Realisierung einer Next Word Prediction mit Hilfe eines rekurrenten neuronalen Netzwerkes

Jakob Stadelmann
Inf-B
Technische Hochschule Ingolstadt
Professionale Textsatzsysteme
jas4613@thi.de

***Zusammenfassung*—This paper documents the implementation of next word prediction. For this purpose, a recurrent neural network was trained with a self-compiled data set. First, an introduction to this technology is given. The steps required to implement next word prediction are then explained. The optimization of the parameters of the recurrent neural network was a major focus. In the course of this work, it was shown that even with limited possibilities in terms of data set and computing power, it is possible to create an already usable Next Word Prediction.**

I. Introduction

The smartphone has become our daily companion and the number one means of communication. One technology has made writing and composing messages and texts on the smartphone much easier, namely Next Word Prediction. Whether we're writing a WhatsApp or researching something on the internet, our smartphone somehow always knows what we want to write in advance. However, this is not magic or anything like that, but rather artificial intelligence (AI). Large companies such as Facebook and Google have almost perfected this next word prediction. Almost every word is predicted correctly. These companies have access to a huge data set and enormous computing power to train their models. But what if you don't have access to a large text dataset and astronomical computing power? Is it still possible to realize a helpful next word prediction? In this work, an attempt is made to realize a word prediction with low resources, i.e. using low computing power with a small data set. The implementation is carried out by selecting a suitable architecture for the model and optimizing the model parameters.

II. Fundamentals

Next Word Prediction is realized in this project with the help of a recurrent neural network. This section explains the technology and its different architectures.

A. Recurrent Neural Network

With conventional neural networks, it is difficult to solve temporal problems such as speech processing, speech recognition or image recognition, as they cannot process time-dependent data [7]. Previous events cannot be used to generate the prediction, which is a major weakness and makes solving the problem much more difficult. This would be equivalent to human thinking, which can only work with current information every second without stored knowledge. Understanding a text would not be possible as we normally understand a word based on our understanding of the previous words. This challenge of processing time-dependent data can be solved by so-called recurrent neural networks, or RNNs for short. The previous information is retained by the network as it consists of loops [14]. The structure is shown in Fig. 1 is shown. You can imagine it as follows: An RNN is several copies of the same network and each one forwards a message to its successor. A is a part of the neural network that receives the input x_t and outputs h_t. The passing on of information from one level of the network to the next is made possible by the loop.

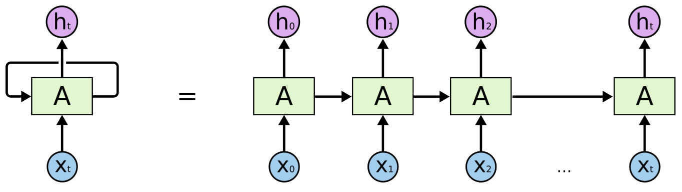


Abbildung 1. Rolled recurrent neural network [14]

This chain-like nature of RNNs is similar to sequences and lists. RNNs are the natural architecture for such data types. Just like feedforward neural networks (FNN) and convolutional neural networks (CNN), RNNs use training data to learn. The advantage of this is 'memory', where information from previous inputs is used to influence current inputs and outputs. The output of recurrent neural networks depends on the previous elements within the sequence [7]. A distinction is made between two further developments of RNNs, which are explained below.

B. LSTM

Standard RNNs are sufficiently good at remembering recent events, so-called short-term dependencies. When predicting the next word in the sentence 'The car is driving on the *road*', the previous context is sufficient to predict 'road'. However, if the context needed to predict a word is several sentences before, it will be difficult for the standard RNN to connect this information. For the text 'I study computer science... I am interested in *computer*.' it becomes almost impossible to predict 'computer'. There are several fundamental reasons for this, which were explored in depth by Bengio et.al [2] in 1994. One of these reasons is that the gradient descent becomes inefficient as the period of dependence increases. The development of Long Short Term Memory networks(LSTMs), which were introduced in 1997 by Hochreiter & Schmidhuber [9], means that long-term dependencies can also be queried. LSTMs are a special type of RNNs and it is almost their natural behavior to remember information over long periods of time [14]. The difference to conventional RNNs is the structure of the repeating modules. As shown in Fig. 2, standard RNNs use a very simple structure, such as a single tanh layer.

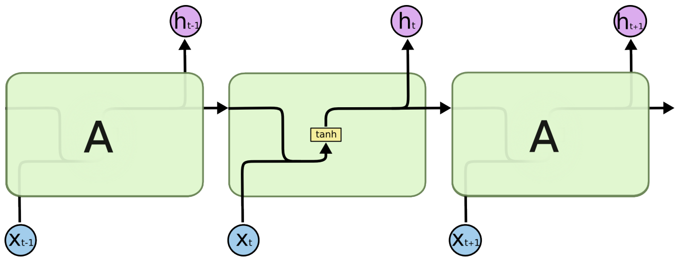


Abbildung 2. Repeating module in a standard RNN [14]

LSTMs, on the other hand, use not just a single neural network layer, but four layers that interact with each other in a fixed sequence, as shown in Fig. **??** shown.

