r.t. to which the derivative is taken. We can write down the entries of $\boldsymbol{J}$ in another way:

$$J_{ij} = \frac{\partial a_i}{\partial b_j} \tag{B.5}$$

A problem often encountered in practice is

$$\frac{\partial}{\partial \boldsymbol{x}}(\boldsymbol{x}^T \boldsymbol{a}) = \boldsymbol{a}. \tag{B.6}$$

We get to this result by writing the sum explicitly

$$\frac{\partial}{\partial x_i} \sum_j x_j a_j = \sum_j \frac{\partial x_j}{\partial x_i} a_j = \sum_j \delta_{ij} a_j = a_i \tag{B.7}$$

where δ_{ij} is the Kronecker-delta, which is defined as

$$\delta_{ij} = \begin{cases} 1, & \text{if } i = j \\ 0, & \text{else.} \end{cases} \tag{B.8}$$

Here we have used the fact that $\frac{\partial x_j}{\partial x_i} = \delta_{ij}$ as the components of $\boldsymbol{x}$ are independent of each other. We also see the interesting effect that δ_{ij} has inside of sums. We see that all of the addends are zero except for a_j. Thus we see that sums over expressions involving a Kronecker-delta can be resolved by mere index replacement.

Next we'll look at the problem of deriving a product of matrices w.r.t. to a scalar:

$$\frac{\partial}{\partial x}(\boldsymbol{A}\boldsymbol{B}) = \frac{\partial \boldsymbol{A}}{\partial x}\boldsymbol{B} + \boldsymbol{A}\frac{\partial \boldsymbol{B}}{\partial x}. \tag{B.9}$$

We can get the result for this by writing it explicitly:

$$\frac{\partial}{\partial x} \sum_j A_{ij} B_{jk} = \sum_j \frac{\partial A_{ij}}{\partial x} B_{jk} + A_{ij} \frac{\partial B_{jk}}{\partial x}, \tag{B.10}$$

where we used the product rule.

Now we'll introduce the chain rule for vector valued functions

$$\frac{\partial \boldsymbol{f}(\boldsymbol{g}(\boldsymbol{x}))}{\partial \boldsymbol{x}} = \frac{\partial \boldsymbol{f}}{\partial \boldsymbol{g}} \frac{\partial \boldsymbol{g}}{\partial \boldsymbol{x}}. \tag{B.11}$$

In index notation this is:

$$\frac{\partial f_i}{\partial x_j} = \sum_k \frac{\partial f_i}{\partial g_k} \frac{\partial g_k}{\partial x_j}. \tag{B.12}$$

In neural nets we often have vector valued functions of a special form, namely elementwise application of a scalar valued function to each element of a vector:

$$\boldsymbol{f}(\boldsymbol{x}) = \begin{pmatrix} f(x_1) \\ \vdots \\ f(x_n) \end{pmatrix}. \tag{B.13}$$

We can see that the Jacobian is a diagonal matrix:

$$\frac{\partial f_i}{\partial x_j} = f'(x_i)\delta_{ij} \tag{B.14}$$

For an NN the following is often needed:

$$\frac{\partial}{\partial \boldsymbol{x}}(\boldsymbol{W}\boldsymbol{f}(\boldsymbol{x})) = \frac{\partial \boldsymbol{W}}{\partial \boldsymbol{x}}\boldsymbol{f}(\boldsymbol{x}) + \boldsymbol{W}\frac{\partial \boldsymbol{f}(\boldsymbol{x})}{\partial \boldsymbol{x}} = \boldsymbol{W}\frac{\partial \boldsymbol{f}(\boldsymbol{x})}{\partial \boldsymbol{x}}. \tag{B.15}$$

In index notation we get a similar result:

$$\begin{aligned}\frac{\partial}{\partial x_j}\left(\sum_k W_{ik} f_k(\boldsymbol{x})\right) =& \sum_k W_{ik}\frac{\partial f_k(\boldsymbol{x})}{\partial x_j} && \text{(B.16)}\\ =& \sum_k W_{ik} f'(x_k)\delta_{jk} && \text{(B.17)}\\ =& W_{ij} f'(x_j), && \text{(B.18)}\end{aligned}$$

from which we get the same result. The above can be interpreted as columnwise multiplication of $\boldsymbol{W}$ with a vector, which can also be expressed as multiplicaton with a diagonal matrix as mentioned above.

B.1 Some Backprop Formulas

Using index notation we can get the Jacobian we want:

$$\begin{aligned}\frac{\partial s_a^{[l]}}{\partial s_i^{[l-1]}} =& \frac{\partial}{\partial s_i^{[l-1]}}\sum_b w_{ab}^{[l]} f(s_b^{[l-1]}) && \text{(B.19)}\\ =& \sum_b w_{ab}^{[l]}\frac{\partial f(s_b^{[l-1]})}{\partial s_i^{[l-1]}} && \text{(B.20)}\\ =& \sum_b w_{ab}^{[l]} f'(s_b^{[l-1]})\frac{\partial s_b^{[l-1]}}{\partial s_i^{[l-1]}} && \text{(B.21)}\\ =& \sum_b w_{ab}^{[l]} f'(s_b^{[l-1]})\delta_{ib} && \text{(B.22)}\\ =& w_{ai}^{[l]} f'(s_i^{[l-1]}). && \text{(B.23)}\end{aligned}$$

Appendix C

Convexity of linear models

C.1 Linear regression

The objective that is minimized for the linear regression model $g(\boldsymbol{x}^n; \boldsymbol{w}) = \boldsymbol{w}^T \boldsymbol{x}^n$ is the following:

$$R_{\text{emp}}(\boldsymbol{y}, \boldsymbol{X}, \boldsymbol{w}) = \frac{1}{2} \sum_{n=1}^{N} \left(y^n - \boldsymbol{w}^T \boldsymbol{x}^n\right)^2 \qquad = \frac{1}{2} \left(\boldsymbol{y} - \boldsymbol{X}\boldsymbol{w}\right)^T \left(\boldsymbol{y} - \boldsymbol{X}\boldsymbol{w}\right). \tag{C.1}$$

The first derivative is the following well-known expression:

$$\nabla_{\boldsymbol{w}} R_{\text{emp}}(\boldsymbol{y}, \boldsymbol{X}, \boldsymbol{w}) = \boldsymbol{X}^T \left(\boldsymbol{X}\boldsymbol{w} - \boldsymbol{y}\right), \tag{C.2}$$

and the Hessian is consequently:

$$\nabla^2_{\boldsymbol{w}} R_{\text{emp}}(\boldsymbol{y}, \boldsymbol{X}, \boldsymbol{w}) = \boldsymbol{X}^T \boldsymbol{X}. \tag{C.3}$$

Thus the Hessian is positive semi-definite.

C.2 (Regularized) Linear Logistic Regression is Strictly Convex

Following Jason D. M. Rennie, we show that linear Logistic Regression is strictly convex.

In the linear case we have

$$g(\boldsymbol{x}^n; \boldsymbol{w}) = \boldsymbol{w}^T \boldsymbol{x}^n. \tag{C.4}$$

For labels $y \in +1, -1$ we have

$$\frac{\partial L}{\partial w_j} = -\sum_{i=1}^{l} y^n x_{ij} \left(1 - p(y^n \mid \boldsymbol{x}; \boldsymbol{w})\right). \tag{C.5}$$

The second derivatives of the objective L that is minimized are

$$H_{jk} = \frac{\partial L}{\partial w_j \partial w_k} = \sum_{i=1}^{l} \left(y^n\right)^2 x_{ij} x_{ik}\, p(y^n \mid \boldsymbol{x}; \boldsymbol{w}) \left(1 - p(y^n \mid \boldsymbol{x}; \boldsymbol{w})\right), \tag{C.6}$$

where $\boldsymbol{H}$ is the Hessian.

Because $p(1-p) \geq 0$ for $p \leq 1$, we can define

$$\rho_{ij} = x_{ij}\sqrt{p(y^n \mid \boldsymbol{x}; \boldsymbol{w})\,(1 - p(y^n \mid \boldsymbol{x}; \boldsymbol{w}))}. \tag{C.7}$$

The bilinear form of the Hessian with a vector $\boldsymbol{a}$ is

$$\begin{aligned}
\boldsymbol{a}^T \boldsymbol{H} \boldsymbol{a} &= \sum_{i=1}^{l} \sum_{j=1}^{d} \sum_{k=1}^{d} x_{ij} x_{ik} a_j a_k \, p(y^n \mid \boldsymbol{x}; \boldsymbol{w})\,(1 - p(y^n \mid \boldsymbol{x}; \boldsymbol{w})) \\
&= \sum_{i=1}^{l} \sum_{j=1}^{d} a_j x_{ij} \sqrt{p(y^n \mid \boldsymbol{x}; \boldsymbol{w})\,(1 - p(y^n \mid \boldsymbol{x}; \boldsymbol{w}))} \\
&= \sum_{i=1}^{l} \left(\boldsymbol{a}^T \boldsymbol{\rho}_i\right) \left(\boldsymbol{a}^T \boldsymbol{\rho}_i\right) = \sum_{i=1}^{l} \left(\boldsymbol{a}^T \boldsymbol{\rho}_i\right)^2 \geq 0.
\end{aligned} \tag{C.8}$$

Because we did not make any restriction on $\boldsymbol{a}$, the Hessian is positive semidefinite.

Adding a term like $\frac{1}{2}\boldsymbol{w}^T \boldsymbol{w}$ to the objective for regularization, then the Hessian of the objective is strict positive definite.

C.3 (Regularized) Linear Softmax is Strictly Convex

Following Jason D. M. Rennie, we show that linear Softmax is strictly convex.

In the linear case we have

$$g_k(\boldsymbol{x}^n; \boldsymbol{w}_k) = \boldsymbol{w}_k^T \boldsymbol{x}^n \tag{C.9}$$

or in vector notation

$$\boldsymbol{g}(\boldsymbol{x}^n; \boldsymbol{W}) = \boldsymbol{W}^T \boldsymbol{x}^n. \tag{C.10}$$

The derivatives are

$$\frac{\partial L}{\partial w_{kn}} = \sum_{i=1}^{l} x_{in}\, p(k \mid \boldsymbol{x}^n; \boldsymbol{W}) - \delta_{y^n = k} \sum_{i=1}^{l} x_{in}. \tag{C.11}$$

To compute the second derivatives of the objective, we need the derivatives of the probabilities with respect to the parameters:

$$\frac{\partial p(v \mid \boldsymbol{x}^n; \boldsymbol{W})}{\partial w_{vm}} = x_{im} p(k \mid \boldsymbol{x}^n; \boldsymbol{W})\,(1 - p(k \mid \boldsymbol{x}^n; \boldsymbol{W})) \tag{C.12}$$

$$\frac{\partial p(k \mid \boldsymbol{x}^n; \boldsymbol{W})}{\partial w_{vm}} = x_{im} p(k \mid \boldsymbol{x}^n; \boldsymbol{W}) p(v \mid \boldsymbol{x}^n; \boldsymbol{W}). \tag{C.13}$$

The second derivatives of L with respect to the components of the parameter vector $\boldsymbol{w}$ are

$$\begin{aligned} H_{kn,vm} &= \frac{\partial L}{\partial w_{kn} \partial w_{vm}} \\ &= (\delta_{k=v}(1 - p(k \mid \boldsymbol{x}^n; \boldsymbol{W})) - (1 - \delta_{k=v}) p(v \mid \boldsymbol{x}^n; \boldsymbol{W})) \\ &\quad \cdot \sum_{i=1}^{l} x_{in} x_{im}\, p(k \mid \boldsymbol{x}^n; \boldsymbol{W}). \end{aligned} \tag{C.14}$$

Again we define a vector $\boldsymbol{a}$ with components a_{uj} (note, the double index is considered as single index so that a matrix is written as vector).

We consider the bilinear from

$$\begin{aligned} \boldsymbol{a}^T \boldsymbol{H} \boldsymbol{a} &= \sum_{k,n} \sum_{v,m} \sum_{i} a_{kn} a_{vm} x_{in} x_{im}\, p(k \mid \boldsymbol{x}^n; \boldsymbol{W}) \\ &\quad \cdot (\delta_{k=v}(1 - p(k \mid \boldsymbol{x}^n; \boldsymbol{W})) - (1 - \delta_{k=v}) p(v \mid \boldsymbol{x}^n; \boldsymbol{W})) \\ &= \sum_{k,n} \sum_{i} a_{kn} x_{in} p(k \mid \boldsymbol{x}^n; \boldsymbol{W}) \sum_{m} x_{im} \left(a_{km} - \sum_{v} a_{vm} p(v \mid \boldsymbol{x}^n; \boldsymbol{W}) \right) \\ &= \sum_{i} \sum_{n} x_{in} \sum_{k} a_{kn} p(k \mid \boldsymbol{x}^n; \boldsymbol{W}) \sum_{m} x_{im} \left(a_{km} - \sum_{v} a_{vm} p(v \mid \boldsymbol{x}^n; \boldsymbol{W}) \right) \\ &= \sum_{i} \left[\sum_{n} x_{in} \sum_{k} a_{kn} p(k \mid \boldsymbol{x}^n; \boldsymbol{W}) \sum_{m} x_{im} a_{km} \right] \\ &\quad - \left[\left(\sum_{n} x_{in} \sum_{k} a_{kn} p(k \mid \boldsymbol{x}^n; \boldsymbol{W}) \right) \left(\sum_{m} x_{im} \sum_{v} a_{vm} p(v \mid \boldsymbol{x}^n; \boldsymbol{W}) \right) \right] \\ &= \sum_{i} \left[\sum_{k} p(k \mid \boldsymbol{x}^n; \boldsymbol{W}) \left(\sum_{n} x_{in} a_{kn} \right) \left(\sum_{m} x_{im} a_{km} \right) \right] \\ &\quad - \left[\left(\sum_{n} x_{in} \sum_{k} a_{kn} p(k \mid \boldsymbol{x}^n; \boldsymbol{W}) \right)^2 \right] \\ &= \sum_{i} \left[\sum_{k} p(k \mid \boldsymbol{x}^n; \boldsymbol{W}) \left(\sum_{n} x_{in} a_{kn} \right)^2 \right] - \left[\left(\sum_{n} x_{in} \sum_{k} a_{kn} p(k \mid \boldsymbol{x}^n; \boldsymbol{W}) \right)^2 \right]. \end{aligned} \tag{C.15}$$

If for each summand of the sum over i

$$\sum_{k} p(k \mid \boldsymbol{x}^n; \boldsymbol{W}) \left(\sum_{n} x_{in} a_{kn} \right)^2 - \left(\sum_{k} p(k \mid \boldsymbol{x}^n; \boldsymbol{W}) \sum_{n} x_{in} a_{kn} \right)^2 \geq 0 \tag{C.16}$$

holds, then the Hessian $\boldsymbol{H}$ is positive semidefinite. This holds for arbitrary number of samples as each term corresponds to a sample.

