Explainability of Neural Networks – A Novel Method for Language Models

Christina Grafeneder

MASTERARBEIT

eingereicht am
Fachhochschul-Masterstudiengang

Interactive Media

in Hagenberg

im Juni 2019

Declaration

I hereby declare and confirm that this thesis is entirely the result of my own original work. Where other sources of information have been used, they have been indicated as such and properly acknowledged. I further declare that this or similar work has not been submitted for credit elsewhere.

Hagenberg, June 25, 2019

Christina Grafeneder

Contents

Acknowledgement

I would like to thank my supervisor FH-Prof. Mag. DI Dr. Andreas Stöckl for guiding me through the whole process. I want to thank him for helping me find an interesting research topic, answering all my questions concerning the project as well as the thesis. Very special gratitude goes out to my study colleague and friend Christa Höglinger for her advises and encouragements during the last years of studying. I am very thankful for her kind support and valuable feedback during the whole study period. I want to say thank you to all my family and friends for their support during my whole school carrier. I would also like to acknowledge my boyfriend Lukas Lehner, who supported me whenever or whatever I needed. Finally, last but not least, gratefully thank goes to my mother for always believing in me. Without her support and encouragement, I would never have accomplished that.

Abstract

Machine Learning gains attractiveness due to their success in several applications. Despite this widespread use of neural networks, the user mostly treats the network as a black box and is not able to interpret the predictions of the model.

In practice, the ability to explain the predictions of a model is very important. It is essential to validate the systems predictions not only by measuring the accuracy but also by checking if the model has a proper problem representation. Interpretability also allows a deeper insight into the network, enables to improve the model, fix misbehaviour and gives more reliance on the system. Therefore some existing explanation methods (e.g., Taylor decomposition, relevance propagation, LIME, etc.) and practical examples, which aim to increase the transparency of the predictions, are described and compared in this thesis. Due to the fact that there do not exist that much explanation approaches for language models, the intent of this thesis is to white-box the predictions of an LSTM character-level language models. Hence, a novel method for analyzing character-level language model is presented and some possible explanations for predictions of such models are given.

Kurzfassung

Maschinelles Lernen gewinnt durch verschiedene Anwendungen an Attraktivität. Trotz der weit verbreiteten Verwendung neuronaler Netzwerke, behandelt der Benutzer das Netzwerk meist als Black-Box und kann die Vorhersagen des Modells nicht interpretieren.

In der Praxis ist es dennoch sehr wichtig, die Vorhersagen eines Modells erklären zu können. Es ist wichtig, die Systemvorhersagen zu überprüfen, indem nicht nur die Genauigkeit gemessen wird, sondern auch, ob das Modell eine korrekte Problemdarstellung aufweist. Die Interpretierbarkeit ermöglicht außerdem einen tieferen Einblick in das Netzwerk, Verbesserungen am Model durchzuführen, Fehlverhalten zu beheben und gibt mehr Vertrauen in das System.

Daher werden in dieser Arbeit einige existierende Erklärungsmethoden (z. B. Taylor decomposition, Relevance Propagation, LIME usw.) und praktische Beispiele, die die Transparenz der Vorhersagen erhöhen sollen, beschrieben und verglichen. Aufgrund der Tatsache, dass es nicht so viele Erklärungsansätze für Sprachmodelle gibt, besteht das Ziel dieser Arbeit darin, die Vorhersagen eines LSTM-Modells auf Buchstabenebene erklärbar zu machen. Aufgrund dessen wird eine neuartige Methode zur Analyse von Sprachmodellen auf Buchstabenebene vorgestellt und mögliche Erklärungen (Pattern) für Vorhersagen solcher Modelle gegeben.

Chapter 1

Introduction

Recently, a significant amount of work has been done to understand and explain machine learning models [30]. For instance, researchers such as Poerner et al. [34], Montavon et al. [30], Bach et al. [5], Landecker et al. [23], Lundberg et al. [37], Broad et al. [48], Li et al. [24] and many more have done reasonable work for improving the overall interpretability of neural networks (NN) and explaining the individual predictions. But there is still room for improvement, as there are few methods in the field of LSTM (long short-term memory) language models. Hence, this thesis will concentrate on explanations of neural networks and focus on developing a novel method to explain the predictions of such language models.

First of all, it is important to clarify the meaning of "understanding", "interpreting", or "explaining" machine learning. With the word "understanding", the functional understanding of the model is meant instead of the lower-level algorithmic understanding of it. This is because the aim is to characterize the models' black box behaviour. A distinction should also be made between *interpretation* and *explanation*. On one hand, an interpretation is the association of an abstract concept (e.g., predicted class) in a domain that a human person can understand. An explanation, on the other hand, is, for example, a heatmap which highlights the pixels of the input image that supports the strongest classification decision [28]. A heatmap emphasizes those pixels of an image which are important for a particular neural network prediction by coloring those pixels [29]. This thesis focuses on the explanation of machine learning models and is categorized as post-hoc interpretability [25]. This means that a trained model is given and the goal is to understand the predictions of a model (e.g., categories) in terms of what is interpretable (e.g., the input variables) [28].

1.1 Problem Statement

Neural networks are the standard for several machine learning problems, such as image classification, natural language processing or sentiment analysis [28]. These networks are widely used and perform impressively well. Nevertheless, the inner working is a mystery and the user mostly treats the network as a black box due to the lack of transparency and the limited interpretability of the predicted outputs [30].

The ability to explain and interpret the predictions of neural networks is yet very

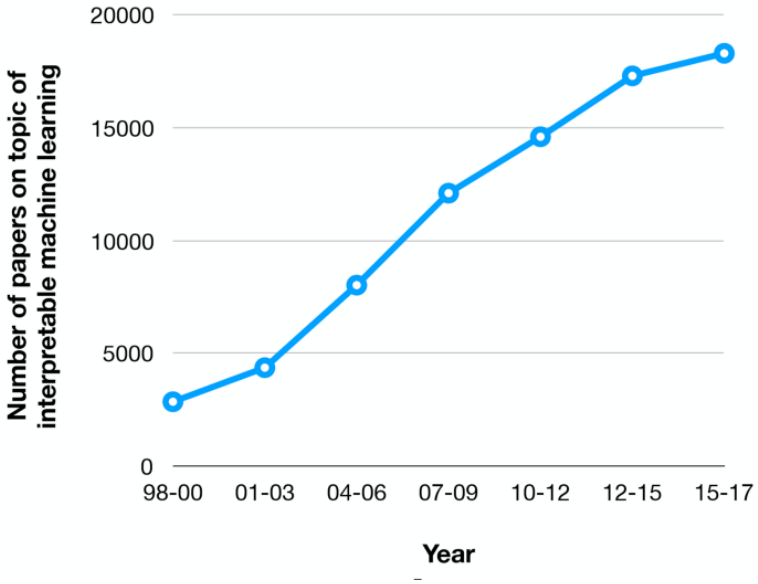


Figure 1.1: The amount of research on interpretable machine learning is growing rapidly. The statistic shows the variety of papers written from 1998 to 2017 [63].

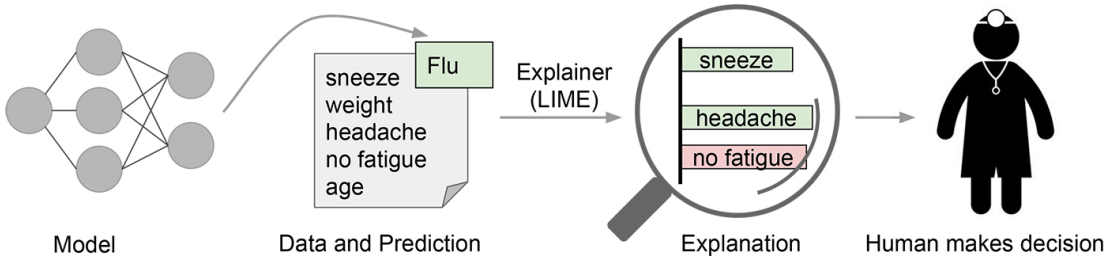


Figure 1.2: Explaining individual predictions to a human decision-maker using LIME (Section 2.5) [35].

important. The rapidly growing amount of research on interpretable machine learning, which is shown in the statistics in Figure 1.1, also proves this.

Hence, it gets more and more common to validate the system's predictions not only by measuring the accuracy but also the proper problem representation. The explainability of NN also allows a deeper insight into the network, enables to improve the model or fix misbehaviour of the model and gives more reliance on the system [28, 37]. Especially in applications such as terrorism detection, medicine [10] or self-driving cars [8] the interpretability is essential [28].

The example in Figure 1.2, shows a way to support the users with their decision to trust or not to trust a prediction of a model. In this example, a model predicts that a certain patient suffers from the flu. The symptoms that are most important to the model's decision are highlighted by the "explainer". Due to this additional information about the reason for the prediction, it is easier for the doctor to trust the model since

it is easier to trust a prediction if the reason is understandable [35].

Moreover, only a clear evaluation of the systems behaviour can help to improve and fix a failure in artificial intelligence (AI). For example, when a self-driving car has high accuracy but does not stop when a child is crossing the street, the model does not have a proper representation of the problem. Hence, high accuracy does not always mean that the system represents the world in a proper way. It is very important to understand which circumstances lead to the systems decision.

In addition, the European Union created new regulations on algorithmic decision-making (enacted 2016, taking effect 2018). This contains a "right to explanation", whereby users have the right to obtain an explanation of an algorithmic decision that was made about them [4]. This means that it is necessary to be able to explain the behaviour of algorithms that make decisions based on user-level predictors.

Those are just a few examples, but they demonstrate that explainability is not only an interesting topic, but it will play an important role in the future of machine learning.

