UNIVERSITÄT OSNABRÜCK

Bachelor Thesis

Representation Learning with Feedforward Neural Networks

Anna-Lena Popkes

1st Supervisor:
Prof. Dr. Gordon Pipa
(University of Osnabrueck)

2nd Supervisor:
Johannes Leugering
(University of Osnabrueck)

August 2015

ABSTRACT

The performance of machine learning algorithms is heavily dependent on the representation of the data they are given. Data described by means of poor or unfortunate features often results in a bad performance of the deployed machine learning algorithm. As it is often difficult for humans to hand-code the explanatory factors or features of data beforehand, representation learning aims at *learning* useful representations of the given data. Various machine learning algorithms can be used in order to perform representation learning.
This thesis introduces the field of representation learning and reviews the major types of feedforward neural networks that can be employed in order to learn representations of data.

ACKNOWLEDGMENTS

There are several people I want to thank for supporting me in the compilation of this thesis.

First of all, my parents who have always supported me in every aspect of my life. I am enormously grateful for all your love, unconditional support and inspiration. Words cannot explain how proud I am to have parents like you. Mom, Dad, you are the best!

Secondly, I have to thank my first supervisor Prof. Dr. Gordon Pipa who managed to find time for me in spite of his unbelievable workload. Furthermore, he had the brilliant idea to suggest Johannes as my second supervisor.
Johannes was by far the biggest help during the last months. He went out of his way to answer any question I had and to get rid of any problem I was faced with. I wasn't expecting that a second supervisor could be such a big support.

Furthermore, I want to thank Julia Eylers, my brilliant English teacher to-be, who assisted with all incoming questions regarding word order, punctuation and tense. I also gratefully acknowledge the support and feedback of Philipp Kuhnke, Swen Wenzel and Simeon Helgers.

Last but not least, I thank Pascal Wenker for open ears, valuable advice and never-ending emotional support.

CONTENTS

1. INTRODUCTION

Today, artificial intelligence (AI) is a rapidly developing research field with many different applications and directions of research. When it first started its growth, most of its successes were related to problems that can be described by a list of formal, mathematical rules (Bengio et al., 2015, p. 4). One of the most popular examples for such a success is DeepBlue, the chess playing system of IBM. After nearly twelve years of active development and an initial loss in 1996, DeepBlue was the first artificial intelligence that succeeded in defeating an incumbent world champion. After six games distributed over ten days, Deep Blue won against the defending world champion Garry Kasparov on May 11th, 1997 (Hsu, 2002).

Abstract and formal tasks like chess are difficult for humans to perform but belong to the easiest ones for a computer. If a task, however, is not describable in terms of formal and explicit rules, computers repeatedly fail to solve it. This mostly involves tasks humans perform intuitively by relying on their general knowledge about the world: recognizing a familiar face, spoken words, or classifying objects. Such tasks reveal the real challenge for artificial intelligence: in order to behave intelligently, the machines need to absorb the massive amount of knowledge about the world humans use and expand every day (Bengio et al., 2015, p. 5). As Bengio et al. (2013, p. 1798) put it: “An AI must fundamentally understand the world around us”.

An early attempt to solve the problem of integrating informal knowledge into a computer is known as the *knowledge based* approach. It aims at designing formal rules that capture knowledge about the world (Lenat, 1995; Bosse et al., 2007, p. 279-306; Aamodt, 1993). Since devising such formal rules has always been a problem for humans, knowledge-based systems were quickly faced with several difficulties. For example, knowledge-based systems for medical diagnosis (e.g. INTERNIST) failed at providing a complete database, i.e. including representations for *all* potentially relevant aspects of the problem (disease profiles, disease variations, symptoms and rules describing relations among symptoms) (Winograd, 1991; Duda and Shortliffe, 1983). This pinpoints the need of artificial intelligence systems “to acquire their own knowledge, by extracting patterns from raw data” (Bengio et al., 2015, p. 5). This ability is commonly referred to as *machine learning.* Since all machine algorithms acquire their knowledge from *data* and since their performance is heavily dependent on the *representation* of that data, another important discipline arises: *representation learning.*

In this thesis, I will approach the topic of representation learning and will introduce the most popular types of feedforward neural networks that can be employed in order to learn representations of data. For this, I will begin with a review on representation learning in chapter 2. In chapter 3, I will introduce artificial neural networks in general, before I will turn to specific types of feefor-ward neural networks in chapters 4, 5 and 6. I will end with a short summary and an outlook on further interesting topics in chapter 7.

2.1 What Is Representation Learning?

The performance of machine learning algorithms is strongly influenced by the representation of the data they are given (Ancona et al., 2004; Bengio et al., 2013, p. 1798). Ancona et al. (2004, p. 1) describe the problem of data representation as "one of the most critical issues concerning the realization of intelligence machines which are able to solve real life problems". A simple example by Bengio et al. (2015, p. 6) illustrates why the choice of representation can have such a huge effect. In figure 2.1 two categories of data should be separated by drawing a straight line between them in the corresponding scatterplot. In the left plot, the data is represented by means of Cartesian coordinates. This type of representation makes a separation by a straight line impossible. In the right plot, the data is represented with polar coordinates and a separation can easily be achieved.

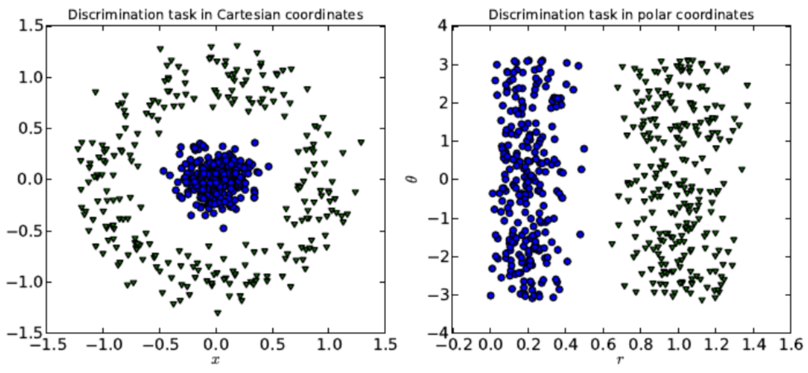

Fig. 2.1: Example of different representations: two categories of data should be separated by drawing a straight line between them in a scatterplot. In the plot on the left, the data is represented using Cartesian coordinates, and the task is impossible. In the plot on the right, the data is represented with polar coordinates and the task becomes simple to solve with a vertical line. Adapted from Bengio et al. (2015, p. 6).

A representation is composed of various *features*. Since many simple machine learning algorithms cannot influence the way in which the features of the representation are defined (Bengio et al., 2015, p. 6), a bad representation (i.e. data represented by means of poor features) often results in a bad performance.
For many tasks, this does not pose a problem since the "right" set of features (i.e. the set of features that should be extracted) can be hand-coded beforehand

and provided to the machine. However, for an enormous amount of tasks, it is difficult to identify beforehand which features should be extracted (Bengio et al., 2015, p. 6). An example that illustrates this problem is the following: the task of detecting chairs (i.e. static objects) in pictures does not appear as difficult as, for instance, face recognition at first sight. But trying to describe clear features of a chair reveals the task's difficulty. As evident in figure 2.2, a simple definition like "a chair is something with legs, a back and a seat" would not suffice for detecting all chairs in the picture correctly. The miniature chair on the desk does indeed have legs, a back and a seat. Still, it should not be detected as being a real chair but a model of a chair. Furthermore, the presence of shadows, different light sources or viewing angles and other disruptive factors in the images could complicate the task. For example, a chair can be masked by other objects. Therefore, also other, more detailed descriptions will not correctly classify all instances of a chair. Already Minsky (1988) analyzed the problem of finding distinct and explicit features of chairs in his book "Society of Mind". He came to the conclusion that "when all is done, there's little we can find in common to all chairs - except for their intended purpose" (Minsky, 1988, p. 123).

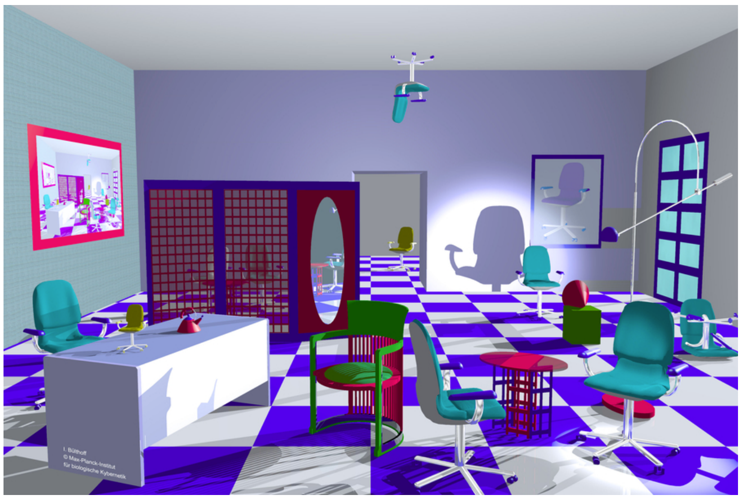

Fig. 2.2: Variations of chairs illustrating the problem of describing clear features of a chair. By Buelthoff and Buelthoff (2003, p. 147).

Representation learning tries to solve the above mentioned problem. It aims at "learning representations of the data that make it easier to extract useful information when building classifiers or other predictors" (Bengio et al., 2013, p. 1798). While many machine learning algorithms only learn a mapping from a given representation to a desired output (e.g. algorithms that classify an object on the basis of predefined features), representation learning uses the algorithm to additionally learn the *representation itself* (Bengio et al., 2015, p. 7). Consequently, representation learning algorithms must discover explanatory factors or features that characterize the data they are given (Bengio et al., 2013, p. 1798). These features are then used to construct a representation of the data. Several ways of learning such representations exist. This work will focus on the major feedforward neural-network approaches.

2.2 Why Is Representation Learning Important?

As mentioned above, the performance of machine learning algorithms is heavily dependent on the representation of the data. Therefore it is not surprising that the topic of representation learning became a field itself. Besides regular workshops at the leading conferences NIPS (Neural Information Processing Systems) and ICML (International Conference on Machine Learning), the new conference ICLR (International Conference on Learning Representations) is concerned solely with the topic at hand (Deng and Yu, 2014, pp. 198-199). The huge scientific interest in representation learning has furthermore caused several striking successes in academia as well as industry (Bengio et al., 2013, p. 1798). One example for such an achievement concerns the field of speech recognition and signal processing. Here, the biggest successes started with the application of *deep learning* architectures in 2009, when several research groups collaborated (see review in Hinton et al. 2012). Since then, newly developed deep learning techniques and architectures have been shown to outperform previously established techniques (e.g. Gaussian mixture models) on a variety of speech recognition benchmarks, sometimes by a large margin (Dahl et al., 2010; Dahl et al., 2012; Deng et al., 2010; Hinton et al., 2012; Mohamed et al., 2012; Seide et al., 2011; Yu et al., 2010). Further information on the topic of deep learning can be found in chapter 6.

2.3 Types of Representations

2.3.1 Local Representations

(Conceptual) information can be represented in neural networks in many different ways. Extreme *localist* theories are on the one end of those possible approaches (Hinton, 1986, p. 1; Bengio, 2009, p. 8). Localist theories suggest that each concept is represented by a single neural unit (Barlow, 1972; Quiroga et al., 2013; Gross, 2002; Hinton, 1986; Bengio, 2009). A famous example of this idea is the so called "grandmother cell" (Murphy, 2012, p. 989; Rolls and Treves, 1998, p. 12). As Gross (2002, p. 84) puts it: "A 'grandmother cell' is a hypothetical neuron that responds only to a highly complex, specific and meaningful stimulus, such as the image of one's grandmother".
In such a local representation, all information about a particular stimulus or event is represented by the activity of a single neuron (Rolls and Treves, 1998, p. 12). Hinton (1986, p. 1) describes the origin of the extreme localist approach as the "natural implementation" of a specific theory of semantics, namely of the structuralist approach. Structuralism is routed in de Saussure's pioneering work on sign systems, in which he argued that the meaning of signs is not intrinsic but constituted by their relationship to other signs (de Saussure, 1959). Likewise, concepts are defined by their relation to other concepts and not by their internal structure (Culler, 1981). Applied to neural networks, this corresponds to the idea that each concept is represented by a single neuron and that relationships between concepts are encoded by the connections between the neurons (Hinton,

1986, p. 1).

2.3.2 Distributed Representations

According to Hinton (1986, p. 1) and Bengio (2009, p. 8), the other extreme of how concepts might be represented in neural networks are *extreme distributed* theories. These are routed in an approach called componential analysis (Hinton, 1986, p. 1). Componential analysis is based on the assumption that every concept can be described by the combination of a limited number of "universally valid features" (Bussmann, 1996, p. 219). Transferred to neural networks, this corresponds to the idea that every concept is represented by a pattern of active neurons distributed across a large part of the cortex. And further, that a single neuron contributes to several concepts (Deng and Yu, 2014, p. 218). A specific person, like one's grandmother, would therefore be represented by a whole population of neurons. The pattern of their firing rates would encode the concept. Furthermore, the individual neurons would not respond solely to the image of one's grandmother, but also to other concepts. Bengio et al. (2013, p. 1801) give the following definition: "Distributed representations: Where k out of N representation elements or feature values can be independently varied, e.g. they are not mutually exclusive. Each concept is represented by having k features being turned on or active, while each feature is involved in representing many concepts".

2.3.3 Sparse Representations

In the middle of the spectrum between extreme local and extreme distributed representations are so called *sparse* representations (Bengio, 2009, p. 8). While still being distributed, sparse representations are characterized by the fact that only a small set of neurons is active and involved in the representation of a concept (Rolls and Treves, 1998, p. 12). According to Bengio et al. (2013, p. 1801), sparse representations are "distributed representations where only a few of the elements can be varied at a time". The representation of a specific stimulus like one's grandmother would therefore be similar to the above mentioned distributed case. However, only a small population of neurons would participate in the representation.

2.3.4 Characteristics

Naturally, every type of representation has different implications, advantages and disadvantages. A local approach to representing concepts would provide the neuron in question "with the ability to represent a complex and specific concept" (Gross, 2002, pp. 89-90). This in turn implies that neurons would fire only in reponse to the concept or event they represent, and therefore only very rarely which is often considered a problem of the localist approach (Rolls and Treves, 1998, p. 12). Opponents further argue, that localist theories require a new neuron for every concept or event (Rolls and Treves, 1998, p. 12) which has the direct consequence that if a neuron was damaged or destroyed, the entire concept would be extinguished (Quiroga et al., 2013, p. 33). Other disadvantages of local representations become evident when considering the properties of distributed and sparse representations, as done below.

One advantage of distributed representations is that similarity between two stimuli can be expressed, namely by the correlation of the two corresponding patterns of activity (Hinton, 1986, p. 4). This is not possible with local encoding. Here, either one or the other stimulus is present and no similarities between stimuli are encoded. The possibility of expressing similarities in distributed representations furthermore "enables generalization to similar stimuli, or to incomplete versions of a stimulus" (Rolls and Treves, 1998, p. 13).

To revisit one of the above mentioned disadvantages of local representations, distributed ones have no problem handling damaged or destroyed neurons (Hinton, 1992, p. 150; Deng and Yu, 2014, p. 218). The reason for this is that the represented information does not rely on a single neuron, but on a large population of them.
Another important advantage of distributed as well as of sparse representations is that they can express a huge number of different stimuli (Rolls and Treves, 1998, p. 13; Bengio et al., 2013, p. 1801). Purely local representations need O(N) parameters to distinguish O(N) stimuli. Distributed and sparse representations, however, can represent up to $O(2^N)$ stimuli using the same number of parameters (Bengio et al., 2013, p. 1801). So the number of stimuli that can be encoded increases *exponentially* with the number of neurons in the population, whereas, with local encoding, it only increases *linearly* (Rolls and Treves, 1998, p. 13).

2.4 Types of Representations Used in the Brain

As Hinton (1992, p. 145) puts it: "The brain is a remarkable computer". Indeed, our brain performs many difficult tasks like recognizing faces or identifying objects extremely fast. With regard to representation learning, especially one ability is interesting: the brain is able to create internal representations of what is perceived or recalled easily and without any specific instructions. This raises the question of how information is coded in the brain.