In the last equation the $p(k \mid \boldsymbol{x}^n; \boldsymbol{W})$ can be viewed as a multinomial distribution over k. The terms $\sum_n x_{in} a_{kn}$ can be viewed as functions depending on k.

In this case $\sum_k p(k \mid \boldsymbol{x}^n; \boldsymbol{W}) \left(\sum_n x_{in} a_{kn}\right)^2$ is the second moment and the squared expectation is $\left(\sum_k p(k \mid \boldsymbol{x}^n; \boldsymbol{W}) \sum_n x_{in} a_{kn}\right)^2$. Therefore the left hand side of inequality (C.16) is the second central moment, which is larger than zero.

Alternatively inequality (C.16) can be proven by applying Jensen's inequality with the square function as a convex function.

We have proven that the Hessian $\boldsymbol{H}$ is positive semidefinite.

Adding a term like $\frac{1}{2} \sum_k \boldsymbol{w}_k^T \boldsymbol{w}_k$ to the objective for regularization, then the Hessian of the objective is strictly positive definite.

Appendix D

Data sets

ELISABETH RUMETSHOFER AND GÜNTER KLAMBAUER

D.1 MNIST

The MNIST (Modified National Institute of Standards and Technology) data set (LeCun et al., 2019) contains black and white images of handwritten digits. It has been compiled from a larger data set from NIST (a mix of NIST SD-1 and NIST SD-3). Each image has a resolution of 28x28 pixels and contains one digit that has been centered in the image. The data set consists of 70,000 images, divided into a training set with 60,000 images and a test set with 10,000 images. Every image belongs to one of ten classes, i.e. to the digits 0-9 (see Fig. D.1). The images are evenly distributed across classes, such that 6,000 training images and 1,000 test images are available per class. A list of the error rates of the best performing approaches for different types of classifiers is provided in Table D.1.

Figure D.1: Examples from the MNIST data set for each of the 10 classes.

Classifier	Error Rate (%)	Reference
Linear Classifier	7.60	LeCun (1998)
Non-Linear Classifier	3.60	LeCun (1998)
Boosted Stumps	0.87	Kegl et al., ICML (2009)
Support Vector Machines	0.56	DeCoste and Scholkopf, MLJ (2002)
K-Nearest Neighbors	0.52	Keysers et al. IEEE PAMI (2007)
Neural Nets	0.35	Cireşan et al. (2010)
Convolutional Neural Nets	0.21	Wang et al. (2013)

Table D.1: Error rates of best performing approaches for several types of classifier on the MNIST dataset (LeCun et al., 2019).

D.2 CIFAR-10 and CIFAR-100

The CIFAR-10 and CIFAR-100 data sets (Krizhevsky et al., 2009) are labeled subsets of the 80 million tiny images data set (Torralba et al., 2008). Both consist of 60,000 color images, each split into 50,000 images for the training set and 10,000 images for the test set. For CIFAR-10 the images are evenly distributed across ten classes (6,000 images per class), namely airplane, automobile, bird, cat, deer, dog, frog, horse, ship and truck. CIFAR-100 distinguishes 100 classes organized within 20 superclasses each containing 5 classes. An overview of the classes and superclasses is given in Table D.2. The CIFAR-100 data set is balanced and therefore contains 500 training images and 100 test images per class. All images in both data sets have a resolution of 32x32 pixels. The top-5 methods on the CIFAR data sets (status 2016) are shown in Table D.3 and D.4.

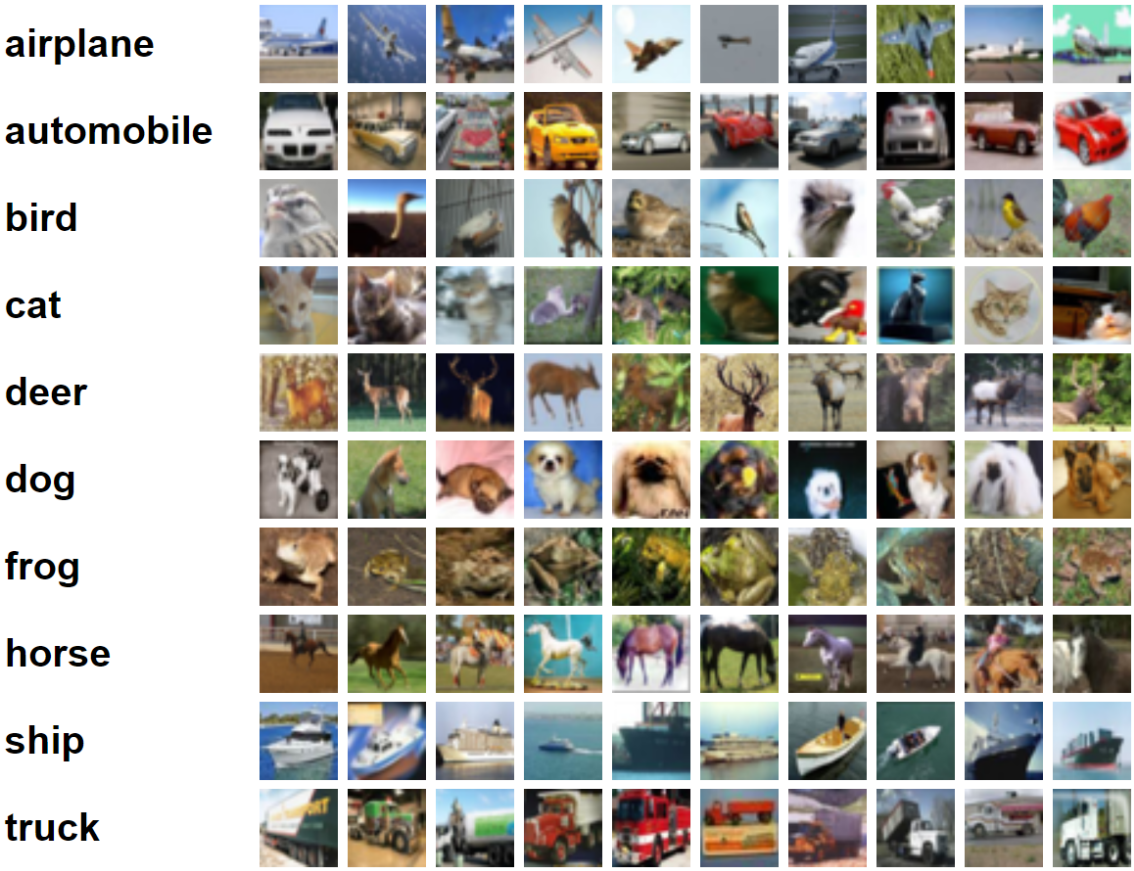


Figure D.2: Examples from the CIFAR-10 data set for each of the 10 classes (Krizhevsky et al., 2019).

Superclass	Classes
aquatic mammals	beaver, dolphin, otter, seal, whale
fish	aquarium fish, flatfish, ray, shark, trout
flowers	orchids, poppies, roses, sunflowers, tulips
food containers	bottles, bowls, cans, cups, plates
fruit and vegetables	apples, mushrooms, oranges, pears, sweet peppers
household electrical devices	clock, computer keyboard, lamp, telephone, television
household furniture	bed, chair, couch, table, wardrobe
insects	bee, beetle, butterfly, caterpillar, cockroach
large carnivores	bear, leopard, lion, tiger, wolf
large man-made outdoor things	bridge, castle, house, road, skyscraper
large natural outdoor scenes	cloud, forest, mountain, plain, sea
large omnivores and herbivores	camel, cattle, chimpanzee, elephant, kangaroo
medium-sized mammals	fox, porcupine, possum, raccoon, skunk
non-insect invertebrates	crab, lobster, snail, spider, worm
people	baby, boy, girl, man, woman
reptiles	crocodile, dinosaur, lizard, snake, turtle
small mammals	hamster, mouse, rabbit, shrew, squirrel
trees	maple, oak, palm, pine, willow
vehicles 1	bicycle, bus, motorcycle, pickup truck, train
vehicles 2	lawn-mower, rocket, streetcar, tank, tractor

Table D.2: CIFAR-100 distinguishes 100 classes grouped into 20 non-overlapping superclasses (Krizhevsky et al., 2019). Each superclass contains 5 classes.

Method	Accuracy (%)	Reference
Fractional Max-Pooling	96.53	Graham (2014a)
Striving for Simplicity: The All Convolutional Net	95.59	Springenberg et al. (2014)
All you need is a good init	94.16	Mishkin and Matas (2015)
Lessons learned from manually classifying CIFAR-10	94	Karpathy (2011)
Generalizing Pooling Functions in Convolutional Neural Networks: Mixed, Gated, and Tree	93.95	Lee et al. (2015)

Table D.3: Prediction accuracy for the top-5 methods on CIFAR-10 (Rodrigo, 2016).

Method	Accuracy (%)	Reference
Fast and Accurate Deep Network Learning by Exponential Linear Units	75.72	Clevert et al. (2015)
Spatially-sparse convolutional neural networks	75.7	Graham (2014b)
Fractional Max-Pooling	73.61	Graham (2014a)
Scalable Bayesian Optimization Using Deep Neural Networks	72.60	Snoek et al. (2015)
Competitive Multi-scale Convolution	72.44	Liao and Carneiro (2015)

Table D.4: Prediction accuracy for the top-5 methods on CIFAR-100 (Rodrigo, 2016).

D.3 ImageNet

ImageNet (Deng et al., 2009) is a large hand-annotated image data set with more than 14 million labeled images in over 20,000 classes published in 2009. The labels are organized according to the WordNet (Miller, 1995) hierarchy, a large lexical database of English (using only nouns), meaning that classes of different specifcity can be chosen. The dog class, for example, can be further broken down into roughly 120 different dog breeds. Of the 14 million images the majority is labeled for classification, while a subset of at least one million images is additionally annotated with bounding box information for object detection. The images in this data set don't all have the same resolution, therefore a common pre-processing step is to rescale all images to the same size. Between 2010 and 2017 the ImageNet Large Scale Visual Recognition Challenge (ILSVRC) was conducted annually, using a subset of roughly 1.2 million images distributed over 1,000 non-overlapping classes. Teams all over the world participated and developed methods with super-human performance in object classification. Table D.5 lists state-of-the-art methods (status October 2019) for the classification task of ILSVRC.

Method	Top 1 Accuracy (%)	Top 5 Accuracy (%)	Reference
Fixing the train-test resolution discrepancy	86.4	98.0	Touvron et al. (2019)
Exploring the Limits of Weakly Supervised Pretraining	85.4	97.6	Mahajan et al. (2018)
RandAugment: Practical data augmentation with no separate search	85.0	97.5	Cubuk et al. (2019)
EfficientNet: Rethinking Model Scaling for Convolutional Neural Networks	84.4	97.2	Tan and Le (2019)
GPipe: Efficient Training of Giant Neural Networks using Pipeline Parallelism	84.3	97.1	Huang et al. (2018)

Table D.5: Prediction accuracy for the top-5 methods on the ILSVRC classification task (ML, 2019).

Appendix E

Arithmetics for convolution operations

This section is based on "A guide to convolution arithmetic for deep learning" by Dumoulin and Visin (2016).

E.1 Discrete convolutions

The bread and butter of neural networks is *affine transformations*: a vector is received as input and is multiplied with a matrix to produce an output (to which a bias vector is usually added before passing the result through a nonlinearity). This is applicable to any type of input, be it an image, a sound clip or an unordered collection of features: whatever their dimensionality, their representation can always be flattened into a vector before the transformation.

Images, sound clips and many other similar kinds of data have an intrinsic structure. More formally, they share these important properties:

- They are stored as multi-dimensional arrays.
- They feature one or more axes for which ordering matters (e.g., width and height axes for an image, time axis for a sound clip).
- One axis, called the channel axis, is used to access different views of the data (e.g., the red, green and blue channels of a color image, or the left and right channels of a stereo audio track).