1.2 Research Objective

Generally, it is not completely perceivable how the LSTM is learning different rules in a language, as for instance when a bracket is opened it is very likely that there is a closing bracket before the sentence ends. Probably this is done by keeping track of long-range dependencies such as line lengths, URLs, brackets or quotes. For example, within an URL (which start is e.g., indicated by "www") the probability of a dot is higher than an empty space. One possible way to find out such dependencies is to visualize how the model changes after different input sequences. For instance, when a language model learns that the probability of a closing bracket increases after an open bracket is in the input sequence. A simple outlook into how such an explanation can look like is shown in Figure 1.3. There is no need to understand this visualization in detail but it demonstrates how it is possible to explain the behaviour of the language model. There it can be seen that the one line shows the probability of the open bracket and the other shows the probability of the closing bracket. The change of the probability over time demonstrates that the open bracket has an influence on the closing bracket as the probability increases (on the position "timeout("). This method is described in Chapter 7 in detail.

There already exist different approaches to find out how such decisions are made in a model. Most of them try to find connections between the input and the output to get knowledge about the models black box. Some of those existing neural network explanation methods are described in this thesis. But as there exist few methods in the field of character-level language models the main goal of the research is to find out if it is possible to explain LSTM character-level models. On this account, it will be evaluated what a visualization of an LSTM language model can tell about the model's self-learned algorithm. Therefore the developed visualization is used to find patterns that explain the system's predictions and how they make their decisions. On this account, three models trained with different input data are evaluated and compared.

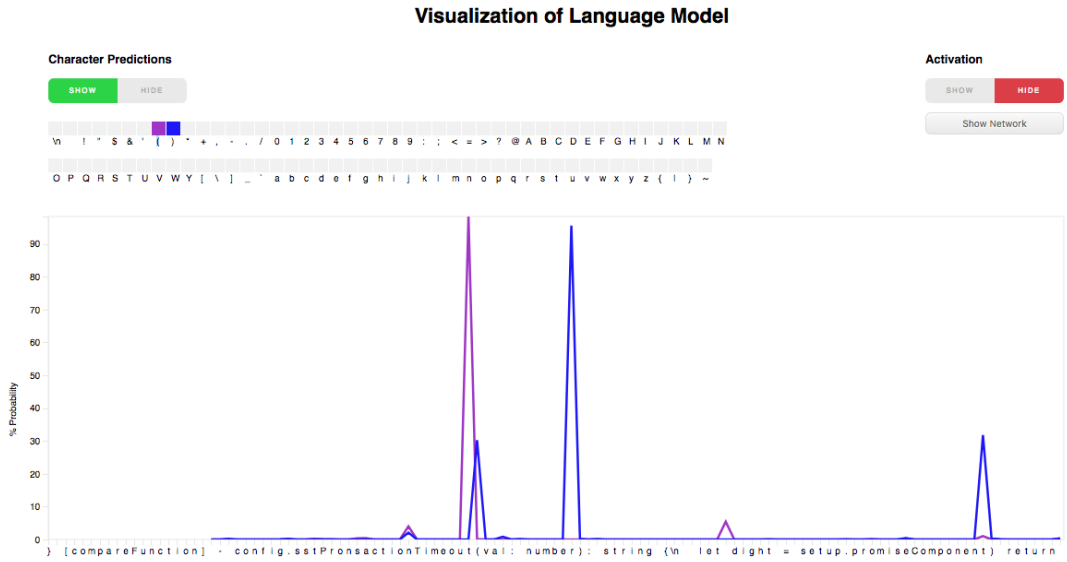


Figure 1.3: This visualization shows a pattern for opening and closing bracket. The input of a open bracket changes the probability for a closing brackets.

1.3 Thesis Structure

The thesis will start with a substantiated description of the main problem and goal of the thesis. Furthermore, in Chapter 2 some elected explanation methods are described and different applications and research about those methods are accentuated. In Chapter 3 some practical examples and work most related to the thesis is presented. Chapter 4 will introduce the main concepts and terms used for the developed explanation method. The methodology for the explanation visualization and the technology stack is part of Chapter 5. Moreover, Chapter 6 describes the model's implementation in detail, while Chapter 7 attends to the application implementation. Chapter 8 contains the most important part of the thesis. It presents the results of the research in the form of different explanation patterns found to explain the language model. Finally, the thesis ends with an honest conclusion of the developed explanation method as well as the ideas for future work.

Chapter 2

Explanation Methods

With the rising usage of artificial intelligence in human everyday life, such as for self-driving cars or in medicine, the interpretability of a model's prediction gains more importance, as demonstrated in the introduction. Hence, this chapter gives a short overview of the different explanation methods still available in literature. As it would go beyond the scope to describe all existing methods in detail, the election focus is set on post-hoc interpretability and explaining the predictions of a network. Those exploration methods have been applied to a wide range of problems (e.g., images, text, etc.). Therefore, some practical examples are described for each of the selected methods. Those should answer the question: "What can be done with does methods in practice?".

2.1 Overview

As interpretability in machine learning models is gaining more and more importance, there exists a huge variation of explanation methods in literature. Figure 2.1 shows an overview of those methods and classifies them into their underlying mechanics and mathematical fundamentals.

Several of the existing methods are gradient-based [28, 34], such as *sentitivity analysis* [6, 42], *Deep Learning Important FeaTures* (DeepLIFT) [41] or *integrated gradient* [45]. For example DeepLift, introduced by Shrikumar et al. [41], is a novel algorithm to assign importance score to the inputs for a given output. Furthermore there are methods based on decomposition, such as LRP (*layer-wise relevance propagation*) and *deep Taylor decomposition*. Other methods are trying to improve those decomposition methods as for example *Local Interpretable Model-Agnostic Explanation* (LIME) [66] does. In addition, further methods are based on deconvolution. There is, for example, a method called *deconvolution* by Zeiler et al. [47] which visualizes how each layer in a convolutional neural network processes an input image by essentially reversing the hierarchical image encoding process [47]. Another method based on the deconvolution is the *guided-backpropagation* presented by Springberg et al. [43].

In the further sections some elected explanation methods, their intention and main underlying mechanics are described. As already mentioned at the beginning, this thesis will focus on the explanation and not on the understanding of neural networks. Thus, the explanation methods in the overview are those which aim to explain the prediction of

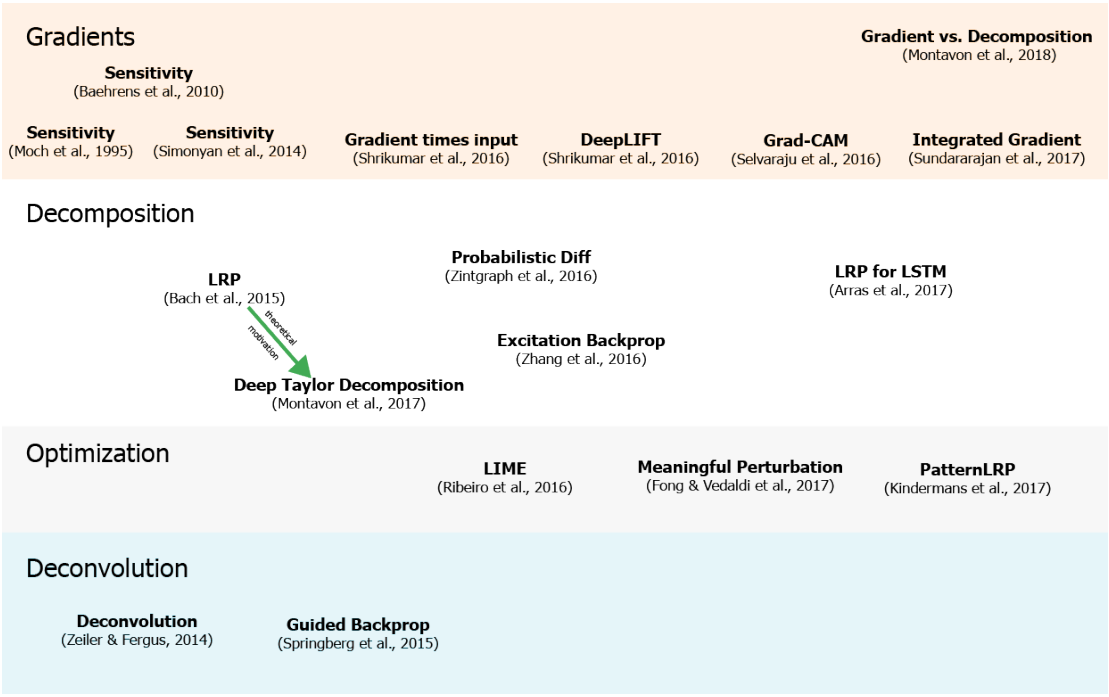


Figure 2.1: This picture shows the explanation methods available. It gives a general overview of how they are all related to each other and how they differ [68, p. 33].

a model, but there exist much more which aim to understand machine learning models.

2.2 Sensitivity Analysis

Sensitivity analysis (short SA) [6, 42] is, as mentioned before, a gradient-based method that tries to identify the most important input features for a prediction. This means that it is based on the locally evaluated gradient or some other local measure of variation of the model. Since the gradient can be computed using backpropagation, the technique is easy to implement for a deep neural network (DNN) [28].

In this method, the local effect instead of the relevance score for the input is measured. Hence, the relevance of each neuron for the prediction is measured, based on the effect of mutating it. So, the most important input feature is the one where the output is most sensitive. In addition, it is important to know that sensitivity analysis does not produce an explanation of the function value $f(x)$ itself, but rather a variation of it [28]. Hence, when classifying a cat in an image the methods do not explain "what makes the image a cat?" but "what makes the image more or less a cat?". Therefore, it highlights parts which, when changed increase or decrease the evidence of a prediction [68].

