Artificial Neural Networks for Textclassification

Mathias Menzel Dennis Köhn Valentin Zambelli

Course: Textklassifikation als Wettbewerb

December 14, 2017

Contents

1 Introduction

Document classification is the task of automatically assigning predefined categories also called labels or classes to specific documents. This problem can be described as follows. Let the *training set* be consisting of the documents $D = (d_1, \ldots, d_N)$ and their corresponding labels $Y = (y_1, \ldots, y_N)$. Furthermore, let the documents $\tilde{D} = (\tilde{d}_1, \ldots, \tilde{d}_{N'})$ and their unknown labels $\tilde{Y} = (\tilde{y}_1, \ldots, \tilde{y}_{\tilde{N}})$ be the *test set*. Moreover, let each document be encoded by a numerical feature vector, such that $X = (\mathbf{x}_1, \ldots, \mathbf{x}_N)$ represents all documents of the *training set* and $\tilde{X} = (\tilde{\mathbf{x}}_1, \ldots, \tilde{\mathbf{x}}_{\tilde{N}})$ all documents of the *test set*. The goal of a document classifier is to find a function

$$f : X \to Y, \tag{1}$$

which correctly assigns as many of the unknown labels in the *test set* as possible, i.e.

$$f(\tilde{\mathbf{x}}_i) = \tilde{y}_i \tag{2}$$

should hold for many of the $\tilde{N}$ unlabeled documents. In this paper, we consider the model class of *artificial neural networks* (ANNs) to find a reasonable mapping f. Additionally, we discuss and empirically test different strategies to extract numerical feature vectors from raw documents.
The remainder of this paper is organized as follows. Section 2 gives an introduction to the basics of ANNs. Common strategies to represent raw text as numerical features are discussed in Section 3. It follows a brief overview of advanced ANN architectures that are used in our empirical analysis in Section 4. The setup of this analysis is described in Section 5 and results are presented in Section 6. Finally, the paper finishes with conclusive remarks Section 7.

2 Feed-forward Neural Networks

Inspired by the nature of the human brain, the idea of ANNs as a mathematical model started to be developed in the 1940s. Beginning with the modeling how neurons in the human brain might work in McCulloch & Pitts (1943) and the theory that the brain learns by strengthening neural connections between neurons over time in Hebb (1949), neuropsychologists laid the foundation for the machine learning method that is nowadays able to solve multiple problems by learning the relevant pattern.
The basic neural network model, the *perceptron*, was first mentioned in (Rosenblatt 1958). Its characteristics, extensions and the mechanism to train such a model are discussed in the following.

2.1 Perceptron and Multi-Layer Neural Networks

The perceptron is based on the idea of the *neuron*. This neuron is a single unit in an ANN that is able to receive input values and produces output values that are affected by the level of activation the neuron has.
Given a feature vector $\mathbf{x}$, a *weight vector* $\mathbf{w}$, and an *activation function* g, the output value y for a neuron is computed as

$$y = g(\mathbf{w}^T\mathbf{x} + b), \tag{3}$$

where b is the *bias* that affects the degree of activation the neuron has independent of the inputs. A directed graph for a neuron with two inputs and one output can be visualized as in Figure 1

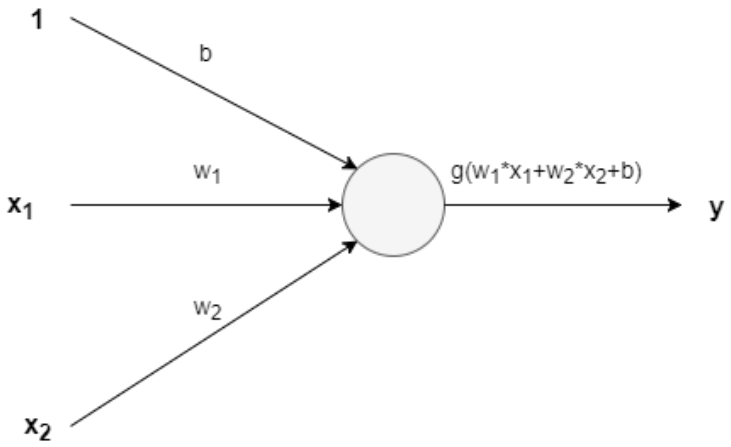

Figure 1: Directed graph for a single neuron

In Figure 1, a neuron is modeled as a node, whereas edges represent the information flow in the direction of the output y. The activation of the neuron is determined by applying an activation function to a linear combination of its input values. The weights affect how much influence a specific input edge has on the activation of the neuron. In principal any function can be chosen as activation function. Nevertheless, there are common choices for ANN which have characteristics that are desirable for this purpose.
Thinking about the neuron as an instance that can either be activated or not activated, functions that only take values in the interval from 0 to 1 can be used to simulate the activation behavior. A popular representative of such a function is the *sigmoid (logistic) function* i.e.

$$g(x) = \frac{1}{1 + exp(-x)}, \tag{4}$$

For large negative values the denominator of the *sigmoid function* becomes large and thus, the value of the function converges towards zero, whereas for