All perception and memory of the world is based on the meaning people attribute to the information they sense or recall (Quiroga, 2012, p. 587). Due to the massive amount of sensory information that is perceivable, the brain not only has to form abstractions of what is sensed but must also extract relevant features (Fabre-Thorpe, 2003; Logothetis and Sheinberg, 1996). So as in general representation learning, the brain constructs representations of what is perceived/recalled out of the explanatory factors that it identifies in the data.

The idea that information in the brain is encoded locally (i.e. single cells responding to one specific object or person) has come up several times in the past (Gross, 2002, p. 90). Although forms of grandmother cells, like the "Jennifer Aniston cell"[1], could be found in the human brain (Quiroga et al., 2005),

[1] The Jennifer Aniston cell was found by Quiroga et al. in 2005 and responded strongly to different photographs of Jennifer Aniston but not to photographs of other actors, celebrities, animals or places. Similar neurons were found for Halle Berry, the Sydney Opera House etc.

the discovering scientists themselves object to the idea of local encoding and name several arguments that speak against it (Quiroga et al., 2008). Applied to the Jennifer Aniston neuron, their first argument illustrates the extremely low probability of finding the one and only cell out of a few million that responds solely to Jennifer Aniston. Their second argument is that finding a neuron that responds only to one person does not eliminate the possibility “that a response to other persons or objects would have been found if more pictures had been presented” (Quiroga et al., 2008, p. 89). In the Jennifer Aniston case, a day after the experiment, the scientists discovered that the neuron also responded to Lisa Kudrow, a co-star of Aniston in the TV-series *Friends* (Quiroga et al., 2013, p. 33). Last, they point out that “theoretical considerations estimate that each cell most probably responds to between 50 and 150 distinct individuals or objects” (Quiroga et al., 2008, p. 89). So a purely local encoding in the brain seems very unlikely. Sparse and distributed representations, however, are much more likely to be used.

Evidence suggests that the brain uses distributed representations in different sensory areas (Rolls et al., 1998; Fabre-Thorpe, 2003; Georgopoulos et al., 1986; Abbott et al., 1996; Rolls et al., 1997). The use of distributed representations in the brain is called *population coding*, since a whole population of neurons is used to represent information (Hinton, 1992, p. 150). Within the population, the single neurons are broadly tuned to a stimulus value (Quiroga et al., 2008, p. 87). An example of such a distributed representation is given by the encoding of movement direction. Georgopoulos et al. (1986) demonstrated that the direction in which a monkey moves its arm is encoded by a whole population of neurons in the monkey’s motor cortex.
But not all information in the brain is represented by the activity of a large population of neurons. Also much smaller populations are used for representation. In such sparse representations, the neurons are tuned less broadly and respond to more specific features (Quiroga et al., 2008, p. 87). An example for such sparse coding is given by the above mentioned “Jennifer-Aniston” neurons. These neurons were found in the hippocampus, a structure of the medial temporal lobe and possess several properties that indicate a sparse, invariant and unique encoding. For instance, their responses are very selective. Also, they fire in response to a specific concept regardless of how the concept is represented. The Jennifer-Aniston neuron, for example, responded not only to pictures of the actress, but also to her spoken or written name (Quiroga et al., 2013, p. 33). What differentiates sparse representations from extreme local representations is the fact that the neurons do not respond to just one specific object or person but to a *concept*. Also, not only a single but *several* neurons are encoding the concept, forming a small population. Furthermore, these cells may fire to more than just one concept. Important to note is, however, that “if they do, the concepts tend to be closely related” (Quiroga et al., 2013, p. 33).

The use of distributed and sparse representations in the brain is not irregular, but follows a simple principle. The way in which a neuron represents information is dependent on its location in the brain. Whereas representations in primary sensory areas are typically distributed, representations in higher areas are sparse (Olshausen and Field, 2004; Perez-Orive et al., 2002; Theunissen, 2003). A possible explanation for this might lie in the properties of the different kinds of representations. Whereas sparse representations are better suited for the formation of novel associations and memories as well as for fast learning, distributed representations allow the system to categorize, generalize and to slowly learn the general structure of the environment (McClelland et al., 1995; O'Reilly and Norman, 2002; Quiroga, 2012).

2.5 What Is a Good Representation?

A good representation exhibits certain important properties. First, it should separate the *factors of variation* that help explaining the data (Bengio et al., 2015, p. 7; Bengio et al., 2013, p. 1802). Often, these factors cannot directly be observed or measured. For example, when considering the image of a car, the factors of variation include the viewing angle, the light source(s), the position of the car as well as its shape and color. Since many of the factors influence all other pieces of the data, it is important to identify them such that relevant ones can be filtered out. For instance, referring back to the car-example, the exact shape of the car depends on the angle from which the image was taken.
Another property of a good representation is *expressiveness*. For a representation to be good, it should be able to capture many possible stimuli/inputs (Bengio et al., 2013, p. 1801). This is particularly easy for distributed and sparse representations which, as stated in section 2.3.4, can represent up to $O(2^N)$ stimuli using only O(N) parameters (Bengio et al., 2013, p. 1801).
Hinton (1992, p. 149) mentions one more desirable property, namely that a good representation should be describable in an efficient way using as little parameters as possible, while at the same time being able to reconstruct the input very closely.

2.6 Two Forms of Representation Learning

As it is the subject of this thesis, I will narrow down the topic of representation learning to neural networks in the following.
Neural networks can learn from their surrounding environment in many different ways, just as humans can (Haykin, 2009, p. 34). For representation learning, two forms of learning are relevant: supervised learning and unsupervised learning.[2]

2.6.1 Supervised Learning

In supervised learning, the aim is to predict an output for a given input vector (Hinton, 2012). It can be divided into *regression*, where the output is either a real number or a vector of real numbers, and *classification*, where the output is a class label (Hinton, 2012; Ng; Fausett, 1994, p. 15). What distinguishes supervised from unsupervised learning is that supervised learning makes use of a *teacher* (Arbib, 1987, p. 70; Bishop, 1995, p. 10). The teacher has knowledge about the surrounding environment (which is unknown to the neural network) and forwards this knowledge to the network by providing it with the desired

[2] A third form of learning is given by *reinforcement learning*. As only supervised and unsupervised learning are relevant for representation learning, reinforcement learning will not be covered in this work.

response for each training input (Fausett, 1994, p. 15; Haykin, 2009, pp. 34-35). While learning, the parameters of the neural network are modified in order to reduce the difference between the current output and the target output (this difference is known as the *error signal*) (Haykin, 2009, pp. 34-35; Hinton, 2012). Furthermore, a transmission of knowledge occurs during the modification process: the knowledge of the environment becomes incorporated into the neural network and is stored in its synaptic weights (Haykin, 2009, p. 35).

2.6.2 Unsupervised Learning

As mentioned, unsupervised learning does not make use of a teacher (Haykin, 2009, p. 37). This means that the neural network is not provided with any labeled examples, but with unlabeled data (Bishop, 1995, p. 10; Haykin, 2009, p. 37; Fausett, 1994, p. 16). According to Hinton (2012), unsupervised learning does not have one clear goal but can aim at several things. One major objective is the creation of an internal representation of the input. This can either be a compact, low dimensional one, or a high-dimensional one that can be described economically (Hinton, 2012). Since the network does not have explicit examples of the target function, a task independent measure is provided that determines the quality of the representation that should be learned. During learning, the weights of the network are modified according to this measure (Arbib, 1987, p. 70; Becker, 1991, p. 3).

3. ARTIFICIAL NEURAL NETWORKS

The development of artificial neural networks (ANN's) was motivated by both the desire to understand the brain and the desire to model some of its impressive skills (Fausett, 1994, p. 1; Haykin, 2009, p. 1). Although the human brain works in a completely different way than common digital computers, it is able to perform specific actions (e.g. object recognition) much faster than any of them. This is due to its highly parallel computation which is based on distributed representations (Haykin, 2009, p. 1; Mitchell, 1997, p. 82).

On the whole, our brain has approximately 100 billion neurons of which each is connected to ten thousand other neurons (Mitchell, 1997, p. 82). In spite of the fact that neurons operate much slower than a computer (up to 10^7 times), this tremendous amount of interconnections together with the parallel processing create a network of impressive power (Haykin, 2009, p. 6; Mitchell, 1997, p. 82).

3.1 Artificial Neural Networks

Artificial neural networks have been developed as mathematical generalizations of information-processing operations happening in biological neural networks (Fausett, 1994, p. 3; Bishop, 2006, p. 226; Haykin, 2009, p. 2). They consist of a large number of neurons with directed interconnections and affiliated, changeable weights (Fausett, 1994, p. 3; Hinton, 1992, p. 145; Haykin, 2009, p. 2). With the help of an activation function, each neuron transforms its weighted inputs into a single output signal which is then propagated to several other neurons (Fausett, 1994, pp. 3-4; Hinton, 1992, p. 145).

An artificial neural network can be characterized by the three properties that are decisive for its functioning (Fausett, 1994, p. 3; Hinton, 1992, p. 145). The first important property is the *architecture* of the network, which describes how the neurons are connected to each other. Common types of such neural architectures are described in section 3.4. The second property is the *learning algorithm* that is used to set the weights of the network. While the connections between the neurons determine whether they can influence each other, the weights define the strength of this influence (Hinton, 1992, p. 145). The last important property that needs to be considered is the *activation function* which determines the output of each neuron. Common types of such activation functions are discussed in section 3.5. As different activation functions possess different properties, a variety of them is employed in neural networks (e.g. linear or threshold activation functions) (Karlik and Olgac, 2011; Sibi et al., 2013). Although it is not obligatory, the same activation function is typically used within a layer of a neural net.

3.2 Biological Inspiration

Although biological neural networks have been the original inspiration for ANN's, many of their properties are not modeled by artificial neural nets. Also, many features of ANN's do not agree with biological networks (Mitchell, 1997, p. 82). Nevertheless, an illustration of some of the biological characteristics helps explaining fundamental properties of artificial neural nets.

A biological neuron receives incoming signals through its *dendrites* (Hinton, 1992, p. 145; Fausett, 1994, p. 5; Haykin, 2009, p. 7). Dendrites, whose structure resembles that of a tree, are fine cell filaments with many branches and an irregular surface (Freeman, 1975, p. 11; Arbib, 1987, p. 16). Signals that arrive at the dendrites are spikes of electrical activity which are sent out by other neurons through their *axons* (Hinton, 1992, p. 145; Fausett, 1994, p. 5; Haykin, 2009, p. 7; Freeman, 1975, p. 12). Axons are also cell filaments, but longer than dendrites and less branched (Freeman, 1975, p. 11). They represent the transmission line of a neuron (Haykin, 2009, p. 7).

At the end of each axon branch, a *synapse* transmits the electrical impulses sent out by a neuron to connected neurons (Hinton, 1992, p. 145; Fausett, 1994, p. 5). A synapse is a connection between two neurons in which the plasma membrane of the *presynaptic* neuron is in close contact to the plasma membrane of the *postsynaptic* neuron. Most common are chemical synapses. In such synapses the presynaptic process releases a neurotransmitter that diffuses across the synaptic cleft between the two membranes and then influences the postsynaptic process (Arbib, 1987, p. 16; Shepherd, 2004, p. 3). Thus, a synapse "converts a presynaptic electrical signal into a chemical signal and back into a postsynaptic electrical signal" (Shepherd, 2004, p. 3). The influence of a synapse can then either inhibit or excite the activity of the postsynaptic neuron (Hinton, 1992, p. 145; Shepherd, 2004, p. 3).

Next, the *soma* (also called *cell body*) of a neuron "sums" the incoming excitatory and inhibitory signals.[1] If the sum is sufficiently large, the neuron fires and sends an electrical impulse (called spike) along its axon to other, connected neurons (Hinton, 1992, p. 145; Fausett, 1994, p. 5). An illustration of a common biological neuron is given in figure 3.1.

Several characteristic features of artificial neural networks are inspired by the mentioned properties of biological neurons (Fausett, 1994, p. 5). First, neurons receive many inputs but transmit only a single output which is sent to several other neurons. Second, incoming activity is modified differently by the postsynaptic weights and only causes the neuron to fire if the overall input is sufficiently large (Fausett, 1994, p. 6).

[1] The summation of incoming signals is only a simplification that is used for constructing artificial neurons. The real processes happening in a biological neuron are much more complicated.

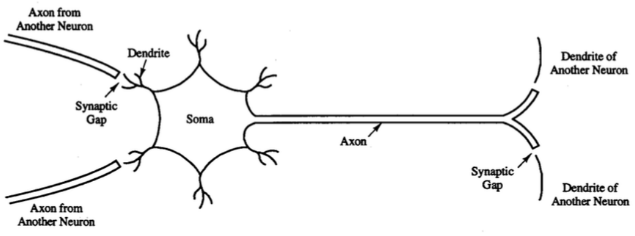

Fig. 3.1: Biological neuron together with axons from two other neurons by Fausett (1994, p. 6).

Furthermore, artificial networks may sometimes exhibit fault tolerance as a byproduct of their functioning. This is similar to biological neural networks, which can compensate the loss of neurons, even in cases of significant damage (Fausett, 1994, pp. 6-7).[2] Last, both biological as well as artificial networks are able to identify input signals they have not encountered before (Fausett, 1994, pp. 6-7).

Some more suggested features exist that will not be covered by this work. These include the distributed organization of memory, modification of synapses by experience and the locality of information processing.

3.3 Formal Neurons

An artificial neuron is the building block of an artificial neural network and elementary for its functioning (Haykin, 2009, p. 10). Compared to biological neurons, it is a very simple abstraction and rather primitive (Mitchell, 1997, p. 82).

An artificial neuron k is made up out of several different elements. The d input signals of a neuron form a vector $\vec{x} \in \mathbb{R}^d$. Each input x_i is multiplied by a synaptic weight $w_{kj}, w \in \mathbb{R}^d$ where k refers to the neuron in question and j to the (respective) input neuron (Haykin, 2009, p. 10). While the brain has excitatory and inhibitory *neurons*, an artificial neural network possesses just one class of neurons but positive and negative *weights* in exchange (Haykin, 2009, p. 10). The synaptic weights determine how much influence the output of a neuron has on other, connected neurons (Anthony, 2001, p. 2).

In addition to the synaptic weights, an artificial neuron is subject to a bias b_k. The bias acts as a threshold by either increasing (if the bias is negative) or decreasing (if the bias is positive) the activation of the neuron. The sum of the weighted inputs and the bias constitutes the activation s_k of the neuron (Haykin, 2009, p. 10; Bishop, 2006, p. 227; Anthony, 2001, p. 3). In mathematical terms, this can be expressed by the following equation:

[2] Fault tolerance does not occur in all types of neural networks but is dependent on the design and type of the net.

$$s_k = b_k + \sum_{j=1}^{d} w_{kj} x_j \tag{3.1}$$

To simplify notation, the bias b_k can be included in the sum by adding a constant component of $x_0 = 1$ to the input vector and a weight $w_{k0} = b_k$ to the weight vector $\vec{w}$ (Haykin, 2009, pp. 11-12). Equation 3.1 can then be reformulated as:

$$s_k = \sum_{j=0}^{d} w_{kj} x_j \tag{3.2}$$

The output y_k of a neuron is computed by applying an activation function g to the activation s_k (Haykin, 2009, p. 11; Bishop, 2006, p. 227):

$$y_k = g(s_k) \tag{3.3}$$

The activation function is also known as a *squashing function*, since it limits the possible range of the neuron's output (Haykin, 2009, p. 10). As mentioned in section 3.1, several choices of activation functions exist. An illustration of a simple model of a formal neuron with the above mentioned properties can be found in figure 3.2.

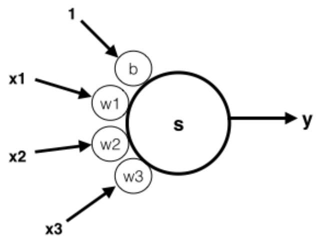

Fig. 3.2: A simple model of a formal neuron with weights $\vec{w} = [b, w_1, w_2, w_3]$. The neuron is subject to an input vector $\vec{x} = [+1, x_1, x_2, x_3]$. The output y of the neuron is computed by applying an activation function g to its activation s.