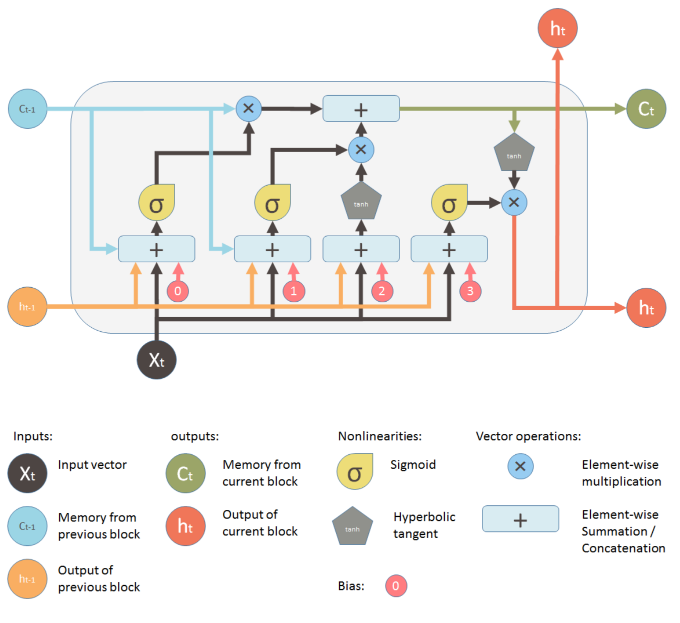


Abbildung 3. Repeating module in an LSTM [19]

The first decision is made by a sigmoid layer, also known as the 'forget gate layer'. It determines which information should remain or be forgotten. New information is then generated/stored in the 'input gate' in a two-stage process. First, existing information is updated. In a second step, a new vector for the information value is created via a tanh layer. Both steps combined result in the new cell status. Finally, the implementation of the previous decisions is carried out in the output gate layer [14].

C. Bidirectional LSTM

The bidirectional RNN is another variant of the RNN. It was developed by Schuster and Paliwal [17]. They had the idea of dividing the state neurons of a regular RNN into two parts. One part is responsible for the positive time direction (forward states) and the other part is responsible for the negative time direction (backward states). The outputs of one state are not connected to the inputs of the other state. By considering both time directions in the same network, it can be trained on both past and future input information. This saves the delay of including future information, as would be the case with regular unidirectional RNNs [17]. In fig. 4 shows the structure of such a bidirectional RNN.

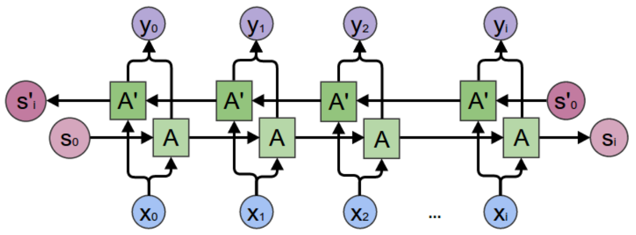


Abbildung 4. Construction of a bidirectional recurrent neural network [13]

Following this principle, the Bidirectional LSTM(BLSTM) uses two layers. One layer performs the operations in the same direction of the data sequence and the other layer applies its operations in the opposite direction of the data sequence [1].

III. Procedure

After clarifying the most important terms, the steps involved in implementing Next Word Prediction are explained below.

A. Description of the Data

A data set is first created for the subsequent training of the RNN. This data set forms the basis of the entire project and is crucial for the quality of Next Word Prediction. If the data set is too small, there is a risk of overfitting when training the model. The data set used is not optimal because it is not too large. Generally speaking, the more data, the better a model can be trained. However, it is very difficult to find a publicly accessible text dataset that is of good quality and large enough. The training also worked with the smaller data set and it had the advantage that the training time was not too long. Several texts by the English writer H. G. Wells were used as a data set, which were merged into a large data set. These texts are freely accessible in English as part of the Gutenberg project. The data set has a size of 2Mb and contains 22252 unique words.

B. Data Preprocessing

However, the raw text dataset still needs to be adapted for training. There are various steps to prepare the text data set for training, this process is called data preprocessing. The first step is to remove capital letters. Then all numbers are removed from the text. Subsequently, all punctuation and special characters are removed. In some places, double spaces are created as a result of the removal; these must also be removed at the end. Another method is to remove so-called stopwords. These are words that occur very frequently. Examples of such words in German would be: der, die, was etc... . In the text used here, the stopwords make up approx. 45% of the entire text. Due to this large proportion, it can happen that the model learns that the stopwords occur very frequently and therefore only outputs these as predictions. This is undesirable, as the model should recognize the context of the input and base its predictions on this. Removing stopwords also has the major disadvantage that, logically, no more stopwords can be predicted. A decision still needs to be made as to whether the stopwords should be removed. Various evaluation methods are used later to make this decision. Data preprocessing is necessary to remove irrelevant information for next word prediction from the text. This allows the size of the data set to be reduced. This has the advantage that the training time is shorter and at the same time the quality of the data set is increased. Below is an example of what the text looked like before preprocessing (raw data) and after (filtered data).