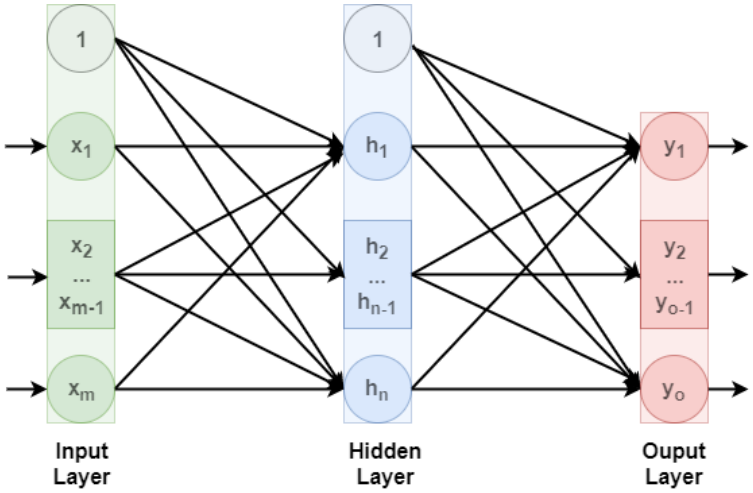

Figure 2: Hidden-Layer Neural Network

large positive values the denominator converges to 1 and thus, the sigmoid function converges to 1.
In order to extend a single neuron to neural network, layers are introduced. A layer is a set of neurons that may be activated in parallel. In an ANN, there exists three types of layers, namely the *input layer*, the *hidden layer*, and the *output layer*. In the network these layers are connected sequentially, while every neuron of the current layer is connected to every neuron of the following layer. Figure 2 shows a directed graph of ANN that has a m-dimensional input and a o-dimensional output. The number of neurons in the input and output layers are determined by the dimension of the input data and the problem specific dimension of the output that is needed. In contrast, the amount of neurons in the hidden layer is not defined by any external property and has to be specified as a hyper-parameter. Theoretically, it can be every natural number $n \in \mathbb{N}$. The characteristic that each neuron in a layer is connected to every neuron in the next layer makes the network *fully connected*. As a result the number of edges between layers is defined by the number of neurons in the adjacent layers. In Figure 2, the number of edges between the input and the hidden layer is equal to $(m+1) \cdot n$. The plus one occurs, because each layer also contains a *bias* that is activated. In Figure 2 the bias is illustrated by the top nodes that contain a one. Neurons within the same layer are not connected which allows for parallel activation. Since each edge has a corresponding weight which determine the degree of influence of the incoming neuron on the outcoming neuron, we can rewrite (3) by

$$\hat{\mathbf{y}} = g^{(3)}(\mathbf{W}^{(2)}\mathbf{h}), \tag{5}$$

where $\mathbf{W}^{(2)} \in \mathbb{R}^{o \times (n+1)}$ is the weight matrix, $\mathbf{h} \in \mathbb{R}^{n+1}$ contains the acti-

vation values of the hidden layer neurons, and $\hat{\mathbf{y}} \in \mathbb{R}^o$ is the output vector. Note that the weights of the bias are now included of the weight matrix. The activation function $g^{(3)}$ is applied element-wise to the resulting vector of the matrix multiplication. Furthermore, the activation values of the hidden layer neurons are computed by

$$\mathbf{h} = g^{(2)}(\mathbf{W}^{(1)}\mathbf{x}). \tag{6}$$

The dimensions follows the same logic as in (5). Inserting 6) into (7) results in

$$\hat{\mathbf{y}} = g^{(3)}(\mathbf{W}^{(2)}(g^{(2)}(\mathbf{W}^{(1)}\mathbf{x}))), \tag{7}$$

which provides the definition how to transform the input vector $\mathbf{x}$ into the output vector $\hat{\mathbf{y}}$. This network is often called *feed-forward neural network*, because the input is forwarded by the connections from layer to layer. Although the computation of the output vector given an input vector is straightforward, ANNs are able to transform linear combinations of inputs into complex functions that are suitable of solving non-linear tasks. In fact, as Hornik et al. (1989) have shown, ANNs are capable of approximating any measurable function to any degree of accuracy. While, this property makes ANNs powerful, it also creates two major problems. Firstly, a machine learning algorithm that perfectly fits the training data, will perform poorly on new data. This phenomenon is known as *overfitting*. In order to prevent overfitting of ANNs, a mechanism called *dropout* was proposed. Dropout randomly excludes some neurons and their connections from the training in order to prevent that neurons in the same layer adapt to learn the same pattern (Srivastava et al. 2014). Secondly, the number of weights that have to be trained increases exponentially with the number of layers and neurons per layer in ANNs. Thus, for more complex architectures, it is computational infeasible to find an optimal combination of weights by iterating through all possibilities. Therefore, a method that iteratively adapts the weights to minimize the error is used to train ANNs. This method is discussed in the next section.

2.2 Training

The goal of ANN training is to find values for the weights that map to the right output and find patterns that generalize for unknown test data. During the training process, the quality of the weights are assessed by a loss function, which measures the difference between the predicted output $\hat{y}$ and the real output y. Since all conventional optimization methods for ANNs are gradient based, the loss function needs to be differential. An common choice in classification tasks for the loss function is the *cross-entropy*

$$l(y_i, \hat{y}_i) = -y_i \ln \hat{y}_i + (1 - y_i) \ln(1 - \hat{y}_i). \tag{8}$$

In order to get a measure for the whole training set, the loss function is summed for all output values and averaged over all training samples, i.e.

$$J = \frac{1}{N} \sum_{i=1}^{N} \sum_{j=1}^{o} l(y_{ij}, \hat{y}_{ij}). \tag{9}$$

Note that J depends on the connection weights due to (5). Given this evaluation metric, an optimization method can be applied to minimize (9). All popular optimization methods that are used for ANN training are based on *Gradient Descent*.
The core idea of *Gradient Descent* is to initialize all weights randomly and then adapt the weights step-wise according to direction of the gradient until a satisfactory result is achieved. More specifically, in every step each weight w is updated by

$$w_t \leftarrow w_{t-1} - \alpha_t \frac{\partial J}{\partial w_{t-1}}, \tag{10}$$

where w_t represents a value of any weight that connects two neurons in the network at step t, $\alpha \in]0, 1]$ is the learning rate and $\frac{\partial J}{\partial w_{t-1}}$ is the partial derivative of the empirical loss with respect to w_{t-1}.

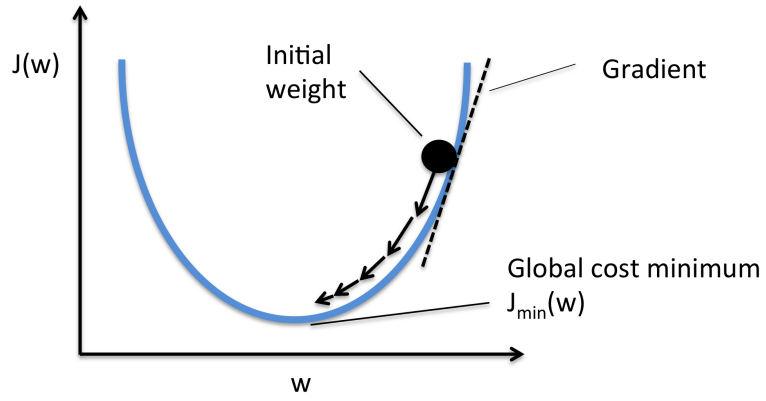

Figure 3: Gradient Descent in the one-dimensional case from Raschka (2017)

