Beispiel-Proceedings, 1–13
Hans-Werner Hilse (ed.)

Recurrent Neural Networks
An introduction to binary classification using Natural Language Processing

F. Süttmann[1,*]

[1]Georg-August-Universität Göttingen, Germany
[*] *Email:* felix.suettmann@gmail.com

Abstract. This essay outlines the applicability of Recurrent Neural Networks for supervised binary classification tasks in Natural Language Processing. It gives a general overview of the relevant theory and different recurrent layers. The theory is than applied to a binary classification task of offensive German language in twitter posts. Different model structures are compared to discover dependencies on batch size and number of iterations for generalization.

1 Introduction

Recurrent Neural Networks are a special Deep Learning architecture optimized for working with sequences. Their name has its origin in a paper by Rumelhart et al. (1986) and has developed much further since then. This essay aims at outlining the general theory behind Recurrent Neural Networks (RNN) and their most prominent extensions. The focus will be on binary classification tasks in Natural Language Processing, because sentences can be viewed as sequences. Information about the meaning of a sentence can be found in multiple words or even span over multiple sentences. To evaluate the sentiment models therefore need to be able to make a connection of information over a long sequence of words with variable length (Goodfellow et al. 2016). We will outline why RNN are specially adapt to that and where they perform less. For simplicity, matters of unsupervised learning are left unattended. The last chapter compares different recurrent layers on a corpus of German Twitter posts that are either offensive or not. Along with that different batch sizes and numbers of iterations are compared on four different recurrent layers.

2 Theoretical Foundations

Recurrent neural networks are designed to handle long sequences of data of variable length, which would be difficult for classical feed-forward networks. To process such a sequence of values $x_0, x_1 \ldots, x_t$ with $t = 0, 1, \ldots, \tau$, parameters are shared over the whole sequence and updated all at once. This chapter first gives an introduction

to simple Recurrent Neural Networks. In the Section 2.2 problems during the optimization of the model, that led to the creation of different, more complex recurrent layers, are described. The most prominent two, Long Short Term Memory networks and Gated Recurrent Units, are outlined in Section 2.3 and 2.4. Section 2.5 gives an overview of different other architectures and how RNN can be deep.

2.1 Recurrent Neural Network

A simple Recurrent Neural Network has a specific, recurrent layer that evaluates a sequence. The sequence of values $x_0, x_1 \ldots, x_t$ with $t = 0, 1, \ldots, \tau$ is first broken down into single pieces. These parts parts of the sequence are then used as input for the recurrent layer, see Figure 1. The special characteristic of a recurrent layer is that it feeds its output back into itself and is able to generalize to sequence length. Figure 1 represents two common representations of RNN, a rolled computational graph on the left and an unrolled one on the right. In both illustrations output h_t of the hidden unit A is, together with the next input from the sequence x_{t+1}, fed back into the same cell.

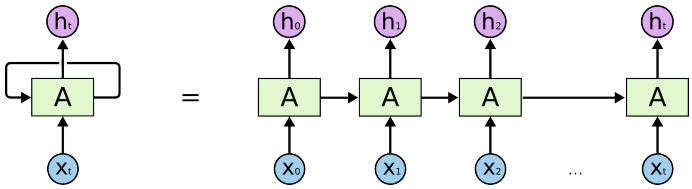

Figure 1. Rolled and unrolled Recurrent Neural Network (Olah 2015)

A is a simple RNN cell which commonly contains either a logistic sigmoid activation function (sigmoid)

$$sigmoid(z) = \sigma(z) = \frac{1}{1 + e^{-z}}. \tag{1}$$

or hyperbolic tangent activation function (tanh)

$$tanh(z) = \frac{sinh(z)}{cosh(z)} = \frac{e^z - e^{-z}}{e^z + e^{-z}} \tag{2}$$

Assuming that we use the tanh function as activation function, h_t the output of the hidden recurrent layer is generated by

$$h_t = tanh(W^{(hh)} h_{t-1} + W^{(hx)} x_t) \tag{3}$$

with $W^{(hh)}$ being the weight matrix between this and the previous hidden state h_{t-1} and $W^{(hx)}$ are the weights between the input and the hidden state. The tanh function is a non-linear function applied element wise to the product

$$\left[\begin{array}{c|c} W^{(hh)} & W^{(hx)} \end{array} \right] \begin{bmatrix} h \\ x \end{bmatrix}.$$