Rohdaten: be convenient to speak of him) was
expounding a recondite matter to us. His pale grey eyes shone and
twinkled, and his usually pale face was flushed and animated. The fire
burnt brightly, and the soft radiance of the incandescent lights in the
lilies of silver caught the bubbles that flashed and passed in our
glasses. Our chairs, being his patents, embraced and caressed us rather
than submitted to be sat upon, and there was that luxurious
after-dinner atmosphere, when thought runs gracefully free

Gefilterte Daten: be convenient to speak of him was expounding recondite matter to us his pale grey eyes shone and twinkled and his usually pale face was flushed and animated the fire burnt brightly and the soft radiance of the incandescent lights in the lilies of silver caught the bubbles that flashed and passed in our glasses our chairs being his patents embraced and caressed us rather than submitted to be sat upon and there was that luxurious after dinner atmosphere when thought runs gracefully free

C. Tokenization

The text data record must now be broken down into words, phrases, symbols or other informative elements. This process is known as tokenization. During tokenization, a unique index is created for each word in the text. The result is a kind of dictionary (tokenizer) in which each word is assigned an index (number). The text consisting of words is now converted into a sequence of numbers. This is done so that vectors can then be created from the indexes for training. After tokenization, the text looks like this:

Eingabe-Text: the time machine an invention by g wells contents introduction ii the machine iii the time

Indexe: [1, 49, 245, 2196, 22, 3532, 2197, 3263, 2633, 648, 1, 245, 851, 1, 49, 476]

D. Sequence Creation

For training, the data still needs to be split into the input sequences and the corresponding responses. The model will later receive the input sequences during training and creates its prediction based on them. This prediction is then compared with the response to check whether the model has made a correct prediction. Depending on this, the weights and biases in the network are then adjusted. In this approach, sequences with a length of 4 words were initially used, this value is also referred to as the sequence length. In other words, 4 words were always drawn and the following word was then used as the response. For the next sequence, the words were moved one word to the left. The first word of the old sequence was discarded and the old answer is now part of the new input sequence. The new answer is then the following word. Using this procedure, the entire text is divided into input sequences and answers. To increase the variance in the data set, the sequences and the corresponding answers are now randomly shuffled as shown in Fig. 6 is shown. In each case, the input sequence and the corresponding response are in one block. This prevents dependencies that arise with a fixed sequence from being learned. As a result, the quality of the data set is increased. In fig. 5 shows how the sequences are present before they are shuffled.

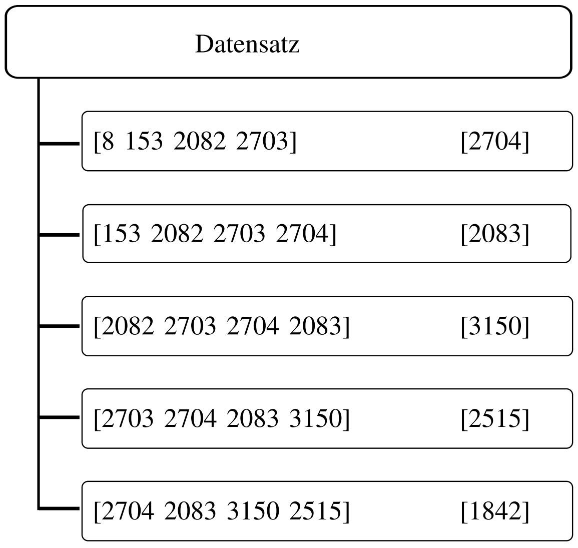


Abbildung 5. Not shuffled data set

The answers are converted into vectors using the one-hot encoding process. Until now, the answers were still available as indexes of a word from the dictionary. These are now converted into a vector that has the size of the dictionary. This vector is assigned a 0 at every position, except at the position of the respective index. There is a 1 there.

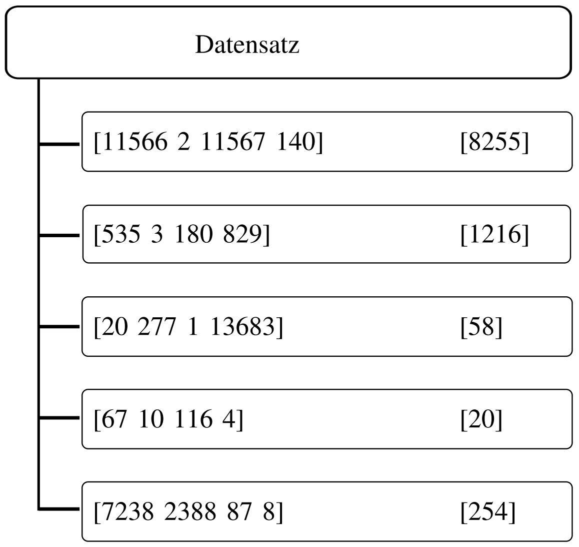


Abbildung 6. Shuffled data set

E. Structure of the Neural Network

Two different architecture models are compared in this paper. The first model works with two LSTM layers, while the second model uses one BLSTM layer. Among other things, it is evaluated which model is better suited for next word prediction. Otherwise, the structure of the models is identical. First, the input sequences are transferred to an embedding layer. The input data must also be adapted so that the neural network can work with it. However, using the one-hot-encoding method again would take up too much memory. With the embedding layer, each word can be converted into a vector of fixed length and defined size (embedding size) [16]. The result is a so-called dense vector with real values instead of 0 or 1, as would be the case with a one-hot-encoded vector. Due to the fixed length of the word vectors, the words are better represented and at the same time the dimensions are reduced [16].