2.2.1 Practical Examples

The scope of applications is in scientific applications of machine learning such as medical diagnosis [21], ecological modeling [14], or mutagenicity prediction [6]. Moreover, Khan

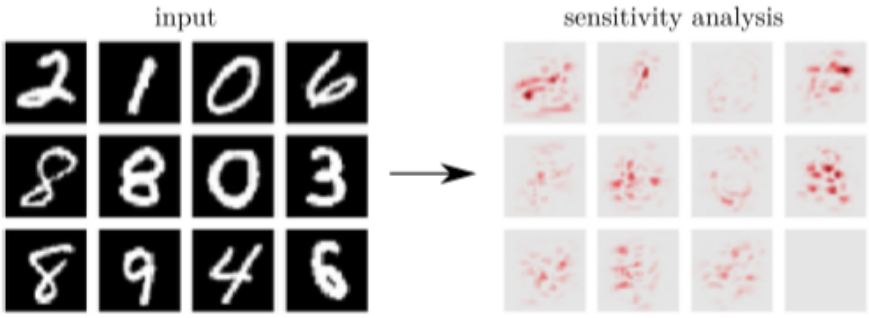


Figure 2.2: Montavon et al. [28] applied SA to a convolutional DNN trained on MNIST (database with handwritten digits). This figure shows the resulting explanation for selected digits presented as a heatmap. A positive relevance score is indicated with red pixels [28].

et al. [21] for instance developed a method of classifying cancer to specific diagnostic categories and used SA for explaining the predictions. The aim is to identify the genes most relevant to the classification of a specific cancer [21]. In addition, Engelbrecht [12] used SA for "selective learning" by feedforward neural networks. Engelbrecht used SA to find the most informative patterns, which are those patterns closest to decision boundaries because those are then selected for training. More recently, the technique was also used for explaining image classifications using deep neural networks, as for example, Simonyan et al. did in their paper [42]. Montavon et al. [28] also used SA to explain image classification of convolutional DNN trained on MNIST, which is a dataset of handwritten digits[1]. Figure 2.2 shows the explanation heatmaps for some selected digits as a result of SA.

Futhermore, Montavon et al. [30] compared heatmaps created with two different explanation methods: Deep Taylor decomposition (described in Section 2.3.2) and SA [30]. The Figure 2.3 shows this comparison on four different images. It appears that the SA heatmap is a little bit noisy in comparison to the deep Taylor heatmaps and do not show the relevance as clear as the other method does. This is because SA only measures a local effect and does not conceptually redistribute relevance onto the input. However, the relative strength of computed sensitivities between examples or pixels can still be measured.

2.3 Decomposition

The general concept of *decomposition*, also called decomposing relevance, means that the function value $f(x)$, which represents the amount of evidence for a given class, is decomposed as a sum of relevance scores. These scores indicate the relevance of every input feature for the output.

First of all, the output neuron x_f is decomposed on its input neurons. Afterwards, the decomposition on these neurons is redistributed on their own inputs. This redistribution process is repeated until the input features are reached. On this account, $[[x_f]_j]_i$ is

[1] Find the MNIST Database of Handwritten Digits here at http://yann.lecun.com/exdb/mnist/.

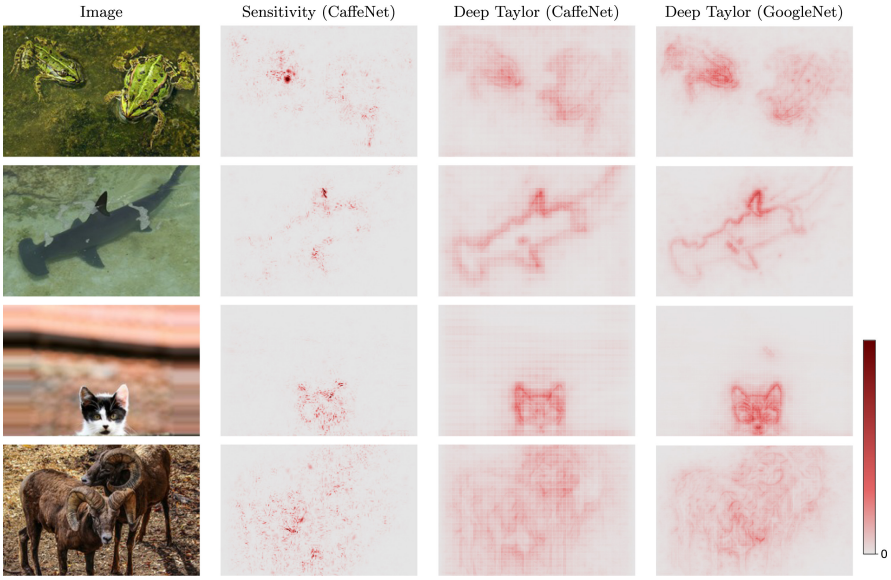


Figure 2.3: The images with different classes ("frog", "shark", "cat" and "sheep") were given as input to a deep network. The images next to it represent the heatmaps done with SA and deep Taylor. Those heatmaps illustrate the relevant pixels for the classification [30].

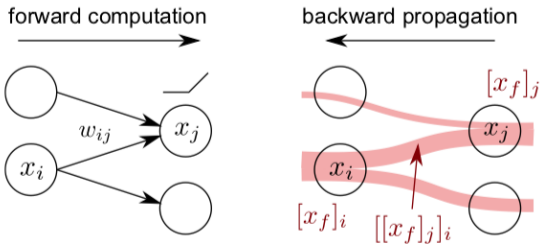


Figure 2.4: A portion of a neural network showing a redistribution from an output variable back to the input variable [29].

defined to determine how much of the neuron x_f is redistributed from an arbitrary neuron x_j to one of its input x_i. Those terms which are redistributed from the neurons $(x_j)_j$ are summed [29],

$$[x_f]_i = \sum_j [[x_f]_j]_i.$$

Figure 2.4 shows a redistribution as well as the summing in propagation phase of a simple neural network [29]. An important property, called relevance conservation, basically guarantees that the sum of all the relevance score corresponds to the prediction [29].

2.3.1 Simple Taylor Decomposition

The simplest and most direct method to perform such a decomposition is the *simple Taylor decomposition*. It is a good technique for measuring the importance of input

variables for a prediction [5]. It utilizes the network structure by backpropagating the explanations from the output to the input layer [28].

The relevance is the product of the sensitivity (given by the locally evaluated partial derivative) and saliency (given by the input value). Which means that the feature is relevant if an input feature is present in the data and the model reacts to it [28].

The main purpose of the method is to explain machine learning classifiers in the field of image analysis [28]. Unfortunately, this *simple Taylor decomposition* does not work really well for DNNs. There are two main reasons for this: The first problem is that the root point at zero is not a good starting point because it is far away from the input and it is difficult to find the right root point. The other problem is that DNNs suffer from a problem called gradient shattering. This means that the function value is getting more and more noisy with depth [29]. Due to these problems, Montavon et al. [29] proposed a novel Taylor approach for deep neural networks, called *deep Taylor decomposition*. This method can overcome the multiple technical limitations of the *simple Taylor decomposition* because instead of decomposing the whole function at once, it decomposes the simpler functions individually. Thus it can be used for a much larger class of functions [30].

2.3.2 Deep Taylor Decomposition

Montavon et al. [30] extends the *simple Taylor decomposition* in a way that takes advantage of the deep structure of neural networks and connects it to rule-based propagation methods. The name *deep Taylor decomposition* (DTD) results from the iterative application of the *simple Taylor decomposition* from the top layer down to the input layer [28].

The decomposition process is shown in Figure 2.5. It is awarded by backpropagating the models output (prediction) in the neural network graph back until the input variables are reached. There, the activation of the top neuron is redistributed into relevance scores of neurons in the layers before. These relevance scores are then expressed as a function of the activations of the layer before, which enables another step of redistribution. This process results in a relevance score for each input variable. DTD can be quickly computed, as is done for the forward pass.

The decomposition underlies a redistribution rule. The main characteristic is that the propagation rules are derived from a *simple Taylor decomposition* executed at each neuron of the network [20].

2.3.3 Practical Examples

As explained in the section of sensitivity analysis, Montavon et al. [28] also used *simple Taylor decomposition* to explain image classification of convolutional DNN trained on MNIST. Figure 2.6 shows the explanation heatmaps for some selected digits. In comparison to SA, this method is also able to show the negative relevance scores. Furthermore, Montavon et al. [30] have done some examples with *deep Taylor decomposition* in the field of image classification. In their paper they compared explanations done with *deep Taylor decomposition* and *sensitivity analysis*, as already described in Figure 2.3. The method is also applicable to other types of input data, learning tasks and network architectures. But generally, the focus of *deep Taylor decomposition* is set on image

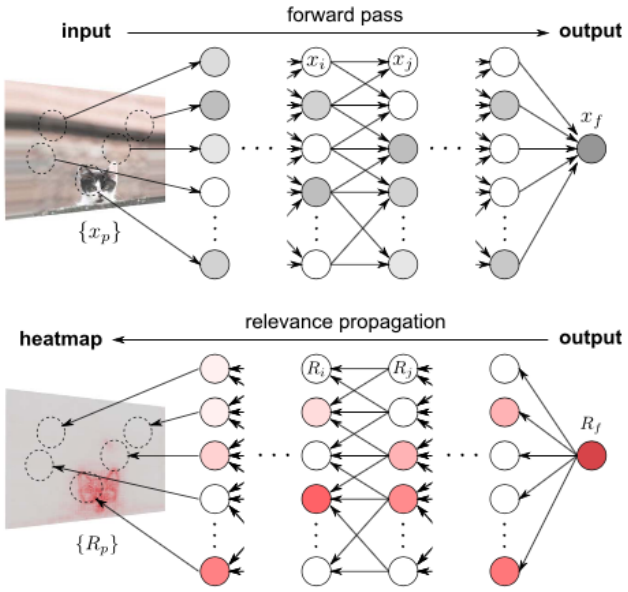


Figure 2.5: This example shows the process of *deep Taylor decomposition* with a DNN on an image of a cat. The left side shows the classification (forward pass) of the image as a cat. The right side shows the process of explanation (relevance propagation) of the image by showing the results on a heatmap [30].