The block matrix has dimensions $D^{(h)} \times (d + D^{(h)})$ and the vector $(d + D^{(h)}) \times 1$, with $x \in \mathbb{R}^d$ and $h \in \mathbb{R}^{D^{(h)}}$ (Manning & Socher 2017). The initialization vector h_0 is commonly just a vector of all zeros and x_t is a row vector of a large matrix E , defining for example a word (see Section 3.2).

Assuming no further hidden layers, the output of our recurrent layer can be evaluated directly. This can either be done at every output of the hidden layer, for example if we want to predict the next word in a sequence, or only at the end, given we want a binary or categorical classification. For that we can choose different activation functions. We want do do a binary classification and a therefore use the sigmoid activation function. The prediction is then done by

$$\hat{y}_t = sigmoid(W^{(s)} h_t), \tag{4}$$

with $\hat{y} \in \mathbb{R}^{|V|}$ and $W^{(s)} \in \mathbb{R}^{|V| \times D^{(h)}}$. $|V|$ represents the number of possible values y_t can take, for example the whole vocabulary or in the binary case, two (Manning & Socher 2017). Equations 3 and 4 are commonly referred to as forward propagation equations (Goodfellow et al. 2016).

2.2 Model Optimization

To optimize the RNN a loss function L_t has to be specified. The loss function determines the divergence of the prediction from the true parameter values. Depending on the task different loss functions have to be used. This is often some form of maximum likelihood criterion, where the cost function is described as the cross-entropy between the training data and the model distribution (Goodfellow et al. 2016). If we assume binary classification the cross entropy loss can be specified as

$$L = -(y \log(p) + (1 - y) \log(1 - p)) \tag{5}$$

with p being the softmax probability for either class and $y \in 0, 1$ our binary target variable. For evaluating the loss at each concatenation of the RNN a negative log-likelihood of the from

$$L(x_t, y_t) = -\sum_t \log p_{model} \left(y_t \mid \{x_0, \ldots, x_t\} \right), \tag{6}$$

where y_t is the entry from the model output vector $\hat{y}_t$, can be defined (Goodfellow et al. 2016). Goodfellow et al. (2016) note that computing the gradient from this loss function can be computationally expensive for RNN as it involves first a long forward pass through the sequence, followed by a backwards pass of the same length. They also mention that the run time cannot be reduced by parallelization as forward propagation is inherently sequential and all states have to be saved resulting in high memory costs.

A RNN, like most other Neural Networks (NN), is trained by computing the gradient and iteratively adjusting the weights. In case of a RNN this process is called

back-propagation through time as we have go back along the concatenation to the first instance $W^{(hh)}$ or $W^{(hx)}$ was used.

As a simplified example we can look at determining the gradient of the $W^{(hh)}$ weight matrix at time t. The derivative of the loss function at time t, with respect to the weight matrix of the hidden layer, is determined by applying the chain rule

$$\frac{\partial L_t}{\partial W^{(hh)}} = \sum_{k=1}^{t} \frac{\partial L_t}{\partial \hat{y}_t} \frac{\partial \hat{y}_t}{\partial h_t} \frac{\partial h_t}{\partial h_k} \frac{\partial h_k}{\partial W^{(hh)}}, \tag{7}$$

where

$$\frac{\partial h_t}{\partial h_k} = \prod_{j=k+1}^{t} \frac{\partial h_j}{\partial h_{j-1}} \tag{8}$$

is the product of jacobian matrices and total derivative

$$\frac{\partial h_k}{\partial W^{(hh)}} \tag{9}$$

(Pascanu et al. 2013).