Example for dense vectors: ('the', 'he') = [0.00234551 0.00112815 0.28401764 -0.9050087], [-0.04001870 0.74302836 0.02436277 0.00820647]

In this case, the embedding size (dimensions) is four; typical embeddings have significantly larger dimensions. This is followed by either the two LSTM layers or the BLSTM layer. How these work has already been explained in the basics. This is followed by the dense layer. In the model with two LSTM layers, there are two dense layers. The dense layer bears its name because it is closely connected to the previous layer. Each neuron of the layer is connected to each neuron of the previous layer and receives an input [18] from each of the previous neurons. The neurons of the dense layer perform a matrix-vector multiplication with these inputs. As output, the dense layer returns an 'm' dimensional vector [15]. The dense layer can therefore be used to change the dimensions of a vector. The entire structure of the architecture used can be seen again in the following code.

```
def build_model_bidirectional_lstm(nodes,
    embedding_size, sequence_len, vocab_size):
    model = Sequential()
    model.add(Embedding(vocab_size, embedding_size,
    input_length=sequence_len))
    model.add(Bidirectional(LSTM(nodes)))
    model.add(Dense(vocab_size,
    activation='softmax'))
    return model
```

Softmax is used as the activation function in the dense layer. The network passes a vector with raw outputs to the Softmax function and the Softmax function creates a vector with probability values [4]. This vector again has the size of the dictionary (tokenizer). A probability is therefore obtained for each word in the dictionary. The word with the highest probability is then returned as a prediction. The formula of the softmax function [4] looks like this:

$$softmax(z)_i = \frac{e^{z_i}}{\sum_{j=1}^{N} e^{z_j}} \tag{1}$$

with z = vector with raw outputs from the neural network, N = number of classes(number of words)

F. Model Training

Before training can begin, the data set must be split up. 70% of the data set is used as the training data set with which the model is trained. The rest is used as a test data set to obtain meaningful results during model evaluation. These two data sets must not overlap under any circumstances, otherwise the evaluation of the model after training will be falsified. The aim of the training is to achieve the best possible fit on the data. In the left graph of Fig. 7, you can see that not all points (responses) in the graph are covered by the line (predictions of the model). The data is not represented well enough by the model. The model is underfitted, it is also said to have a high bias [3]. The graph on the right-hand side in Fig. 7 shows that the model accurately maps all points in the graph. However, this is not good because every point that is a noise or outlier is also mapped. Such a model is overfitted and also provides poor predictions due to its complexity [3]. It is also called high variance. The aim is for the model to represent the majority of responses while ignoring the outliers and noise. For the graph in the middle of Fig. 7 is such a model.

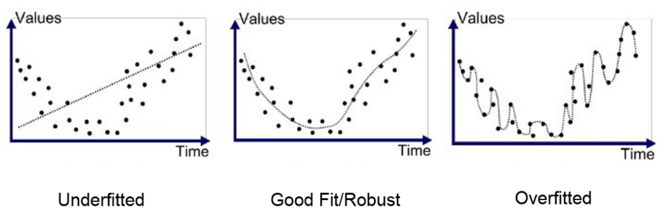


Abbildung 7. Fit of a model [3]

Cross-entropy is used as a loss function for multi-class classification where there are two or more output forks. If the output forks are available as one-hot-encoded vectors, as is the case in this work, categorical crossentropy [12] is used.

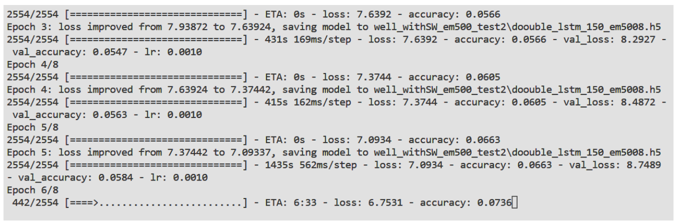
Formula of the categorical crossentropy [8]:

$$CE = -\log \frac{e^{z_p}}{\sum_{j}^{N} e^{z_j}} \tag{2}$$

where z = vector with raw outputs from the neural network, z_p = vector of the neural network for the positive class, N = number of classes(number of words)

