


Technische Universität München
Department of Informatics

Master's Thesis in Informatics

Deep Learning in Smart Spaces

Markus Loipfinger

Technische Universität München

Department of Informatics

Master's Thesis in Informatics

Deep Learning in Smart Spaces

Deep Learning in intelligenten Umgebungen

Author	Markus Loipfinger
Supervisor	Prof. Dr.-Ing. Georg Carle
Advisor	Dr. Marc-Oliver Pahl, Stefan Liebald
Date	September 15, 2017

Informatik VIII
Chair for Network Architectures and Services

I confirm that this thesis is my own work and I have documented all sources and material used.

Garching b. München, September 15, 2017

Signature

Abstract

We aim to provide machine learning as a service with the result that users with little or even no pre-knowledge in the area of machine learning are able to create, train and deploy their own neural network. This is achieved by modularizing three machine learning algorithms into suitable building blocks.

Both the training times and the running times of our neural network services are comparable to the training times and running times of a regular implementation of the respective neural networks. The advantage of our approach, however, is that users do not have to implement the whole machine learning algorithm from scratch. Hence, our service approach saves time and does not imply expert knowledge in machine learning and the respective machine learning library.

Zusammenfassung

Wir wollen maschinelles Lernen als Service zur Verfügung stellen. Das ermöglicht unerfahrenen Benutzern die Erstellung, das Trainieren und die Anwendung eines neuronalen Netzes. Folglich ist weder Wissen in dem Bereich des maschinellen Lernens notwendig, noch Erfahrung im Umgang mit der jeweiligen Programmbibliothek vorausgesetzt. Dies wird dadurch erreicht, dass wir drei verschiedene Algorithmen des maschinellen Lernens modularisieren.
Die Trainingszeiten und Laufzeiten unserer Services sind vergleichbar mit den Trainings- und Laufzeiten der regulären Implementierung des entsprechenden neuronalen Netzes. Ein Vorteil unserer Vorgehensweise ist jedoch, dass sich der Nutzer Zeit spart, in dem Sinne, dass er das neuronale Netz nicht von Grund auf implementieren muss. Des Weiteren wird kein Wissen im Bereich des maschinellen Lernens und in der entsprechenden Programmbibliothek vorausgesetzt.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The population of the world is growing older. About 8.5% of the people worldwide were 65 years or older in 2015. Scientists estimate that this percentage rises up to 12.0% in 2030 and 16.7% in 2050 [17]. On the one hand, getting older involves that more elderly people need help in their Activities of Daily Living (ADL). Hence, about 9% of adults of age 65 and older and already 50% of people of age 85 and older are dependent on assistance with *ADL* [18]. On the other hand, elderly people often want to be independent of others assistance. One key to solve this problem are *smart spaces*. With the help of them elderly people can be supported in their ADL in a certain range without needing the assistance of another person.
Smart spaces are built up by using *smart devices*. A smart device is an embedded system. It can be remotely controlled using a communication entity (e.g. an app) and it is able to capture its environmental state by using sensors and act according to its purpose by using actuators. A smart space controls all devices and can extract all information from them. To be able to support people in their ADL another important factor needs to be considered. Smart spaces need learning algorithms to control the environment. With the help of a learning algorithm a smart space is able to learn the behaviour of people acting in it. It can, for instance, predict their behaviour and adapt to their personal preferences. This is done by employing machine learning and deep learning, respectively. The algorithm is capable of optimizing itself during a training process. Hence, the smart space can improve itself autonomously. This yields an increasing need for machine learning services which can be created, configured and trained easily to one's preferences.
However, the creation and implementation of such learning algorithms, e.g. neural networks, requires detailed knowledge in the area of machine learning and the corresponding machine learning library. Additionally, if no expert knowledge is existing, the implementation and training of such algorithms is time consuming. Due to that reason an easy-to-use machine learning functionality with a high degree of parametrization is necessary. Moreover, the machine learning service has to be provided in a way that

even users with little or no pre-knowledge in the area of machine learning are able to create and train a neural network.
Besides usability, the focus of the machine learning services implemented in this thesis is on reusability. Having the configuration of the neural network seperated from the machine learning algorithm enables users to test different network configurations fast and in an easy way. Additionally, this ensures portability as the file containing the configuration, i.e. the current state, is only needed to restore a trained model.
Against this background, this thesis designs and implements three machine learning services, each one equipped with a different neural network architecture. Moreover, it is ensured that these services provide easy-to-use machine learning algorithms in terms of both usablility and reusability.

1.1 Goal of the thesis

We aim at building a machine learning service which enables users with little or even no pre-knowledge to build and train a neural network. Therefore, we provide an easy-to-use machine learning functionality. This is why we modularize three different machine learning algorithms and provide them as services. The user can choose between a *Feedforward Neural Network (FFNN)*, a *Deep Belief Network (DBN)* and a *Recurrent Neural Network (RNN)*. When calling one of the three services the user is provided with a configuration file which contains all hyperparameters and parameters of the respective neural network. This file can be changed according to one's needs and is used to create and train the neural network afterwards. When trained, new outputs can be computed by restoring the saved neural network by means of the configuration file.

1.2 Outline

The thesis is structured as follows. In Chapter 2 we explain the principles of machine learning and more specifically deep learning. We further introduce different kinds of neural networks and their deep counterpart. Additionally, the *Distributed Smart Space Orchestration System (DS2OS)* with its *Virtual State Layer (VSL)* is described. The application of machine learning in smart spaces is illustrated afterwards. The chapter is concluded by showing various machine learning libraries. Chapter 3 gives an overview over different approaches of neural networks in smart environments and in classification tasks. It concludes with a comparison table containing the most important things to keep in mind. Essential design ideas are explained in Chapter 4. In Chapter 5 details about the implementation of our machine learning services are given. Moreover, it contains an example at the end where we explain how to work with the implemented machine learning services by training them on the MNIST data set of handwritten digits.

Chapter 6 starts with a quantitative evaluation using different data sets. A performance analysis and a detailed qualitative evaluation are conducted afterwards. Finally, the thesis is concluded in Chapter 7 where we summarize our finding.

1.3 Methodology

As mentioned in Section 1.1 we aim at building an easy-to-use machine learning functionality. Our approach is the following. We start with introducing several machine learning and deep learning architectures, respectively. Additionally, we describe various techniques to improve training. Based on the insights obtained from Chapter 3 we identify three machine learning architectures, a FFNN, a DBN and a RNN as frequently used in smart spaces. By applying the knowledge gained in Chapter 2 we list all possible parameters of the respective machine learning algorithms in Chapter 4. We cluster them into common parameters and parameters which are unique for each algorithm. That is why each one of the three algorithms can be changed by only modifying its parameters. Drawing on this we implement the three machine learning services in Chapter 5. We thereby focus on a parameterized implementation based on a configuration file which contains all parameters of the respective machine learning architecture. An evaluation is conducted in Chapter 6. Considering the evaluation we determine our approach as comparable to regular implementations of the respective neural networks in terms of performance, i.e. training time and running time. The accuracy reached by the neural networks is identical, as the same originial machine learning library underlies.

Chapter 2

Analysis

This chapter describes the principles of deep learning in the beginning. We explain different learning mechanisms, three machine learning classifiers and the components of an *Artificial Neural Network (ANN)*. Additionally, different neural network architectures are introduced including their deep counterpart. We reveal four machine learning libraries afterwards. Then, we introduce the *Distributed Smart Space Orchestration System (DS2OS)*. This chapter concludes by showing different machine and deep learning approaches in smart spaces, respectively.

2.1 Deep Learning

Deep learning is an emerging field in the area of *machine learning* [19]. It denotes models which are built-up by multiple processing layers between the input and output layer. By exploiting the depth of such models which are called *neural networks*, deep learning algorithms are able to break down a complex input into simpler representations [6]. Hence, deep learning turned out to work well for high-dimensional input data [19].
One of the best known problem solved by neural networks is image processing. Figure 2.1 shows how a deep learning algorithm learns to detect faces. In this case the model gets grey-scaled values of pixels as input which define the corresponding image. This is why *layer 1* is also called input layer or visible layer. In the first hidden layer (*layer 2*) the model maps a set of pixels to edges and simple shapes like corners and contours, for instance. It is called hidden layer as, in contrast to the input layer, the values computed in the hidden layers are not observable in the data [6]. The second hidden layer (*layer 3*) takes the output of the previous layer and detects objects, e.g. a nose, a mouth or an eye. Finally, using these objects a deep learning algorithm puts them together to learn how to detect human faces (*layer 4*).

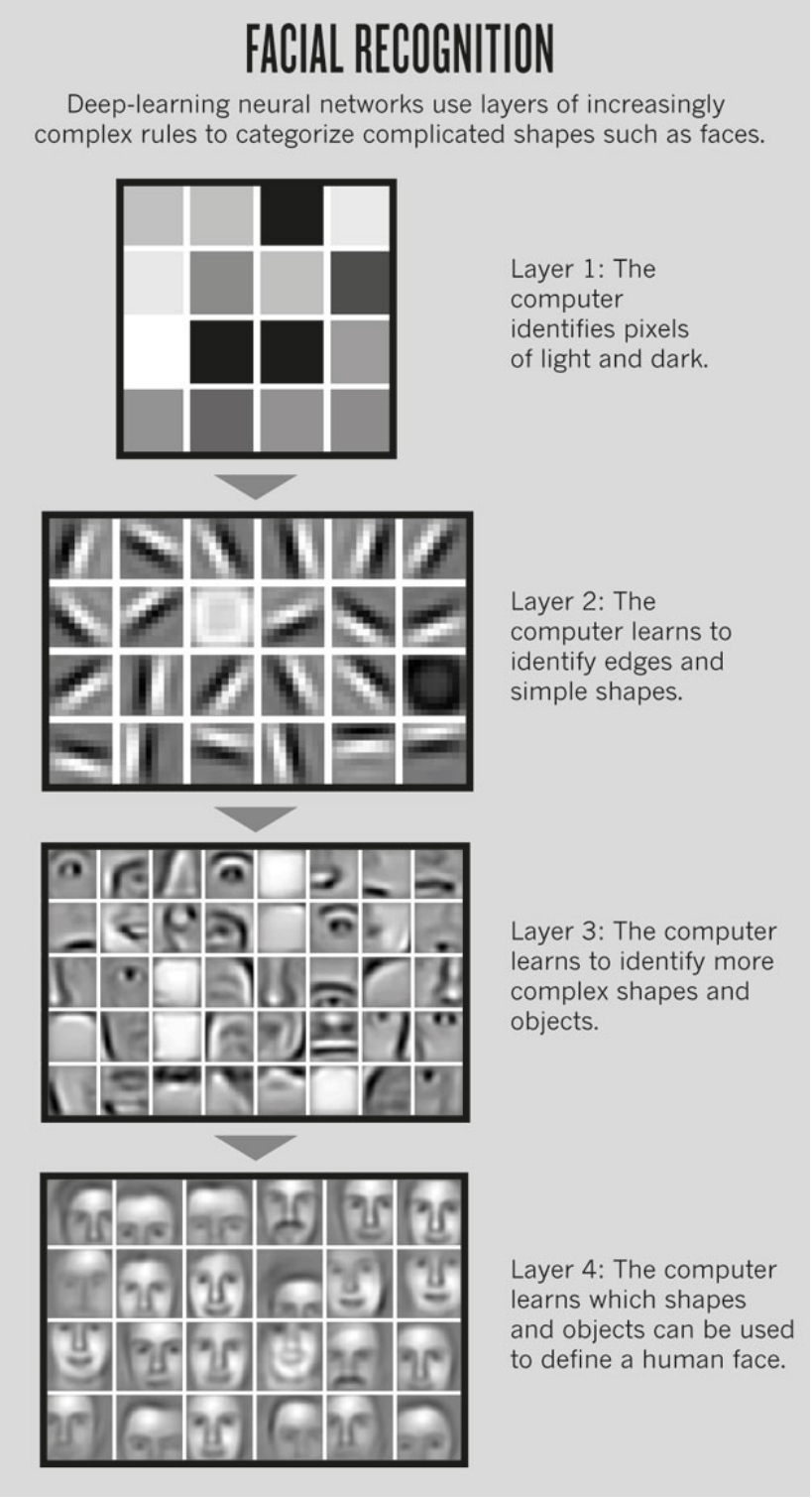


Figure 2.1: An example for learning complex features out of simpler representations [1].

2.1.1 Background

Deep learning algorithms are based on neural networks. They consist of one input layer, several hidden layers and one output layer. Before discussing different kinds of such networks, we start this section by illustrating various artificial neurons. They are the foundation for constructing neural networks. In addition, softmax as a special kind of output unit is explained. Finally, different learning methods which are used in both machine learning and deep learning are introduced. The learning method is not only depending on the problem which we try to solve, but also on the available training data.

2.1.1.1 Artifcial Neurons

In general, neural networks are built by connecting artificial neurons. The term *neuron* is based on the biological neuron [6]. Each artificial neuron has its own activation function and, as a result, fires in another way. In this section four different neurons get introduced, beginning with the first announced artificial neuron, the so-called *perceptron* [20]. Although using this neuron is not common anymore, it helps us understanding the functional principle of an artificial neuron. A neuron (see Figure 2.2) takes several inputs x_i and produces one output y using its activation function a. In addition, each input x_i is weighted by a factor w_i. Thus, the output is given by

$$y = a(z) \tag{2.1}$$

with z being the weighted sum of inputs added to a bias b

$$z = \sum_i x_i w_i + b. \tag{2.2}$$

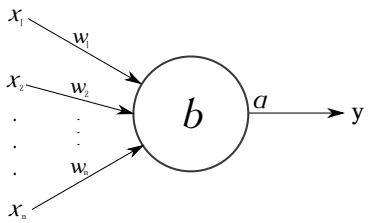


Figure 2.2: An artificial neuron with inputs x_i and output y. Each input value is weighted by a weight w_i. The bias of the neuron is added to the weighted sum of inputs. The output is computed by applying an activation function a.

Perceptron—This neuron takes binary values as input and produces a binary output using a threshold value.

$$y = \begin{cases} 0, if \sum_j x_j w_j \leq threshold \\ 1, if \sum_j x_j w_j > threshold \end{cases}$$

where $\sum_j x_j w_j$ is the weighted sum of inputs to the neuron. By changing $\sum_j x_j w_j$ to a dot-product $x \cdot w$, as x and w are vectors and moving the threshold value to the left side of the inequality, we get

$$y = \begin{cases} 0, if x \cdot w + b \leq 0 \\ 1, if x \cdot w + b > 0 \end{cases}$$

where the threshold value is replaced by a bias b [5]. The perceptron is limited in its functionality, since it only takes binary values as input and produces a binary value as output. As a result, a small change in the input can yield a total change in the output, i.e. from 0 to 1.

Sigmoid Neuron—For learning algorithms it is important that small changes in any weight or bias result in small changes in the output. This is not possible using perceptrons since small changes in the input can lead to a complete change in the output. Sigmoid neurons overcome this problem. The activation function looks as following

$$y = a(z) = \sigma(z) = \frac{1}{1 + e^{-z}} \tag{2.3}$$

where z is $x \cdot w + b$. The graph of the sigmoid function is shown in Figure 2.3. It ranges from 0 to 1, i.e. for large negative z the output becomes approximately 0

$$\lim_{z \to -\infty} \frac{1}{1 + e^{-z}} = 0 \tag{2.4}$$

for large positive z the output is approximately 1

$$\lim_{z \to \infty} \frac{1}{1 + e^{-z}} = 1. \tag{2.5}$$

Sigmoid neurons are the most common ones, but there are other types of neurons emerging in the area of deep learning and neural networks [5].

Tanh Neuron—Tanh neurons use the hyperbolic tangent function as activation function (see Figure 2.4). The output of it is given by

$$y = a(z) = \tanh(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}} \tag{2.6}$$

where z is $x \cdot w + b$. Figure 2.4 shows the graph of the tanh function. The difference compared to the sigmoid function (see Figure 2.3) can be observed. The output of the

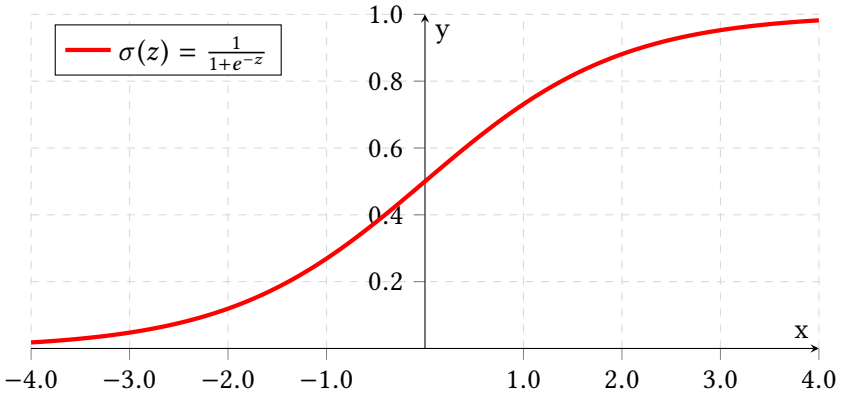


Figure 2.3: Sigmoid function

latter one ranges from 0 to 1, whereas the output of the tanh function ranges from -1 to 1. Hence, the hyperbolic tangent function is just a rescaled sigmoid function [21].

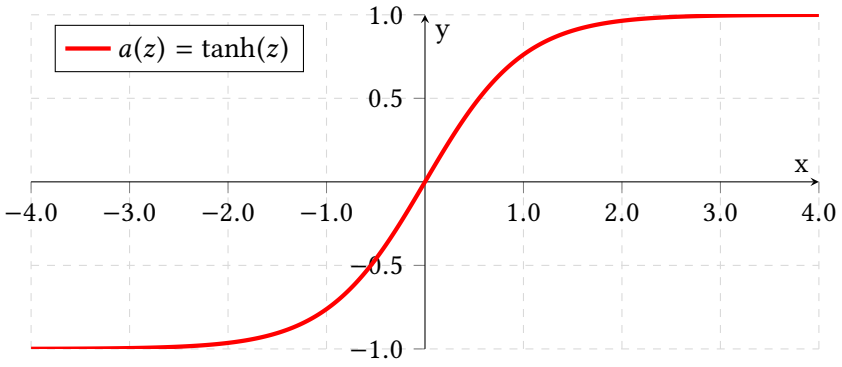


Figure 2.4: Hyperbolic tangent function

Rectified Linear Neuron—This neuron is also called *Rectified Linear Unit (ReLU)* and has the following activation function:

$$y = a(z) = max(0, z) \tag{2.7}$$

where z is $x \cdot w + b$. The function is drawn in Figure 2.5. ReLUs are one-sided, meaning that they do not have any symmetry like sigmoid or tanh activation functions (see Figures 2.3, 2.4). Furthermore, according to Glorot *et al.* [22], ReLUs make it easier to obtain sparse representations as there is a *true* 0. If we consider, for instance, an initialization of the weights of a network using a uniform distribution, around 50% of the hidden unit's outputs are true 0. Subsequent sections capture issues like *sparsity* (see Section 2.1.3.3) and *parameter initialization* (see Section 2.1.3.14). A further advantage of ReLUs is that the computational costs are cheaper since there is no need for computing

an exponentional function [22]. However, if symmetric or antisymmetric behaviour in the data should be represented, neural networks built up by ReLUs require twice the number of units compared to symmetric or antisymmetric activation functions [22]. Possible variations of ReLUs are *leaky* ReLU [23] and *parametric* ReLU [24], but we do not cover them in this approach.

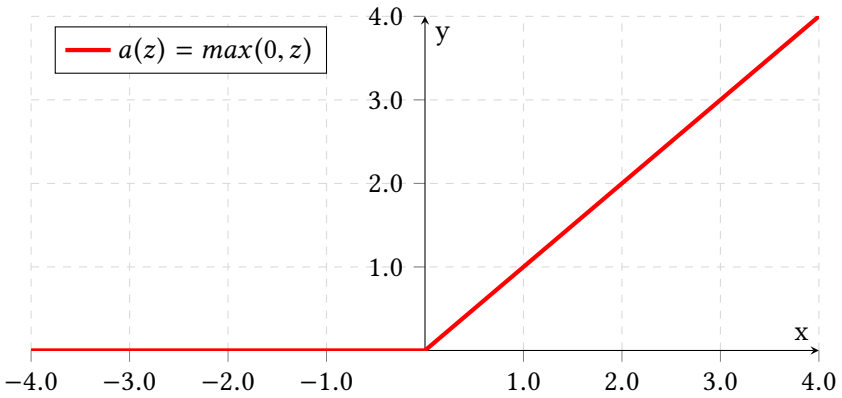


Figure 2.5: ReLU function

One term which needs to be explained in this context is *saturation of neurons*. It affects non-linear activation functions such as sigmoid or tanh. Saturation means that the neuron's activation output, meaning the result of $a(z)$, is close to the interval boundary of the particular neuron. In case of sigmoid neurons this is close to 0 or 1, and in case of tanh neurons it is close to -1 or 1. Saturation is caused by large weights or biases, whereas the weights comprise most of it. This is why we emphasize the initialization of the parameters as very important (see Section 2.1.3.14). The computed gradient of a saturated neuron is close to zero. Hence the weights are only updated by a very small amount and so, learning slows down or terminates. ReLUs, on the other hand, do not suffer from this problem as they are linear. If the weighted input to a ReLU is negative, however, it stops learning entirely as the output is 0.

Maxout Unit—Goodfellow *et al.* [2] proposed another type of activation function called *maxout unit*. It facilitates dropout (see Section 2.1.3.13) and is given by

$$a(x) = h_i(x) = \max_{j \in [1,k]} z_{ij} \tag{2.8}$$

where $z_{ij} = x^T W_{\ldots ij} + b_{ij}$ with $W_{\ldots ij}$ representing the weight vector of the unit in row i and column j. Thereby, $W \in \mathbb{R}^{d \times m \times k}$ and $b_{ij} \in \mathbb{R}^{m \times k}$ are respectively the weights and biases. Both are the same as in the previous artificial neurons, but with a slight difference. Instead of being only of dimension $d \times m$ and m, they have an additional factor k. It denotes the number of unit groups into which the input is divided up before it gets fed into the activation function [25]. From each unit group the maximum weighted

input is choosen. This circumstance is shown in Figure 2.6.
Maxout units are best used together with a regularization technique called dropout (see Section 2.1.3.13). They have shown improvement in both the optimization by dropout and the accuracy [2].

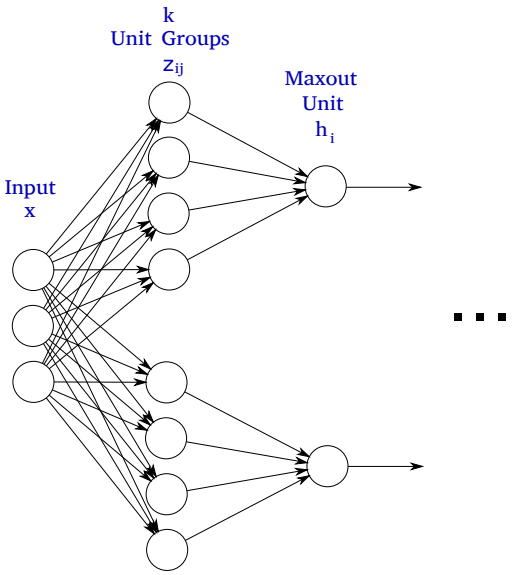


Figure 2.6: Two maxout units h_1, h_2 with $k = 4$ unit groups in front (derived from [2]). The output of both maxout units can be used for further processing.

We introduced a large variety of different artificial neurons which are available nowadays. However, there is no precise definition when to use a particular kind of neuron. For that reason, the appropriate type of neuron for a neural network can only be found by trying different neurons and choosing the one for which the network performs best.

2.1.1.2 Softmax - an Output Unit

In general, every unit introduced in the previous section can be used as an output unit (see Section 2.1.1.1). However, sometimes we want to describe the output of a neural network in terms of a probability distribution among n values. Therefore, we need a function transforming each weighted input z_i into a probability. The *softmax function* is defined as follows

$$softmax(z_i) = \frac{e^{z_i}}{\sum_j e^{z_j}}. \tag{2.9}$$

All probabilites computed by the function above always sum up to 1

$$\sum_j softmax(z_j) = 1. \tag{2.10}$$

To get an idea of how softmax works, have a look at the interactive applet at [5].

Depending on the problem we want our network to train on and depending on the data available for training, different types of learning and training a neural network can be used. We can not apply a supervised learning technique which uses labeled training data, for instance, if we only have unlabeled training examples available.
Below, we provide information about three different learning methods which use labels, no labels or both. Further, a fourth method which learns by interacting with its environment is introduced.

2.1.1.3 Supervised Learning

A *supervised learning* algorithm gets labeled training data as input. Each input x_i has a desired output y_i. According to how well the actual output y_i^* matches y_i, the algorithm adjusts its weights and biases. In general, algorithms which apply supervised learning try to predict y^* from x. This is usually done by computing $p(y^*|x)$ [6].
Supervised learning is often used for classification tasks. Let us consider the recognition of handwritten digits. The input is an image of a handwritten digit and the output is the classification of the digit by the algorithm. The desired output, on the other hand, is the digit shown in the image. For example, if we use an image of a handwritten 4 as input, then the corresponding label is 4.
Supervised learning exploits the labeling of the data set which allows it to be very accurate. However, the labeling process might be expensive, for instance, if a vast amount of data has to be labeled manually.

2.1.1.4 Unsupervised Learning

Unsupervised learning is the opposite to afore-mentioned supervised learning. This algorithm uses solely unlabeled training data (input vector x) and tries either to find features and structure patterns in the training set by reconstructing the input or to learn the entire probability distribution *p(x)* that generated the data set [6].
If we consider the recognition of handwritten digits again, unsupervised learning uses in contrast to the supervised learning approach explained above, an input image without the corresponding label. Thus, it learns by reconstructing the input without the help of labels.
An advantage over supervised learning is the absence of labels. This is why the raw input can directly used withou labeling it. This is cost-effective in terms of a non-existing manual labeling process, for instance.

2.1.1.5 Semi-Supervised Learning

A *semi-supervised learning* algorithm uses both labeled and unlabeled training data and, therefore, requires one or another a priori assumption on the input. For instance, the *semi-supervised smoothness assumption* states that if two points in a high-densitiy regions are close to each other, so are the corresponding labels, and the *cluster assumption* implies that points which are located in one cluster do likely have the same label [26]. Considering the above mentioned handwritten digit recognition again, we have a small number of input images with corresponding labels. The majority of pictures is not labeled.
Semi-supervised learning approaches are used if there is almost no labeled data available but a vast amount of unlabeled data. Moreover, if the labeling process is too expensive, semi-supervised learning is a tradeoff between labeling all data and leaving the data unlabeled.

2.1.1.6 Reinforcement Learning

A r*einforcement learning* algorithm learns by interacting with its environment. The interacting part is also often referred to as *agent*. Each action can have a reward as a result. Considering the rewards, the agent learns which action to choose next in a certain state. These rules are called *policy*. The explanation below is based on [12]. Reinforcement learning is formalized by a *Markov decision process*, containing a set of states, actions and rules for transitioning between states: s_i represents state i, s_n the terminal state, a_i action i and r_{i+1} is the reward after performing action i. The probability of the next state s_{i+1} depends solely on the current state s_i and action a_i (*Markov assumption*).

Discounted Future Reward—As it is important to perfom well in the long-term, rewards of the future have to be taken into account. The future reward from time point t is the following

$$R_t = r_t + r_{t+1} + r_{t+2} + \cdots + r_n. \tag{2.11}$$

As we are situated in a stochastic environment, the equation above needs to be modified, i.e. we can not be sure that we get the same reward for the same action. Hence, we need to add a factor γ expressing the uncertainity of rewards in the future

$$R_t = r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \cdots + \gamma^{n-t} r_n. \tag{2.12}$$

This equation is called *discounted future reward* and γ is the *discount factor* (between 0 and 1). Considering this equation, we can see that the more a reward is in the future, the less it is taken into account. Rewriting the last equation expresses R_t short in terms

of R_{t+1}

$$R_t = r_t + \gamma(r_{t+1} + \gamma(r_{t+2} + \dots) = r_t + \gamma R_{t+1}. \tag{2.13}$$

Summarizing, the algorithm must always choose an action which maximizes the discounted future reward.

Q-Learning—Q-Learning defines a function *Q(s,a)*, which exposes the maximum discounted future reward when action *a* is performed in state *s* and it is continued optimally from that point on

$$Q(s_t, a_t) = maxR_{t+1}. \tag{2.14}$$

Choosing the best action means choosing the highest Q-value

$$\pi(s) = argmax_a Q(s, a) \tag{2.15}$$

where π represents the policy. Getting this Q-function is almost the same as in Equation 2.13. By adding the discounted future reward to the current reward, we get

$$Q(s, a) = r + \gamma max_{a'} Q(s', a') \tag{2.16}$$

where the transition is given by <s,a,r,s'>. The Q-function is iteratively approximated by repeating the following update-step

$$Q[s, a] \leftarrow Q[s, a] + \alpha(r + \gamma max_{a'} Q[s', a'] - Q[s, a]) \ ,$$

where α is the learning rate, which determines how much of the difference between the previous Q-value and the new computed Q-value is taken into account, i.e. if $\alpha = 1$ then both Q-values cancel each other out and we get Equation 2.16. $Q[S, A]$ is a table, where S are the states and A are the actions. The current estimate of the Q-value of $Q(s, a)$ is given by $Q[s, a]$.
An example of a reinforcement learning approach in use is a roboter whose aim is to navigate in an unknown environment. The roboter moves in this environment and every move brings a cost which is either positive or negative. Thus, the roboter learns to move in this unknown environment.

2.1.2 Machine Learning Classifier

Since some deep learning classification approaches in the related works (see Chapter 3) use *machine learning classifier* on top of neural networks, we briefly introduce three of them. They are supervised learning approaches as they aim to estimate a probability distribution $p(y|x)$ (see Section 2.1.1.3) [6].

2.1.2.1 Logistic Regression

A *Logistic Regression (LR)* algorithm calculates the probability of an input belonging either to class 0 or 1 (binary classification). It computes the probability of class 0 by knowing the probability of class 1 with $p(y = 0|x;\theta) = 1 - p(y = 1|x;\theta)$. Thus, $p(y = 1|x;\theta)$ is large if the input belongs to class 1 and low otherwise. The following hypothesis is applied to compute this probability

$$h_\theta(x) = \sigma\left(\theta^T x\right) \tag{2.17}$$

where σ denotes a sigmoid function (see Equation 2.3) and θ represents a parameter vector. To find the optimal weights, the log-likelihood has to be maximized

$$\sum_{i=1}^{m} \log p\left(y^{(i)}|x^{(i)};\theta\right) \tag{2.18}$$

where $y^{(i)}$ is the corresponding label to the *i-th* input $x^{(i)}$. This can be done, for instance, by minimizing the negative log-likelihood with Gradient Descent (GD) [6].

2.1.2.2 Multinomial Logistic Regression

If we are not only interested in classifying the input into two classes, but rather multiple classes, we have to use *multinomial LR*. It is also referred to as *Softmax Regression (SR)* [27]. The training set is equal to the one we use in LR $\left\{\left(x^{(1)}, y^{(1)}\right), \ldots, \left(x^{(m)}, y^{(m)}\right)\right\}$. Instead of $y^{(i)} \in \{0, 1\}$, we now have $y^{(i)} \in \{0, \ldots, K\}$. The probability denotes how likely an input belongs to each class K. Thus, we compute $p(y = k|x;\theta)$ for each $k = 1, \ldots, K$. The probabilities are computed using the principle of the softmax function (see Equation 2.9). The hypothesis $h_\theta(x)$ is computed using

$$h_\theta(x) = \begin{bmatrix} p(y = 1|x;\theta) \\ p(y = 2|x;\theta) \\ \vdots \\ p(y = K|x;\theta) \end{bmatrix} = \frac{1}{\sum_{j=1}^{K} e^{\theta^{(j)T} x}} \begin{bmatrix} e^{\theta^{(1)T} x} \\ e^{\theta^{(2)T} x} \\ \vdots \\ e^{\theta^{(K)T} x} \end{bmatrix} \tag{2.19}$$

where $\theta^{(i)}$ for $i \in \{1, \ldots, K\}$ indicates the parameters of the model [27]. Training is then done by minimizing an objective function $J(\theta)$ which is, for instance, given by

$$J(\theta) = -\left[\sum_{i=1}^{m}\sum_{k=1}^{K} \mathbf{1}\left\{y^{(i)} = k\right\} \log \frac{e^{\theta^{(k)T} x^{(i)}}}{\sum_{j=1}^{K} e^{\theta^{(j)T} x^{(i)}}}\right] \tag{2.20}$$

where $\mathbf{1}\{\cdot\}$ corresponds to a *indicator function*, which results in 1 if $\mathbf{1}\{$ a true statement $\}$ and outputs 0 otherwise [27]. To find the minimum of $J(\theta)$ one can use, for example,

a gradient-based learning technique (see Section 2.1.3.7).

2.1.2.3 Support Vector Machine

A *Support Vector Machine (SVM)* [28] is a supervised learning approach which does not compute probabilites [6]. Unlike both above-mentioned approaches, SVMs decide whether an input belongs to class 1 or 2 according to the class identity. If $w^T x + b$ is either postive or negative, either the positive or negative class is present. As shown in Figure 2.7, a SVM tries to fit a *hyperplane* between two classes represented by points in a n-dimensional feature space. It is easy to seperate datapoints with 2 features using a linear hyperplane as it is shown in Figure 2.7

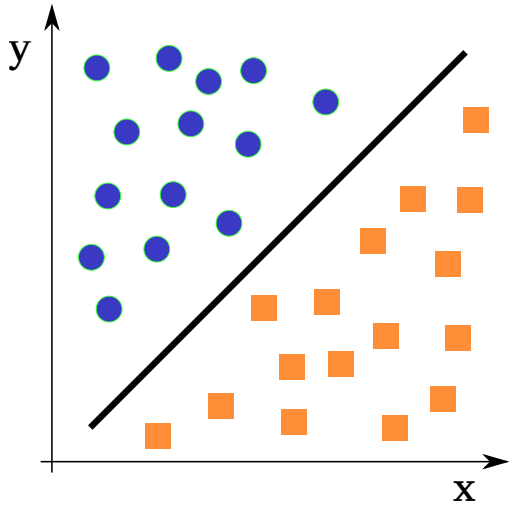


Figure 2.7: Dividing two classes in a low dimensional space using one linear hyperplane (derived from [3]).

If the datapoints consist of more features, the *kernel-trick* has to be applied. The kernel-trick transforms the data from a lower dimensional input space into a higher dimensional space. High enough to make the datapoints seperateable (see Figure 2.8) [3].

The kernel-trick bases upon the observation that many machine learning algorithms can be solely expressed in terms of dot-products between examples [6]. For instance, we can rewrite the linear function mentioned above into

$$w^T x + b = b + \sum_{i=1}^{m} \alpha_i x^T x^{(i)} \tag{2.21}$$

where α is a vector of coefficients. The dot-product can now be replaced by a *Kernel-Function* $k(x, x^{(i)}) = \phi(x) \cdot \phi\left(x^{(i)}\right)$, where $\phi(x)$ is a given feature function, which replaces x with its output. $\cdot$ indicates an inner product. Equation 2.21 can be rewritten

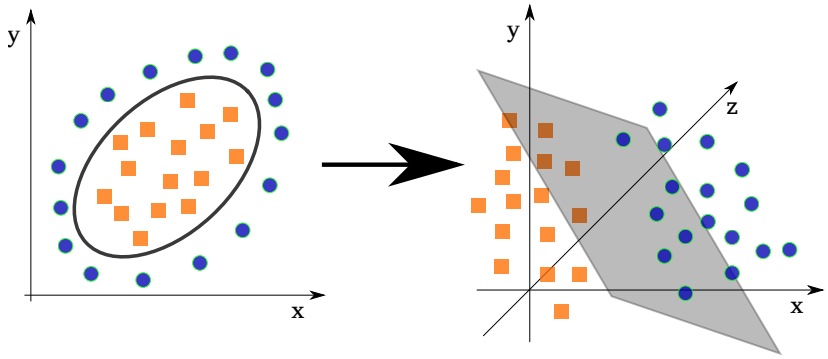


Figure 2.8: The difficulty of dividing two non-linear separateable classes in low dimensions *(on the left)*. This issue can be solved by increasing the dimensionality into a higher space *(on the right)* [4].

into a non-linear function with respect to x

$$f(x) = b + \sum_i \alpha_i k\left(x, x^{(i)}\right). \tag{2.22}$$

On the other hand, the relations between $f(x)$ and $\phi(x)$, and $f(x)$ and α are linear, though [6].
Later in the related works a *Gaussian kernel* and *Radial Basis Function (RBF)* are mentioned. They both use the following kernel

$$k(u, v) = \mathcal{N}\left(u - v; 0, \sigma^2 I\right) \tag{2.23}$$

where $\mathcal{N}\left(x; \mu, \sigma^2\right)$ represents the standard normal density given by

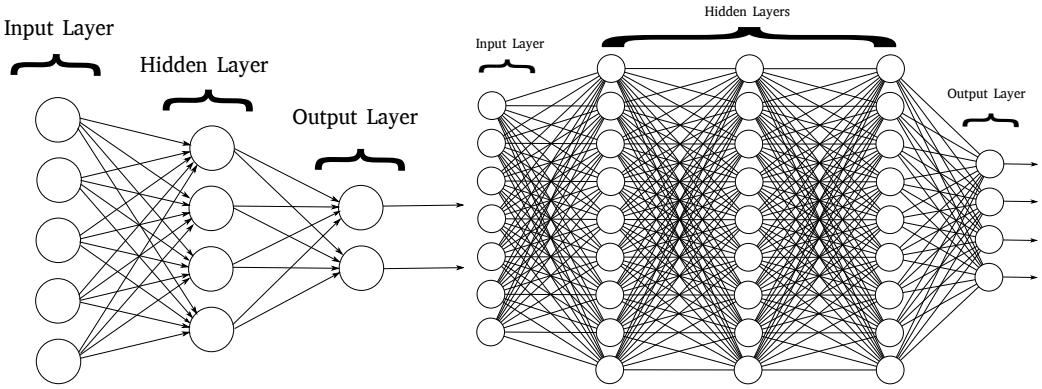
$$\sqrt{\frac{1}{2\pi\sigma^2}} e^{\left(-\frac{1}{2\sigma^2}(x-\mu)^2\right)}. \tag{2.24}$$

Here, μ represents the mean of the distribution, meaning $\mathbb{E}[x] = \mu$. The variance is indicated by σ^2. Thus, the standard deviation is denoted with σ.