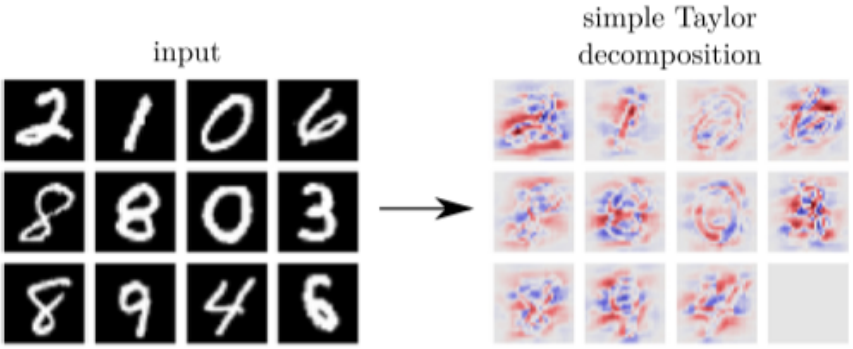


Figure 2.6: Montavon et al. [28] applied *simple Taylor decomposition* to a convolutional DNN trained on MNIST. This figure shows the resulting explanation for selected digits presented as a heatmap. A positive relevance score is indicated with red pixels whereas negative relevance score is indicated with blue pixels [28].

classification and creating heatmaps as an explanation.

2.4 Relevance Propagation

Instead of decomposing the prediction of a DNN, a feasible way is to use the feed-forward graph structure of such a network. The relevance propagation algorithm moves

backwards in the graph (backpropagate) and redistributes the prediction score (or total relevance), as seen in Figure 2.7. This figure also shows that the propagation starts at the output of the network (generated during the forward pass) and ends when the input is reached [28].

So, assume a DNN with two neurons, j and k, at two successive layers. The relevance score associated with neurons in the higher layer is defined by $(R_k)_k$. $R_{j\leftarrow k}$ defines the share of relevance that flows from neuron k to neuron j. The share is decided by the contribution of neuron j to R_k, on the condition that the local relevance conservation constraint

$$\sum\nolimits_j R_{j\leftarrow k} = R_k$$

is satisfied [28]. The relevance of a neuron in the lower layer is then determined by the total relevance it receives from the higher layer,

$$R_j = \sum\nolimits_k R_{j\leftarrow k}.$$

When combining the two equations, they ensure a relevance conservation property between all consecutive layers. The local relevance conservation constraint is a principle that has to be satisfied for the redistribution process. This also leads to a global conservation property from the neural network output to the input relevance scores,

$$\sum\nolimits_{i=1}^{d} R_i = ... = \sum\nolimits_j R_j = \sum\nolimits_k R_k = ... = f(x).$$

Landecker et al. [23, pp. 32–38] introduced this propagation approach to explain the predictions of hierarchical networks. Bach et al. [5] established the approach in the context of convolutional DNNs, described in the Section 2.4.1, for explaining the predictions of these models [28].

There also exist other explanation methods that utilize the DNN graph structure, even though they are not producing a decomposition of the network output $f(x)$. *Deconvolution* introduced by Zeiler and Fergus [47] and *guided backpropagation* introduced by Springenberg et al. [43] are just two examples. Their work also comprises a technique where a backward mapping through the graph is applied and interpretable patterns in the input domain are generated. These patterns are associated with certain predictions or feature map activations [30].

2.4.1 Layer-wise Relevance Propagation (LRP)

In 2015 Bach et al. introduced a novel concept denoted as layer-wise relevance propagation (LRP). It is based on the propagation approach described in Section 2.4. LRP is a special type of the *deep Taylor decomposition* (Section 2.3.2) and therefore the mathematical interpretation is based on it [38].

The main idea of the LRP method is to redistribute the evidence of the output $f(x)$ back to the input. So the output of the network is decomposed into the sum of the relevance of the input features. Starting at the output neuron the method redistributes the evidence back to the next neuron in the layer before, as seen in Figure 2.7. Therefore it applies a propagation rule, at each neuron within the network [38]. At each layer, the sum of the relevance is preserved, and the final propagation maps the relevance back

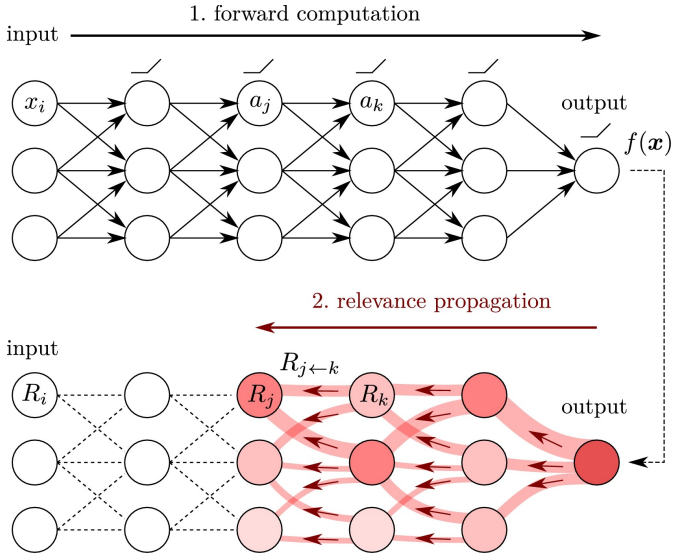


Figure 2.7: In the diagram, the LRP procedure can be seen. Three steps of redistribution are already done. The red arrows indicate the relevance propagation flow [28].

onto the input vector. The result shows the relevance of the single neurons for the classification decision.

There are different *propagation rules*: The first one, called *simple LRP rule*, assumes two factors that affect the redistribution. The first important parameter is the activation of the neuron. This means that if the neuron is more active it should get a larger share of the relevance redistributed to it because it encodes something important for the particular output. The second parameter is the weight, that shows the strength of the connection between the neurons. So, if there is a stronger connection between the neuron redistributing to then this neuron should get a higher relevance [38]. The second propagation rule is called *alpha-beta rule*. In this alternative rule, the main difference is that positive and negative values are redistributed separately. This means that every input that enlarges the evidence for the predicted class is redistributed separate from those inputs that minimize the evidence for the predicted class. Therefore, it can be analyzed what input speaks for or against a classification. Bach et al. show that this redistribution rule corresponds with a *deep Taylor decomposition* [38].

Moreover, the decomposition follows a conservation principle, which claims that the sum of relevance scores must match the model output [7]. The goal of LRP is to determine those features in a particular input vector, which are most related to a models output [5]. For image recognition tasks, the procedure results in a heatmap, as shown in the practical examples below. They are frequently used to understand how a neural network interprets an image.

2.4.2 Practical Examples

LRP is developed for fully connected networks and convolutional neural networks (CNNs) [5]. Later it also was extended to LSTMs, called LRP for LSTM [22]. Hence, LRP is a

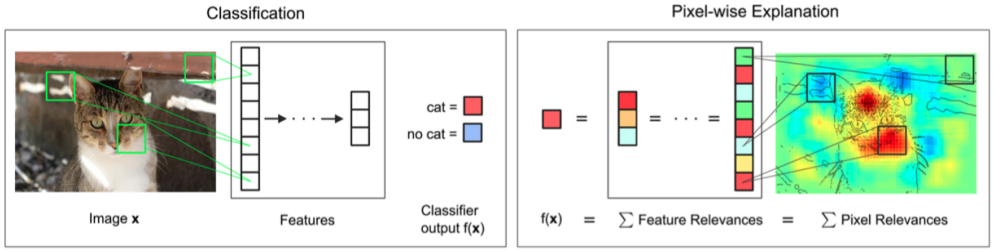


Figure 2.8: This example shows a heatmap of an image classification of a cat. The final relevances show the contributions of each pixel to the prediction [5]. A red pixel indicates positive evidence for the classification of a cat, whereas a blue pixels indicates "no cat" classification [5].

very generic concept applied to different models, not only to neural networks but also to SVM (Support Vector Machine [36, p. 744]), CNN or LSTM. Moreover, LRP has been used in a number of applications such as model validation or analysis of scientific data [28]. It can help to get more *reliance* on a model. Therefore, Yang et al. [46] have applied the LRP algorithm to explain medical-related decisions proposed by DNNs. The evaluation shows that the features, identified as relevant by the algorithm, mainly correspond with clinical knowledge and guidelines [46]. Furthermore, LRP can be used to explain image classification. Bach et al. [5] have proposed a solution for visualizing the contributions of single pixels to predictions. Therefore Figure 2.8 shows a visualization of the pixel-wise decomposition process and which of the pixels in the image are most related to the output prediction and vice versa.

Arras et al. [22] has introduced a strategy for extending the LRP method to recurrent architectures, such as LSTMs. Therefore an LSTM model for the sentiment prediction of sentences is used. Arras et al. [3] also worked on understanding the text classification process, by tracing the classification decision back to individual words using LRP. Furthermore, Arras et al. [2] use LRP to explain the predictions of a CNN trained on a topic categorization task. The work analyses the relevant words for a specific prediction, as seen in Figure 2.9. Two heatmaps, using either LRP or SA for highlighting words, are shown. A positive relevance of a word is colored red, a negative blue. The target class for the explanation is indicated on the left side [2].

Moreover, Samek [67] presented some applications based on the LRP technique introduced by Bach et al. [5]. The interactive application (see Figure 2.10) presents a heatmap for natural images. The heatmap highlights the important areas for the classification of the image by coloring it red. Samek also created such interactive applications[2] for text classification.