These properties are not exploited when an affine transformation is applied; in fact, all the axes are treated in the same way and the topological information is not taken into account. Still, taking advantage of the implicit structure of the data may prove very handy in solving some tasks, like computer vision and speech recognition, and in these cases it would be best to preserve it. This is where discrete convolutions come into play.

A discrete convolution is a linear transformation that preserves this notion of ordering. It is sparse (only a few input units contribute to a given output unit) and reuses parameters (the same weights are applied to multiple locations in the input).

Figure E.1 provides an example of a discrete convolution. The light blue grid is called the *input feature map*. To keep the drawing simple, a single input feature map is represented, but it is not uncommon to have multiple feature maps stacked one onto another.[1] A *kernel* (shaded area)

[1] An example of this is what was referred to earlier as *channels* for images and sound clips.

of value

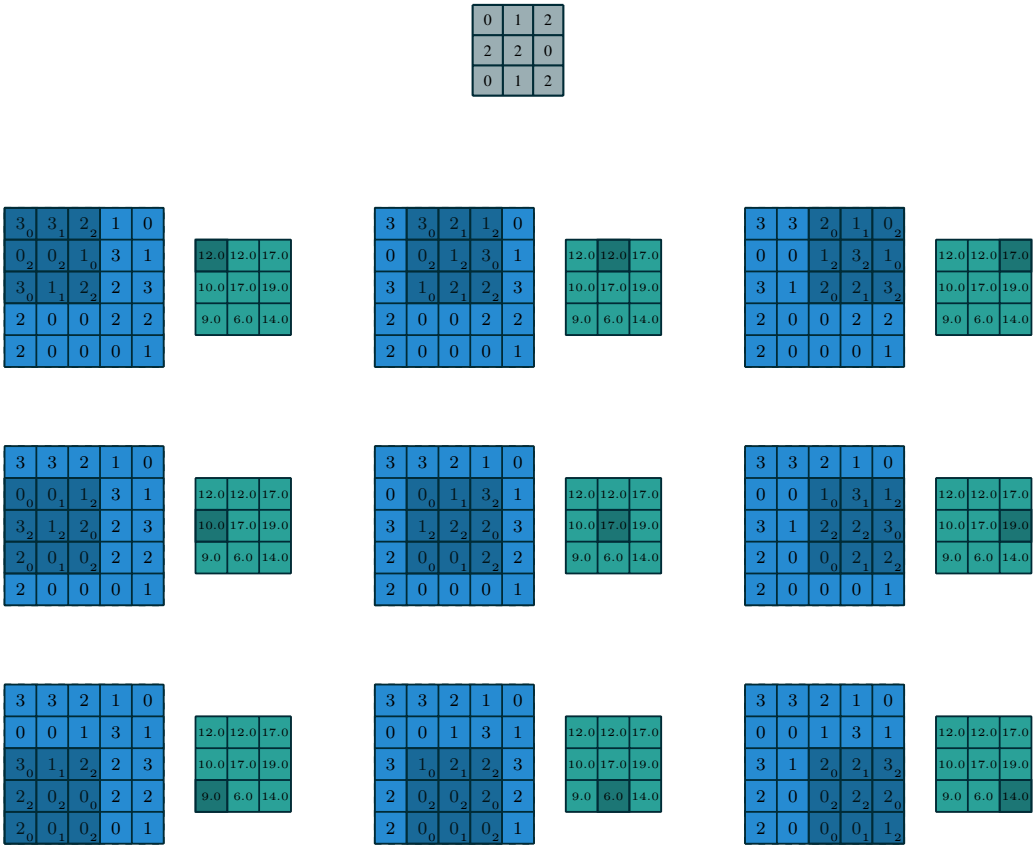


Figure E.1: Computing the output values of a discrete convolution.

slides across the input feature map. At each location, the product between each element of the kernel and the input element it overlaps is computed and the results are summed up to obtain the output in the current location. The procedure can be repeated using different kernels to form as many output feature maps as desired (Figure E.3). The final outputs of this procedure are called *output feature maps*.[2] If there are multiple input feature maps, the kernel will have to be 3-dimensional – or, equivalently each one of the feature maps will be convolved with a distinct kernel – and the resulting feature maps will be summed up elementwise to produce the output feature map.

The convolution depicted in Figure E.1 is an instance of a 2-D convolution, but it can be generalized to N-D convolutions. For instance, in a 3-D convolution, the kernel would be a *cuboid* and would slide across the height, width and depth of the input feature map.

The collection of kernels defining a discrete convolution has a shape corresponding to some permutation of $(n, m, k_1, \ldots, k_N)$, where

[2]While there is a distinction between convolution and cross-correlation from a signal processing perspective, the two become interchangeable when the kernel is learned. For the sake of simplicity and to stay consistent with most of the machine learning literature, the term *convolution* will be used in this guide.

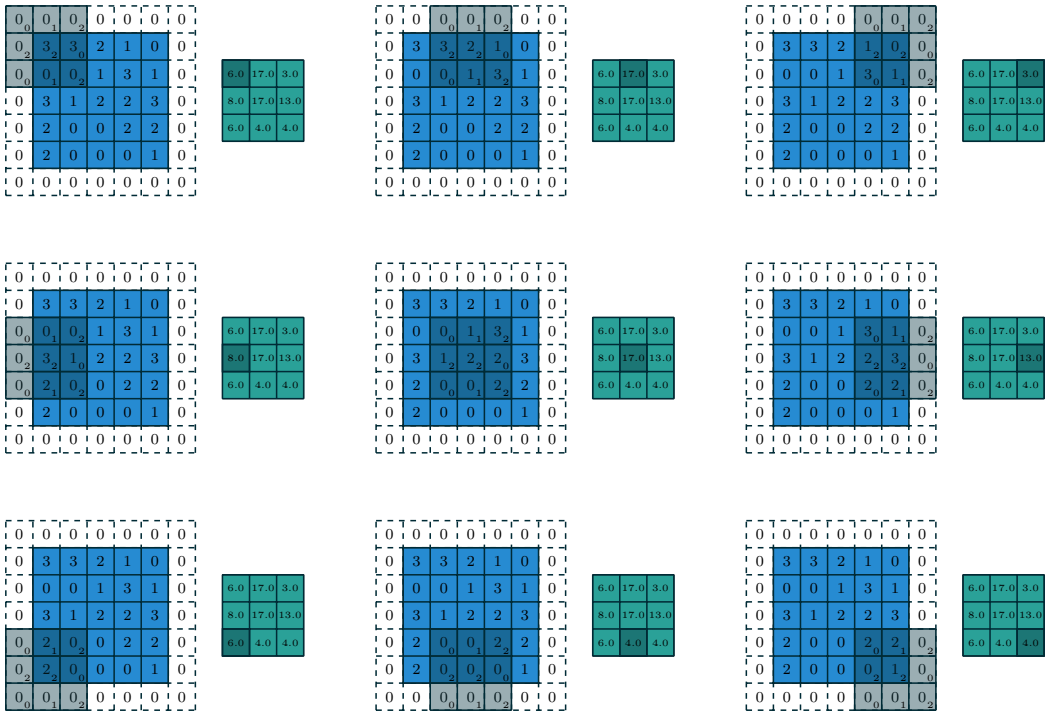


Figure E.2: Computing the output values of a discrete convolution for $N = 2$, $i_1 = i_2 = 5$, $k_1 = k_2 = 3$, $s_1 = s_2 = 2$, and $p_1 = p_2 = 1$.

$$n \equiv \text{number of output feature maps},$$
$$m \equiv \text{number of input feature maps},$$
$$k_j \equiv \text{kernel size along axis } j.$$

The following properties affect the output size o_j of a convolutional layer along axis j:

- i_j: input size along axis j,
- k_j: kernel size along axis j,
- s_j: stride (distance between two consecutive positions of the kernel) along axis j,
- p_j: zero padding (number of zeros concatenated at the beginning and at the end of an axis) along axis j.

For instance, Figure E.2 shows a 3×3 kernel applied to a 5×5 input padded with a 1×1 border of zeros using 2×2 strides.

Note that strides constitute a form of *subsampling*. As an alternative to being interpreted as a measure of how much the kernel is translated, strides can also be viewed as how much of the output is retained. For instance, moving the kernel by hops of two is equivalent to moving the kernel by hops of one but retaining only odd output elements (Figure E.4).

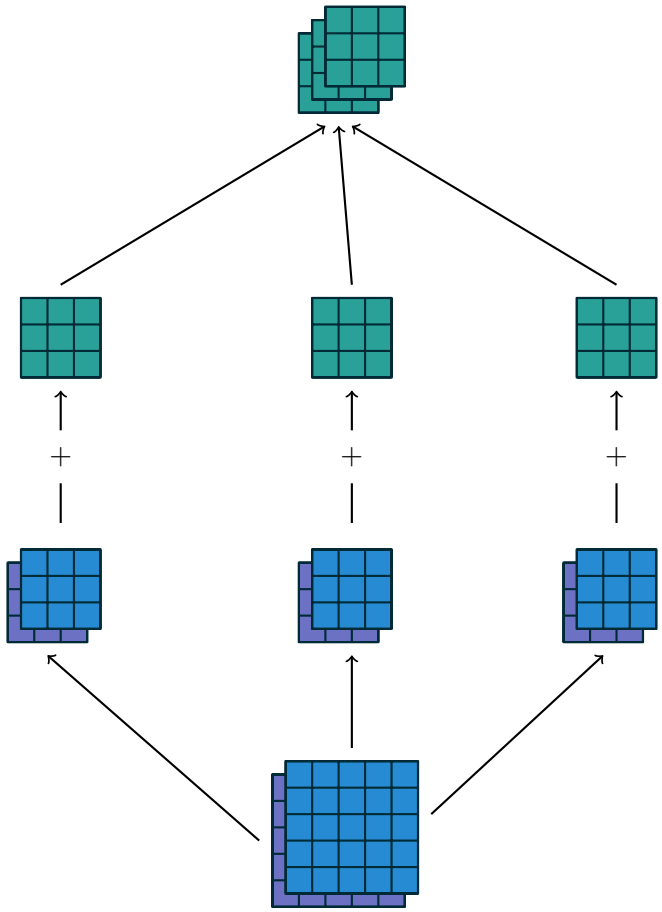


Figure E.3: A convolution mapping from two input feature maps to three output feature maps using a $3 \times 2 \times 3 \times 3$ collection of kernels $\mathbf{w}$. In the left pathway, input feature map 1 is convolved with kernel $\mathbf{w}_{1,1}$ and input feature map 2 is convolved with kernel $\mathbf{w}_{1,2}$, and the results are summed together elementwise to form the first output feature map. The same is repeated for the middle and right pathways to form the second and third feature maps, and all three output feature maps are grouped together to form the output.

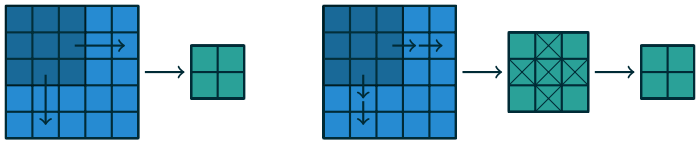


Figure E.4: An alternative way of viewing strides. Instead of translating the 3×3 kernel by increments of $s = 2$ (left), the kernel is translated by increments of 1 and only one in $s = 2$ output elements is retained (right).

E.2 Pooling

In addition to discrete convolutions themselves, *pooling* operations make up another important building block in CNNs. Pooling operations reduce the size of feature maps by using some function to summarize subregions, such as taking the average or the maximum value.

Pooling works by sliding a window across the input and feeding the content of the window to a *pooling function*. In some sense, pooling works very much like a discrete convolution, but replaces the linear combination described by the kernel with some other function. Figure E.5 provides an example for average pooling, and Figure E.6 does the same for max pooling.

The following properties affect the output size o_j of a pooling layer along axis j:

- i_j: input size along axis j,
- k_j: pooling window size along axis j,
- s_j: stride (distance between two consecutive positions of the pooling window) along axis j.

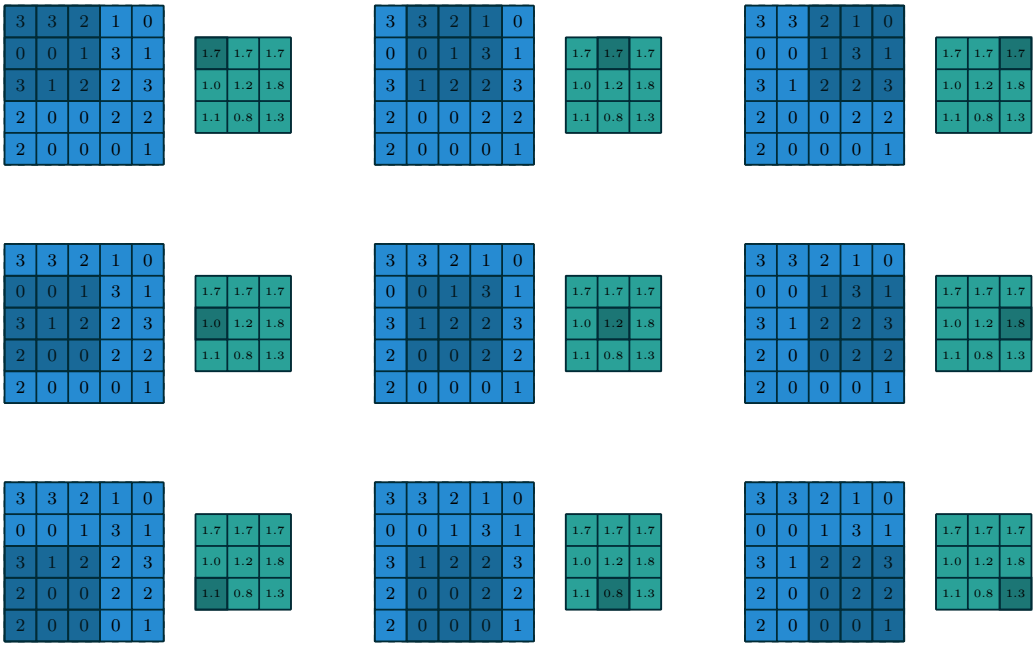


Figure E.5: Computing the output values of a 3×3 average pooling operation on a 5×5 input using 1×1 strides.