G. Model Evaluation

There are various ways of evaluating the model. Three different methods are used in this project. The first method uses validation data in order to be able to make statements about the performance of the currently training model during training. The training data set is split as follows: 90% of the data set is used for training and 10% for validating the model. Again, it is important that the data sets do not overlap under any circumstances. The purpose of the validation data set is to test the model on unknown data. It can happen that the model maps the training data too well and it looks as if the model is training well, but in reality the model overfits and only learns the training data "by heart". In Fig. 8 you can see an excerpt of a training with validation data. In this example, the model has overfitted, which can be seen from the fact that the validation loss(val_loss) increases over the course of the training over the epochs, while the training loss continues to fall. In general, you can say that something is wrong with the training if the results from the validation dataset do not match those from the training dataset.



```
2554/2554 [==============================] - ETA: 0s - loss: 7.6392 - accuracy: 0.0566
Epoch 3: loss improved from 7.93872 to 7.63924, saving model to well_withSW_em500_test2\doouble_lstm_150_em5008.h5
2554/2554 [==============================] - 431s 169ms/step - loss: 7.6392 - accuracy: 0.0566 - val_loss: 8.2927 -
 val_accuracy: 0.0547 - lr: 0.0010
Epoch 4/8
2554/2554 [==============================] - ETA: 0s - loss: 7.3744 - accuracy: 0.0605
Epoch 4: loss improved from 7.63924 to 7.37442, saving model to well_withSW_em500_test2\doouble_lstm_150_em5008.h5
2554/2554 [==============================] - 415s 162ms/step - loss: 7.3744 - accuracy: 0.0605 - val_loss: 8.4872 -
 val_accuracy: 0.0563 - lr: 0.0010
Epoch 5/8
2554/2554 [==============================] - ETA: 0s - loss: 7.0934 - accuracy: 0.0663
Epoch 5: loss improved from 7.37442 to 7.09337, saving model to well_withSW_em500_test2\doouble_lstm_150_em5008.h5
2554/2554 [==============================] - 1435s 562ms/step - loss: 7.0934 - accuracy: 0.0663 - val_loss: 8.7489
- val_accuracy: 0.0584 - lr: 0.0010
Epoch 6/8
 442/2554 [====>.........................] - ETA: 6:33 - loss: 6.7531 - accuracy: 0.0736
```

Abbildung 8. Training with validation data

Another method is to look at the graph of the training process after training a model. There you can also quickly find out whether the model has trained well or not. There is a graph of the training in which the course of the training accuracy (dark blue) see 10 and the validation accuracy (light blue) see 9 is shown. The other graph shows the progression of the training loss(dark blue) and the validation loss(light blue). In the optimal case, both accuracies should continue to increase, while the loss should continue to fall. Here too, the different behavior of the loss graphs is an indication of overfitting.

Informally, we can say that the accuracy is the proportion of predictions that our model got right [5]. The loss is the penalty for a bad prediction. That is, the loss is a number that indicates how bad the model's prediction was for a single example [6]. If the prediction is perfect, the loss is zero, otherwise the loss is greater. In the figure 11, for example, you can see a model with a high loss on the left and a model with a low loss on the right. The aim of training a model is to adjust the weights and biases so that, on average, all examples have a low loss [6].

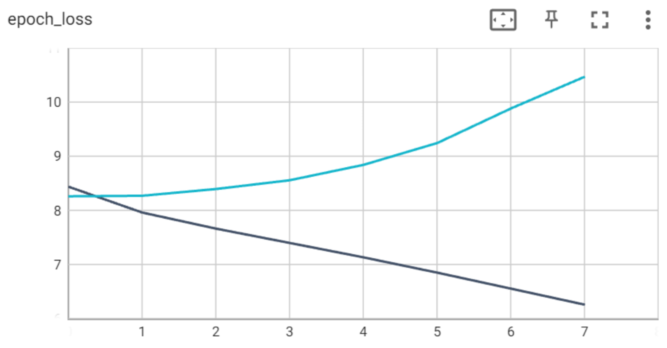


Abbildung 9. Example of training chart (Loss)

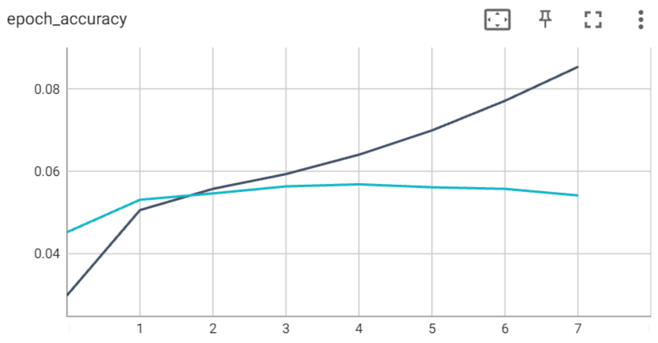


Abbildung 10. Example of training chart (Accuracy)

In the last method, the predictions of the model are visualized. The advantage is that you can see directly what the model actually predicts and do not have to interpret graphs or figures all the time. Based on the actual predictions, anomalies quickly become apparent. The model creates the predictions based on the test data set in order to verify the quality of the model. In the table Tab. **??** shows a section of this visualization. It is important to note that this is an excerpt of the predictions that were correctly predicted by the model.

Tabelle I
VISUALIZATION OF THE PARAMETERS

input sequences	***next word***	***prediction***	***Confidence***
from print the british	museum	museum	[0.9337]
rolling stock on the	liverpool	liverpool	[0.0984]
terrified at the possibilities	of	of	[0.9226]
about tenth of what they	had	had	[0.6761]

The visualization is good for checking. You can see at a glance whether the model predicts "normally" or whether there are anomalies. This procedure was used to decide whether the stopwords should be used for training. As already mentioned in Chap. II. B, there can be the problem that a model only predicts stopwords if they have not been removed from the text dataset. This was not the case here and the model that was trained with stopwords made similar predictions to the model that was trained without stopwords. It is also important to look at the confidence of the model. If the model has a confidence of 90% or higher for most of the correct predictions, it can be assumed that it has trained well and not overfitted.