3.4 Network Architectures

An artificial neural network is typically composed of *layers* of neurons (Haykin, 2009, p. 21). As mentioned in section 3.1, the arrangement of neurons and their connections to each other is called the *net architecture*. Since neural networks are designed to perform certain tasks, the architecture of the network is closely linked to the learning algorithm that is used to train it (Haykin, 2009, p. 21).

Most neural nets have an *input layer* composed of several *input neurons*. The input layer does not receive signals from other neurons but from the outside environment (Fausett, 1994, p. 12; Anthony, 2001, p. 5). Therefore, the number

of input neurons is determined by the external problem and the activation of the neurons is equal to the external input signal (Fausett, 1994, p. 12; Hagan et al., 2014, p. 2-11; Anthony, 2001, p. 5).
Similar to the input layer, the output of a neural network is determined by the net's *output layer* (Fausett, 1994, p. 12; Hagan et al., 2014, p. 2-12). The output layer is composed of *output neurons* whose number is also fixed by the external problem (Hagan et al., 2014, p. 2-12). Dependent on the kind of architecture, a neural network may furthermore have one or more *hidden layers* between the input and output layer.

3.4.1 Single-Layer Network

The simplest form of a neural network is a *single-layer* network. Such a network consists of only an input layer which projects directly onto an output layer, usually in a *feedforward* fashion (Fausett, 1994, p. 12; Haykin, 2009, p. 21). The term feedforward refers to networks that do not possess any directed loops. In such networks, the output of one layer is used exclusively as an input to the next layer (Anthony, 2001, p. 6). In a single-layer network this would correspond to an architecture in which the input neurons are connected *only* to the output neurons (and not to each other) and in which the output neurons are likewise not connected to any other output neurons (Fausett, 1994, p. 12). Important to note is that the layers of a neural net do not have to be fully connected. Various forms of partially connected nets are also possible.
Although, strictly speaking, the just described network is composed of two layers, it is called a "single-layer" network. This is due to the fact that input layers are typically not counted as layers since they do not perform any computation (Fausett, 1994, p. 12; Haykin, 2009, p. 21). A simple feedforward single-layer network is illustrated in figure 3.3.

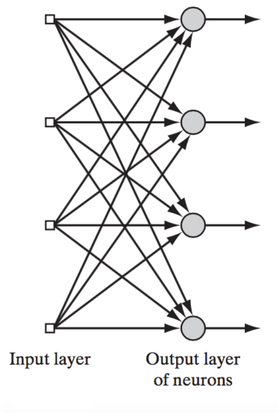

Fig. 3.3: Simple, feedforward network with a single layer of neurons by Haykin (2009, p. 21).

3.4.2 Multilayer Network

A network with several layers is called a *multilayer* network. Such networks can be distinguished from single-layer nets by the presence of one or more *hidden*

layers that are composed of so called *hidden neurons* (Fausett, 1994, p. 14; Haykin, 2009, p. 22; Hagan et al., 2014, p. 2-11; Anthony, 2001, p. 6). The layers and neurons are called "hidden" because their structure and properties cannot be directly observed from either the input or the output of the network (Haykin, 2009, p. 22).
Designing the hidden layers is much more complicated than designing the input and output layers since neither the number of neurons nor the connections between them can be directly predicted by means of the external problem (Hagan et al., 2014, p. 2-12). Dependent on its characteristics and structure, a hidden layer performs a certain computation of the input and transforms it before it is passed on to the next layer (Haykin, 2009, p. 22).
As a first step in a multilayer network, the input - represented by the input layer - is applied to the neurons of the first hidden layer. The output of the first hidden layer is then applied as an input to the second hidden layer and so forth, until the output layer is reached. Like single-layer networks, multilayer nets are typically feedforward, that is, the input of a layer is constituted solely by the output of the previous layer (Haykin, 2009, p. 22; Anthony, 2001, p. 6). A simple, feedforward multilayer network is illustrated in figure 3.4.

The presence of one or more hidden layers makes multilayer networks much more powerful than single-layer networks and provides them with the ability to solve more complicated problems (Haykin, 2009, p. 22; Fausett, 1994, p. 14). Further and more detailed information on the topic of multilayer networks and the representational power of hidden layers can be found in chapters 5 and 6.

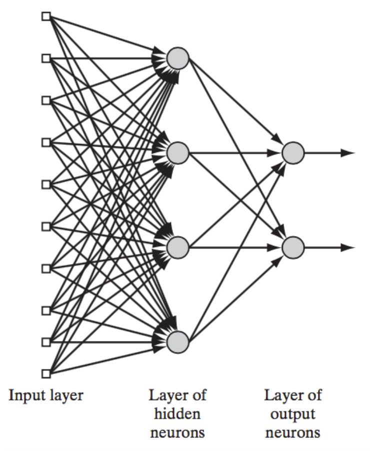

Fig. 3.4: Fully connected feedforward network with one hidden layer and one output layer by Haykin (2009, p. 22).

3.4.3 Recurrent Network

Another class of neural network architectures that is fundamentally different from single- and multilayer networks is represented by *recurrent networks.* Recurrent neural networks contain at least one *feedback loop* (Haykin, 2009, p. 23; Hagan et al., 2014, p. 2-14). In contrast to a feedforward network, in which the input is not affected by the previous activation of the net, the input of a recurrent net is subject to such internal inputs (feedback) which also influence the activation of the hidden units (Churchland and Sejnowski, 1992, p. 95). The feedback in a recurrent network can take many different forms. Neurons of the same layer can have lateral interactions, higher layers can project onto lower layers, etc. (Churchland and Sejnowski, 1992, p. 99). Also self-feedback is possible. This means that the output of a neuron is fed back to itself as an input (Haykin, 2009, p. 23). Adding feedback to a network has a huge impact on its learning abilities and performance. Although recurrent nets are much harder to train than feedforward networks, their architecture discloses several capacities that cannot be achieved by feedforward nets (Churchland and Sejnowski, 1992, p. 95; Haykin, 2009, p. 23). For instance, multiple time scales can be integrated into the layers (Churchland and Sejnowski, 1992, p. 95). Also, recurrent networks are much more similar to the way the brain works than feedforward networks.

Boltzmann machines and the subclass of restricted Boltzmann machines (as discussed in sections 6.4.2 and 6.4.3) exhibit some form of recurrence as they have *symmetric synaptic connections* between their neurons. However, since it is beyond the scope of this work, the topic of recurrent neural networks will not be discussed beyond that.

3.5 Types of Activation Functions

As mentioned in section 3.1, the activation function(s) used in a network decide about the form of its output. Several choices for such activation functions exist. A particular function is chosen according to both the character of the data and the desired form of the output (Hagan et al., 2014, p. 2-3; Bishop, 2006, p. 227). The entirety of activation functions that are typically used in ANN's can be divided into three categories: linear, threshold and sigmoid (Hinton, 1992, p. 145; Hagan et al., 2014, p. 2-3). As pointed out earlier, neurons in one layer of a neural net typically have the same activation function (Fausett, 1994, p. 17).

The simplest choice of activation function is given by a *linear activation function* (Fig. 3.5). The output of a linear unit in a neural net is proportional to the sum of its weighted inputs (Hinton, 1992, p. 145; Hagan et al., 2014, p. 2-4; Bishop, 2006, p. 228). Since a linear activation function just rescales its input, no squashing of the input is performed, i.e. the possible range of the neuron's output is not limited.

The next category of activation functions is given by *threshold functions*. The first threshold function for neural networks was introduced by McCulloch and Pitts (1943) and aimed at modelling the behavior of a single neuron in a biological nervous system. It takes the form of the *Heaviside step function* which is also known as the *binary step function* (Fig. 3.6) and is given by:

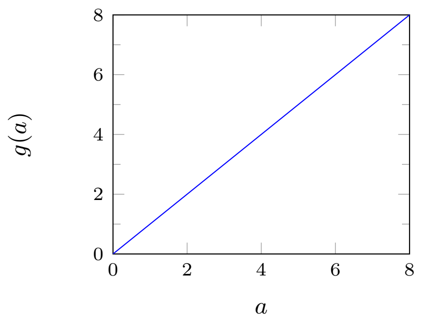

Fig. 3.5: Plot of a linear activation function

$$g(a) = \begin{cases} 0 & x < 0 \\ 1 & x \geq 0 \end{cases} \tag{3.4}$$

In general, the output of a threshold function is dependent on whether its total input is less or greater than some threshold value (Hinton, 1992, p. 145). Since the output is set at one of two values, threshold functions are usually used in nets that classify inputs into two distinct categories (Hagan et al., 2014, p. 2-4). Networks of such threshold neurons were studied by Rosenblatt under the name "perceptrons" and by Widrow and Hoff under the name "adalines" (Bishop, 1995, p. 84; Hagan et al., 2014, p. 2-4). Perceptrons will be discussed in more detail in chapter 4.

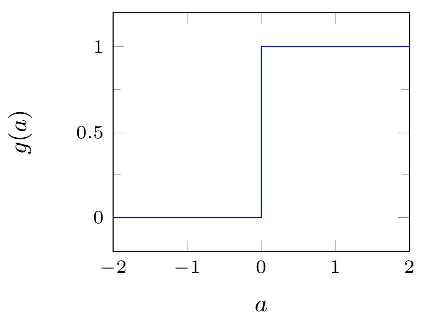

Fig. 3.6: Plot of the Heaviside step function

The third category of activation functions is constituted by continuous, typically *sigmoid activation functions* which are non-linear and more powerful than linear activation functions (Fausett, 1994, p. 17; Bishop, 1995, p. 83).[3] Sigmoid activation functions are commonly used in multilayer networks (cf. section 3.4.2) since they bear several advantages regarding training of the net by means of the backpropagation algorithm (cf. section 5.2) (Fausett, 1994, p. 17; Hagan et al., 2014, p. 2-5; Bishop, 1995, p. 83): not only are they differentiable, but they also possess a simple relationship between the values of the function and its derivative. This, in turn, considerably reduces the computational cost during training (Fausett, 1994, p. 17; Mitchell, 1997, p. 97). The most popular

[3] The term 'sigmoid' refers to the function's S-shape

sigmoid functions are the *logistic sigmoid* and the *hyperbolic tangent* (Fausett, 1994, p. 17; Bishop, 1995, p. 127). The logistic sigmoid (also called *binary sigmoid* (Fausett, 1994, p. 18)) is given by:

$$g(a) = \frac{1}{1 + exp(-a)} \tag{3.5}$$

It maps the input a (which can range from $-\infty$ to ∞) onto a real number in the interval $(0, 1)$ as an output (Hagan et al., 2014, p. 2-5; Anthony, 2001, p. 4). Hence, it is often used in neural networks whose desired output values are either binary or in the interval $(0, 1)$ (Fausett, 1994, p. 18). An illustration of the logistic sigmoid is given in Fig. 3.7.

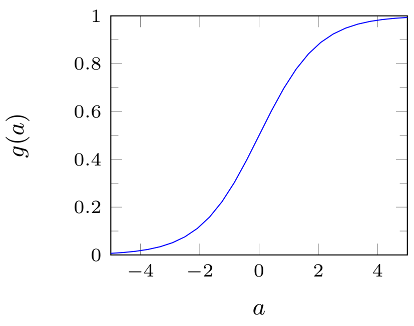

Fig. 3.7: Plot of the logistic sigmoid activation function

The hyperbolic tangent function ('tanh') is the second sigmoid function that is often used in multilayer networks. It is given by:

$$tanh(a) = \frac{exp(a) - exp(-a)}{exp(a) + exp(-a)} \tag{3.6}$$

Like the logistic sigmoid, the hyperbolic tangent maps the input a (which can range from $-\infty$ to ∞) onto a real number. However, different from the logistic sigmoid, this real number lies in the range $(-1, 1)$ (Fausett, 1994, p. 19; Bishop, 1995, p. 127). An illustration is given in Fig. 3.8.

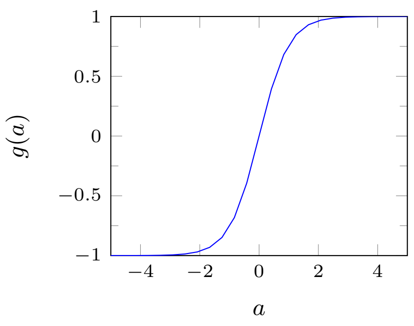

Fig. 3.8: Plot of the hyperbolic tangent activation function

Although all mentioned activation functions are only rough approximations of the behavior of biological neurons, it has been argued that the resemblance between real neurons and sigmoid neurons is greater than the resemblance between real and linear/threshold units (Hinton, 1992, p. 145).

4. THE PERCEPTRON

Due to its high impact, the perceptron is probably one of the most widely known early neural nets. It was proposed by Rosenblatt in 1958 as the first single-layer neural network that employed supervised learning. Block (1962, p. 123), who reviewed the properties of perceptrons in detail, describes their primary purpose as giving valuable information about the problem of how brain function can be explained by brain structure. Also Rosenblatt himself underlined the apparent connection between the perceptron and biological systems (Rosenblatt, 1958, p. 387).

The perceptron's high influence arose from several properties that could not be offered by any other network model at that time. The first and fundamental key advantage of the perceptron was its simple and efficient learning algorithm (Rosenblatt, 1958). Second, thanks to the precisely defined terms of the model, the perceptron's performance could be tested (Rosenblatt, 1958, p. 406; Block, 1962, p. 126). Furthermore, the model was complex enough to potentially offer interesting behavior while, at the same time, being simple enough to allow for an analysis and prediction of its performance (Rosenblatt, 1958, p. 406; Block, 1962, p. 126). Last, as mentioned above, it not only had a strong connection to biological systems, but was also consistent with the known biological facts (Rosenblatt, 1958, pp. 388-389; Block, 1962, p. 126).

4.1 Design/Functioning

The original perceptron was conceived as an approximate model of the retina (Rolls and Treves, 1998, p. 77; Fausett, 1994, p. 59). Therefore, it consisted of three layers of neurons: a retina of sensory cells (called S-points), a layer of association cells (called A-units) and a layer of response cells (called R-units) (Block, 1962, p. 123; Rosenblatt, 1958, p. 389; Fausett, 1994, p. 59). An illustration can be found in figure 4.1.

As a stimulus (originally thought of as a pattern of light and shadow) is presented to the retina, it activates the sensory cells which are connected to a set of association cells by random many-to-many connections in a feedforward fashion. Thus, impulses are conducted from the activated sensory cells to the connected association cells. These impulses can either be negative (inhibitory) or positive (excitatory). The association cells in turn are either connected to response units or to each other through modifiable weights (Arbib, 1987, p. 61; Block, 1962, p. 123; Rosenblatt, 1958, pp. 389-390). At this point, also backward connections are allowed. These most often take the form of inhibitory feedback connections from a response cell to the association cells from which it does **not** receive im-

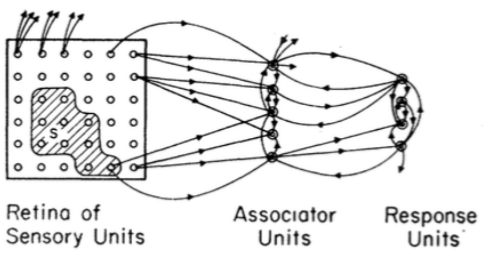

Fig. 4.1: Original organization of a perceptron by Block (1962, p. 127).

pulses (Rosenblatt, 1958, p. 390).
If the total signal arriving at an association cell exceeds a certain threshold, it becomes activated and sends an impulse to its connected cells. Since the association cells respond on an all-or-nothing basis, they output either a 1, if the threshold is surpassed, or a 0, if it is not (Fausett, 1994, p. 59; Rolls and Treves, 1998, p. 77; Rosenblatt, 1958, p. 389; Block, 1962, p. 123).[1]
Then, due to the connections between association and response cells, the output of the association cells is passed on to the response cells. Like the association cells, the response cells sum their total input and respond on an all-or-nothing basis, dependent on whether their activation threshold is exceeded or not (Mitchell, 1997, p. 86; Fausett, 1994, p. 59; Rolls and Treves, 1998, p. 77; Rosenblatt, 1958, p. 389; Block, 1962, p. 123).