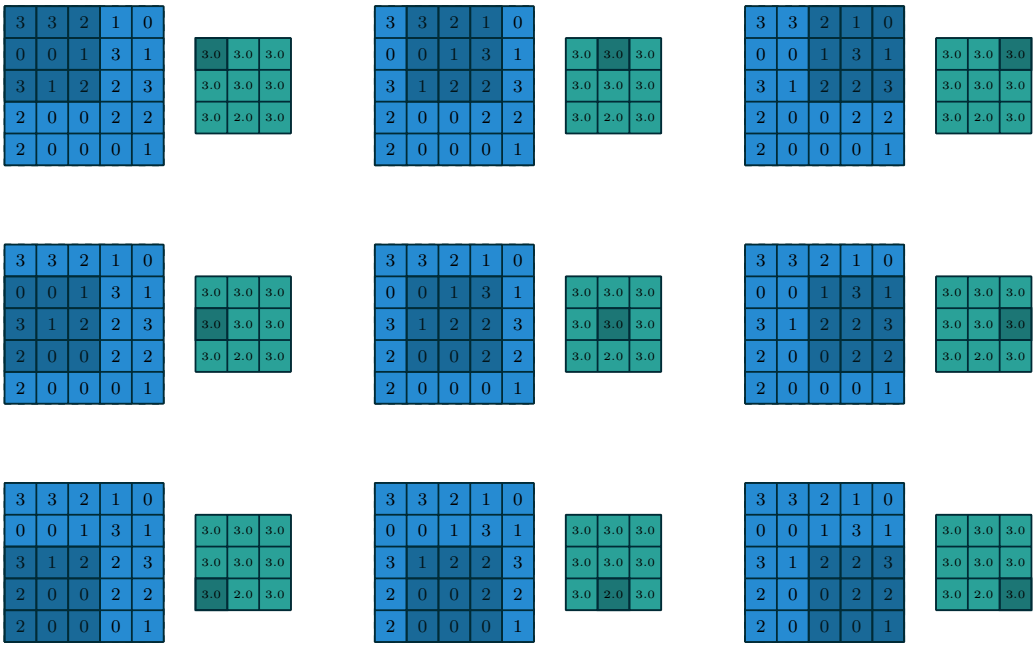


Figure E.6: Computing the output values of a 3×3 max pooling operation on a 5×5 input using 1×1 strides.

E.3 Convolution arithmetic

The analysis of the relationship between convolutional layer properties is eased by the fact that they don't interact across axes, i.e., the choice of kernel size, stride and zero padding along axis j only affects the output size of axis j. Because of that, this chapter will focus on the following simplified setting:

- 2-D discrete convolutions ($N = 2$),
- square inputs ($i_1 = i_2 = i$),
- square kernel size ($k_1 = k_2 = k$),
- same strides along both axes ($s_1 = s_2 = s$),
- same zero padding along both axes ($p_1 = p_2 = p$).

This facilitates the analysis and the visualization, but keep in mind that the results outlined here also generalize to the N-D and non-square cases.

E.3.1 No zero padding, unit strides

The simplest case to analyze is when the kernel just slides across every position of the input (i.e., $s = 1$ and $p = 0$). Figure E.7 provides an example for $i = 4$ and $k = 3$.

One way of defining the output size in this case is by the number of possible placements of the kernel on the input. Let's consider the width axis: the kernel starts on the leftmost part of the input feature map and slides by steps of one until it touches the right side of the input. The size of the output will be equal to the number of steps made, plus one, accounting for the initial position of the kernel (Figure E.14a). The same logic applies for the height axis.

More formally, the following relationship can be inferred:

Relationship 1. *For any i and k, and for $s = 1$ and $p = 0$,*

$$o = (i - k) + 1.$$

E.3.2 Zero padding, unit strides

To factor in zero padding (i.e., only restricting to $s = 1$), let's consider its effect on the effective input size: padding with p zeros changes the effective input size from i to $i + 2p$. In the general case, Relationship 1 can then be used to infer the following relationship:

Relationship 2. *For any i, k and p, and for $s = 1$,*

$$o = (i - k) + 2p + 1.$$

Figure E.8 provides an example for $i = 5$, $k = 4$ and $p = 2$.

In practice, two specific instances of zero padding are used quite extensively because of their respective properties. Let's discuss them in more detail.

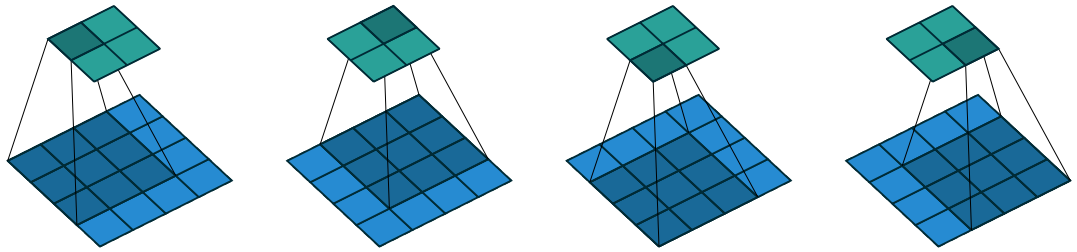


Figure E.7: (No padding, unit strides) Convolving a 3×3 kernel over a 4×4 input using unit strides (i.e., $i = 4$, $k = 3$, $s = 1$ and $p = 0$).

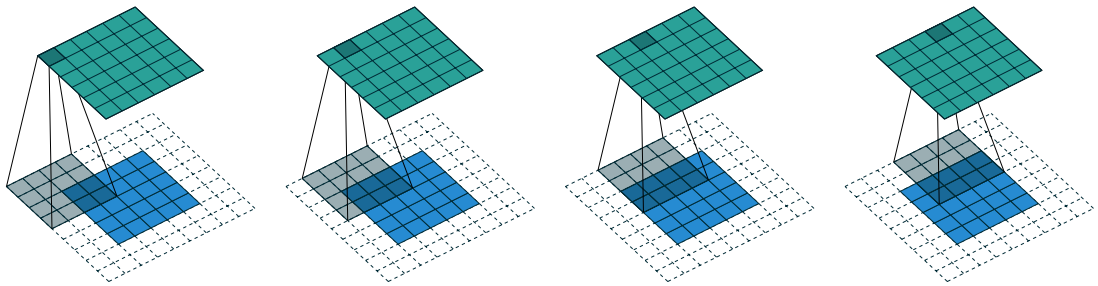


Figure E.8: (Arbitrary padding, unit strides) Convolving a 4×4 kernel over a 5×5 input padded with a 2×2 border of zeros using unit strides (i.e., $i = 5$, $k = 4$, $s = 1$ and $p = 2$).

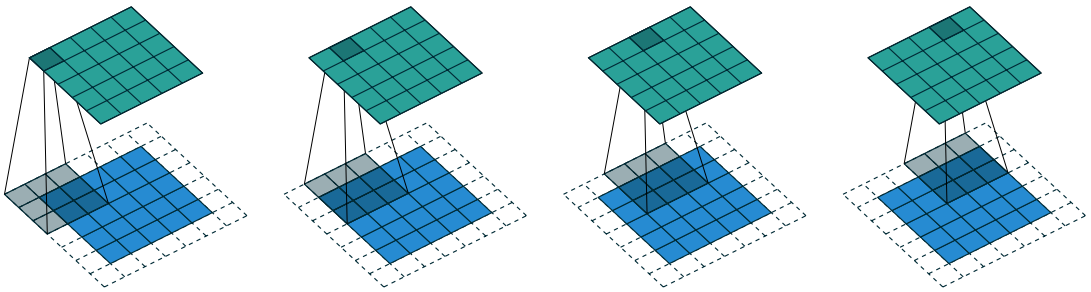


Figure E.9: (Half padding, unit strides) Convolving a 3×3 kernel over a 5×5 input using half padding and unit strides (i.e., $i = 5$, $k = 3$, $s = 1$ and $p = 1$).

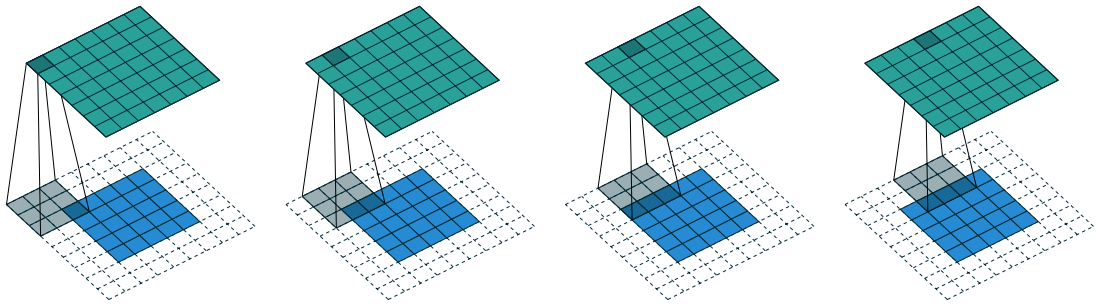


Figure E.10: (Full padding, unit strides) Convolving a 3×3 kernel over a 5×5 input using full padding and unit strides (i.e., $i = 5$, $k = 3$, $s = 1$ and $p = 2$).

E.3.2.1 Half (same) padding

Having the output size be the same as the input size (i.e., $o = i$) can be a desirable property:

Relationship 3. *For any i and for k odd ($k = 2n + 1, \quad n \in \mathbb{N}$), $s = 1$ and $p = \lfloor k/2 \rfloor = n$,*

$$\begin{aligned} o &= i + 2\lfloor k/2 \rfloor - (k - 1) \\ &= i + 2n - 2n \\ &= i. \end{aligned}$$

This is sometimes referred to as *half* (or *same*) padding. Figure E.9 provides an example for $i = 5$, $k = 3$ and (therefore) $p = 1$.

E.3.2.2 Full padding

While convolving a kernel generally *decreases* the output size with respect to the input size, sometimes the opposite is required. This can be achieved with proper zero padding:

Relationship 4. *For any i and k, and for $p = k - 1$ and $s = 1$,*

$$\begin{aligned} o &= i + 2(k - 1) - (k - 1) \\ &= i + (k - 1). \end{aligned}$$

This is sometimes referred to as *full* padding, because in this setting every possible partial or complete superimposition of the kernel on the input feature map is taken into account. Figure E.10 provides an example for $i = 5$, $k = 3$ and (therefore) $p = 2$.

E.3.3 No zero padding, non-unit strides

All relationships derived so far only apply for unit-strided convolutions. Incorporating non unitary strides requires another inference leap. To facilitate the analysis, let's momentarily ignore zero padding (i.e., $s > 1$ and $p = 0$). Figure E.11 provides an example for $i = 5$, $k = 3$ and $s = 2$.

Once again, the output size can be defined in terms of the number of possible placements of the kernel on the input. Let's consider the width axis: the kernel starts as usual on the leftmost part of the input, but this time it slides by steps of size s until it touches the right side of the input. The size of the output is again equal to the number of steps made, plus one, accounting for the initial position of the kernel (Figure E.14b). The same logic applies for the height axis.

From this, the following relationship can be inferred:

Relationship 5. *For any i, k and s, and for $p = 0$,*

$$o = \left\lfloor \frac{i - k}{s} \right\rfloor + 1.$$

The floor function accounts for the fact that sometimes the last possible step does *not* coincide with the kernel reaching the end of the input, i.e., some input units are left out (see Figure E.13 for an example of such a case).

E.3.4 Zero padding, non-unit strides

The most general case (convolving over a zero padded input using non-unit strides) can be derived by applying Relationship 5 on an effective input of size $i + 2p$, in analogy to what was done for Relationship 2:

Relationship 6. *For any i, k, p and s,*

$$o = \left\lfloor \frac{i + 2p - k}{s} \right\rfloor + 1.$$

As before, the floor function means that in some cases a convolution will produce the same output size for multiple input sizes. More specifically, if $i + 2p - k$ is a multiple of s, then any input size $j = i + a, \quad a \in \{0, \ldots, s - 1\}$ will produce the same output size. Note that this ambiguity applies only for $s > 1$.

Figure E.12 shows an example with $i = 5$, $k = 3$, $s = 2$ and $p = 1$, while Figure E.13 provides an example for $i = 6$, $k = 3$, $s = 2$ and $p = 1$. Interestingly, despite having different input sizes these convolutions share the same output size. While this doesn't affect the analysis for *convolutions*, this will complicate the analysis in the case of *transposed convolutions*.