2.5 Local Interpretable Model-Agnostic Explanation (LIME)

Local interpretable model-agnostic explanation (LIME) is a model-agnostic method, which means that it can be applied to any machine learning model (such as NN, SVM

[2]Demos found at https://lrpserver.hhi.fraunhofer.de.

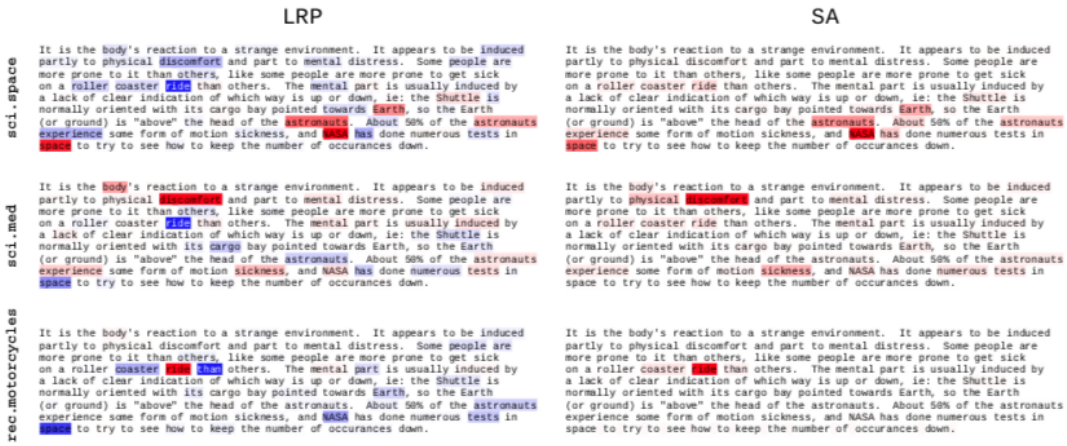


Figure 2.9: Arras et al. compared the usage of LRP and SA in the field of text classification. This example shows a heatmap where the most relevant words are highlighted red and the words which have negative influence on the prediction blue [2].

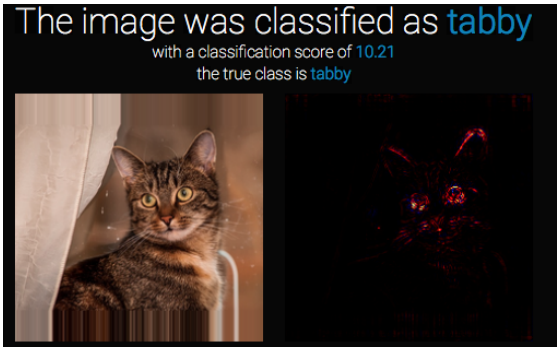


Figure 2.10: A heatmap as an explanation for a image classification. It shows that the ears and the eyes are the most relevant parts for the classification of a cat [67].

and random forest[3]) and explain their predictions [54]. Therefore, as seen in Figure 2.11, the technique treats the model as a black box. The model is really simple but can be applied even if the model is using more complicated components.

LIME learns the behaviour of the underlying model by perturbing the input to detect how the model's prediction changes [66]. So, instead of trying to understand the whole model at the same time, a single input instance is modified and the changes in the predictions are monitored. In the context of text classification, this means that some words are e.g., replaced or removed, to monitor which input feature has an impact on the predictions [66]. So, this is done by weighting the perturbed input (e.g., word or image) by their similarity to the instance that should be explained. A possible explanation looks like the example in Figure 1.2. The symptoms most related to the predicted disease are highlighted. The highlighted symptoms are just an explanation for this special person

[3]Read more about "random forest" in [19, p. 70].

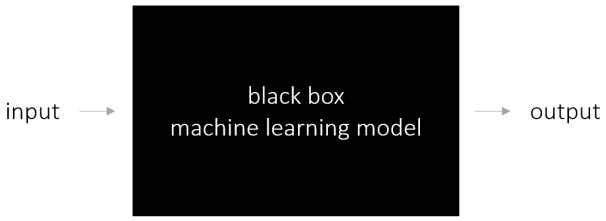


Figure 2.11: LIME assumes that the machine learning model is a black box. Therefore it is able to explore the relationship between input and output for any model [54].

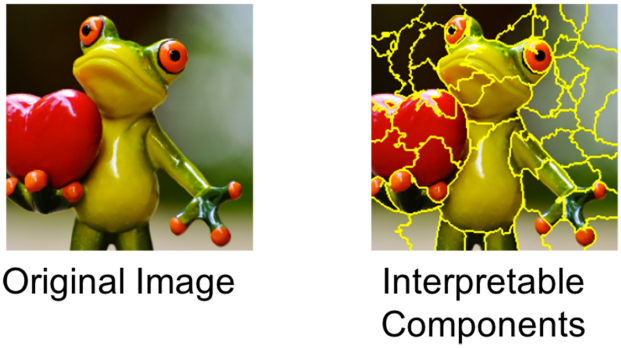


Figure 2.12: The image shows the transformation of an image into interpretable components [66].

but they do not represent the behaviour of the model for all specific patients [66].

Thus with LIME the question "Why should I trust you?" can be answered. In addition, there exists a method inspired by LIME [35], called LIME for NLP (short LIMSSE). It is a framework for explaining predictions of complex classifiers (e.g., RNNs (Recurrent Neural Networks), CNNs) and is designed for word-order sensitive task methods [34].

2.5.1 Practical Examples

Ribeiro et al. [66] used LIME to explain numerous classifiers (such as random forests, SVMs, and NNs) in the text and image domains. Figure 2.12 for example shows how LIME is used for image classification. LIME explains a classifier that predicts how likely it is for the image to contain a tree frog. For this explanation, LIME takes the image on the left (input) and divides it into interpretable components (contiguous superpixels) as seen on the right side. As illustrated in Figure 2.13, a dataset of perturbed instances is generated by turning "off" some of the interpretable components (e.g., making them gray). For each of these perturbed instances, it gives a probability that a tree frog is in the image. Then a simple model is trained on this dataset. In the end, those superpixels with the highest positive weights are presented as an explanation, everything else is grayed out. The explanation reveals that the classifier primarily focuses on the frog's face to make the prediction [66].

Another example in Figure 2.14 shows a explanation of a text classifier created by

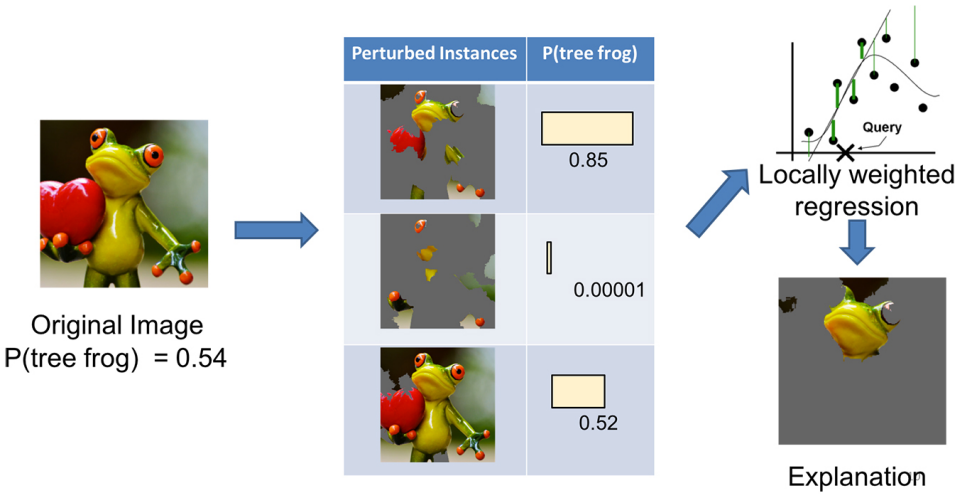


Figure 2.13: The figure shows an explanation of an image classification with LIME [66].

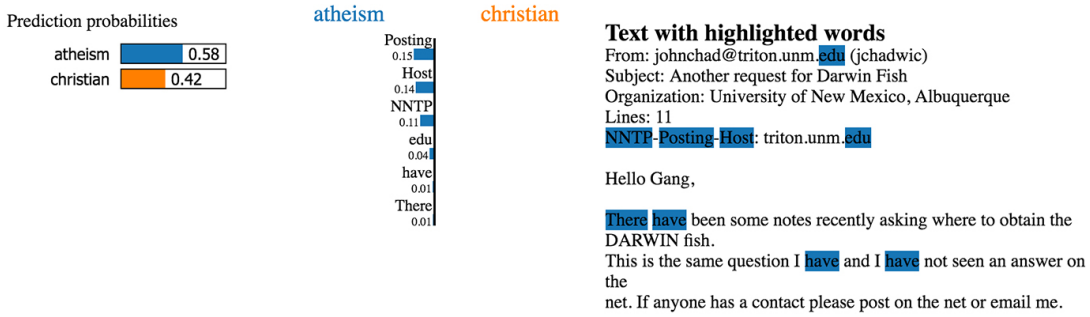


Figure 2.14: Explanation for a prediction in the "20 newsgroups" dataset [66].

Ribeiro et al. [66]. The "20 newsgroups" dataset[4] is a benchmark in the field of text classification and has been used to compare different models in several papers. Ribeiro et al. took two classes which are hard to distinguish because they share many words: Christianity and atheism. They trained a random forest with 500 trees, and got a high test set accuracy of 92.4%. This seems to be trustworthy if accuracy is the only measure. However, on closer inspection the classifier predicts the instance correctly, but for the wrong reasons. The additional exploration shows that the word "posting" (part of the email header) appears in 21.6% of the examples in the training set but only two times in "Christianity". The same can be discovered in the test data, where the word appears in almost 20% of the examples but only twice in "Christianity". Such patterns would not be found in the real world but they make the problem much easier. When understanding what the model is actually doing, the insights become easy [66].