2.1.3 Techniques

The groundwork of a deep learning algorithm is a deep neural network. A neural network is composed of three parts: an input layer, one or more hidden layers and an output layer. We later introduce an exception to the rule, called *Restricted Boltzmann Machine (RBM)*, which consists of only two layers. The structure of two neural networks is shown in Figure 2.9. Each layer is made up of multiple artificial neurons of the same type (see Section 2.1.1.1). The output layer, however, might be composed of softmax units (see Section 2.1.1.2). Each neuron in one layer is connected to every neuron in the

adjacent layer.
Different techniques can be used to build a neural network based on the purpose it must fulfill. This is why we introduce various types of neural networks in the following. Additionally, we describe how to obtain a deeper representation of the respective shallow neural network. For instance, Figure 2.9(a) shows a shallow neural network. By adding both more hidden layers and hidden units we are able to make it deeper (see Figure 2.9(b)). Moreover, we explain how neural networks get trained and learn, respectively. We conclude this section by illustrating different optimization techniques.



(a) Standard neural network (b) Deep neural network

Figure 2.9: Two different kinds of neural networks (derived from [5]). To preserve overview, the arrows in (b) were omitted.

2.1.3.1 Feedforward Neural Network

The first type of neural networks introduced in this chapter are *Feedforward Neural Networks (FFNNs)*. FFNNs are often also referred to as *Multilayer Perceptrons (MLPs)*. An example of a FFNN is shown in Figure 2.9(a). FFNNs approximate a function f^* (see Equation 2.25) with a function $f(x;\theta)$ (see Equation 2.26) by learning the parameter θ

$$y^* = f^*(x) \tag{2.25}$$

$$y = f(x;\theta). \tag{2.26}$$

During training, the network adjusts the values of parameter θ to get the output of $f(x;\theta)$ as close to f^* as possible. θ consists of the weights and biases. The training data set contains examples x with desired labels y^* with $y^* \approx f^*(x)$. Each input has to be of the same size, and it has to be strutured as a 1-D vector. Thereby, training is done by using Backpropagation (BP) (see Section 2.1.3.8) with gradient based learning (see Section 2.1.3.7) to adjust the weights and biases of the network.

Deep Feedforward Neural Network—Stacking more and more hidden layers consecutively yields a deeper FFNN. Figure 2.9(b) shows a Deep Feedforward Neural Network (DFFNN) with three hidden layers and more units in each hidden layer than the shallow representation in Figure 2.9(a).

2.1.3.2 Convolutional Neural Network

A *Convolutional Neural Network (CNN)* [29] works with a mathematical operation called *convolution*. CNNs are mostly used for image classification [7]. Their input has a grid-like structure, for instance, time series data represented as a 1D-grid or an image consisting of a 2D-grid of pixels [6]. We first explain the principle of convolution before we illustrate CNNs and the principle behind them.
Vividly explained, convolution is a weighted average operation, meaning to weight a function x (*input*) by w (*kernel*). This operation, which outputs a *feature map* $s(t)$, is given by

$$s(t) = (x * w)(t) = \int x(\tau)w(t-\tau)d\tau \tag{2.27}$$

where $*$ is the convolution operator and τ represents a variable which states how strong the value of the weight function $w(t-\tau)$, which is computed τ steps back in time, is included in the output at time step t. Since we work with computer data, meaning with discrete time data, Equation 2.27 can be rewritten. By taking into account that the functions x and w are only defined on integer values t a discrete convolution is obtained

$$s(t) = (x * w)(t) = \sum_{\tau=0}^{\infty} x(\tau)w(t-\tau). \tag{2.28}$$

As we pointed out in the beginning CNNs are often used for image data. This is why the input and the kernel of Equation 2.28 are replaced with a two-dimensional input I and kernel K, respectively. This yields the following equation

$$S(i,j) = (I * K)(i,j) = \sum_{m}\sum_{n} I(m,n)K(i-m, j-n). \tag{2.29}$$

This functionality is illustrated in Figure 2.10. While shifting the input window over the 2D-grid we obtain a 2D-grid with reduced dimensionality.

So far, we know how convolution is applied to a 2D-grid, for example to an image. Now we describe how CNNs exploit the convolution operation and how they work. CNNs are based on following ideas: local receptive fields, shared parameters and pooling [5]. The first one means that, in contrast to usual neural networks, output units interact with only a small fraction of the input units. This leads to *sparse interactions* [6]. For instance, consider a 2D-grid of pixels of an image as input. Each hidden neuron of the first layer has its own small region (*local receptive field*) to interact with, meaning

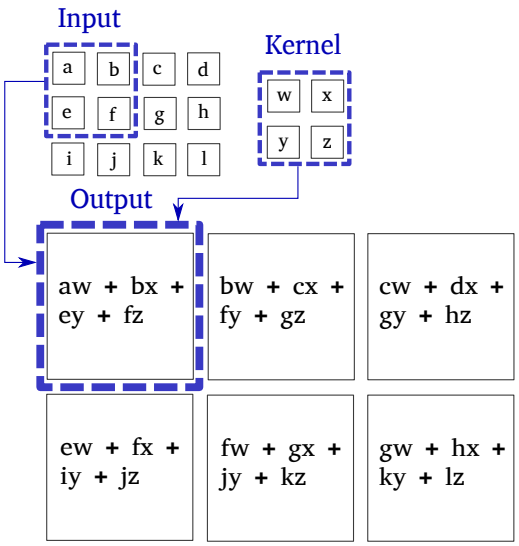


Figure 2.10: An example of 2D-convolution (derived from [6]). Note the dimensionality reduction after applying the convolution operation to the whole 2D-grid.

each hidden neuron analyzes its particular local receptive field [5]. According to our explanation above, the size of the local receptive field is the size of the kernel except for boundary positions. Using these sparse interactions the dimensionality of the input can be reduced rapidly, yet the important features (e.g. edges) of the input remain [6]. This greatly reduces the runtime if we consider that a normal matrix multiplication of m inputs and n outputs takes $O(m \times n)$, whereas the approach with limited output connections k has a runtime of $O(k \times n)$ [6].
The principle of *shared parameters* is shown in Figure 2.11. We assume that if a feature in one receptive field is detected successfully, it works also for the same feature in another position [5]. This means, that the parameter sharing function used by the convolution operation detects exactly the same features of the input [5]. The same kernel is used among the whole input, meaning, we apply every value of the kernel to every position of the input. An exceptional case are the boundary positions where, according to the design constraints of the CNNs, not every position is accumulated with every kernel member [6]. This circumstance can be seen in Figure 2.10, where the boundary fields are not multiplied with every kernel value. This greatly reduces the number of parameters. Thus, there is no need to learn a seperate set of parameters for every input unit like in fully-connected neural networks [5]. Suppose each hidden neuron is connected to its local receptive field by a 5×5 kernel K (weight matrix) and a bias b. Then, the $i, j - th$ neurons activation would look like

$$y_{i,j} = a\left(b + \sum_{m=0}^{4}\sum_{n=0}^{4} I_{j+m,i+n}K_{m,n}\right) \tag{2.30}$$

where a denotes an activation function and I the input activation at position $j+m$, $i+n$ [5].

The CNN uses the same weights K for every neuron in this layer. Another property convolutional layers have is *equvariance to translation*, meaning that if the input changes the output changes the same way [6]. For example, if we move objects in the input image, these objects move the same way in the output.

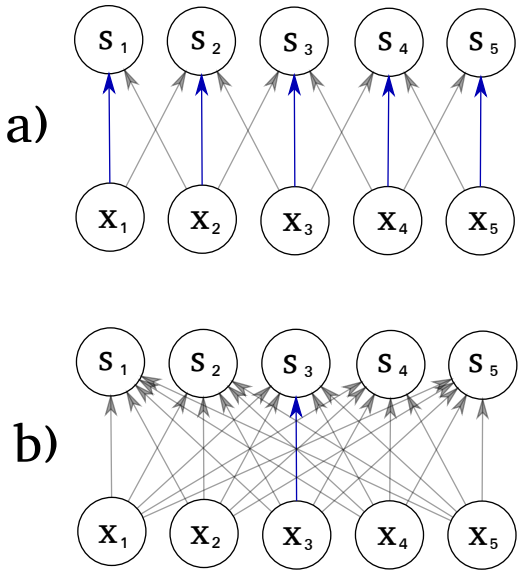
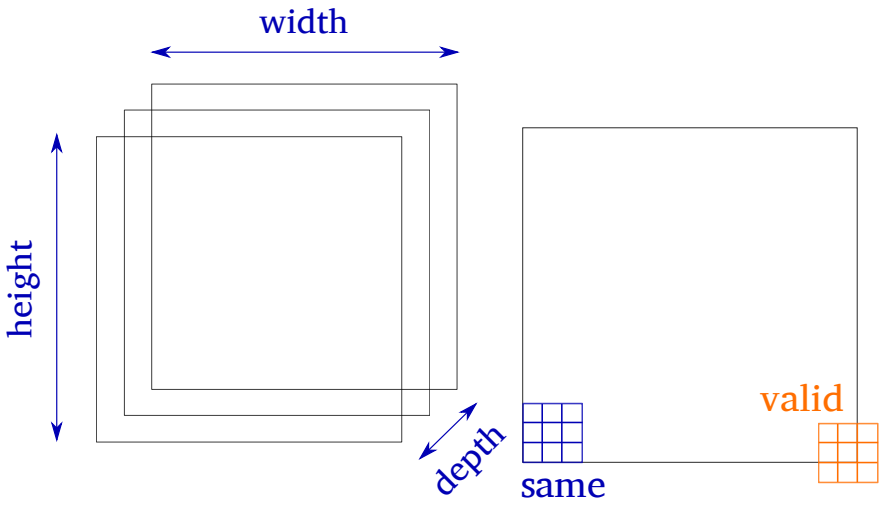


Figure 2.11: Parameter sharing. *a)* shows the parameter sharing of a convolution model, where the blue-colored arrow denotes the *middle element* of a three-element kernel. Each input uses the same parameter (middle element). *b)* shows a fully-connected model. The blue-colored arrow indicates the middle element of the weight matrix. There is no weight sharing, as each parameter is only used once (derived from [6]).

As we do not always have grey-valued input images (2D-grids) but sometimes RGB-images (3D-tensors), we need to explain *width*, *height* and *depth*. For example, a colored image of size 28×28 has a width of 28, a height of 28 and a depth of 3 (see Figure 2.12(a)). The latter number indicates the three color channels red, green and blue. This leads to three *feature maps* of size 28×28. If a convolution step is applied to this image with the help of a kernel, we get k feature maps, which yields an output depth of k [7]. Sometimes the kernel is often referred to as *patch* or *filter*. Another important term is called *stride*. This denotes the number of pixels that are shifted when the filter moves. A stride of 2, for instance, roughly halves the size of the input [7]. The above-mentioned problem with boundary positions is solved by using *padding*. We can either choose *valid* or *same* padding. The first one refers to moving the filter in such a way that it fits in the input and does not go past the edge. This reduces the output size. The latter one moves the filter of the edge, leading to an output size that is exactly the size of the input map [7]. Figure 2.12(b) illustrates both padding variations.

The last idea from above which we need to figure out is called *pooling*. Pooling layers are used directly after convolutional layers [5]. According to [7], pooling is a better



(a) Height, Width and Depth (b) Padding variations *Same* and *Valid*

Figure 2.12: Denotations in CNNs. 2.12(a) shows a layer of depth three and 2.12(b) shows same and valid padding (derived from [7]).

way to reduce the dimensionality of the feature maps than striding. The latter removes a lot of information, and is hence not the optimal way. Each unit in the pooling layer summarizes a certain location of the output of the convolutional layer. This is done with the help of a pooling function in order to get a condensed feature map [5]. Next, several pooling functions are described. First, *max-pooling* displays the maximum output activation within a rectangular region. On the other hand, *average-pooling* reports the average output activation within a rectangular region. Another pooling function is called L^2*-pooling*, which outputs the L^2-norm of the activations within the region defined by the square root of the sums of the activations [5] [7]. The last function reports the weighted average based on the distance from the central pixel [6]. Pooling can be viewed as discarding the exact positional information of a feature in the input. The approximate location relative to the other features is sufficient [5]. After the last pooling layer, one or more fully-connected layers are attached. The output consists of softmax functions to classify the input, for example an image [6].
The activation functions used in CNNs are typically ReLUs (see Section 2.1.1.1) [6]. One core principal of CNNs to keep in mind is that the large dimensionality in width and height is compensated layer by layer by a larger depth, yielding a *Convolution Pyramid* (see Figure 2.13) [7].

Deep Convolutional Neural Network—A deeper representation of a CNN is obtained the same way as of a FFNN, by stacking more layers. For an example of a deep CNN applied to face representation have a look at [30]. Another deep CNN is proposed in [31]. It is used to detect pedestrians. Another method to make standard CNNs deeper is to add 1×1 convolutions, looking on only one pixel instead of a small patch. Adding

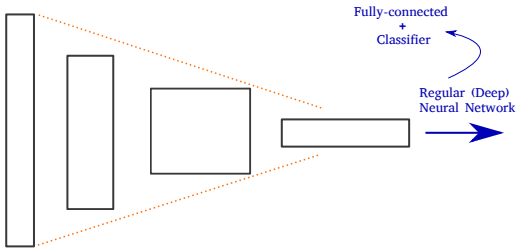


Figure 2.13: A convolution pyramid resulting in compensating a large dimensionality in width and height by a larger depth. This is done layer by layer (derived from [7]). Attached after the last convolution layer is a regular fully-connected neural network used for classification [6].

a 1×1 convolution between the kernel operation and its output yields a *mini* neural network running over the patch instead of a linear classifier (see Figure 2.14) [7]. Such an insertion gives us a deeper model with more parameters in an inexpensive way, as 1×1 convolutions are only matrix multiplications [7].

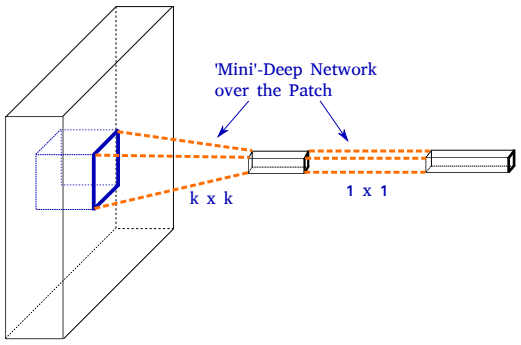


Figure 2.14: A 1×1 convolution added to a patch to obtain a *mini* neural network (derived from [7]).

2.1.3.3 Deep Belief Network

A *Deep Belief Network (DBN)* consists of either several stacked *Restricted Boltzmann Machines (RBMs)* or several stacked *Autoencoders (AEs)*. That is why we at first describe the functionality of a RBM and an AE in more detail before we discuss the structure and principle behind a DBN in the end.

Autoencoder—An AE is an unsupervised approach which aims to copy its input to its output [6]. It is mostly shallow and built-up by three layers, one input layer, one hidden layer h that tries to learn a *'code'* to represent the input x and one output layer. More hidden layers are possible, if more dimensionality reduction is needed. Typically, the number of hidden units is less than the number of input and output units. This is why AEs are often used for dimensionality reduction, as they are forced to pack all the

information of the input into less units. Using this sparse representation it needs to reconstruct the input. Briefly, this network type consists of two parts, *encoding* and *decoding*. The input is first encoded using $h = f(x)$, where f denotes the encoding function and then decoded to represent the reconstruction $r = g(h)$. Here, g indicates the decoding function [6]. Summarizing, this is how AEs try to figure out the underlying structure of a data set. The important part of an AE is not the exact copy of the input to the output but h, as h comprises useful properties of the input. Hence, when trained, the reconstruction layer of an AE is removed together with its parameters and only the input layer and hidden layer are used for further computation [14].

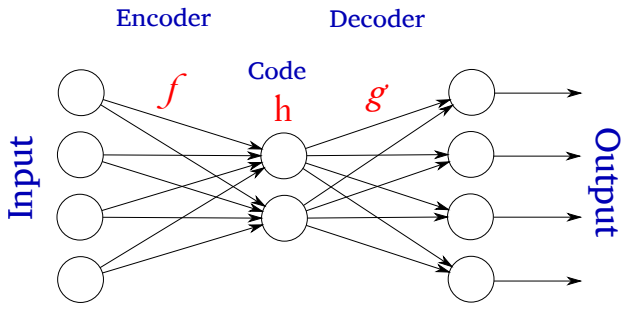


Figure 2.15: An AE with the encoding step on the left and the decoding step on the right.

There is a variety of different types of AEs, such as *undercomplete* and *overcomplete* AEs, *Regularized Autoencoders (RAEs)*, *Sparse Autoencoders (SpAEs)* and *Denoising Autoencoders (DAEs)*. Furthermore, a *Variational Autoencoder (VAE)* is an example for a *deep generative model* [6]. Below, the different kinds of AEs are depicted briefly. An undercomplete AE was in parts already described above, because it restricts the code h to a smaller dimension than the input x has. That forces it to compress the most important features of the input. Its learning process is given by

$$L(x, g(f(x))) \tag{2.31}$$

where L is a loss function computing the difference between input x and output $g(f(x))$. However, if undercomplete AEs, or AEs with code dimension equal to the input dimension, is given too much capacity, they are not able to learn important features. This is due to their learning of copying the input to the output without extracting important features of the input [6].
Although the code of overcomplete AEs is composed of a greater dimension than the input, they also suffer from the problem mentioned above. And what is worse, even linear encoder and decoder are able to skip the feature extraction. A way around this problem is to limit the model capacity by, for instance, choosing a small code size and both a shallow encoder and decoder.
Instead, RAEs use a loss function which solves the problem of only copying the input to the output the following way. It fosters the AE to have other properties in addition, for

example sparsity of the representation, smallness of the derivative of the representation and robustness to noise or to missing inputs [6].
A SpAE adds a *sparsity penalty* $\Omega(h)$ to Equation 2.31, which results in

$$L(x, g(f(x))) + \Omega(h). \tag{2.32}$$

Due to its sparsity on the hidden units, this type of AE is often used to learn features for another task, such as classification, which are then again used in pretraining this task [6].
A DAE minimizes in contrast to Equation 2.31

$$L(x, g(f(\tilde{x}))) \tag{2.33}$$

where $\tilde{x}$ represents a by some form of noise corrupted copy of x. That is why DAEs are not able to just copy the input to the output as they have to rather resolve this corruption. Have a look at Chapter 20.10.3 in [6] or [32] to get detailed information about VAEs.
AEs are widely used in classification tasks in order to get the most important features out of the input, which are then in turn used as input to a supervised learning algorithm (see Section 3.2).

Restricted Boltzmann Machine—A RBM is also used for unsupervised learning. The principle behind RBMs is similar to the one of AEs. This network type is shallow and consists of only two layers, though, one layer of visible units connected to a layer of hidden units. This concept is illustrated in Figure 2.16.

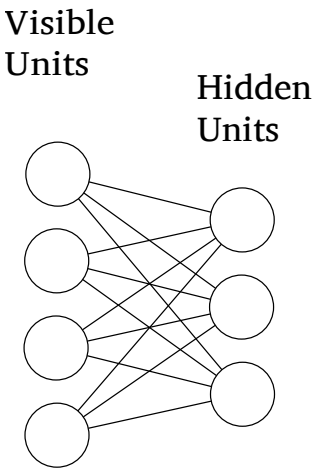


Figure 2.16: RBM with four visible units connected to three hidden units.

A RBM is an energy-based model. Its visible and hidden units are canonically binary but there are other types of visible and hidden units, too [6]. The joint probability function

is specified by the energy function E

$$P(v_m = v, h_n = h) = \frac{1}{Z} e^{-E(v,h)} \tag{2.34}$$

where v_m represents a vector of visible variables of size m and h_n a vector of hidden variables of size n. The *energy function* is given by

$$E(v, h) = -b^T v - c^T h - v^T W h \tag{2.35}$$

where b and c are respectively the biases from the visible units v and the hidden units h. W represents the weight matrix connecting the visible and the hidden layer. Z is a normalizing factor called *partition function* and is computed the following way

$$Z = \sum_v \sum_h e^{-E(v,h)}. \tag{2.36}$$

Due to the restricted structure of a RBM (see Figure 2.16, only connections between visible and hidden units), the units in one layer are conditionally independent from each other given the opposite layer [33]. Thus, the conditional probabilities $p(h|v)$ and $p(v|h)$ are given by

$$p(h|v) = \prod_i p(h_i|v) \tag{2.37}$$

$$p(v|h) = \prod_j p(v_j|h). \tag{2.38}$$

The individual conditional probabilities of a binary RBM are computed using

$$P(h_i = 1|v) = \sigma\left(v^T W_{:,i} + b_i\right) \tag{2.39}$$

$$P(h_i = 0|v) = 1 - \sigma\left(v^T W_{:,i} + b_i\right) \tag{2.40}$$

where $v^T W_{:,i}$ denotes the weighted sum of inputs $\sum_{j=0}^{m} w_{j,i} v_j$.

The above-mentioned properties make *Gibbs Sampling* efficient. Simply put, Gibbs sampling can be performed in two steps (*Block Gibbs Sampling*) [6]. We can sample a new vector h based on $p(h|v)$ and a new vector v based on $p(v|h)$.
Gibbs sampling takes a vector x of variables of length n as input. To get an appropriate probability distribution $p(x)$, a Gibbs sampling step has to be repeated several times using the last sampled vector as input. The output of sampling step i is a new vector $x^{(i)}$, where each element of the vector $x_j^{(i)}$ is derived the following way

$$x_j^{(i)} = p\left(x_j^{(i)} | x_{-j}^{(i)}\right) \tag{2.41}$$

where $x_{-j}^{(i)}$ involves all variables $x_k^{(i)}$, $k \in \{0, \ldots, j-1, j+1, \ldots, n\}$. In general Gibbs sampling $x_{-j}^{(i)}$ consists of variables $\left(x_k^{(i)}, x_l^{(i-1)}\right)$, where $k \in \{0, \ldots, j-1\}$ and $l \in$

$\{j+1, \ldots, n\}$. This means, for all sampled variables the current sample is used, and for all yet non-sampled variables the last sample is used. The conditional independence from all variables of one layer given the other in a RBM allows us to perform block Gibbs sampling. As a result, Gibbs sampling is performed for each variable $x_j^{(i)}$ of vector $x^{(i)}$ simultaneously (see Equation 2.41) [6].
Training an RBM is performed by the help of *Contrastive Divergence (CD)* or *Kullback-Leibler (KL) divergence* [6].

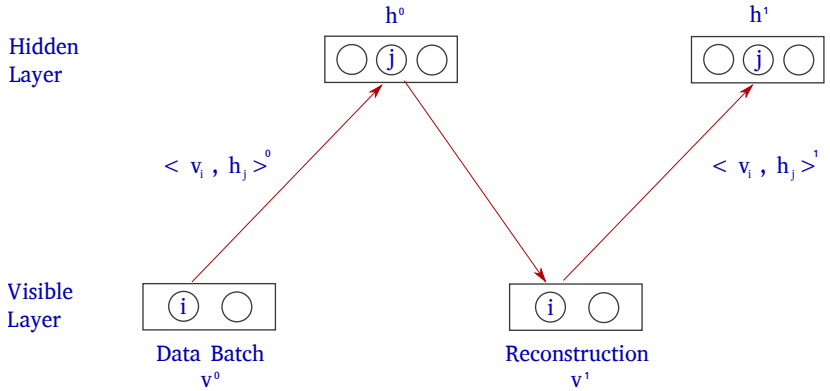


Figure 2.17: Reconstruction of the RBM input using a positive and negative gradient, respectively.

CD is mostly used to update the weights of RBMs. Recall Figure 2.17 to follow the subsequent explanation. First, the input data from vector v^0 of the visible layer is used to compute a hidden vector. A hidden vector h^0 is sampled from this outcome. Afterwards, the *positive gradient* $< v^0, h^0 >^0$, which represents the outer product of the visible vector and the hidden vector, is computed. Continuing from h^0, a reconstruction vector v^1 is sampled. A hidden vector h^1 is sampled using v^1, afterwards. As before, the outer product $< v^1, h^1 >^1$, which is called the *negative gradient*, is computed. Eventually, the weights w and the biases b and c are updated with the following equations

$$w \longleftarrow w + \eta \left(< v^0, h^0 >^0 - < v^1, h^1 >^1 \right) \tag{2.42}$$

$$b \longleftarrow b + \eta \left(v^0 - v^1 \right) \tag{2.43}$$

$$c \longleftarrow c + \eta \left(h^0 - h^1 \right) \tag{2.44}$$

where η indicates the learning rate [34].

Altogether, RBMs work by reconstructing and recreating the input data, respectively. This is done without labels since RBMs extract important features on their own.
As mentioned above, DBNs consist of stacked RBMs (or stacked AE). Figure 2.18 visualizes the principle of a DBN built-up by three RBMs on top of each other which works as follows. The first RBM consisting of a visible layer with input x and the first hidden layer h_1 is trained as a normal RBM by reconstructing the input. After training this

RBM, the hidden layer h_1 and the second hidden layer h_2 form the next RBM, which is trained with the output of the underlying RBM. This concept continues until the last layer h_i, with layer h_{i-1} acting as visible layer, is reached. As a result, DBNs are able to learn high hierarchical structure features of the input x.
In both cases, stacked RBMs and stacked AEs, two methods called *pre-training* and *fine-tuning* show performance improvements [35]. Pre-training is the *unsupervised* process we already introduced above, meaning stacked RBMs/AEs get trained *greedy layer-wise* [36]. First, the first RBM/AE gets trained with the input, then the next RBM/AE gets trained using the output of the former as input, and so on. In case of stacked AEs only the input layer and the hidden layer of a single AE remain, as they are responsible for the compressed feature representation. The reconstruction layer is only used for training. Thus, a stacked AE consists of stacked encoders [14]. If a DBN is used for classification tasks, a fine-tuning step is applied. This is performed in a *supervised* manner. On top of the pre-trained DBN another layer is stacked upon, for instance a SVM, a LR classifier or another neural network [37]. Then, the whole network is trained using a gradient-based learning technique (see Section 2.1.3.7).

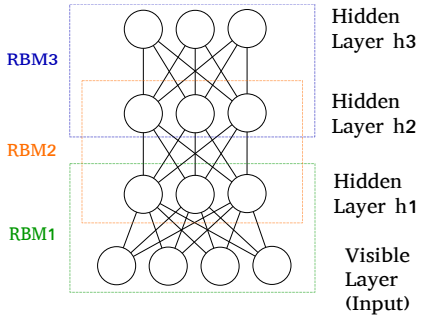


Figure 2.18: An example for a DBN consisting of stacked RBMs (derived from [8]).

2.1.3.4 Recurrent Neural Network and Long Short-Term Memory

The kind of neural network explained next is called *Recurrent Neural Network (RNN)*. We show how such networks work and illustrate the principle of *unfolding* RNNs. Furthermore, the problem of long-term dependencies gets captured. Along with it, we propose two possible solutions to overcome this problem which are called Long Short-Term Memory (LSTM) and Gated Recurrent Unit (GRU). Both are explained in detail at the end of this section.
On the contrary to FFNNs which only use the current input, RNNs take also the output of the hidden layer of the previous step into account. Hence, the decision at time step t is affected by the decision reached at time step $t-1$. As it incorporates decisions from previous steps, it possesses a backward loop which is shown in Figure 2.19. This figure also displays the unfolded illustration of a RNN over time, which implies that

the network's input at time step t is defined by the hidden state s_{t-1} and x_t, where s_{t-1} represents the *memory* of the network of the previous step. The parameters U, V and W are shared among all steps. This is another difference to FFNNs which use different parameters at every layer. In this Figure, three different weight matrices U, V and W are shown. Thereby, U weights the input, V the output of the hidden state and W is a transition matrix which weights the values of the hidden state.
The calculation steps are as follows. First, the hidden state s_t needs to be computed with

$$s_t = a(Ux_t + Ws_{t-1} + b) \tag{2.45}$$

where a is a non-linear activation function (see Section 2.1.1.1) and b represents an added bias.
Then, s_t is multiplied with V and added to a bias c. The result is fed into, for instance, a softmax function (see Section 2.1.1.2) to form an output o_t. In this case the output o_t is given by

$$o_t = softmax(Vs_t + c). \tag{2.46}$$

In the subsequent time step $t + 1$, s_t is multiplied with W and added to b, and behaves as an input to the hidden state s_{t+1}.
For training RNNs, meaning to adjust U, V, W and b, c, also a gradient-based learning algorithm is applied but with a little extension and it is therefore called *Backpropagation Through Time (BPTT)*. As shown in Figure 2.19, the parameters are shared among all time steps and this is why the gradient needs to be computed for all previous time steps, too. The principle of BPTT is explained in Section 2.1.3.8.

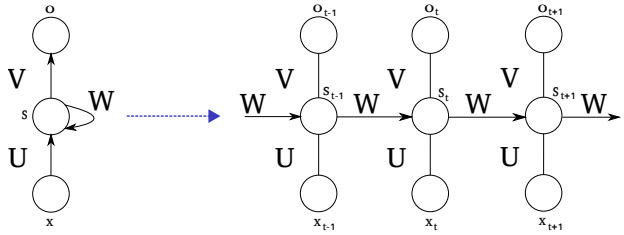


Figure 2.19: A RNN on the left, the unfolded representation over time t on the right (derived from [9]).

A simpler representation of a RNN is given in Figure 2.20. The input x_t is weighted with U and fed into the hidden layer s_t. Concurrently, the state of the hidden layer of the previous time step is also fed into the hidden layer weighted by W. The output of the hidden layer is then weighted by V and provided to the output layer, which computes the output o_t.

As already mentioned, RNNs take previous information into account. However, there comes a problem along with it, which involves *long-term dependencies*. This is not a particular problem of RNNs since it might equally include deep neural networks with a

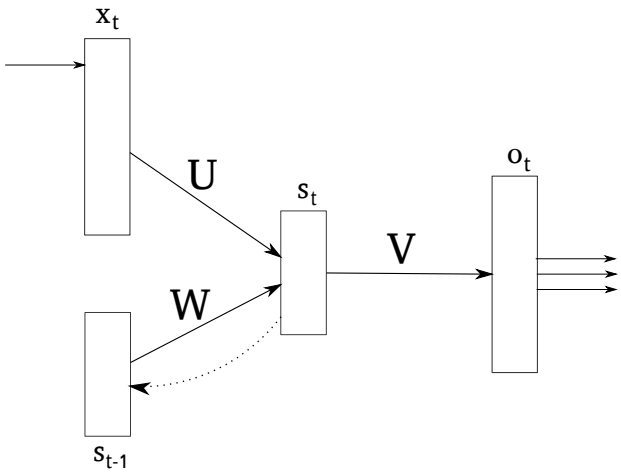


Figure 2.20: Another representation of a RNN to clarify the usage of the previous hidden state which is fed into the current hidden state (derived from [10]).

large depth. The issue with learning long-term dependencies was explored in detail by Bengio *et al.* [38] and Hochreiter *et al.* [39]. Hochreiter and Schmidhuber [40] introduced a type of network which is able to learn those long-term dependencies and which was subsequently refined by others. It is called *long short-term memory network*.

Long Short-Term Memory—LSTM networks belong to *gated* RNNs which are according to [6] the most effective sequence models used in practical applications. These kinds of RNNs solve the long-term dependencies problem by constructing a path through time with neither vanishing nor exploding derivatives and gradients, respectively. Gated RNNs learn on their own, at which point in time to discard the long-term information from the previous steps.
The usual hidden units of RNNs are replaced by LSTM *cells*. As illustrated in Figure 2.21, the input x_t and the output of the previous step h_{t-1} are given as usual input to the activation function a of the input unit, where a represents a non-linearity, for instance, tanh (see Section 2.1.1.1). In addition to that, both values are also fed into the three gating units, all equipped with a sigmoid activation function. First, the LSTM cell decides if and how much of the input is accumulated into the *state* s_t. This is done by using the *input gate*. The input gate, therefore, updates the state by combining the input and the information needed from the previous steps with the information in the state unit. The *forget gate* is next. It determines which of the values in the state unit must be kept for further processing steps and which ones to forget. It does this by merging the information of the previous state s_{t-1} with the output of the sigmoidal forget gate which expresses a value between 0 and 1, where a 0 deletes a value completely and a 1 keeps a value completely. The linear loop built by feeding back the state into the output of the forget gate is called *self-loop* (see Figure 2.21). The output of the state s_t is fed into a non-linear activation function a as in usual RNNs. There is, however, a slight

difference since the *output gate* is in charge of deciding whether the output is allowed to pass through or not and if, which parts of the information are needed. Additionally, Goodfellow *et al.* [6] mentioned that the state unit can also be used as an additional input to the three gating units (dotted arrows in Figure 2.21).
Figure 2.22 shows an unfolded representation of a LSTM cell. There it can be seen that the cell state is modified with solely linear operations. For simplicity issues, we let out the additional third input of the state to the three gating units.

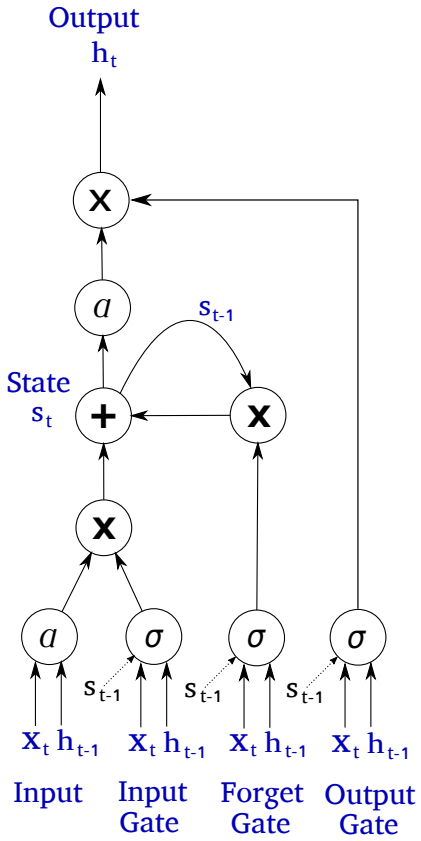


Figure 2.21: A LSTM cell, where both a denote the opportunity of using any non-linear activation function and σ corresponds to the sigmoid activation function. The dotted arrows indicate the possibility of the state of the previous step s_{t-1} as extra input to the gating units at time t (derived from [6]).

In the following, the equations needed to perform learning in LSTM networks are described briefly. These equations are derived from both [6] and [11]. As mentioned, the forget gate is in charge of deleting information from the cell state. It computes

$$f_t = \sigma\left(U_f x_t + W_f h_{t-1} + b_f\right) \tag{2.47}$$

where x_t is the input, h_{t-1} the output of the previous step and U_f and W_f indicate weight matrices for the input and the hidden vector, respectively. A bias is given by b_f.

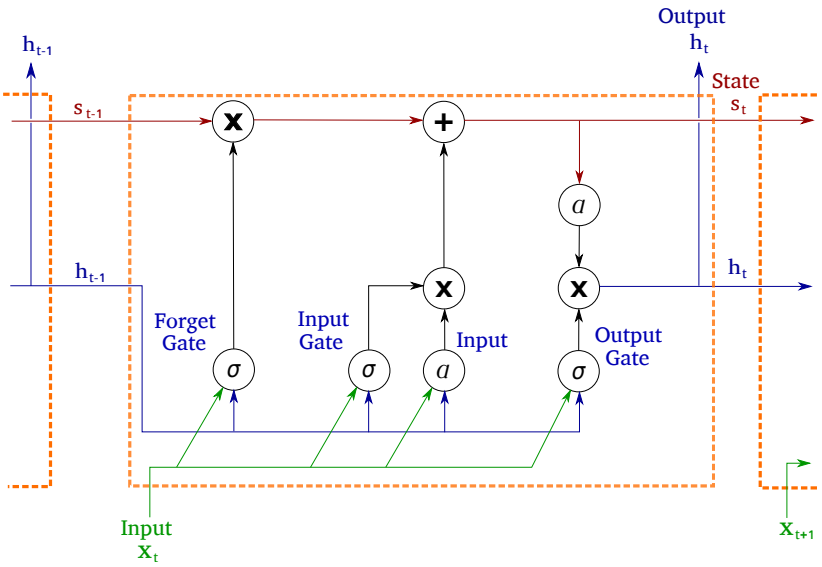


Figure 2.22: An unfolded LSTM network to illustrate the flow of the state s_t and the output h_t (derived from [11]).

The input gate unit g_t is computed as follows

$$g_t = \sigma\left(U_g x_t + W_g h_{t-1} + b_g\right) \tag{2.48}$$

where as above, the input x_t and hidden vector of the previous step h_{t-1} are weighted by matrices U_g and W_g. A bias is represented by b_g.
The ordinary input unit as in standard RNNs is given by

$$i_t = a\left(U_i x_t + W_i h_{t-1} + b_i\right) \tag{2.49}$$

where a denotes any non-linear activation function and U_i, W_i and b_i represent the matrices and the bias into the LSTM cell, respectively.
The internal cell state is updated, using Equations 2.47, 2.48, 2.49 and the previous state s_{t-1}, with

$$s_t = f_t s_{t-1} + g_t i_t. \tag{2.50}$$

Finally, the output h_t, where the output gate is capable of how much information is let through, is given by

$$h_t = a\left(s_t\right) q_t \tag{2.51}$$

with q_t representing the output gate

$$q_t = \sigma\left(U_q x_t + W_q h_{t-1} + b_q\right) \tag{2.52}$$

where U_q, W_q and b_q indicate the weight matrices and the bias of the output gate. The activation function a of Equation 2.51 can again be choosen from any non-linear

activation function, for example, tanh.
Every gate possesses its own parameters, meaning its own weights and biases. As already stated out above, the state s_t can be used with its own weight as an additional input to the three gating units, which results in three extra parameters. For further knowledge about this kind of LSTM, have a look into [6] or at [11], where more variatons of LSTM cells are described.