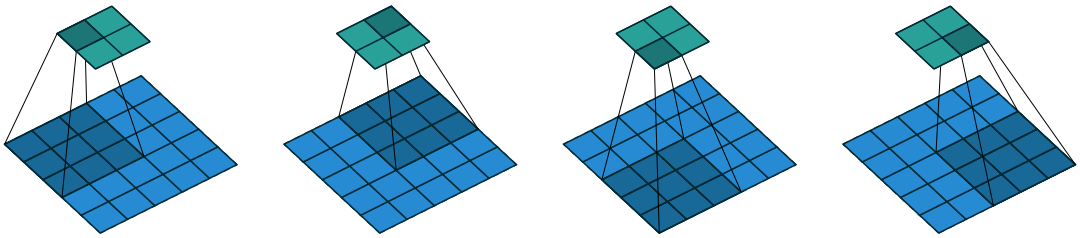


Figure E.11: (No zero padding, arbitrary strides) Convolving a 3×3 kernel over a 5×5 input using 2×2 strides (i.e., $i = 5$, $k = 3$, $s = 2$ and $p = 0$).

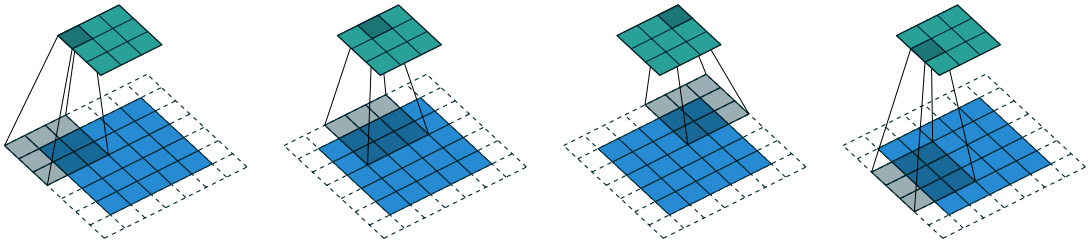


Figure E.12: (Arbitrary padding and strides) Convolving a 3×3 kernel over a 5×5 input padded with a 1×1 border of zeros using 2×2 strides (i.e., $i = 5$, $k = 3$, $s = 2$ and $p = 1$).

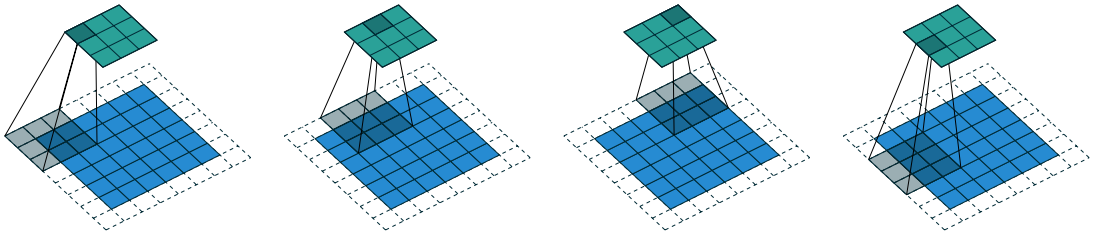
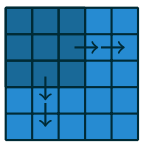
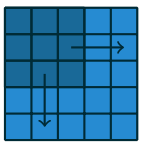


Figure E.13: (Arbitrary padding and strides) Convolving a 3×3 kernel over a 6×6 input padded with a 1×1 border of zeros using 2×2 strides (i.e., $i = 6$, $k = 3$, $s = 2$ and $p = 1$). In this case, the bottom row and right column of the zero padded input are not covered by the kernel.



(a) The kernel has to slide two steps to the right to touch the right side of the input (and equivalently downwards). Adding one to account for the initial kernel position, the output size is 3×3.



(b) The kernel has to slide one step of size two to the right to touch the right side of the input (and equivalently downwards). Adding one to account for the initial kernel position, the output size is 2×2.

Figure E.14: Counting kernel positions.

E.4 Pooling arithmetic

In a neural network, pooling layers provide invariance to small translations of the input. The most common kind of pooling is *max pooling*, which consists in splitting the input in (usually non-overlapping) patches and outputting the maximum value of each patch. Other kinds of pooling exist, e.g., mean or average pooling, which all share the same idea of aggregating the input locally by applying a non-linearity to the content of some patches.

Some readers may have noticed that the treatment of convolution arithmetic only relies on the assumption that some function is repeatedly applied onto subsets of the input. This means that the relationships derived in the previous chapter can be reused in the case of pooling arithmetic. Since pooling does not involve zero padding, the relationship describing the general case is as follows:

Relationship 7. *For any i, k and s,*

$$o = \left\lfloor \frac{i - k}{s} \right\rfloor + 1.$$

This relationship holds for any type of pooling.

E.5 Transposed convolution arithmetic

The need for transposed convolutions generally arises from the desire to use a transformation going in the opposite direction of a normal convolution, i.e., from something that has the shape of the output of some convolution to something that has the shape of its input while maintaining a connectivity pattern that is compatible with said convolution. For instance, one might use such a transformation as the decoding layer of a convolutional autoencoder or to project feature maps to a higher-dimensional space.

Once again, the convolutional case is considerably more complex than the fully-connected case, which only requires to use a weight matrix whose shape has been transposed. However, since every convolution boils down to an efficient implementation of a matrix operation, the insights gained from the fully-connected case are useful in solving the convolutional case.

Like for convolution arithmetic, the dissertation about transposed convolution arithmetic is simplified by the fact that transposed convolution properties don't interact across axes.

The chapter will focus on the following setting:

- 2-D transposed convolutions ($N = 2$),
- square inputs ($i_1 = i_2 = i$),
- square kernel size ($k_1 = k_2 = k$),
- same strides along both axes ($s_1 = s_2 = s$),
- same zero padding along both axes ($p_1 = p_2 = p$).

Once again, the results outlined generalize to the N-D and non-square cases.

E.5.1 Convolution as a matrix operation

Take for example the convolution represented in Figure E.7. If the input and output were to be unrolled into vectors from left to right, top to bottom, the convolution could be represented as a sparse matrix $\boldsymbol{W}$ where the non-zero elements are the elements $w_{i,j}$ of the kernel (with i and j being the row and column of the kernel respectively):

$$\begin{pmatrix} w_{0,0} & w_{0,1} & w_{0,2} & 0 & w_{1,0} & w_{1,1} & w_{1,2} & 0 & w_{2,0} & w_{2,1} & w_{2,2} & 0 & 0 & 0 & 0 & 0 \\ 0 & w_{0,0} & w_{0,1} & w_{0,2} & 0 & w_{1,0} & w_{1,1} & w_{1,2} & 0 & w_{2,0} & w_{2,1} & w_{2,2} & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & w_{0,0} & w_{0,1} & w_{0,2} & 0 & w_{1,0} & w_{1,1} & w_{1,2} & 0 & w_{2,0} & w_{2,1} & w_{2,2} & 0 \\ 0 & 0 & 0 & 0 & 0 & w_{0,0} & w_{0,1} & w_{0,2} & 0 & w_{1,0} & w_{1,1} & w_{1,2} & 0 & w_{2,0} & w_{2,1} & w_{2,2} \end{pmatrix}$$

This linear operation takes the input matrix flattened as a 16-dimensional vector and produces a 4-dimensional vector that is later reshaped as the 2×2 output matrix.

Using this representation, the backward pass is easily obtained by transposing $\boldsymbol{W}$; in other words, the error is backpropagated by multiplying the loss with $\boldsymbol{W}^T$. This operation takes a 4-dimensional vector as input and produces a 16-dimensional vector as output, and its connectivity pattern is compatible with $\boldsymbol{W}$ by construction.

Notably, the kernel $\mathbf{w}$ defines both the matrices $\boldsymbol{W}$ and $\boldsymbol{W}^T$ used for the forward and backward passes.

E.5.2 Transposed convolution

Let's now consider what would be required to go the other way around, i.e., map from a 4-dimensional space to a 16-dimensional space, while keeping the connectivity pattern of the convolution depicted in Figure E.7. This operation is known as a *transposed convolution*.

Transposed convolutions – also called *fractionally strided convolutions* or *deconvolutions*[3] – work by swapping the forward and backward passes of a convolution. One way to put it is to note that the kernel defines a convolution, but whether it's a direct convolution or a transposed convolution is determined by how the forward and backward passes are computed.

For instance, although the kernel $\mathbf{w}$ defines a convolution whose forward and backward passes are computed by multiplying with $\boldsymbol{W}$ and $\boldsymbol{W}^T$ respectively, it *also* defines a transposed convolution whose forward and backward passes are computed by multiplying with $\boldsymbol{W}^T$ and $(\boldsymbol{W}^T)^T = \boldsymbol{W}$ respectively.[4]

Finally note that it is always possible to emulate a transposed convolution with a direct convolution. The disadvantage is that it usually involves adding many columns and rows of zeros to the input, resulting in a much less efficient implementation.

Building on what has been introduced so far, this chapter will proceed somewhat backwards with respect to the convolution arithmetic chapter, deriving the properties of each transposed convolution by referring to the direct convolution with which it shares the kernel, and defining the equivalent direct convolution.

E.5.3 No zero padding, unit strides, transposed

The simplest way to think about a transposed convolution on a given input is to imagine such an input as being the result of a direct convolution applied on some initial feature map. The trasposed convolution can be then considered as the operation that allows to recover the *shape* [5] of this initial feature map.

Let's consider the convolution of a 3×3 kernel on a 4×4 input with unitary stride and no padding (i.e., $i = 4$, $k = 3$, $s = 1$ and $p = 0$). As depicted in Figure E.7, this produces a 2×2 output. The transpose of this convolution will then have an output of shape 4×4 when applied on a 2×2 input.

Another way to obtain the result of a transposed convolution is to apply an equivalent – but much less efficient – direct convolution. The example described so far could be tackled by convolving a 3×3 kernel over a 2×2 input padded with a 2×2 border of zeros using unit strides (i.e., $i' = 2$, $k' = k$, $s' = 1$ and $p' = 2$), as shown in Figure E.15. Notably, the kernel's and stride's sizes remain the same, but the input of the transposed convolution is now zero padded.[6]

[3] The term "deconvolution" is sometimes used in the literature, but we advocate against it on the grounds that a deconvolution is mathematically defined as the inverse of a convolution, which is different from a transposed convolution.

[4] The transposed convolution operation can be thought of as the gradient of *some* convolution with respect to its input, which is usually how transposed convolutions are implemented in practice.

[5] Note that the transposed convolution does not guarantee to recover the input itself, as it is not defined as the inverse of the convolution, but rather just returns a feature map that has the same width and height.

[6] Note that although equivalent to applying the transposed matrix, this visualization adds a lot of zero multiplications in the form of zero padding. This is done here for illustration purposes, but it is inefficient, and software implementations will normally not perform the useless zero multiplications.

One way to understand the logic behind zero padding is to consider the connectivity pattern of the transposed convolution and use it to guide the design of the equivalent convolution. For example, the top left pixel of the input of the direct convolution only contribute to the top left pixel of the output, the top right pixel is only connected to the top right output pixel, and so on.

To maintain the same connectivity pattern in the equivalent convolution it is necessary to zero pad the input in such a way that the first (top-left) application of the kernel only touches the top-left pixel, i.e., the padding has to be equal to the size of the kernel minus one.

Proceeding in the same fashion it is possible to determine similar observations for the other elements of the image, giving rise to the following relationship:

Relationship 8. *A convolution described by $s = 1$, $p = 0$ and k has an associated transposed convolution described by $k' = k$, $s' = s$ and $p' = k - 1$ and its output size is*

$$o' = i' + (k - 1).$$

Interestingly, this corresponds to a fully padded convolution with unit strides.

E.5.4 Zero padding, unit strides, transposed

Knowing that the transpose of a non-padded convolution is equivalent to convolving a zero padded input, it would be reasonable to suppose that the transpose of a zero padded convolution is equivalent to convolving an input padded with *less* zeros.

It is indeed the case, as shown in Figure E.16 for $i = 5$, $k = 4$ and $p = 2$.

Formally, the following relationship applies for zero padded convolutions:

Relationship 9. *A convolution described by $s = 1$, k and p has an associated transposed convolution described by $k' = k$, $s' = s$ and $p' = k - p - 1$ and its output size is*

$$o' = i' + (k - 1) - 2p.$$

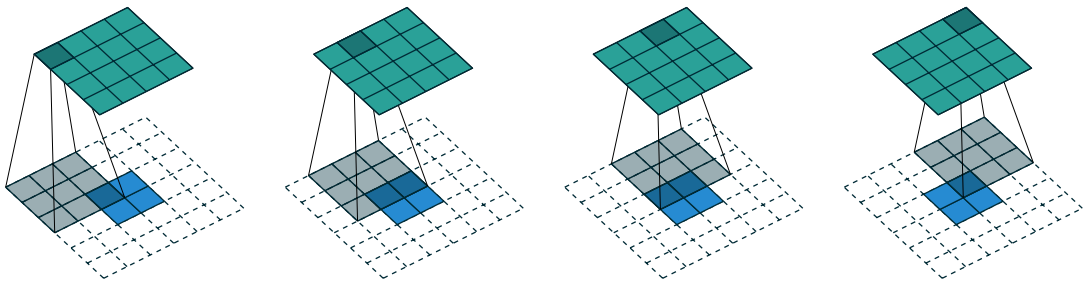


Figure E.15: The transpose of convolving a 3×3 kernel over a 4×4 input using unit strides (i.e., $i = 4$, $k = 3$, $s = 1$ and $p = 0$). It is equivalent to convolving a 3×3 kernel over a 2×2 input padded with a 2×2 border of zeros using unit strides (i.e., $i' = 2$, $k' = k$, $s' = 1$ and $p' = 2$).