[4] http://qwone.com/~jason/20Newsgroups/

Chapter 3

Related Work

There has been a significant amount of work focusing on the analysis and understanding of neural network predictions. This section provides an overview of the related work that was found during research. Several applications are based on the explanation methods described before but others have different underlying principles. Therefore, this section is about the projects with other underlying mechanics, which are working with language model visualizations. Those are most related to the developed project and the topic of the thesis.

Andrew Karpathy [1] is one of the key players in the field of LSTM character-level explanation. Karpathy et al. [1] used static visualization techniques for a better understanding of language models. Their work demonstrates that some neurons are interpretable in a way that they recognize open parentheses, the start of URLs, brackets and other patterns, as shown in Figure 3.1. It is like a heatmap for text that indicates in which parts of the text the neurons are active or inactive. Based on the analyses

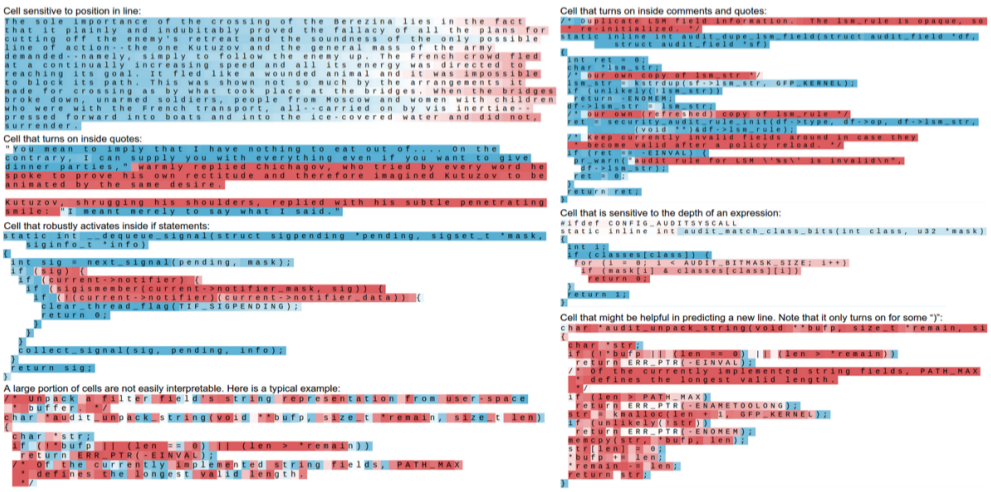


Figure 3.1: An example of cells with interpretable activations discovered in the "War and Peace" LSTM. The color in which the text is highlighted corresponds to the activation of the neuron where inactive ("-1") is red and active ("1') is blue [1].

Figure 3.2: This screenshot shows an LSTM visualization interface [44].

of Karpathy et al. [1], E. Chen [50] created an interactive web-based visualization for exploring the inner working of an LSTM. Li et al. [24] present additional techniques, particularly some visualization for understanding compositionality and other semantic properties of deep networks. Rauber et al. [32] use projections to explore relationships between neurons and learned observations. In addition, Strobelt et al. [44] created a visual analysis tool for RNNs with a focus on understanding the hidden state dynamics, as seen in Figure 3.2. The visualization ensures a user to select a range of text for specifying a hypothesis about the model in the selected view. The selection can be used to find some similar hidden state patterns in the dataset [44].

3.1 Contribution

As Chapter 2 has shown, there already exist different methods and practical applications to make the models behaviour explainable. While there exist a bunch of research about image classification, text classification and sentiment analysis, there is only little research in the field of character-level language models. Especially when talking about visualization types. There only exist a few different approaches such as heatmaps or the work by Karpathy et al. [1] and some emulators of his work.

Hence, the goal this thesis is to create a visualization for getting a better understanding of the learning process of a character-level language model. The developed explanation method is most similar to the work by Karpathy et al. [1] and Strobelt et al. [44] in that we are concerned with interpreting the cell activations and predictions of neural network models. However, the specific goal is to create an interactive visualization, with the focus on LSTM character-level language models, to make those models more perspicuous and interpretable. The detailed methodology and process of this explanation application are described in Chapter 5.

Chapter 4

State of the Art

The subject of this thesis is automated learning, also called machine learning (ML). Learning can be understood as gaining knowledge by studying or being taught. So, ML can be seen as using algorithms for acquiring structural descriptions from data examples. The computer receives some raw data and tries to learn from the structure that represents the information. These structural descriptions are used by the model to predict unknown data [33].

Machine learning is a sub-area of artificial intelligence. There does not exist an explicit definition but many practitioners have tried to describe it. Arthur L. Samuel [39], one of the pioneers in machine learning, provides the following general definition:

> [Machine Learning is the] field of study that gives computers the ability to learn without being explicitly programmed.

Moreover, Tom M. Mitchell [27, p. 2] defined machine learning in an engineering-oriented way:

> A computer program is said to learn from experience E with respect to some class of tasks T and performance measure P, if its performance at tasks in T, as measured by P, improves with experience E.

4.1 Supervised Learning

In the field of Artificial Intelligence there exist three main types of learning: Unsupervised learning, Supervised learning and Reinforcement learning.

Unsupervised learning is a learning type where the algorithm learns patterns in the input without explicit feedback. Clustering is the most common unsupervised learning task. For instance, unsupervised learning is when a taxi agent develops step-wise to differ between "good traffic days" and "bad traffic days" without getting labeled examples by a teacher. When the agent learns from a series of reinforcements rewards or punishments then it is called *Reinforcement learning*. For instance, the algorithm learns that he did something well by getting a reward for the win [36, pp. 694–695]. Finally, the last type of learning is called *Supervised learning*. A supervised scenario is characterized by the concept of a supervisor [40]. For this, the input data must contain the desired output, which is often provided by a human person (the supervisor). Generally, nearly

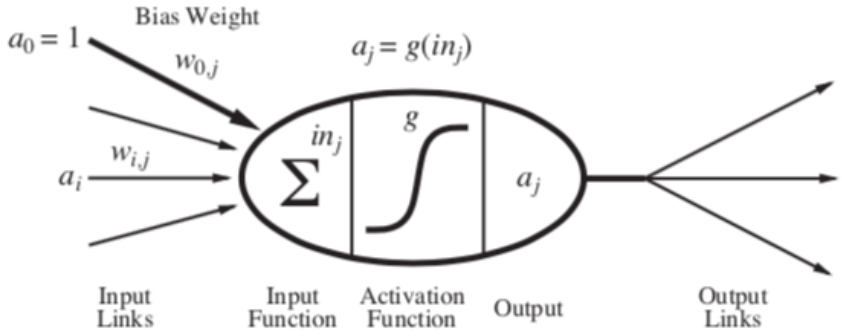


Figure 4.1: This image is a representation of a neural network neuron. It is a simple mathematical model [36, p. 728].

all common applications in the field of Machine Learning are of this learning type, as for instance text recognition, image classification or natural language processing (NLP) [11, pp. 129–130]. For example, in image recognition tasks, the inputs are camera images and the outputs are defined by a user with "cat" or "no cat" [36, p. 695].

4.2 Artificial Neural Network

Humans were inspired by birds to fly, burdock plants inspired velcro, and nature has inspired many other inventions. Therefore, the architecture of the brain has inspired humans to build an intelligent machine. This was the key idea that inspired artificial neural networks (ANNs). But, although planes were inspired by birds, they do not have to flap their wings. Similarly, ANNs differ from the neural network in the humans brain [13, p. 253].

A brain, in a simplified manner, consists of a multitude of neurons (computing devices) that are linked together in a complex communication network through which the brain can perform highly complex calculations. ANNs are built after this network structure. The main idea is to carry out complex computations by joining many neurons together using communication links [40].

A neural network is a directed graph in which the nodes correspond to neurons and edges correspond to links between them. These links enable the communication between the nodes by sending the output of some neuron to the input of another neuron [40, p. 268]. The link from node `i` to node `j` is responsible to propagate the activation a_i from the one node to the next. To represent the strength of the connection between some nodes, each link also has an associated numeric weight $w_{i,j}$.

The most general network type is a so-called *feed-forward network*. The connections of such a network are only in one direction. This means that every node receives input from nodes before them and propagates their output to their next nodes. Such feed-forward networks are arranged in layers. So, each node only receives input from nodes in the layer before them [36].

Figure 4.1 shows a mathematical view of such a neuron [26, 36]. In the first step,

each neuron `j` computes a weighted sum of its inputs [36],

$$in_j = \sum_{i=0}^{n} w_{i,j} a_i.$$

For deriving the output the neuron applies an activation function `g` to the sum of the inputs [36],

$$a_j = g(in_j) = g\left(\sum_{i=0}^{n} w_{i,j} a_i\right).$$

A neural network with more than two layers of representation is called *deep neural network*. Deep learning is defined as it deals with such deep neural networks. This is a problematic definition as it sounds as if deep learning existed since the 1980s. But neural networks had to change architecturally from the earlier network styles to perform such good results. The evolution of NNs changed in different important facets: The networks have more neurons than previous networks, they have more complex ways of connecting layers and neurons, the amount of computing power available to train the networks increased enormously and the automatic feature extraction was invented [33].

4.2.1 Recurrent Neural Network (RNN)

Most of the different types of NNs, as for instance feed-forward networks, do not have a "brain". The inputs are processed independent of each other and the caused state is not saved. Thus, to process a sequence of data with a NN the whole sequence has to be transferred at the same time. A human, on the other hand, is processing a sentence each word after the other and remembers the previous words. In this way, a human gets the context of the sentence when reading. Biological intelligence gradually handles information and produces an internal model of the received information. The model is continuously supplemented by new information [11, p. 252].