Gated Recurrent Unit—Another type of a gated unit was proposed by Cho *et al.* [41]. It is called *Gated Recurrent Unit (GRU).* A GRU cell is similar to a LSTM cell but uses one less gate. The equations performed in a GRU cell are described below according to [41] and [42].
A reset gate r_t is obtained by

$$r_t = \sigma\left(W_r x_t + U_r h_{t-1}\right). \tag{2.53}$$

Further, an update gate z_t is computed with

$$z_t = \sigma\left(W_z x_t + U_z h_{t-1}\right). \tag{2.54}$$

The current input is denoted by x_t. h_{t-1} represents the hidden state of the previous time step *t-1*. W_r, U_r and W_z, U_z indicate the weight matrices of the reset and the update gate, respectively. The candidate activation is given by

$$\tilde{h_t} = a\left(W x_t + U\left(r_t \cdot h_{t-1}\right)\right) \tag{2.55}$$

where U and W again represent weight matrices of the candidate function and $\cdot$ is an element-wise multiplication. Moreover, a can be any activation function. For instance, [41] use a sigmoid activation function σ, whereas [42] apply a tanh activation function. r_t represents the reset gate, which is computed in Equation 2.53. The actual activation and, thus, the output of the GRU cell at time step t is finally computed by

$$h_t = \left(1 - z_t\right) h_{t-1} + z_t \tilde{h_t} \tag{2.56}$$

where z_t and $\tilde{h_t}$ denote respectively the update gate (see Equation 2.54) and the candidate activation (see Equation 2.55). The previous hidden state is given by h_{t-1}. h_t in Equation 2.56 is the equivalent to the output of a LSTM cell represented with h_t in Equation 2.51, Figure 2.21 and Figure 2.22.

Deep Recurrent Neural Network—As depicted in the context of DBNs in Section 2.1.3.3, where we stacked several RBMs on top of each other, we can also build a deeper architecture of a RNN by stacking up multiple RNN hidden layers, e.g. multiple LSTM cells. On the other hand, if we pay attention to the unfolded representation of a RNN, we

may already denote this as deep (see Figure 2.19). Furthermore, Pascanu *et al.* [43] show other ways of expanding a RNN to a deeper architecture.

2.1.3.5 Deep Q-Network

A *deep Q-network*, which relies on the Q-learning algorithm (see Section 2.1.1.6), is the last neural network type introduced in this chapter. In this context, we also explain the meaning of experience replay and exploration - exploitation. With the Q-learning algorithm introduced in Section 2.1.1.6, a deep neural network, which is able to compute the Q-value, can be created. Its input is a state s and it outputs the Q-value for each possible action a (see Figure 2.23). This is beneficial, since it gets easy to pick the action with the highest Q-value this way. Two techniques making deep Q-learning work more effectively are called *experience replay* and *exploration - exploitation*. The former uses a replay memory to store all the experiences $< s, a, r, s' >$ computed during the run of the algorithm. From this replay memory, random mini-batches are used to train the network instead of the most recent transition. Exploration - exploitation is based on ε-greedy exploration. As the network is initialized randomly, the choice of the highest Q-value is random. The problem that comes along with it is that this choice is *greedy*, which means that it chooses the first possible highest Q-value. To overcome this problem ε-greedy exploration was introduced. This strategy chooses a random action with probability ε, with probability $(1 - \varepsilon)$, on the other hand, it chooses the action with the highest Q-value. ε should be decreased from 1 to 0.1 while training is progressing.
All in all, the exploitation step makes the best decision out of the current information and exploration gathers more information. Hence, enought information has to be collected to make the best decisions out of it [12].
A deep reinforcement algorithm, namely *Deep Q-Learning with Experience Replay* was shown in [44] from *DeepMind Technologies*. The approach is applied to Atari-Games.

2.1.3.6 Cost Function

A *cost function C*, often also referred to as *loss function L* is used to compute the error during training the network and helps to adjust the parameter θ. Note that we use the terms cost and loss function synonymously throughout this chapter. The error is mostly measured by taking the difference between the acutal output and the desired one. In the following, important cost functions are explained.
Cross-entropy is defined by

$$C = -\frac{1}{n} \sum_{x} (\hat{y} \ln y + (1 - \hat{y}) \ln(1 - y)) \tag{2.57}$$

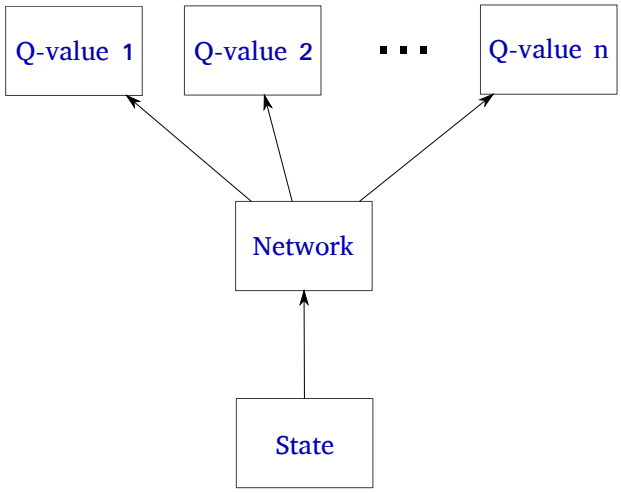


Figure 2.23: The principle of a deep Q-network (derived from [12]).

where n indicates the total number of training data, $\hat{y}$ represents the desired output for input x and y is the outcome of the activation function $a(z)$ with z being the weighted sum of inputs $\sum_j x_j w_j + b$, where w are the specified weights and b is an overall bias [5]. Cross-entropy is a positive cost function, meaning C > 0 and if the network outputs all y close to $\hat{y}$, C is close to zero, C ≈ 0.
The *log-likelihood* cost function is given by

$$C = -\frac{1}{n} \sum_x \ln y \tag{2.58}$$

where y denotes the outcome of the activation function a, x indicates the input and n is the total number of training data. The log-likelihood function is small if the network is sure of the right output.
The *Sum of Squared Errors (SSE)* is different from the *Mean Squared Error (MSE)* in that way, as the latter one calculates the average of the SSE. The equation for SSE is given by

$$SSE = \sum_{i=1}^{n} (\hat{y}_i - y_i)^2 \tag{2.59}$$

and the MSE is defined by

$$MSE = -\frac{1}{n} \sum_{i=1}^{n} (\hat{y}_i - y_i)^2 \tag{2.60}$$

where n represents the total number of training data, $\hat{y}$ is the desired output and y indicates the predicted output.
RBMs sometimes use the difference between the computed probability distribution P and the corresponding true probability distribution Q of the input as cost function. It is

measured by using the KL divergence given by

$$D_{KL}(P||Q) = \int_{-\infty}^{\infty} p(x) log\left(\frac{p(x)}{q(x)}\right) dx \tag{2.61}$$

where $p(x)$ and $q(x)$ represent the density functions of P and Q, respectively. Figure 2.24 illustrates both density functions and their mismatch. After several iterations, this gap should nearly close and both functions overlap.

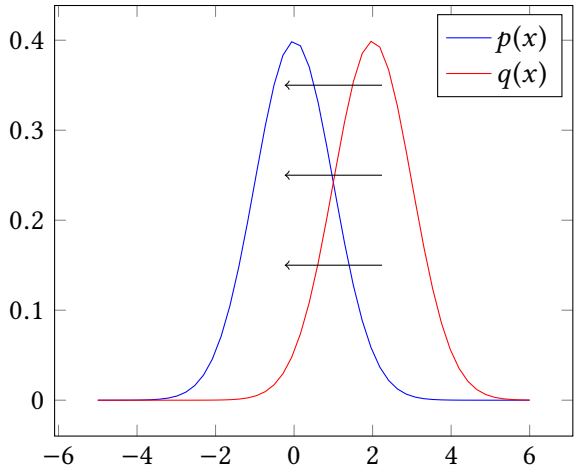


Figure 2.24: $p(x)$ and $q(x)$ denote the density functions of the probability distributions P and Q. The aim is to let both density functions overlap by minimizing the error computed with the KL divergence and thus adjusting the weights.

2.1.3.7 Gradient-based Learning

According to the computed error, *gradient-based learning* algorithms are used to update the parameters θ (weights and biases) of a network in order to make it more precise. These algorithms are mostly used in combination with BP methods (see Section 2.1.3.8), which are necessary for computing the gradient of the cost function.
Hereafter, two gradient descent techniques are introduced. Thereby, *Stochastic Gradient Descent (SGD)* is an improvement of the standard *GD*. The aim of gradient-based learning is to get a cost function as small as possible, meaning $C(\theta) \approx 0$. This indicates that a neural network learns well as the output is close to the desired result.

Gradient Descent—With GD, we want to find weights w_k and biases b_l to make a cost function (see Section 2.1.3.6) as small as possible. The GD update rule is defined as follows

$$w_k \longleftarrow w_k - \eta \frac{\delta C}{\delta w_k} \tag{2.62}$$

$$b_l \longleftarrow b_l - \eta \frac{\delta C}{\delta b_l}. \tag{2.63}$$

Here, η denotes the *learning rate* (see Section 2.1.3.10). All components $\frac{\delta C}{\delta w_k}$ and $\frac{\delta C}{\delta b_l}$, respectively, are contained in a gradient vector ∇C. This process of updating the weights and biases can be thought of as moving these values closer to the minimum of the cost function with step size η. Gradient descent comes along with some problems, though. It takes, on the one hand, a long time if a large number of training inputs is used and hence the network seems to learn slowly. That is, to get the gradient ∇C, all gradients ∇C_x for each training input x have to be computed, as the cost function averages over all costs ∇C_x. The principle of GD is illustrated in Figure 2.25.

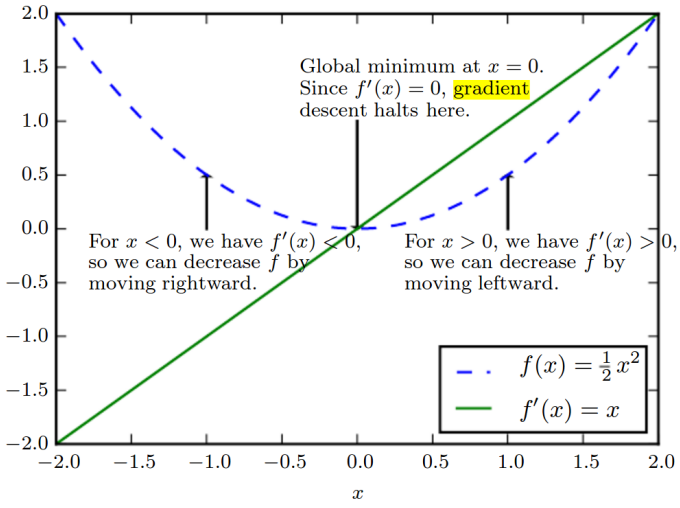


Figure 2.25: The principle behind GD [6].

Stochastic Gradient Descent—As mentioned previously, SGD is an improvement over GD. This technique speeds up learning, as it only uses a small number of training inputs to estimate the gradient ∇C. These training inputs are randomly choosen and often referred to as *mini-batch*. The update rule for SGD is given by

$$w_k \longleftarrow w_k - \frac{\eta}{m} \sum_j \frac{\delta C_{X_j}}{\delta w_k} \tag{2.64}$$

$$b_l \longleftarrow b_l - \frac{\eta}{m} \sum_j \frac{\delta C_{X_j}}{\delta b_l} \tag{2.65}$$

where m denotes the size of the mini-batch. The elements of the mini-batch are termed with X_j. After all X_j are used up, another randomly picked mini-batch is trained. This process is repeated until all training examples are used up, meaning one training *epoch*

is completed. Besides, the number of how often a mini-batch is fed into the neural network, is referred to as number of *iterations*. An efficient way to compute the partial derivatives shown in Equations 2.62, 2.63 and Equations 2.64, 2.65 is the BP approach described in the next section.

Optimization of SGD—SGD can suffer from high oscillations and hence learning occurs slowly. *Momentum* [45] names an additional term that overcomes this problem by including knowledge from previous steps. Thus, it helps SGD to find the right direction faster and dampens the oscillations as shown in Figure 2.26 [7]. The update rule for the momentum approach with SGD is given by

$$v \longleftarrow \phi v - \eta \nabla_\theta \frac{1}{m} \sum_{i=1}^{m} L_i \tag{2.66}$$

$$\theta \longleftarrow \theta + v \tag{2.67}$$

where the parameters (weights and biases) are denoted generally with θ, η expresses the learning rate and L_i is the computed loss of mini-batch i. Besides, a new variable v is introduced. It indicates a kind of velocity, meaning it involves both the direction and the speed of the parameters through the parameter space. ϕ is yet again a new hyperparameter terming the contribution of the update vector of the past time step v_{t-1} to the current one v_t. The larger ϕ is relative to η, the more previous gradients are involved in the computation of the current direction [6].

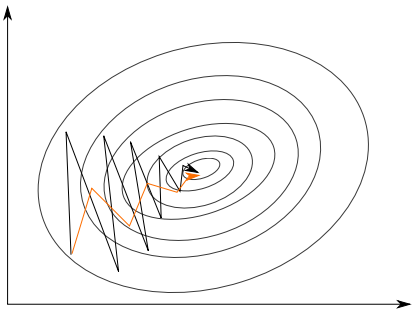


Figure 2.26: The principle behind momentum optimization. The gradients with an additional momentum term (colored orange) accelerate learning (derived from [6]).

2.1.3.8 Backpropagation and Backpropagation Through Time

The weights and biases in a deep neural network are adjusted during training by applying gradient-based learning (see Section 2.1.3.7). Therefore, a method is needed that computes the gradient of a cost function with respect to weights and biases. Moreover, it has to propagate the error computed by the cost function (see Section 2.1.3.6) back

through the network. Hence, this algorithm is called *Backpropagation (BP)*. Below, the principle behind this method is explained in more detail.
The general BP algorithm can be applied to almost all neural network structures except RNNs. These nets need a more specific algorithm called *Backpropagation Through Time (BPTT)*. BP terms the computation of the gradient from the output layer back to the input layer. One important constraint of BP is, that the cost function, which is used, has to be differentiable.

Backpropagation—The subsequent algorithm makes use of different abbreviations which are quickly explained below.

- $w^l_{jk} \longrightarrow$ weight for the connection from the k^{th} neuron in the $(l-1)^{th}$ layer to the j^{th} neuron in the l^{th} layer
- $b^l_j \longrightarrow$ bias for the j^{th} neuron in the l^{th} layer
- $a^l_j \longrightarrow$ activation of the j^{th} neuron in the l^{th} layer
- $\delta^l_j \longrightarrow$ error in the j^{th} neuron in the l^{th} layer
- $z^l_j \longrightarrow$ weighted input to the j^{th} neuron in the l^{th} layer
- $a \longrightarrow$ activation function of the neurons (see Section 2.1.1.1)

As computing the partial derivatives $\frac{\delta C}{\delta w^l_{jk}}$ and $\frac{\delta C}{\delta b^l_j}$ for GD (see Section 2.1.3.7) is expensive, the BP approach is applied to approximate these derivatives.
As already mentioned, training the network is done by adjusting and changing the weights and biases, respectively. This requires three steps. First, a forward pass called *forward propagation* is performed. This step computes the output of the neural network depending on the input. Afterwards, the error is measured by means of comparing the actual output computed in the first step with the desired output. This is done with the help of a cost function. In the last step, a backward pass is executed. The error calculated in the previous step is propagated back to the input units. Along the way the weights and biases are modified to reduce the error term. This is where BP takes place. It computes the gradients of the cost function with respect to the weights and biases.
Of course, the steps described above are only suitable for supervised learning algorithms (see Section 2.1.1.3), since they use pairs of input and corresponding output labels to train a network.
According to [5], four fundamental equations are required to understand and apply BP. First, the overall error in the output layer is denoted with δ^L and its components δ^L_j are computed as follows

$$\delta^L_j = \frac{\delta C}{\delta a^L_j} a'(z^L_j). \tag{2.68}$$

As a matrix-based form is required, Equation 2.68 needs to be rewritten into

$$\delta^L = \nabla_a C \odot a'(z^L) \tag{2.69}$$

where $\nabla_a C$ is a vector which includes the partial derivatives $\frac{\delta C}{\delta a_j^L}$.
The error δ^l is expressed in terms of the error in the next layer using

$$\delta^l = ((W^{l+1})^T \delta^{l+1}) \odot a'(z^l) \tag{2.70}$$

where $(W^{l+1})^T$ is the transponed weight matrix for layer *l+1*. Intuitively explained, $((W^{l+1})^T \delta^{l+1})$ moves the error backward and returns the output error at layer *l*. Applying the remaining part of the equation gives back the error δ^l in the weighted input to layer *l*.
By combining Equations 2.69 and 2.70 the error δ^l can be computed for any layer in the neural network.
The third equation is given by

$$\frac{\delta C}{\delta b_j^l} = \delta_j^l. \tag{2.71}$$

This implies that the error is equal to $\frac{\delta C}{\delta b_j^l}$ which represents the rate of change of the cost with respect to a bias. The following equation is similar to Equation 2.71 but instead of being applicable with respect to the bias it is used with respect to the weights

$$\frac{\delta C}{\delta w_{jk}^l} = a_k^{l-1} \delta_j^l. \tag{2.72}$$

Considering Equations 2.71 and 2.72, both derivatives necessary for GD (*left side* of the equations) can be substituted with terms whose computation is already known (*right side* of the equations).
The five steps of the BP algorithm derived from the equations above are shown below:

1. **Input**: Set the corresponding activation a^1 for the input layer.
2. **Feedforward:** For each $l = 2, 3, \ldots, L$ compute $z^l = w^l a^{l-1} + b^l$ and $a^l = a(z^l)$.
3. **Output error** δ^L: Compute the vector $\delta^L = \nabla_a C \odot a'(z^L)$.
4. **Backpropagate the error:** For each $l = L-1, L-2, \ldots, 2$ compute $\delta^l = ((W^{l+1})^T \delta^{l+1}) \odot a'(z^l)$.
5. **Output:** The gradient of the cost function is given by $\frac{\delta C}{\delta b_j^l} = \delta_j^l$ and $\frac{\delta C}{\delta w_{jk}^l} = a_k^{l-1} \delta_j^l$.

The output of *step 5* is then used for updating the weights and biases with gradient-based learning (see Section 2.1.3.7).

In the following a short parenthesis about the *chain rule of calculus* is given, as the BP algorithm builds upon it [6]. This rule states that the derivative of a function can be computed by using derivatives which are already known

$$\frac{dz}{dx} = \frac{dz}{dy}\frac{dy}{dx}. \tag{2.73}$$

Here $y = f(x)$ and $z = f(g(x)) = f(y)$, where x and y are real numbers. Generalizing the equation above into using vectors $x \in \mathbb{R}^m$ and $y \in \mathbb{R}^n$ with functions $g : \mathbb{R}^m \longmapsto \mathbb{R}^n$ and $f : \mathbb{R}^n \longmapsto \mathbb{R}$ gives us

$$\frac{\partial z}{\partial x_i} = \sum_j \frac{\partial z}{\partial y_j}\frac{\partial y_j}{\partial x_i}. \tag{2.74}$$

Equation 2.74 can be equivalently rewritten into

$$\nabla_x z = \left(\frac{\partial y}{\partial x}\right)^T \nabla_y z \tag{2.75}$$

which represents a vector notation, where $\frac{\partial y}{\partial x}$ is the $n \times m$ Jacobian matrix of g. Such a matrix contains all partial derivatives of a function. Thus, the Jacobian matrix $J \in \mathbb{R}^m \times \mathbb{R}^n$ of a function $f : \mathbb{R}^m \longmapsto \mathbb{R}^n$ has the following entries: $J_{i,j} = \frac{\partial f(x)_i}{\partial x_j}$.

Backpropagation Through Time—As already mentioned, BPTT is applied in RNNs. Figure 2.19 indicates that the error has to be propagated back through several layers. BPTT works similar to the general approach, but with slight differences in the computation. Considering the same parameters as depicted in Section 2.1.3.4, we have the weight matrices U, V and W together with the bias vectors b and c. Forward propagation needs the following equations (the initial state $h^{(0)}$ needs a certain initialization). Due to another index i, which arises in the following, the denomiation varies slightly from 2.1.3.4, meaning that the time t is now superscript.
The weighted input $z^{(t)}$ to the hidden neuron $h^{(t)}$ is given by

$$z^{(t)} = Ux^{(t)} + Wh^{(t-1)} + b. \tag{2.76}$$

Section 2.1.3.4 already stated out that RNNs use the input x multiplied by matrix U and additionally the previous state $h^{(t-1)}$ weighted with W. The outcome of the equation above is then fed into the activation function of the hidden neuron (here tanh activation function)

$$h^{(t)} = tanh\left(z^{(t)}\right). \tag{2.77}$$

Finally, the output of the RNN is computed with

$$o^{(t)} = Vh^{(t)} + c \tag{2.78}$$

where t ranges from 1 to τ. Let the loss function or, alternatively, cost function be denoted by $L^{(t)}$ for each time step t. The total loss L, in other words the sum of all $L^{(t)}$, is given by

$$L\left(\{x^{(1)}, \ldots, x^{(\tau)}\}, \{\hat{y^{(1)}}, \ldots, \hat{y^{(\tau)}}\}\right) = \sum_t L^{(t)} \tag{2.79}$$

where x is the input sequence and $\hat{y_i}$ are corresponding target values for each $x_i \in x$. It can be recognized from Equation 2.79 that all previous time steps need to be taken into account to compute the gradient. This makes the BPTT algorithm performing with a runtime of $O(\tau)$, since the unrolled graph needs to be considered.
The algorithm needs to compute the gradient for each node recursively. Goodfellow *et al.* [6] assume that the output of the last layer $o^{(t)}$ is used as input for a softmax function (see Equation 2.9) to get a vector $\hat{y}$ of probabilities over the output. Thus, at time step t the gradient $\nabla_{o^{(t)}} L$ of the output is given by

$$(\nabla_{o^{(t)}} L)_i = \frac{\partial L}{\partial o_i^{(t)}} = \frac{\partial L}{\partial L^{(t)}} \frac{\partial L^{(t)}}{\partial o_i^{(t)}} \tag{2.80}$$

for all i. The gradient at the final time step τ is

$$\nabla_{h^{(t)}} L = V^T \nabla_{o^{(t)}} L \tag{2.81}$$

since only the hidden state $h^{(\tau)}$ succeeds the output $o^{(\tau)}$. Then, the BPTT algorithm starts. It iterates backward from $t = \tau - 1$ down to $t = 1$. Considering that each $h^{(t)}$ with $t \neq \tau$ has two descendents, $o^{(t)}$ and $h^{(t+1)}$, the gradient with respect to the hidden unit h^t is given by

$$\nabla_{h^{(t)}} L = \left(\frac{\partial h^{(t+1)}}{\partial h^{(t)}}\right)^T (\nabla_{h^{(t+1)}} L) + \left(\frac{\partial o^{(t)}}{\partial h^{(t)}}\right)^T (\nabla_{o^{(t)}} L) \tag{2.82}$$

$$= W^T (\nabla_{h^{(t+1)}} L)\, diag\left(1 - \left(h^{(t+1)}\right)^2\right) + V^T (\nabla_{o^{(t)}} L) \tag{2.83}$$

where $diag\left(1 - \left(h^{(t+1)}\right)^2\right)$ denotes a diagonal matrix whose elements are $1 - \left(h^{(t+1)}\right)^2$. This is in turn the Jacobian matrix of the activation function (here tanh) of hidden unit i at time step $t + 1$.
The gradients of the other parameters are calculated using the following equations

$$\nabla_c L = \sum_t \left(\frac{\partial o^t}{\partial c}\right)^T \nabla_{o^t} L = \sum_t \nabla_{o^t} L \tag{2.84}$$

$$\nabla_b L = \sum_t \left(\frac{\partial h^t}{\partial b^t}\right)^T \nabla_{h^t} L = \sum_t diag\left(1 - \left(h^{(t)}\right)^2\right) \nabla_{h^t} L \tag{2.85}$$

$$\nabla_V L = \sum_t \sum_i \left(\frac{\partial L}{\partial o_i^t}\right) \nabla_V o_i^{(t)} = \sum_t \left(\nabla_{o^{(t)}} L\right) h^{(t)^T} \tag{2.86}$$

$$\nabla_W L = \sum_t \sum_i \left(\frac{\partial L}{\partial h_i^t}\right) \nabla_{W^{(t)}} h_i^{(t)} = \sum_t diag\left(1 - \left(h^{(t)}\right)^2\right) \left(\nabla_{h^t} L\right) h^{(t-1)^T} \tag{2.87}$$

$$\nabla_U L = \sum_t \sum_i \left(\frac{\partial L}{\partial h_i^t}\right) \nabla_{U^{(t)}} h_i^{(t)} = \sum_t diag\left(1 - \left(h^{(t)}\right)^2\right) \left(\nabla_{h^t} L\right) x^{(t)^T}. \tag{2.88}$$

Another explanation regarding BP and BPTT is given in [46].

2.1.3.9 Problem with Unstable Gradients

As already mentioned several times, gradients might explode or vanish. The latter occurs, for instance, if the weights are initialized randomly and choosen either very high or very low and hence the activation functions (sigmoid or tanh) saturate. This issue is counteracted with the methods explained in Section 2.1.3.14. The vanishing gradient problem is mentioned by Hochreiter *et al.* in [39]. Although this is mostly about the gradient problem in RNNs, the essential part is the same as for other neural networks.

2.1.3.10 Learning Rate η

The *learning rate* η measures *how fast* an algorithm learns. It should be choosen in a decreasing way. In early training steps it is good to apply a high learning rate to make the weights change quickly. This is due to the reason that the weights and biases were recently initialized. As learning progresses the learning rate should be more and more decreased in order to perform more fine-tuning steps. That is to only make little adjustments to the weights of a neural network [5]. This circumstance is called *learning rate decay*.

2.1.3.11 Hyperparameters and Cross-Validation

One important aspect to keep in mind when setting up a neural network is that the right hyperparameters are not known from beginning. This is why the hyperparameters need to be adjusted while training. Hyperparameters denote, for instance, a learning rate η, a momentum value ϕ, a regularization parameter α, a mini-batch size m, a number of training iterations, a number of hidden neurons and so forth but not the weights and biases. Mostly, various values have to be tried in order to get the best network performance.

However, there are some methods which help tuning the hyperparameters. Some are introduced in the corresponding chapters, *learning rate decay* affecting the learning rate in Section 2.1.3.10, *early stopping* helping to set the number of training iterations in Section 2.1.3.13 and *initialization of parameters* in Section 2.1.3.14 giving fasciliations on how to initialize the weights and biases in the beginning. As weights and biases are no hyperparameters it might not fit into this section but initializing these parameters is an important step to efficient training. Other techniques and hyperparameters can be derived from [47].
For choosing the right hyperparameters, a *validation set* is required. If only a training and test set is used, the model tends to overfit, since the parameters always choose the maximal model capacity (see Section 2.1.3.12) [6]. This is where the validation set is applied. It contains examples which are neither in the training set nor in the test set. The validation set is constructed using the training data. This is done by splitting it up into two disjoint subsets, typically 80 % of this data for training and 20 % for validation [6].

Cross-Validation—If the dataset we need for training/validation or training/testing is too small, *cross-validation* can be applied. The most common form of it is *k-fold* cross-validation [6]. The dataset $\mathbb{D}$ is thereby partitioned into k disjount subsets $\mathbb{D}_i$. Then, k trials get performed. On trial j the subset $\mathbb{D}_j$ is used for validation/testing and the remaining ones $\{\mathbb{D}_0, \ldots, \mathbb{D}_{k-1}\} \backslash \{\mathbb{D}_j\}$ are used for training. After each run, an error e_j is computed. This procedure is repeated k times. After that, the validation set error/generalization error e is computed by taking the mean among all errors e_i, $i \in \{0, \ldots, k-1\}$ [6].

2.1.3.12 Generalization, Overfitting and Underfitting

Learning algorithms need to generalize well on new situations, on which they are not trained on. This is why measures and regulations are required in order to assure this. Below, *generalization* of neural networks is described. As known from Section 2.1.3.11, a data set is divided up into three parts: a training set, a validation set and a test set. The first one is used for learning, that is to adjust the weights reasonable. The validation set contains less examples as the former set used for training and is applied to tune the hyperparameters of a neural network such as the number of hidden neurons or the learning rate. If all parameters and hyperparameters are adjusted well the latter set is used to evaluate the performance, meaning the generalization of the trained neural network. This leads to following two terms, *underfitting* and *overfitting*. Underfitting results in a network that does not model the training data well and hence can not generalize well to new data. On the other hand, overfitting occurs if a network learns the training data too well, meaning that it adapts all the details of this set as well as the noise. This yields a low generalization. In the example of Figure 2.27, a linear, a quadratic and a degree-9 predictor are applied to solve a quadratic problem. As it can

be seen, algorithms that tend to underfit are not able to find an appropriate fit for the model (here the linear function). On the contrary, overfitting algorithms model the problem too accurate although it captures all points (here the degree-9 polynomial). Only the quadratic function generalizes well to the quadratic problem.
All in all, underfitting and overfitting of a neural network can be regulated by changing its *capacity*, meaning the ability to model a large variety of functions. If a neural network suffers from underfitting (low capacity) the number of hidden units should be increased, as it is not able to fit the underlying complex structure of the model appropriately. This can be seen, for instance, when the training error is still large after training. How to prevent a neural network from overfitting (high capacity) is explained in Section 2.1.3.13.

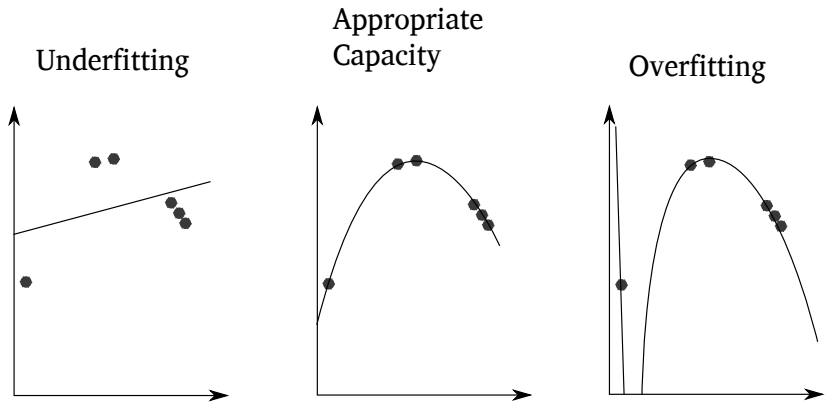


Figure 2.27: An illustration of underfitting on the left and overfitting on the right side. The optimal capacity is shown in the middle (derived from [6]).

2.1.3.13 Regularization

Regularization is used to prevent a neural network from overfitting by mostly limiting the capacity of it [6]. This is done by adding an additional term, called *parameter norm penalty* $\Omega(\theta)$, to the objective function J (a function which needs to get optimized, e.g. a cost function) which is given by

$$\tilde{J}(\theta; X; y) = J(\theta; X; y) + \alpha\Omega(\theta) \tag{2.89}$$

where the hyperparameter $\alpha \in [0, \infty)$ measures the contribution of the parameter norm penalty $\Omega(\theta)$ to the objective function J. A small α ensures that the original objective function J is taken into account and thus gets minimized. In contrast, a high α prefers small weights. Goodfellow *et al.* [6] suggest to choose a parameter norm penalty that only affects the weights and leaves the biases unregularized, as this can lead to underfitting.
Different regularization techniques are described afterwards. First, L^2 *parameter regularization* also known as *weight decay* is introduced. Here, the regularization term

corresponds to $\Omega(\theta) = \frac{1}{2}||w||_2^2$, where $||w||_2$ represents the $L^2 norm$ of w, defined by $\sqrt{\sum_{i=1}^{n} |w_i|^2}$. The effect of weight decay can be seen in the following equations [6]. Consider the regularized objective function

$$\tilde{J}(w; X; y) = \frac{\alpha}{2} w^T w + J(w; X; y) \tag{2.90}$$

where θ is just w, since the biases are left out for simplification. The term $w^T w$ belongs to $||w||_2^2$, multiplying the weights vector with itself. The corresponding gradient of the objective function with respect to w is given by

$$\nabla_w \tilde{J}(w; X; y) = \alpha w + \nabla_w J(w; X; y). \tag{2.91}$$

A single step to update the weights, for instance with GD, is accomplished with

$$w \longleftarrow w - \eta(\alpha w + \nabla_w J(w; X; y)). \tag{2.92}$$

Equation 2.92 can be rewritten into

$$w \longleftarrow (1 - \eta\alpha) w - \eta \nabla_w J(w; X; y). \tag{2.93}$$

From Equation 2.93 it can be recognized that each update step involves an additional factor $(1 - \eta\alpha)$, which shrinks the weight vector by a constant factor before performing the usual update.
Another regularization technique is $L^1 regularization$. The regularization term conforms thereby to $\Omega(\theta) = ||w||_1 = \sum_i |w_i|$. Therefore, the regularized cost function is given by

$$\tilde{J}(w; X; y) = \alpha ||w||_1 + J(w; X; y). \tag{2.94}$$

The corresponding gradient function is determined by

$$\nabla_w \tilde{J}(w; X; y) = \alpha sign(w) + \nabla_w J(w; X; y) \tag{2.95}$$

where $sign(w)$ is the sign of w applied element-wise [6]. A single gradient update step is performed with

$$w \longleftarrow (1 - \eta\alpha) sign(w) - \eta \nabla_w J(w; X; y). \tag{2.96}$$

The regularization contribution to the gradient is now a constant with sign *sign(w)*. The effect of L^1 regularization on the weights is completely different from L^2 regularization. Whereas the latter one focuses on shrinking the weights with each step, the first one concentrates on important weights. That is why L^1 Regularization decreases weights much less if they are large than it decreases them if they are small. Hence, this regularization type results in a solution that is more *sparse*, meaning that some parameters values are driven towards zero [5].

The third type of regularization which is explained in this section is called *dropout* [48]. This method is quite different from the both explained above as it does not add a parameter norm penalty to the objective function. It rather modifies the neural network itself. The principle behind dropout is as follows. For every training example, half of the activation functions of the hidden neurons are primarily set to zero, meaning to virtually delete half of the hidden neurons. These neurons are chosen randomly. After that, forward and backward propagation takes place and the weights and biases are updated appropriately. Finally, the virtually deleted neurons are restored and another subset is chosen randomly. The procedure is repeated for the next training example afterwards. Thus, the neural network can not be sure that a certain activation function and a hidden neuron, respectively, is present. That is why the network needs to learn a redundant representation of the training examples. Training a neural network with dropout is like training different networks at the same time. Then, the neural network takes the consensus of the collectivity of all networks. A type of activation function which works efficiently together with dropout is maxout (see Section 2.1.1.1) [2].
The last method introduced here is called *early stopping*. It differs from the regularization techniques in a way that it does not apply artificial constraints to the neural network. Strictly speaking, early stopping is not a regularization method. Nevertheless, it prevents a neural network from overfitting. The concept of early stopping is best illustrated using Figure 2.28. If the validation performance reaches its peak, the training procedure stops.

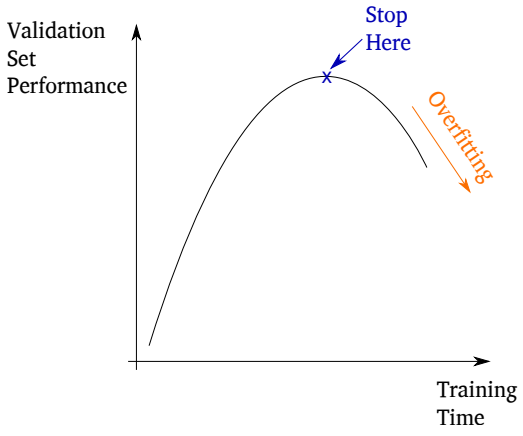


Figure 2.28: The principle of early stopping (derived from [7]).

2.1.3.14 Initialization of parameters

The initialization of all parameters of a neural network must not be disregarded and is an important aspect of training. The choice of the parameters can affect both the speed by which learning converges and the generalization of comparable costs. Furthermore, a reasonable initialization is also necessary to *break symmetry* [6]. That means that if

two hidden units with the same inputs and the same activation functions have the same parameters, they are always the same and thus, redundant. This leads to the assumption that the parameters of a neural network should be initialized randomly. According to [6] unlike weights, biases need to be set to constants which are heuristically chosen. The following aspects indicate that larger weights have to be chosen. First, they result in a stronger symmetry-break effect and hence avoid redundant units. Furthermore, larger weights prevent multiplicated matrices of getting to small. Consequently, they help to not lose signal by forward and backward propagation through the linear component of each layer [6]. This linear component is mostly the weighted input without involving the activation function.
Too large weights, however, yield exploding values and gradients during forward and backward propagation, respectively. Additionally, they result in an extreme sensibility to small changes in the input of RNNs and hence the behaviour of the forward propagation step seems to be random [6].
Large weights can also cause the neurons, more specifically their activation functions, to saturate because of extreme values (see Section 2.1.1.1). This results in a loss of the gradient.
In summary, it can be stated that it is important to find a right initialization regarding all the competing factors mentioned above. Hence, there needs to be a compromise between large weights favouring the successful propagation of information through the network and smaller weights facilitating regularization.
Below, both a very common technique and a specific heuristic for weight initialization are described. For one thing, the weights can be uniformily drawn from an interval $U\left(-\sqrt{\frac{1}{m}}, \sqrt{\frac{1}{m}}\right)$, where m is the number of inputs to a fully-connected layer. Its outputs are denoted with n. Another initialization technique established by Glorot and Bengio [49] sets the weights using a *normalized initialization*

$$W_{i,j} \sim U\left(-\sqrt{\frac{6}{m+n}}, \sqrt{\frac{6}{m+n}}\right). \tag{2.97}$$

Most weight initialization heuristics aim at choosing the weights with mean 0 and standard deviation $\frac{1}{\sqrt{m}}$. However, there comes one problem along with initializing all weights to the same standard deviation. They become extremly small if the network layers become larger. All in all, a reasonable weight initialization is necessary to prevent a neural network from both slowing down learning and saturation.

2.1.3.15 Batch Normalization

Batch normalization was introduced by Ioffe and Szegedy in 2015 [50]. It helps to avoid lower learning rates and hence speeds up training. They address a problem

called *internal covariate shift*. This means that the input distribution of internal nodes changes every training step [51]. This is due to the reason that weights and parameters are modified every training step and thus, the data is changed. Simply put, batch normalization is an additional step between two layers. The output of layer $n-1$ gets batch normalized before it becomes the input to layer n, i.e. the normalized value $\hat{x}$ after layer n becomes the input to a sub-network which applies a linear transformation to $\hat{x}$ by using Equation 2.101 [50]. This output becomes the input to the regular layer $n-1$. The batch normalization algorithm works the following way [50]. First, both the mean μ and the variance σ^2 of the mini-batch x with elements $x_i, i = \{1, \ldots, m\}$, have to be computed

$$\mu = \frac{1}{m} \sum_{i=1}^{m} x_i \tag{2.98}$$

$$\sigma^2 = \frac{1}{m} \sum_{i=1}^{m} (x_i - \mu)^2 . \tag{2.99}$$

The normalization of every element x_i is then given by

$$\hat{x}_i = \frac{x_i - \mu}{\sqrt{(\sigma^2 + \epsilon)}} \tag{2.100}$$

where ϵ is an additonal parameter to avoid a division through zero. $\hat{x}$ is afterwards scaled by γ and shifted by β using

$$y = \gamma \hat{x}_i + \beta \tag{2.101}$$

where γ as well as β indicate parameters which need to be learned.
Due to two more parameters (γ, β) the BP algorithm needs to be adjusted. The further learning procedure of batch normalized neural networks is shown in detail in [50] and [51].