Figure 3 shows a convex problem that is optimized by Gradient Descent. Since the convexity assumption is fulfilled for J in this case, each update of w will lead to decrease in its function value. Hence, repeating this procedure will eventually lead to a value of w that lies in the global optimum. However, optimizing ANNs is a non-convex problem, i.e. there exist other local minima and saddle points. In consequence, Gradient Descent might converge to one of these local minima rather than to global minimum depending on the initialization of the weight. Furthermore, having saddle points could lead to

a state where the derivative of J becomes zero even though a minimum is not reached.
Additionally, the choice of α may have an influence of the chance to reach a global cost minimum. If α is small the adaption of w happens slowly and it may need a lot of steps to reach the global cost minimum. On the other hand, if α is large, Gradient Descent might overshoot the global cost minimum. Apart from these drawback, gradient descent methods often deliver satisfying solutions for ANNs training. In order to apply these methods, the partial derivatives of weights in each layer, e.g. in single hidden layer case $\frac{\partial J}{\partial W^{(1)}}$ and $\frac{\partial J}{\partial W^{(2)}}$, are needed. For this purpose a method called *backpropagation* was proposed by Rumelhart et al. (1988). Below, we will explain this method on the example of a single hidden layer feed-forward network.
Mathematically, backpropagation is simply applying the chain rule of calculus in order to work out the contribution of each weight to the total loss. Applying this concept, the partial derivative of J with respect to weight matrix $\mathbf{W}^{(2)}$ is given by

$$\frac{\partial J}{\partial \mathbf{W}^{(2)}} = \frac{\partial J}{\partial \hat{\mathbf{y}}} * \frac{\partial \hat{\mathbf{y}}}{\partial \mathbf{W}^{(2)}} \tag{11}$$

From (5) it is known that $\hat{\mathbf{y}}$ is computed applying an activation function to the results of the linear combinations of $\mathbf{W}^{(2)}$ and $\mathbf{h}$. Naming the result of that linear combination $\hat{\mathbf{y}}^{(3)}$ and applying the chain rule again, (11) can be rewritten as

$$\frac{\partial J}{\partial \hat{\mathbf{y}}} * \frac{\partial \hat{\mathbf{y}}}{\partial \mathbf{W}^{(2)}} = \frac{\partial J}{\partial \hat{\mathbf{y}}} * \frac{\partial \hat{\mathbf{y}}}{\partial \hat{\mathbf{y}}^{(3)}} * \frac{\partial \hat{\mathbf{y}}^{(3)}}{\partial \mathbf{W}^{(2)}} \tag{12}$$

As $\hat{\mathbf{y}}$ is the result of applying an activation function $g^{(3)}$ to $\hat{\mathbf{y}}^{(3)}$, the change of $\hat{\mathbf{y}}$ with respect to $\hat{\mathbf{y}}^{(3)}$ is exactly the derivation of $g^{(3)}$ at the point $\hat{\mathbf{y}}^{(3)}$. Moreover, it shows up from (5) that the change of $\hat{\mathbf{y}}^{(3)}$ with respect to $\mathbf{W}^{(2)}$ is $\mathbf{h}$. Consequently, (12) results into the following equation

$$\frac{\partial J}{\partial \hat{\mathbf{y}}} * \frac{\partial \hat{\mathbf{y}}}{\partial \hat{\mathbf{y}}^{(3)}} * \frac{\partial \hat{\mathbf{y}}^{(3)}}{\partial \mathbf{W}^{(2)}} = \frac{\partial J}{\partial \hat{\mathbf{y}}} * g^{(3)\prime}(\hat{\mathbf{y}}^{(3)}) * \mathbf{h}, \tag{13}$$

that gives the update rule for $\mathbf{W}^{(2)}$ in Gradient Descent as a result of backpropagation.
The rule for $\mathbf{W}^{(1)}$ can be derived similarly with the difference that the error has be further propagated backwards to the ANN. The result for the network is

$$\frac{\partial J}{\partial \mathbf{W}^{(1)}} = \frac{\partial J}{\partial \hat{\mathbf{y}}} * g^{(3)\prime}(\hat{\mathbf{y}}^{(3)}) * \mathbf{W}^{(2)} * g^{(2)\prime}(\mathbf{h}^{(2)}) * \mathbf{x}, \tag{14}$$

where $\mathbf{h}^{(2)}$ is the result from the linear combination of the input vector $\mathbf{x}$ and the weight matrix $\mathbf{W}^{(1)}$.

The derived update rules represent an update after a single observation in the training set. Besides this, an update can be made after deriving the results for a batch of observations or the complete training set. In the former case, gradient descent is called *stochastic gradient descent* (SGD). The direction of the update is then computed by adding up the results for the single observations. As the update for single observations can be computed independently from others, those computations can be parallelized. The optimal choice of observations after which an update should be made is problem specific. The trade-off is the following. On the one hand, making very frequent updates of the weight matrices after only a few observations is faster, but can lead into the wrong direction as those few observations may not generalize well. On the other hand, making only few updates that are based on general information about the data after backpropagating the error for a lot of observations leads to a more reasonable update but takes more time to be computed. Normally, a choice in between is made.

3 Feature Representations for Text

All state-of-the-Art machine learning algorithms commonly used in document classification are not capable of dealing with raw text as an input. ANNs are no exception. Thus, a major question when building a document classifier is how to represent text in the documents as numerical features. In this section, we will discuss common strategies to answer this question. For simplicity, we use words as entity on which features are build, but all discussed methods also work on other common entities such as n-grams.

3.1 Sparse Representations

Given the set of all unique words in the vocabulary $V = \{w_1, \dots, w_{|V|}\}$, documents can be represented by a $|V|-$dimensional vector

$$\mathbf{x}_i = (f_1^{(i)}, \dots, f_{|V|}^{(i)}) \quad \forall i = 1, \dots, N,$$

where $f_j^{(i)} \in \mathbb{R}$ is the weight for word w_j from the vocabulary V in the document d_i. Since many words in V are not included in a specific document d_i, this representation tends to produce many weights that are zero. Hence, they often result into sparse feature vectors.
The most basic form of $\mathbf{x}_i$ known as *bag-of-words* (BOW) uses binary weights which indicate whether a word in the vocabulary is present in a document or not, i.e.

$$f_j^{(i)} = \begin{cases} 1, & \text{if } w_j \in d_i \\ 0, & \text{otherwise} \end{cases} \tag{15}$$

A more sophisticated alternative is the *term frequency–inverse document frequency* (tf-idf). It is based on the idea that a word is important for a specific document if it fulfills the following two properties:

(1) the word w_j appears often in a specific document d_i and

(2) w_j does not appear often in all documents $d_k \in D$.