Since only the weights from associator to response units could be adjusted in the original model (Fausett, 1994, p. 60; Arbib, 1987, p. 64), the modern simplification of it (illustrated in Fig. 4.2) is limited to only a single output neuron (Mitchell, 1997, p. 86; Haykin, 2009, p. 48). Rosenblatt called such models *simple perceptrons* (Rosenblatt, 1962). In accordance with the original model, the simple perceptron first computes a linear combination of the inputs. In mathematical terms this operation can be defined as:

$$v = \sum_{j=1}^{d} w_j x_j \tag{4.1}$$

As in the original model, the simplified perceptron responds dependent on whether the resulting sum v exceeds the perceptron's threshold θ or not. The output furthermore depends on the kind of activation function that is chosen for the output units. As mentioned in section 3.5, the perceptron makes use of *threshold* neurons. Therefore, its output is of the following form (Mitchell, 1997, p. 86)[2]:

$$y(v) = \begin{cases} +1 & v > \theta \\ -1 & v \leq \theta \end{cases} \tag{4.2}$$

[1] The choice of activation function (and therefore the output of the association cells) can also be different.
[2] The exact form of the threshold function can also be different, compare e.g. Rolls and Treves (1998, p. 77), Bishop (1995, p. 99).

Since the goal of the perceptron is the correct classification of its externally applied stimuli, it must be able to learn (Haykin, 2009, p. 49; Rolls and Treves, 1998, p. 75; Bishop, 1995, p. 98; Fausett, 1994, p. 60). This learning is achieved by modifying the synaptic weights between the association and response cells according to the perceptron learning rule (Rosenblatt, 1958, p. 391; Mitchell, 1997, p. 86; Fausett, 1994, p. 59). In short, given the simplified model of the perceptron, the learning rule works as follows: first, the net calculates the response of the output unit for each input pattern. By comparing the obtained output to a given target output, an error between the actual and the desired output is computed. The synaptic weights are then changed in order to minimize the error (Fausett, 1994, p. 59; Lippmann, 1987, p. 16; Rolls and Treves, 1998, p. 76). A detailed description of the learning rule can be found in section 4.2.

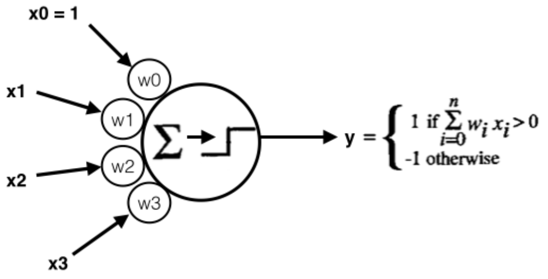
x0 = 1
x1
x2
x3
w0
w1
w2
w3
Σ
$y = \begin{cases} 1 \text{ if } \sum_{i=0}^{n} w_i x_i > 0 \\ -1 \text{ otherwise} \end{cases}$

Fig. 4.2: Simplified model of the original perceptron with weights $\vec{w} = [w_0, w_1, w_2, w_3]$. The perceptron is subject to an input vector $\vec{x} = [+1, x_1, x_2, x_3]$. The output y is 1 if the perceptron's weighted input is greater than 0 and -1 otherwise. Adapted from Mitchell (1997, p. 87).

4.2 Learning Algorithm

For analyzing the learning algorithm of the perceptron, it is sufficient to consider just the simplified version with only one neuron (Fig. 4.2). In this simple form, the perceptron's ability to learn to recognize patterns is restricted to patterns with only two classes c_1 and c_2 (Lippmann, 1987, p. 16; Haykin, 2009, p. 48). Accordingly, the perceptron should output +1 if the presented stimulus belonged to class c_1 and -1 if it belonged to class c_2.[3]
To illustrate the effect of the learning algorithm, it is helpful to rewrite the perceptron's input and output. As can be seen in equation 4.2, the perceptron's output is dependent on a threshold θ. To simplify notation, the threshold can be moved to the other side of the inequality (by a simple subtraction) and replaced by the perceptron's *bias* b (yielding $b = -\theta$). Thus, equation 4.2 can be reformulated as:

$$y(v) = \begin{cases} +1 & v + b > 0 \\ -1 & v + b \leq 0 \end{cases} \tag{4.3}$$

As mentioned in section 3.3, the bias b can be included in the linear combination

[3] The output/class assignment could also be the other way around.

of the perceptron's inputs. It was further mentioned that the d input signals (including a fixed input of $+1$ for the bias) form a vector:

$$\vec{x} = [+1, x_1, x_2, ..., x_d]^\top \in \mathbb{R}^{d+1}$$

Correspondingly (with the bias treated as a synaptic weight), the weight vector can be defined as:

$$\vec{w} = [b, w_1, w_2, ..., w_d]^\top \in \mathbb{R}^{d+1}$$

By rewriting the perceptron's input and output in this way, equation 4.1 can be reformulated as:

$$\begin{aligned} v &= \sum_{j=0}^{d} w_j x_j \\ &= \vec{w}^\top \vec{x} \end{aligned} \tag{4.4}$$

In this sense, the perceptron can be viewed as representing a hyperplane which functions as a decision surface between the two classes c_1 and c_2. The hyperplane is given by the equation $\vec{w}^\top \vec{x} = 0$ (Haykin, 2009, p. 50; Mitchell, 1997, p. 86). The perceptron outputs a $+1$ if the presented stimulus is lying on one side of the hyperplane and a -1 if it is lying on the other. An illustration is given in figure 4.3.

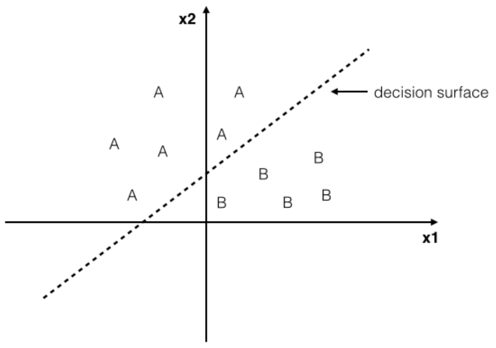

Fig. 4.3: Illustration of how a perceptron divides the space spanned by the input into two regions separated by a hyperplane. Adapted from Lippmann (1987, p. 16).

The equation of the decision surface (and therefore its position) is dependent on both the synaptic weights and the threshold of the perceptron. By replacing the threshold with the perceptron's bias and further including the bias in the total input, the bias and weights can be learned using the same learning algorithm. Regarding figure 4.3, the effect of the bias is simply to shift the decision surface away from the origin (Haykin, 2009, p. 49).

Learning a perceptron is performed iteratively, using a labeled data set $D = \{(\vec{x_i}, t_i)\}, \vec{x_i} \in \mathbb{R}^{d+1}$, where t_i is the desired target output for training input $\vec{x_i}$. First, the weight vector $\vec{w}$ is initialized with small random numbers (Mitchell, 1997, p. 88; Lippmann, 1987, p. 16). Next, one of the training input vectors $\vec{x_i}$ together with its desired target output t_i is applied to the perceptron and the perceptron's output is calculated (Fausett, 1994, p. 59; Lippmann, 1987, p. 16). The computed output y_i is then compared to the desired output t_i. The difference between the desired target output and the actual output represents an error signal (Fausett, 1994, p. 59; Rolls and Treves, 1998, p. 76). If such an error occurred (meaning that the perceptron misclassified the example), the synaptic weights $\vec{w_i}$ of the perceptron are adapted according to the rule (Lippmann, 1987, p. 16):

$$\vec{w_i} \longleftarrow \vec{w_i} + \eta[t_i - y_i]\vec{x_i}$$

In this equation, η is a small positive constant representing the *learning rate* of the perceptron. It decides about the degree to which the weights are adapted at each step (Lippmann, 1987, p. 16). When choosing a value for η, two conflicting requirements have to be kept in mind (Lippmann, 1987, p. 16). On the one hand, η should be chosen such that it allows for a *fast adaption* with respect to real changes in the input distributions. This requires a large value for η. On the other, in order to provide solid weight estimates, it must allow for an *averaging* of already presented inputs. This, in turn, requires a small value for η.

The above mentioned learning procedure is repeated until no more changes of the weights occur (or in other words no error), meaning that the perceptron classifies all training examples correctly (Mitchell, 1997, p. 88). An example of the application of the learning procedure is given in figure 4.4.

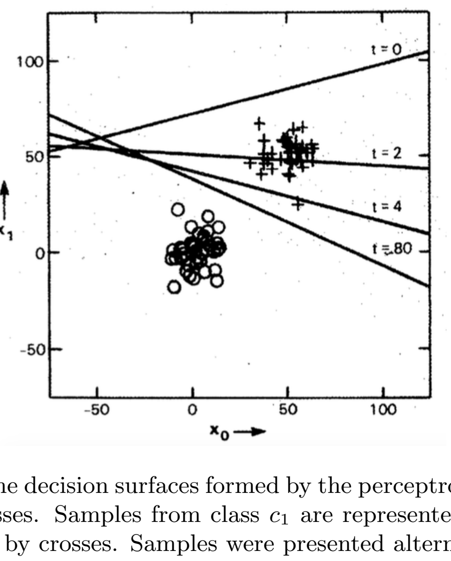

Fig. 4.4: Example of the decision surfaces formed by the perceptron learning algorithm with two classes. Samples from class c_1 are represented by circles, samples from class c_2 by crosses. Samples were presented alternately until 80 inputs had been presented (t = 80) using a learning rate of $\eta = 0.01$. By Lippmann (1987, p. 17).

What made the perceptron's learning algorithm special was its *proof of convergence* (known as the perceptron convergence theorem). Rosenblatt (1962) demonstrated that the iterative learning procedure converges to the weights that allow the net to classify all input patterns correctly, given that they stem from two *linearly separable* classes (i.e. classes that lie on opposite sides of a hyperplane). Moreover, the weights will be found in a finite number of training steps. If the training examples are not linearly separable, however, convergence is not assured. This is further analyzed in section 4.3.

4.3 Representational Power and Limitations of the Perceptron

As mentioned above, a perceptron can only learn to recognize patterns that are linearly separable. Although this clearly limits the range of functions to which it can be applied, the perceptron is still of high historical as well as practical importance (Bishop, 1995, p. 88; Haykin, 2009, p. 66). Furthermore, even though the limitations of perceptrons motivated the use of multilayer networks (Bishop, 1995, p. 85), they also have certain advantages over them. One advantage, for instance, is their relatively quick training (Bishop, 1995, p. 88). As will be shown in following chapters, training of more complex networks often involves a high computational effort.

4.3.1 Representational Power

Perceptrons can be trained to classify input patterns of four basic boolean functions: AND, OR, ¬AND and ¬OR (Fausett, 1994, p. 44; Mitchell, 1997, p. 87). The truth table of the AND function is given by:

x_1	x_2	*Output*
1	1	+1
1	-1	-1
-1	1	-1
-1	-1	-1

The desired responses for the logic function AND are illustrated in figure 4.5. As can be seen in the figure, it is possible to separate the input coordinates for which an output of +1 is required from those that require an output of -1 by a hyperplane. Therefore, as mentioned in section 4.4, the problem is linearly separable and can thus be solved by a simple perceptron. Important to note is that several solutions for a suitable hyperplane exist. Also, given a perceptron with just two inputs, several choices for w_0, w_1 and w_2 exist that will result in exactly the same hyperplane (Fausett, 1994, p. 44). The decision about which side of the hyperplane corresponds to an output of +1 and which to an output of -1 is given by the sign of w_0 (Fausett, 1994, p. 44). One possible solution for a hyperplane with $w_0 = -1, w_1 = 1, w_2 = 1$ is illustrated in figure 4.5.

Another boolean function that can be solved by a simple perceptron is the OR

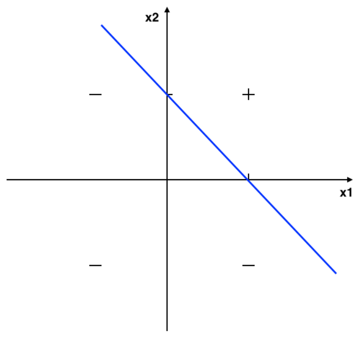

Fig. 4.5: Desired responses for the logic function AND together with a possible solution of a decision surface, adapted from Fausett (1994, p. 45).

function. Its truth table is given by:

x_1	x_2	*Output*
1	1	+1
1	-1	+1
-1	1	+1
-1	-1	-1

The desired responses of the function as well as a possible solution with $w_0 = 1, w_1 = 1, w_2 = 1$ are illustrated in figure 4.6.

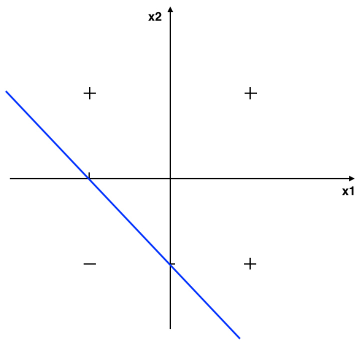

Fig. 4.6: Desired responses for the logic function OR together with a possible solution of a decision surface, adapted from Fausett (1994, p. 45).

4.3.2 Limitations

A simple example of a boolean function that cannot be represented by a simple perceptron is the XOR function whose value is +1 if and only if $x_1 \neq x_2$ (Bishop, 1995, p. 86; Rolls and Treves, 1998, p. 79). Its truth table is given by:

x_1	x_2	*Output*
1	1	-1
1	-1	+1
-1	1	+1
-1	-1	-1

As can be seen in figure 4.7, it is not possible to define a hyperplane that separates the inputs for which an output of +1 is required from those for which an output of -1 is required. Thus, the XOR problem is not linearly separable and therefore not solvable for a simple perceptron.

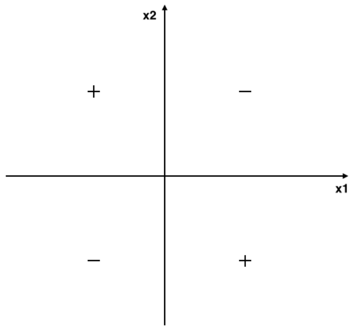

Fig. 4.7: Desired responses for the logic function XOR by Fausett (1994, p. 47).

The XOR problem is just one example of the limitations faced by perceptrons. A formal mathematical analysis of the entirety of the perceptron's limitations was given by Minsky and Papert (1969, 1988) in their book "Perceptrons". In their book, Minsky and Papert discuss several forms of perceptrons and, by means of detailed mathematical analyses, point out multiple problems that cannot be solved by any of them.
After the initial enthusiasm for perceptrons and especially the fact that initially unstructured networks could perform many interesting tasks (and even exhibit certain aspects of learning), research in the field was disrupted (Fausett, 1994, pp. 23, 25). This was also due to the fact that Minsky and Papert questioned not only the capabilities of perceptrons, but also the capabilities of its multilayer variants (Minsky and Papert, 1988, pp. 231-232):

> *The perceptron has shown itself worthy of study despite (and even because of!) its severe limitations. It has many features to attract attention: its linearity; its intriguing learning theorem; its clear paradigmatic simplicity as a kind of parallel computation. There is no reason to suppose that any of these virtues carry over to the many-layered version. Nevertheless, we consider it to be an important research problem to elucidate (or reject) our intuitive judgement that the extension to multilayer systems is sterile.*

Although research in the field of neural networks did not stop entirely, it was not until the 1980s that Minsky's and Papert's concerns were refutet and enthusiasm in the field renewed (Fausett, 1994, p. 25; Rolls and Treves, 1998, p. 87).

5. MULTILAYER PERCEPTRON

As pointed out in section 4.3.2, perceptrons with only a single layer of adjustable weights face several problems, as they can only represent linearly separable patterns. To allow for the representation of a broader range of functions (and thus to overcome the perceptron's limitations), a lot of effort has been put into the development of fully functioning networks with several layers of adjustable weights (Bishop, 1995, p. 116; Haykin, 2009, p. 123; Mitchell, 1997, p. 88). Although they do not always make use of threshold activation functions (as perceptrons do), such networks are typically called *multilayer perceptrons* (Bishop, 1995, p. 116). Despite the concerns expressed by Minsky and Papert (cf. section 4.3.2), multilayer perceptrons overcame the perceptron's limitations and could successfully be trained to represent *any* continuous function by means of just two layers of weights (Mitchell, 1997, p. 87; Bishop, 1995, p. 116).