2.1.4 Application Scenarios

Deep learning is ambitious of detecting highly hierachical features in data sets (see Section 2.1) [19]. That is why it is used in computer vision tasks such as detecting faces and recognizing objects in an image. Further applications are natural language processing, the use of human speech by a computer, and speech recognition, the mapping of human speech to the intended words. Additionally, in our approach it is applied to smart spaces. All in all, deep learning can be applied to a large variety of use cases. It will help artificial intelligence to be spread among various areas like driverless cars [52], TV programm recommendations [53] or smart homes [54]. The latter one applies deep learning for energy saving, security issues, health care and home care (see Table I in [54] and Tables 2.1, 2.2 in Section 2.5).

2.1.5 Machine Learning & Deep Learning Frameworks

Different machine learning and deep learning frameworks, respectively, are available. In the following, four of them are introduced. Three of them are using Python and one is using Java. *Sonnet* and *Caffe2* were released recently in April 2017. All of them have pre-trained models and there a several tutorials for each library.

2.1.5.1 Theano

Theano[1] is a Python library. Below, a Theano example[2] is shown. It shows the computation steps for a XOR neural network.

Listing 2.1: Theano example

```
#A code snippet of a simple network consisting of
#2 input units, 2 hidden units, 1 output unit

import theano
import theano.tensor as T
import theano.tensor.nnet as nnet
import numpy as np


x = T.dvector()
y = T.dscalar()

#define a layer with input x, bias b=1, weight matrix w, sigmoid unit
def layer(x, w):
   b = np.array([1], dtype=theano.config.floatX)
   new_x = T.concatenate([x, b])
   m = T.dot(w.T, new_x) #theta1: 3x3 * x: 3x1 = 3x1 ;;; theta2: 1x4 * 4x1
   h = nnet.sigmoid(m)
   return h

#gradient descent: T.grad() computes the gradient
def grad_desc(cost, theta):
   alpha = 0.1 #learning rate
   return theta - (alpha * T.grad(cost, wrt=theta))

#define and initalize the weight matrices randomly
#'shared' variable as we want to update it
theta1 = theano.shared(np.array(np.random.rand(3,3),
    dtype=theano.config.floatX))
```

[1] http://deeplearning.net/software/theano/
[2] http://outlace.com/Beginner-Tutorial-Theano/

```
theta2 = theano.shared(np.array(np.random.rand(4,1),
    dtype=theano.config.floatX))


hid1 = layer(x, theta1) #hidden layer

out1 = T.sum(layer(hid1, theta2)) #output layer
fc = (out1 - y)**2 #cost expression

cost = theano.function(inputs=[x, y], outputs=fc, updates=[
      (theta1, grad_desc(fc, theta1)),
      (theta2, grad_desc(fc, theta2))])
run_forward = theano.function(inputs=[x], outputs=out1)

inputs = np.array([[0,1],[1,0],[1,1],[0,0]]).reshape(4,2) #training data X
exp_y = np.array([1, 1, 0, 0]) #training labels Y
cur_cost = 0
for i in range(10000):
   for k in range(len(inputs)):
      cur_cost = cost(inputs[k], exp_y[k]) #call our Theano-compiled cost
          function, it will auto update weights
```

2.1.5.2 TensorFlow & Sonnet

Sonnet[3] builds on top of *TensorFlow*[4] and is based on Python. A TensorFlow example[5] on the MNIST dataset of handwritten digits is given below [16].

Listing 2.2: Tensorflow example

```
#A code snippet of a single layer network:
#The input is fed through a softmax-output-layer
from tensorflow.examples.tutorials.mnist import input_data
mnist = input_data.read_data_sets("MNIST_data/", one_hot=True)

import tensorflow as tf

x = tf.placeholder(tf.float32, [None, 784])
W = tf.Variable(tf.zeros([784,10]))
b = tf.Variable(tf.zeros([10]))

#y contains the actual output
y = tf.nn.softmax(tf.matmul(x, W) + b)
```

[3] https://github.com/deepmind/sonnet
[4] https://www.tensorflow.org/
[5] https://www.tensorflow.org/get_started/mnist/beginners

```
#y_ contains the correct answers (1,...,10)
y_ = tf.placeholder(tf.float32, [None, 10])

#loss function: cross-entropy
cross_entropy = tf.reduce_mean(cross_entropy =
    tf.reduce_mean(tf.nn.softmax_cross_entropy_with_logits(labels=y_,
    logits=y)))

train_step = tf.train.GradientDescentOptimizer(0.5).minimize(cross_entropy)

#operation to initialize the variables created
# outdated: init = tf.initialize_all_variables()
init = tf.global_variables_initializer()

#launch the model in a Session
sess = tf.Session()
sess.run(init)

#Let's train -> perform the training step 1000 times
for i in range(1000):
   batch_xs, batch_ys = mnist.train.next_batch(100)
   sess.run(train_step, feed_dict={x: batch_xs, y_: batch_ys})
```

Our learning algorithms are implemented with TensorFlow (see Chapter 5). This is due to the reason that a variety of applications use this framework. Moreover, a large community provides additional information and we are able to apply parametrization efficiently. Furthermore, it provides a good performance and a well-structured architecture [55].
A detailed evaluation of different machine learning and deep learning frameworks is given in [56] and [57].

2.1.5.3 Caffe & Caffe2

Caffe2[6] builds on *Caffe*[7] and is Python-based. Caffe differs from the other libraries explained here in that it stores the definiton of the network in a protobuf-structure[8]. The data is organized in blobs (chunk of data in memory) and workspaces store all the blobs. The example[9] below is implemented in Caffe2. The first *def*-part defines the model with its convolutional layers and a softmax output layer. The second one adds training parameters to the model.

[6] https://github.com/caffe2/caffe2
[7] http://caffe.berkeleyvision.org/
[8] https://developers.google.com/protocol-buffers/docs/overview
[9] https://github.com/caffe2/caffe2/blob/master/caffe2/python/tutorials/MNIST.ipynb

Listing 2.3: Caffe2 example

```
def AddLeNetModel(model, data):
    """Adds the main LeNet model.

    This part is the standard LeNet model: from data to the softmax prediction.

    For each convolutional layer we specify dim_in - number of input channels
    and dim_out - number or output channels. Also each Conv and MaxPool layer
        change
    image size. For example, kernel of size 5 reduces each side of an image by
        4.

    While when we have kernel and stride sizes equal 2 in a MaxPool layer, it
        devides each side in half.
    """
    # Image size: 28 x 28 -> 24 x 24
    conv1 = model.Conv(data, 'conv1', dim_in=1, dim_out=20, kernel=5)
    # Image size: 24 x 24 -> 12 x 12
    pool1 = model.MaxPool(conv1, 'pool1', kernel=2, stride=2)
    # Image size: 12 x 12 -> 8 x 8
    conv2 = model.Conv(pool1, 'conv2', dim_in=20, dim_out=50, kernel=5)
    # Image size: 8 x 8 -> 4 x 4
    pool2 = model.MaxPool(conv2, 'pool2', kernel=2, stride=2)
    # 50 * 4 * 4 stands for dim_out from previous layer multiplied by the image
        size
    fc3 = model.FC(pool2, 'fc3', dim_in=50 * 4 * 4, dim_out=500)
    fc3 = model.Relu(fc3, fc3)
    pred = model.FC(fc3, 'pred', 500, 10)
    softmax = model.Softmax(pred, 'softmax')
    return softmax


def AddTrainingOperators(model, softmax, label):
    """Adds training operators to the model."""
    xent = model.LabelCrossEntropy([softmax, label], 'xent')
    # compute the expected loss
    loss = model.AveragedLoss(xent, "loss")
    # track the accuracy of the model
    AddAccuracy(model, softmax, label)
    # use the average loss we just computed to add gradient operators to the
        model
    model.AddGradientOperators([loss])
    # do a simple stochastic gradient descent
    ITER = model.Iter("iter")
    # set the learning rate schedule
    LR = model.LearningRate(
       ITER, "LR", base_lr=-0.1, policy="step", stepsize=1, gamma=0.999 )
```

```
   # ONE is a constant value that is used in the gradient update. We only need
   # to create it once, so it is explicitly placed in param_init_net.
   ONE = model.param_init_net.ConstantFill([], "ONE", shape=[1], value=1.0)
   # Now, for each parameter, we do the gradient updates.
   for param in model.params:
      # Note how we get the gradient of each parameter - CNNModelHelper keeps
      # track of that.
      param_grad = model.param_to_grad[param]
      # The update is a simple weighted sum: param = param + param_grad * LR
      model.WeightedSum([param, ONE, param_grad, LR], param)
   # let's checkpoint every 20 iterations, which should probably be fine.
   # you may need to delete tutorial_files/tutorial-mnist to re-run the
       tutorial
   model.Checkpoint([ITER] + model.params, [],
               db="mnist_lenet_checkpoint_%05d.leveldb",
               db_type="leveldb", every=20)
```

2.1.5.4 DeepLearning4J

DeepLearning4J[10] is a library based on Java. Below, an example[11] of a FFNN with one hidden layer is given. It is trained on the MNIST dataset of handwritten digits [16].

Listing 2.4: DeepLearning4J example

```
//A code snippet of building and training a simple FFNN in DL4J
final int numRows = 28;
final int numColumns = 28;
int outputNum = 10; // number of output classes
int batchSize = 128; // batch size for each epoch
int rngSeed = 123; // random number seed for reproducibility
int numEpochs = 15; // number of epochs to perform

//Get the DataSetIterator:
DataSetIterator mnistTrain = new MnistDataSetIterator(batchSize, true,
    rngSeed);

MultiLayerConfiguration conf = new NeuralNetConfiguration.Builder()
            .seed(rngSeed) //include a random seed for reproducibility
            // use stochastic gradient descent as an optimization algorithm
            .optimizationAlgo(OptimizationAlgorithm.STOCHASTIC_GRADIENT_DESCENT)
            .iterations(1)
            .learningRate(0.006) //specify the learning rate
```

[10] https://deeplearning4j.org/

[11] https://deeplearning4j.org/mnist-for-beginners

```
            .updater(Updater.NESTEROVS).momentum(0.9) //specify the rate of
                change of the learning rate.
            .regularization(true).l2(1e-4)
            .list()
            .layer(0, new DenseLayer.Builder() //create the first, input layer
                with xavier initialization
                  .nIn(numRows * numColumns)
                  .nOut(1000)
                  .activation(Activation.RELU)
                  .weightInit(WeightInit.XAVIER)
                  .build())
            .layer(1, new
                OutputLayer.Builder(LossFunction.NEGATIVELOGLIKELIHOOD)
                //create hidden layer
                  .nIn(1000)
                  .nOut(outputNum)
                  .activation(Activation.SOFTMAX)
                  .weightInit(WeightInit.XAVIER)
                  .build())
            .pretrain(false).backprop(true) //use backpropagation to adjust
                weights
            .build();

MultiLayerNetwork model = new MultiLayerNetwork(conf);
model.init();

for( int i=0; i<numEpochs; i++ ){
  model.fit(mnistTrain);
}
```

2.2 Smart Space Orchestration with VSL

Smart Space Orchestration [58] faces some difficulties nowadays, as the used entities (e.g. temperature sensor, light sensor, smart TV) are usually heterogenous. For instance, mostly only entities with the same functionality (e.g. heating entities) can communicate with each other. To overcome this problem of heterogeneity, the *Distributed Smart Space Orchestration System (DS2OS)* was developed [59]. It is a middelware framework consisting of a *Virtual State Layer (VSL)* middleware (see Figure 2.29), a *Service Management Layer (SML)* and a *Smart Spaces Store (S2S)*. Among these, VSL is the most important one for our approach.

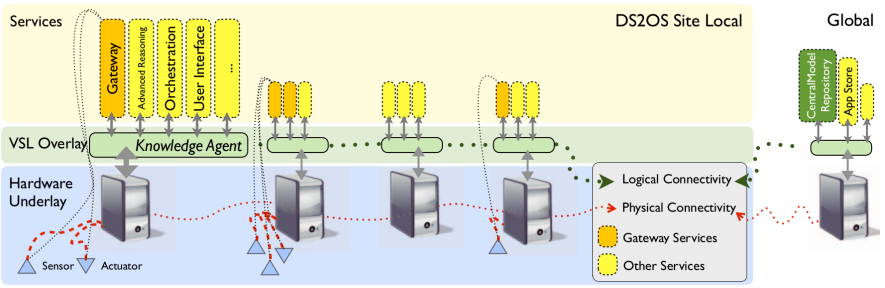


Figure 2.29: The VSL [13].

Therefore, VSL is explained in more detail now. VSL is a middleware based on a peer-to-peer system, meaning all nodes in the network are equally priviledged and can communicate with each other, no matter if their underlying software or hardware is heterogenous. It implements a so-called *blackboard communication pattern* which results in a loose coupling of the services. *Loose coupling* means that we communicate only with the interfaces of the services, which are also working asynchronously. This allows *full encapsulation* of the services, since the communication is done with the help of the interfaces (context models) [60]. This is where data models (see Section 2.2.1) come into focus, as structured data is required.
The principle of blackboard communication is as follows [60]: A service can write data produced by itself on the board or it can read data from the board, consuming it. Thus, each service needs its functionality autonomously, for instance, for reading sensor values or changing the environmental state with its actuators.

2.2.1 Context Models

To define *context model*, we illustrate the term *context* first. While a service is on-line, it gathers information to accomplish its purpose. The appropriate information is thereby called context. Context models are used for structuring such context information and hence represent structured data about the real world. That means that context models

express a *virtual state* of the real world. The context is structured using a simple XML representation [58]. In DS2OS, VSL stores the context for and brokers between the services [60]. If a service is registered, a context model is initialized. In this context model, all context data the particular service produces is stored. Thus, the context model of each service becomes its abstract interface. A service can also access context from other services. Context can be changed using the VSL interface with the commands *GET* and *SET* [60].

Context Model Repository—A global *Context Model Repository (CMR)* stores all context models which can be used in all VSL Smart Spaces. The context models are identified by means of a unique *model identifiers* (ModelID) [60].

2.2.2 Knowledge Graph

As mentioned above, the VSL is based on a peer-to-peer system (see Section 2.2). The peers in the VSL are called *Knowledge Agents (KAs)*. As it can be seen in Figure 2.30, a KA offers principally two functionalities, *context management* and *context repository*. The interface of a KA provides following methods. Values of a context node are returned or changed with the methods *get* or *set*. To get notified, when a value of a node changes, you can *subscribe* to that node. On the other hand, you can *unsubscribe* to remove the subscription. The last method refers to the *registration of virtual nodes*. If a node is registered as virtual, all queries to this node will not be handled by the VSL. This task is taken over by the virtual node handler, specified for that node. You can also unregister a virtual node to enable VSL handling again [60].

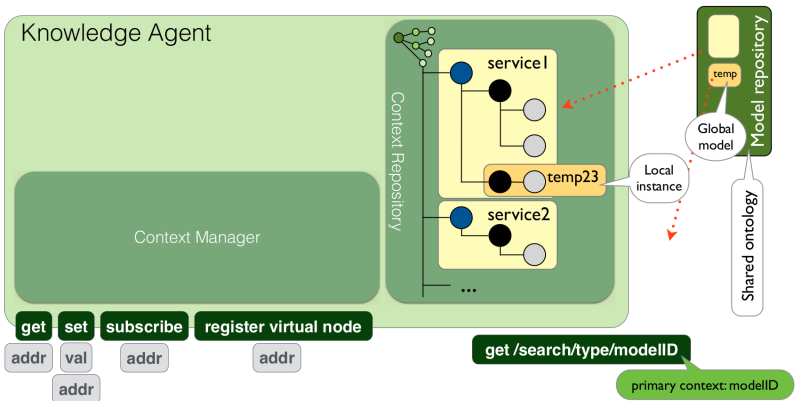


Figure 2.30: Context management in DS2OS [13].

Furthermore, it can be seen in Figure 2.29, that each computing node equipped with sensors and actuators runs its own instance of a KA. All KAs connected to a root

node build up a hierarchy, a so-called *knowledge graph*, containing services and their parameter values. This circumstance is shown in Figure 2.31. Before a service can use the methods provided by the VSL through the KAs it needs to register itself to a particular KA [60].

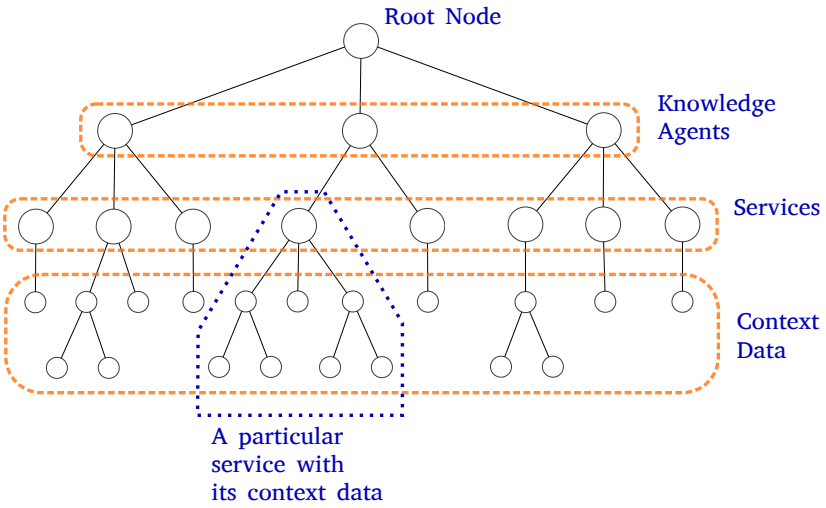


Figure 2.31: A knowledge graph constructed with three KAs, each one connected to the root node. Each KA has its own services.

2.2.3 Knowledge Structuring

The context data of a smart space is structured using context models. Each service connected to its KA processes its own data. As shown in Figure 2.31 each service has its own number of nodes to store context in. These nodes are determined by its context model. For example, a weather service stores context data like outdoor and indoor temperature, humidity and rain probability.

2.2.4 Knowledge Vectors

The virtual state can not only be represented via a knowledge graph but also by a *knowledge vector* or *state vector*. This possibility is exploited in our deep learning approach, as neural networks use vectors as input (see Section 2.1). Knowledge vectors contain the same data as context models, but are less structred. This is due to the flat structure of a vector in contrast to the tree-like structure of a knowledge graph (see Section 2.2.2). These vectors look like the following [60]

$$\overbrace{(\overbrace{< s1, s2, s3, \ldots, sN >}^{Service}, \overbrace{< a1, a2, a3, \ldots, aN >}^{Process}, < b1, \ldots, bl >, \ldots)}^{EntireSpace} . \tag{2.102}$$

The state vector of an entire space, for example a flat, contains both services and processes which go over multiple services. The knowledge vector changes and develops, respectively, over time. This is due to the ongoing collection of sensor data, since data from the outside is taken continously (e.g. temperature) or something from the inside changes (e.g. a reasoning mechanism concludes it is getting night and, thus, changes the setting of a lamp).

2.3 Using Machine Learning and Deep Learning in Smart Spaces

Machine learning and deep learning, respectively, can be applied in various sectors in smart spaces. As smart spaces are built up using sensors, actuators and smart devices we are able to get every information about the particular smart space. This information is provided for the machine learning algorithms. In the case of DS2OS, all the information about the smart space is encapsulated in knowledge vectors (see Section 2.2.4). Some application scenarios in smart spaces are summarized in Tables 2.1, 2.2.
As learning algorithms are based on vectors and matrices, they use a vector notation as input. Hence, we exploit the knowledge vector representation described above in Section 2.2.4 and Equation 2.102. However, the vectors must have the same length. This is due to the fixed size of input units in neural networks (see Section 2.1). Due to this constraint, we need to either pad the vectors with zeroes or resize them in a different way. Another important aspect to keep in mind is that the vectors used as input have to have the same structure. As we already know, state vectors develop over time. Therefore, they should not lose their inner structure, meaning for example, all services get listed before the processes are. This is due to the reason that after training, neural networks have adjusted their weights according to their training input.
To apply a learning algorithm it is sufficient to have a large data set. This data set is then split up into a training set, a validation set and a test set. Thus, before a machine learning algorithm can be applied in smart spaces it needs to be trained using the training set. The trained neural network is then able to predict concrete results using an abstract input, e.g. a new knowledge vector representing the current state of the smart space.
It is almost impossible to find related works (see Chapter 3) about such rather new approaches like deep learning applications in smart spaces. On the other hand, there exists a number of different approaches which apply machine learning classifier and other simple neural networks to smart spaces. Tables 2.1, 2.2 in Section 2.5 list such approaches.
It is difficult to implement a machine learning algorithm without detailed knowledge in both machine learning and the corresponding machine learning library. Hence, we develop an approach which makes machine learning available for users, respectively developers, with little or almost no pre-knowledge in these areas. Therefore, we attach importance to usability and reusability. This approach aims at modularizing machine learning algorithms. By modifying a created configuration file one can simply build a machine learning algorithm and train it on a particular data set. Depending on the values provided to the configuration file, the user can either train a shallow or a deep neural network architecture. This is why we refer to our services as general *machine learning services*. Furthermore, this approach can be applied to smart spaces.
Chapter 4 describes further details about the design of our implemented machine learning services.

2.4 Summary

In this chapter, we introduced *deep learning* as an area of *machine learning*. We focused thereby on different types of neural networks and provided information about how to make each neural network deeper. Furthermore, the DS2OS was explained. Afterwards, we described how to apply machine learning in smart spaces by exploiting the DS2OS.
We started with explaining four different kinds of artificial neurons (Section 2.1.1.1), beginning with the first announced neuron called *perceptron*, a binary neuron. We added the *sigmoid neuron*, ranging from 0 to 1 and the *tangens hyperbolicus neuron* with a range of -1 to 1. The *Rectified Linear Unit (ReLU)* formed the tail. It is either 0 or the input itself if it is positive. In addition to that, we explained the term *saturation of neurons*. Unlike sigmoid and tanh neurons, ReLUs are unaffected by this phenomenon. Additionally, another kind of activation function called *maxout* was introduced.
Afterwards, we depicted a special kind of output unit called *softmax* (Section 2.1.1.2). This unit describes an output with a probability distribution among several values. It is therefore important for classification tasks. For example, if we are processing an image with a cat on it, the output can look as follows: 0.5 % chicken, 5.5 % dog, 94 % cat (three output units).
Then, four different learning techniques called *supervised learning* (Section 2.1.1.3), *semi-supervised learning* (Section 2.1.1.5), *Reinforcement Learning* (Section 2.1.1.6) and *unsupervised learning* (Section 2.1.1.4) were introduced. We also pointed out the use of labeled training data. The amount decreases from the first to the latter, meaning the first needs solely labeled training data and the latter no labeled training data at all. The third works by getting feedback from the environment, meaning right actions are advantageous and wrong actions get penalized.
As three machine learning classifier are used in the related works, they are examined in Section 2.1.2: *Logistic Regression (LR)*, *Softmax Regression (SR)* and a *Support Vector Machine (SVM).* The first two compute *probabilites* to solve a classification task, whereas the latter uses a *class identity* which is either positive or negative.
Next, several types of *neural networks* were illustrated (Section 2.1.3). We provided thereby additional information about how to make the respective network deep. First, a *Feedforward Neural Network (FFNN)* was described (Section 2.1.3.1). After the explanation of a *Convolutional Neural Network (CNN)* (Section 2.1.3.2) with its distinctive *local receptive fields*, *shared parameters* and *pooling layers*, we introduced *Deep Belief Networks (DBNs)* (Section 2.1.3.3). They can either consist of several stacked *Autoencoders (AEs)* or several stacked *Restricted Boltzmann Machines (RBMs).* When we described the functionality of RBMs, we also introduced *Contrastive Divergence (CD)* as a method to update their parameters (see Figure 2.17). Subsequently, *Recurrent Neural Networks (RNNs)* were depicted (Section 2.1.3.4). Important to keep in mind is that RNNs include previous states into their current output. As this yields *long-term dependencies*, the *Long Short-Term Memorys (LSTMs) network* was presented to overcome this

problem. We finished the different kinds of neural networks with a *deep Q-network* (Section 2.1.3.5). It makes use of reinforcement learning.
As we need to measure an error to train a neural network, the following *cost functions* (Section 2.1.3.6) were introduced: *cross-entropy* (Equation 2.57), *log-likelihood* (Equation 2.58), *Sum of Squared Errors (SSE)* (Equation 2.59), *Mean Squared Error (MSE)* (Equation 2.60) and *Kullback-Leibler (KL) divergence* (Equation 2.61). They compute the error by taking the difference of the actual output and the desired one, in general.
To update the weights and biases of a neural network two *gradient-based learning* techniques were described in Section 2.1.3.7, usual *Gradient Descent (GD)* (Equations 2.62, 2.63) and *Stochastic Gradient Descent (SGD)* (Equations 2.64, 2.65). The latter is an improvement over the first, as it speeds up learning by only using a small number of training inputs called *mini-batch* to estimate the gradient. SGD can be further optimized by applying a *momentum* (Equations 2.66, 2.67). This approach takes previous knowledge about the gradient into account.
The above-mentioned gradient-based learning techniques require a gradient of the cost function. This is why, *Backpropagation (BP)* and *Backpropagation Through Time (BPTT)* were depicted (Section 2.1.3.8). The latter is used in RNNs. It is called BP as the gradient is calculated at the output layer and propagated back through the network.
Thereafter, we illustrated the *problem of unstable gradients* (Section 2.1.3.9), explained the role of the *learning rate* η (Section 2.1.3.10), mentioned some methods for *hyperparameter tuning* and introduced *cross-validation* (Section 2.1.3.11).
The next terms described were *generalization* (Section 2.1.3.12), *underfitting* and *overfitting*. The first means that neural networks need to generalize well to unknown data. The latter two terms are reasons for a non-appropriate generalization. A neural network suffering from underfitting is not able to model the training set well. On the other hand, a network that tends to overfit models the training data too well. Thus, it adapts all details of the training set as well as the included noise.
To overcome the above mentioned problem of overfitting, several *Regularization* techniques were introduced in Section 2.1.3.13. First, L^2 *regularization* and L^1 *regularization* were described. Both techniques add an additional term called *rarameter norm penalty* $\Sigma(\theta)$ to the objective function. Then, we mentioned *dropout* which modifies the neural network itself. This is done by randomly setting half of the activation functions of the hidden units to zero, for each training example. Finally, we illustrated *early stopping* which does not apply any constraints to the network. It states that the neural network should stop training if the validation set performance has reached its peak.
Afterwards, we represented why the *initialization of parameters* is important to cobe with learning slowdown and the generalization problem (Section 2.1.3.14). We suggested two possibilities. First, the weights can be drawn *uniformily* from an interval U with boundaries $-\sqrt{\frac{1}{m}}$ and $\sqrt{\frac{1}{m}}$. The alternative is setting the weights according to a *normalized initialization* with boundaries $-\sqrt{\frac{6}{m+n}}$ and $\sqrt{\frac{6}{m+n}}$. The parameters m

and n denote the number of input units and the number of output units, respectively.
We further introduced *batch normalization* as a technique to speed up learning by avoiding smaller learning rates in Section 2.1.3.15. This is done by normalizing each input to an internal node. Furthermore, the normalized input gets scaled by γ and shifted by β using Equation 2.101. Both describe parameters which need to be learned additionally. The BP algorithm needs to be modified in order to train batch normalized neural networks. We did not cover this step as it is described in detail in [50] and [51].
We concluded the section about deep learning with some *Application Scenarios* (Section 2.1.4).
The second part was about the *Distributed Smart Space Orchestration System (DS2OS)* (Section 2.2). We described the *Virtual State Layer (VSL)* middleware as the core of the DS2OS. It enables simple communication between heterogenous smart devices and services, respectively. By using *context models* the real world is represented in a structured way (Section 2.2.1). Each registered service uses its own context model to store all its data in it. A service runs on a *Knowledge Agent (KA)* (Section 2.2.2). Furthermore, we introduced the possibility to represent the data stored in the context models as *knowledge vectors* (Section 2.2.4). These knowledge vectors allow us to use machine learning and deep learning in smart spaces.
Next, we described how to apply machine learning and deep learning, respectively, to smart spaces. This is, for instance, possible by using the knowledge vectors introduced above. Moreover, we explained the basic structure of our approach meaning to develop machine learning services which are easy-to-use and do not require pre-knowledge in both machine learning and the corresponding machine learning library. Hence, usability and reusability of the services are important aspects. Due to the reason that the user is able to decide on his own whether he wants to apply a shallow or a deep neural network, we refer to our services as *machine learning services* in general.

2.5 Overview over Machine / Deep Learning Approaches in Smart Spaces

Tables 2.1, 2.2 summarize important approaches about deep learning and machine learning in smart spaces, respectively. Although a similar table (Table 3.3) is depicted in Chapter 3, these two outline the approaches in more detail. We revert to them in Chapter 4. Both tables are constructed the same way. The first column holds the name of the work, the second denotes the method(s) applied and the third one represents the accuracies of the methods. Column four contains details about the dataset(s) used. The fifth column gives an overview about the architecture of the methods. The last two columns express remarks about the approaches and the year of the work, respectively. The tables show that the most used neural networks are FFNNs or MLPs, DBNs and RNNs.

Approach	Method	Accuracy	Dataset	Architecture Details	Remarks	Year
Smart Home Systems Design based on ANN [61]	FFNN, RNN	96.97 % & 65.52 %	160 Training, 100 Test	Error: SSE, Gradient Descent, FFNN: SLP, LR: 0.01, Epochs: 117	Accuracy: Sensitifity & Specificity, Data is taken continuously from the user, stored in memory and fed into the RNN after a day, thus it generates a profile for the particular person, FFNN independent from RNN: rule based approach, Only trained for temperature and humidity	2011
Optimization of the Use of Residential Lighting with Neural Network [62]	FFNN	Average error: 0.0303, User Satisfaction: 88.42 %	Gathered over 11 weeks, 295680 data (80 % training, 20 % validation)	Error: MSE, Epochs: 281, HL1: 30 tanh, HL2: 20 tanh, Out: linear	One neural network for 5 lightings not possible due to memory issues → one network for each lighting	2010
Smart Home Design for Disabled People based on Neural Networks [63]	FFNN	95 %	75 % training, 25 % test (63/23)	Perceptrons, In: 5, Out: 1	Virtual case study on a disabled person	2014
	RNN	80 %	Training set: 1 week, test set: sample of 1 day (65/21)	Sigmoid, In:3, Out: 2	Set of 2 or 3 actions as input, predicts 2 actions → RNN interacts with the user: asks him which action of the predicted 2 he prefers	
A Novel One-Pass Neural Network Approach for Activities Recognition in Intelligent Environments [64]	OPNN	92 %	200 Training, 100 Test, conflicted data deleted	1 hidden layer, weights of In → HL learned, weights of HL → Out fixed to 1	Data simulated a user bedroom, Sensor state vector is arrayed by neighborhood and function methods, Abnormal behaviour prediction by adding a decision layer (if . . . elseif . . .)	2008
Recognizing Human Activity in Smart Home using Deep Learning algorithm [33]	DBN built by RBMs	Mean 86.8876 %	Gathered during more than 50 days, manually labeled, careful timing of activites	In: data, HL1,2,3: 500,300,100, Out: 10	Accuracy was computed by taking the mean over the accuracies for each activity, Each output represents 1 out of 10 activities, Pre-Training + Fine-Tuning steps	2014

Table 2.1: Overview over machine / deep learning approaches in smart spaces *(1/2)*.

Approach	Method	Accuracy	Dataset	Architecture Details	Remarks	Year
Reinforcement Learning aided Smart Home Decision-making in an Interactive Smart Grid [65]	Q-Learning	Better performance than greedy and random algorithms	/	LR: 0.3, Training Period: 10^3 steps, ϵ: 0.1	No accuracy evaluation and no data set information given	2014
Human Behaviour Prediction for Smart Homes Using Deep Learning [34]	SVM	96.0 & 97.2 %	MIT Home Data Set MIT1 & MIT2	SVM with RBF-Kernel	2.6 % & 8.3 %	2013
	DBN-R	95.0 % & 93.9 %		Reconstruction	17.0 % & 51.8 %	
	DBN-SVM	95.0 % & 95.3 %		SVM with RBF-Kernel	17.0 % & 34.7 %	
	DBN-ANN	93.7 % & 92.8 %		3 layer ANN (HL: 300 units)	2.4 % & 37.3 %	
				3 layer DBN (In, 200 units, 100 units)	Percentages above represent the Rising Edge Accuracy (REA), measuring the right prediction of newly acitvated sensors (compare Table 3.3), Prediction about which sensor will be activated in the next 5 minutes is based on sensory data from the previous 45 minutes	
Human Activity Recognition based on Feature Selection in Smart Home using Back-Propagation Algorithm [66]	MLP	91.8 %	Gathered over 55 days, 600 instances of activities, 647485 sensor events from 2 people	Error: SSE, Momentum: 0.9, LR: 0.005, Iter.: 100000, In: varying, HL: varying, Out: 10, Neuron: Sigmoid	Each activity has 10 features → remove unimportant features according to their inter-class distance, Varying number of input and hidden units depending on the subset of activites, Evaluation: 3-Fold Cross Validation	2014
User Activity Recognition in Smart Homes Using Pattern Clustering applied to Temporal ANN Algorithm [67]	MLP	Mean: 88.0 % & 83.0 %, Runtime: 3.99 s & 25.15 s	Gathered over 1 & 2 week(s) by 2 people	3 Layers (In, HL, Out)	Sensor data preprocessing, Symbolic Aggregation Approximation, Frequent pattern mining, K-Pattern clustering, ANN, Evaluation: 10-Fold Cross Validation	2015

Table 2.2: Overview over machine / deep learning approaches in smart spaces *(2/2)*.

Chapter 3

Related Work

As mentioned in the previous chapter, there is not much related work available regarding deep learning approaches in smart spaces. However, appropriate learning applications in smart environments are described below. These apply, for instance, simple neural networks or machine learning. Machine learning and deep learning approaches in classification tasks are presented afterwards. This chapter concludes with a summary comparing the introduced approaches to our requirements.
Tables 2.1, 2.2 show additional machine learning approaches in smart spaces.

3.1 Machine Learning and Deep Learning in Smart Environments

3.1.1 ACHE - A Neural Network House

One of the oldest approaches is *Adaptive Control of Home Environment (ACHE)*, a neural network house built in 1992 [68]. This approach wants to anticipate the needs of the inhabitans and on the other hand reduce the overall energy consumption. It uses Feedforward Neural Networks (FFNNs) to predict future states and control the physical devices. This is done by means of models of the house and the devices. Reinforcement learning is applied to satisfy the user needs. In this case, any manual adjustments of the users are considered in a *discomfort cost* $d(x_t)$, where x_t denotes an environmental state x at time t. This means that if the user is not satisfied with the settings, the algorithm gets penalized. In addition to that, an *energy cost* $e(u_t)$ is computed, involving the use of electricity and gas resources. Thereby, u_t expresses the control decision u at time t. Both costs are combined in a way, that an *expected average cost* $J(t_0)$ can be calculated. The aim of the reinforcement learning algorithm is to minimize this expected average

cost starting at t_0, given by

$$J(t_0) = E\left[\lim_{\kappa \longrightarrow \inf} \frac{1}{\kappa} \sum_{t=t_0+1}^{t_0+\kappa} d(x_t) + e(u_t)\right]. \tag{3.1}$$

For that reason, a mapping from states x_t to decisions u_t has to be computed yielding an optimal control. As states and decisions have to be compared, both have to be represented in the same currency, in particular dollars. Hence, the energy costs can be represented straightforwardly. The discomfort cost needs an allocation to dollars using a misery-to-dollar conversion factor. It is computed either by measuring the loss of productivity, if the inhabitant's needs are not reached or it is adjusted over several months taking the inhabitant's willingness to pay for gas and electricity into account.
The following architecture is replicated among all control domains, for example air heating, lighting and ventilation. The instantaneous environmental state is used as input for both a *state transformation* and an *occupancy model*. The result after the first one gives statistics in a given temporal window providing more information about the environmental state. The latter determines whether a *zone* (usually a room) is occupied or not. The output of both is fed into *predictors* which predict a future state given the current state. These are implemented either as FFNNs trained with Backpropagation (BP) or as a combination of a neural net and a lookup table. With the help of the predicted future states control decisions need to be made. This is done by using a *setpoint generator* followed by a *device regulator*. The first one specifies a setpoint profile. It denotes a target value of an environmental variable like lighting level or air temperature. To achieve this computed setpoint, the device regulator controls the physical devices. The decision process is split into two part in order to *encapsulate knowledge*. Thus, if the needs or preferences of the inhabitants change over time, only the setpoint generator needs to adjust its values. The reason for this is that the setpoint generator involves knowledge about the preferences of the users. On the other hand, the device regulator knows only about the physical environment. There are two approaches for controlling, *indirect control* or *direct control*. The first one is based upon dynamic programming and models of both the environment and the inhabitants, whereas the latter one uses reinforcement learning. The setpoint generator as well as the device regulator are built by one of these two approaches. The direct approach involving reinforcement learning is used, for example, by a setpoint generator for the lighting controller. On the contrary, the device regulator for indoor air temperature applies the indirect approach using a neural network to learn deviations from a thermal model and the actual behaviour of the house.
Unfortunately, the ACHE approach has no realised evaluation. Due to this lack of evaluation, it is difficult to rate the ACHE approach. In any case, the idea behind ACHE is well elaborated. An example for this is the dividing of the preferences of inhabitants and physical devices in different sections. Also, the use of reinforcement learning to involve the needs of the user is a good attempt. Moreover, the encapsulation of

knowledge enhances reusability which is required in our approach.