The first property is measured by the *term frequency*, which counts occurrences of the words in a document. Let the function

$$fr(w_j, d_i) = \begin{cases} 1 \text{ if } w_j = w_t \\ 0 \text{ otherwise} \end{cases} \qquad \forall w_t \in d_i \tag{16}$$

be the frequency function. Then in it simplest form the term frequency weights are given with

$$tf(w_j, d_i) = \sum_{w_j \in d_i} fr(w_j, d_i). \tag{17}$$

Property (2) on the other hand is quantified by the *inverse document frequency*, which can be defined by

$$idf(w_j) = log \frac{1 + N}{1 + df(w_j, D)} + 1, \tag{18}$$

where the *document frequency* $df(w_j, D) = |\{d_i \in D : w_j \in d_i\}|$ counts how many documents include the word w_j. Note that there exist different variants of *idf* in the literature. However, the version in (18) is implemented in the popular *scikit-learn* library[1]. Finally, the weights of tf-idf are calculated by the product of the two measures

$$f_j^{(i)} = tf(w_j, d_i) * idf(w_j). \tag{19}$$

All of the feature representations discussed above ignore the positions and order of the words in a document. Furthermore, they do not capture semantic similarities of words. These drawbacks serve as a motivation for *weight vectors* that are introduced below.

3.2 Dense Representations

Dense feature representation represent each word as a vector of weights, the *word vectors*. The weights of these vectors are obtained by some pre-training phase that makes use of a domain-specific corpus. For example, in the case of medical documents a corpus of texts with medical content can be used to pre-train the weights of the word vectors. After the pre-training phase, the

[1] http://scikit-learn.org

words in the documents are matched to their corresponding word vectors. Thus, documents are represented by a T-dimensional vector

$$\mathbf{x}_i = (\mathbf{v}_j, \ldots, \mathbf{v}_k) \in \mathbb{R}^T, \tag{20}$$

where $\mathbf{v}_j \in \mathbb{R}^l$ is the word vector of word w_j that appears first in d_i. Note that the user will have to fix T, since each document feature vector should have the same dimension. Similarly, the dimensions of the *word vectors* l need to be set beforehand.
Word vectors or *word embeddings* were introduced by Bengio et al. (2003). They intent to find weights such that vectors of similar words have a similar positions in the vector space. In consequence, word vectors allow for arithmetic operations with words that lead to meaningful results. For instance, let $\mathbf{v}_{king}, \mathbf{v}_{man}, \mathbf{v}_{woman}$ and $\mathbf{v}_{queen}$ be the word vectors for their corresponding words. Then it should hold that

$$\mathbf{v}_{king} - \mathbf{v}_{men} + \mathbf{v}_{woman} \approx \mathbf{v}_{queen}. \tag{21}$$

This arithmetic is not restricted to the gender dimension as in (21), but also applies to other semantic relationships among words (Mikolov et al. 2013a). Popular approaches to learn word vectors are *GloVe* (Pennington et al. 2014), *FastText* (Bojanowski et al. 2016), *StarSpace* (Wu et al. 2017) and *word2vec* (Mikolov et al. 2013c). Since we have experimented with feature representations based on the latter, we will focus on it here.
Word2vec builds on the intuition that similar words should appear in similar contexts. This idea is applied in form of two related algorithms, the *skip-gram model* and *continuous bag-of-words* (CBOW). In the following, we will briefly introduce both algorithms.

Skip-Gram Model

Given a word w_t in a document at position t, the *skip-gram model* seeks to predict all words that are in m-neighborhood of w_t, i.e. all words in the set

$$C(w_t) = \Big\{ w_{t+j} | -m \leq j \leq m, j \neq 0 \Big\}. \tag{22}$$

For instance if $m = 5$, the goal is to predict the five words before and the five words after w_t. In the following, we will refer to w_t as target word and to all words $c \in C(w_t)$ as context words. Formally, the *skip-gram model* maximizes the average log probability that a word appears in the context of the target word given the target word, i.e

$$\max_{\theta} \frac{1}{T} \sum_{t=1}^{T} \sum_{c \in C(w_t)} log\ \mathbb{P}(c|w_t; \theta). \tag{23}$$

Note that the set of parameters θ are given by the word vector of the target word $\mathbf{v}_t$ and the word vectors of the context words $\mathbf{v}_c$. The maximization problem in (23) is used as an objective for a Feed-Forward Neural Network with one hidden layer as introduced in Section 2. More specifically, Mikolov et al. (2013c) proposed an architecture that uses one hot-encoded vectors for each word w_t as inputs and a linear activation function for the hidden layer. The output layer has a *softmax function* for each context word

$$\mathbb{P}(c|w_t;\theta) = \frac{exp(\mathbf{v}_c\mathbf{v}_t^T)}{\sum_{i=1}^{|V|} exp(\mathbf{v}_i\mathbf{v}_t^T)}. \tag{24}$$

The softmax function is expensive to compute, since the normalization factor in the denominator sums over all words in the vocabulary V. Therefore, different approximation strategies that reduce the computational effort have been proposed (see Morin & Bengio (2005) and Mikolov et al. (2013c) for more details). Given this architecture, the hidden layer can be represented by a weight matrix $\mathbf{H} \in \mathbb{R}^{|V|\times l}$, where each of the $|V|$ rows corresponds to a word in the vocabulary. After training the model, these rows are extracted as word vector that can be used for the actual task.

Continuous Bag-of-Words (CBOW)

The problem setting that CBOW formulates is exactly the opposite of the skip-gram model. Given the context words, CBOW seeks to predict the target word. Consequently, the maximization problem is similar to the one in (23) with the difference that the conditioning is flipped, i.e.

$$\max_{\theta} \frac{1}{T}\sum_{t=1}^{T}\sum_{c\in C(w_t)} log\ \mathbb{P}(w_t|c;\theta). \tag{25}$$

Again this is used to train a feed-forward neural network. Figure 3.2 illustrates the ANN architectures of CBOW and the skip-gram model. As with the objective, the ANN architecture for CBOW is exactly flipped. Analogously, the $|V|$ rows of the hidden layer weight matrix are extracted as weight vectors after the training.
A major difference between both models is that in CBOW the word vectors of the context words are averaged before predicting the target word, which is not necessary in the skip-gram model. Due to this difference, Mikolov et al. (2013b) argue that the skip-gram model creates more accurate representations for infrequent words, while CBOW is faster.