This product of partial derivatives (Equation 7) can lead to problems with the gradients resulting in the optimization either failing completely or loosing the ability memorize past information. Both issues where first formally described by Bengio et al. (1994). We separate between two sub-problems. Firstly, by multiplying many large values, gradients can get so large that the weights are updated to NaN such that they cannot be updated anymore. This issue is called exploding gradients and Pascanu et al. (2013) recommend solving it by clipping the gradients if they get too large.
The larger problem are vanishing gradients. They are the result of taking the product of many small partial derivatives (Hochreiter 1991), resulting in very low values for the gradient. As the consequence networks loose their ability to memorize information from earlier parts of the sequence. The cause of this phenomenon can have multiple reasons. Most commonly the derivatives have values close to zero. If the chosen activation function in the recurrent layer is for example a sigmoid function, with a derivative that is bound upwards by 0.25 (Figure 2), products of derivatives can get small fast. As such it is better to use other activation functions like tanh instead. Figure 2 gives an illustration of different activation functions and their derivatives. The advantage of tanh is its derivative is bound upwards by one. Another cause of vanishing gradients can be the initialization of the weight matrices close to zero (Pascanu et al. 2013), this can be solved by specifying initialization values that are not close to zero. The most severe problem is that the sequence length evaluated by the network can make the problem more severe. The shorter the sequence, the less of a problem it poses. To solve this within a simple RNN is difficult, specially if the network is supposed to remember information far in the past. Other network structures building on top of the idea of RNN are therefore introduced in Section 2.3 and 2.4, that uplift this restriction and are better at retaining information far in the

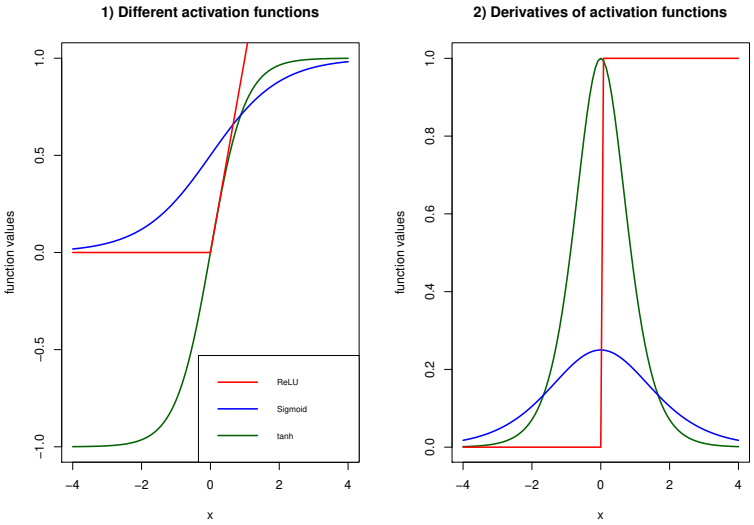

Figure 2. Different activation functions and their derivatives (self)

past. Vanishing and exploding gradients are not unique to RNN, but because of the special structure and their recurrence they are most prominent in networks of this kind.

2.3 Long Short Term Memory

A common extension of RNN are Long Short Term Memory (LSTM) networks which are specialized on connecting information over long sequences. First described by Hochreiter & Schmidhuber (1997) to create a network that is designed to learn long-term dependencies, it introduces a much more complex structure withing the hidden recurrent cells than just the one activation function from before. An illustration of a LSTM in its unrolled state can be seen in Figure 3.

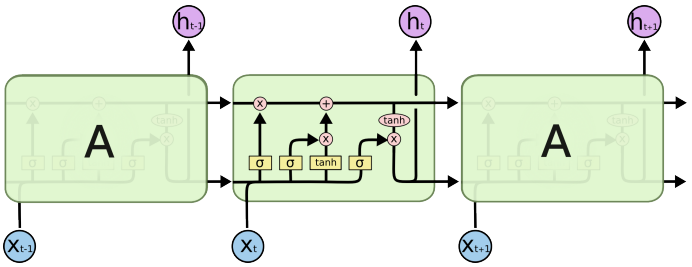

Figure 3. Unrolled Long Short Term Memory network (Olah 2015)

The general functionality of a recurrent neural network stays the same and the se-

quence is still evaluated step by step while taking the previous inputs into account. Newly added is the cell state C_t which is represented by the upper horizontal line, running from left to right. It acts as the memory of the network where information can be stored and removed.