3.1.2 Reinforcement Learning aided Smart-Home Decision-Making in an Interactive Smart Grid

Li and Jayaweera [65] use a *Hidden Mode Markov Decision Process (HM-MDP)* model for on-line decision making and a *Q-learning* algorithm to solve the HM-MDP problem. Using this Q-learning approach requires no need for building up a Markov model previously. With probability ϵ (*exploration factor*) the customer selects his actions randomly, and with probability ϵ an action according to the lookup table is choosen. The *Q-function* is defined as follows

$$Q(s, \mu, a) = R(s, \mu, a) + \sum_{s'} Prob(s'|s, \mu, a) V^*(s', \mu^a_{s'}) \tag{3.2}$$

where s denotes the current state and μ the estimated belief mode pair. Given the *current state* s and the current *belief mode* μ, the next belief mode is determined by an *action* a and a next state s'

$$\mu^a_{s'}(m') = \frac{\sum_m x_{mm'} P^{a,m'}_{ss'} \mu(m)}{\sum_{m''} \sum_m x_{mm''} P^{a,m''}_{ss'} \mu(m)} \tag{3.3}$$

where m defines the current *environmental mode*. $x_{mm'}$ expresses a *transition probability* of environment mode m to m', which does not depend on the current state s and action a. On the other hand, the transition of state s to s' is independed of the current mode m. This is due to the determination of the environmental mode on external factors like weather conditions. These factors are not relying on the user action and the user state. If action a is performed by the user in the current state s, the state switches to s' with transition probability $P^{a,m'}_{ss'}$, while the environment mode changes to m'. V^* indicates the value function, meaning the expected return

$$V^\pi_t(s) = \mathbb{E}_\pi \left[\sum_{\tau=0}^{T-1} \gamma^\tau R_{t+\tau}(s_{t+\tau}, a_{t+\tau}) | s_t = s \right] \tag{3.4}$$

which is the expected discounted sum of rewards over the entire time period $[t, T]$. Above, γ denotes the discount factor, s the current state. The policy π is given by a sequence of descision rules $\pi = (d_1, d_2, \ldots, d_T)$ for each time step t, where $d_t : \mathbb{S} \longrightarrow \mathbb{A}$ maps a set of states to a set of actions. The Q-learning algorithm with the Q-function of Equation 3.2 chooses action a based on the pair (s, μ). Then, the agent observes the subsequent state s' and computes the resultant estimated belief mode $\mu^a_{s'}$. After that, a reward $R(s, \mu, a)$ is received. Finally, its old Q-value Q_{n-1} is adjusted using a learning

factor α_n according to the following equation

$$Q_n(s,\mu,a) = \begin{cases} (1-\alpha_n)Q_{n-1}(s,\mu,a) + \alpha_n \left[R(s,\mu,a)\gamma \max_{a'} Q_{n-1}(s',\mu_s^a,a'\right], & (s,\mu,a) = (s_n,\mu_n,a_n) \\ Q_{n-1}(s,\mu,a), & otherwise \end{cases} . \tag{3.5}$$

The training steps of this approach are set to 10^4, the learning rate is fixed to 0.3 and the exploration factor is chosen to be 0.1.
As in the *ACHE* approach (see Section 3.1.1) no real evaluation, to compare with the other approaches, is conducted. However, they state out, that the Q-Learning approach shows a better performance than a greedy and a random algorithm. As the Q-learning algorithm constantly updates its Q-Table, it outperforms the other two after several training steps. This approach maximizes its own profit taking both into account, the fully observable local information and the estimated hidden information of the environment. Q-learning is a deep learning approach exploiting reinforcement learning and is thus well appropriated for smart homes. However, the Q-learning algorithm is provides a rather poor performance in terms of reusability since the underlying reinforcement learning algorithm is trained on a specific environment.

3.1.3 MavHome: An Agent-based Smart Home

Another reinforcement learning algorithm is used in the *MavHome* project [69]. The algorithm uses an inhabitant *history* consisting of actions of the user. Additionally, the frequency of how often a certain action is performed in a certain state is calculated. Thereby, the next action is predicted using a ranking algorithm. As you can see in Table 3.3, the MavHome project uses four different approaches. The *Smart Home Inhabitant Prediction (SHIP)* algorithm works with sequences collected in the histories. The most recent sequence is matched to a sequence in the afore-mentioned history. It returns the action which has the greatest prediction value by ranking different matches. An online algorithm called *Active LeZi (ALZ)* is also used for sequential prediction, as the interaction of inhabitants with the devices is modeled using a Markov chain of the events. It starts by building a Markov model and predicts an action with the highest probability. The third prediction algorithm discussed in this approach is called *Task-based Markov Model (TMM).* Thereby, a Markov model is built-up from collected action sequences. This model is used to predict the next action given the current state. As additional information is useful for a better prediction, TMM first partitiones the action sequence into individual tasks. Then, the partitioned tasks are clustered using a *k-means custering* algorithm. With the help of the clustered output the initial Markov model can be refined using the connection of tasks in the same clusters. A fourth data-mining algorithm called *Episode Discovery (ED)* is used to improve prediction. It identifies significant episodes (e.g. interactions with devices) in the event history of an inhabitant. These significant episodes are then used as basis for further activity prediction.

All in all, the agent wants to maximize its goal. Therefore, a function to maximize the comfort and productivity of the inhabitants while minimizing the operation cost is implemented. As in the previous presented approaches a reinforcement learning algorithm is implemented. It takes the preferences of the inhabitants into account. On the other hand, it wants to operate efficiently. Although an evaluation was realised, the outcome of it can not really be compared to real world applications. This is due to the evaluation being mostly on synthetic data. The algorithm performs well on this synthetic data set, though. Only the predictive accuracy of SHIP was analysed on real world data, on which it did not achieve a good result, namely 53.4 %.

3.1.4 Smart Home Design for Disabled People based on Neural Networks

Hussein *et al.* [63] designed a smart home for disabled people by applying both a FFNN and a *Recurrent Neural Network (RNN)*. The first one is used for safety and security issues determining the outcome of several alarms, e.g. fire alarm. The latter one predicts human behaviour. Both Artificial Neural Networks (ANNs) are needed for prediction issues and thus, the environment adapts to the needs of the inhabitants according to the predicted scenarios. The networks learn by using sampled data from sensors and cameras. The FFNN and RNN together accomplish the main goals of this smart home, *security* and *automation*. For example, the first one uses access codes on the main door and windows, motion sensors and cameras which perform face recognition. The latter facilitates controlling and monitoring of all devices in the smart home. Other goals, not to neglect especially for disabled people, are *health care* and *safety*. These are achieved using, for instance, fall detection mechanisms, constant monitoring in search for abnormal events of both the vital signs and the daily activities of the inhabitants. Additionally, the emergency personnel is informed in case it is needed. Furthermore, a *suitable interaction method* for the users according to their disability is required, which acts as an interface to the environment, e.g. speech recognition or a visual interface like a tablet or computer. Technically, the FFNN-based fire alarms permanently monitor the levels of, for example, carbon monoxide and oxygen. If something is wrong with these values, the fire department is informed automatically. Moreover, in case of fire it is traced and passed on to the fire department personnel to help them extinguishing it. The inhabitants are also informed using a fire alarm and the emergency lights are turned on leading them the way to the nearest exit. In order to be 100% reliable, particularly in case of an emergency, *redundant connections* to the outside world are established. The FFNN is implemented with *perceptrons* as type of artificial neuron. It uses an input layer with five units, representing the five main parameters indicating a fire (carbon monoxide level, oxygen concentration, smoke detection, heat level, flame detection). The output layer consists of one single output determining either if there is fire or not. The data collected from sensors is divided into two subsets, a training set (75%) and a test set (25%). Furthermore, the RNN is implemented having an input layer of three

units and an output layer consisting of two units. To find the best fitting type of neuron, the sigmoid and tanh neuron are tested. The sigmoid neuron is choosen. The data used for training this network is collected from the Activities of Daily Living (ADL) of the inhabitants. The outcome of the RNN represents two actions, predicted with respect to the two or three input actions. The FFNN is evaluated using a learning set of 63 sets and a test set of 23 sets. Its test result is 95%. On the other hand, the RNN is analysed using learning samples of 65 sets and test samples of 21 sets with a test result of 80% correct samples.
This approach, however, is only a *theoretical design proposal*. Thus, the data used is virtually collected and the RNN and FFNN are not evaluated on real world applications. Furthermore, no real specification about the number of hidden neurons in both networks is indicated, if there even are any hidden neurons. However, using a FFNN to forecast fire and other emergencies is a good idea, as it can be trained in a supervised manner with labeled training data. Moreover, the RNN fits well to sequential data. It further depends on previous steps and is therefore ideal for human behaviour prediction in a smart environment. It is a good attempt to use different neural networks for different purposes. Each network is thus applied where it performs best. Hence, a FFNN and a RNN are appropriate neural networks which can be applied in smart homes. The first one suits well for classifying data and the latter is good in predicting human behaviour.

3.1.5 Recognizing Human Activity in Smart Home using Deep Learning Algorithm

A deep learning algorithm which recognizes human activity in a smart home is implemented by Hongqing and Chen [33]. Their algorithm builds upon a *Deep Belief Network (DBN)* consisting of several *Restricted Boltzmann Machines (RBMs)*. They divide the training steps into *pre-training* and *fine-tuning*. The first one is a bottom-to-top process, as it trains each RBM on its own beginning with the first one. The output of the hidden layer of the first one is then used as visible layer for the next RBM and so on until the last RBM is reached. After the pre-training step fine-tuning is applied. As this approach uses supervised learning, the error is computed and propagated back. This process is repeated until the error becomes less than a predefined threshold value. The output of this DBN is one out of ten actvities, e.g. leaving the house, bathroom and lunch. These activities are chosen as they are thought of being difficult for disabled people to be finished on their own. The data used for training and testing was collected from the Center of Advanced Studies in Adaptive Systems (CASAS) research project at Washington State University. Training data is gathered by students performing activities in this smart home. The data is labeled with an activity-ID and a person-ID. As already mentioned, ten activities are chosen. Each of them is executed in this smart environment for more than 50 days. Each activity sample has five parts, representing date, time, sensor-ID, sensor value, and activity label. The DBN was built-up with 500

units at the first hidden layer followed by 300 units in the next hidden layer and another 100 hidden units, adding up to three hidden layers. The output is constructed by 10 units, each representing one out of the 10 activities. The learning rate in this approach is chosen to be 0.1. To update the weights, a gradient-based *Contrastive Divergence (CD)* algorithm is used. Then *Gibbs sampling* is applied to update the reconstruction distribution of the model. Hongqing and Chen detect that initializing the weights randomly does not work well as the model performs worse. Hence, they decide to use a layer by layer pre-training step to find appropriate weights according to [70]. Instead of initalizing the weights randomly in the beginning, they are drawn from the standard normal distribution. Then, pre-training is applied. After the probability of the top layer units is computed, the probability and weights of every layer can be obtained. Afterwards, BP is performed as a fine-tuning step. The above mentioned contrastive gradient is used for updating the weights. This method yields both a fast convergence and a fast search in different directions of the function. Each activity is evaluated on its own, thus, we calculated the average value of all the probabilities of the activities. This accuracy is 86.89%. The deep learning approach is compared with a *hidden Markov model* and a *naïve Bayes classifier* model. It performes better, although in two activities a lower recognition rate is reached. The accuracy of their DBN gets worse if more pre-training epochs are executed.
The approach introduced above is evaluated on real world data collected in a smart home project called CASAS. Hence, this DBN is suitable for activity recognition in smart environments as it achieves good results in analyzing human activities. This, however, is a supervised approach, as it already has labeled training data. Nevertheless, a DBN is well suited to be applied in smart spaces since it can be trained in an effective way and does not necessarily need labeled training data.

3.1.6 Human Behavior Prediction for Smart Homes using Deep Learning

Another approach, which uses a *DBN* with *RBMs* as well, is the one by Choi *et al.* [34]. However, it makes use of an improved *CD* method using *bootstrapping* and *selective learning*. The standard CD method has a computational complexity of $O\left(2^{V+H}\right)$, where V and H are respectively the numbers of visible and hidden units. This bootstrapping method is evaluated on the Pima-Indians-Diabetes data set resulting in a classification rate of 74.21%. The online-learning method reached 72.67%, mini-batch learning 73.89%, and full-batch learning 72.86%, not differing much from each other. To perform this evaluation, the weights are drawn from the Gaussian distribution with zero mean and a standard deviation of 0.01. Additionally, a learning rate of 0.01 is choosen. Full-batch learning updates the weights over all training data, whereas in on-line training the weights are updated by using one training instance at a time. On the other hand, mini-batch learning updates weights over a mini-batch, which size is less than the whole data set. They also introduce two hybrid deep architectures, a DBN combined with a

ANN using BP with *Mean Squared Error (MSE)* and a DBN with a *reconstruction method*. The neural network consists of three hidden layers, each having 100 hidden units. The latter concatenates the input and output data in the visible layer to train the DBN. By propagating up and down the hierarchy of the DBN, they reconstruct the visible layer (like in a single RBM). The predicted output is the output part of this reconstructed visible layer, though. The DBN consists of one visible layer and two hidden layers, the first hidden layer consisting of 200 nodes and the second one consisting of 100 nodes. The methods introduced in this paper are evaluated on both data sets MIT1 and MIT2. As this evaluation includes both activation and deactivation of sensors and sensors can be active (deactive) a long period of time, an algorithm can make a high accuracy value while having poor prediction on the activation (deactivation) of a sensor. Due to the reason that many sensors are inactive most of the time, a new evaluation metric called *Rising Edge Accuracy (REA)* is introduced. It measures the prediction of newly activated sensors (see Table 3.3).
As the deep learning approaches in this paper are evaluated on real world data (MIT dataset), they suit well for human behaviour prediction in smart homes. The accuracy for both DBNs is above 93%. Hence, a DBN is an architecture which can be used in smart spaces, applying deep learning and yielding both a good accuracy and an efficient performance.

3.1.7 Smart Home System Design based on Artificial Neural Networks

Badlani and Bhanot [61] designed a smart home system using both a *FFNN* and a *RNN* to reduce the overall power consumption. A system based on an *ANN* tracks the action of the user at different times in order to predict human behaviour accurately. Moreover, it switches of devices accordingly. For instance, if the user wants to study the intensity of the desk lamp is increased and the light is dimed. This is how energy is saved. As the approach in Section 3.1.4, the FFNN is used for security and safety applications like fire alarms and the RNN predicts the human behaviour by taking previous actions into account. Both networks are trained with BP and Backpropagation Through Time (BPTT), respectively, using Gradient Descent (GD) and minimizing the *Sum of Squared Errors (SSE)*. Additionally, values of power measurement devices can be used as input to the system. Supervised learning is used to implement the RNN. Continuously, data is taken from the user and stored in memory. All this data becomes the learning data, which is fed into the neural network after one day. As RNNs are efficient in processing sequential data, changes in the human behaviour over time can be detected. The weights for feeding back the current hidden state into the *memory*, representing the state of the previous hidden unit, are set to 1.0. The storing of previous states requires a memory which holds data for a certain time window and adapts to the rapid input changes. To determine the size of this window is crucial. If it is too small between important events, the gradient vanishes exponentially and, thus, the

error diminishes faster. On the other hand, a large window size allows for storing more data and this helps to not miss any important information needed. The RNN is independent from the FFNN which also uses a supervised approach. It is trained using labeled training data to detect security issues, e.g. a bursting fire. The reason why a RNN is used for human behaviour prediction is clearly the including of past states into the current prediction. This proposed system feeds the data collected over one day into the RNN and trains it that way. Subsequently, a profile of the specific person is created. All in all, the FFNN uses sensor outputs for processing and the RNN user inputs. Moreover, the system uses an ANN-based approach to register and authenticate users. First, a *registration phase* uses a hashed user name as input and the corresponding hashed password as desired output. Then, the network gets trained and afterwards the weights are stored in the system. These weights can not be exploited by an intruder and, hence, the neural network-based user registration allows to securely store the passwords, user profiles, device profiles and access controls in smart home applications. Furthermore, an *authentication phase* is used for user recognition. Therefore, the system applies the same hash function to both the user name and the password. Then, the output of the trained neural network is compared to the hashed password. Depending on the identity of the results the user is either authenticated or rejected. The RNN is built as a single layer perceptron model with two input units. The learning rate is set to 0.01 and the final number of training epochs is 117. The training set consists of 160 samples and the test set of 100 samples. The evaluation yields a specificity of 70.96% and 65.52% and a sensitifity of 97.47% and 96.97% on the training set and test set, respectively.
Badlani and Bhanot use a *home controller* which acts as a middleware system. Thus, the devices and sensors are able to communicate with the controller. This approach is comparable to the *Distributed Smart Space Orchestration System (DS2OS)* with *Virtual State Layer (VSL)* as corresponding middleware [59]. Furthermore, the idea of a RNN generating a user profile to adjust the smart home devices is well thought out. Moreover, performing user authentication with the help of a neural network is a good idea for further developments. Here again, it can be seen that FFNNs and RNNs are well suited as application in smart spaces. The first one classifies data efficiently and the latter one predicts human behaviour precisely by taking previous time steps into account.

3.2 Machine Learning and Deep Learning in Classification Tasks

3.2.1 Deep Big Simple Neural Nets Excel on Handwritten Digit Recognition

Ciresan *et al.* [71] propose a statistic with *Multilayer Perceptrons (MLPs).* They evaluate MLPs of different size on the MNIST handwritten digit benchmark data set [16]. MNIST consists of two sets, a training set and a test set. The first one has 60.000 examples of

handwritten digits and the latter one 10.000. The image size is 28×28. The training set is thereby often divided into two sets, one for training (55.000) and one for validation (5.000) [71]. Five MLPs are used in that study, the number of hidden layers ranging from 2 to 9. The number of hidden units is also varying, yielding 1.34 to 12.11 million free parameters and hyperparameters. The networks use the *BP* approach without momentum, but with a *decaying learning rate* as the training progresses (from 10^{-3} to 10^{-6}). The weights are initially *uniformily* drawn from $[-0.05, 0.05]$ and the activation function used in this approach is a scaled version of the *hyperbolic tangent*: $y(a) = \mathrm{A}\ tanh(\mathrm{B}a)$ with A = 1.7159 and B = 0.6666. They increase the number of training examples by applying affine and elastic deformations on the already available images. Table 3.4 shows the accuracy of the different MLPs. As it can be recognized, the more layers and units are added, the better the accuracy gets. *MLP 5*, for instance, has more layers but a smaller number of neurons per layer and thus performs worse than *MLP 4*.
Hence, if using MLPs or FFNNs it is necessary to be able to choose different numbers of hidden layers and hidden units, respectively, in order to obtain the best performance. This can be achieved by providing all necessary hyperparameters and parameters in one file, which can be easily changed to the needs of the user.

3.2.2 Deep Learning-Based Feature Representation for AD/MCI Classification

To diagnose Alzheimer's Disease (AD) and its prodromal stage Mild Cognitive Impairment (MCI), Suk *et al.* [72] make use of a *Stacked Autoencoder (SAE)* and *multi-task and Multi-Kernel (MK) Support Vector Machine (SVM) learning*. In this work, they use the *Magnetic Resonance Imaging (MRI)*, *Positron Emission Tomography (PET)* and *Cerebrospinal Fluid (CSF)* data available in the ADNI dataset [73]. The data is gathered from 51 AD patients, 99 MCI patients and 52 healthy normal patients. The MCI patients are further divided into 43 patients who progressed to AD and 56 patients who did not progress to AD in 18 months. Each data file consistes of a brain image and two types of clinical scores, called *Minimum Mental State Examination (MMSE)* and *Alzheimer's Disease Assessment Scale-Cognitive Subscale (ADAS-Cog)*. Both, MRI and PET images are preprocessed using different procedures. After this step, 93 features from a MRI image and a PET image are filtered out, respectively (see Section *2 Materials and Preprocessing* in [72]). These features are used as input for the SAE, which then discovers the underlying feature represenation. A *Sparse Autoencoder (SpAE)* along with a sigmoid activation function is used as building block for the SAE. In particular, SpAEs penalize a large average activation of a hidden unit over the training samples. Thus, sparse connections are obtained as a result of driving the activation function of many hidden units toward zero. In addition, unsupervised *pre-training* is performed to initialize the parameters before the they are further optimized using supervised *fine-tuning*. To conduct fine-tuning, an additonal output layer is stacked on top of the SAE. The number of units contained in this layer are equal to the number of different classes. The resulting multi-layer

network can now be optimized using BP with GD. Furthermore, the top output layer of the multi-layer network is used to find the optimal SAE structure, which yields the best classification accuracy. The output of the last hidden layer from the SAE determines the final underlying feature representation. This latent feature representation is concatenated with the original input features to build an *augmented feature vector*, which is then fed as input to the multi-task learning. The aim of the multi-tasking step is to find optimal weight coefficients $a_s^{(m)}$ in order to regress the target response vector $t_s^{(m)}$ with a combination of the features in $F^{(m)}$ with a group sparsity constraint. $F^{(m)} \in \mathbb{R}^{N \times D}$ represents the augmented feature vector, where N denotes the number of samples and D indicates the dimension of the vectors. The objective function $\mathcal{J}$ is given by

$$J\left(A^{(m)}\right) = \min_{A^{(m)}} \frac{1}{2} \sum_{s=1}^{S} ||t_s^{(m)} - F^{(m)} a_s^{(m)}||_2^2 + \lambda ||A^{(m)}||_{2,1} \tag{3.6}$$

with λ denoting a sparsity control parameter and $A^{(m)} = \left[a_1^{(m)} \dots a_s^{(m)} \dots a_S^{(m)}\right]$. $m \in 1, \dots, M$ and $s \in 1, \dots, S$ indicate respectively the modality index and the task index. Furthermore, $||A^{(m)}||_{2,1} = \sum_{d=1}^{D} ||A^{(m)}[d]||_2$ represents the $L_{2,1}$-norm, which is applied to select jointly-used features. $A^{(m)}[d]$ incidates the d-th row of the matrix $A^{(m)}$.
The decision function of the MK-SVM depends on the selected features $\tilde{X}^{(m)} = \left\{\tilde{x}_i^{(m)}\right\}_{i=1}^{N}$, which are used as training samples. They are derived from the multi-task learning above (see Equation 3.6) by taking only the features with weight coefficients larger than zero. Thus, this decision function is given by

$$f\left(\tilde{x}^{(1)}, \dots, \tilde{x}^{(M)}\right) = sign\left\{\sum_{i=1}^{N} \zeta_i \alpha_i \sum_{m=1}^{M} \beta_m k^{(m)}\left(\tilde{x}_i^{(m)}, \tilde{x}^{(m)}\right) + b\right\} \tag{3.7}$$

where ζ_i denotes the class-label of example i, and α_i and b are a Lagrangian multiplier and a bias, respectively. The kernel function of the m-th modality is indicated by $k^{(m)}\left(\tilde{x}_i^{(m)}, \tilde{x}^{(m)}\right) = \phi^{(m)}\left(\tilde{x}_i^{(m)}\right)^T \phi^{(m)}\left(\tilde{x}^{(m)}\right)$, where $\phi^{(m)}$ represents a kernel-induced mapping function. Further, $\beta_m \geq 0$ expresses the weight coefficient of the m-th modality, constrained by $\sum_m \beta_m = 1$. Additionally, Suk *et al.* reference [74] and [75] for a more detailed explanation.
The evaluation is conducted using the following three classification problems: AD vs. Healthy Normal Controls (HC), MCI vs. HC, and Mild Cognitive Impairment-Converter (MCI-C) vs. Mild Cognitive Impairment-Non-Converter (MCI-NC). They make use of 10-fold *cross-validation* for training/testing. For unbiased evaluation cross-validation is repeated 10 times. Furthermore, a linear kernel is used in the SVM. The SAE is built-up by three hidden layers for MRI, PET and *CONCAT* and by two hidden layers for CSF based on experiments, where CONCAT expresses the concatenated MRI, PET and CSF features in a single vector. Moreover, the number of hidden units is derived by using the classification results of the SAE-classifier. Table 3.1 shows the number of

Number of hidden units based on the best performance result of the classifier				
Classification Task	MRI	PET	CSF	CONCAT
AD vs. HC	0.857 ± 0.018	0.859 ± 0.021	0.831 ± 0.016	0.899 ± 0.014
	500, 50, 10	1000, 50, 30	50, 3	500, 100, 20
MCI vs. HC	0.706 ± 0.021	0.670 ± 0.018	0.683 ± 0.020	0.737 ± 0.025
	100, 100, 20	300, 50, 10	10, 3	100, 50, 20
MCI-C vs. MCI-NC	0.549 ± 0.037	0.595 ± 0.044	0.589 ± 0.026	0.602 ± 0.031
	100, 100, 10	100, 100, 10	30, 2	500, 50, 20

Table 3.1: Performance of the SAE-classifier denoted with mean ± standard deviation. The number of hidden units is given from bottom-to-top layer.

hidden units of the SAE based on the best classifier performance. The performance of the MK-SVM is represented in Table 3.4. The SVM is evaluated using the augmented feature vector including both the low-level features, meaning the input to the SAE, and the latent features, meaning the learned feature representation of the SAE.

3.2.3 Domain Adaption for Large-Scale Sentiment Classification: A Deep Learning Approach

A deep learning approach, which extracts important features in an *unsupervised* manner, is conducted in [76]. The work focuses on *domain adaption*. This means that two data sets are used, a *source domain S* and a *target domain T*. Thereby, S provides labeled training examples and T provides examples of data, on which the classifier is applied. In case of domain adaption S and T do not have to be sampled from the same distribution, in other words S is drawn from distribution p_S, whereas T is drawn from distribution p_T. The aim of the deep learning algorithm is to find a good function for mapping S to T. This is, the algorithm is trained on p_S and should generalize well on p_T. Glorot *et al.* [76] use a *Stacked Denoising Autoencoder (SDAE)* in their work. For measuring the reconstruction error $loss\,(x, r(x))$ the Kullback-Leibler (KL) divergence between x and *r(x)* was used. A *Denoising Autoencoder (DAE)* is fed with a stochastically corrupted input vector $\tilde{x}$ instead of the usual input x. As the name reveals it, the DAE has to *denoise* the input, meaning to minimize the reconstruction error $loss\,(x, r(\tilde{x}))$. Thus, the DAE can not just copy its input $\tilde{x}$ to its output.
The domain-adaption for sentiment classifier is approached the following way. They use unlabeled data from various domains and a labeled dataset from one domain only. First, a SDA learns higher-level features from text reviews of all available domains. The hidden layer is composed of Rectified Linear Units (ReLUs). Training is done *greedy-layer wise* using *Stochastic Gradient Descent (SGD)*. Furthermore, sigmoid neurons are used in the first layer of the decoder. The corruption of the input vector is done by a *masking noise*, meaning each input is set to zero with probability p. As already mentioned, for measuring the reconstruction error, the KL divergence is chosen. The DAEs in upper

layers do not only use the *softplus* activation function, meaning $log\,(1 + e^x)$, as decoder output units, but also the *squared error* as reconstruction loss and a *Gaussian* corruption noise. Furthermore, ReLUs are used in the input layer in upper DAEs. The reason behind it is to keep the representation sparse. The new feature representation of the data is thus defined in the code layer at different depths. Afterwards, as a second step, a linear *SVM* is trained on the extracted features of the SDAE which uses the labeled training data of the source domain S as input. The error of the SVM is measures by the *squared hinge loss* $\max(0, 1 - ty)^2$, with y being the output of the SVM. Due to the use of ReLUs, sparse representations with exact zeroes are provided for the code layer. This is advantageous for a linear SVM.

For evaluation, the described SDAE-SVM is compared against a linear SVM which is trained on the raw data and acts as a *baseline*. The hyperparameters of both SVMs are choosen by *cross-validation*. The masking noise probability is explored in {0.0, 0.3, 0.5, 0.6, 0.7, 0.8, 0.9}, the optimal value is 0.08. A Gaussian noise standard deviation, which is used in upper layers instead of the masking noise, is examined in $\{0.01, 0.1, 0.25, 0.5, 1.0\}$. The size of hidden layers is varied between $\{1000, 2500, 5000\}$ and the best performance is obtained using 5000 hidden units. Furthermore, a L_1-regularization penalty is imposed on the activation values, it ranges between $\left\{0.0, 10^{-8}, 10^{-5}, 10^{-3}, 10^{-2}\right\}$. The learning rate is chosen from $\left\{10^{-4}, 10^{-3}, 10^{-2}, 10^{-1}\right\}$. To analyze the performance the *transfer loss* t is defined by $t(S,T) = e(S,T) - e_b(T,T)$, where $e(S,T)$ indicates the *transfer error* which corresponds to the test error from a method which is trained on the source domain S and tested on the target domain T. Moreover, $e(T,T)$ represents the *in-domain error*. On the other hand, $e_b(T,T)$ denotes the *baseline in-domain error* describing the test error which is obtained by the baseline method, in this case, the linear SVM trained on the raw data. Furthermore, the *in-domain ratio* $I = \frac{1}{m} \sum_S \frac{e(T,T)}{e_b(T,T)}$ and the *transfer ratio* $Q = \frac{1}{n} \sum_{(S,T)_{S \neq T}} \frac{e(S,T)}{e_b(T,T)}$ are computed. The evaluation is conducted using the *Amazon-benchmark* consisting of more than 340.000 reviews from 22 different product types. The labels of the reviews are either *positive* or *negative*. This dataset is heterogeneous, heavily unbalanced and large-scaled. In Table 3.4 three methods are compared to the baseline SVM: a SDAE-SVM-1 which uses one layer, a SDAE-SVM-3 composed of three layers of 5000 units and a MLP consisting of one hidden layer with 5000 units with tanh activation functions. On top of it, a *Softmax Regression (SR)* classifier is stacked. In [76] the averaged generalization transfer error according to the transfer ratio of each model and their in-domain ratio is depicted.

It can be recognized from the results shown Table 3.4 that stacking more layers yields a better error if applying unsupervised learning.

3.2.4 Deep Learning-Based Classification of Hyperspectral Data

Chen *et al.* [14] propose a deep learning approach to classify hyperspectral data. The problem of classifying hyperspectral data is that due to the high dimensionality (high number of spectral channels) and the large spatial variability of spectral signature, almost no labeled training examples are available. They present an unsupervised approach for feature extraction of hyperspectral data which makes use of a *SAE*. The Autoencoders (AEs) which build the SAE use the principle of *tied weights* (see Equation 3.10). Simply put, the encoded input y (Equation 3.8) and the reconstructed output z (Equation 3.9) are computed using the same weights W

$$y = f\left(W_y x + b_y\right) \tag{3.8}$$

$$z = f\left(W_z x + b_z\right). \tag{3.9}$$

Thereby, the activation function is denoted by f, x indicates the input. W_y, b_y and W_z, b_z are respectively the weights and the bias of the encoder and decoder. Applying the tied weights constraint

$$W_y = W_z = W \tag{3.10}$$

the parameters get almost halved. To train this SAE approach, first, *pre-training* is applied. After stacking a *Logistic Regression (LR)* classifier on top of the SAE, a *fine-tuning* step is adapted using BP. The activation functions of the encoder and decoder in Equations 3.8, 3.9 are set to a sigmoid functions. The cost function used in this approach is *cross-entropy* (Equation 3.11, since it works well together with sigmoid neurons. It further helps to change weights although the neurons saturate

$$C = -\frac{1}{m}\sum_{i=1}^{m}\sum_{k=1}^{d}\left[x_{ik}\log\left(z_{ik}\right) + \left(1 - x_{ik}\right)\log\left(1 - z_{ik}\right)\right]. \tag{3.11}$$

Since the weights are updated using a *mini-batch* update strategy, two sums are needed, where the second one sums over the mini-batch of size m. On the other hand, d indicates the size of the input vector and x_{ik} and z_{ik} denote respectively the k-th element of the i-th input and the i-th reconstruction of the mini-batch. As the reconstruction layer is only needed for adjusting the weights and biases and obtaining a good feature representation, it is removed after training. The SAE is built-up by stacking several encoders layer by layer. The LR layer stacked on top of the SAE uses *softmax* as its output activation function. The LR is a neural network consisting of one single layer only. That is why it uses the features of the last layer of the SAE as input and produces the output according to the number of classes defined by the number of softmax output units. Furthermore, the dataset is partitioned into 60% training set, 20% validation set and 20% test set yielding a split ratio of 6:2:2. First, a single AE using 100 hidden units is trained and evaluated on the *Kennedy Space Center* (KSC) dataset, which represents

the mixed vegetation site over the KSC. The result is shown in Figure 3.1. As it can be seen there, after merely 100 training epochs a almost perfect reconstruction is reached. Chen *et al.* further analyse the runtime according to different hidden and input sizes while the pre-training epochs are fixed to 5000 and the fine-tuning epochs to 50000. Figure 3.2 shows the analysis results. The training time grows with increasing input and hidden layer sizes whereas the training time grows proportionally with respect to the number of training epochs with fixed input or hidden layer sizes. Furthermore, they evaluate the testing time. It shows, for instance, that the AE-LR with hidden size 20 requires 1.14 s on the KSC dataset, whereas a linear SVM needed 52.32 s. This evaluation depicts that deep learning approaches are much faster on testing than conventional machine learning techniques.
Additionally, the approach studies the impact of the depth of a SAE according to the classification accuracy. Table 3.2 illustrates this accuracy and the testing runtime. Therefore, SAEs with different depths are tested on the KSC data consisting of 176 spectral channels and 13 classes. This results in a SAE with 176 input nodes, the size of the hidden layer is set to 20 and the output layer of the classifier is set to 13, one unit for each class. Higher accuracies can be reached if the pre-training and fine-tuning epochs are increased. Furthermore, the depth is evaluated using the *Pavia* data set, which shows an urban site over Pavia, Italy. It consists of 103 spectral channels and 9 classes.
However, the above mentioned analysis is conducted using only spectral data. They discovered that joint data consisting of both spectral and of spatial-dominated data yields a higher accuracy. In Table 3.4, the SAE-LR approach introduced in this paper is compared to a SVM using a *Radial Basis Function (RBF)* (RBF-SVM). The SAE-LR is built by one hidden layer with 20 units. Its pre-training step consists of 3300 epochs and the fine-tuning step takes 400000 epochs. The classification is repeated 100 times and the average accuracy can be seen in Table 3.4. The RBM-SVM and the proposed approach are tested on both the KSC and Pavia dataset.
From Table 3.4 it can be seen that the difference between the RBF-SVM and the proposed SAE-LR classifer do not vary much. Furthermore, this approach does not only evaluate the accuracy of the classifiers but also perform tests on the impact of depth and different sizes of hidden and input layers. From it, we can deduce that an increasing depth yields on the one hand a higher accuracy, but on the other hand a depth which is too large causes the opposite. Note that, according to the depth evaluation, the accuracy for KSC dataset is best using a depth of 3. However, in the evaluation a depth of 1 is used and it is called stacked although it is a simple single-layer AE.

3.2.5 Using Deep Learning to enhance Cancer Diagnosis and Classification

Another unsupervised feature learning approach together with supervised classification is conducted in [77]. Cancer classification is based upon the *gene expression data*, which measures the level of activity of genes within a given tissue. The problem regarding

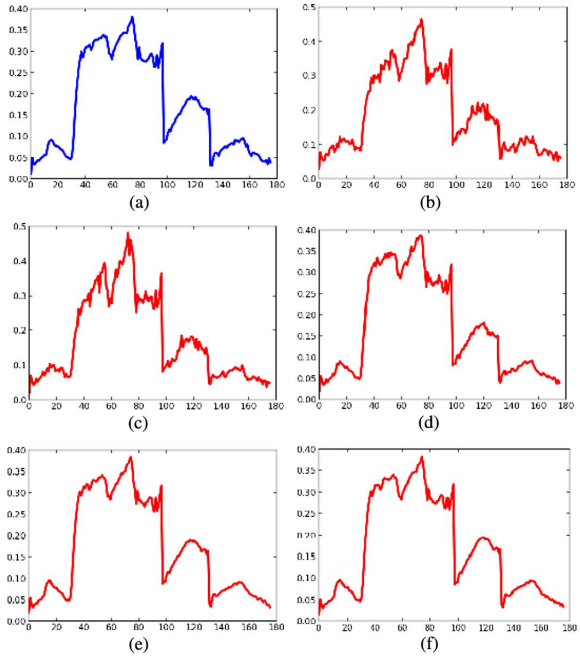


Figure 3.1: Reconstruction of the input (a) at different iteration epochs 1, 10, 100, 1000 and 3500, respectively from (b) to (f) [14].

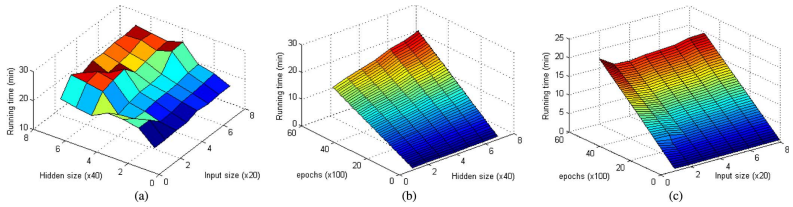


Figure 3.2: Training time of an AE according to different hidden and input sizes (a), the elapsed time on each epoch with varying hidden sizes (b) and the elapsed time on each epoch while varying the input size [14].