5.1 Design/Functioning

As described in section 3.4, a multilayer network consists of an input layer, one or several hidden layers and an output layer (Fig. 5.1). Specific to the multilayer perceptron is the use of a differentiable, nonlinear activation function and its entirely feedforward structure (Mitchell, 1997, p. 96; Haykin, 2009, p. 123; Bishop, 1995, p. 121). If the activation function used in the net was linear, an equivalent network without any hidden layers could be found. This is due to the fact that a succession of linear transformations still produces only linear functions (Bishop, 1995, p. 121; Mitchell, 1997, p. 96). Therefore, multilayer perceptrons make use of a sigmoid function which typically takes the form of the logistic sigmoid (cf. section 3.5)[1].

As depicted in section 3.4, the activation of the *input neurons* simply represents the information that is fed into the network. Similar to the perceptron, the activation s_h of each *hidden* neuron h is determined by a linear combination of its weighted inputs (Hinton, 1992; Bishop, 1995, p. 118; Mitchell, 1997, p. 96). In mathematical terms, this is described by equation 3.2. The output y_h of a hidden neuron is determined by applying the logistic simgoid function (denoted as g) to the activation s_h:

$$\begin{aligned} y_h &= g(s_h) \\ &= \frac{1}{1 + exp(-s_h)} \end{aligned} \tag{5.1}$$

[1] Another popular choice for the activation function of a multilayer perceptron is the tanh function as introduced in section 3.5. The choice of activation function is dependent on the form of the data and especially the target values.

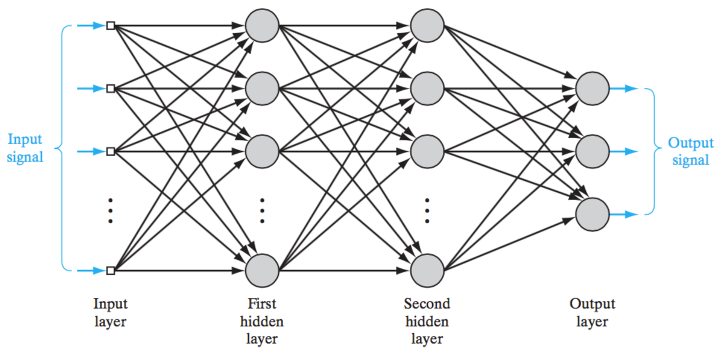

Fig. 5.1: Architectural graph of a fully connected multilayer perceptron with two hidden layers by Haykin (2009, p. 124).

The activation and response of the output units is computed in the same fashion.

Besides being differentiable, the logistic sigmoid activation function has another useful property: its derivative can be expressed easily by means of its output (Fausett, 1994, p. 17; Mitchell, 1997, p. 97). This characteristic is especially useful for the training of multilayer perceptrons with the backpropagation algorithm, as will become evident in section 5.2.

The mentioned hidden units play a key role in the functioning of a multilayer perceptron. It is this role that distinguishes multilayer perceptrons from Rosenblatt's perceptron (Haykin, 2009, p. 126). The importance of the hidden layers arises from the fact that they are free to construct their *own* representations of the data (Hinton, 1992, p. 146). As this is done by detecting explanatory factors in the given input, each hidden unit functions as a *feature detector* (Haykin, 2009, p. 126). The combination of the outputs of *all* feature detectors can be interpreted as a transformation of the input signal into a so called "feature space". In this space, the input classes can often be separated more easily so that the output layer is provided with a problem that is linearly separable, and can thus be solved (Rolls and Treves, 1998, p. 87; Haykin, 2009, p. 126).

Two types of signals are important for the functioning of a multilayer perceptron (cf. Fig. 5.2). The first type are the *input signals*. The input of a network is initially applied to its input neurons and then propagated *forward* through the network by consecutively calculating the output of each neuron, including the output layer (Haykin, 2009, p. 125; Bishop, 1995, p. 120).
The second type of signal is represented by the *error signals*. Error signals are a measure of how closely the overall output of the network currently matches the desired output. Accordingly, an error signal originates at an output neuron of the network (Haykin, 2009, p. 125). While training of the net with the backpropagation algorithm, the error signal is propagated *backward* through the net, layer by layer, starting with the output layer (Rolls and Treves, 1998, p. 87).

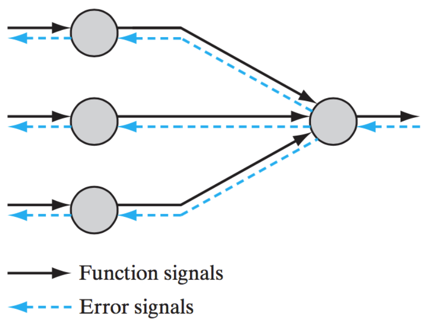

Fig. 5.2: llustration of the directions of two basic signal flows in a multilayer perceptron: forward propagation of input signals and backpropagation of error signals. By Haykin (2009, p. 125).

5.2 Backpropagation Algorithm

As mentioned in section 4.3.2, the limitations of the perceptron and (to some extend) also the criticism of Minsky and Papert caused a decline of interest in neural networks in the 1970s. Another factor that contributed to the drop in attention was the absence of an efficient method for training a multilayer network (Fausett, 1994, p. 25). Although such a method, known as the backpropagation algorithm, was already discovered in 1974 by Paul J. Werbos, it took years until it gained attention (Hinton, 1992; Fausett, 1994, p. 25). Only after it was rediscovered independently by several scientists (e.g. by LeCun in 1986) and finally publicized by Rumelhart et al. (1986), widespread interest in neural networks has reappeared (Rolls and Treves, 1998, p. 87; Fausett, 1994, p. 289).

Similar to the perceptron, the aim of learning a multilayer perceptron is to find a set of weights that enables the network to produce the right output vector for each input vector (Rumelhart et al., 1986). In this regard, the learning problem can be viewed as *searching* an extensive *hypothesis space* that consists of all possible weight values for all neurons in the network (Mitchell, 1997, p. 97). The key idea behind this search (and therefore also the basis of the backpropagation algorithm) is the use of *gradient descent*. This technique will be described in more detail in the next section.

The backpropagation algorithm basically consists of three stages. First, as mentioned in section 5.1, the input is propagated *forward* through the net, by successively computing the output of each neuron in the network. Next, the error of each output unit is computed and then propagated *backward* through the net, layer by layer. In a last step, the weights of all hidden and output neurons are updated.

5.2.1 Gradient Descent

In order to derive a successful weight learning rule for the output and hidden units of a multilayer perceptron, a measure for the total training error of the network for a hypothesis $\vec{w}$ (i.e. a weight vector) needs to be defined. This total error E for a training example d is commonly defined as (Mitchell, 1997, p. 97; Haykin, 2009, p. 127; Bishop, 2006, p. 242):

$$E_d(\vec{w}) = \frac{1}{2} \sum_{k \in C} (t_k - y_k)^2 \tag{5.2}$$

where C is the set of all output neurons and y_k and t_k are the current and target output value of output neuron k on training example d.

Now, gradient descent is used to find a hypothesis that minimizes E. This is done by modifying the initial weight vector of the network repeatedly in small steps into the direction that minimizes the error function most quickly (Mitchell, 1997, p. 91). The direction in which the function *increases* most quickly is given by the *gradient* of the error function with respect to each weight in the network (Rumelhart et al., 1986, p. 534; Mitchell, 1997, p. 91; Fausett, 1994, p. 296). In mathematical terms, the gradient can be described as (Mitchell, 1997, p. 91):

$$\nabla E(\vec{w}) = [\frac{\partial E}{\partial w_0}, \frac{\partial E}{\partial w_1}, ..., \frac{\partial E}{\partial w_n}]$$

Since the gradient describes the direction in which E *increases* most quickly, the *negative* of the gradient $-\nabla E(\vec{w})$ gives the direction in which the error *decreases* most quickly (Mitchell, 1997, p. 91; Fausett, 1994, p. 296).
This knowledge (i.e. knowing in which direction the total error of the network decreases most rapidly) is used for successfully updating the weights of a multilayer perceptron during training as described in section 5.2.2.

5.2.2 Algorithm

The application of the backpropagation algorithm starts with initializing all weights in the network to small random numbers. Then, the different stages of the algorithm are executed iteratively until a certain stopping criterion is met (Mitchell, 1997, p. 98; Fausett, 1994, p. 294).

For the forward step (i.e. step one) of the algorithm, a single training example is presented to the net and the outputs of all hidden and output neurons are calculated according to equation 5.1 (Bishop, 2006, p. 243; Rumelhart et al., 1986, p. 533; Mitchell, 1997, p. 98; Fausett, 1994, p. 294).
For step two (also called the backward step), the error of each hidden and output neuron in the network needs to be computed. When computing this error, two cases need to be distinguished. First, the error signal δ_k of each output unit k is computed. The exact form of δ_k for both the output and hidden units follows from the derivation of the weight adaption rule which can be found in Mitchell (1997, p. 102) or Haykin (2009, p. 131). According to this derivation, the error signal of an output unit is given by (Mitchell, 1997, p. 98; Fausett, 1994, p. 295; Haykin, 2009, p. 131):

$$\delta_k = g'(s_k)(t_k - y_k) \tag{5.3}$$

As can be seen in the equation, the error of an output neuron k is simply the derivative of its activation function, multiplied by the difference between the target and current output of the neuron. As the logistic sigmoid is used as the activation function, the derivate $g'(s_k)$ can easily be computed:

$$\begin{aligned} g(a) &= \frac{1}{1+exp(-a)} \\ g'(a) &= g(a)[1-g(a)] \end{aligned} \tag{5.4}$$

Second, the error signal of each hidden unit h must be computed. Since the training examples do not directly provide target outputs for the hidden units, this is more complicated than in the previous case (Mitchell, 1997, p. 99; Haykin, 2009, p. 131). The error δ_h of a hidden neuron h is given by:

$$\delta_h = g'(s_h) \sum_k \delta_k w_{kh} \tag{5.5}$$

Again, the first term of equation 5.5 corresponds to the derivative of the hidden neuron's activation function (and therefore to equation 5.4). The second term sums up all the errors of the neurons k that are directly influenced by h and weights each of those errors by w_{kh}, the weight from neuron h to neuron k (Mitchell, 1997, p. 99; Haykin, 2009, p. 134; Bishop, 2006, p. 244). In this respect, the weight's w_{kh} represent the extend to which hidden neuron h accounts for the error in neuron k (Mitchell, 1997, p. 99). Altogether, the second term of equation 5.5 represents a *backward* propagation of errors from neurons higher up in the network to neuron h (Haykin, 2009, p. 131; Bishop, 2006, p. 244). This process is illustrated in figure 5.3.

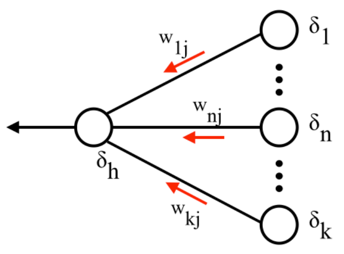

Fig. 5.3: Illustration of the calculation of δ_h by propagating the errors from those neurons k backward, that are directly influenced by h. Adapted from Bishop (2006, p. 244) and Haykin (2009, p. 134).

As a last step of the algorithm, the weights of the hidden and output neurons are adapted according to the method of gradient descent (cf. section 5.2.1). Due to the use of gradient descent, for each training example d, the correction $\triangle w_{ji}$ applied to the weight w_{ji} is proportional to the partial derivative of the total error E_d with respect to w_{ji} (Haykin, 2009, pp. 130-131; Mitchell, 1997,

p. 99):

$$\triangle w_{ji} = -\eta \frac{\partial E_d}{\partial w_{ji}} \tag{5.6}$$

As already mentioned, the partial derivative of the error function with respect to each weight represents the direction in which E_d *increases* most quickly. Therefore, (i.e. to minimize the total error) the *negative* of this gradient is used in equation 5.6. Similar to the weight-tuning rule of the perceptron, η represents the *learning rate* of the multilayer perceptron.
When deriving an expression for $\frac{\partial E_d}{\partial w_{ji}}$, the previously established error terms reappear:

$$\begin{aligned}\triangle w_{ji} &= -\eta \frac{\partial E_d}{\partial w_{ji}} \\ &= \eta\, \delta_j\, x_{ji}\end{aligned} \tag{5.7}$$

The whole derivation can be found in Mitchell (1997, p. 102) or Haykin (2009, p. 131). To successfully train a multilayer perceptron, the mentioned stages of the algorithm are usually repeated thousands of times until the stopping criterion is met (Mitchell, 1997, p. 99; Fausett, 1994, p. 296). Important to note is that the weights do not have to be updated after every single presentation of a training example. A different mode of the algorithm (called "batch mode") exists, which accumulates the weight updates $\triangle w_{ji}$ over the presentation of *all* training examples before applying them to the weights (Fausett, 1994, p. 296; Rumelhart et al., 1986, p. 535; Haykin, 2009, p. 127).

5.3 Representational Power of Multilayer Perceptrons

Different to Rosenblatt's perceptron, multilayer perceptron's can represent a wide range of functions. First, every possible boolean function can be approximated, independent of whether the function is linearly separable or not. Therefore, multilayer perceptrons can also solve the XOR problem (cf. Fig. 4.7), which could not be solved by a perceptron (Rolls and Treves, 1998, p. 89; Mitchell, 1997, p. 105).
Second, a multilayer perceptron with just *one* layer of hidden neurons can approximate any bounded continuous function to an arbitrary degree of accuracy (Hornik et al., 1989; Cybenko, 1989). Important to note, however, is that the corresponding theorem presented by Hornik et al. (1989) and Cybenko (1989) applies to multilayer perceptrons with sigmoid neurons in the hidden, and linear neurons in the output layer. Furthermore, the required number of hidden neurons in the network is not known in advance but determined by the problem. Last, it was proven by Cybenko that *any* function can be approximated to an arbitrary degree of accuracy by a multilayer perceptron with *two* layers of hidden units (as cited in Mitchell 1997, p. 105). As for the previous theorem, the hidden layers consist of sigmoid neurons and the output layer of linear neurons.

5.4 Autoencoder

The common multilayer perceptron employs supervised learning in order to learn a mapping between the input and the output space by means of labeled training examples. Nevertheless, multilayer perceptrons have also been applied to unsupervised learning in order to perform *dimensionality reduction* of high-dimensional data (Bishop, 2006, p. 593; Hinton et al., 2006, p. 504; Bengio et al., 2013, p. 1811). Such multilayer perceptrons are called *autoencoders* (or also auto-associator or Diabolo networks).[2] Regarding representation learning, autoencoders are especially interesting because they learn some internal representation of the input that often captures the key factors of variation in the data (Bengio, 2009, pp. 45-46).
Although autoencoders were originally seen as a dimensionality reduction technique, they have recently taken center stage in the research area of *deep learning* where they are stacked in order to build deep neural networks. Further information on this can be found in chapter 6.

5.4.1 Design

The basic model of an autoencoder consists of an input layer with d input neurons, a hidden layer with $M < d$ hidden neurons (if applied to the task of dimension reduction), and an output layer with d output units (illustrated in Fig. 5.4).[3] By means of its architecture, an autoencoder is trained to *reconstruct* its input as accurately as possible. Thus, the target values for the input vectors are just the input vectors themselves. During training, the autoencoder learns some representation $c(x)$ of the input x such that the input can be reconstructed from that representation (Bengio, 2009, p. 45; Bishop, 2006, p. 593; Sirois, 2004, p. 134).

By using fewer hidden than input neurons, the hidden layer acts as a *bottleneck*, enforcing the network to encode its input as efficiently as possible (Bengio et al., 2013, p. 1811; Bourlard and Kamp, 1988, p. 292). In this connection, the first part of the network (see Fig. 5.4) is regarded as an *encoder*, transforming the high-dimensional input vector x into a represention $c(x)$ (also called feature vector or code) with a lower dimension. The second part constitutes the *decoder* which reconstructs the input from the representation $c(x)$ and thus represents a mapping from the feature space back into the input space (Bengio et al., 2013, p. 1810; Hinton et al., 2006, pp. 504-505).