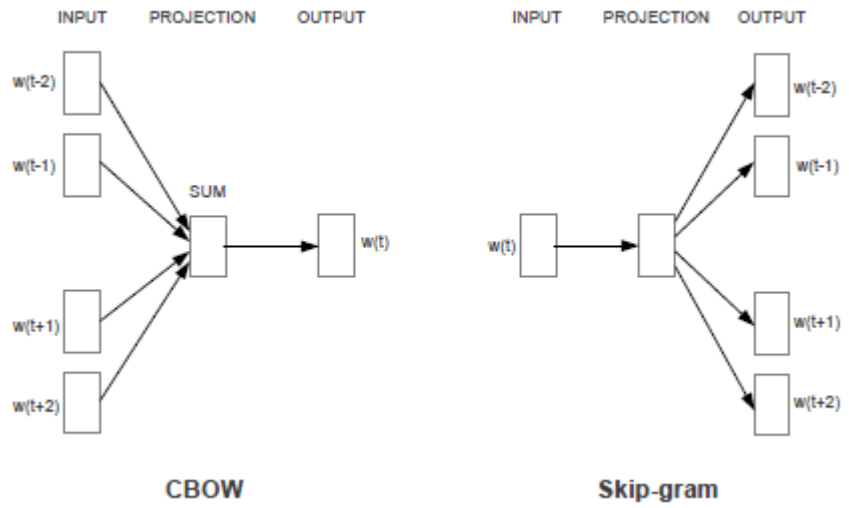

Figure 4: ANNs for *Word2Vec* from Mikolov et al. (2013c)

4 Advanced Neural Network Architectures

4.1 Recurrent Neural Networks

The feed-forward neural networks that were introduced above deliver a framework that is flexible to solve a wide range of tasks. Nevertheless, these networks are not capable of modeling contextual information. Generally, the order of the input is not used as a property of the model. Considering natural language processing, the order of words plays a major role if we want to understand the content. For this purpose, a class of models called *recurrent neural networks* (RNNs) can be used. RNNs contain cycles such that the output of a unit can be fed back into the same unit and thus, it is possible to include ordering information into the model (Rojas 1996). The basic architecture includes an input layer that feeds forward the input to a recurrent layer that contains a cycle as well as an output edge to the output layer. This cycle enables to add a weight to the activation value from time point t and use it as an additional input edge for the input at time point $t+1$. This procedure can be repeated indefinitely and thus, RNNs are theoretically able to model contextual dependencies over an infinite time horizon (Mikolov et al. 2010).
The major problem of RNNs is the *vanishing gradient problem*. It describes the phenomenon that gradient based learning algorithms have serious difficulties to learn long term dependencies as the gradient computation becomes unstable (Bengio et al. 1994). To overcome this problem Hochreiter

& Schmidhuber (1997) introduced an extension to RNNs called *long-short term memory*(LSTM).

4.2 Long Short Term Memory Model

The core of the LSTM model is the memory cell that is composed of four parts. An input gate, a neuron with a self-recurrent cycle, a forget gate, and an output gate (Figure 5). Every time an input value arrives at the cell input, an activation function is applied to it. In a next step, the input gate decides how much of the activation value gets passed to the neuron. Input activation and input gate are concatenated with multiplication. The neuron has a cell state s that stores the information that was passed earlier in the process. As the weight of the self-recurrent edge is equal to one, the forget gate that was introduced in Gers et al. (1999) determines how much of the cell state at time point t is kept until time point $t+1$. The forget gate and s are connected by multiplication. If the input gate is open at time point $t+1$ and new information flows to the neuron, the new input state is computed by adding up the part of the cell state from t that was not forgotten and the proportion of the input at $t+1$ that passed the input gate. Finally, another activation function is applied to s and the output gate determines again by multiplication how much of current cell state passes to the cell's output.

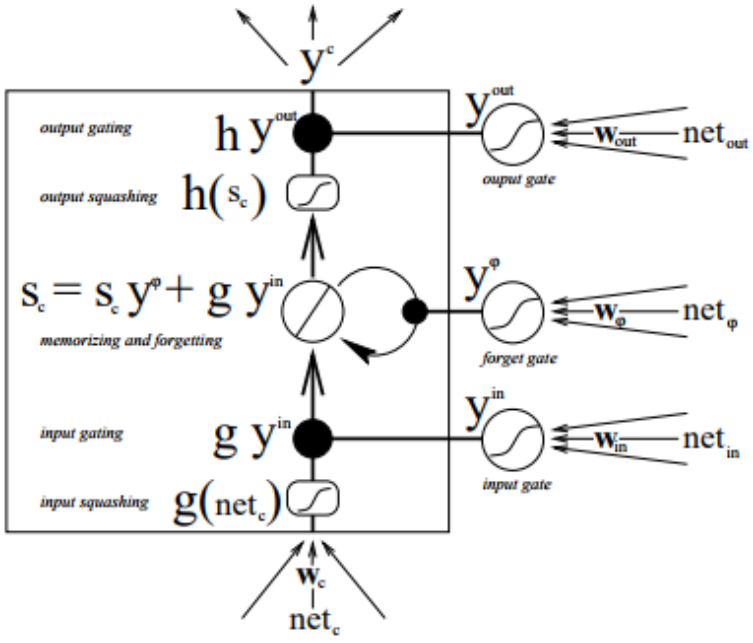

Figure 5: LSTM memory cell with forget gate from Gers et al. (1999)

The multiplicative gates allow the LSTM cell to store information over a long period of time. If the input gate is closed, new information cannot flow to the neuron and thus, the current cell state will not be overridden. With the output gate it can be controlled when to release the cell's information to the output. Thus, the LSTM cell is able to overcome the vanishing gradient problem (Graves et al. 2012).

4.3 Gated Recurrent Unit

Similar to the LSTM cell, the *Gated Recurrent Unit*(GRU) is able to handle the vanishing gradient problem. The main difference between both models is that the GRU reveals its content to the subsequent parts of the network at every point in time as it has no output gate to control the output flow (Chung et al. 2014). Besides, the update of the unit's activation value happens with linear interpolation between the previous activation value and the new candidate activation value. An update gate decides how much the value of the previous activation gets updated. The candidate activation value is computed as the tangens hyperbolicus of the sum of the input value and the previous activation value while a reset gate determines how much of the previous activation gets into this computation (Cho et al. 2014).
Compared to the LSTM cell, the GRU contains fewer gates and thus, it needs less computation to update the unit. As both frameworks also share some properties, the choice which to use on a specific problem cannot be generally answered beforehand (Chung et al. 2014).