Effect of Depth on the Classification Accuracy				
Depth	Overall Test Set Accuracy		Running Time on Test Set	
	KSC	Pavia	KSC	Pavia
1	94.63 %	92.93 %	0.12 s	0.19 s
2	95.45 %	94.95 %	0.15 s	0.27 s
3	**96.55** %	94.99 %	0.20 s	0.35 s
4	95.27 %	**95.16** %	0.22 s	0.42 s
5	93.91 %	95.13 %	0.24 s	0.48 s

Table 3.2: The impact of the depth of a SAE according to the classification accuracy [14].

cancer diagnosis with learning techniques is the high dimensionality of the gene expression data as well as the small number of available training examples for a given tumor (only a few hundred). To overcome this issue, an unsupervised feature learning approach is applied. As there is no need for labeled training data, the unlabeled data is obtained by combining data from different tumor cells. This approach is divided into two phases, *feature learning* and *classifier learning*.
The first is further split up into two steps. First, the dimensionality of the feature space is reduced using *Principal Component Analysis (PCA)*. Due to the high dimensionality of the data, ranging from 20000 to 50000 features, which also contains redundant and noisy data, this step is necessary. The result of PCA is a linear function of the input data. As there is also an interest in the non-linear data, meaning the underlying structure of the dataset, a second step is attached. The output of the PCA is extended by randomly adding some of the original features of the data. By applying a *SpAE* to the augmented feature data, the non-linearities of the data can be captured. The SpAE with K hidden units is trained using BP by minimizing the *squared reconstruction error*

$$\min_{b,a} \sum_{i}^{m} ||x_u^{(i)} - \sum_{j}^{K} a_j^{(i)} b_j||_2^2 + \beta ||a^{(i)}||_1 \tag{3.12}$$

where the last part $\beta||a^{(i)}||_1$ represents a *sparsity penalty*, which fosters the activations to have a low L_1-norm yielding most of them to zero. $x_u^{(i)}$ denotes the unlabeled training input, b is a basis vector and a is a vector of activations, where $a_j^{(i)}$ is the activation of basis b_j for input $x_u^{(i)}$. Equation 3.12 is choosen according to [78]. They choose the sigmoid function as activation function. Additionally, Fakoor *et al.* build a *SAE* with two layers. It was trained using greedy layer-wise *pre-training*. The output of the SpAE is used as input for the next phase.
In the classifier learning phase, a *SR* classifier is trained using both a set of labeled data and a sparse feature representation.
For the evaluation four architectures are compared. The above-mentioned SpAE and SAE with pre-training, a SAE with pre-training and additional *fine-tuning*, a *SVM* with Gaussian kernel and SR. The latter two (SVM, SR) are used as classifier on the PCA-based data and therefore act as a baseline. Furthermore, 13 different datasets are used. *10-fold cross-validation* is applied in order to get the averaege classification accuracy. Due to overview issues only two accuracies out of the 13 of this paper are shown in Table 3.4.
According to Table 2 of [77], the SpAE outperformes the others four times, the SAE with fine-tuning six times and the PCA + SR/SVM together two times. Hence, fine-tuning is a good technique to increase the accuracy of a SAE. Further, it can be recognized that the SpAE works well, too.

3.3 Summary

The evaluation of different approaches regarding learning algorithms in smart spaces is depicted in Table 3.3. Thereby, the first column denotes the approaches described above, the second the method(s) used, the third the evaluation accuracy of this method(s) and the last column shows several remarks to the approaches. Summarizing, the ACHE project [68] made use of a *FFNN* to predict future states and control the physical devices. Additionally, *reinforcement learning* was applied to satisfy the preferences of the users. The next approach about human-behaviour prediction [34] used a *DBN* constructed by stacked *RBMs* as basis. Stacked upon it were either a *SVM* or a simple *ANN* consisting of three hidden layers, each built of 100 units. DBN-R denotes a DBN which uses a reconstruction method. These different approaches were further compared to a standard kernel-SVM and a k-means clustering algorithm. The MavHome project [69] analysed three different prediction algorithms called *SHIP*, *ALZ* and *TMM*. The last one, *ED* denotes a data mining algorithm. The subsequent work [65] made use of a reinforcement learning algorithm using *Q-learning* to solve a HM-MDP. Another DBN consisting of RBMs was applied in [33]. The output of the DBN described one out of ten predefined activities in a smart room. The next approach built a smart home for disabled people [63]. They used a *FFNN* and a *RNN*. The first acts as an alarm system, e.g. a fire alarm. The RNN, on the other hand, is deployed to predict human behaviour. The last approach [61] applied a FFNN and a RNN. The FFNN maked use of sensor outputs, whereas the RNN employed user inputs. Furthermore, this approach used an ANN to authenticate users.
Table 3.4, on the other hand, shows the evaluation of the proposed classification approaches using deep learning algorithms. It has the same structure as the above-mentioned table. The first approach [71] analyzed MLPs with a various layer size on the MNIST dataset of handwritten digits. The overall result of that statistic is that the more layers a MLP has, the better the accuracy. However, the number of neurons has also an influence on the accuracy, as you can see at *MLP 5*. The second proposition [72] used a *SAE* to detect features in the dataset. Afterwards the thus filtered features were used to train a MK-SVM. The evaluation was conducted on three different classification tasks. Next, various learning algorithms were used for a domain-adaption task [76]. A simple SVM was used as a baseline for comparison. The other algorithms used either a MLP with a *SR* classifier stacked upon of it or a SVM stacked upon a *SDAE*. The SDAE-SVM 3 with a larger size of hidden layers had the best accuracy among the four proposed methods. Subsequently, an approach to classify hyperspectral data well was analyzed [14]. Thereby, a SVM with a *RBF* as kernel was compared to a *SAE* with a *LR* classifier stacked upon it on two different datasets. Here, the SAE was again used to extract important features from the hyperspectral data. The last proposal was applied in cancer diagnosis [77]. A SpAE was compared to a SAE, a SAE with fine-tuning and either a *SR* classifier or a SVM stacked upon it. All theses approaches use *PCA*-based

data as input.

There is a variety of different neural network approaches, which are applied in smart spaces, available. Moreover, they achieve good results. As we can further see, each approach uses different parameters and hyperparameters to train a neural network or a classifier. Hence, trying several network configurations is necessary in order to obtain an good performance. That is one reason why a simple way to change the parameters and hyperparameters is needed. However, to implement different neural network configurations knowledge in both machine learning and the respective machine learning library is required. Hence, an easy-to-use machine learning functionality is needed. That means a service has to be provided which can be called and trained by the user. Therefore, no detailed knowledge in the area of machine learning and the respective framework has to be necessary since the service has to provide both the whole structure of the neural network and the corresponding learning algorithm.
As recognized in Section 3.2, deeper neural networks often perform better. Deeper networks can be achieved by, for instance, using more hidden layers or stacking several unsupervised approaches together. This is why the user of a machine learning service has to be able to adapt the respective neural network to his preferences. Due to that reason it is important that the service allows for good usability and reusability.
From Section 3.1 it can be seen that following three neural networks and deep neural networks, respectively, showed a good performance in smart spaces: FFNNs, DBNs and RNNs. That is why the machine learning service should provide these algorithms.
Further design ideas and detailed information of our approach are presented in the next chapter (see Chapter 4).

Evaluation of Different Approaches in Smart Environments			
Approach	Method	Accuracy	Remark
Ache [68]	FFNN & RL	—	No evaluation conducted
Human-Behaviour Prediction [34]	Kernel-SVM	96.0 % & 97.2 %	2.6 % & 8.3 % (*)
	DBN-R	95.0 % & 93.9 %	17.0 % & 51.8 % (*)
	DBN-SVM	95.0 % & 95.3 %	17.0 % & 34.7 % (*)
	DBN-ANN	93.7 % & 92.8 %	20.4 % & 37.3 % (*)
	k-means	78.8 % & 69.6%	19.1 % & 32.7% (*)
MavHome [69]	SHIP	94.4%	On real world data 53.4%
	ALZ	87%	Actions of weekdays and weekends for 30 days
	TMM	74%	Seperated actions of weekdays and weekends for 30 days
	ED	47% & 100%	Simple sequence based prediction algorithm & enhanced with ED
Reinforcement Learning [65]	Q-learning	—	No evaluation conducted
Recognizing Human Activity [33]	DBN	86.8876%	Average over all accuracies of each activity recognition
Smart Home [63]	FFNN	95%	Alarm system
	RNN	80%	Human behaviour prediction
Smart Home with ANN [61]	FFNN & RNN	96.97% & 65.52%	Accuracy based on *Sensitivity* & *Specificity*

Table 3.3: Evaluation of the different approaches mentioned in the related works according to smart environments. (*) These networks were evaluated on both datasets MIT1 & MIT2. These percentages show the $REA = \frac{\#correctlypredictednewlyactivatedsensors}{\#ofnewlyactivtedsensors}$, meaning to predict which sensors will be newly activated.

Evaluation of Different Approaches in Classification Tasks			
Approach	Method	Accuracy	Remark
MLPs MNIST [71]	MLP 1	99.51%	1000, 500, 10 (1)
	MLP 2	99.54%	1500, 1000, 500, 10
	MLP 3	99.59%	2000, 1500, 1000, 500, 10
	MLP 4	99.65%	2500, 2000, 1500, 1000, 500, 10
	MLP 5	99.56%	9×1000, 10
AD/MCI Classification [72]	MK-SVM 1	95.90%	AD vs. HC (2)
	MK-SVM 2	85.00%	MCI vs. HC
	MK-SVM 3	75.80%	MCI-C vs. MCI-NC
Domain-Adaption [76]	SVM	14.50% (3)	Baseline SVM (4)
	MLP	13.90%	5000 hidden units (tanh) + SR
	SDAE-SVM 1	11.50%	5000 units + SVM
	SDAE-SVM 3	10.90%	3×5000 units + SVM
Hyperspectral Data [14]	RBF-SVM	97.69%	KSC dataset (5)
	SAE-LR	97.90%	176, 20, 13
	RBF-SVM	96.20%	Pavia dataset
	SAE-LR	97.82%	103, 60, 60, 60, 60, 9
Cancer diagnosis [77]	SpAE	74.36% & 73.33%	1 hidden layer
	SAE	51.35% & 73.33%	2 hidden layers
	SAE	95.12% & 73.33%	2 hidden layers + fine-tuning
	PCA + SR/SVM	94.04% & 94.167%	(6)

Table 3.4: Evaluation of the different approaches mentioned in the related works according to classification tasks. (1) The size of the different hidden layers in the MLP. The output layer has 10 units, each for one digit (0 - 9). (2) The classification task which was used. (3) The generalization error. (4) The used architecture. (5) The pair of RBF-SVM and SAE-LR was evaluated using the corresponding data set. The number of units in the input layer, hidden layer(s) and output layer is given for both SAE-LR. (6) The better average accuracy of both was choosen.

Chapter 4

Design

This chapter makes use of the deep learning algorithms described in Section 2 and captures essential design ideas for the subsequent implementation. We aim at providing three *machine learning services*, each one providing a different kind of neural network (Feedforward Neural Network (FFNN), Deep Belief Network (DBN), Recurrent Neural Network (RNN)). Since the user decides on his own whether to use a shallow or a deep neural network, we refer to our services as machine learning services. Subsequently, two main aspects of the services are explained. These are reusability and usability. This is due to the reason that the services shall be easily portable and easy-to-use. To achieve both issues each service contains its own *configuration file* which can be changed by the user. Therefore, all configurable parameters and hyperparameters of the respective machine learning algorithm are illustrated. Finally, the design of each service is described in detail.

4.1 Reusability & Usability

The design approach is constrained by means of two terms, *reusability* and *usability*. The user has to be provided with a easy-to-use machine learning functionality. Thus, even users with little or no pre-knowledge in the area of machine learning and the respective machine learning library is able to train, evaluate and deploy a neural network.
Usability is ensured by means of a configuration file which contains all parameters and hyperparameters necessary to use the respective neural network. The file is provided after calling the service. Furthermore, the configuration file contains default values. Hence, the user does not have to change every value but has the possibility to do so. Additionally, the data used for training and deploying the neural network has to be of a certain shape. A functionality is therefore provided to the user which prepares the data to be suitable for the respective neural network.
The machine learning service ensures reusability through seperating the state of the

neural network from the learning algorithm. This is due to the fact that the above-mentioned configuration file contains all parameters and hyperparameters necessary to train and restore a neural network. This further facilitates portability. Moreover, the user is able to experimentally find the right configuration of the neural network by simply changing the configuration file and starting training anew. Since no other actions have to be made, this does not take much time.

4.2 Parameters and Hyperparameters used in Neural Networks

As it can be seen in Tables 2.1, 2.2 in Section 2.5 there exists are large number of parameters which can be changed in different ways in order to fit to a certain problem. We want to facilitate the user to configure all parameters and hyperparameters necessary to train and deploy a neural network on his own. Below the common parameters and hyperparameters of a FFNN, DBN and a RNN are listed.

- Size of the input
- Size of the output
- Type of activation function a (see Section 2.1.1.1)
- Type of bias / weights initialization (see Section 2.1.3.14)
- Size of the mini-batch (see Section 2.1.3.7)
- Number of training epochs (see Section 2.1.3.7)
- Learning rate η (see Section 2.1.3.10)

As a DBN in our case consists of stacked Restricted Boltzmann Machines (RBMs) following additional parameters and hyperparameters are needed.

- Number of stacked RBMs
- Number of hidden units for each RBM

A FFNN as well as a RNN learns by backpropagating an error and updating the weights and biases. The error is computed using a cost function. The weights as well as the biases are updated by applying an optimization algorithm. The additional parameters and hyperparameters for both a FFNN and a RNN are the following.

- Type of optimization algorithm (see Section 2.1.3.7)
- Type of cost function (see Section 2.1.3.6)
- If regularization is applied, the type of regularization and the contribution of the parameter norm penalty to the objective function measured with α (see Section 2.1.3.13)

- If momentum is applied, the momentum rate ϕ (see Section 2.1.3.7)
- If decaying learning rate is used, a value for both the decay rate and the decay steps (see Section 2.1.3.10)
- If early stopping is applied, the metric to use, meaning comparing either the loss or the accuracy at every training step and a threshold value representing the number of rounds (see Figure 2.28)
- If k-fold cross validation is used, a value for the parameter k (see Section 2.1.3.11)

By stacking several hidden layers we are able to construct a deeper FFNN. Thus, following additonal parameters and hyperparameters are needed in the respective configuration file.

- Number of hidden layers
- Number of hidden units for each hidden layer

A RNN uses additional parameters and hyperparameters as it gets sequential data as input. Due to that reason, an error is backpropagated a certain number of steps through time. This number can be fixed in the beginning or alternating for every sequence. Our RNN implementation consists of Long Short-Term Memory (LSTM) cells which can be stacked.

- Number of steps the error gets propagated back if fixed
- Activation function of a LSTM cell
- Size of the LSTM cell
- Forget bias applied in the forget gate
- If stacking several LSTM cells, the number of stacked LSTM cells

According to Section 2.1.3.11 every parameter and hyperparameter might have to be adjusted in order to obtain a better performance, and thus a better accuracy. These adjustments can be easily made by only changing the parameters and hyperparameters in the respective configuration file. Table 4.1 shows a summary of all parameters and hyperparameters mentioned above.

4.3 Machine Learning Algorithm as VSL-Service

We aim at implementing three machine learning components to apply in Distributed Smart Space Orchestration System (DS2OS). The easiest way to do so, is by implementing each algorithm as a Virtual State Layer (VSL) service. Therefore, a context model is needed for each neural network first. The context model is almost the same for each

Parameter and Hyperparameter	FFNN	DBN	RNN
Input size	x	x	x
Output size	x	x	x
Type of activation function	x	x	x
Type of weight and bias initialization	x	x	x
Mini-batch size	x	x	x
Number of training epochs	x	x	x
Learning rate η	x	x	x
Type of optimization algorithm	x		x
Type of cost function	x		x
Type of regularization technique and parameter norm penalty	x		x
Momentum rate	x		x
Learning rate decay rate and decay steps	x		x
Early stopping rounds and metric	x		x
K-fold cross validation	x		x
Number of stacked RBMs		x	
Number of hidden units in each RBM		x	
Number of hidden layers	x		
Number of hidden units in each layer	x		
Number of time steps			x
Activation function of a LSTM cell			x
Type of optimization algorithm			x
Forget bias			x
Number of stacked LSTM cells			x

Table 4.1: Enumeration of the parameters and hyperparameters used in a FFNN, a DBN and a RNN.

network. The user can interact with the service itself by changing its nodes. The context model of a FFNN is given in Listing 4.1. The machine learning services have two modes. One for training including testing, and one for applying the trained algorithm itself.
If a service is in training mode, it creates a configuration file which is provided to the user. The user can change it to his preferences. The respective neural network is created by means of the configuration file and trained afterwards. While in application mode, the machine learning service will only compute a new output if the input data changes. This is due to performance and efficiency issues. Therefore, the service subscribes to its *input node*. After a notification callback, meaning the input has changed, a new output is computed and saved in its *output node*.

Listing 4.1: Context model of a FFNN. The trainMode is set to 1 by default.

```
<feedForward type="/basic/composed">
  <trainMode type="/derived/boolean">1</trainMode>
  <input type="/basic/text"></input>
  <output type="/basic/text"></output>
  <configFile type="/basic/text"></configFile>
  <trainingData type="/basic/text"></trainingData>
</feedForward>
```

Figure 4.1 shows the desired design principle of the machine learning services. The user starts by calling the respective service. A configuration file is provided which can be modified by the user. When the configuration file is saved, the service creates a neural network by taking the different configurations contained in the file into account. The network is trained with the provided data set. After the training step the learned network configuration is saved in the configuration file and the service is ready to predict new outputs which are provided to its context model. This is done by restoring the trained configuration contained in the respective configuration file.
As mentioned above, three machine learning algorithms are implemented as a service. These include a *FFNN* (Section 2.1.3.1), a *DBN* (Section 2.1.3.3) and a *RNN* (Section 2.1.3.4). According to the occurrence in the related works (see Chapter 3) and the application in various use cases (see Tables 2.1, 2.2 in Section 2.5) we identifiy these three machine learning algorithms as most relevant (see Section 3.3). This is why we do not make use of a *Convolutional Neural Network (CNN)* (Section 2.1.3.2) or a *Deep Q-Network* (Section 2.1.3.5) which is difficult to parameterize and generalize. Furthermore, the DBN consists of several stacked *RBMs* and not of *Autoencoders (AEs)*. This is due to the reason that RBMs are more often applied to smart spaces in the related works. Moreover, it is important to provide the ability to make the respective networks deeper. As shown in Section 3.2, increasing the number of processing layers results more often in a better accuracy. This can be achieved, for instance, by increasing the number of hidden layers (see Section 3.2.1).
Data provided to train, validate or test a neural network needs to be of a certain shape.

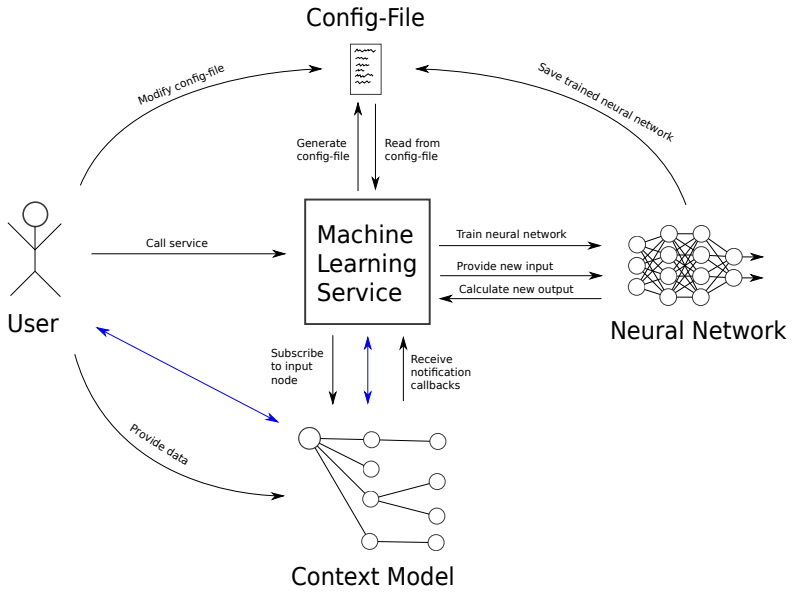


Figure 4.1: Functionality of a machine learning service.

Therefore, each network has its own conditions on the data sets. This is why a further functionality is implemented which prepares the data sets. Each section below involves brief information about the structure of the respective data set. The methods to prepare the data sets are described in Section 5.2.3 in more detail.

4.3.1 Feedforward Neural Network

When applying a FFNN, the user can decide on his own whether it should be deep or not by providing the appropriate number of hidden layers. This can be done by adding more hidden layers to the respective section of the configuration file. The user further has to provide values for the input size and the output size. Both sizes are needed to create the respective neural network. These values have to exactly match the sizes of the training data input and the training data labels. Moreover, a path has to be provided where the model is saved after training.
Since the number of hidden layers is not fixed in advance, a recursive method is applied in order to predict an output. That is why the predicted output y is computed by going backward all the way to the input x. See Figure 4.3 for an illustration of this method. The output of the last hidden layer h_n is computed using the output of the $h_{n-1} - th$ hidden layer as input. This hidden layer then uses again the $h_{n-2} - th$ hidden layer's output as input. This process is continued until the first hidden layer is reached. The output y is then computed by using the resulting outputs of the lower layers to compute the outputs of the higher layers until the last hidden layer is reached. Its output is used to get the output y by applying the output activation function to it.

Data Set—A FFNN needs its data as *(data, label)*-pairs. Since the size of the input and the output is determined in the beginning via the configuration file, every data example needs to have exactly that size.

4.3.2 Deep Belief Network

A DBN is illustrated in Figure 2.18. A deeper representation of a DBN can be achieved by adding more RBMs to the respective section in the configuration file. Moreover, the user has to provide a value for the feature size and a path to save the model has to be provided. The configuration file belonging to a DBN is shown in Figure A.1.
As mentioned in Section 2.1.3.3, the output of DBNs can be processed in two ways. On the one hand it can be directly used. On the other hand, the output can be fed into a supervised learning algorithm, e.g. a FFNN, stacked on top of the last RBM. The latter requires an additional parameter setting in the configuration file. The output size has to be provided, which matches the size of the training data labels.

Data Set—A DBN needs its data solely unlabeled, meaning no label is required. As the input size is provided in the beginning via the configuration file, the data size has to match exactly this size.

4.3.3 Recurrent Neural Network

In order to train a RNN the user has to provide a feature size and an output size. Both have to match the training data sizes. If using the same sequence lengths, this value has to be changed, too. A definition of the sequence length is given below. The configuration file belonging to a RNN is shown in Figure A.2. Moreover, the path to save the model needs to be changed.
Training a RNN is slightly different from the two above-mentioned networks. The design idea is shown in Figure 4.4. It shows a RNN, which consists of three stacked LSTM cells. It is further unfolded n times. Thereby, n is depending on the length of the input sequence x. This sequence consists of n vectors x_i. Since each input x_i entails an output y_i, an output sequence of size n is obtained in the end. Depending on the task, we are either interested in the whole output sequence or in the last output y_{n-1} and thus discard the previous outputs. This subdivided procedure is due to the two different approaches in the related works (see Section 3), where human behaviour is either predicted or recognized. Consider, for instance, an input sequence disclosing that the user first lays in bed, then turns on the lights and stands up. The RNN either classifies the activity into *standing up* or concludes that the user has woken up and opens the shutters. In both cases, however, the last output contains the appropriate information. As mentioned above, the whole output sequence can be used, too. The network can be

```
[Neural Network]
type = Feed Forward Neural Network
save_model_in = /path/to/save/model
[Input Layer]
feature_size = -1
[Hidden Layers]
hidden_layer_1 = 500
hidden_layer_2 = 300
hidden_layer_3 = 100
number_of_hidden_layers = 3
[Output Layer]
output_size = -1
softmax_unit = True
no_activation = False
[Weight]
mean = 0.0
standard_deviation = 0.1
seed = 123
[Bias]
constant = 0.0
[Cost Function]
cross_entropy = True
squared_errors = False
[Optimization Technique]
gradient_descent = True
momentum = False
adagrad_optimizer = False
[Regularization Technique]
dropout = False
weight_decay = False
[Additional Methods]
learning_rate_decay = False
early_stopping = True
k_cross_validation = False
[Hyperparameters]
activation_fct_tanh = True
activation_fct_sigmoid = False
activation_fct_relu = False
learning_rate = 0.08
number_of_training_epochs = 100
mini-batch = 300
momentum_rate = 0.8
p_keep = 0.75
wc_factor = 0.6
lr_decay_step = 100000
lr_decay_rate = 0.96
early_stopping_rounds = 100
early_stopping_metric_loss = True
early_stopping_metric_accuracy = False
validation_k = 10
[Parameters]
display_step = 10
```

Figure 4.2: An example of a configuration file used to initiate a FFNN. The file contains the default values. It is necessary to change the feature size and the output size accordingly. Furthermore, one has to provide a path to save the model. To create a deeper model the number of hidden layers can be extended in the respective section.

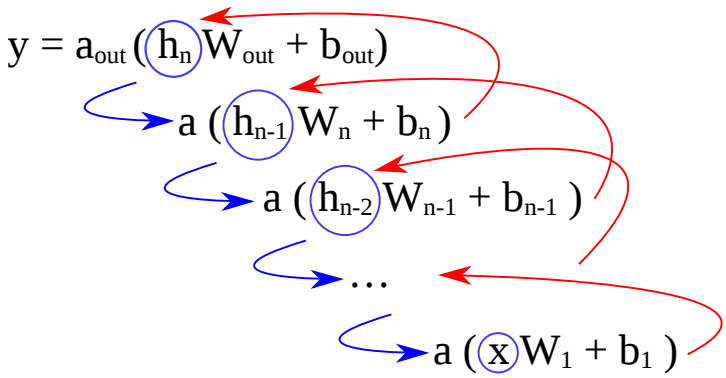


Figure 4.3: An unfolded representation of the recursive method used to compute the predicted output of a FFNN. As the output activation function might differ from the activation functions of the hidden layers, the last step, i.e. to compute the outcome of the output layer, is excluded from the recursion.

trained to predict a vector after each input. Thus, the predicted output y_i of input x_i has to be equal to the next input x_{i+1}, as x_{i+1} is the vector which comes chronological after x_i. To explain this princple, an example from character-based prediction is used in the following. Suppose the input word x is [n e t w o r] and the output y has to be [e t w o r k]. The RNN is thus unfolded six times. The output y_0 (letter e) is based on the input x_0 (letter n) and so on. We are of course interested in the whole output sequence in this example. For our needs, however, this approach might be computational expensive and to fussy.
Figure 4.4 further shows following additional configurable parameters and hyperparameters of a RNN: initial state H_{in}, number of stacked LSTM cells and the sequence length n.
In real world applications, however, not all sequences have the same length n. This is why a possibility to pad sequences of shorter length to match a certain length n is required. This is done by adding vectors consisting of solely zeroes to the sequence. Why this does not affect the training of a RNN is explained in Section 5.2.3.

Data Set—The data provided for a RNN needs further preparing as it uses a 3D-tensor as input. Therefore, a method to create such a tensor is required. It has to stack sequences of data together. If the sequence lengths are not of the same size, the respective sequences have to be padded with zero vectors. Moreover, the input has to contain consecutive sequence vectors. This is due to the reason that RNNs take previous predictions into account. Hence, with the help of chronological ordered sequences, RNNs predict new outputs.

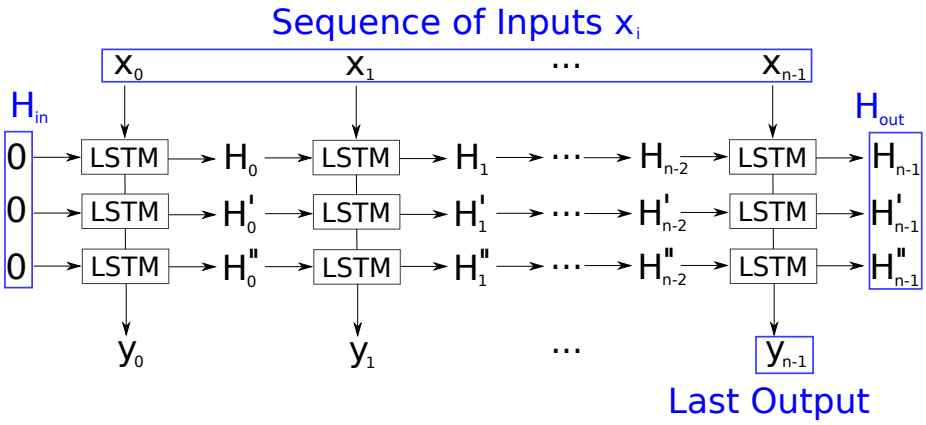


Figure 4.4: Design of a RNN consisting of three stacked LSTM cells.

Chapter 5

Implementation

This chapter summarizes the most important details about the implementation of the machine learning services mentioned in the previous chapter (see Chapter 4). Every service includes another learning algorithm: a Feedforward Neural Network (FFNN) (see Section 4.3.1), a Deep Belief Network (DBN) (see Section 4.3.2) or a Recurrent Neural Network (RNN) (see Section 4.3.3). First, the applied programming language and both additional tools and libraries are introduced. Afterwards, we show specific details about the implementation. An explanation of how to use the different files of the implemented services is given. It further mentions parts of the code where problems might arise if wrong parameters are provieded. Finally, a machine learning *Hello World!* example is described. Using the MNIST data set [16] we explain how to use the three machine learning services to train a FFNN, a DBN and a RNN step by step. Additionally, we want to show how both the accuracy and the loss change while training progresses. Hence, graphs are provided to illustrate their behaviour.

5.1 Tools

The three learning algorithms of the respective services are implemented using *Python*[1] as programming language. This is due to the reason that Google's *TensorFlow*[2] machine and deep learning library is applied (see Section 2.1.5.2). Furthermore, a python interface [79] is employed in order to access the Virtual State Layer (VSL) of the Distributed Smart Space Orchestration System (DS2OS) (see Section 2.2) or more specifically the data stored in the context models (see Section 2.2.1).

[1] `https://www.python.org/`
[2] `https://www.tensorflow.org/`

5.2 Implementation Details

As mentioned previously, each learning algorithm possesses its own context model and thus is independent from each other. Figure 4.1 shows the general functionalitiy of our implemented service approaches. In the following the structure of the implemented machine learning services is given. Two helper libraries are described afterwards. The first one is used to read values from the configuration file easily. The other one is applied to prepare the provided training data. Brought into the right shape it can be fed into a learning algorithm. Subsequently, details about each neural network are described.

5.2.1 Structure of the Services

Each service is implemented using a the python interface mentioned in Section 5.1. The structure of each service is the same as the underlying context model is almost identical (see Listing 4.1).
In general, when calling a service it subscribes to its *input node* first. A configuration file is provided to the user afterwards. When this file is changed and saved, a *build-method* is called. It builds the general structure of the respective neural network and initializes the parameters and hyperparameters. Subsequently, a *train-method* is called which starts training the neural network with the provided training data. When training is finished the *train-mode* is set to 0. Afterwards, the service is ready to predict new outputs from inputs. This is done by restoring the trained neural network with the help of the configuration file.

5.2.2 Read Configuration File

This library applies functions which read a configuration file and return a specified value. As it can be seen in Figure 4.2, each configuration file contains almost all parameters used to create a neural network. This makes reading the specific values easy, since only a particular function needs to be applied. The library returns the value in the required data type. The code snippet represented in Listing 5.1 shows one getter-function which returns the number of hidden layers set in a configuration file.

Listing 5.1: Configuration File Reader

```
'''A code snippet of the configuration file reader library'''

# return int
def get_number_of_hidden_layers(self):
  return int(self.config_reader['Hidden Layers']['number_of_hidden_layers'])
```

5.2.3 Prepare Data Sets

As the data sets fed into the learning algorithms need to be of a certain shape another library is implemented. On the one hand, it provides a function to read data from one or more csv[3]-files. On the other hand, this library is used to create mini-batches while training.
The user has to provide the training data in terms of csv-files to the respective node of the context model (see Listing 4.1).
Listing 5.2 shows a method to read data from csv-files using *tf.TextLineReader()* from the TensorFlow framework. The input parameter *filename_queue* is initalized by the method displayed in Listing 5.3. First, default values, used for columns of the csv-file which are not occupied, need to be provided. Their data type has to be identically equal to the data type given in the csv-file. For example, the parameter *col1* is of type *int* and the remaining columns of type *float*. In this case the data has six columns, of which one represents the corresponding label. Thus, 5 input values are stacked together to form one *tensor*. Finally, the label column is converted into a *one-hot vector*. A one-hot vector consists of only *0* and one *1*. The single *1* indicates the corresponding label.
Listing 5.3 shows how a mini-batch is created. First, the above-mentioned *filename_queue* is created using a *filename_list* which contains the paths to every csv-file the user provides. The variables *min_after_dequeue* and *capacity* are choosen according to the notes in the documentation[4]. Both an *example_batch* and a *label_batch* of size *batch_size* are created using *tf.train.shuffle_batch()*. This method shuffles the training data. If a RNN is applied, this method must not shuffle the data. This is due to the fact that RNNs make use of sequential data (see Section 4.3.3).

Listing 5.2: Prepare Data Set Library - read data method

```
'''A code snippet of the prepare data set library'''

def get_data_example(filename_queue):
   reader = tf.TextLineReader()
   _, value = reader.read(filename_queue)

   record_defaults = [[0], [0.], [0.], [0.], [0.], [0.]]
   col1, col2, col3, col4, col5, col6 = tf.decode_csv(value,
       record_defaults=record_defaults)
   features = tf.stack([col2, col3, col4, col5, col6])
   label = tf.one_hot(col1, 5, 1., 0.)


   return features, label
```

[3] comma separated values
[4] https://www.tensorflow.org/programmers_guide/reading_data#feeding

Listing 5.3: Prepare Data Set Library - create batch method

```
'''A code snippet of the prepare data set library. This method is derived from
    https://www.tensorflow.org/programmers_guide/reading_data#feeding'''

def create_batch(batch_size, num_epochs=None):
   filename_queue = tf.train.string_input_producer(
      filename_list, num_epochs=num_epochs, shuffle=True)
   example, label = get_data_example(filename_queue)

   min_after_dequeue = 100
   capacity = min_after_dequeue + 3 * batch_size
   example_batch, label_batch = tf.train.shuffle_batch(
      [example, label], batch_size=batch_size, capacity=capacity,
      min_after_dequeue=min_after_dequeue)

   return example_batch, label_batch
```

Another method creates a 3D-tensor which can be fed into a RNN. Figure 4.4 indicates why a simple mini-batch created with the methods above is not sufficient. This is due to the reason that three dimensions are required. They arise from following three parameters: *batch_size*, *num_steps*, *length(example)*. Compared to Figure 4.4 *length(example)* is the length of $x_i, i = 0, \ldots, n-1$, *num_steps* is n, specifically the length of the sequence, and *batch_size* represents the size of the mini-batch. The length of each x_i has to be identical in every batch. On the other hand, the sequences can be of different lengths, meaning they can hold different *num_steps* values. However, if different sequence lengths occur, they have to be padded with zero-vectors. Hence, several data examples with different sequence lengths can be stacked together to form one mini-batch.
The training data has to be given either in different csv-files or in one csv-file with extra labels for the *begin* and the *end* of a sequence. When working with different csv-files, each of them has to contain one sequence, meaning each row contains one vector of the sequence. The last element of the sequence is equipped with a label. First, the length of each sequence has to be checked. Sequences with different lengths get padded with zero-vectors. Afterwards, all tensors created are stacked together to one 3D-tensor. However, there is one problem concerning the label tensor. As the label is only existing in the last row, we do not want all default values listed in the label tensor. Hence, only the last value is used. When using one csv-file which contains all sequences, each sequence has to be extracted separately. This is done by looking for *begin* and *end* statements. The next steps are equal to the one described above.
Padding with zero-vectors does not affect the training step. The training algorithm can be provided with the real sequence lengths, meaning the lengths without padded zero-vectors. Thus, there is no influence of the zero-vectors on the training outcome.

5.2.4 Feedforward Neural Network

The data used to train a FFNN needs to be a tuple of *(input data, corresponding label)* (see Section 2.1.3.1). That is why two variables are required, one for the input data and one for the label.
Algorithm 1 shows the corresponding pseudocode to Figure 4.3. This recursive algorithm requires a weight W_i and a bias B_i. Both are elements of their corresponding lists W and B which contain all weights and biases of the neural network, respectively. The activation function is denoted with a, x represents the input data. This method works recursively. It starts at the output of the last hidden layer of the neural network and ends at the input nodes. Consider Figure 2.9 to visualize this principle. The computation starts from the last hidden layer at the right side and ends at the input layer on the left side. Recursively, the method decrements the index i until it reaches *0*, which ends the recursion and starts the computation. The output is then fed into an output activation to obtain the predicted output of the nerual network.

Algorithm 1 Compute the output of the last hidden layer recursively.

```
function COMPUTE_HIDDEN_OUTPUT(W_i, B_i)
    if i == 0 then
        return a(x · W_i + B_i)
    else
        return a(COMPUTE_HIDDEN_OUTPUT(W_{i-1}, B_{i-1}) · W_i + B_i)
    end if
end function
```

5.2.5 Deep Belief Network

Since a DBN consists of several stacked Restricted Boltzmann Machines (RBMs) (compare Section 4.3.2) a training procedure for both the whole DBN and a single RBM is required.
To initialize a DBN the input size and the number of hidden layers are obtained from the configuration file. Then, a list which contains every RBM in the right order is created and the RBMs are trained. The first one with the input data, the next one with the output of the first one and so on.
Since a DBN is often used as a pre-training step for a classifier which is stacked on top of the last RBM, possibility to do so is provided. Thereby, the input size to the classifier corresponds to the number of units in the hidden layer of the last RBM.

5.2.6 Recurrent Neural Network

A RNN is implemented using Long Short-Term Memory (LSTM) cells (see Figures 2.21, 2.22). There is also the opportunity to stack several LSTM cells. The data fed into a RNN needs to be of a 3D-structure of shape $[size\_mini\_batch, num\_steps, feature\_size]$ (see Section 5.2.3). The implementation is straightforward using the functions provided by the TensorFlow library. Nevertheless, there is one important aspect to mention. An unfolded RNN produces one output every time step i, $i = 0, \ldots, num\_steps - 1$ (see Figure 2.19). We are, however, only interested in the output of the last step (compare Figure 4.4). This is due to the reason, that we want to predict the next step or classify the input sequence. Listing 5.4 shows the two lines of code needed to get the last output. The first part transposes the output tensor of the LSTM cell called *state_series*, the second part takes the last element of it.

Listing 5.4: Get the last output

```
'''A code snippet of the RNN class used to get the last output'''

# from [size_mini_batch, num_steps, cell_size] to [num_steps, size_mini_batch,
    cell_size]
transformed_output = tf.transpose(states_series, [1,0,2])

# take the last element of the transformed states_output
last_output = transformed_output[-1]
```

5.3 Example: MNIST Data Set

When starting with machine learning, *MNIST* is the counterpart to a *Hello World!* program when starting programming. The MNIST data set consists of 70.000 data points which are split into a training set (55.000), a validation set (5.000) and a test set (10.000) [16]. As explained in Section 2.1.3.12 the training set is used to train the weights and biases of a model, the validation set is used to change its hyperparameters accordingly and the test set is used on the trained model to get the overall accuracy. Each data example is composed of an 28×28 image containing the handwritten digit and a corresponding label. Figure 5.1 shows four different digits from the data set. The corresponding labels are $5, 0, 4$ and 1. These labels tell the learning algorithm which digit the corresponding image contains.
In the following, all three machine learning services are trained on the MNIST data set. Thereby, information on how the respective neural network is trained is provided in order to show the usability. Moreover, a figure is provided for every machine learning service to illustrate the development of both the loss function and the accuracy, or the development of the reconstruction error, while training progresses. Additionally, the

accuracy of a FFNN and a RNN on the test set is given. This is done to demonstrate the varying range of accuracies of neural networks trained on the MNIST data set.