The first upwards arrow to the cell state on the left is called "forget get gate layer" and consists of a single sigmoid activation function

$$f_t = \sigma(W^{(f)} \cdot [h_{t-1}, x_t] + b_f) \tag{10}$$

which outputs a matrix of values between zero and one (Hochreiter & Schmidhuber 1997). By multiplying this matrix with the cell state values close to zero indicate that specific information in the cell state should be forgotten and vice versa.

The two arrows in the middle, that are multiplied and then added to the cell state are called "input gate layer". They form a unit to update the cell state with new information. The update to the cell state is determined by applying a *tanh* activation function, creating a proposed new cell state $\tilde{C}_t$ and multiplying it with the output of a sigmoid activation function. In this case the sigmoid layer acts as a type of update function, where values close to one indicate storing new information. The "input gate layer" cannot delete information from the cell state because it is added to the old cell state only after that. The input gate can be expressed as

$$\begin{aligned} i_t &= \sigma(W^{(i)} \cdot [h_{t-1}, x_t] + b_i) \\ \tilde{C}_t &= tanh(W^{(c)} \cdot [h_{t-1}, x_t] + b_c). \end{aligned} \tag{11}$$

To generate output h_t a *tanh* activation function is applied to the updated cell state at time t and multiplied with the output of a sigmoid function

$$\begin{aligned} C_t &= f_t \cdot C_{t-1} + i_t \cdot \tilde{C}_t \\ o_t &= \sigma(W^{(o)} \cdot [h_{t-1}, x_t] + b_o) \\ h_t &= o_t \cdot tanh(C_t). \end{aligned} \tag{12}$$

The matrix o_t can be considered as a filter and also means that the hidden state h_t does not go directly through the whole network. Only the cell state forms a direct path through the whole sequence of recurrent cells and can be considered as having an identity function as activation. Because of that the LSTM does not suffer from the vanishing gradient problem caused by small derivatives.

The weight matrices $W^{(f)}$, $W^{(i)}$, $W^{(o)}$ and $W^{(C)}$ are now being trained instead of $W^{(hh)}$ and $W^{(hx)}$. This greatly increases the number of parameters a LSTM has to learn. The parameters $b_{(f)}$, $b_{(i)}$, $b_{(o)}$ and $b_{(C)}$ are added bias terms. Jozefowicz et al. (2015) recommend choosing 1 as an initialization for bias $b_{(f)}$ to boost the models learning speed by preventing issues with the gradients. There are also other variations of the classical LSTM that either merge the forget and input gate layers or add peepholes for the hidden state to have a look at the cell state (Gers & Schmidhuber 2000).

2.4 Gated Recurrent Unit

The most common alternative to LSTMs is the Gated Recurrent Unit (GRU) introduced by Cho et al. (2014). It has the advantage of having slightly less parameters to train than a LSTM, while retaining the same memory capacity. Figure 4 represents the structure of a single unrolled GRU cell.

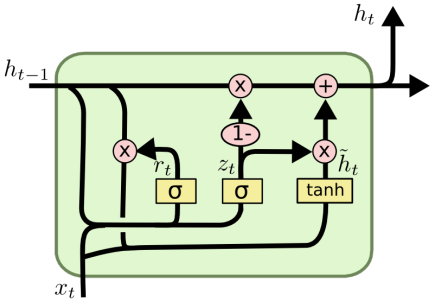

Figure 4. Unrolled Long Short Term Memory network (Olah 2015)

The most notable difference is that it lacks a dedicated cell state like the LSTM. Instead the hidden state is again looped back into the cell. In a GRU the hidden state and the cell state are one and the input and forget gates are also merged. This reduces the number of trainable parameters (Cho et al. 2014). A GRU can be represented by

$$\begin{aligned} z_t &= \sigma(W^{(z)} \cdot [h_{t-1}, x_t]) \\ r_t &= \sigma(W^{(r)} \cdot [h_{t-1}, x_t]) \\ \tilde{h}_t &= tanh(W \cdot [r_t \cdot h_{t-1}, x_t]) \\ h_t &= (1 - z_t) \cdot h_{t-1} + z_t \cdot \tilde{h}_t \end{aligned} \tag{13}$$

A comparison between different LSTM variations can be found in a a paper by Greff et al. (2017). They find that most of these different variations perform about the same. This goes along with Jozefowicz et al. (2015) that resume that there are no different RNN architectures that can consistently outperform LSTM and GRU.