4.4 Convolutional Neural Networks

Convolutional Neural Networks (CNN) are heavily used in computer vision and were introduced to text classification tasks by Collobert et al. (2011).
A CNN consists at least of one *convolutional layers*, which applies non-linear functions called *filters* on a sliding window. This sliding window captures different regions of the data. While in the image recognition case these regions refer to a specific area defined by 2-dimensional pixels, regions in texts are defined by 1-dimensional sequences of words. In both cases, the output of a convolutional layer is aggregated by a following *pooling layer*.
Given a sequence of words and their corresponding word vector, a convolutional layer moves a sliding window of size k and concatenates the word vectors lying within this window

$$\mathbf{c}_i = (\mathbf{v}_i; \mathbf{v}_{i+1}; \ldots; \mathbf{v}_{i+k-1}). \tag{26}$$

k is known as *kernel size* and needs to be specified manually. Furthermore, the number words that the successor window jumps ahead, called *strides*, also constitutes a hyper-parameter. For instance, if *strides* are set to 2, then

the window after $\mathbf{c}_i$ starts with word vector of the word on position $i+2$. After a window is extracted, a *filter* is applied to it, i.e.

$$\mathbf{p}_i = g(\mathbf{W}\mathbf{c_i} + \mathbf{b}). \tag{27}$$

A *filter* is simply a non-linear function similar as in feed-forward neural networks, where g is the activation function, $\mathbf{W} \in \mathbb{R}^{k \cdot l \times d_{conv}}$ is weight matrix and $\mathbf{b} \in \mathbb{R}^{d_{conv}}$ is the *bias*. Finally, the results of all vectors $\mathbf{p}_1, \ldots, \mathbf{p}_m$ are aggregated. Often, this aggregation is done by extracting the highest value out of each of vector $\mathbf{p}_i \in \mathbb{R}^{d_{conv}}$, i.e.

$$\max_{1<i\leq m} \mathbf{p}_i[j],$$

which is called *max-pooling* in the literature. Alternatives include *average-pooling*, however for text classification tasks max-pooling has been shown to be often more effective (see Zhang & Wallace (2015)).
The *convolutional-pooling* architecture can be repeated multiple times to allow a deeper level of feature extraction. Typically, the last pooling layer is followed by a fully-connected layer that maps the network to the desired output dimension. However, it is also possible to use any of the recurrent structures described above as a successor layer.

5 Experimental Setup

5.1 Data

In order to empirically evaluate ANNs for document classification, two datasets have been used. These two data sets include one binary classification and one multiclass classification problem.
The dataset for the binary classification problem contains 2155 titles and abstracts from journal articles in the PubMed database. The aim was to classify whether an article talks about cancer or not. The distribution of labels can be seen in figure 6. On average the abstracts were about 1500 words long. The multiclass dataset was substantially larger, with 45587 texts distributed among 23 categories. Figure 7 shows the distribution of the categories. We can see that they are highly skewed. Three categories (*Pathological Conditions, Signs and Symptoms , Neoplasms and Cardiovascular Diseases*) make up more than a third of the corpus. On average the texts were about 1200 words long.

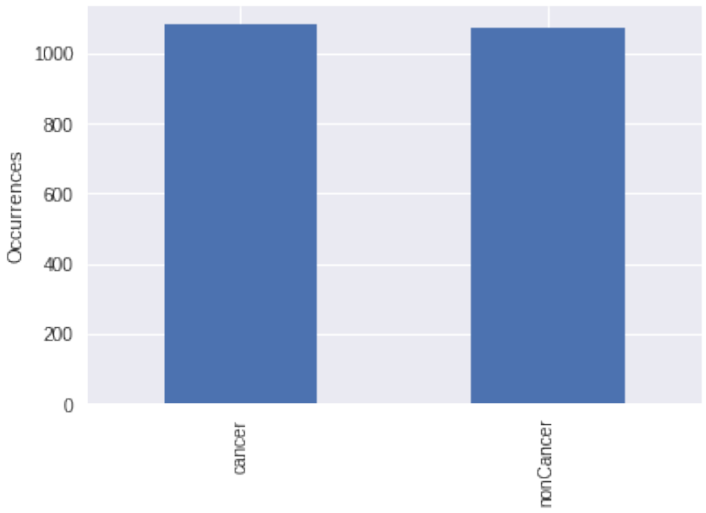

Figure 6: Distribution of classes in the binary dataset

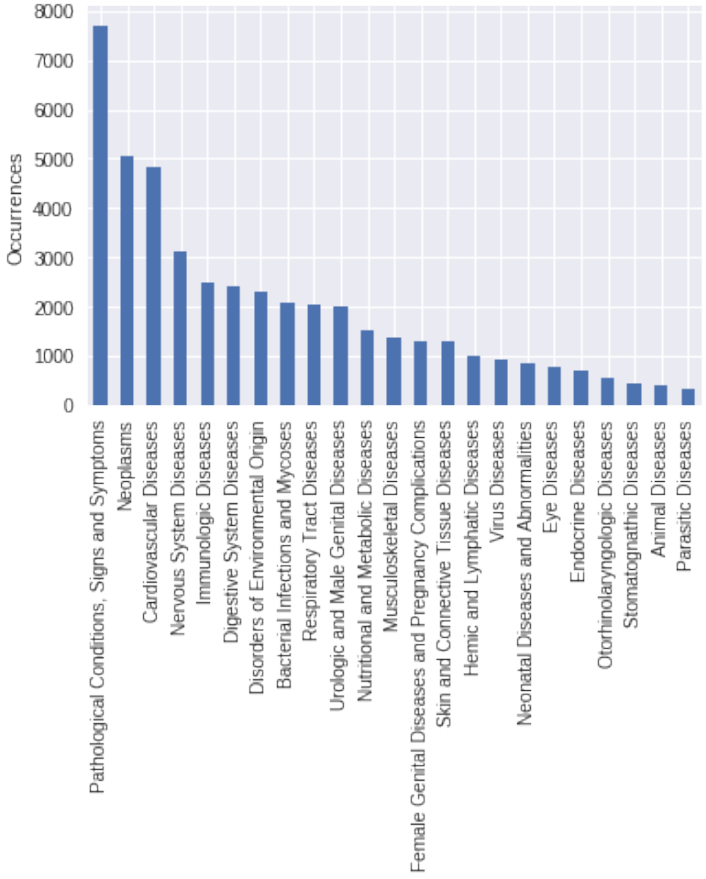

Figure 7: Distribution of classes in the multiclass dataset

5.2 Preprocessing

The dataset for the binary problem was preprocessed with scikilearn's *CountVectorizer* and *Tf-Idf-Transformer* functions. Through this we tokenized the corpus by whitespace and kept only tokens that occur at a maximum of 80% of the documents and at a minimum in 3 documents. We considered uni-grams and bi-grams. The documents were then transformed into sparse vectors and the word occurrences weighted with the *TF-IDF* function.
Our approach for the multiclass dataset was quite different. Before any preprocessing was done we upsampled the three rarest classes by simply duplicating all documents in each class. Through this we were able to achieve slightly better results overall and in these rare classes in particular. We then tokenized the documents with the regular expression *w+*. This splits the documents at whitespace and special characters. We found that this tokenization led to the lowest amount of non-matches when substituting the word vectors. We then replaced all tokens with pre-trained PubMed word vectors based on the *word2vec* algorithm. Due to memory constraints we had to limit the documents to the first 100 tokens.