A recurrent neural network (short RNN) works with the same principle, although with a much simpler version of it [11, p. 252]. The RNN process sequences by running through the elements and saving the information relative to the previous elements in a state or also called memory. So, in contrast to other models, the RNN handles the data not as a whole but runs the sequence elements internally in a loop [11]. This means that it does not only processes the information into one direction but feeds the outputs of a neuron back as the input of the same neuron [36, p. 729].

Figure 4.2 shows the RNN loop unrolled, as seen on the right side of the picture. This means that the network is drawn for the complete input sequence. The internal state, that enables information to persist, is computed by the value of the hidden vector, which is the sum of the product of the weight matrix w and the hidden state at time t-1 and the product of the weight matrix and the input at time t, passed through the activation function *tanh* [17]. This makes the network form a dynamical system that may reach a stable state [36, p. 729].

Hence, recurrent networks, unlike feed-forward networks, are theoretically able to support short-term memory [36, p. 729]. In the past, people thought that these networks were difficult to train but improvements in research have made them more practical [33, p. 143].

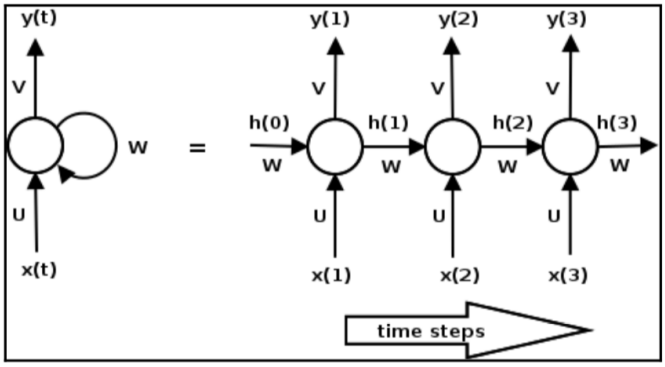


Figure 4.2: The network shown in this figure is a three-layer RNN, suitable for processing three element sequences [17, p. 177].

4.2.2 Long Short-Term Memory (LSTM)

Theoretically, RNNs should be able to access the different inputs from the future to every time. In practice, it is impossible to learn such long-term dependencies. This is due to the *vanishing gradient problem*, an effect that remembers problems occurring in not-recurrent NNs (feed-forward networks) that contain several layers. When a NN is too deep it gets impossible to train at some point. This problem occurs when the gradient becomes too large or too small, which makes it difficult to model long-range dependencies in the structure of the input dataset. The LSTM-layer, a variant of RNN, is developed to overcome this problem [11, p. 260]. The underlying principle, called Long Short-Term Memory (LSTM) was invented by Hochreiter and Schmidhuber 1997 [18]. It was the peak in their research of the vanishing gradient problem [18].

The LSTM is a special type of RNN, but it adds the possibility to save information over a longer time-span. Essentially, an LSTM-layer saves information for later use and prevents the older signal to get lost over time. Therefore, it implements recurrence in a similar way as RNNs do but instead of a single *tanh* layer, an LSTM has four layers interacting in a special way [17, p. 187].

The illustration in Figure 4.3 shows the changes of the hidden state at time-step t. The cell state c is represented by the line across the top of the diagram. It represents the internal memory of the node. The LSTM-layer also has a hidden state, which is represented by the line across the bottom of the diagram [17]. The following equations show how to calculate the hidden state h_t at time t from the hidden state $h_{t\text{-}1}$ at the previous time step [17]:

$$i = \sigma(W_i h_{t-1} + U_i x_t) \tag{4.1}$$
$$f = \sigma(W_f h_{t-1} + U_f x_t) \tag{4.2}$$
$$o = \sigma(W_o h_{t-1} + U_o x_t) \tag{4.3}$$
$$g = tanh(W_g h_{t-1} + U_g x_t) \tag{4.4}$$
$$c_t = (c_{t-1} \otimes f) \oplus (g \otimes i) \tag{4.5}$$
$$h_t = tanh(c_t) \otimes o \tag{4.6}$$

The LSTM has three gates: input gate (i), forget gate (f) and output gate (o). These

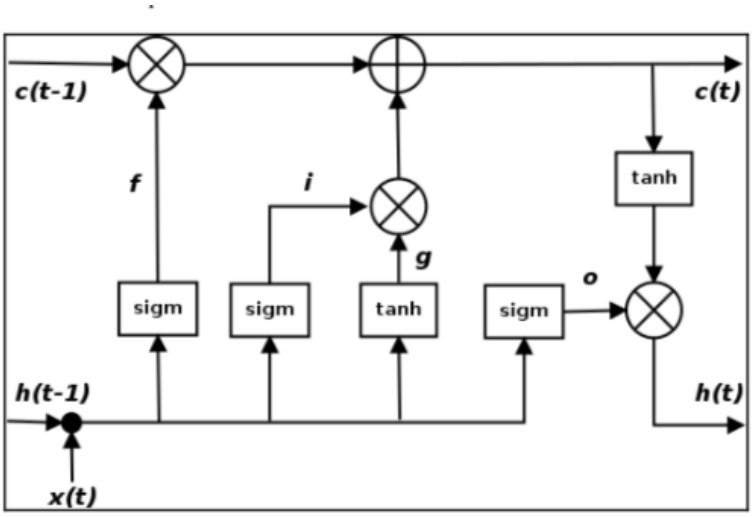


Figure 4.3: This diagram shows the repeating module in an LSTM that contains four interacting layers [17, p. 187].

gates are the main mechanism by which the LSTM works around the vanishing gradient problem. The different gates (i, f and o) are computed using the same equations but with different parameter matrices. The first step in the LSTM is to decide what information from the cell state should be forgotten. This decision is made by a sigmoid layer called the forget gate. It looks at $h_{t\text{-}1}$ and x_t and outputs a number between "0" and "1" for each number in the cell state $c_{t\text{-}1}$. When setting the forget gate to "0" the old memory is ignored while setting it to 1 all information is let through. The next step is to decide what of the newly computed state is stored in the cell state. Therefore, first, a sigmoid layer called the input gate decides which values should get updated (i). Secondly, a *tanh* layer creates a vector of new candidate values g, which represents the internal hidden state. In the next step, these two (i, g) are combined to create an update to the cell state [17].

After these steps the old state is multiplied with the computed f of the forget gate, to forget the information decided to forget earlier. The state is calculated by updating each state value by adding the update state $i \cdot g$. Finally, the output gate defines how much of the internal state should be exposed to the next layer. The output depends on the cell state, but a filtered version of it. First, a sigmoid layer decides which parts of the cell state are passed as output o. Secondly, the cell state c_t is passed through a *tanh*, which pushes the values between "–1" and "1". To receive h_t the result of the *tanh* gets multiplied with the output o so that only the computed parts get out of the LSTM [17].

4.3 Character-level Language Model

A common method to generate sequential data is to train a neural network (usually an RNN) which predicts a token or the next tokens of a sequence, whereas the previous tokens are used as input. When the input is "the cat is on the" then the neural network is trained to predict "mat" as the next word [11, p. 346].

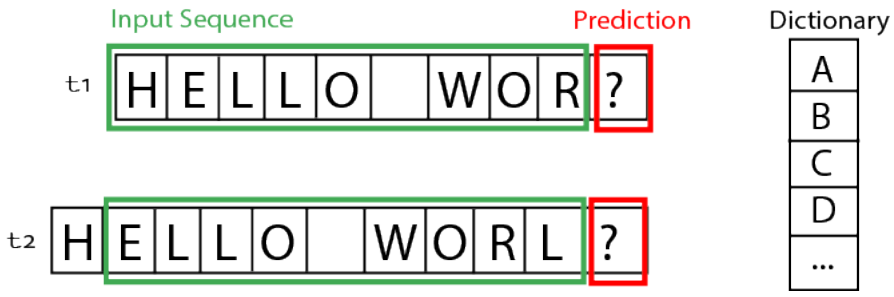


Figure 4.4: A language model predicts the next character/word after a input sequence. The next token is chosen out of the dictionary. In a word model the dictionary contains whole words whereas in a character model the dictionary consists of single characters. After predicting a character (t_1) the character is added to the sequence and the first character is pushed out. This new sequence is then used to predict the next character (t_2).

A neural network which is able to predict the probability distribution for the next token based on the previous tokens is called *language model*. A language model captures the statistical structure of a language [11, p. 346]. In most cases, language modeling (also called text prediction) is performed at word level, which means that the model is trained with sequences of words. Nevertheless, it is also possible to perform a language model on character level, called character-level language modeling [16]. In this case the model is trained with sequences of characters, so it is also able to generate new words that it has not seen before.

As far as such a trained language model is generated, new sequences can be predicted. Therefore, a text (sequence of words or characters) is given to the model and the model predicts the next word or character. To generate a longer sequence the new predicted token is added to the input and given to the model another time to repeat the process. This enables to generate a sequence of any length which structure should be similar to the input structure the model was trained with [11, p. 346]. This process is shown in Figure 4.4.

Language models are important for various higher-level tasks in the field of Natural Language Processing such as machine translation or spelling correction [17, p. 178].

Chapter 5

Methodology

In this chapter, the main process of the developed explanation method is described and the single steps, to receive an explanation for the generated text sequence, are stated. Furthermore, an overview of the used technologies is given and they are described shortly.

5.1 Visualization as Explanation Method

The main idea, to get a simple explanation method for a character-level language model, is to visualize the activation of the neurons and predictions of a character-level language model. This enables a user to gain knowledge about the learning process of the model and the decisions made for the predictions. By comparing the connection between neuron activation and predictions the decisions taken by the language model can be analyzed more clearly. This technique is inspired by previous work on visualizing neural networks as stated in Chapter 3.