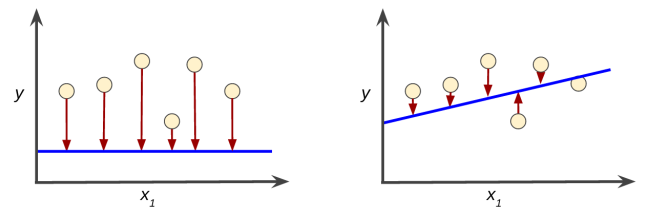


Abbildung 11. Representation of the loss of a model [6]

IV. Optimization of the Model

There are many different parameters in an RNN. Each of the parameters has an influence on the training of the RNN. In the optimization phase, it is now important to assign the parameters as well as possible so that the model can improve its performance. Performance includes the accuracy and loss of the model, as well as the results when visualizing the predictions. In this document, the focus was on selecting a suitable architecture model and optimizing the following three parameters:

- Sequence length: The sequence length specifies how many words are used as input for the training. In general, the value of the sequence length should be as large as possible, as this allows more data to be considered for the prediction. A sequence length of 10 was set as the initial value. However, there are two limiting factors: 1. conventional computers can no longer cope with the amount of data if the sequence length is set too high and this results in the training being aborted. 2. the training duration increases significantly as more data has to be processed by the network. So you have to weigh up time and performance again and find the right compromise.
- Embedding size: The embedding size determines how large the vector is that is generated by the embedding layer. As we have already seen, this is where the words are converted into vectors. The size of the vector plays a major role in how well the embedding works. All words in the text data set must be able to be displayed with the vector. The vector must not be too small, otherwise information would be lost in the embedding process. However, the vector must not be too large either, otherwise either the storage space will not be sufficient or the training duration will be too long. An embedding size of 100 was initially selected as the starting value.
- Number of nodes in the layers: the value should be adapted to the complexity of the text data set. This is similar to the embedding size: if the number of nodes is too low, not all of the information contained can be mapped and therefore not taken into account for the prediction. If the number of nodes is set too high, the training duration suffers again as more nodes have to be trained. The initial value here is 128.

The parameters are optimized one after the other. This is very important because if you change two parameters at the same time, for example, you cannot determine which parameter has made which difference. In this case, the sequence length is improved first. To do this, several neural networks with different sequence lengths are trained. The networks are then compared with each other. If necessary, this process is repeated several times until a suitable value is found. This process is also used for the other parameters so that a model with the best possible parameters is obtained at the end.

A. Trainings Pipeline

A so-called training pipeline was set up to optimize the model in order to make this process as efficient as possible. This pipeline automates the preparation of the data, the training and the evaluation of several neural networks. This has the advantage that each of the networks has been trained on exactly the same data, which means that they can then be compared with each other. First, the text data set is read in and then prepared for training. Several different models are then created and trained. The focus is always placed on one parameter that is to be optimized. The trained networks are then automatically evaluated with the test data set and the predictions are visualized. The process is also shown in Fig. 12 graphically illustrated. The process can also be traced using the code example.

```
def trainings_pipeline():
    file="wells.txt"
    data=get_data(file)
    data=preprocess_data(data)
    sequence_data, vocab_size,
    entrys=tokenize_data(data, foldername)

    sequence_len = 15

    # Aufteilen in Trainings- und Testdatensatz
    idx_train=int(entrys * 0.7)
    idx_test=int(entrys)
    data_train =
    create_sequences(sequence_data[:idx_train],
    sequence_len, vocab_size, False)
    data_test =
    create_sequences(sequence_data[idx_train:
    idx_test], sequence_len, vocab_size, False)

    embedding_size = 500
    epochen = 8

    for nodes in range(32, 128, 32):
        model=build_model_bidirectional_lstm(nodes,
    embedding_size, sequence_len, vocab_size)
        start_time=time.time()
        train_model(data_train, model, epochen)
        print("Trainingsdauer gesamt: ")
        print(time.time() - start_time, "seconds")
        test_model(data_test, model)
        visualization_of_model(data_test, model)
```

B. Model architecture

First, the two model architectures were compared with each other. The criteria for a 'good' architecture were firstly the training duration and secondly the performance of the model after training. In order to be able to compare the two architecture models, they were trained on the exact same data set and the time required for the respective training was recorded. Both models were trained with 4, 8 and 16