5.3 Applied Architectures

All models were implemented with the popular *keras* library[2]. Unless otherwise specified all models were trained to minimize the *binary* or *categorical cross-entropy* (for binary and multiclass dataset respectively) loss using the *adagrad* optimizer. All models were trained for 15 epochs with a batch size of 256. *ReLu* activation was used for all layers except the output layers. The hyper-parameter configuration for the CNN models was largely taken from Zhang & Wallace (2015). Adding additional layers to the architectures described below did not improve results and often led to overfitting.

Feed Forward Neural Network: For the binary classification problem we used a traditional feed forward network with various hidden layer and node count configurations.

1. LR: No hidden layer, only one node with sigmoid activation. Basically equivalent to a Logistic Regression.
2. Perceptron: One hidden layer with one node.
3. 1-hidden: One hidden layer with 256 nodes. Dropout regularization 0.5 between hidden and output layer.
4. 2-hidden: Two hidden layers, the first with 256 nodes and the second with 128 nodes. Dropout of 0.5 between both hidden layers and the output layer.

[2]https://keras.io/

LSTM: We used a 1-layer LSTM network with 128 neurons and a dense output layer. The network was regularized with dropout of 0.3 between the LSTM and dense layer.

GRU: The GRU network is identical to the LSTM network with the LSTM layer replaced by a GRU layer.

CNN: The basic CNN model includes two convolution layers. The first layer includes 100 filters with a kernel size of one. This corresponds to looking at uni-grams. The second layer has 256 filters and a kernel size of two, which corresponds to looking at bi-grams. The output of the convolution layers is then squashed with a Global-Max-Pooling layer and fed into a dense layer with 256 neurons before reaching the dense output layer. All layers were regularized with a dropout of 0.4.

R-CNN: The architecture for the recurrent CNN is largely identical to the basic CNN. The Global-Max-Pooling layer was replaced with a regular Max-Pooling layer with a pool size of 32 (corresponding to a sequence length of 32 timesteps). The hidden dense layer was replaced with a GRU layer with 128 nodes.

6 Results

6.1 Binary Classification

Figure 8 shows the cross-validation results for the binary classification problem. As we can see the difference between all four architectures is very small and all models exhibit a somewhat large variance of around 5% between the splits. This is most likely due to the very limited number of documents left in each of the cross-validation splits. 1-hidden shows the highest median accuracy of 94.21%, which is why we choose this architecture for the final submission. On the private dataset we were able to achieve a slightly higher accuracy of 95.78%. Runtime for all models was insignificant (below 5 seconds) on a GTX 1080 GPU. We have briefly experimented with a word vector based approach for the binary classification task. However, due to the very limited amount of data we were not able to train more complex recurrent or convolution based models and subsequently achieved worse results than with a simple feed forward architecture.

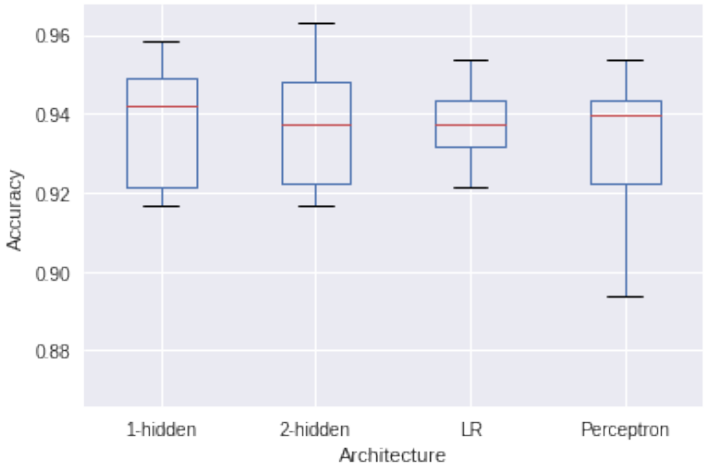

Figure 8: 10-fold cross-validation results of binary dataset

6.2 Multiclass Classification

Figure 9 shows the cross-validation results for the multiclass classification problem. The variance of each model is a lot smaller than in the binary classification problem, likely due to the tenfold increase in the number of documents. Based on a Wilcoxon test the LSTM model performed significantly (p=0.005) worse than the best model, the R-CNN, which in turn performed significantly better than both the basic CNN and GRU model (p=0.005 for both). It achieved a median accuracy of 49.59%. We therefore choose the R-CNN for the submission. On the private dataset we were able to reach an accuracy of 48.37%.

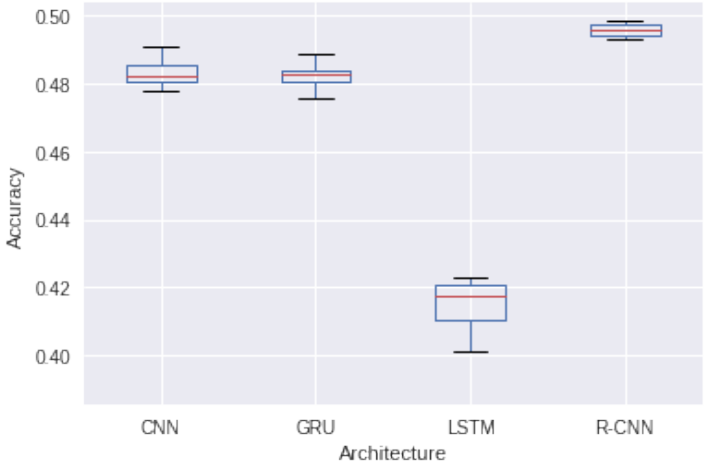

Figure 9: 10-fold cross-validation results of multiclass dataset

Additionally to superior classification performance the R-CNN also trained much faster than basic recurrent models and only slightly slower than the basic CNN model (see table 1).