5.4.2 Functioning

As mentioned in the beginning of section 5.4.1, an autoencoder aims at reconstructing its original input as accurately as possible. In order to do so, the

[2] Another popular method for dimensionality reduction that has several things in common with autoencoders is *principal component analysis*, commonly called PCA (see e.g. Bishop (2006)).
[3] It is also possible to use a hidden layer with more hidden than input neurons ($M > d$). Such autoencoders are called *overcomplete*.

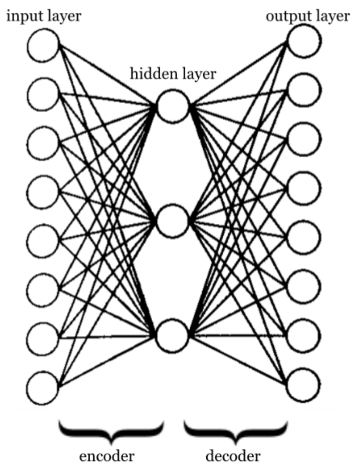

Fig. 5.4: Illustration of a basic autoencoder with d input, d output and M hidden units. Adapted from Mitchell (1997, p. 107).

parameters of the encoder and decoder are determined simultaneously by minimizing the *reconstruction error* J of the network. This error measures the discrepancy between the original input vectors and their reconstructions. Typically, the mean squared error is chosen as a measure for the reconstruction error (Bengio et al., 2013, p. 1810; Bishop, 2006, p. 593; Bourlard and Kamp, 1988, p. 292; Hinton et al., 2006, p. 506). For input vector $x_k (k = 1, 2, ..., N)$ and reconstruction $r_k (k = 1, 2, ..., N)$ the mean squared error is given by:

$$J = \sum_{k=1}^{N} ||x_k - r_k||^2 \tag{5.8}$$

Since the number of hidden neurons is smaller than the number of output neurons, the reconstruction r constructed from the representation $c(x)$ cannot be perfect for all possible input vectors (Bishop, 2006, p. 593; Bengio, 2009, p. 46). Therefore, the autoencoder generalizes in terms of finding a representation $c(x)$ that produces a low reconstruction error for the *training* examples but a high reconstruction error for other, arbitrary inputs (Bengio et al., 2013, p. 1810).

When the number of hidden units is not restricted to $M < d$, it is furthermore important to include some type of constraint that prevents the hidden layer from only learning the identity function. Learning the identity function means that the autoencoder would simply *copy* the input without extracting any meaningful features. This, of course, would always yield a reconstruction error of zero (Bengio et al., 2013, p. 1810; Bengio, 2009, p. 46).
Depending on the type of autoencoder, various options for such constraints exist. For instance, the representation $c(x)$ can be forced to have a lower dimension than the input, as employed in classical autoencoders (Bengio et al., 2013, p. 1811).

6. DEEP LEARNING

Deep learning (also known as hierarchical or deep structured learning) is a relatively new field of machine learning that became popular after the publication of an efficient learning algorithm in 2006 (cf. Hinton et al., 2006). By combining simpler concepts into more complex ones, deep learning algorithms and architectures learn *multiple levels of representation*. Since 2006, the newly developed techniques have had a big impact on several traditional AI applications such as natural language processing or object recognition (Hinton et al., 2006; Huang and LeCun, 2006; Ciresan et al., 2012; Lee et al., 2009; Deng et al., 2013b; Bengio et al., 2003; Collobert et al., 2011). Furthermore, companies like Google or Microsoft are investing in deep learning research and employing their results in their own products (e.g. Google Goggles or Google Image Search) (Deng et al., 2013b; Weston et al., 2010; Ng et al., 2015; Bengio and Heigold, 2014; Satkin and Hebert, 2013).

6.1 Definition

As mentioned in section 2.1, a representation learning algorithm discovers explanatory factors or features in the data it is given. A *deep learning* algorithm is a specific type of representation learning algorithm that learns a *hierarchy* of features by means of several layers of hidden neurons. In this hierachy, higher-level features are formed by the composition of lower-level features (Bengio, 2013, p. 2; LeCun et al., 2015, p. 436). An illustration of the differences in learning between classical machine learning, general representation learning and deep learning is given in figure 6.1.

The automatic learning of different levels of abstraction represents the key idea behind deep learning (Bengio, 2013, p. 3). It allows a system to learn very complex functions directly from the data, without the need for hand-coded features (Bengio, 2009, p. 6; LeCun et al., 2015, p. 436). This becomes especially important when dealing with problems like the chair challenge (cf. Fig. 2.2). As described in section 2.1, it is often difficult for humans to specify the low-level features of high-level abstractions like a chair, especially when they have to be defined in terms of raw sensory input (e.g. pixel values when aiming at detecting chairs in images). By combining simple representations to higher and more abstract ones, deep learning can solve this problem (Bengio et al., 2015, p. 7). An easy example of such a combination of simpler concepts is the classification of an image. Here, as can be seen in figure 6.2, the learned features of the first layer of the network typically represent the presence or absence, location, and orientation of edges in the picture. The second layer combines the different edges to form corners and contours, which are in turn combined to form object

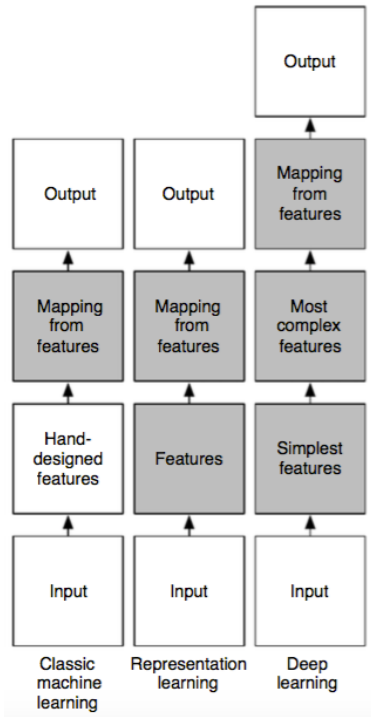

Fig. 6.1: Flow-charts showing how the different parts of an AI architecture in 1. classical machine learning, 2. representation learning and 3. deep learning are related to each other. Shaded boxes indicate components that are able to learn from data. Adapted from Bengio et al. (2015, p. 11).

parts in the third layer. The output layer finally identifies the object by means of its learned parts.

6.2 Design

As illustrated in figure 6.2, a deep learning model consists of multiple layers of simple modules that are stacked on top of each other. Each module represents a processing stage of the network and thus performs some (non-linear) transformation of its input (LeCun et al., 2015, p. 438; Bengio, 2009, p. 7; Deng and Yu, 2014, p. 201). This transformation always happens with the objective of transforming the output of the previous module into a new representation that increases both the selectivity and invariance of the old (given) representation (Bengio, 2009, p. 7; LeCun et al., 2015, p. 438). Thereby, each level learns a representation that is slightly more abstract than the one in the previous level. By stacking enough layers of such non-linear transformations on top of each other, a deep neural architecture can learn extremely complex functions (LeCun et al., 2015, p. 436).

The specific form of the single modules depends on the choice of deep archi-

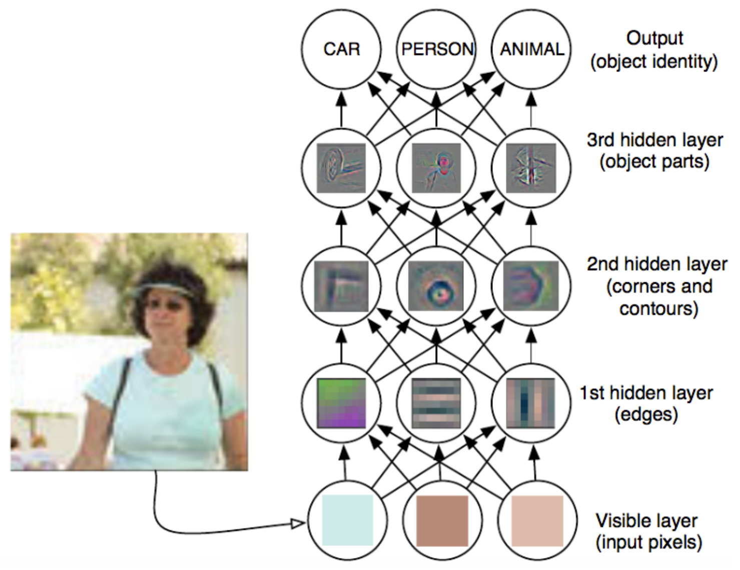
CAR
PERSON
ANIMAL
Output (object identity)
3rd hidden layer (object parts)
2nd hidden layer (corners and contours)
1st hidden layer (edges)
Visible layer (input pixels)

Fig. 6.2: Illustration of how a deep learning model breaks a desired complicated mapping into a series of nested simple mappings, each described by a different layer of the model. The input is presented at the visible layer. Then a series of hidden layers extracts increasingly abstract features from the image. The images here are visualizations of the kind of feature represented by each hidden unit. Given the pixels, the first layer can easily identify edges, by comparing the brightness of neighboring pixels. Given the first hidden layers description of the edges, the second hidden layer can easily search for corners and extended contours, which are recognizable as collections of edges. Given the second hidden layers description of the image in terms of corners and contours, the third hidden layer can detect entire parts of specific objects, by finding specific collections of contours and corners. Finally, this description of the image in terms of the object parts it contains can be used to recognize the objects present in the image. By Bengio et al. (2015, p. 9).

tecture. For example, a feedforward multilayer perceptron with several hidden layers can be employed. Such an architecture is often referred to as a *deep neural network* (DNN) (Deng and Yu, 2014, p. 217). But also other types of deep architectures exist, like the *deep autoencoder* or the *deep belief network*. The latter two architectures will be discussed in more detail in chapters 6.4 and 6.5.

6.3 Historical Background

Until 2006, mainly shallow architectures of neural networks (with at most one or two layers of hidden units) were used in machine learning applications (Deng and Yu, 2014, p. 205; Bengio, 2009, p. 6). Despite their advantages over single-layer networks like the perceptron, they were still limited in their modeling and representational power. For example, it was shown that some functions cannot be efficiently represented by architectures with insufficient depth (Bengio, 2009, chapter 2 and 3). In this case, the term “efficiently” refers to the number of computational elements (i.e. neurons) in the network. To be more precise:

"Functions that can be compactly represented by a depth k architecture might require an exponential number of computational elements to be represented by a depth $k-1$ architecture" (Bengio, 2009, p. 14). Therefore, shallow networks could not be applied to most complex, high-dimensional real world applications like for instance the processing of speech signals (Bengio and LeCun, 2007; Deng and Yu, 2014, p. 205).
In addition to the limitations of shallow networks, also the organization of the mammal brain has motivated the use of deep neural networks: it has a hierarchical organization in which a given input perception is represented at multiple levels of abstraction (Deng, 1999; Serre et al., 2007). For this reason, researchers tried for decades to develop efficient training algorithms for deep mulitlayer networks (see e.g. Utgoff and Stracuzzi (2002) or Bengio and LeCun (2007)). However, the training of deep multilayer networks turned out to be much more difficult than the training of shallow networks. This led to the widespread believe that training deep neural networks was too difficult. In particular, it was reported several times that learning techniques based on gradient descent (like the backpropagation algorithm) may get stuck in poor solutions when starting from randomly initialized weights (Bengio et al., 2007; Glorot and Bengio, 2010; Erhan et al., 2009; Tesauro, 1992). Only recently it was reported that local minima are not a serious issue when training a neural net by means of such a learning algorithm (Dauphin et al., 2014).

Today's huge success of deep learning started in 2006 when Hinton et al. introduced *deep belief networks* (DBNs) that could successfully learn *layers* of feature detectors in an unsupervised way. A DBN is composed of several *restricted boltzmann machines* (RBM) and has a simple learning principle: each level of the network is trained locally, layer-by-layer, applying an unsupervised learning algorithm. DBN's are discussed in more detail in section 6.4. Shortly after the presentation of deep belief networks, two alternative deep models based on autoencoders were published that exploit the same learning principle (Bengio et al., 2007; Ranzato et al., 2006).

Since the publication of Hinton et al. in 2006, deep architectures have been successfully applied to a variety of problems (e.g. classification tasks, object recognition, robotics or speech and audio processing) (Bengio, 2009, p. 7; Deng and Yu, 2014). In addition to better learning algorithms, their popularity was strongly facilitated by the improvement of chip processing abilities and the increasing amount of data available for training (Deng and Yu, 2014, p. 201; LeCun et al., 2015, p. 439). A popular example for the application of deep architectures is speech recognition (Hinton et al., 2012; Deng et al., 2013a). After initial successes in phone recognition tasks, several speech and machine learning groups worldwide have developed powerful new models and learning algorithms. Furthermore, several of these groups are part of major companies - like for instance Google - and directly deploy obtained results in their own voice products (Dean et al., 2012; Heigold et al., 2013; Liao, 2013; Jaitly et al., 2012). In Google's case, for example, deep learning methods are deployed in their voice search and Android products (Jaitly et al., 2012; LeCun et al., 2015).

6.4 Deep Belief Network

Deep belief networks (DBNs) as introduced by Hinton et al. (2006), are a special type of neural network, namely *probabilistic* ones.

6.4.1 Probabilistic Models

When regarding neural networks as probabilistic models, representation learning can be interpreted as the problem of finding a set of latent random variables that describe a distribution over the given data. The goal of the learning process is then to determine the probability of the latent variables h given the data x, i.e. $p(h|x)$. This probability is also known as the *posterior* probability and typically found by means of a maximum likelihood estimation (Bengio et al., 2013, p. 1804; Hinton and Sejnowski, 1986, p. 293).

Two forms of probabilistic graphical models can be distinguished, namely *directed* (also called Bayesian networks) and *undirected* ones (also called Markov random fields) (Bishop, 2006, p. 360). Restricted Boltzmann machines, the building blocks of a deep belief network, are *undirected* graphical models.

6.4.2 Boltzmann Machine

As mentioned in section 6.3, a DBN consists of several restricted Boltzmann machines (RBMs). A restricted Boltzmann machine is a special type of Boltzmann machine which, in turn, is an undirected graphical model.

A Boltzmann machine (BM) is a network of *stochastic neurons* that reside between two possible states (e.g. $+1$ denoting the "on" state and -1 denoting the "off" state) in a probabilistic manner (Deng and Yu, 2014, p. 217; Haykin, 2009, p. 598). Similar to the general structure of a neural network, the stochastic neurons of the BM are partitioned into *visible* and *hidden* neurons. A central difference are however the *symmetric synaptic connections* between the neurons (Haykin, 2009, p. 598) which are illustrated in figure 6.3. These connections represent some form of *recurrence* in the network since the hidden and visible neurons are also connected *among themselves.*

Similar to the input neurons of a general neural network (cf. section 3.4), the visible neurons of a Boltzmann machine are "clamped" onto specific states during training. These states are determined by the input to the network. The states of the hidden units, however, are not clamped but used to capture explanatory factors that characterize the environmental input vectors (Hinton and Sejnowski, 1986, p. 290; Haykin, 2009, p. 598).

In simple terms, the main goal of learning a Boltzmann machine consists in finding a set of weights "so that when the network is running freely, the patterns of activity that occur over the visible units are the same as they would be if the environment was clamping them" (Hinton and Sejnowski, 1986, p. 292). Or in other words, the goal consists in finding the set of weights that has most likely produced the environmental input vectors. If such a set of weights has

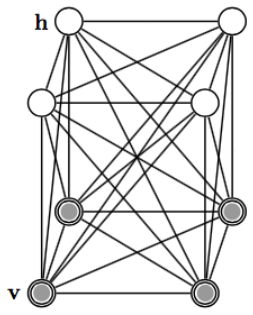

Fig. 6.3: A general Boltzmann machine. The top layer represents a vector of stochastic binary "hidden" features and the bottom layer represents a vector of stochastic binary "visible" variables. Adapted from Salakhutdinov and Hinton (2009).

been found, it is said to constitute a *perfect model* of the environmental structure (Hinton and Sejnowski, 1986, p. 294).

A central disadvantage of a general Boltzmann machine is that it is hard to infer the posterior distribution over all possible configurations of hidden causes. Even getting a sample from the posterior is extremely difficult due to the extensive connections between the neurons. As a direct consequence, training can take very long, especially if the number of hidden neurons is large (Salakhutdinov and Hinton, 2009, p. 449; Haykin, 2009, p. 604; Bengio, 2009, p. 55; Bengio et al., 2013, p. 1806). If, however, the interactions between hidden and visible neurons are restricted, learning becomes easier. For this reason, restricted Boltzmann machines are deployed in deep belief nets.