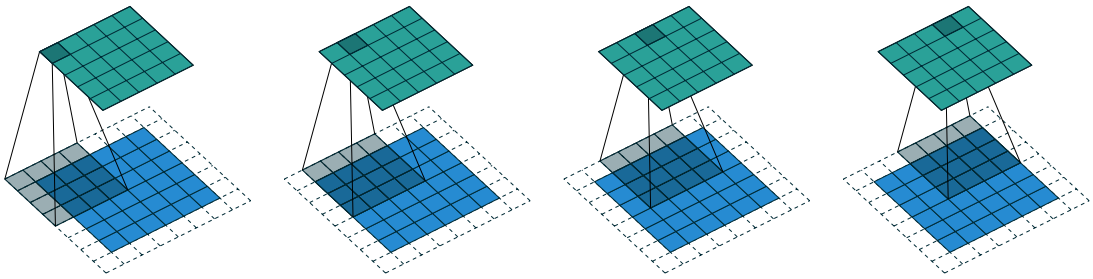


Figure E.16: The transpose of convolving a 4×4 kernel over a 5×5 input padded with a 2×2 border of zeros using unit strides (i.e., $i = 5$, $k = 4$, $s = 1$ and $p = 2$). It is equivalent to convolving a 4×4 kernel over a 6×6 input padded with a 1×1 border of zeros using unit strides (i.e., $i' = 6$, $k' = k$, $s' = 1$ and $p' = 1$).

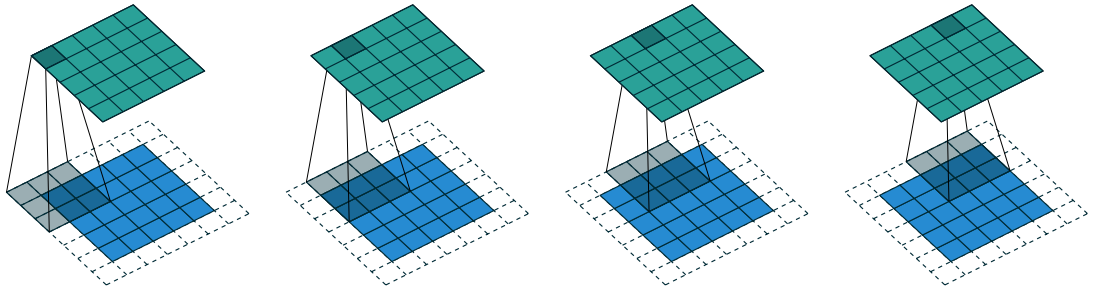


Figure E.17: The transpose of convolving a 3×3 kernel over a 5×5 input using half padding and unit strides (i.e., $i = 5$, $k = 3$, $s = 1$ and $p = 1$). It is equivalent to convolving a 3×3 kernel over a 5×5 input using half padding and unit strides (i.e., $i' = 5$, $k' = k$, $s' = 1$ and $p' = 1$).

E.5.4.1 Half (same) padding, transposed

By applying the same inductive reasoning as before, it is reasonable to expect that the equivalent convolution of the transpose of a half padded convolution is itself a half padded convolution, given that the output size of a half padded convolution is the same as its input size. Thus the following relation applies:

Relationship 10. *A convolution described by* $k = 2n + 1, \quad n \in \mathbb{N}$, $s = 1$ *and* $p = \lfloor k/2 \rfloor = n$ *has an associated transposed convolution described by* $k' = k$, $s' = s$ *and* $p' = p$ *and its output size is*

$$\begin{aligned} o' &= i' + (k - 1) - 2p \\ &= i' + 2n - 2n \\ &= i'. \end{aligned}$$

Figure E.17 provides an example for $i = 5$, $k = 3$ and (therefore) $p = 1$.

E.5.4.2 Full padding, transposed

Knowing that the equivalent convolution of the transpose of a non-padded convolution involves full padding, it is unsurprising that the equivalent of the transpose of a fully padded convolution is a non-padded convolution:

Relationship 11. *A convolution described by* $s = 1$, k *and* $p = k - 1$ *has an associated transposed convolution described by* $k' = k$, $s' = s$ *and* $p' = 0$ *and its output size is*

$$\begin{aligned} o' &= i' + (k - 1) - 2p \\ &= i' - (k - 1) \end{aligned}$$

Figure E.18 provides an example for $i = 5$, $k = 3$ and (therefore) $p = 2$.

E.5.5 No zero padding, non-unit strides, transposed

Using the same kind of inductive logic as for zero padded convolutions, one might expect that the transpose of a convolution with $s > 1$ involves an equivalent convolution with $s < 1$. As will be explained, this is a valid intuition, which is why transposed convolutions are sometimes called *fractionally strided convolutions*.

Figure E.19 provides an example for $i = 5$, $k = 3$ and $s = 2$ which helps understand what fractional strides involve: zeros are inserted *between* input units, which makes the kernel move around at a slower pace than with unit strides.[7]

For the moment, it will be assumed that the convolution is non-padded ($p = 0$) and that its input size i is such that $i - k$ is a multiple of s. In that case, the following relationship holds:

[7] Doing so is inefficient and real-world implementations avoid useless multiplications by zero, but conceptually it is how the transpose of a strided convolution can be thought of.

Relationship 12. *A convolution described by $p = 0$, k and s and whose input size is such that $i - k$ is a multiple of s, has an associated transposed convolution described by $\tilde{i}'$, $k' = k$, $s' = 1$ and $p' = k - 1$, where $\tilde{i}'$ is the size of the stretched input obtained by adding $s - 1$ zeros between each input unit, and its output size is*

$$o' = s(i' - 1) + k.$$

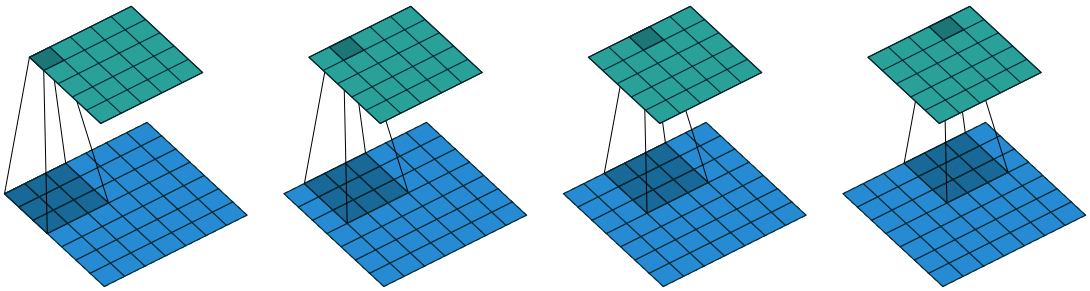


Figure E.18: The transpose of convolving a 3×3 kernel over a 5×5 input using full padding and unit strides (i.e., $i = 5$, $k = 3$, $s = 1$ and $p = 2$). It is equivalent to convolving a 3×3 kernel over a 7×7 input using unit strides (i.e., $i' = 7$, $k' = k$, $s' = 1$ and $p' = 0$).

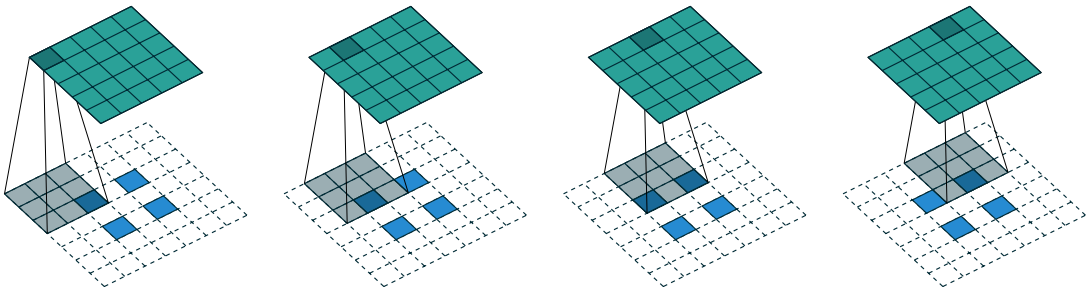


Figure E.19: The transpose of convolving a 3×3 kernel over a 5×5 input using 2×2 strides (i.e., $i = 5$, $k = 3$, $s = 2$ and $p = 0$). It is equivalent to convolving a 3×3 kernel over a 2×2 input (with 1 zero inserted between inputs) padded with a 2×2 border of zeros using unit strides (i.e., $i' = 2$, $\tilde{i}' = 3$, $k' = k$, $s' = 1$ and $p' = 2$).

E.5.6 Zero padding, non-unit strides, transposed

When the convolution's input size i is such that $i + 2p - k$ is a multiple of s, the analysis can extended to the zero padded case by combining Relationship 9 and Relationship 12:

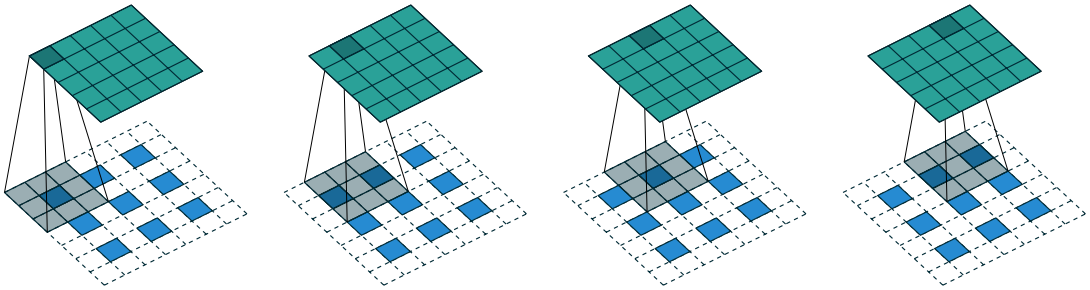


Figure E.20: The transpose of convolving a 3×3 kernel over a 5×5 input padded with a 1×1 border of zeros using 2×2 strides (i.e., $i = 5$, $k = 3$, $s = 2$ and $p = 1$). It is equivalent to convolving a 3×3 kernel over a 3×3 input (with 1 zero inserted between inputs) padded with a 1×1 border of zeros using unit strides (i.e., $i' = 3$, $\tilde{i}' = 5$, $k' = k$, $s' = 1$ and $p' = 1$).

Relationship 13. *A convolution described by k, s and p and whose input size i is such that $i + 2p - k$ is a multiple of s has an associated transposed convolution described by $\tilde{i}'$, $k' = k$, $s' = 1$ and $p' = k - p - 1$, where $\tilde{i}'$ is the size of the stretched input obtained by adding $s - 1$ zeros between each input unit, and its output size is*

$$o' = s(i' - 1) + k - 2p.$$

Figure E.20 provides an example for $i = 5$, $k = 3$, $s = 2$ and $p = 1$.

The constraint on the size of the input i can be relaxed by introducing another parameter $a \in \{0, \ldots, s - 1\}$ that allows to distinguish between the s different cases that all lead to the same i':

Relationship 14. *A convolution described by k, s and p has an associated transposed convolution described by a, $\tilde{i}'$, $k' = k$, $s' = 1$ and $p' = k - p - 1$, where $\tilde{i}'$ is the size of the stretched input obtained by adding $s - 1$ zeros between each input unit, and $a = (i + 2p - k) \mod s$ represents the number of zeros added to the bottom and right edges of the input, and its output size is*

$$o' = s(i' - 1) + a + k - 2p.$$

Figure E.21 provides an example for $i = 6$, $k = 3$, $s = 2$ and $p = 1$.

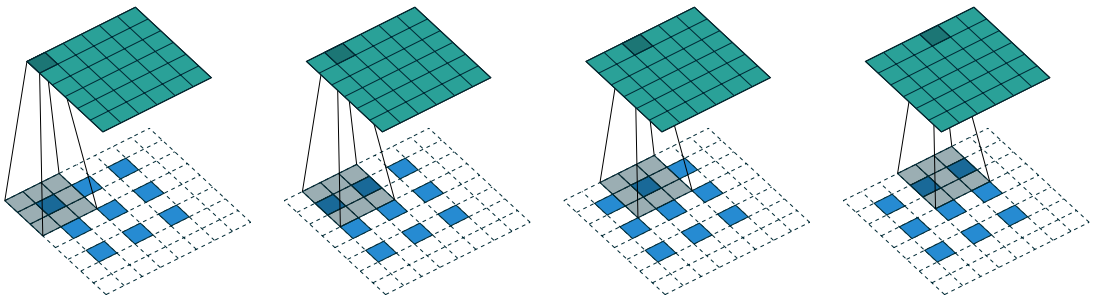


Figure E.21: The transpose of convolving a 3×3 kernel over a 6×6 input padded with a 1×1 border of zeros using 2×2 strides (i.e., $i = 6$, $k = 3$, $s = 2$ and $p = 1$). It is equivalent to convolving a 3×3 kernel over a 2×2 input (with 1 zero inserted between inputs) padded with a 1×1 border of zeros (with an additional border of size 1 added to the bottom and right edges) using unit strides (i.e., $i' = 3$, $\tilde{i}' = 5$, $a = 1$, $k' = k$, $s' = 1$ and $p' = 1$).

E.6 Miscellaneous convolutions

E.6.1 Dilated convolutions

Readers familiar with the deep learning literature may have noticed the term "dilated convolutions" (or "atrous convolutions", from the French expression *convolutions à trous*) appear in recent papers. Here we attempt to provide an intuitive understanding of dilated convolutions.