Model	**Runtime 15 epochs**
LSTM	310 seconds
GRU	264 seconds
CNN	108 seconds
R-CNN	126 seconds

Table 1: Runtime of multiclass models on a GTX 1080 GPU

7 Conclusion

Traditionally pre-processing was the main driver of success in natural language processing. Dense representations in word vectors combined with recurrent neural network architectures that natively deal with sequences have shifted the focus towards innovations in algorithms. In this paper we have examined various ways to solve the text classification task using neural networks. We have successfully combined convolutions (as a form of feature extraction) and recurrent layers to a model that provides excellent results at a much better runtime performance than recurrent models traditionally used in NLP. However, for problems with a small amount of data or a simple classification problem more complex architectures do not necessarily provide better results, as evidenced by the private leader-board results for the binary classification problem. More complex problems with more data, like the multiclass problem, do benefit from the flexibility that the neural network family of models provides. Furthermore, the fact that neural networks can trivially be extended to more than two classes, with no additional computational cost, make them a natural fit for text classification problems.

References

Bengio et al. 2003
Bengio, Yoshua; Ducharme, Réjean; Vincent, Pascal; Jauvin, Christian: A neural probabilistic language model. In: *Journal of machine learning research* 3 (2003), Nr. Feb, pages 1137–1155

Bengio et al. 1994
Bengio, Yoshua; Simard, Patrice; Frasconi, Paolo: Learning long-term dependencies with gradient descent is difficult. In: *IEEE transactions on neural networks* 5 (1994), Nr. 2, pages 157–166

Bojanowski et al. 2016
Bojanowski, Piotr; Grave, Edouard; Joulin, Armand; Mikolov, Tomas: Enriching word vectors with subword information. In: *arXiv preprint arXiv:1607.04606* (2016)

Cho et al. 2014
Cho, Kyunghyun; Merriënboer, Bart van; Bahdanau, Dzmitry; Bengio, Yoshua: On the Properties of Neural Machine Translation: Encoder–Decoder Approaches. In: *Syntax, Semantics and Structure in Statistical Translation* (2014), pages 103

Chung et al. 2014
Chung, Junyoung; Gulcehre, Caglar; Cho, KyungHyun; Bengio, Yoshua: Empirical evaluation of gated recurrent neural networks on sequence modeling. In: *arXiv preprint arXiv:1412.3555* (2014)

Collobert et al. 2011
Collobert, Ronan; Weston, Jason; Bottou, Léon; Karlen, Michael; Kavukcuoglu, Koray; Kuksa, Pavel: Natural language processing (almost) from scratch. In: *Journal of Machine Learning Research* 12 (2011), Nr. Aug, pages 2493–2537

Gers et al. 1999
Gers, Felix A.; Schmidhuber, Jürgen; Cummins, Fred: Learning to forget: Continual prediction with LSTM. (1999)

Graves et al. 2012
Graves, Alex et al.: *Supervised sequence labelling with recurrent neural networks*. Bd. 385. Springer, 2012

Hebb 1949
Hebb, DO: The organization of behavior. A neuropsychological theory. (1949)

Hochreiter & Schmidhuber 1997
HOCHREITER, Sepp; SCHMIDHUBER, Jürgen: Long short-term memory. In: *Neural computation* 9 (1997), Nr. 8, pages 1735–1780

Hornik et al. 1989
HORNIK, Kurt; STINCHCOMBE, Maxwell; WHITE, Halbert: Multilayer feedforward networks are universal approximators. In: *Neural networks* 2 (1989), Nr. 5, pages 359–366

McCulloch & Pitts 1943
MCCULLOCH, Warren S.; PITTS, Walter: A logical calculus of the ideas immanent in nervous activity. In: *The bulletin of mathematical biophysics* 5 (1943), Nr. 4, pages 115–133

Mikolov et al. 2013a
MIKOLOV, Tomas; CHEN, Kai; CORRADO, Greg; DEAN, Jeffrey: Efficient estimation of word representations in vector space. In: *arXiv preprint arXiv:1301.3781* (2013)

Mikolov et al. 2010
MIKOLOV, Tomáš; KARAFIÁT, Martin; BURGET, Lukáš; ČERNOCKỲ, Jan; KHUDANPUR, Sanjeev: *Recurrent Neural Network Based Language Model.* In: *Eleventh Annual Conference of the International Speech Communication Association*, 2010

Mikolov et al. 2013b
MIKOLOV, Tomas; LE, Quoc V.; SUTSKEVER, Ilya: Exploiting similarities among languages for machine translation. In: *arXiv preprint arXiv:1309.4168* (2013)

Mikolov et al. 2013c
MIKOLOV, Tomas; SUTSKEVER, Ilya; CHEN, Kai; CORRADO, Greg S.; DEAN, Jeff: *Distributed representations of words and phrases and their compositionality.* In: *Advances in neural information processing systems*, 2013, pages 3111–3119

Morin & Bengio 2005
MORIN, Frederic; BENGIO, Yoshua: *Hierarchical Probabilistic Neural Network Language Model.* In: *Aistats* Bd. 5, 2005, pages 246–252

Pennington et al. 2014
PENNINGTON, Jeffrey; SOCHER, Richard; MANNING, Christopher D.: *Glove: Global vectors for word representation.* In: *EMNLP* Bd. 14, 2014, pages 1532–1543

Raschka 2017
RASCHKA, Sebastian: *Machine Learning FAQ.* `https:`

//sebastianraschka.com/faq/docs/closed-form-vs-gd.html. Version: 2017

Rojas 1996
ROJAS, Raul: *Neural Networks: A Systematic Introduction.* Springer Science & Business Media, 1996

Rosenblatt 1958
ROSENBLATT, Frank: The perceptron: A probabilistic model for information storage and organization in the brain. In: *Psychological review* 65 (1958), Nr. 6, pages 386

Rumelhart et al. 1988
RUMELHART, David E.; HINTON, Geoffrey E.; WILLIAMS, Ronald J. et al.: Learning representations by back-propagating errors. In: *Cognitive modeling* 5 (1988), Nr. 3, pages 1

Srivastava et al. 2014
SRIVASTAVA, Nitish; HINTON, Geoffrey E.; KRIZHEVSKY, Alex; SUTSKEVER, Ilya; SALAKHUTDINOV, Ruslan: Dropout: a simple way to prevent neural networks from overfitting. In: *Journal of machine learning research* 15 (2014), Nr. 1, pages 1929–1958

Wu et al. 2017
WU, Ledell; FISCH, Adam; CHOPRA, Sumit; ADAMS, Keith; BORDES, Antoine; WESTON, Jason: StarSpace: Embed All The Things! In: *arXiv preprint arXiv:1709.03856* (2017)

Zhang & Wallace 2015
ZHANG, Ye; WALLACE, Byron: A sensitivity analysis of (and practitioners' guide to) convolutional neural networks for sentence classification. In: *arXiv preprint arXiv:1510.03820* (2015)