2.5 Recurrent Neural Network Variants

In the field of deep learning depth is considered as the number of hidden layers in a NN and width is the number of hidden units (nodes) each hidden layer posses. The RNN variants we have seen so far in Section 2.1 to 2.4 only had a depth of one and a width depending on the maximum sequence length. These are only some possible variants of recurrent neural network layers. They can also be extended to be deep by returning a sequence as input for the next recurrent layer and are often paired with some non recurrent dense layers, before generating output. Recent developments have also often paired recurrent layers with convolutional layers that are then mostly

used for preprocessing the data. For an example in augmented attention see Xu et al. (2015).

RNN can not only be one directional but also bidirectional (Schuster & Paliwal 1997). The advantage of Bidirectional Recurrent Neural Networks (BRNN) is that they can also take future context information into account and do not require a fixed input length. BRNN have shown to perform very well in language based models (Salehinejad et al. 2017). There are many other variants of RNN, for an overview of advantages and disadvantages of different layers see Salehinejad et al. (2017).

3 Example

To illustrate the previous theory a binary classification task of German Twitter posts regarding offensive and non offensive language was chosen. The data was collected by Wiegand et al. (2018) and originally designed as a shared task to explore the identification of offensive language under the name "GermEval-2018". It consist of a binary classification and a finer differentiation between four categories. Here only the binary classification of offensive language will be attempted. Wiegand et al. (2018) define offensive language as "hurtful, derogatory or obscene comments made by one person to another person". We will first give an overview of the data and how it had to be processed to match the word embeddings in Section 3.2. At last different models are compared in Section 3.3 and problems with training NN are discovered.

3.1 Data

The raw Twitter data from "GermEval-2018" shared task requires a couple preprocessing steps to be useful for the evaluation with Deep Learning models. As we attempt a binary classification task we first have to check if both classes appear equally often. From 5009 observations in total 3321 (66%) are declared as "other" and 1688 (34%) as "offensive". If a model is trained on imbalanced data it tends towards guessing the more frequent class without adjusting its weights properly. This phenomenon is called "accuracy paradox" and results is a poor model performance (Sun et al. 2009). To avoid this different approaches are feasible. Ideally more data is collected. As that is not possible here one can either under-sample cases from the dominant class or over-sample cases from the underrepresented class (Chawla 2009). Other approaches like model penalization or different metrics can also be feasible. For simplicity and due to the small size of the data set the oversampling approach was chosen. This led to a total of 6369 observations.

Due to the special nature of Twitter data, strings have to be preprocessed (Cieliebak et al. 2017). Hashtags and URLs, along with digits that are larger than nine, are replaced by tokens. Some special characters are removed but otherwise most punctuation will be kept as it often contains information in social media posts. Emoji where not considered in these processing steps and where split up into.

Next all the words and symbols are assigned a word token in form of a number.

These are then used to generate unique number vectors for each post. The maximum length was 58 words or symbols long.

At last we split the training data into three parts. For that we first split of 20% of the data for testing purposes. The remaining 80% are then split into 75% for training and 25% for validation. This concludes the data preparation. The corpus of words and symbols is now in a tokenized form consisting of number vectors with a length of 58. Stings with less than 58 words or symbols are padded, longer ones would be truncated.

3.2 Word Embedding

To improve the performance of language based Deep Learning models one often uses word embeddings. These embeddings map a vocabulary into a high dimensional vector space, usually 300 or larger, by analyzing a huge corpus of text. Word embeddings are language specific and can be trained using different criteria (Lavelli et al. 2004). Finding good and in **R** usable German word embeddings is more difficult than for English ones. The word embeddings that where used in the end are from Müller (2019) and are trained using "word2vec" (Mikolov et al. 2013) on a corpus of Wikipedia and news articles from 2013 to 2015.