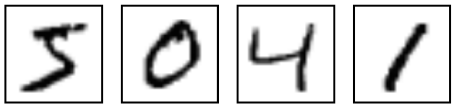


Figure 5.1: Example images from the MNIST data set of handwritten digits [15] [16].

5.3.1 Feedforward Neural Network

As mentioned in Section 2.1.3.1 a FFNN takes only vectors as input. This is why an image can not be fed directly to the network. Yet, to continue, we need to flatten the images to a vector of size 784 ($= 28 \times 28$). This way, information about the 2D-structure of the image is dropped. To keep the 2D-structure while training, a Convolutional Neural Network (CNN) has to be used (see Section 2.1.3.2). However, this is out of scope here since we do not implement such a neural network.
Training our FFNN service as designed in Section 4.3.1 works the following way. Listing 5.5 shows the training loop of the FFNN implementation which consists of two nested loops. The outer loop iterates over the training epochs and the inner loop iterates over the whole data set with step size being the size of the mini-batch. In each iteration, mini-batches of both the input data and the corresponding labels are provided to the feed-dictionary (*feed-dict*). After calling *sess.run()* the algorithm executes one training step (*train_step*), which is determined in advance according to the configuration file.

Listing 5.5: Training loops

```
''' A code snippet of the training loops '''

for epoch in range(training_epochs):
  for start, end in zip(range(0, len(train_data), size_mini_batch),
                range(size_mini_batch, len(train_data), size_mini_batch)):

    batch_xs = train_data[start:end]
    batch_ys = train_labels[start:end]

    sess.run(train_step, feed_dict={self._input_x: batch_xs, y_: batch_ys})
```

When using one's own training data, the data needs to be prepared first in order to be feedable to the feed-dictionary using the helper library described in Section 5.2.3.
The service trains the model using the code, more specifically the methods, shown in Listing 5.6. The first line initializes the neural network class. Afterwards the con-

figuration file is built. Next, a question which requires an answer is directed to the user (see Listing 5.7). This is due to the fact that he needs time to change the created configuration file. Additionally, the path to the configuration file is displayed. The name of the configuration file can be changed before starting training. Every hyperparameter and parameter listed in this file contains a default value but there are parameters which need to be changed. In this case, these are the input size and the output size. Thus, we choose an input size of 784 which corresponds to the length of the input vector described above. The output size is set to 10, which represents the classes $0, \ldots, 9$.

Listing 5.6: Train a model

```
my_first_nn = DeepFeedforwardNeuralNetwork()
my_first_nn.build_ini_file()
change = input('Have you changed the desired parameters and are ready to train
    the neural network? (Y) \n')
if change == "Y":
  my_first_nn.build_model()
  my_first_nn.train_model()
```

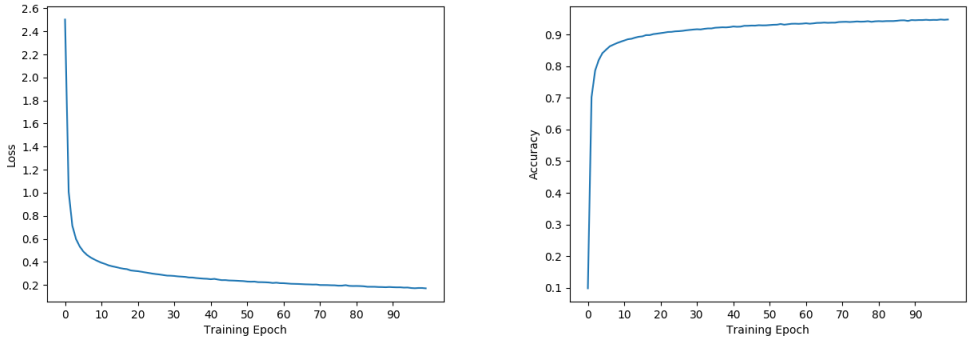
Listing 5.7: Console output

```
Destination of config file at
    /home/markus/deepLearningProjects/machine_learning_service/MyFirstNeuralNetwork.ini
Have you changed the desired parameters and are ready to train your first
    neural network? (Y)
```

The hidden layers are further changed to 500, 300 and 100 units and the number of training epochs to 100. After entering *Y*, the neural network starts training. Figure 5.2 depicts the graphs showing the development of the loss function and the accuracy during increasing training epochs. After 10 training epochs, the loss and accuracy increase and decrease dramatically, respectively. An accuracy of 94.0% on the test data set is reached which is sufficient but not good. This is due to the reason that we are using a simple feedforward model. As it can be seen in Section 5.3.3, a RNN can reach an even better accuracy. Using a CNN increases the accuracy further as it can process the 2D-structure of an image. The best models reach an accuracy of over 99.7% [80].

5.3.2 Deep Belief Network

As a DBN makes use of unsupervised learning, only the images need to be provided in order to let the respective service train the model (see Section 2.1.3.3). The training loop looks almost the same as in Listing 5.5 but with different computation steps. In Figure 5.3 the decay of the reconstruction error of a DBN with 6 hidden layers with



(a) Loss (b) Accuracy

Figure 5.2: The development of the loss function and the accuracy during the training phase of a FFNN.

600, 500, 400, 300, 200, 100 units trained on the MNIST data set is shown. Pre-training is applied to train every RBM on its own using the output of the previous RBM as input (see Figure 2.18). Each RBM is trained for 20 training epochs. The peaks at every 20-th training epoch indicate the training of the next RBM which slightly rises the reconstruction error. We can further stack a supervised classifier on top of the DBN to classify the images with the output of the DBN, meaning the output of the last RBM, as input.

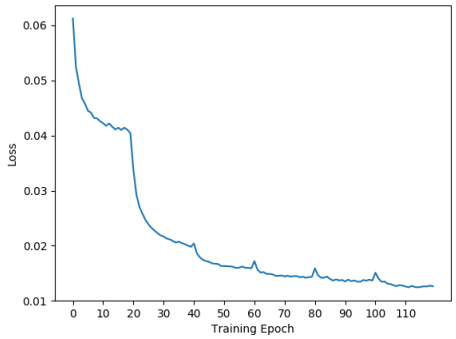
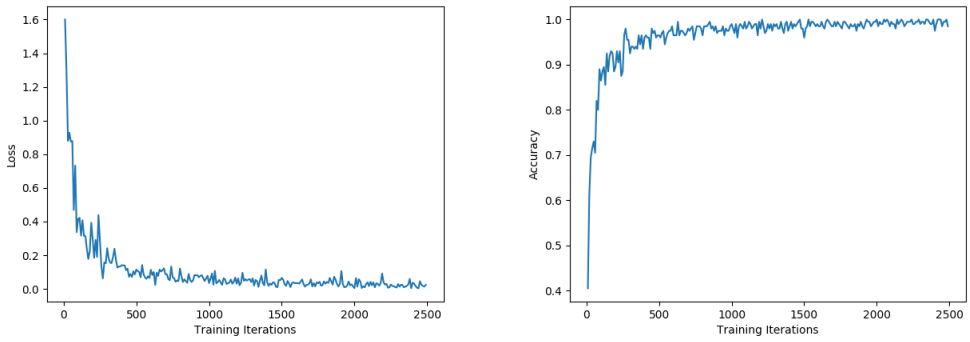


Figure 5.3: Decaying reconstruction error of a DBN built-up by stacking 6 RBMs.

5.3.3 Recurrent Neural Network

Training a RNN on the MNIST data set is slightly different, since sequences need to be provided as input. Section 2.1.3.4 describes why the data set has to be transformed. The RNN can not be fed with a vector of size 784. This is due to the reason that a RNN gets unfolded over n time steps. Therefore, the original vector is transformed into 28 vectors

of size 28, yielding $n = 28$ time steps. To be consistent with Section 5.2.3 we denote the time steps with *num_steps* which is the sequence length and the vector size with *length(example)*. Figure 5.4 depicts the development of the loss function and the accuracy when training a RNN built-up by one LSTM cell with a cell size of 200. The accuracy on the test data set is 98.46% which is better than the accuracy the Deep Feedforward Neural Network (DFFNN) could achieve. In comparison to Figure 5.2 the graphs shown in Figure 5.4 are not that smooth. This is due to the fact that the RNN evaluates the loss function and accuracy at every training iteration, whereas the previous FFNN does the same at every training epoch.



(a) Loss (b) Accuracy

Figure 5.4: The development of the loss function and the accuracy during the training phase of a RNN. Both, loss and accuracy were taken every training iteration.

Chapter 6

Evaluation

This chapter focuses on the evaluation of the three different machine learning services designed in Chapter 4 and implemented in Chapter 5. The evaluatiion starts using two data sets similar to the data sets mentioned in Section 3 and one additonal data set. Additionally, we analyze the service approaches using the MNIST data set introduced in Section 5.3. The results are compared against a regular implementation of the respective neural network. Afterwards, we conduct a performance analysis of the three neural network services and their regular counterpart implementation. We focus thereby on the training time and running time. This produces results for analyzing the usability and reusability of the machine learning services. We conclude with a qualitative evaluation including experience with the concept, reusability and usability. This evaluation is conducted with the knowledge and experince gained from the previous qualitative evaluations.

6.1 Quantitative Evaluation Results using different Data Sets

This section focuses, on the one hand, on the comparison of our approaches to the respective neural networks mentioned in the related works (see Section 3). On the other hand, we consider the time for implementing the particular use case which includes, for instance, the providing and pre-processing of the respective data set. The time for creating the neural network measures the time needed to set up the network computations. Both times are estimated from the viewpoint of a machine learning expert who is familiar with the respective machine learning library. Hence, the time taken for unexperienced users is considerably longer. One issue to mention is that we are not able to compare our approach to the related works directly as we do not get all information needed to set up the exact neural networks. This section concludes by comparing the three neural network services to their regular counterpart implementation using the MNIST data set of handwritten digits. This is due to the fact that we conduct a per-

formance analysis afterwards on the same data set. The MNIST data set is taken as a ground truth as it is a common data set used in machine learning.

6.1.1 ADL Data Set

The Center of Advanced Studies in Adaptive Systems (CASAS) smart home project provides several data sets representing Activities of Daily Living (ADL). The following evaluation is conducted using a data set containing 13 activities performed by two persons. As we are not able to get the identical data set used in the approach described in Section 3.1.5 we use a similar one. Furthermore, there is no response to our request on how they prepare their data. Thus, our results do not compare to the ones in [33]. We use the following features to represent the data (see Section 3.1.5).

- Day of the week (0 - 6)
- Hour of the day (0 - 23)
- Triggering sensor (0 - 9, according to the room the sensor is installed)
- Previous activity (0 - 12)
- Number of sensors activated during the activity
- Duration of the activity in minutes
- Label of the current activity (0 - 12)

Besides, the data is normalized. Further pre-processing is necessary to get better results. The data consists of 3741 data points.

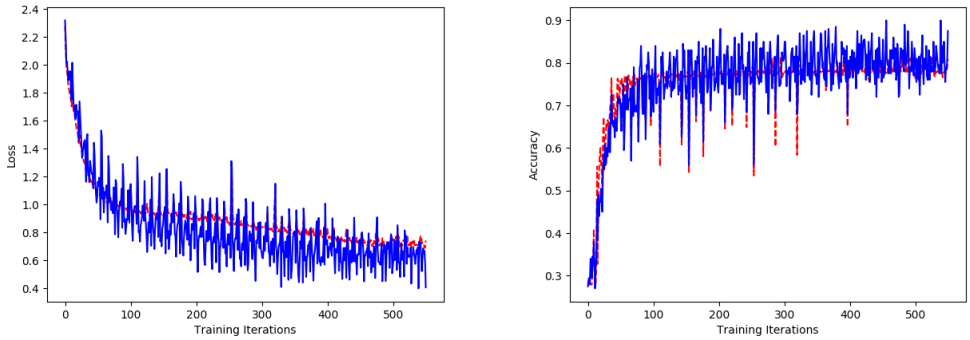
6.1.1.1 Deep Belief Network

We set up our service approach using a Deep Belief Network (DBN) with the information available in [33]. Thus, the DBN consists of stacked Restricted Boltzmann Machines (RBMs) with following sizes of the respective hidden layers: 500, 300, 100. The approach ends training when a certain threshold, involving the reconstruction error, is reached, namely 0.001. However, using their set-up we are not able to reach this goal. This is due to the reason that we do not really know how to pre-process their data, since we do not receive an answer to our request on this topic.
We try various alterations of a DBN, i.e. different hyperparameters and parameters, to get a smiliar reconstruction error. The best reconstruction error we obtain is 0.049 which yield an accuracy on our test set of 28.51%.
Both the implementation of the use case (ca. 30 s) and the creating of the neural network do not take much time (ca. 30 s). This is due to the reason that our service approach

provides the respective neural network structure and hence allows us to train various neural networks with different configurations easily.

6.1.1.2 Feedforward Neural Network

As the DBN does not perform well on this data set, we try a Deep Feedforward Neural Network (DFFNN). It is composed of three layers with 500, 300 and 100 units, respectively. We set the learning rate to 0.05 and use tanh as activation function. Simple Gradient Descent (GD) was applied. The size of the mini-batch is 200 and the training lasts for 100 training epochs.
The total training time was 4434.89 s. However, the DFFNN yields an accuracy of 82.91% on the test set. The loss of the validation set is reduced to 0.62. On the other hand the loss on the training set is further reduced to less than 0.40. Both values are not the best but we can demonstrate that by applying a different Artificial Neural Network (ANN), a data set which does not fit one network (e.g. DBN) can yield a better performance on another network (e.g. DFFNN). Figure 6.1 shows both the loss of the training and validation set and the accuracy of the training and validation set. There, one can notice the broad distribution of the error and the accuracy while training. Deploying other configurations of a DFFNN might narrow this broad distribution further.
By using our service approach, it is easy to apply other configurations to the neural network, and hence training the same network with different configurations does not take much time. Both implementing the use case and creating the neural network takes 30 s. Moreover, using another neural network takes not much effort since only the respective machine learning service needs to be called.



(a) Loss (b) Accuracy

Figure 6.1: Two graphs representing the loss (6.1(a)) and the accuracy (6.1(b)). Both were taken every training iteration. The blue, continuous line indicates the training set performance and the red, dashed line denotes the performance on the validation set.

6.1.1.3 Recurrent Neural Network

As the ADL data set consists of sequences of activities it is possible to train a Recurrent Neural Network (RNN) on it. Every activity consists of several time-series vectors which describe the particular activity. The number of vectors represents the number of unfolding steps of the RNN. As every activity does not have the same number of vectors we have to pad them with zero vectors. There, another problem ocurrs. Each sequence length varies widely. The smallest has a length of about 10, whereas the largest has a length of above 9000. Due to the reason that the vast majority does not have such a large length, we make only use of smaller lengths, i.e. sequences of length smaller than 500.
We do not really make good results on this data set. The training time is 12833.97 s after applying early stopping after iteration 254. The smallest error we achieve is 1.69 which yield an accuracy on the test set of 34.57%. This is due to the reason that we might use the wrong features and the sequence lengths vary too much. Several RNN configurations are considered.
The use case is implemented in 30 s. The creating of a new network takes about 30 s due to our modularization approach. This helps in considering a various amount of RNNs.

6.1.2 MIT Smart Home Data Set

In order to predict human activity we use the MIT smart home data sets MIT1 and MIT2. Both are also available in the CASAS smart home project. We prepare the data set our own way according to the information available in [34], since again our request about how they pre-process the data set exactly is not answered.
We used a sliding window size W of size 9, and a time interval T of size 5. If a sensor is activated in T its value is *1*, otherwise its *0*. Hence, the prediction of the next sensor value is based upon the previous 45 minutes (see Section 3.1.6). The data set consists of 20.088 data points.

6.1.2.1 Deep Belief Network

As given in [34] we create a DBN built-up by stacked RBMs, using the respective machine learning service. The hidden layer sizes are 200 and 100. The outcome of the DBN is fed into an ANN. During training we encountered following abnormality. When training the first RBM the reconstruction error is decreased to 0.009. However, training the next RBM increases the reconstruction error to 0.25. The output of the ANN stacked upon the DBN yield an accuarcy of 0.31%. This is due to the high reconstruction error of the second RBM. Figure 5.3 shows the correct decreasing of the reconstruction error of a DBN approach applied to the MNIST data set of handwritten digits.
Using the knowledge gained from above, we try another approach in order to get better

results. Therefore, a single RBM consisting of 300 hidden units is applied. Above, a ANN is stacked. We receive a test accuracy of 99.74%. The RBM produced a reconstruction error of 0.004. However, these results are easy to obtain as the data set consists of only *0* and *1*. As [34] already mentioned, if a sensor is activated, it is activate for a extended period of time. On the other hand, if a sensor is deactivated, it is not active for a extended period of time. Thus, they introduce the Rising Edge Accuracy (REA) which represents the prediction of newly activated sensors. Nevertheless, we are not able to conduct a evaluation using REA due to the reason that we do not know how to construct the data set for it.
From the two significantly different results obtained using a DBN built-up by stacking two RBMs and a single RBM we realise that the pre-processing of a data set is highly influencing the construction of an ANN.
The time which is needed to implement both DBNs is about 30 s as we only need to pass the data points to the respective machine learning service. Creating the neural network itself is brief, too. In about 30 s the configuration file is changed to our purpose. The training time of the DBN with the two hidden layers (80.29 s) is higher than the training time of the single RBM - DBN (28.76 s).

6.1.2.2 Recurrent Neural Network

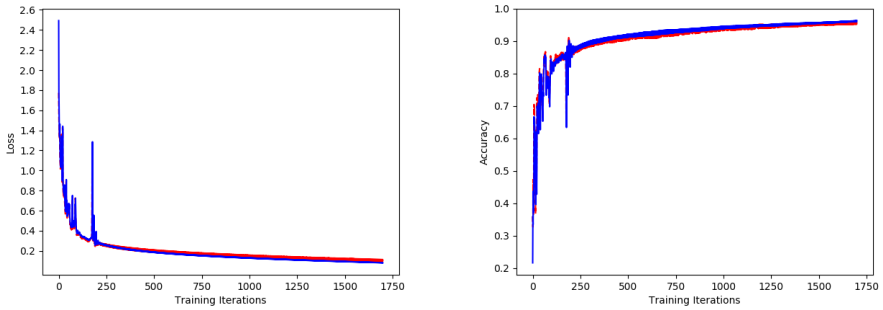
Additionally, we try to train our RNN service on the data set mentioned above. Every sliding window is used as a sequence, meaning that each sequence is the total number of sensors and the time steps are fixed to 9. An interesting fact in this case is the following. We are able to reach 100% at the beginning of the training. While training continues, the accuracy is decreasing. Moreover, the error was oscillating between a low and a high error value. Thus, after training, the accuracy yields a poor result. One reason for that is that the data set consists of almost only zeroes. At training start the labels consist of only zeroes, too, and hence outputting a zero vector yields an accuracy of 100%. Due to that reason this network is not suited for the respective pre-processed data.
However, we are able to try and train various RNNs with different configurations due to the service approach. Hence, the implementation of the use case lasts 30 s and the changing of the configuration file further 30 s.

6.1.3 Recognition Data Set

As both data sets above do not show satisfactory results, we apply another data set called *Smartphone-Based Recognition of Human Activities and Postural Transitions Data Set* [81]. The data consists of 561 features and labels one out of 12 activities. It is based upon different sensor values, e.g. accelerometer and gyroscope. Moreover, the features are normalized between -1 and 1.

6.1.3.1 Feedforward Neural Network

We use the Feedforward Neural Network (FFNN) service with three layers and 500, 300 and 100 hidden units, respectively. Furthermore, tanh is chosen as activation function and the learning rate is set to 0.08. Figure 6.2 shows the result of the training step. After 50 training epochs we are able to reach a test accuracy of 95.38% and the loss on the training set is reduced to 0.08.
Again, the implementation of this use case takes 30 s and the configuration file is changed in another 30 s.



(a) Loss (b) Accuracy

Figure 6.2: Two graphs representing the loss (6.2(a)) and the accuracy (6.2(b)). Both are taken every training iteration. The blue, continuous line indicates the training set performance and the red, dashed line denotes the performance on the validation set.

6.1.4 MNIST Data Set

Since a performance analysis is applied on the MNIST data set of handwritten digits, we use this data set to obtain a quantitative evaluation in terms of *latency*, *accuracy* or *reconstruction error*, *Lines of Code (LOC)* and both *time for implementing the use case* and *time for creating the neural network*. The latter describes the time required to set up the neural network computations. Each of our three machine learning services is compared to a regular implementation of the respective neural network. Each neural network pair is thereby initialized with the same parameters and hyperparameters. The results are shown in Tables 6.1, 6.2, 6.3.
As expected, the latency of the service is slightly higher than the latency of the regular implementation of the respective neural network. However, the difference between the two latencies is not worth mentioning. As the services as well as the regular neural networks are implemented with the same machine library the accuracy of both is almost the same. Considering the LOC a major difference between the service implementation and the respective regular implementation can be recognized. The small number of LOC of the service approach is due to the reason that the neural network structure and

	FFNN Service	FFNN
Latency	0.70 s	0.48 s
Accuracy	97.50 %	97.58 %
Lines of code	2	85
Implement use case	ca. 30 s	ca. 5 min
Create ANN	ca. 30 s (change configuration file)	ca. 2 min

Table 6.1: Evaluation of our machine learning service acting as a FFNN compared to a regular FFNN implementation.

	DBN Service	DBN
Latency	0.02 s	0.0003 s
Reconstruction error	0.02	0.02
Lines of code	2	148
Implement use case	ca. 30 s	ca. 8 min
Create ANN	ca. 30 s (change configuration file)	ca. 3 min

Table 6.2: Evaluation of our machine learning service acting as a DBN compared to a regular DBN implementation.

the learning algorithm are already implemented. The user needs to provide only the training data which is afterwards prepared for training. The regular neural network, however, has to be built-up from scratch, meaning all computations of the learning algorithm have to be implemented. This is why the time for both implementing the use case and creating the neural network is considerably higher than the particular times of the respective services. Another point to mention here is that the times of the regular implementations are estimated from the viewpoint of an expert in the area of machine learning. That is why users with almost no expert knowledge in both machine learning and the corresponding machine learning library need exceedingly more time. On the contrary, our three services do not require expert knowledge in these areas. Hence, the use case is implemented fast, sometimes one does not even need 30 s. Moreover, the changing of the configuration file does not take much time as it is structured in a plain way, i.e. all parameters and hyperparameters are clustered in sections which indicate the purpose of the respective parameters and hyperparameters.

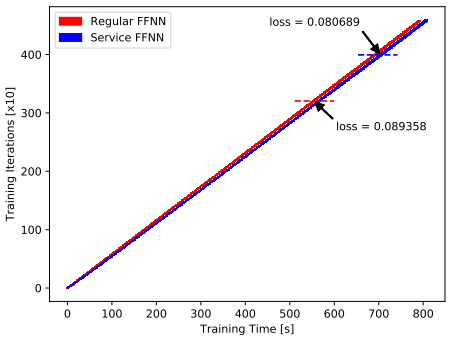
	RNN Service	RNN
Latency	2.07 s	1.78 s
Accuracy	98.50 %	98.44 %
Lines of code	2	128
Implement use case	ca. 30 s	ca. 10 min
Create ANN	ca. 30 s (change configuration file)	ca. 5 min

Table 6.3: Evaluation of our machine learning service acting as a RNN compared to a regular RNN implementation.

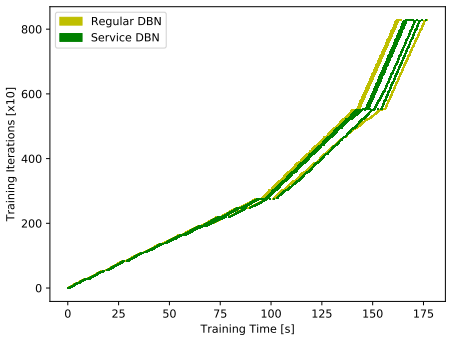
6.2 Performance Analysis

A performance evaluation is conducted on all three machine learning services and their regular counterparts. The evaluation is split up into two part. The first one measures the *training time* and the second one analyzes the distribution of the *running time*. The running time indicates the time needed to predict a new output after training is finished. Each net is trained 50 times anew. We ascertain the convergence points of each neural network before starting the process with the result that we do not train them for too many training epochs. The training epochs of the FFNNs are set to 25 and the ones of the RNNs to 4. The training epochs of each RBM are fixed to 10. Moreover, we use the same configuration in each neural network pair, i.e. the service network has the same parameter and hyperparameter values as the regular network. The respective training times are shown in Figure 6.3. We further depicted the mean convergence point including the corresponding loss value for both the FFNN and the RNN. A comparison of all the training times is shown in Figure B.1. The longest training time combined with the least iterations of the RNNs is due to the reason that a RNN has to backpropagate its error a certain number of time steps which is in our case 28 times. For comparison, a FFNN backpropagates its error the number of hidden layers which is in our case three times. A detailed representation of the training times at iteration 500 is given in Figure 6.4. Each subfigure depicts the training times measured at each of the 50 runs. It further shows the difference between the times of the service and the respective regular implementation.
As we presumed in the beginning, our implemented services train slightly slower than the regular approach. The FFNN service is in the end by an average of 14 s slower and the DBN service by an average of 6 s. This is due to the reason that our implementation is parameterized. This is why it needs to handle more requests during training, e.g. IF ... ELSE statements. Moreover, the service implementation involves a slightly higher latency as the regular implementation. Hence, the performance of our service in terms of training time is not as good as the one of the regular implementation but negligible as an enormous amount of time is saved by not having to set up the whole neural network

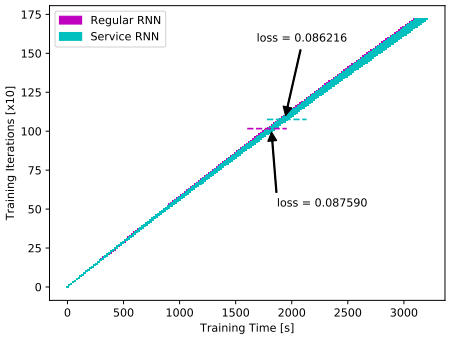
computations from scratch (see Section 6.1.4). Moreover, we figured out that the RNN service and the regular RNN implementation alternate by an average of above 20 s regarding the training time. The alternating times are shown in Figure 6.4(c). Figure 6.5 shows the difference of the training times of the respective runs. The figure further depicts the mean difference. It can be concluded that the training times of the service approach and the respective regular implementation do not differ much.



(a) FFNN



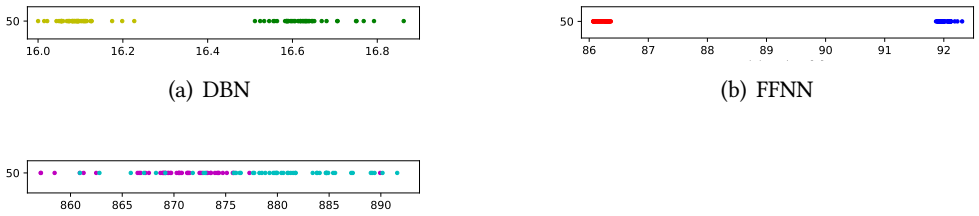
(b) DBN



(c) RNN

Figure 6.3: The training times of our approach and the corresponding regular implementation. Each training procedure is repeated 50 times. Furthermore, in Figure 6.3(a) and Figure 6.3(c) the mean convergence point including its corresponding loss value is depicted. Both bends in Figure 6.3(b) indicate the training of a new RBM. Figure 6.3(c) shows the overlapping training times of the RNN service and the regular RNN implementation. A more detailed representation of the training times of iteration 500 is shown in Figure 6.4.

The analysis of the running time is conducted the following way. Each neural network restores the saved trained model and computes a new output using the MNIST test set as input. This process is repeated 1000 times for each neural network. The regular implementations run slightly faster than the service approach. The RNN service, however, runs faster than the regular implementation. Figure 6.6 shows a comparison of the service implementation and the respective regular neural network. The individual



(a) DBN (b) FFNN

(c) RNN

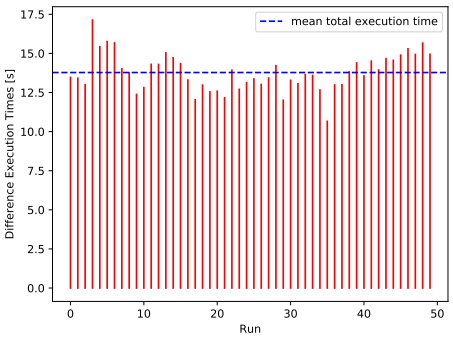
Figure 6.4: A more detailed representation of training iteration 500 showing the difference in the training times of the regular implementation and our service approach. Every data point indicates on run of the respective network.

running time distributions are presented in Figure B.3. A comparison of all the running times is shown in Figure B.2. The box thereby depicts the area, where the median 50% of the data points are located. Further, the median value is depicted by a horizontal line in the box.

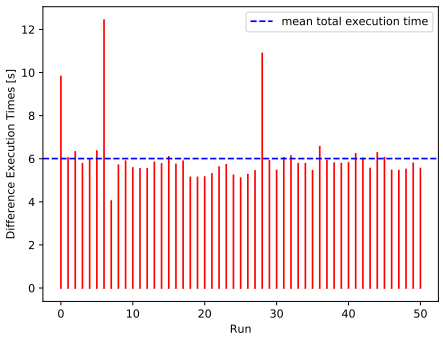
In summary, it can be stated out that both the training times and the running times of our service approach and the regular implementation do not differ that much. Nevertheless, the service approach provides one huge benefit. It saves an enormous amount of time as the user does not have to implement the whole neural network and its computations from scratch. This is due to the reason that the neural network of a service is created by only using the configuration file. Thus, by changing it, the user is able to train different neural networks with little effort. This yields a high usability. Moreover, the restoring of the network by means of the configuration file provides reusability efficiently.

6.3 Qualitative Evaluation Results

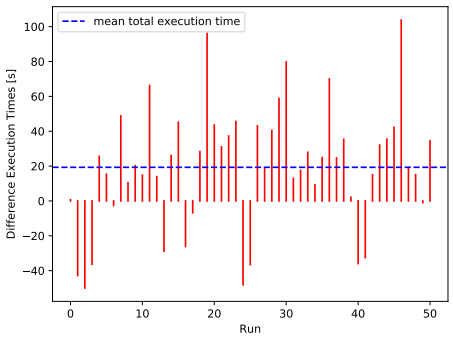
Table 6.4 represents a qualitative evaluation of the implemented machine learning services independent of a particular data set. We focus on experience with the concept, usability and reusability. The rating ranges from *0* to *+* to *++*. The first one indicates a neutral rating whereas the latter expresses a rather easy understanding of the particular concept. The evaluation is conducted by applying the gained experience and knowledge from the previous qualitative evaluations.
The experience with the concept is conducted from the perspective of a person who is not familiar with the matter. Both the handling of the particular services and the understanding and changing of the respective configuration files is relatively straightforward. The first one requires only the modifying of the neural network specific parameters, e.g. feature size, output size. The source code, on the other hand, is rather hard to understand as it requires pre-knowledge in machine learning and deep learning, respectively, as well as in TensorFlow library. As we aim at providing a machine learning



(a) FFNN



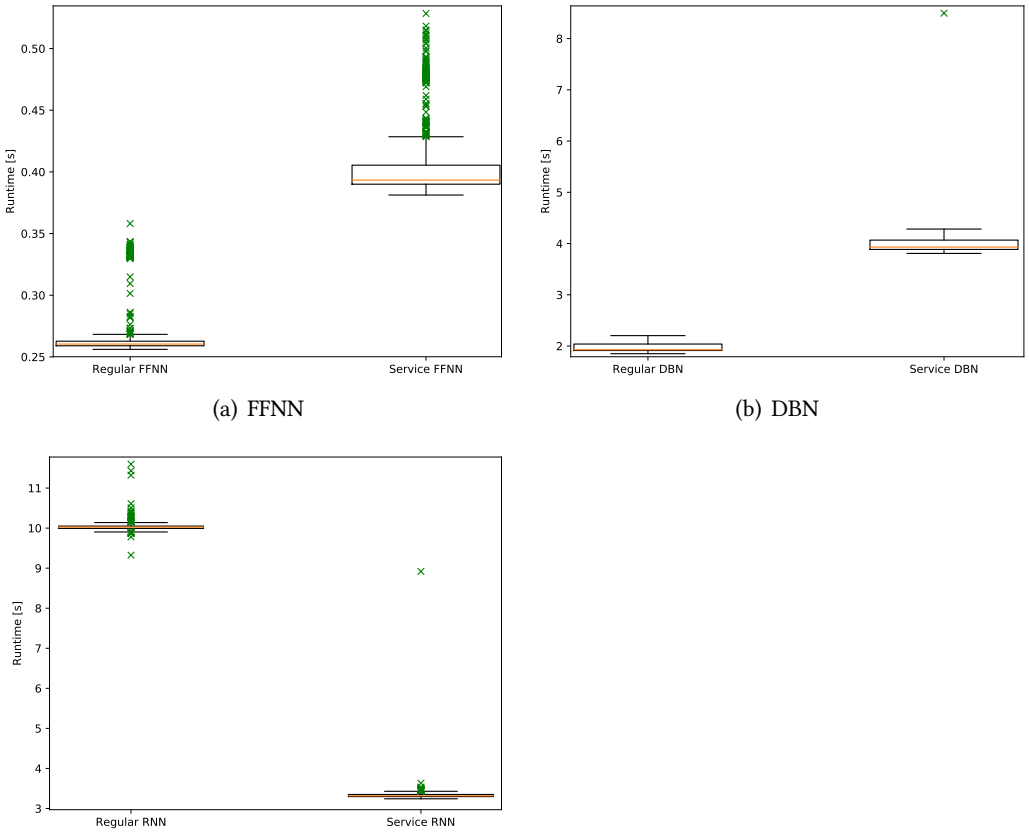
(b) DBN



(c) RNN

Figure 6.5: The difference in the training time between our approach and the corresponding regular implementation. Furthermore, the mean value of the difference in the training times is shown. In Figure 6.5(a) and Figure 6.5(b) our approach is always slightly slower than the regular implementation. In Figure 6.5(c), however, the difference in the training times alternates. A negative value indicates that the RNN service is faster than the regular implementation.

service for everybody no matter how much pre-knowledge one possesses, the source code is annotated as much as possible. This is why it can be followed along by everyone. However, if you want to modify the source code, which is not necessary since most functionality is already implemented, some pre-knowledge is required. The creation of a new neural network is relatively simple. The user only has to call the particular service and change the configuration file. The training of the created neural network requires some more action as data sets need to be provided to the respective prepare data set library (see Section 5.2.3). Due to the reason that the library file in turn requires slight modifications, it is easy to train a DBN which requires unlabeled input data, relatively easy to train a FFNN which requires labeled input data and rather hard to train a RNN, since a 3D-tensor needs to be built (see Section 4.3.3). Nevertheless, we provide useful methods and annotations which facilitate the preparing of the particular input data. As a user with no pre-knowledge shall be able to use the machine learning services, the



(a) FFNN

(b) DBN

(c) RNN

Figure 6.6: A detailed representation of the run time distribution of each neural network pair.

last evaluation point is important. Regarding this, some pre-knowledge helps but is not necessarily required.

Usability and reusability are evaluated according to Section 4.1. Both the FFNN service and the DBN service are easier to use as they are simple straightforward approaches. On the other hand, the RNN service is not as easy to use as the other two, since it has a recurrent loop and makes use of 3D-tensors as input, which requires both some understanding and further preprocessing of the data (see Section 5.2.3). Besides, the user only has to change a configuration file in order start training. Hence, the machine learning services are easy-to-use. Regarding reusability the FFNN service as well as the DBN service work well. This is due to the reason that state of the trained model is saved in a particular configuration file. The trained state of the model of the RNN service can also be saved after training. Due to an inital state vector which is applied in our implementation in order to improve usability, however, one is only able to feed data of exactly the size of the mini-batch. A good reusability is achieved since the state of the neural network is separated from the learning algorithm. This allows for efficient portability.

		FFNN	DBN	RNN
Experience with the concept	Easy - Hard to handle the service	++	++	++
	Easy - Hard to understand the code	+	0	0
	Easy - Hard to understand the service	+	+	+
	Easy - Hard to understand the configuration file	++	++	++
	Easy - Hard to modify the configuration file	++	++	++
	Easy - Hard to create a new ANN	+	+	+
	Easy - Hard to train the ANN	+	++	0
	Easy - Hard without in-depth ML/DL knowledge	+	+	+
Usability		++	++	+
Reusability		++	++	+

Table 6.4: Qualitative evaluation of the three implemented machine learning services. ML/DL represents thereby the terms machine learning and deep learning, respectively.

6.4 Summary

This chapter started with a quantitative evaluation in Section 6.1. Different data sets were thereby applied to analyze our machine learning service approaches. Two data sets were similar to the ones presented in the related works (see Sections 6.1.1, 6.1.2). However, we were not able to reach the same results as we got no answer to our request on how the authors of the corresponding related works have pre-processed their data. From there we could see that pre-processing of the available data set is very important. Furthermore, we tried one or another of our services with different parameters on these data sets. This did not take much time as only the configuration file of the respective service has to be changed in order to start training anew. The third data set applied was trained using the FFNN service and was without further pre-processing able to reach a good accuracy.
As the MNIST data set of handwritten digits is a well-known and often-used data set in machine learning we defined it as ground truth. Hence, a more detailed quantitative evaluation in terms of latency, accuracy or reconstruction error, LOC and time for both implementing the use case and creating the neural network was conducted. It showed that although the latency of the service implementations is slightly higher than the latency of the respective regular counterpart, this is negligible. This is due to the reason that the service approaches required much less LOC as the whole neural network structure is already implemented and only the data set needs to be provided in order to prepare it for training. Furthermore, the differences between the machine learning services and their regular counterpart in both the time for implementing the use case and the time for creating the neural network showed that each service requires a very small amount of time until it can start training. The times presented in the regular implementation are estimated from the view of a machine learning and deep learning expert, however. Hence, exceedingly more time is required if the user is not familiar

with the matter. Moreover, this evaluation demonstrated that the accuracies reached are almost identical. This is due to the fact that the same machine learning libraries are used.
A performance analysis was conducted afterwards in Section 6.2. It was split up into two part. The first part evaluated the training time. Each machine learning service was thereby trained 50 times anew. Although in case of the FFNNs and DBNs our service approach trained and ran slightly slower than the regular implementation, this difference in time is negligible. This is due to the reason that using our approach does not require implementing the whole neural network structure from scratch, and hence saves an enormous amount of time. The training times of the RNNs alternated, i.e. sometimes our approach was faster, sometimes it was slower. The second part measured the running time, meaning the time required to predict a new output after the network was already trained. It showed that the FFNN service and the DBN service are slightly slower than their respective regular implementations. On the other hand, the RNN service is considerably faster than its regular implementation. The compared results of both performance analysis are illustrated in Figures B.1, B.2.