Dilated convolutions "inflate" the kernel by inserting spaces between the kernel elements. The dilation "rate" is controlled by an additional hyperparameter d. Implementations may vary, but there are usually $d - 1$ spaces inserted between kernel elements such that $d = 1$ corresponds to a regular convolution.

Dilated convolutions are used to cheaply increase the receptive field of output units without increasing the kernel size, which is especially effective when multiple dilated convolutions are stacked one after another.

To understand the relationship tying the dilation rate d and the output size o, it is useful to think of the impact of d on the *effective kernel size*. A kernel of size k dilated by a factor d has an effective size

$$\hat{k} = k + (k - 1)(d - 1).$$

This can be combined with Relationship 6 to form the following relationship for dilated convolutions:

Relationship 15. *For any i, k, p and s, and for a dilation rate d,*

$$o = \left\lfloor \frac{i + 2p - k - (k - 1)(d - 1)}{s} \right\rfloor + 1.$$

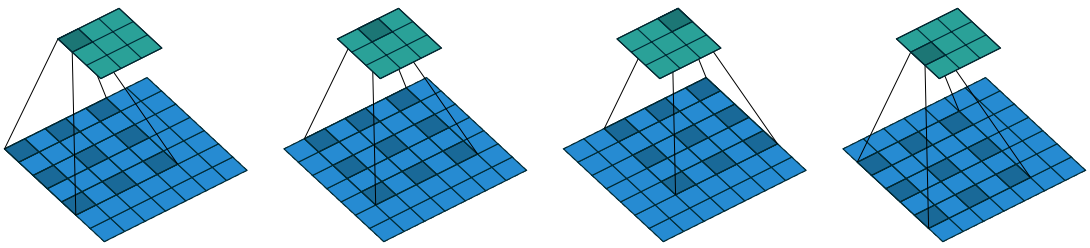


Figure E.22: (Dilated convolution) Convolving a 3×3 kernel over a 7×7 input with a dilation factor of 2 (i.e., $i = 7$, $k = 3$, $d = 2$, $s = 1$ and $p = 0$).

Figure E.22 provides an example for $i = 7$, $k = 3$ and $d = 2$.

Bibliography

Ackley, D. H., Hinton, G. E., and Sejnowski, T. J. (1985). A learning algorithm for Boltzmann machines. *Cognitive Science*, 9:147–169.

Ba, J. L., Kiros, J. R., and Hinton, G. E. (2016). Layer normalization. *arXiv preprint arXiv:1607.06450*.

Barto, A. G. and Sutton, R. S. (1981). Goal seeking components for adaptive intelligence: An initial assessment. Technical report, University of Massachusetts Amherst, Department of Computer and Information Science.

Bishop, C. M. (1993). Curvature-driven smoothing: A learning algorithm for feed-forward networks. *IEEE Transactions on Neural Networks*, 4(5):882–884.

Bishop, C. M. (1995). *Neural Networks for Pattern Recognition*. Clarendon Press, Oxford.

Bliss, T. V. and Lømo, T. (1973). Long-lasting potentiation of synaptic transmission in the dentate area of the anaesthetized rabbit following stimulation of the perforant path. *The Journal of physiology*, 232(2):331–356.

Bradley, R. C. (1981). Central limit theorems under weak dependence. *Journal of Multivariate Analysis*, 11(1):1–16.

Carter, M. J., Rudolph, F. J., and Nucci, A. J. (1990). Operational fault tolerance of CMAC networks. In Touretzky, D. S., editor, *Advances in Neural Information Processing Systems 2*, pages 340–347. San Mateo, CA: Morgan Kaufmann.

Caruana, R. (1997). Multitask learning. *Machine Learning*, 28(1):41–75.

Caruana, R. A. (1993). Multitask learning: A knowledge-based source of inductive bias. In *Proceedings of the tenth international conference on machine learning*, pages 41–48. University of Massachusetts.

Chaudhari, P. and Soatto, S. (2018). Stochastic gradient descent performs variational inference, converges to limit cycles for deep networks. In *2018 Information Theory and Applications Workshop (ITA)*, pages 1–10. IEEE.

Ciresan, D. C., Meier, U., Gambardella, L. M., and Schmidhuber, J. (2011a). Convolutional neural network committees for handwritten character classification. In *11th International Conference on Document Analysis and Recognition (ICDAR)*, pages 1250–1254.

Ciresan, D. C., Meier, U., Masci, J., Gambardella, L. M., and Schmidhuber, J. (2011b). Flexible, high performance convolutional neural networks for image classification. In *Intl. Joint Conference on Artificial Intelligence IJCAI*, pages 1237–1242.

Cireşan, D. C., Meier, U., Gambardella, L. M., and Schmidhuber, J. (2010). Deep, big, simple neural nets for handwritten digit recognition. *Neural Computation*, 22(12):3207–3220.

Clevert, D.-A., Unterthiner, T., and Hochreiter, S. (2015). Fast and accurate deep network learning by exponential linear units (elus).

Cubuk, E. D., Zoph, B., Mane, D., Vasudevan, V., and Le, Q. V. (2018). Autoaugment: Learning augmentation policies from data. *arXiv preprint arXiv:1805.09501*.

Cubuk, E. D., Zoph, B., Shlens, J., and Le, Q. V. (2019). Randaugment: Practical data augmentation with no separate search.

Cybenko, G. (1989). Approximation by superposition of a sigmoidal function. *Math. Control Systems Signals*, 2(4):303–314.

Dayan, P., Abbott, L. F., and Abbott, L. (2001). *Theoretical neuroscience: computational and mathematical modeling of neural systems*. MIT press Cambridge, MA.

Deng, J., Dong, W., Socher, R., Li, L.-J., Kai Li, and Li Fei-Fei (2009). ImageNet: A large-scale hierarchical image database. In *2009 IEEE Conference on Computer Vision and Pattern Recognition*, pages 248–255. IEEE.

Dreyfus, S. E. (1962). The numerical solution of variational problems. *Journal of Mathematical Analysis and Applications*, 5(1):30–45.

Duchi, J., Hazan, E., and Singer, Y. (2011). Adaptive subgradient methods for online learning and stochastic optimization. *Journal of Machine Learning Research*, 12(Jul):2121–2159.

Dumoulin, V. and Visin, F. (2016). A guide to convolution arithmetic for deep learning. *arXiv preprint arXiv:1603.07285*.

Euler, L. (1744). Methodus inveniendi.

Fahlman, S. E. (1991). An empirical study of learning speed in back-propagation networks. Technical Report CMU-CS-88-162, Carnegie-Mellon, Computer Science Department.

Fergus, R. (2015). Neural networks mlss 2015 summer school. MLSS 2015 Summer School.

Flower, B. and Jabri, M. (1993). Summed weight neuron perturbation: An $O(N)$ improvement over weight perturbation. In S. J. Hanson, J. D. C. and Giles, C. L., editors, *Advances in Neural Information Processing Systems 5*, pages 212–219. Morgan Kaufmann, San Mateo, CA.

Fukushima, K. (1979). Neural network model for a mechanism of pattern recognition unaffected by shift in position - Neocognitron. *Trans. IECE*, J62-A(10):658–665.

Fukushima, K. (1980). Neocognitron: A self-organizing neural network for a mechanism of pattern recognition unaffected by shift in position. *Biological Cybernetics*, 36(4):193–202.

Gal, Y. and Ghahramani, Z. (2016). Dropout as a bayesian approximation: Representing model uncertainty in deep learning. In *International Conference on Machine Learning*, pages 1050–1059.

Gauss, C. F. (1809). Theoria motus corporum coelestium in sectionibus conicis solem ambientium.

Gauss, C. F. (1821). Theoria combinationis observationum erroribus minimis obnoxiae (theory of the combination of observations least subject to error).

Glorot, X. and Bengio, Y. (2010). Understanding the difficulty of training deep feedforward neural networks. In Teh, Y. W. and Titterington, M., editors, *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics*, volume 9 of *Proceedings of Machine Learning Research*, pages 249–256, Chia Laguna Resort, Sardinia, Italy. PMLR.

Glorot, X., Bordes, A., and Bengio, Y. (2011). Deep sparse rectifier neural networks. In Gordon, G., Dunson, D., and Dudï¿œk, M., editors, *JMLR W&CP: Proceedings of the Fourteenth International Conference on Artificial Intelligence and Statistics (AISTATS 2011)*, volume 15, pages 315–323.

Goodfellow, I., Bengio, Y., and Courville, A. (2016). *Deep learning*. MIT press.

Goodfellow, I. J., Pouget-Abadie, J., Mirza, M., Xu, B., Warde-Farley, D., Ozair, S., Courville, A., and Bengio, Y. (2014). Generative adversarial networks. *ArXiv e-prints*, abs/1406.2661.

Goodfellow, I. J., Warde-Farley, D., Mirza, M., Courville, A., and Bengio, Y. (2013). Maxout networks. In *Proceedings of the 30th International Conference on International Conference on Machine Learning - Volume 28*.

Graham, B. (2014a). Fractional max-pooling.

Graham, B. (2014b). Spatially-sparse convolutional neural networks.

Hadamard, J. (1908). *Mémoire sur le problème d'analyse relatif à l'équilibre des plaques élastiques encastrées*. Mémoires présentés par divers savants à l'Académie des sciences de l'Institut de France: Éxtrait. Imprimerie nationale.

Hanson, S. J. and Pratt, L. Y. (1989). Comparing biases for minimal network construction with back-propagation. In Touretzky, D. S., editor, *Advances in Neural Information Processing Systems 1*, pages 177–185. San Mateo, CA: Morgan Kaufmann.

Hassibi, B. and Stork, D. G. (1993). Second order derivatives for network pruning: Optimal brain surgeon. In S. J. Hanson, J. D. C. and Giles, C. L., editors, *Advances in Neural Information Processing Systems 5*, pages 164–171. San Mateo, CA: Morgan Kaufmann.

He, K., Zhang, X., Ren, S., and Sun, J. (2015). Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. In *2015 IEEE International Conference on Computer Vision (ICCV)*, pages 1026–1034.

He, K., Zhang, X., Ren, S., and Sun, J. (2016). Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778.

Hebb, D. O. (1949). *The Organization of Behavior*. Wiley, New York.

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

Hochreiter, S. (1990). Implementierung und Anwendung eines 'neuronalen' Echtzeit-Lernalgorithmus für reaktive Umgebungen. Practical work, Institut für Informatik, Lehrstuhl Prof. Brauer, Technische Universität München.

Hochreiter, S. (1991). Untersuchungen zu dynamischen neuronalen Netzen. Diploma thesis, Institut für Informatik, Lehrstuhl Prof. Brauer, Technische Universität München. See www7.informatik.tu-muenchen.de/~hochreit.

Hochreiter, S. and Schmidhuber, J. (1995). Long Short-Term Memory. Submitted to NIPS 95.

Hochreiter, S. and Schmidhuber, J. (1997a). Flat minima. *Neural Computation*, 9(1):1–42.

Hochreiter, S. and Schmidhuber, J. (1997b). Long Short-Term Memory. *Neural Computation*, 9(8):1735–1780.

Hochreiter, S. and Schmidhuber, J. (1997c). Long short-term memory. *Neural Computation*, 9(8):1735–1780.

Hopfield, J. J. (1982). Neural networks and physical systems with emergent collective computational abilities. *Proc. of the National Academy of Sciences*, 79:2554–2558.

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

Huang, Y., Cheng, Y., Bapna, A., Firat, O., Chen, M. X., Chen, D., Lee, H., Ngiam, J., Le, Q. V., Wu, Y., and Chen, Z. (2018). Gpipe: Efficient training of giant neural networks using pipeline parallelism.

Ioffe, S. and Szegedy, C. (2015). Batch normalization: Accelerating deep network training by reducing internal covariate shift.

Ivakhnenko, A. G. (1971). Polynomial theory of complex systems. *IEEE Transactions on Systems, Man and Cybernetics*, (4):364–378.

Ivakhnenko, A. G. and Lapa, V. G. (1965). *Cybernetic Predicting Devices*. CCM Information Corporation.

Jacobs, R. A. (1988). Increased rates of convergence through learning rate adaptation. *Neural networks*, 1(4):295–307.

Karpathy, A. (2011). Lessons learned from manually classifying cifar-10. `http://karpathy.github.io/2011/04/27/manually-classifying-cifar10/`. Accessed: 2019-10-10.

Kelley, H. J. (1960). Gradient theory of optimal flight paths. *ARS Journal*, 30(10):947–954.

Kerlirzin, P. and Vallet, F. (1993). Robustness in multilayer perceptrons. *Neural Computation*, 5(1):473–482.

Kingma, D. P. and Ba, J. (2014). Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*.

Klambauer, G., Unterthiner, T., Mayr, A., and Hochreiter, S. (2017). Self-normalizing neural networks. In Guyon, I., Luxburg, U. V., Bengio, S., Wallach, H., Fergus, R., Vishwanathan, S., and Garnett, R., editors, *Advances in Neural Information Processing Systems 30*, pages 971–980. Curran Associates, Inc.

Korolev, V. and Shevtsova, I. (2012). An improvement of the berry–esseen inequality with applications to poisson and mixed poisson random sums. *Scandinavian Actuarial Journal*, 2012(2):81–105.

Krizhevsky, A. et al. (2009). Learning multiple layers of features from tiny images. Technical report, Citeseer.

Krizhevsky, A. et al. (2019). The cifar-10 dataset. `https://www.cs.toronto.edu/~kriz/cifar.html`. Accessed: 2019-10-10.