The advantage of using word embeddings is that they bring in a lot of information about the language they are trained on and how vocabulary interacts. As such they can offset problems when working with small data sets, where some words might be very infrequent and seldom appear during training. The downside to the specific word embeddings by Müller (2019) is, that they do not contain emoji which can be important to understand irony or emotions in social media posts. The common procedure would be to update the pretrained word embeddings with the information on emoji as done by Cieliebak et al. (2017). This was not done here. After the preprocessing steps of the data, about 87% of the words or punctuation signs in the data corpus match with words or signs in the embeddings.

3.3 The Model

For the model design different types of Recurrent Neural Networks were tested. The general structure was inspired by the design chosen by Corazza et al. (2018), who won the "GermEval-2018" task with the best performing model. Their model takes emoji into account and also splits hashtags with the help of n-grams, both were not done here. Four models are compared. All four consist of a fixed embedding layer followed by one or two recurrent layers. The output of the recurrent layers then moves through two dense layers with rectified linear unit activation functions, one with 500 units and the next with 300 units. The last layer is a single unit with sigmoid activation function. Apart from the recurrent layers all hidden layers are the same in all four models.

Model_1 has two bidirectional recurrent layers. The first is a LSTM layer returning a sequence which is used as input for a GRU layer, Model_2 is a bidirectional

LSTM and Model_3 a bidirectional GRU. Model_4 only has a simple recurrent layer. The models were trained using binary crossentropy as loss function. The disadvan-

	Model	Batch	Epoch	Loss	Binary Acc.	F1	F1 (test)	Time
1	Model_1	25	6	0.65	0.64	0.55	0.71	3.84
2	Model_2	25	6	0.63	0.63	0.54	0.69	2.20
3	Model_3	25	6	0.65	0.61	0.52	0.67	1.86
4	Model_4	25	6	0.70	0.47	0.00	0.00	32.82
5	Model_1	25	12	0.62	0.64	0.53	0.70	7.48
6	Model_2	25	12	0.64	0.64	0.59	0.73	4.69
7	Model_3	25	12	0.64	0.64	0.53	0.69	3.67
8	Model_4	25	12	0.69	0.53	0.63	0.79	1.11
9	Model_1	25	18	0.63	0.67	0.58	0.73	10.93
10	Model_2	25	18	0.70	0.65	0.53	0.69	6.32
11	Model_3	25	18	0.63	0.67	0.62	0.75	5.24
12	Model_4	25	18	0.69	0.52	0.56	0.72	1.67
13	Model_1	75	6	0.64	0.63	0.54	0.68	2.69
14	Model_2	75	6	0.65	0.62	0.59	0.74	1.64
15	Model_3	75	6	0.66	0.60	0.53	0.66	1.30
16	Model_4	75	6	0.69	0.54	0.54	0.69	32.79
17	Model_1	75	12	0.64	0.65	0.57	0.71	5.03
18	Model_2	75	12	0.65	0.66	0.60	0.75	3.18
19	Model_3	75	12	0.65	0.63	0.52	0.66	2.64
20	Model_4	75	12	0.69	0.52	0.52	0.64	1.03
21	Model_1	75	18	0.64	0.67	0.55	0.69	7.64
22	Model_2	75	18	0.66	0.66	0.60	0.73	4.96
23	Model_3	75	18	0.64	0.66	0.58	0.72	4.40
24	Model_4	75	18	0.69	0.53	0.55	0.68	1.66
25	Model_1	125	6	0.66	0.63	0.59	0.75	3.30
26	Model_2	125	6	0.64	0.63	0.58	0.73	2.21
27	Model_3	125	6	0.65	0.62	0.54	0.68	1.88
28	Model_4	125	6	0.69	0.53	0.63	0.79	52.92
29	Model_1	125	12	0.63	0.65	0.54	0.67	7.56
30	Model_2	125	12	0.63	0.64	0.55	0.70	5.84
31	Model_3	125	12	0.66	0.62	0.51	0.65	4.93
32	Model_4	125	12	0.69	0.51	0.54	0.68	1.85
33	Model_1	125	18	0.63	0.65	0.54	0.68	17.87
34	Model_2	125	18	0.64	0.65	0.56	0.73	11.56
35	Model_3	125	18	0.65	0.65	0.52	0.64	9.77
36	Model_4	125	18	0.68	0.56	0.63	0.79	3.31