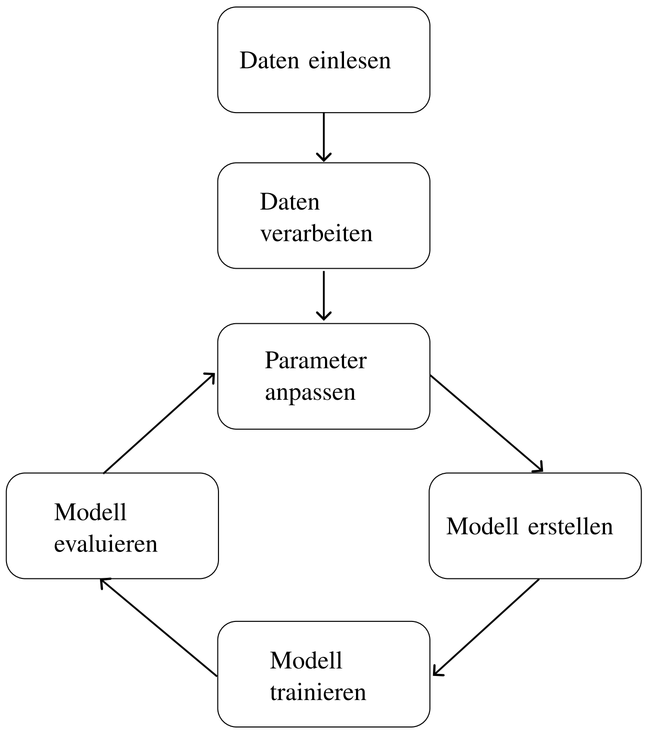


Abbildung 12. Training pipeline

epochs respectively, and the initial parameter values defined above were used so that the models were compared on the same basis. The result of this comparison was as follows: The model with two LSTM layers has a lower training time, while the model with the BLSTM layer has a lower loss and higher accuracy. This result is also consistent with the result from [11]. Both architecture models therefore have certain advantages, one has a lower training duration and the other a better performance. The BLSTM model took an average of 7 minutes per epoch, while the model with the double LSTM layer took 6 minutes and 15 seconds. On the other hand, the accuracy of the first model was 15%, while the second model could only achieve 11%. The loss was neglected because it was around 3.86% for both models. Since the gain in 4% accuracy outweighed the time savings, the architecture with the BLSTM layer was used for the rest of the optimization. It was also shown that both architecture models start to overfit from nine epochs. Therefore, the remaining models were always trained with a number of epochs of eight.

C. Parameter anpassen

Now that a model architecture has been agreed upon, the optimization of the parameters can begin. First, the sequence length was adjusted. As mentioned above, the longer the sequence length, the more information is displayed. The aim was therefore to find the range in which the performance is as high as possible and at the same time the training duration is still acceptable. Sequence lengths of 5, 10, 15 and 20 were used in the first test. The remaining parameters were still set to the default values. The best performance was achieved with a sequence length of 15. An accuracy of 24%, a loss of 2.52 was achieved with a training duration of 7 minutes and 30 seconds. Based on this result, further runs were carried out with a sequence length of 8 and 15. It was found that a sequence length of 15 was the most suitable in this case.

The next parameter to be adjusted was the embedding size. In the first test, three different embedding size values were compared with each other. To find a starting value, the following rule of thumb from [10] was used: The embedding dimension should be approximately 1.6 times the square root of the number of unique elements. In this case, there are 22252 unique words, resulting in an embedding size of 240 according to this rule. This value was used as a starting value and an additional value above (300) and below (100) was selected to narrow down the range. The first test showed that the model with an embedding size of 300 delivered the best results. The model outperformed the model(Embedding-Size=100) by approx. 6% accuracy and the model(Embedding-Size=240) by approx. 4% accuracy. The training duration was in a similar range for all three models and therefore negligible. In the next experiment, the focus was once again on the embedding size. Now the values were 300, 400, 500, 600. Values greater than 300 were selected, as the lower values performed worse. The model with an embedding size of 500 achieved the best result. It achieved an accuracy of 37% with an epoch duration of 8 minutes and 10 seconds. The loss of the model was 2.36.

The next step was to optimize the number of nodes in the layers. This involves the number of nodes in the bidirectional layer and the dense layer. The two layers always have the same number of nodes. We started with the values 64, 128 and 256. The models with a node count of 128 and 256 overfitted significantly faster than the model with a node count of 64, which makes sense, as the data set was not too large and, as mentioned above, the node count should be similar to the complexity of the text data set. In the second test, 32, 64 and 96 were used as values. As in the first test, the model with a node count of 64 was again the best. It had the highest accuracy (41%) and the lowest loss (1.98). The training duration of the model was 8 minutes and 23 seconds per epoch.

The selected parameters were all optimized and the results are amazing. From an initial accuracy of 15%, the model improved to an accuracy of 41%. At the same time, the loss was reduced from 3.86 to 1.98. At the same time, the training duration per epoch only increased by 1 minute and 20 seconds. The results are all shown in Tab. **??** summarized.

Tabelle II
RESULTS OF THE OPTIMIZATION

Optimization	***Accuracy***	***Loss***	***Time per epoch***
Architecture	15%	3,86	7:00min
Sequence length	24%	2,52	7:30min
Embedding size	37%	2,36	8:10min
Number of nodes	41%	1,98	8:20min

V. CONCLUSIOIN

As part of this work, a next word prediction was realized based on the use of a recurrent neural network. The final network consisted of an embedding layer, followed by the bidirectional LSTM layer and a dense layer with 64 nodes each. The developed model was trained with a relatively small data set and a PC with standard computing power. The initial word prediction accuracy was 15%. By optimizing the model parameters, an accuracy of 41% could be achieved. The final assignment of the parameters is as follows: Sequence Length = 15, Embedding Size = 300 and Node Count = 64. One way to further improve the model is to use a larger dataset. Here, the public accessibility of large data sets from the network is a limiting factor. In addition, there is a limit to the size of the data set above which conventional computers can no longer cope with the amount of data. A further increase in accuracy can be achieved if the parameters batch size, learning rate or number of layers are optimized instead of the existing parameters sequence length, embedding size and number of nodes. These could not be optimized in the course of this work due to time constraints. In addition, the TF-IDF (term frequency inverse document frequency) approach can be used to balance the weighting of stopwords, i.e. words that can influence the predictions in a bad way due to their frequency of occurrence. This could further increase the quality of the data set and thus the accuracy of the predictions.