6.4.3 Restricted Boltzmann Machine

Restricted Boltzmann machines (RBMs) were first introduced in 1986 under the name *Harmonium* (Smolensky, 1986). Specific to the RBM is that the connections in the network are restricted to those between visible and hidden neurons - no hidden-hidden or visible-visible connections exist. Apart from this, and in accordance with a general Boltzmann machine, the connections are symmetrical (Deng and Yu, 2014, p. 242; Bengio et al., 2013, p. 1806; Haykin, 2009, p. 606). The architectural differences between a general and a restricted Boltzmann machine are illustrated in figure 6.4.
By only allowing for connections between the visible and hidden neurons, the states of the hidden neurons in an RBM become *conditionally independent* of each other, given the visible states. Thus, an unbiased sample from the posterior distribution can be drawn and training can be carried out efficiently given an input vector (Salakhutdinov and Hinton, 2009, p. 449; Bengio, 2009, p. 55; Bengio et al., 2013, p. 1807).

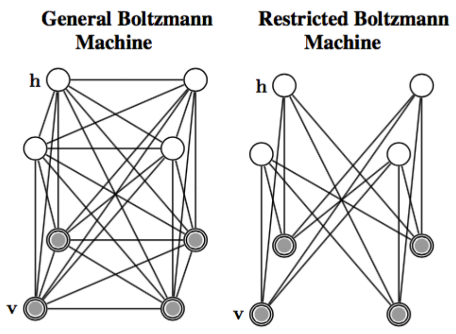

Fig. 6.4: Left: Architecture of a general Boltzmann machine. Right: Architecture of a restricted Boltzmann machine with no hidden-to-hidden and no visible-to-visible connections. Adapted from Salakhutdinov and Hinton (2009).

6.4.4 Deep Belief Network

The main idea behind a deep belief network (DBN) is to learn a complex model by combining a set of simpler models that are learned sequentially (Hinton et al., 2006, p. 1535). The greedy, unsupervised learning algorithm that was proposed by Hinton et al. (2006) utilizes this idea and learns a stack of RBM's, one layer at a time. The resulting network is referred to as a *deep belief network* and depicted in figure 6.5. As can be seen in the figure, the top two layers of a DBN have undirected, symmetrical connections whereas lower layers receive top-down, directed connections from the layer above. Thereby, the top two layers form a restricted Boltzmann machine which is an undirected graphical model, but the lower layers form a directed generative model.
It is important to note, however, that not only RBMs can be stacked in this way. For instance, also autoencoders can be used to form a deep neural architecture, as discussed in section 6.5 (Hinton and Salakhutdinov, 2006; Bengio et al., 2007). Therefore, the algorithm proposed by Hinton et al. can be seen as a more general technique that allows for an unsupervised, layer-by-layer pre-training of simple models in order to form a complex model (Bengio, 2009, p. 6; Deng and Yu, 2014, p. 245).

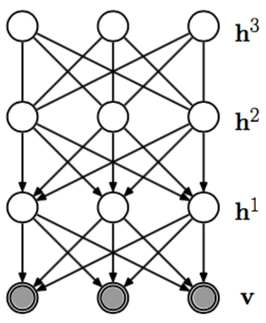

Fig. 6.5: A three-layer deep belief network. Adapted from Salakhutdinov and Hinton (2009).

The unsupervised learning algorithm of a DBN works in the following way (Hinton et al., 2006, p. 1537; Deng and Yu, 2014, p. 245; Bengio, 2013, p. 4). As a first step, one RBM is trained directly on the input data, therefore referred to as the first hidden layer of the DBN. Then, the activation probabilities of this first hidden layer/RBM are used as input data to train a second RBM, which is stacked on top of the first. This procedure is applied repeatedly, bottom-up and layer by layer, until a specified number of hidden layers has been trained. In a last step, the stack of trained RBMs is transformed into a single generative model (the DBN). This is done by replacing the undirected connections of the lower level RBMs by top-down, directed connections (Hinton et al., 2012, p. 87). The complete learning process is illustrated in figure 6.6.
As each hidden layer of the network acts as a feature detector, the learned representations become increasingly more complex with each step (as illustrated in Fig. 6.2). Furthermore, approximate maximum likelihood learning is achieved (Deng and Yu, 2014, p. 246).[1] As a last step, the parameters of all layers can be fine-tuned, for instance with the backpropagation algorithm (Bengio et al., 2007, p. 156; Deng et al., 2010, p. 1692).

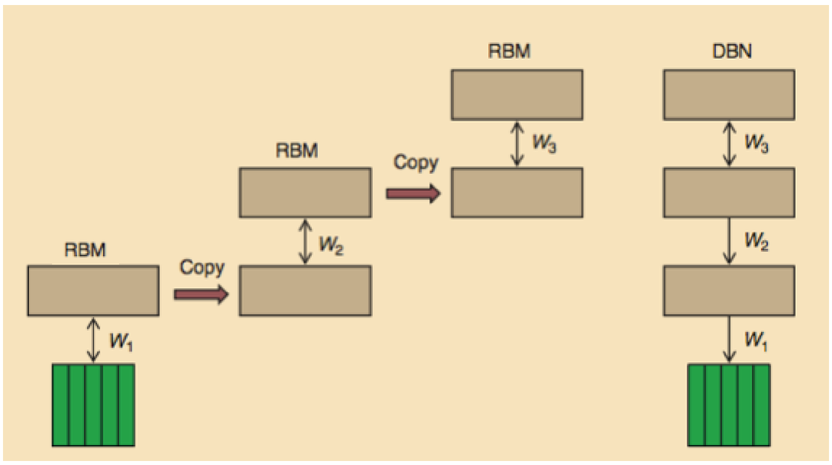

Fig. 6.6: The sequence of operations used to create a DBN with three hidden layers. First, an RBM is trained directly on the input data. Then, the states of the hidden units of the first RBM are used as data for training a second RBM. This is repeated to create as many hidden layers as desired. Last, the stack of RBMs is converted to a single generative model, the DBN. Adapted from Hinton et al. (2012).

More recently, unsupervised training procedures like the above mentioned have been used to successfully *initialize* the weights of more complex deep networks (Bengio et al., 2007; Erhan et al., 2009; Glorot and Bengio, 2010). This allows for an effective training of deep supervised nets, which were found to compute poor solutions when starting from randomly initialized weights (Bengio, 2009).

[1] More precisely, a variational lower bound on the likelihood of the training data under the composite model is improved (Hinton et al., 2006, p. 1530; Deng and Yu, 2014, p. 245).

6.5 Deep Autoencoder

Autoencoders, as introduced in section 5.4, are considered to be deep if they have more than one hidden layer. Although shallow autoencoders can successfully be trained with one of the backpropagation variants (Deng and Yu, 2014, p. 231), the training of autoencoders with multiple hidden layers is difficult (DeMers and Cottrell, 1993; Deng and Yu, 2014, p. 231; Kambhatla and Leen, 1997). As Hinton and Salakhutdinov (2006, p. 505) put it: “With large initial weights, autoencoders typically find poor local minima; with small initial weights, the gradients in the early layers are tiny, making it infeasible to train autoencoders with many hidden layers”. Pre-training the network with the greedy, layerwise training algorithm proposed by Hinton et al. (2006) solves this problem.

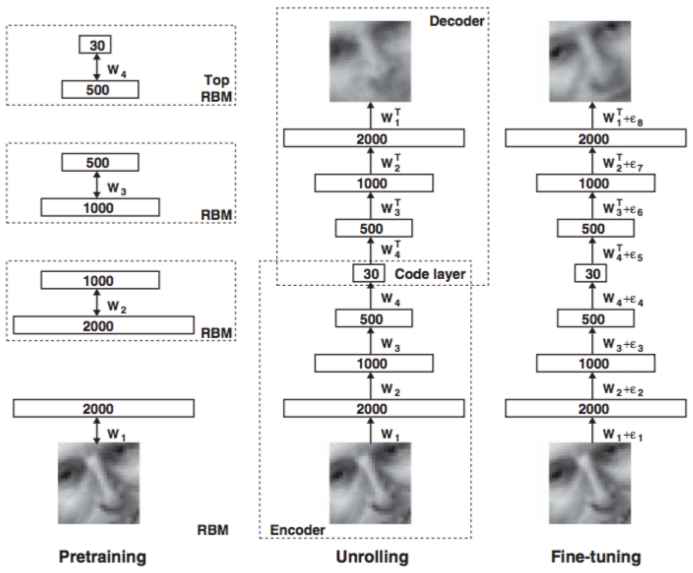

Fig. 6.7: The process of pretraining, unrolling and finetuning of a deep autoencoder, using a black and white image as input. Pretraining consists of learning a stack of restricted Boltzmann machines (RBMs), each having only one layer of feature detectors. The learned feature activations of one RBM are used as the data for training the next RBM in the stack. After the pretraining, the RBMs are unrolled to create a deep autoencoder, which is then fine-tuned using backpropagation of error derivatives. By Hinton and Salakhutdinov (2006).

To successfully initialize the weights of a deep autoencoder, several layers of RBMs are pre-trained as described in section 6.4.4. Then, the resulting stack of RBMs is “unfolded” in order to produce a deep autoencoder (Hinton and Salakhutdinov, 2006, p. 507, Deng et al., 2010, p. 1693). This is done by using the weight-matrices of the RBM-stack in their original order in the lower layers (i.e. the *encoder* part of the network) and in reverse order in higher layers (i.e. the *decoder* part of the network). As a last step, the stochastic activities of the hidden units are replaced by deterministic, real-valued probabilities and all weights are fine-tuned using backpropagation (Hinton and Salakhutdinov, 2006, p. 507, Deng et al., 2010, p. 1693). The whole process of pre-training, unrolling and fine-tuning is exemplified for binary input (in this case a black and white image) in figure 6.7.

7. CONCLUSION

In this thesis, I have introduced the field of representation learning and reviewed the most popular types of feedforward neural networks that are employed in this context. For this, I have started with a general introduction to representation learning. Representation learning is a specific type of machine learning, motivated by the fact that the performance of machine learning algorithms heavily depends on the representation of the data they are given. For this reason, representation learning aims at discovering explanatory factors/features in the data that can be combined in order to construct a useful representation.
In order to learn such representations, various architectures of artificial neural networks can be employed. Following the historical line of introduction, I have first presented *perceptrons*, simple one-layer neural networks that had a high impact on the field of neural networks. But, as they could only solve linearly separable problems, perceptrons were very limited in their representational power. Therefore, researchers long tried to develop an efficient method for training networks with *several* layers of adjustable weights, commonly known as *multilayer perceptrons*. Although such a method was already discovered in 1974, it took until 1986 that the *backpropagation algorithm* became popular.
But also backpropagation turned out to be limited. As only shallow neural architectures (with at most two layers of hidden neurons) could be trained efficiently, researchers were still unable to apply neural networks to most complex real world problems. This gave rise to the field of *deep learning*.

Deep learning is a relatively new field of representation (and therefore machine) learning that gained wide attention after the presentation of a new learning algorithm in 2012 (Hinton et al., 2012). The key idea behind deep learning is to learn multiple levels of representation by combining simpler concepts into more complex ones. At the moment, deep learning is an extremely popular research area with a huge media coverage and impressive successes. For example, the *New York Times* covered the subject twice already in 2012, with front-page articles (Markoff, 2012). Since then, further articles on more recent advances in deep learning have been published (see e.g. Markoff, 2015). This raises the question of why there is such a big need for deep learning and deep architectures in the first place.

In order for an artificial system to behave intelligently, it must be able to learn not only low-, but also high-level abstractions. This becomes evident when considering problems like the chair challenge introduced in section 2.1. As it is often difficult for humans to hand-code the explanatory factors of such high-level abstractions, a lot of work has been put into the development of algorithms that automatically learn powerful features. Although it was shown that shallow neural networks with only two layers of hidden neurons can represent any function (cf. section 5.3), the corresponding theorems state nothing about the *efficiency* of the representation. This is were the advantages of deep learning come into being.

Theoretical results reviewed and discussed by Bengio (2009) suggest that in order to learn the kind of complicated functions that can represent high-level abstractions (e.g. in vision or language), one may need deep architectures. This conclusion is mainly motivated by the inability of shallow architectures to efficiently represent complex high-dimensional functions. Mathematical as well as empirical evidence discussed by Bengio and LeCun (2007) demonstrates that shallow architectures are a lot less expressive than deep ones. The power of deep architectures is furthermore underlined by investigations into computational complexity of shallow circuits. For example, Hastad (1986) showed that functions that can efficiently be represented by an architecture of depth k, require exponential size when the depth is restricted to $k-1$.

The notion that deep architectures are needed in order to represent high-level abstractions is furthermore supported by the fact that deep learning methods significantly outperform comparable shallow architectures in several AI-related tasks as for instance object recognition or natural language processing. Also, they often match or beat the state of the art. For example, regarding object recognition, an essential breakthrough was achieved with the application of deep convolutional networks on the *ImageNet* dataset by Krizhevsky et al. in 2012, bringing down the state-of-the-art error rate from 26.1% to 15.3%.

Although feedforward neural networks allow for many interesting applications, one should not forget about *recurrent* systems. Recurrent neural networks, as introduced shortly in the beginning of this thesis, represent a promising research area in machine learning that offers many interesting possibilities. However, due to the breadth of the field, recurrent networks could not be discussed further. Nevertheless, detailed information is widely available. See e.g. Haykin (2009).

BIBLIOGRAPHY

Aamodt, A. (1993). A case-based answer to some problems of knowledge-based systems. In Sandewall, E. and Jansson, C., editors, *Scandinavian Conference on Artificial Intelligence*, pages 168–182. IOS Press.

Abbott, L. F., Rolls, E. T., and Tovee, M. J. (1996). Representational capacity of face coding in monkeys. *Cerebral Cortex*, 6:498–505.

Ancona, N., Maglietta, R., and Stella, E. (2004). Sparse vs. dense data representations in kernel methods.

Anthony, M. (2001). *Discrete Mathematics of Neural Networks: Selected Topics*. Siam.

Arbib, M. A. (1987). *Brains, Machines and Mathematics*. Springer, 2nd edition.

Barlow, H. B. (1972). Single units and sensation: a neuron doctrine for perceptual psychology? *Perception*, 1:371–394.

Becker, S. (1991). Unsupervised learning procedures for neural networks. *International Journal of Neural Systems*, 2:17–33.

Bengio, S. and Heigold, G. (2014). Word embeddings for speech recognition. In *Proceedings of the 15th Conference of the International Speech Communication Association*.

Bengio, Y. (2009). Learning deep architectures for AI. *Foundations and trends in Machine Learning*, 2:1–127.

Bengio, Y. (2013). Deep learning of representations: Looking forward. In *Statistical language and speech processing*, pages 1–37. Springer.

Bengio, Y., Courville, A., and Vincent, P. (2013). Representation learning: A review and new perspectives. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 35(8):1798–1828.

Bengio, Y., Ducharme, R., Vincent, P., and Janvin, C. (2003). A neural probabilistic language model. *The Journal of Machine Learning Research*, 3:1137–1155.

Bengio, Y., Goodfellow, I. J., and Courville, A. (2015). Deep learning. Book in preparation for MIT Press.

Bengio, Y., Lamblin, P., Popovici, D., and Larochelle, H. (2007). Greedy layer-wise training of deep networks. *Advances in neural information processing systems*, 19:153–160.

Bengio, Y. and LeCun, Y. (2007). Scaling learning algorithms towards AI. In Bottou, L., Chapelle, O., DeCoste, D., and Weston, J., editors, *Large-Scale Kernel Machines*. MIT Press.

Bishop, C. M. (1995). *Neural Networks for Pattern Recognition*. Oxford University Press.

Bishop, C. M. (2006). *Pattern Recognition and Machine Learning*. Springer.

Block, H. D. (1962). The perceptron: A model for brain functioning. *Reviews of Modern Physics*, 34(1):123–135.

Bosse, E., Roy, J., and Wark, S. (2007). *Concepts, Models, and Tools for Information Fusion*. Artech House Publishers.