Table 1. Four different models trained with nine different specifications of batch size and number of epochs.

tage of doing so can be illustrated using a short example. If the output layer returns a vector for four different strings of the form $(0.3, 0.2, 0.44, 0.36)$ then the binary crossentropy with a threshold of 0.5 sets this to $(0, 0, 0, 0)$. Assuming that the true target is $(0, 0, 1, 0)$, binary accuracy is 75%. Although this indicates high accuracy it needs to be treated with care. If the target of the NN is to classify all tweets with

offensive language correctly, then binary crossentropy is not a good measure for the loss. In that case a custom metric that measures the accuracy in one respective class is recommended. The "GermEval-2018" shared task measured models performance with the F1-score, which gives equal weight to both classes. It represents the harmonic mean between precision and recall and its disadvantages are outlined by Hand & Christen (2018). We failed to specify a working custom loss function in **R** and where forced to work with the binary crossentropy. The F1-score is still computed as custom metric to measure model performance after training.

Each of the four models was trained nine times with different specifications, resulting in 36 model fits. The nine different specifications had batch sizes of 25, 75 and 125, with 6, 12 and 18 training epochs each. The result can be seen in Table 3.1.

The table reports the name of the model along with the batch size and number of training epochs to look for trends during training. As performance measures loss, binary accuracy and F1-score on the test data that was split of the training data. The F1-score of the real test data with $n = 3532$ is displayed as "F1 (test)". Finally the computation time, using three CPU processors, is reported in the last column. The average computation time was 7.6min, although very long computation times for Model_4 seem to be the cause of the high mean.

If we take a look at performance, by comparing the F1-score, there appears to be no trend regarding the choice of hyperparameters. Values are consistently between 0.51 and 0.63 disregarding the one zero value for Model_4. The models even achieve F1 values as high as 0.79 on the test data.

		True		
		Other	Offensive	Total
Prediction	Other	26	13	39
	Offensive	1176	2317	3493
	Total	1202	2330	3532

Table 2. Confusion matrix for Model_4 in row 36.

The best performing model from Corazza et al. (2018), that won the competition, achieved a F1-score of 0.68. This indicates that we might have some issues that are not displayed by the F1-score. By looking at a confusion matrix for Model_4 in row 36 (Table 3.2) one can see that the simple RNN mostly predicted "offensive". This led to a sensitivity of 0.02 and a specificity of 0.99 resulting in such a high F1-score. The cause of this problem is probably the binary crossentropy loss used for training the model.

We can look at Model_1 in row 12 of Table 3.1 for another comparison. The confusion matrix (Table 3.3) for the test data that achieved an F1-score of 0.72 has a much better allocation of predictions. As a result sensitivity is 0.50 and a specificity of 0.63. These results are to be favored compared to the simple RNN layer as we are

		True		
		Other	Offensive	Total
Prediction	Other	598	858	1456
	Offensive	604	1472	2076
	Total	1202	2330	3532

Table 3. Confusion matrix for Model_1 in row 12.

trying to predict both classes as accurate as possible.

Similar observations can be made by examining Model_2 and Model_3. Both have much better values for sensitivity and specificity compared to the simple RNN in Model_4.

4 Conclusion

We have shown how Neural Networks can be adapted to evaluate sequences of flexible length and connect information at different time points. The particularity of Recurrent Neural Networks is that they loop the output of each time step back into itself. Because this can increase the chance of vanishing and exploding gradients, solutions were proposed. Clipping large gradients was recommended to prevent exploding gradients and different, advanced RNN architectures like Gated Recurrent Units and Long Short Term Memory networks where proposed to prevent vanishing gradients and boost the memory capacity of RNN.