Literatur

[1] Khaled A. Althelaya, El-Sayed M. El-Alfy und Salahadin Mohammed. “Evaluation of bidirectional LSTM for short-and long-term stock market prediction”. In: *2018 9th International Conference on Information and Communication Systems (ICICS)*. 2018, S. 151–156. DOI: 10.1109/IACS.2018.8355458.

[2] Y. Bengio, P. Simard und P. Frasconi. “Learning long-term dependencies with gradient descent is difficult”. In: *IEEE Transactions on Neural Networks* 5.2 (1994), S. 157–166. DOI: 10.1109/72.279181.

[3] Anup Bhande. *What is underfitting and overfitting in machine learning and how to deal with it.* URL: https://medium.com/greyatom/what-is-underfitting-and-overfitting-in-machine-learning-and-how-to-deal-with-it-6803a989c76. (accessed: 24.11.2022).

[4] Bala Priya C. *Softmax Activation Function: Everything You Need to Know*. URL: https://www.pinecone.io/learn/softmax-activation/. (accessed: 28.11.2022).

[5] Google Developers. *Classification: Accuracy*. URL: https://developers.google.com/machine-learning/crash-course/classification/accuracy?hl=en. (accessed: 30.11.2022).

[6] Google Developers. *Descending into ML: Training and Loss*. URL: https://developers.google.com/machine-learning/crash-course/descending-into-ml/training-and-loss?hl=en. (accessed: 30.11.2022).

[7] IBM Cloud Education. *Recurrent Neural Networks*. URL: https://www.ibm.com/cloud/learn/recurrent-neural-networks. (accessed: 20.11.2022).

[8] Raul Gomez. *Understanding Categorical Cross-Entropy Loss, Binary Cross-Entropy Loss, Softmax Loss, Logistic Loss, Focal Loss and all those confusing names*. URL: https://gombru.github.io/2018/05/23/cross_entropy_loss/. (accessed: 13.11.2022).

[9] S. Hochreiter und J. Schmidhuber. “LSTM can Solve Hard Long Time Lag Problems”. In: *Advances in Neural Information Processing Systems*. Hrsg. von M.C. Mozer, M. Jordan und T. Petsche. Bd. 9. MIT Press, 1996. URL: https://proceedings.neurips.cc/paper/1996/file/a4d2f0d23dcc84ce983ff9157f8b7f88-Paper.pdf.

[10] V. Lakshmanan, S. Robinson und M. Munn. *Machine Learning Design Patterns: Solutions to Common Challenges in Data Preparation, Model Building, and MLOps*. O’Reilly Media, 2020. ISBN: 1098115783.

[11] Sanidhya Mangal, Poorva Joshi und Rahul Modak. *LSTM vs. GRU vs. Bidirectional RNN for script generation*. 2019. DOI: 10.48550/ARXIV.1908.04332. URL: https://arxiv.org/abs/1908.04332.

[12] Vlastimil Martinek. *Cross-entropy for classification*. URL: https://towardsdatascience.com/cross-entropy-for-classification-d98e7f974451. (accessed: 02.12.2022).

[13] Christopher Olah. *Neural Networks, Types, and Functional Programming*. URL: http://colah.github.io/posts/2015-09-NN-Types-FP/. (accessed: 05.12.2022).

[14] Christopher Olah. *Understanding LSTM Networks*. URL: https://colah.github.io/posts/2015-08-Understanding-LSTMs/. (accessed: 24.11.2022).

[15] Palash S. *Keras Dense Layer Explained for Beginners*. URL: https://machinelearningknowledge.ai/keras-dense-layer-explained-for-beginners/. (accessed: 06.12.2022).

[16] Sawan Saxena. *Understanding Embedding Layer in Keras*. URL: https://medium.com/analytics-vidhya/understanding-embedding-layer-in-keras-bbe3ff1327ce. (accessed: 01.12.2022).

[17] M. Schuster und K.K. Paliwal. “Bidirectional recurrent neural networks”. In: *IEEE Transactions on Signal Processing* 45.11 (1997), S. 2673–2681. DOI: 10.1109/78.650093.

[18] Yugesh Verma. *A Complete Understanding of Dense Layers in Neural Networks*. URL: https://analyticsindiamag.com/a-complete-understanding-of-dense-layers-in-neural-networks/. (accessed: 01.12.2022).

[19] Shi Yan. *Understanding LSTM and its diagrams*. URL: https://blog.mlreview.com/understanding-lstm-and-its-diagrams-37e2f46f1714. (accessed: 05.12.2022).