To summarize, the three machine learning services showed a comparable performance in terms of training time and running time to regular implementations of the respective neural network (see Figures 6.3, 6.6). The slight deficit in both times is compensated by the time required to create a particular use case and the time needed to create a neural network. Moreover, the machine learning services can be used by users who are unexperienced in the area of machine learning. Furthermore, there is no need to be an expert in the respective machine learning library as the services provide the whole structure of the neural network and the learning algorithm. This can, for instance, be seen in the LOC mentioned in Tables 6.1, 6.2, 6.3. Only a few LOC are required in order to set up and train our approach. Additionally, reusability is enhanced since the state of the neural network, meaning its configuration, is separated from the respective learning algorithm. This allows for efficient portability. Since the configuration file contains all parameters and hyperparameters necessary to set up a neural network, the user can experimentally determine the appropriate configuration of the particular neural network yielding a high usability.

Chapter 7

Conclusion

The outcome of this thesis are three machine learning services. These services are easy-to-use since no pre-knowledge in machine learning and deep learning, respectively, is required. This is due to the reason that the learning algorithms were modularized into suitable building blocks. Each service provides another neural network. These networks were chosen according to both their occurrence in the related works and the capability to parameterize them. Hence, three services were implemented, each one using respectively a Feedforward Neural Network (FFNN), a Deep Belief Network (DBN) and a Recurrent Neural Network (RNN) (see Sections 3.3, 4.3). They further allow for efficient application in smart spaces.
The usability of the services is ensured by means of a configuration file. This file contains all hyperparameters and parameters necessary to set up and train a neural network (see Section 4.2). The user is able to adjust these values to his preferences. Moreover, due to this configuration file, the state of the neural network is separated from the learning algorithm itself. This allows for efficient portability since the neural network can be restored with the help of the respective configuration file.
The machine learning services are deployed as Virtual State Layer (VSL)-services in the Distributed Smart Space Orchestration System (DS2OS) (see Section 2.2). Each service is equipped with its own context model which acts as abstact interface to the real world (see Section 2.2.1). The nodes of a context model can hold different values. Each of the three services possesses almost the same context model. The most important nodes contain information about the *train-mode*, the destination of the *configuration file*, the *input* data, the *output* and the path to the *training data* (see Listing 4.1). The services apply two modes, one is used for training and one for application. By default, training mode is set. When in training mode, the service provides a configuration file to the user. The user can modify the file. A neural network is built-up by means of this file. The particular neural network is trained afterwards by applying the provided training data set. After successful training the learned parameters and hyperparameters of the model are saved in its configuration file and the train mode is changed. The parameters

contained in a configuration file were chosen according to Table 4.1.
The machine learning services were designed and implemented with respect to *usability* and *reusability* (see Section 4.1). The conducted evaluation showed that our service approaches consider both terms. The services were analyzed in terms of training time and running time against a regular implementation of the respective neural network (see Section 6.2). Comparable results were achieved. However, our service approach facilitates the experimental training of different neural network configurations since only the configuration file needs to be changed. Hence, no implementation of the respective neural network computations from scratch is required. We also showed that applying the configuration file ensures portability. Hence, the services are able to restore a model by using the respective configuration file.
In conclusion, the requirements specified in Chapter 2, designed in Chapter 4 and implemented in Chapter 5 were met. We realized three machine learning services which yield a good performance in terms of usability and reusability. Moreover, due to the modularization of the learning algorithms, these services allow users, which are unexperienced in the area of machine learning and deep learning, to train, evaluate and deploy a neural network. Furthermore, the three machine learning services are applicable in smart spaces.

7.1 Future work

The machine learning services make use of different Artificial Neural Networks (ANNs). Due to an abundance of possibilities to construct neural networks following ideas can be implemented in the future.

- Define different classifiers to stack upon the DBN
- Implement a DBN of stacked Autoencoders (AEs).
- Implement further machine learning services
- Implement additional training techniques, e.g. other cost functions, other weight initialization techniques, batch normalization
- Pre-train several neural networks

Furthermore, the service functionality can be extended.

- Implement the configuration file as a service, i.e. create a context model containing every parameter and hyperparameter of the corresponding neural network
- Implement an additional service which finds the optimal neural network configuration automatically

Appendix A

Further Configuration Files

We introduced the configuration file of a Feedforward Neural Network (FFNN) in Chapter 4. As we implemented two more Artificial Neural Networks (ANNs) the configuration files belonging to a Deep Belief Network (DBN) (see Figure A.1) and a Recurrent Neural Network (RNN) (see Figure A.2) are provided below.

```
[Neural Network]
type = Deep Belief Network
stacked_units = Restricted Boltzmann Machines
save_model_in = /path/to/save/model
[Input Layer]
feature_size = -1
[Output Layer]
output_size = -1
[Weight]
mean = 0.0
standard_deviation = 0.1
seed = 123
[Bias]
constant = 0.0
[Stacked RBMs]
out_units_rbm_1 = 100
out_units_rbm_2 = 100
out_units_rbm_3 = 100
number_of_stacked_RBMs = 3
[RBM]
learning_rate = 1.0
mini_batch = 200
epoch = 10
activation_fct_tanh = False
activation_fct_sigmoid = True
```

Figure A.1: An example of a configuration file used to initiate a DBN. The file contains the default values. It is necessary to change the feature size accordingly. Furthermore, one has to provide a path to save the model. To create a deeper model the number of RBMs can be extended in the respective section. If an ANN, e.g. a FFNN, is stacked on top of the DBN an output size is required.

```
[Neural Network]
type = Recurrent Neural Network
gated_recurrent_unit = Long Short-Term Memory
save_model_in = /path/to/save/model
[Input Layer]
feature_size = -1
[Output Layer]
output_size = -1
softmax_unit = True
no_activation = False
[LSTM]
activation_fct = tanh
lstm_size = 200
forget_bias = 1.0
stack_cells = False
number_of_stacked_layers = 1
num_steps = -1
[Weight]
mean = 0.0
standard_deviation = 0.1
seed = 123
[Bias]
constant = 0.0
[Cost Function]
cross_entropy = True
squared_errors = False
[Optimization Technique]
gradient_descent = False
momentum = False
adagrad_optimizer = True
[Additional Methods]
learning_rate_decay = False
early_stopping = False
k_cross_validation = False
[Hyperparameters]
activation_fct_tanh = True
activation_fct_sigmoid = False
activation_fct_relu = False
learning_rate = 0.8
mini-batch = 300
number_of_training_epochs = 100
momentum_rate = 0.8
p_keep = 0.75
wc_factor = 0.6
lr_decay_step = 100000
lr_decay_rate = 0.96
early_stopping_rounds = 100
early_stopping_metric_loss = True
early_stopping_metric_accuracy = False
validation_k = 10
[Parameters]
display_step = 10
```

Figure A.2: An example of a configuration file used to initiate a RNN. The file contains the default values. It is necessary to change the feature size, the output size and the number of time steps accordingly. Furthermore, one has to provide a path to save the model. To create a deeper model the number of LSTM cells can be extended in the respective section.

Appendix B

Compared Training Times and Running Times

All training times are shown in Figure B.1. It can be seen that the Recurrent Neural Networks (RNNs) are the slowest and the Deep Belief Networks (DBNs) the fastest approaches when training. Besides, the DBNs have the most iterations whereas the RNNs have the fewest of all. Moreover, Figure B.1 shows that in cases of the Feedforward Neural Networks (FFNNs) and DBNs our service approach is slightly slower than the regular implementation. On the other hand, comparing our RNN approach to the regular approach we can see that their training times alternate, i.e. sometimes our approach is faster, sometimes the regular implementation.
The running times are depicted in B.2. A more detailed representation of the services and the respective neural networks is shown in B.3. Each running time distribution is depicted solely.

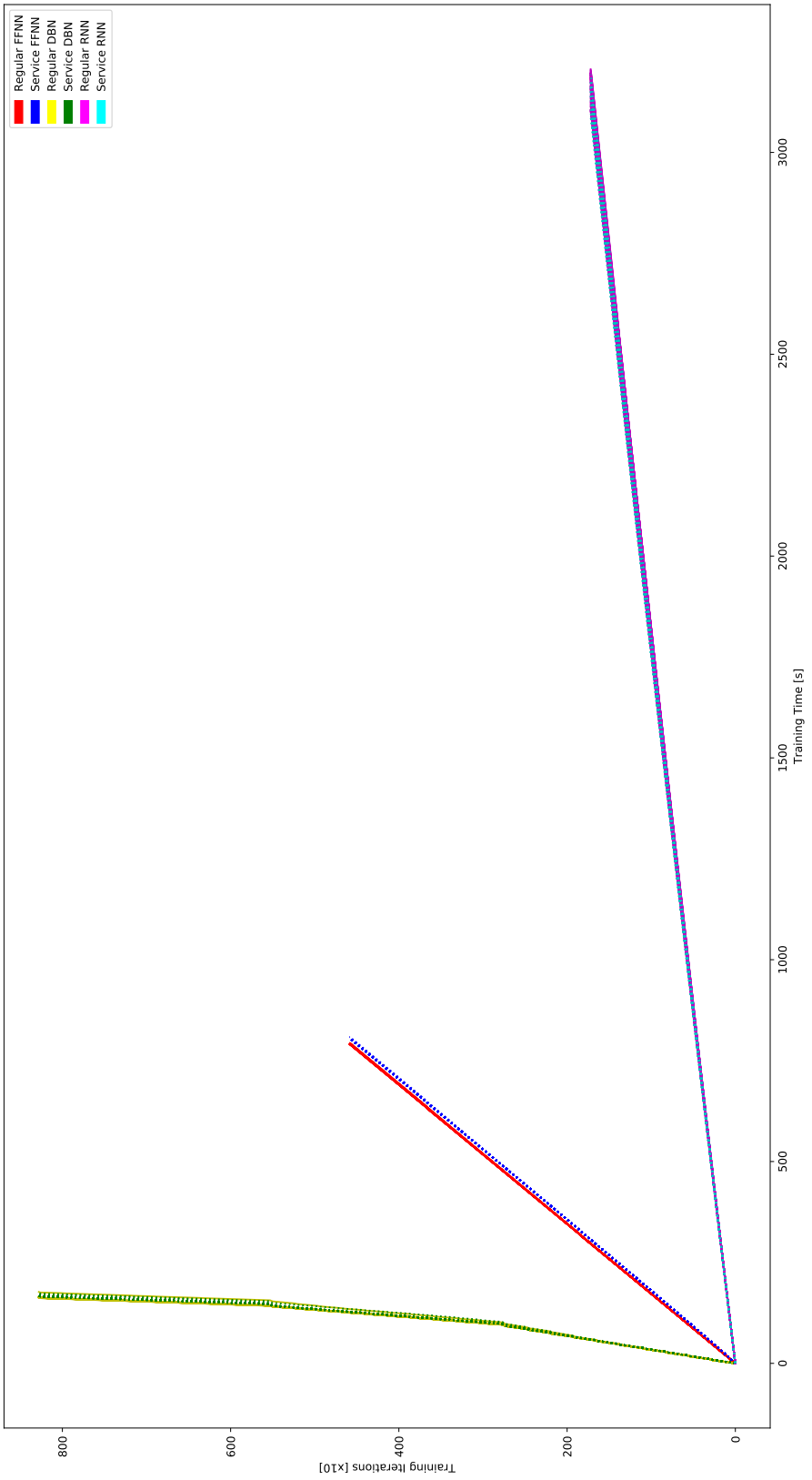


Figure B.1: A comparison of all training times. The DBNs have the least training time but the most iterations whereas the RNNs have the least iterations and the longest training time.

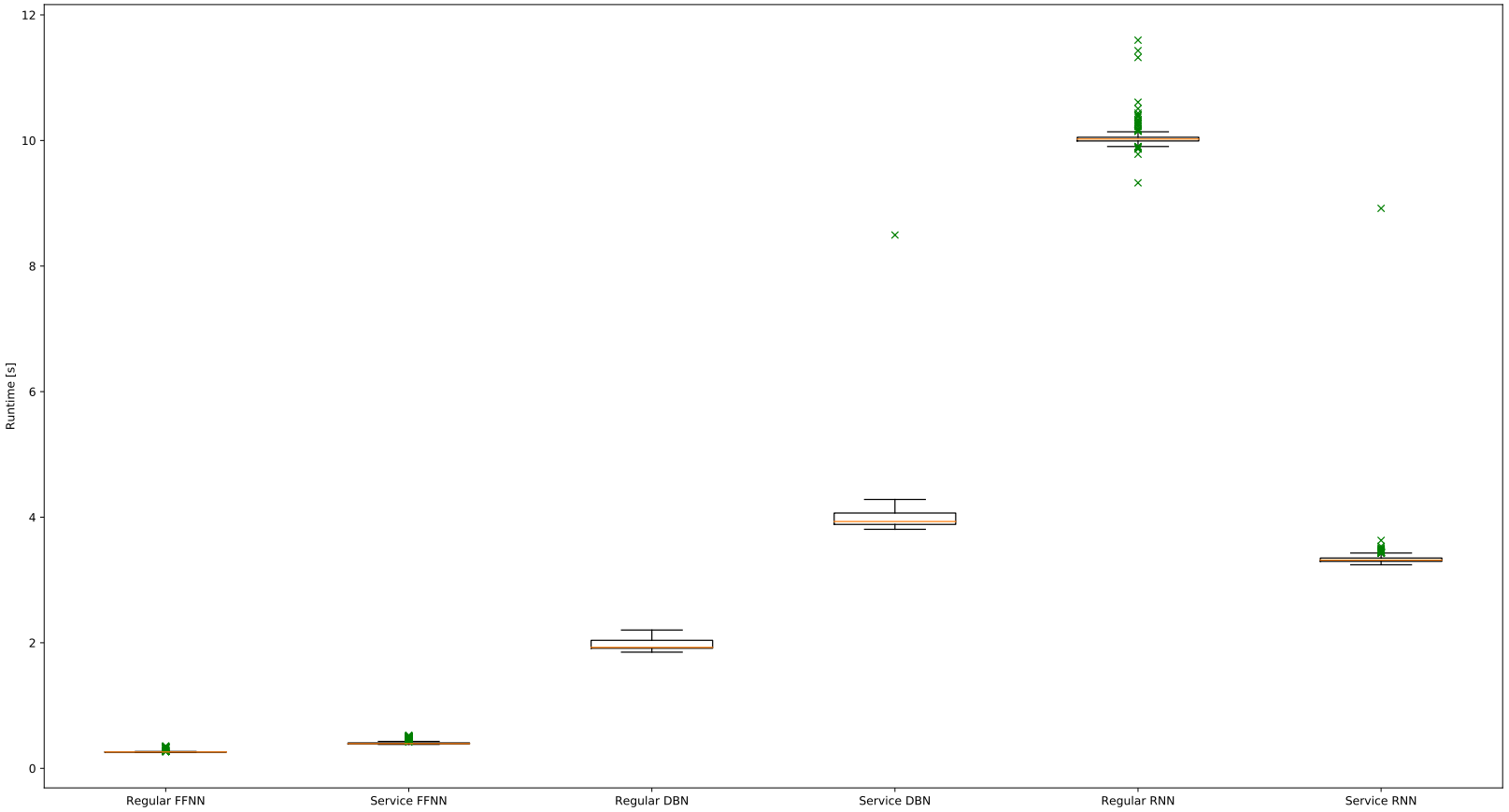
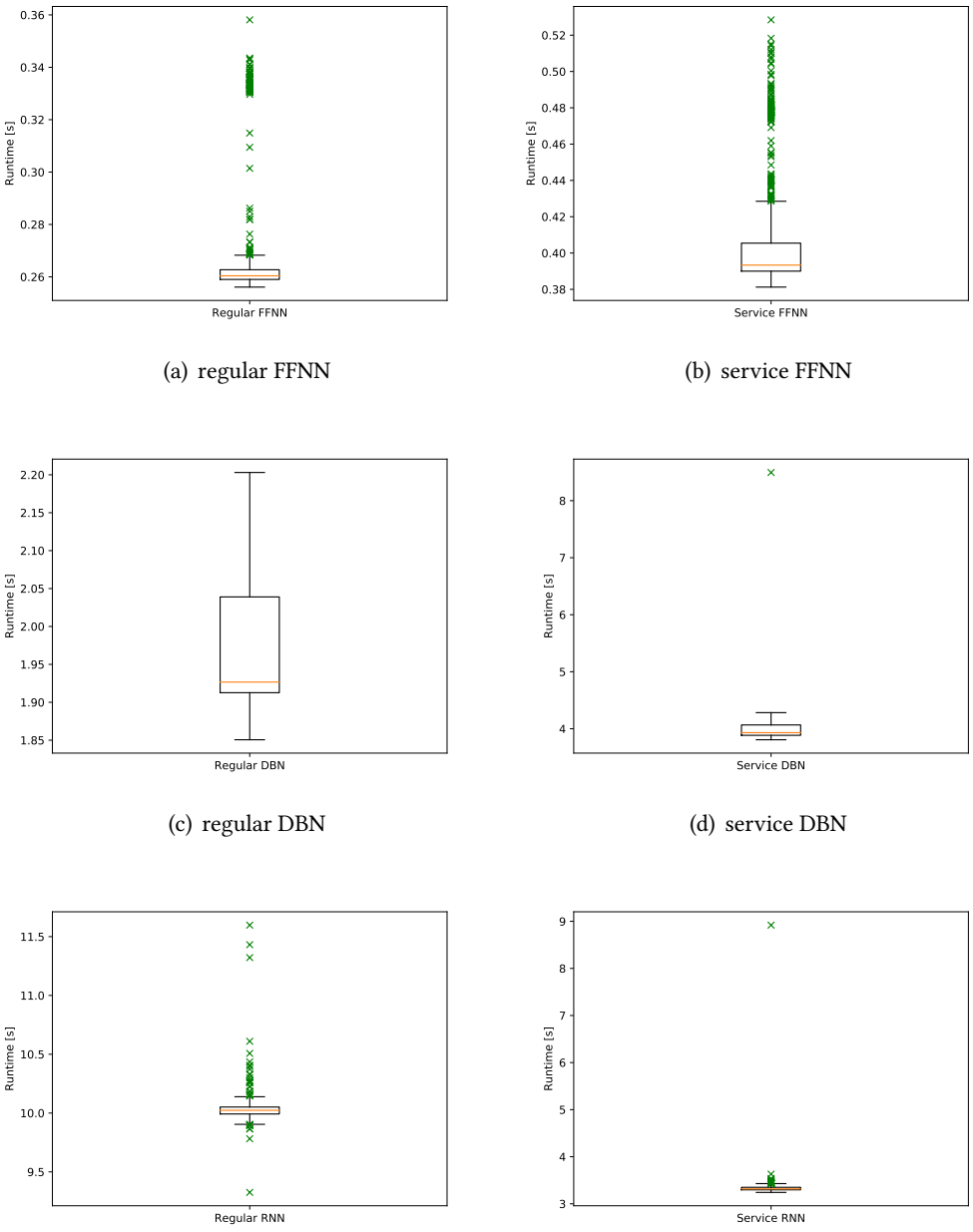


Figure B.2: A comparison of all running times. Except for the RNNs our service approaches run slightly slower than the regular implementation. However, our RNN approach runs considerably faster than the regular implementation.



(a) regular FFNN

(b) service FFNN

(c) regular DBN

(d) service DBN

(e) regular RNN

(f) service RNN

Figure B.3: A box plot showing the distribution of the running times of each neural network. A cross marks an outlier, and the horizontal line in a box marks the median running time. Each running process is repeated 1000 times.

Bibliography

[1] N. Jones. (2014) Computer science: The learning machines. [Online]. Available: http://www.nature.com/news/computer-science-the-learning-machines-1.14481

[2] I. J. Goodfellow, D. Warde-farley, M. Mirza, A. Courville, and Y. Bengio, "Maxout networks," in *Proceedings of the thirtieth International Conference on Machine Learning*, Atlanta, Georgia, USA, 2013.

[3] S. Ray. (2015) Understanding support vector machine algorithm from examples (along with code). [Online]. Available: https://www.analyticsvidhya.com/blog/2015/10/understaing-support-vector-machine-example-code/

[4] T. Slaff. (2014) Trading the rsi using a support vector machine. [Online]. Available: https://www.linkedin.com/pulse/20141103165037-172934333-trading-the-rsi-using-a-support-vector-machine

[5] M. Nielsen. (2017) Neural networks and deep learning. [Online]. Available: http://neuralnetworksanddeeplearning.com/

[6] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. MIT Press, 2016, http://www.deeplearningbook.org.

[7] G. Udacity. (2017) Deep learning classroom. [Online]. Available: https://de.udacity.com/course/deep-learning--ud730/

[8] DeepLearning4J. (2016) Restricted boltzmann machine. [Online]. Available: https://deeplearning4j.org/restrictedboltzmannmachine

[9] WILDML. (2015) Recurrent neural networks tutorial, part 1 – introduction to rnns. [Online]. Available: http://www.wildml.com/2015/09/recurrent-neural-networks-tutorial-part-1-introduction-to-rnns/

[10] J. Guo, "Backpropagation through time," 2013.

[11] C. Olah. (2015) Understanding lstm networks. [Online]. Available: http://colah.github.io/posts/2015-08-Understanding-LSTMs/

[12] T. Matiisen. (2015) Demystifying deep reinforcement learning. [Online]. Available: http://neuro.cs.ut.ee/demystifying-deep-reinforcement-learning/

[13] M.-O. Pahl. (2016) Video "smart space orchestration – how to make the internet of things smart?". Youtube. Sophia Antipolis, France. [Online]. Available: https://youtu.be/4sxRaubBG4s

[14] Y. Chen, Z. Lin, X. Zhao, G. Wang, and Y. Gu, "Deep learning-based classification of hyperspectral data," *IEEE Journal of Selected Topics in Applied Earth Observations and Remote Sensing*, vol. 7, no. 6, pp. 2094–2107, June 2014.

[15] Tensorflow. (2017) Mnist for ml beginners. [Online]. Available: https://www.tensorflow.org/get_started/mnist/beginners

[16] C. J. B. Yann LeCun, Corinna Cortes. MNIST database. [Online]. Available: http://yann.lecun.com/exdb/mnist/

[17] W. He, D. Goodkind, and P. Kowal, "An aging world: 2015," in *International Population Reports, P95/16-1*. U.S. Government Publishing Office, Washington, DC: U.S. Census Bureau, 2016.

[18] G. Singla, D. J. Cook, and M. Schmitter-Edgecombe, "Recognizing independent and joint activities among multiple residents in smart environments." *J. Ambient Intelligence and Humanized Computing*, vol. 1, no. 1, pp. 57–63, 2010.

[19] Y. Lecun, Y. Bengio, and G. Hinton, "Deep learning," *Nature*, vol. 521, no. 7553, pp. 436–444, 5 2015.

[20] F. Rosenblatt, *The Perceptron, a Perceiving and Recognizing Automaton Project Para*, ser. Report: Cornell Aeronautical Laboratory. Cornell Aeronautical Laboratory, 1957.

[21] A. Ng, J. Ngiam, C. Y. Foo, Y. Mai, and C. Suen. (2013) Unsupervised feature learning and deep learning - neural networks. [Online]. Available: http://ufldl.stanford.edu/wiki/index.php/Neural_Networks

[22] X. Glorot, A. Bordes, and Y. Bengio, "Deep sparse rectifier neural networks," in *Proceedings of the Fourteenth International Conference on Artificial Intelligence and Statistics (AISTATS-11)*, G. J. Gordon and D. B. Dunson, Eds., vol. 15. Journal of Machine Learning Research - Workshop and Conference Proceedings, 2011, pp. 315–323.

[23] A. L. Maas, A. Y. Hannun, and A. Y. Ng, "Rectifier Nonlinearities Improve Neural Network Acoustic Models," in *Proceedings of the thirtieth International Conference on Machine Learning*, Atlanta, Georgia, USA, 2013.

[24] K. He, X. Zhang, S. Ren, and J. Sun, "Delving deep into rectifiers: Surpassing human-level performance on imagenet classification," *CoRR*, vol. abs/1502.01852, 2015.

[25] M. Haloi, "Improved microaneurysm detection using deep neural networks," *CoRR*, vol. abs/1505.04424, 2015.

[26] X. Zhu, A. B. Goldberg, R. Brachman, and T. Dietterich, *Introduction to Semi-Supervised Learning*. Morgan and Claypool Publishers, 2009.

[27] A. Ng, J. Ngiam, C. Y. Foo, Y. Mai, C. Suen, A. Coates, A. Maas, A. Hannun, B. Huval, T. Wang, and S. Tandon. Softmax regression. [Online]. Available: http://ufldl.stanford.edu/tutorial/supervised/SoftmaxRegression/

[28] C. Cortes and V. Vapnik, "Support-vector networks," *Machine Learning*, vol. 20, no. 3, pp. 273–297, 1995.

[29] Y. LeCun, P. Haffner, L. Bottou, and Y. Bengio, "Object recognition with gradient-based learning," in *Shape, Contour and Grouping in Computer Vision*. London, UK, UK: Springer-Verlag, 1999, pp. 319–. [Online]. Available: http://dl.acm.org/citation.cfm?id=646469.691875

[30] Y. Sun, X. Wang, and X. Tang, "Deep learning face representation from predicting 10,000 classes," in *Proceedings of the 2014 IEEE Conference on Computer Vision and Pattern Recognition*, ser. CVPR '14. Washington, DC, USA: IEEE Computer Society, 2014, pp. 1891–1898.

[31] W. Ouyang and X. Wang, "Joint deep learning for pedestrian detection," in *2013 IEEE International Conference on Computer Vision*, Dec 2013, pp. 2056–2063.

[32] J. Altosaar. (2017) Tutorial - what is a variational autoencoder? [Online]. Available: https://jaan.io/what-is-variational-autoencoder-vae-tutorial/

[33] H. Fang and C. Hu, "Recognizing human activity in smart home using deep learning algorithm," in *Proceedings of the 33rd Chinese Control Conference*, July 2014, pp. 4716–4720.

[34] S. Choi, E. Kim, and S. Oh, "Human behavior prediction for smart homes using deep learning," in *2013 IEEE RO-MAN*, Aug 2013, pp. 173–179.

[35] D. Erhan, Y. Bengio, A. Courville, P.-A. Manzagol, P. Vincent, and S. Bengio, "Why does unsupervised pre-training help deep learning?" *J. Mach. Learn. Res.*, vol. 11, pp. 625–660, Mar. 2010.

[36] Y. Bengio, P. Lamblin, D. Popovici, and H. Larochelle, "Greedy layer-wise training of deep networks," 2007, pp. 153–160.

[37] D. Learning. (2017) Deep belief networks. [Online]. Available: http://deeplearning.net/tutorial/DBN.html

[38] Y. Bengio, P. Simard, and P. Frasconi, "Learning long-term dependencies with gradient descent is difficult," *Trans. Neur. Netw.*, vol. 5, no. 2, pp. 157–166, Mar. 1994. [Online]. Available: http://dx.doi.org/10.1109/72.279181

[39] S. Hochreiter, Y. Bengio, P. Frasconi, and J. Schmidhuber, "Gradient flow in recurrent nets: the difficulty of learning long-term dependencies," 2001.

[40] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural Comput.*, vol. 9, no. 8, pp. 1735–1780, Nov. 1997. [Online]. Available: http://dx.doi.org/10.1162/neco.1997.9.8.1735

[41] K. Cho, B. van Merrienboer, Ç. Gülçehre, F. Bougares, H. Schwenk, and Y. Bengio, "Learning phrase representations using RNN encoder-decoder for statistical machine translation," *CoRR*, vol. abs/1406.1078, 2014.

[42] J. Chung, Ç. Gülçehre, K. Cho, and Y. Bengio, "Empirical evaluation of gated recurrent neural networks on sequence modeling," *CoRR*, vol. abs/1412.3555, 2014.

[43] R. Pascanu, Ç. Gülçehre, K. Cho, and Y. Bengio, "How to construct deep recurrent neural networks," *CoRR*, vol. abs/1312.6026, 2013. [Online]. Available: http://arxiv.org/abs/1312.6026

[44] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. A. Riedmiller, "Playing atari with deep reinforcement learning," *CoRR*, vol. abs/1312.5602, 2013.

[45] N. Qian, "On the momentum term in gradient descent learning algorithms," *Neural Networks*, vol. 12, no. 1, pp. 145 – 151, 1999. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0893608098001166

[46] R. Rojas, "Neural networks - a systematic introduction," *Springer-Verlag*, vol. 37, pp. 151 – 184, 1996.

[47] Y. Bengio, "Practical recommendations for gradient-based training of deep architectures," *CoRR*, vol. abs/1206.5533, 2012.

[48] G. E. Hinton, N. Srivastava, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, "Improving neural networks by preventing co-adaptation of feature detectors," *CoRR*, vol. abs/1207.0580, 2012.

[49] X. Glorot and Y. Bengio, "Understanding the difficulty of training deep feedforward neural networks," in *In Proceedings of the International Conference on Artificial Intelligence and Statistics (AISTATS'10). Society for Artificial Intelligence and Statistics*, 2010.

[50] S. Ioffe and C. Szegedy, "Batch normalization: Accelerating deep network training by reducing internal covariate shift," *CoRR*, vol. abs/1502.03167, 2015. [Online]. Available: http://arxiv.org/abs/1502.03167

[51] J. Kuhn. (2017) Batch normalization. [Online]. Available: https://wiki.tum.de/display/lfdv/Batch+Normalization

[52] E. Ackerman. (2017) How drive.ai is mastering autonomous driving with deep learning. [Online]. Available: https://spectrum.ieee.org/cars-that-think/transportation/self-driving/how-driveai-is-mastering-autonomous-driving-with-deep-learning

[53] S. H. Hsu, M.-H. Wen, H.-C. Lin, C.-C. Lee, and C.-H. Lee, *AIMED- A Personalized TV Recommendation System.* Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, pp. 166–174. [Online]. Available: http://dx.doi.org/10.1007/978-3-540-72559-6_18

[54] H. D. Mehr, H. Polat, and A. Cetin, "Resident activity recognition in smart homes by using artificial neural networks," in *2016 4th International Istanbul Smart Grid Congress and Fair (ICSG)*, April 2016, pp. 1–5.

[55] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng, "TensorFlow: Large-scale machine learning on heterogeneous systems," 2015, software available from tensorflow.org. [Online]. Available: http://tensorflow.org/

[56] zer0n, dwiel, junjieqian, and bamos. (2016) Evaluation of deep learning toolkits. [Online]. Available: https://github.com/zer0n/deepframeworks

[57] DeepLearning4J. (2017) Comparing frameworks: Deeplearning4j, torch, theano, tensorflow, caffe, paddle, mxnet, keras and cntk. [Online]. Available: https://deeplearning4j.org/compare-dl4j-torch7-pylearn#caffe

[58] M.-O. Pahl, G. Carle, and G. Klinker, "Distributed smart space orchestration," in *Network Operations and Management Symposium 2016 (NOMS 2016) - Dissertation Digest*, May 2016.

[59] M.-O. Pahl, G. Carle, and G. Klinker, "Distributed smart space orchestration," in *NOMS 2016 - 2016 IEEE/IFIP Network Operations and Management Symposium*, April 2016, pp. 979–984.

[60] M.-O. Pahl, "Distributed smart space orchestration," Ph.D. dissertation, Technische Universität München, München, jun 2014.

[61] A. Badlani and S. Bhanot, "Smart home system design based on artificial neural networks," in *Proc. of the Word Congress on Engineering and Computer Science*, 2011.

[62] C. A. Hernandez, R. Romero, and D. Giral, "Optimization of the use of residential lighting with neural network," in *2010 International Conference on Computational Intelligence and Software Engineering*, Dec 2010, pp. 1–5.

[63] A. Hussein, M. Adda, M. Atieh, and W. Fahs, "Smart home design for disabled people based on neural networks," *Procedia Computer Science*, vol. 37, pp. 117 – 126, 2014. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S1877050914009855

[64] H. Li, Q. Zhang, and P. Duan, "A novel one-pass neural network approach for activities recognition in intelligent environments," in *2008 7th World Congress on Intelligent Control and Automation*, June 2008, pp. 50–54.

[65] D. Li and S. K. Jayaweera, "Reinforcement learning aided smart-home decision-making in an interactive smart grid," in *2014 IEEE Green Energy and Systems Conference (IGESC)*, Nov 2014, pp. 1–6.

[66] H. Fang, L. He, H. Si, P. Liu, and X. Xie, "Human activity recognition based on feature selection in smart home using back-propagation algorithm," *{ISA} Transactions*, vol. 53, no. 5, pp. 1629 – 1638, 2014, {ICCA} 2013. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0019057814001281

[67] S. T. M. Bourobou and Y. Yoo, "User activity recognition in smart homes using pattern clustering applied to temporal ann algorithm," *Sensors*, vol. 15, no. 5, pp. 11 953–11 971, 2015. [Online]. Available: http://www.mdpi.com/1424-8220/15/5/11953

[68] M. C. Mozer, "The neural network house: An environment hat adapts to its inhabitants," in *Proc. AAAI Spring Symp. Intelligent Environments*, vol. 58, 1998.

[69] D. J. Cook, M. Youngblood, E. O. Heierman, K. Gopalratnam, S. Rao, A. Litvin, and F. Khawaja, "Mavhome: an agent-based smart home," in *Proceedings of the First IEEE International Conference on Pervasive Computing and Communications, 2003. (PerCom 2003).*, March 2003, pp. 521–524.

[70] G. E. Hinton, S. Osindero, and Y.-W. Teh, "A fast learning algorithm for deep belief nets," *Neural Comput.*, vol. 18, no. 7, pp. 1527–1554, Jul. 2006. [Online]. Available: http://dx.doi.org/10.1162/neco.2006.18.7.1527

[71] D. C. Ciresan, U. Meier, L. M. Gambardella, and J. Schmidhuber, "Deep big simple neural nets excel on handwritten digit recognition," *CoRR*, vol. abs/1003.0358, 2010.

[72] H.-I. Suk and D. Shen, *Deep Learning-Based Feature Representation for AD/MCI Classification.* Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 583–590.

[73] Alzheimer's Disease Neuroimaging Initiative. ADNI database. [Online]. Available: http://adni.loni.usc.edu/

[74] M. Gönen and E. Alpaydin, "Multiple kernel learning algorithms," *J. Mach. Learn. Res.*, vol. 12, pp. 2211–2268, jul 2011.

[75] D. Zhang and D. Shen, *Multi-Modal Multi-Task Learning for Joint Prediction of Clinical Scores in Alzheimer's Disease.* Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 60–67.

[76] X. Glorot, A. Bordes, and Y. Bengio, "Domain adaptation for large-scale sentiment classification: A deep learning approach," in *Proceedings of the 28th International Conference on Machine Learning (ICML-11)*, L. Getoor and T. Scheffer, Eds. New York, NY, USA: ACM, 2011, pp. 513–520.

[77] R. Fakoor, F. Ladhak, A. Nazi, and M. Huber, "Using deep learning to enhance cancer diagnosis and classification," in *Proceedings of the thirtieth International Conference on Machine Learning*, Atlanta, Georgia, USA, 2013.

[78] R. Raina, A. Battle, H. Lee, B. Packer, and A. Y. Ng, "Self-taught learning: Transfer learning from unlabeled data," in *Proceedings of the 24th International Conference on Machine Learning*, ser. ICML '07. New York, NY, USA: ACM, 2007, pp. 759–766.

[79] F. Kuperjans, "Native Service Interfaces for the Virtual State Layer," Bachelor's Thesis, Technische Universität München, 2017.

[80] R. Benenson. (2016) Classification datasets results. [Online]. Available: http://rodrigob.github.io/are_we_there_yet/build/classification_datasets_results.html

[81] J.-L. Reyes-Ortiz, L. Oneto, A. Samà, X. Parra, and D. Anguita, "Transition-aware human activity recognition using smartphones," *Neurocomputing*, vol. 171, pp. 754 – 767, 2016. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0925231215010930

Acronyms

ACHE Adaptive Control of Home Environment

AD Alzheimer's Disease

ADL Activities of Daily Living

ADAS-Cog Alzheimer's Disease Assessment Scale-Cognitive Subscale

AE Autoencoder

ALZ Active LeZi

ANN Artificial Neural Network

BP Backpropagation

BPTT Backpropagation Through Time

CASAS Center of Advanced Studies in Adaptive Systems

CD Contrastive Divergence

CMR Context Model Repository

CNN Convolutional Neural Network

CSF Cerebrospinal Fluid

DAE Denoising Autoencoder

DBN Deep Belief Network

DFFNN Deep Feedforward Neural Network

DS2OS Distributed Smart Space Orchestration System

ED Episode Discovery

FFNN Feedforward Neural Network

GD Gradient Descent

GRU Gated Recurrent Unit

HM-MDP Hidden Mode Markov Decision Process

HC Healthy Normal Controls

KA Knowledge Agent

KL Kullback-Leibler

LOC Lines of Code

LR Logistic Regression

LSTM Long Short-Term Memory

MCI Mild Cognitive Impairment

MCI-C Mild Cognitive Impairment-Converter

MCI-NC Mild Cognitive Impairment-Non-Converter

MK Multi-Kernel

MLP Multilayer Perceptron

MRI Magnetic Resonance Imaging

MMSE Minimum Mental State Examination

PCA Principal Component Analysis

PET Positron Emission Tomography

RAE Regularized Autoencoder

RBF Radial Basis Function

RBM Restricted Boltzmann Machine

REA Rising Edge Accuracy

ReLU Rectified Linear Unit

RNN Recurrent Neural Network

SAE Stacked Autoencoder

SDAE Stacked Denoising Autoencoder

SGD Stochastic Gradient Descent

SHIP Smart Home Inhabitant Prediction

SpAE Sparse Autoencoder

SR Softmax Regression

SSE Sum of Squared Errors

MSE Mean Squared Error

SVM Support Vector Machine

TMM Task-based Markov Model

VAE Variational Autoencoder

VSL Virtual State Layer