Chapter 3 introduced the applicability of RNN to Natural Language Processing by applying four different RNN layers to a binary classification task of Twitter posts in German. Additionally the use and benefit of word embeddings was described. We where able to show that the number of training epochs and the batch size had little effect on the model performance. Further LSTM and GRU layers seemed to be better adapted to the task of binary classification with social media data. Specially the simple RNN tended towards guessing only one class, resulting in a decent score, but low performance. This illustrated how important it is to choose the right loss for training and choosing certain meaningful evaluation metrics. Because of the special nature of NN issues with misclassification can be more difficult to notice than in classical statistics. For further reading on advanced RNN and state of the art developments Xu et al. (2015) can be recommended.

References

Bengio, Y., Simard, P., Frasconi, P., et al. 1994, IEEE transactions on neural networks, 5, 157

Chawla, N. V. 2009, in Data mining and knowledge discovery handbook (Springer), 875–886

Cho, K., Van Merriënboer, B., Gulcehre, C., et al. 2014, arXiv preprint arXiv:1406.1078

Cieliebak, M., Deriu, J. M., Egger, D., & Uzdilli, F. 2017, in 5th International Workshop on

Natural Language Processing for Social Media, Boston, MA, USA, December 11, 2017, Association for Computational Linguistics, 45–51

Corazza, M., Menini, S., Arslan, P., et al. 2018, in GermEval 2018 Workshop

Gers, F. A. & Schmidhuber, J. 2000, in Proceedings of the IEEE-INNS-ENNS International Joint Conference on Neural Networks. IJCNN 2000. Neural Computing: New Challenges and Perspectives for the New Millennium, Vol. 3, IEEE, 189–194

Goodfellow, I., Bengio, Y., & Courville, A. 2016, Deep Learning (MIT Press), `http://www.deeplearningbook.org`

Greff, K., Srivastava, R. K., Koutník, J., Steunebrink, B. R., & Schmidhuber, J. 2017, IEEE transactions on neural networks and learning systems, 28, 2222

Hand, D. & Christen, P. 2018, Statistics and Computing, 28, 539

Hochreiter, S. 1991, TU Münich

Hochreiter, S. & Schmidhuber, J. 1997, Neural computation, 9, 1735

Jozefowicz, R., Zaremba, W., & Sutskever, I. 2015, in International Conference on Machine Learning, 2342–2350

Lavelli, A., Sebastiani, F., & Zanoli, R. 2004, in Proceedings of the thirteenth ACM international conference on Information and knowledge management, ACM, 615–624

Manning, C. & Socher, R. 2017, CS224n: Natural Language Processing with Deep Learning (Winter 2017), `https://web.stanford.edu/class/archive/cs/cs224n/cs224n.1174/`, accessed: 2019-03-24

Mikolov, T., Chen, K., Corrado, G., & Dean, J. 2013, arXiv preprint arXiv:1301.3781

Müller, A. 2019, German Word Embeddings, `https://devmount.github.io/GermanWordEmbeddings/`, accessed: 2019-03-05

Olah, C. 2015, Understanding LSTM Networks, `https://colah.github.io/posts/2015-08-Understanding-LSTMs/`, accessed: 2019-03-24

Pascanu, R., Mikolov, T., & Bengio, Y. 2013, in International conference on machine learning, 1310–1318

Rumelhart, D. E., Hinton, G. E., & Williams, R. J. 1986, Learning internal representations by error propagation, Tech. rep., California Univ San Diego La Jolla Inst for Cognitive Science

Salehinejad, H., Sankar, S., Barfett, J., Colak, E., & Valaee, S. 2017, arXiv preprint arXiv:1801.01078

Schuster, M. & Paliwal, K. K. 1997, IEEE Transactions on Signal Processing, 45, 2673

Sun, Y., Wong, A. K., & Kamel, M. S. 2009, International Journal of Pattern Recognition and Artificial Intelligence, 23, 687

Wiegand, M., Siegel, M., & Ruppenhofer, J. 2018, in 14th Conference on Natural Language Processing KONVENS 2018

Xu, K., Ba, J., Kiros, R., et al. 2015, in International conference on machine learning, 2048–2057