Krizhevsky, A., Sutskever, I., and Hinton, G. E. (2012a). Imagenet classification with deep convolutional neural networks. In *Advances in Neural Information Processing Systems (NIPS 2012)*, page 4.

Krizhevsky, A., Sutskever, I., and Hinton, G. E. (2012b). Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097–1105.

Krogh, A. and Hertz, J. A. (1992). A simple weight decay can improve generalization. In Moody, J. E., Hanson, S. J., and Lippman, R. P., editors, *Advances in Neural Information Processing Systems 4*, pages 950–957. San Mateo, CA: Morgan Kaufmann.

Kuzovkin, I., Vicente, R., Petton, M., Lachaux, J.-P., Baciu, M., Kahane, P., Rheims, S., Vidal, J. R., and Aru, J. (2018). Activations of deep convolutional neural networks are aligned with gamma band activity of human visual cortex. *Communications biology*, 1(1):107.

LeCun, Y. (1989). Generalization and network design strategies. In Pfeifer, R., Schreter, Z., Fogelman, F., and Steels, L., editors, *Connectionism in Perspective*, Zurich, Switzerland. Elsevier. an extended version was published as a technical report of the University of Toronto.

LeCun, Y. (1998). MNIST handwritten digit database. Available as http://www.research.att.com/∼yann/ocr/mnist/.

LeCun, Y. and Bengio, Y. (1995). Convolutional networks for images, speech, and time-series. In Arbib, M. A., editor, *The Handbook of Brain Theory and Neural Networks*. MIT Press.

LeCun, Y., Bengio, Y., and Hinton, G. (2015). Deep learning. *nature*, 521(7553):436.

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

LeCun, Y., Bottou, L., Bengio, Y., and Haffner, P. (1998a). Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86:2278–2324.

LeCun, Y., Bottou, L., Orr, G. B., and Müller, K.-R. (1998b). Efficient backprop. In *Neural Networks: Tricks of the Trade*, pages 9–50. Springer.

LeCun, Y., Cortes, C., and Burges, C. (2019). The mnist database. `http://yann.lecun.com/exdb/mnist/`. Accessed: 2019-10-10.

LeCun, Y., Denker, J. S., and Solla, S. A. (1990). Optimal brain damage. In Touretzky, D. S., editor, *Advances in Neural Information Processing Systems 2*, pages 598–605. San Mateo, CA: Morgan Kaufmann.

Lee, C.-Y., Gallagher, P. W., and Tu, Z. (2015). Generalizing pooling functions in convolutional neural networks: Mixed, gated, and tree.

Lee, H., Grosse, R., Ranganath, R., and Ng, A. Y. (2011). Unsupervised learning of hierarchical representations with convolutional deep belief networks. *Communications of the ACM*, 54(10):95–103.

Legendre, A. M. (1805). *Nouvelles méthodes pour la détermination des orbites des cometes*. F. Didot.

Leibniz, G. W. (1676). Memoir using the chain rule (cited in TMME 7:2&3 p 321-332, 2010).

Leibniz, G. W. (1684). Nova methodus pro maximis et minimis, itemque tangentibus, quae nec fractas, nec irrationales quantitates moratur, et singulare pro illis calculi genus. *Acta Eruditorum*, pages 467–473.

Levin, A. U., Leen, T. K., and Moody, J. E. (1994). Fast pruning using principal components. In Cowan, J. D., Tesauro, G., and Alspector, J., editors, *Advances in Neural Information Processing Systems 6*, pages 35–42. Morgan Kaufmann, San Mateo, CA.

Liao, Z. and Carneiro, G. (2015). Competitive multi-scale convolution.

Lillicrap, T. P., Cownden, D., Tweed, D. B., and Akerman, C. J. (2016). Random synaptic feedback weights support error backpropagation for deep learning. *Nature communications*, 7(13276).

Linnainmaa, S. (1970). The representation of the cumulative rounding error of an algorithm as a Taylor expansion of the local rounding errors. Master's thesis, Univ. Helsinki.

Mahajan, D., Girshick, R., Ramanathan, V., He, K., Paluri, M., Li, Y., Bharambe, A., and van der Maaten, L. (2018). Exploring the limits of weakly supervised pretraining.

Matsuoka, K. (1992). Noise injection into inputs in back-propagation learning. *IEEE Transactions on Systems, Man, and Cybernetics*, 22(3):436–440.

Mayr, A., Klambauer, G., Unterthiner, T., and Hochreiter, S. (2016). Deeptox: toxicity prediction using deep learning. *Frontiers in Environmental Science*, 3:80.

McCulloch, W. S. and Pitts, W. H. (1943). A logical calculus of the idea immanent in nervous activity. *Bull. Math. Biophy*, 5:115–133.

Miller, G. A. (1995). WordNet: a lexical database for English. *Communications of the ACM CACM ; a publ. of the Association for Computing Machinery*, 38(11).

Minai, A. A. and Williams, R. D. (1994). Perturbation response in feedforward networks. *Neural Networks*, 7(5):783–796.

Minsky, M. and Papert, S. (1969). *Perceptrons*. Cambridge, MA: MIT Press.

Mishkin, D. and Matas, J. (2015). All you need is a good init.

ML, A. (2019). Image classification on imagenet. `https://paperswithcode.com/sota/image-classification-on-imagenet`. Accessed: 2019-10-10.

Moody, J. E. (1992). The effective number of parameters: An analysis of generalization and regularization in nonlinear learning systems. In Moody, J. E., Hanson, S. J., and Lippman, R. P., editors, *Advances in Neural Information Processing Systems 4*, pages 847–854. San Mateo, CA: Morgan Kaufmann.

Mordvintsev, A., Olah, C., and Tyka, M. (2015). Inceptionism: Going deeper into neural networks, 2015. *URL https://research. googleblog. com/2015/06/inceptionism-going-deeper-into-neural. html*.

Mozer, M. C. and Smolensky, P. (1989). Skeletonization: A technique for trimming the fat from a network via relevance assessment. In Touretzky, D. S., editor, *Advances in Neural Information Processing Systems 1*, pages 107–115. San Mateo, CA: Morgan Kaufmann.

Murray, A. F. and Edwards, P. J. (1993). Synaptic weight noise during MLP learning enhances fault-tolerance, generalisation and learning trajectory. In S. J. Hanson, J. D. C. and Giles, C. L., editors, *Advances in Neural Information Processing Systems 5*, pages 491–498. San Mateo, CA: Morgan Kaufmann.

Nair, V. and Hinton, G. E. (2009). 3d object recognition with deep belief nets. In *Advances in Neural Information Processing Systems*.

Neti, C., Schneider, M. H., and Young, E. D. (1992). Maximally fault tolerant neural networks. In *IEEE Transactions on Neural Networks*, volume 3, pages 14–23.

Nowlan, S. J. and Hinton, G. E. (1992). Simplifying neural networks by soft weight sharing. *Neural Computation*, 4:173–193.

Oh, K.-S. and Jung, K. (2004). Gpu implementation of neural networks. *Pattern Recognition*, 37(6):1311–1314.

Paulsen, O. and Sejnowski, T. J. (2000). Natural patterns of activity and long-term synaptic plasticity. *Current opinion in neurobiology*, 10(2):172–180.

Perez, L. and Wang, J. (2017). The effectiveness of data augmentation in image classification using deep learning. *arXiv preprint arXiv:1712.04621*.

Refenes, A. N., Francis, G., and Zapranis, A. D. (1994). Stock performance modeling using neural networks: A comparative study with regression models. *Neural Networks*, 7(2):375–388.

Riedmiller, M. and Braun, H. (1993). A direct adaptive method for faster backpropagation learning: the rprop algorithm. In *IEEE International Conference on Neural Networks*, pages 586–591.

Rissanen, J. (1978). Modeling by shortest data description. *Automatica*, 14:465–471.

Rodrigo, B. (2016). Classification datasets results. `http://rodrigob.github.io/are_we_there_yet/build/classification_datasets_results.html`. Accessed: 2019-10-10.

Rosenblatt, F. (1957). *The perceptron, a perceiving and recognizing automaton Project Para*. Cornell Aeronautical Laboratory.

Rosenblatt, F. (1958a). The perceptron: a probabilistic model for information storage and organization in the brain. *Psychological review*, 65(6):386.

Rosenblatt, F. (1958b). The perceptron: A probabilistic model for information storage and organization in the brain. *Psychological Review*, 65(6):386–408.

Rosenblatt, F. (1962). *Principles of Neurodynamics*. Spartan, New York.

Rumelhart, D. E., Hinton, G. E., and Williams, R. J. (1986a). Learning internal representations by error propagation. In *Parallel Distributed Processing*, chapter 8, pages 318–362. MIT Press, Cambridge, MA.

Rumelhart, D. E., Hinton, G. E., and Williams, R. J. (1986b). Learning representations by back-propagating errors. *Nature*, 323(9):533–536.

Rumelhart, D. E., Hinton, G. E., and Williams, R. J. (1986c). *Parallel Distributed Processing*. MIT Press, Cambridge, MA.

Salimans, T. and Kingma, D. P. (2016). Weight normalization: A simple reparameterization to accelerate training of deep neural networks. In *Advances in Neural Information Processing Systems*, pages 901–901.

Saxe, A. M., McClelland, J. L., and Ganguli, S. (2014). Exact solutions to the nonlinear dynamics of learning in deep linear neural networks. *arXiv preprint arXiv:1312.6120*.

Schmidhuber, J. (2015). Deep learning in neural networks: An overview. *Neural networks*, 61:85–117.

Sejnowski, T. J. and Rosenberg, C. R. (1987). Parallel networks that learn to pronounce english text. *Complex Systems*, 1.

Silver, D., Huang, A., Maddison, C. J., Guez, A., Sifre, L., Van Den Driessche, G., Schrittwieser, J., Antonoglou, I., Panneershelvam, V., Lanctot, M., et al. (2016). Mastering the game of go with deep neural networks and tree search. *nature*, 529(7587):484–489.

Smolensky, P. (1986). Parallel distributed processing: Explorations in the microstructure of cognition, vol. 1. chapter Information Processing in Dynamical Systems: Foundations of Harmony Theory, pages 194–281. MIT Press, Cambridge, MA, USA.

Snoek, J., Rippel, O., Swersky, K., Kiros, R., Satish, N., Sundaram, N., Patwary, M. M. A., Prabhat, and Adams, R. P. (2015). Scalable bayesian optimization using deep neural networks.

Springenberg, J. T., Dosovitskiy, A., Brox, T., and Riedmiller, M. (2014). Striving for simplicity: The all convolutional net.

Srivastava, N., Hinton, G., Krizhevsky, A., Sutskever, I., and Salakhutdinov, R. (2014). Dropout: a simple way to prevent neural networks from overfitting. *The Journal of Machine Learning Research*, 15(1):1929–1958.

Tan, M. and Le, Q. (2019). Efficientnet: Rethinking model scaling for convolutional neural networks. In Chaudhuri, K. and Salakhutdinov, R., editors, *Proceedings of the 36th International Conference on Machine Learning*, volume 97 of *Proceedings of Machine Learning Research*, pages 6105–6114.

Tieleman, T. and Hinton, G. (2012). Lecture 6.5-rmsprop: Divide the gradient by a running average of its recent magnitude. *COURSERA: Neural networks for machine learning*, 4(2):26–31.

Torralba, A., Fergus, R., and Freeman, W. (2008). 80 Million Tiny Images: A Large Data Set for Nonparametric Object and Scene Recognition. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 30(11):1958–1970.

Touvron, H., Vedaldi, A., Douze, M., and Jégou, H. (2019). Fixing the train-test resolution discrepancy.

Wallace, C. S. and Boulton, D. M. (1968). An information measure for classification. *Computer Jrnl.*, 11(2):185–194.

Wang, C., Wang, Y., and Yuille, A. L. (2013). An Approach to Pose-Based Action Recognition. *2013 IEEE Conference on Computer Vision and Pattern Recognition*, pages 915–922.

Weigend, A. S., Rumelhart, D. E., and Huberman, B. A. (1991). Generalization by weight-elimination with application to forecasting. In Lippmann, R. P., Moody, J. E., and Touretzky, D. S., editors, *Advances in Neural Information Processing Systems 3*, pages 875–882. San Mateo, CA: Morgan Kaufmann.

Werbos, P. J. (1974). *Beyond Regression: New Tools for Prediction and Analysis in the Behavioral Sciences*. PhD thesis, Harvard University.

Werbos, P. J. (1981). Applications of advances in nonlinear sensitivity analysis. In Drenick, R. F. and Kozin, F., editors, *System Modeling and Optimization: Proceedings of the 10th IFIP Conference, 31.8 - 4.9, NYC*, pages 762–770. Springer-Verlag. number 38 in Lecture Notes in Control and Information Sciences.

White, H. (1989). Learning in artificial neural networks: A statistical perspective. *Neural Computation*, 1(4):425–464.

Williams, P. M. (1994). Bayesian regularisation and pruning using a Laplace prior. Technical report, School of Cognitive and Computing Sciences, University of Sussex, Falmer, Brighton.

Wilson, D. R. and Martinez, T. R. (2003). The general inefficiency of batch training for gradient descent learning. *Neural networks*, 16(10):1429–1451.

Zeiler, M. D. (2012). Adadelta: an adaptive learning rate method. *arXiv preprint arXiv:1212.5701*.

Zeiler, M. D. and Fergus, R. (2014). Visualizing and understanding convolutional networks. In *European conference on computer vision*, pages 818–833.

Index