Bourlard, H. and Kamp, Y. (1988). Auto-association by multilayer perceptrons and singular value decomposition. *Biological Cybernetics*, 59:291–294.

Buelthoff, I. and Buelthoff, H. (2003). Image-based recognition of biological motion, scenes, and objects. In *Perception of faces, objects, and scenes*, pages 146–176. Oxford University Press.

Bussmann, H. (1996). *Routledge Dictionary of Language and Linguistics*. Routledge.

Churchland, P. S. and Sejnowski, T. J. (1992). *The Computational Brain*. MIT Press.

Ciresan, D., Meier, U., and Schmidhuber, J. (2012). Multi-column deep neural networks for image classification. In *Computer Vision and Pattern Recognition (CVPR)*, pages 3642–3649.

Collobert, R., Weston, J., Bottou, L., Karlen, M., Kavukcuoglu, K., and Kuksa, P. (2011). Natural language processing (almost) from scratch. *The Journal of Machine Learning Research*, 12:2493–2537.

Culler, J. (1981). *The Pursuit of Signs - Semiotics, literature, deconstruction*. Routledge.

Cybenko, G. (1989). Approximation by superposition of a sigmoidal function. *Mathematics of Control, Signals and Systems*, 2:303–314.

Dahl, G., Mohamed, A.-r., Hinton, G. E., et al. (2010). Phone recognition with the mean-covariance restricted boltzmann machine. In *Advances in Neural Information Processing Systems*, pages 469–477.

Dahl, G. E., Yu, D., Deng, L., and Acero, A. (2012). Context-dependent pre-trained deep neural networks for large-vocabulary speech recognition. *IEEE Transactions on Audio, Speech, and Language Processing*, 20(1):30–42.

Dauphin, Y. N., Pascanu, R., Gulcehre, C., Cho, K., Ganguli, S., and Bengio, Y. (2014). Identifying and attacking the saddle point problem in high-dimensional non-convex optimization. In *Advances in Neural Information Processing Systems*, volume 27, pages 2933–2941.

de Saussure, F. (1959). *Course in General Linguistics.* William Collins, Sons.

Dean, J., Corrado, G. S., Monga, R., Chen, K., Devin, M., Mao, M. Z., Senior, A., Tucker, P., Yang, K., Le, Q. V., Ng, A. Y., and Ranzato, M. (2012). Large scale distributed deep networks. In *Advances in Neural Information Processing Systems*, pages 1223–1231.

DeMers, D. and Cottrell, G. W. (1993). Non-linear dimensionality reduction. In Hanson, S., Cowan, J., and Giles, C., editors, *Advances in Neural Information Processing Systems 5*, pages 580–587. Morgan-Kaufmann.

Deng, L. (1999). Computational models for speech production. In *Computational models of speech pattern processing*, pages 199–213. Springer.

Deng, L., Hinton, G. E., and Kingsbury, B. (2013a). New types of deep neural network learning for speech recognition and related applications: An overview. In *IEEE International Conference on Acoustics, Speech, and Signal Processing (ICASSP)*, pages 8599–8603.

Deng, L., Li, J., Huang, J.-T., Yao, K., Yu, D., Seide, F., Seltzer, M. L., Zweig, G., He, X., Williams, J., Gong, Y., and Acero, A. (2013b). Recent advances in deep learning for speech research at microsoft. In *Proceedings of International Conference on Acoustics Speech and Signal Processing.*

Deng, L., Seltzer, M. L., Yu, D., Acero, A., Mohamed, A., and Hinton, G. E. (2010). Binary coding of speech spectrograms using a deep auto-encoder. In *Interspeech 2010*, pages 1692–1695. International Speech Communication Association.

Deng, L. and Yu, D. (2014). Deep learning: Methods and applications. *Foundations and Trends in Signal Processing*, 7(3-4):197387.

Duda, R. O. and Shortliffe, E. H. (1983). Expert systems research. *Science*, 220(4594):261–268.

Erhan, D., Manzagol, P.-A., Bengio, Y., Bengio, S., and Vincent, P. (2009). The difficulty of training deep architectures and the effect of unsupervised pre-training. In *International Conference on Artificial Intelligence and Statistics*, volume 5, pages 153–160.

Fabre-Thorpe, M. (2003). Visual categorization: accessing abstraction in non-human primates. *Philosophical Transactions of the Royal Society B: Biological Sciences*, 358:1215–1223.

Fausett, L. (1994). *Fundamentals of neural networks: architectures, algorithms and applications.* Pearson Prentice Hall.

Freeman, W. J. (1975). *Mass Action in the Nervous System.* Academic Press.

Georgopoulos, A. P., Schwartz, A. B., and Kettner, R. E. (1986). Neuronal population coding of movement direction. *Science*, 233(4771):1416–1419.

Glorot, X. and Bengio, Y. (2010). Understanding the difficulty of training deep feedforward neural networks. In *International Conference on Artificial Intelligence and Statistics*, volume 9, pages 249–256.

Gross, C. G. (2002). Genealogy of the grandmother cell. *The Neuroscientist*, 8(1):84–90.

Hagan, M. T., Demuth, H. B., Beale, M. H., and De Jess, O. (2014). *Neural Network Design.* Martin Hagan, USA, 2nd edition.

Hastad, J. (1986). Almost optimal lower bounds for small depth circuits. In *Proceedings of the 18th Annual ACM Symposium on Theory of Computing*, pages 6–20. ACM Press.

Haykin, S. (2009). *Neural networks and learning machines.* Pearson Prentice Hall, 3rd edition.

Heigold, G., Vanhoucke, V., Senior, A., Nguyen, P., Ranzato, M., Devin, M., and Dean, J. (2013). Multilingual acoustic models using distributed deep neural networks. In *Proceedings of International Conference on Acoustics Speech and Signal Processing (ICASSP)*, pages 8619–8623.

Hinton, G., Deng, L., Yu, D., Dahl, G. E., Mohamed, A.-r., Jaitly, N., Senior, A., Vanhoucke, V., Nguyen, P., Sainath, T. N., and Kingsbury, B. (2012). Deep neural networks for acoustic modeling in speech recognition. *Signal Processing Magazine, IEEE*, 29(6):82–97.

Hinton, G. E. (1986). Learning distributed representations of concepts. In *Proceedings of the Eighth Annual Conference of the Cognitive Science Society*, volume 1, pages 1–12. Amherst, MA.

Hinton, G. E. (1992). How neural networks learn from experience. *Scientific American*, 267(3):145–151.

Hinton, G. E. (2012). Neural networks for machine learning, Lecture 1. Online Lecture: `https://www.coursera.org/course/neuralnets`. Accessed: 2015-05-09.

Hinton, G. E., Osindero, S., and Teh, Y.-W. (2006). A fast learning algorithm for deep belief nets. *Neural Computation*, 18(7):1527–1554.

Hinton, G. E. and Salakhutdinov, R. R. (2006). Reducing the dimensionality of data with neural networks. *Science*, 313:504–507.

Hinton, G. E. and Sejnowski, T. J. (1986). Learning and relearning in boltzmann machines. In Rumelhart, D. E., McClelland, J. L., and the PDP Research Group, editors, *Parallel Distributed Processing: Explorations in the Microstructure of Cognition, Vol. 1*, pages 282–317. MIT Press.

Hornik, K., Stinchcombe, M., and White, H. (1989). Multilayer feedforward networks are universal approximators. *Neural Networks*, 2:359–366.

Hsu, F.-H. (2002). *Behind Deep Blue: Building the computer that defeated the world chess champion.* Princeton University Press.

Huang, F. J. and LeCun, Y. (2006). Large-scale learning with SVM and convolutional for generic object categorization. In *Computer Vision and Pattern Recognition Conference (CVPR'06).* IEEE.

Jaitly, N., Nguyen, P., Senior, A., and Vanhoucke, V. (2012). Application of pretrained deep neural networks to large vocabulary speech recognition. In *Proceedings of Interspeech 2012*.

Kambhatla, N. and Leen, T. K. (1997). Dimension reduction by local principal component analysis. *Neural Computation*, 9(7):1493–1516.

Karlik, B. and Olgac, A. V. (2011). Performance analysis of various activation functions in generalized MLP architectures of neural networks. *International Journal of Artificial Intelligence And Expert Systems (IJAE)*, 1(4):111–122.

Krizhevsky, A., Sutskever, I., and Hinton, G. E. (2012). Imagenet classification with deep convolutional neural networks. In Pereira, F., Burges, C., Bottou, L., and Weinberger, K., editors, *Advances in Neural Information Processing Systems 25*, pages 1097–1105. Curran Associates, Inc.

LeCun, Y. (1986). Learning processes in an asymmetric threshold network. In Bienenstock, E., Fogelman-Soulié, F., and Weisbuch, G., editors, *Disordered systems and biological organization*, pages 233–240. Springer-Verlag.

LeCun, Y., Bengio, Y., and Hinton, G. E. (2015). Deep learning. *Nature*, 521:436–444.

Lee, H., Pham, P., Largman, Y., and Ng, A. Y. (2009). Unsupervised feature learning for audio classification using convolutional deep belief networks. In *Advances in Neural Information Processing Systems (NIPS'09)*.

Lenat, D. B. (1995). Cyc: A large-scale investment in knowledge infrastructure. *Communications of the ACM*, 38(11):33–38.

Liao, H. (2013). Speaker adaptation of context dependent deep neural networks. In *Proceedings of International Conference on Acoustics Speech and Signal Processing (ICASSP)*.

Lippmann, R. P. (1987). An introduction to computing with neural nets. *IEEE ASSP Magazine*, 3(4):4–22.

Logothetis, N. K. and Sheinberg, D. L. (1996). Visual object recognition. *Annual Review of Neuroscience*, 19:577–621.

Markoff, J. (2012, November 24). Scientists see promise in deep-learning programs. *New York Times*, page A1.

Markoff, J. (2015, May 21). New approach trains robots to match human dexterity and speed. *New York Times*, page A15.

McClelland, J. L., McNaughton, B. L., and O'Reilly, R. C. (1995). Why there are complementary learning systems in the hippocampus and neocortex: Insights from the successes and failures of connectionist models of learning and memory. *Psychological Review*, 102(3):419–457.

McCulloch, W. S. and Pitts, W. (1943). A logical calculus of the ideas immanent in nervous activity. *Bulletin of Mathematical Biophysics*, 5:115–133.

Minsky, M. (1988). *Society of mind*. Simon and Schuster.

Minsky, M. L. and Papert, S. A. (1969). *Perceptrons: An Introduction to Computational Geometry.* MIT Press.

Minsky, M. L. and Papert, S. A. (1988). *Perceptrons - Expanded Edition: An Introduction to Computational Geometry.* MIT press.

Mitchell, T. M. (1997). *Machine Learning.* McGraw-Hill.

Mohamed, A., Dahl, G. E., and Hinton, G. E. (2012). Acoustic modeling using deep belief networks. *IEEE Transactions on Audio, Speech, and Language Processing*, 20(1):14–22.

Murphy, K. P. (2012). *Machine Learning: A Probabilistic Perspective.* MIT Press, 1st edition.

Ng, A. Machine learning, Week 1 - Introduction. Online Lecture: `https://www.coursera.org/learn/machine-learning`. Accessed: 2015-05-09.

Ng, J. Y.-H., Hausknecht, M., Vijayanarasimhan, S., Vinyals, O., Monga, R., and Toderici, G. (2015). Beyond short snippets: Deep networks for video classification. In *Computer Vision and Pattern Recognition.*

Olshausen, B. A. and Field, D. J. (2004). Sparse coding of sensory inputs. *Current Opinion in Neurobiology*, 14:481–487.

O'Reilly, R. C. and Norman, K. A. (2002). Hippocampal and neocortical contributions to memory: Advances in the complementary learning systems framework. *Trends in Cognitive Sciences*, 6(12):505–510.

Perez-Orive, J., Mazor, O., Turner, G. C., Cassenaer, S., Wilson, R. I., and Laurent, G. (2002). Oscillations and sparsening of odor representations in the mushroom body. *Science*, 297:359–365.

Quiroga, R. Q. (2012). Concept cells: the building blocks of declarative memory functions. *Nature Reviews Neuroscience*, 13:587–597.

Quiroga, R. Q., Fried, I., and Koch, C. (2013). Brain cells for grandmother. *Scientific American*, 308(2):30–35.

Quiroga, R. Q., Kreiman, G., Koch, C., and Fried, I. (2008). Sparse but not 'grandmother-cell' coding in the medial temporal lobe. *Trends in Cognitive Sciences*, 12(3):87–91.

Quiroga, R. Q., Reddy, L., Kreiman, G., Koch, C., and Fried, I. (2005). Invariant visual representation by single neurons in the human brain. *Nature*, 435(7045):1102–1107.

Ranzato, M., Poultney, C., Chopra, S., and LeCun, Y. (2006). Efficient learning of sparse representations with an energy-based model. In *Proceedings of Neural Information Processing Systems.*

Rolls, E., Treves, A., Robertson, R., Georges-Francois, P., and Panzeri, S. (1998). Information about spatial view in an ensemble of primate hippocampal cells. *Journal of Neurophysiology*, 79:17971813.

Rolls, E. T. and Treves, A. (1998). *Neural Networks and Brain Function*. Oxford University Press.

Rolls, E. T., Treves, A., and Tovee, M. J. (1997). The representational capacity of the distributed encoding of information provided by populations of neurons in primate temporal visual cortex. *Experimental Brain Research*, 114:149–162.

Rosenblatt, F. (1958). The perceptron: A probabilistic model for information storage in the brain. *Psychological Review*, 65(6):386–408.

Rosenblatt, F. (1962). *Principles of Neurodynamics: Perceptrons and the Theory of Brain Mechanisms*. Spartan Books.

Rumelhart, D. E., Hinton, G. E., and Williams, R. J. (1986). Learning representations by back-propagating errors. *Nature*, 323:533–536.

Salakhutdinov, R. and Hinton, G. E. (2009). Deep boltzmann machines. In *International Conference on Artificial Intelligence and Statistics*, pages 448–455.

Satkin, S. and Hebert, M. (2013). 3DNN: Viewpoint invariant 3d geometry matching for scene understanding. In *Proceedings of the International Conference on Computer Vision (ICCV)*.

Seide, F., Li, G., and Yu, D. (2011). Conversational speech transcription using context-dependent deep neural networks. In *Interspeech*, pages 437–440.

Serre, T., Kreiman, G., Kouh, M., Cadieu, C., Knoblich, U., and Poggio, T. (2007). A quantitative theory of immediate visual recognition. *Progress in Brain Research*, 165:33–56.

Shepherd, G. M. (2004). *The Synaptic Organization of the Brain*. Oxford University Press, 5th edition.

Sibi, P., Jones, S. A., and Siddarth, P. (2013). Analysis of different activation functions using back propagation neural networks. *Journal of Theoretical and Applied Information Technology*, 47(3):1264–1268.

Sirois, S. (2004). Autoassociator networks: insights into infant cognition. *Developmental Science*, 7:133–140.

Smolensky, P. (1986). Information processing in dynamical systems: Foundations of harmony theory. In Rumelhart, D. E., McClelland, J. L., and PDP Research Group, C., editors, *Parallel Distributed Processing: Explorations in the Microstructure of Cognition, Vol. 1*, chapter 6, pages 194–281. MIT Press.

Tesauro, G. (1992). Practical issues in temporal difference learning. *Machine Learning*, 8:257–277.

Theunissen, F. E. (2003). From synchrony to sparseness. *TRENDS in Neurosciences*, 26(2):61–64.

Utgoff, P. E. and Stracuzzi, D. J. (2002). Many-layered learning. *Neural Computation*, 14(10):2497–2529.

Weston, J., Bengio, S., and Usunier, N. (2010). Large scale image annotation: learning to rank with joint word-image embeddings. *Machine learning*, 81(1):21–35.

Winograd, T. (1991). Thinking machines: Can there be? Are we? In Sheehan, J. and Sosna, M., editors, *The Boundaries of Humanity: Humans, Animals, Machines*, pages 198–223. Berkeley: University of California Press.

Yu, D., Deng, L., and Dahl, G. E. (2010). Roles of pre-training and fine-tuning in context-dependent dbn-hmms for real-world speech recognition. In *NIPS 2010 Workshop on Deep Learning and Unsupervised Feature Learning*.