Figure 5.1 shows the process of the method developed to find explanations for LSTM character-level models. The first step of this method is to find suitable input data (e.g., simple text data of a poem, Harry Potter book, etc.). This data is used to train an LSTM character-level language model. After validating and testing the model some text can be generated. This process is described in Chapter 6. The generated text contains additional information about the neuron activation and the probability distribution during the generation. This text, as well as the information, is used for the visualization application. As next step, the generated data is visualized by loading the information into the developed application. Chapter 7 describes the main features of the application. It enables the user to identify explanation patterns for the network's decisions. Chapter 8 will show some example explanations using this explanation technique.

5.2 Technology Stack

As already mentioned, the development of the method is divided into two parts: the language model and the visualization. Figure 5.2 shows the structure and the technology used for the implementation of this explanation method.

The first part describes the language model implementation, which is built with

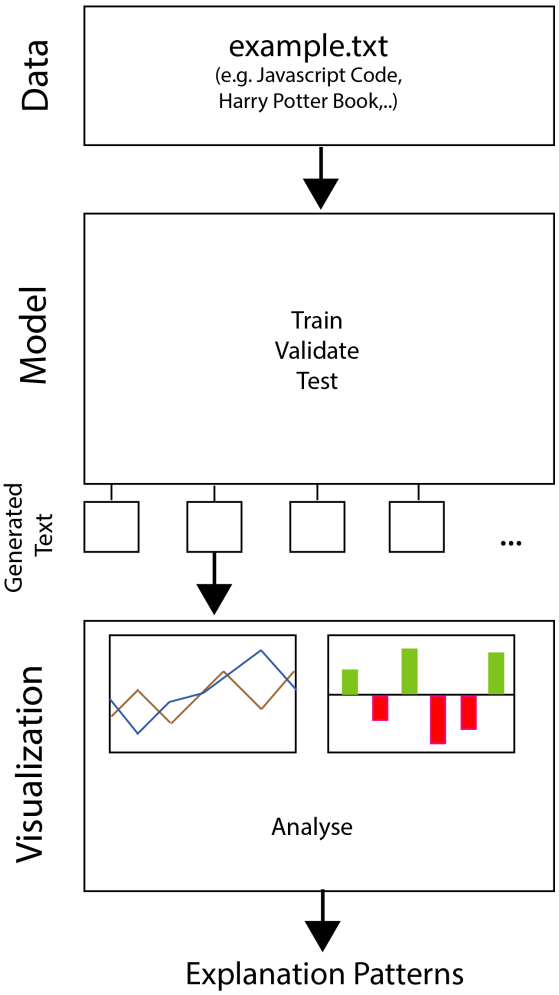


Figure 5.1: The process of finding explanation patterns contains three main steps: The implementation of the model (training, validation, testing), the generation of text and the analyzing of the decision process for those text sequences.

Keras[1]. Additionally, the machine learning library *Numpy*[2] is used. To get an intuition about the performance of the model, the plotting library *Matplotlib*[3] is in action. *Google Colab*[4] is the development environment for the implementation of the model because it provides a GPU (Graphics Processing Unit) for faster training. Furthermore, a library called *Keract*[5] makes it possible to get the necessary data (activations) from the model for the visualization. This data is structured as an object and saved into a *JSON* (JavaScript Object Notion)[6]. This step has to be done after each text generation,

[1] https://keras.io
[2] http://www.numpy.org
[3] https://matplotlib.org
[4] https://colab.research.google.com/notebooks/welcome.ipynb
[5] https://github.com/philipperemy/keract
[6] https://www.json.org

Figure 5.2: The picture shows an overview of the technology stack. The data is transferred via JSON from the Language Model to the visualization.

therefore it should be simplified as good as possible.

The second part is about the visualization, which is an interactive web application implemented with the JavaScript library *D3*[7].

The detailed architecture and applied approaches are covered in Chapter 6 and 7. Furthermore, the main decisions, the challenges faced on the way as well as the particular results are outlined in those two chapters.

5.2.1 JavaScript Framework D3

D3.js is a JavaScript library for manipulating documents based on data. The library helps to bring data to life using HTML, SVG and CSS. Due to D3's emphasis on web standards, the user has full capabilities of modern browsers. It enables the users to design the right visual interface for their data by combining powerful visualization components and the data-driven approach to DOM manipulation [52].

Furthermore, the library allows a user to bind arbitrary data to a Document Object Model (DOM) and apply data-driven transformations to the document. For instance, D3 can be used to generate an HTML table from an array of numbers. Furthermore, the same data can be used to create an interactive SVG bar chart with transitions and interaction. Generally, D3 is extremely fast, supports large datasets and dynamic behaviours for interaction and animation [52].

5.2.2 Keras

The deep learning library Keras is a high-level neural network API written in python. It was developed by François Chollet and is part of the TensorFlow core. Therefore, it can run on top of TensorFlow[8], Theano[9] and CNTK[10]. Keras can also be run on both, CPU (Central Processing Unit) and GPU. Furthermore, with Keras, it is simple to build even

[7] https://d3js.org
[8] https://www.tensorflow.org
[9] http://www.deeplearning.net/software/theano/
[10] https://github.com/Microsoft/CNTK

very complex models in a short time as it is designed to enable fast experimentation with deep neural networks.

Keras guiding principles are to be user-friendly, modular, extensible and enable working with python. *User friendliness* means that it puts user experience at first place. Keras offers consistent and simple APIs. *Modularity* means that a model in Keras is understood as a sequence or a graph of standalone, fully configurable modules. They can be plugged together with as few restrictions as possible. Hence it enables to create new models by combining standalone modules such as neural layers, cost functions, optimizers, initialization schemes, activation functions and regularization schemes. Furthermore, Keras is built to be easily *extensible*. It is also possible to create new modules which makes Keras suitable for advanced research. Finally, Keras works with *Python*. So, there do not exist separate model configuration files in a declarative format. The models are described in Python, which makes the API compact, easier to debug and expandable [58].

5.2.3 Keract

The API called Keract (short for Keras activation) is providing an easy way to access the activation map (layer outputs). The API provides functions to receive the activation of the model and also enables the user to visualize the heatmaps of these activations. Keract is developed by Philippe Rémy [57] and is a simple way to get the values needed for the neural network explanation [57].

5.2.4 NumPy

NumPy is the fundamental package for scientific computing with python. It contains a powerful n-dimensional array object, as well as sophisticated functions. Furthermore, it provides the ability to generate random numbers and contains useful linear algebra. NumPy is licensed under the BSD license[11], which enables to reuse it with just a few restrictions [53].

5.2.5 Matplotlib

The python 2D plotting library Matplotlib produces figures in different hard copy formats and interactive environments. It also supports cross-platform usage and can be used in different environments such as python scripts, Jupyter[12] notebooks and web application servers. With a few lines of code, Matplotlib enables the user to create plots, histograms, bar charts, error charts as well as scatter plots [56].

The library is often used to get an intuition about the performance of the machine learning model. The *accuracy* and *loss* over epochs are saved in the history variable and can be visualized using Matplotlib.

[11] Find the description of the license here http://www.numpy.org/license.html

[12] https://jupyter.org

5.2.6 Google Colaboratory

As described in the "Welcome Notebook" [51], Google Colaboratory (also called Google Colab or solely Colaboratory) is a free research tool for machine learning education and research. It is a Jupyter notebook environment that requires no setup and runs entirely in the cloud. Therefore a user can write and execute code everywhere directly from a browser. Furthermore, Colab enables to save and share analyses and access powerful computing resources. The notebook is not a static page but an interactive environment that enables not only to write and execute code in Python but also other languages. The most important feature that distinguishes Colab from other free cloud services is, that it provides the use of GPU.

Chapter 6

Model Implementation

To explain a model and its predictions it is necessary to implement the model and let it predict data. Hence, this chapter provides a description of the technical design of the language model, the text prediction and the extraction of the data for the explanation visualization. In addition, the main decisions are justified and the results and challenges are described.

6.1 Implementation

The first part is about creating a character-level language model (described in Section 4.3) using an LSTM (described in Section 4.2.2). The main steps of implementing and training such a model are described in this section.

6.1.1 Data Collection

At the beginning of the implementation, the main focus was to understand how a language model with an LSTM is able to close brackets. Therefore, the input data should have a clear structure, especially when opening and closing brackets. Thus, the idea was to use machine code, more specifically *JavaScript*[1] code. For the first model, a small dataset of only 170 KB (169,855 characters) is used. Therefore, the Github repository "JavaScript Algorithms"[2] containing code of many popular algorithms and data structures and the source code of the JavaScript SDK PubNub[3] was merged and saved into one text file `javascript_1.txt`. Additionally, another model was trained with a bigger dataset. As it is difficult to get a big amount of JavaScript code, the source code of some JavaScript libraries (D3, JQuery, MaterialUI, MomentJS, Require.js, Three.js, Underscore.js) were collected. The merged code was saved into the text file `javascript_2.txt`. This file has about 3 MB and more than 3 million characters. As a comparison to machine code as input data, some natural language should be tested as well. Hence, a "Harry Potter"-book was chosen to train another model with. The file "Harry Potter 4 - The Goblet Of Fire"[4] was chosen because this text file is bigger than 1 MB and contains

[1] https://www.javascript.com

[2] Find Trekhleb source code on https://github.com/trekhleb/javascript-algorithms.

[3] Find PubNub JavaScript SDK source code on https://github.com/pubnub/javascript.

[4] Find download link here: http://www.glozman.com/textpages.html.

about 1 million characters. Lastly, due to the clear structure of machine code a file with *LaTeX*[5] code was used as input data for another model. This has about 200 MB and about 200,000 characters.

The whole model implementation is the same in each of the models instead of the hyperparameters. Therefore, the next steps are generally done for all those models and the chosen hyperparameters are then described and justified in Section 6.4.

6.1.2 Data Preparing

First of all, the data (e.g., `javascript.txt`) has to be loaded into the notebook. Therefore, a connection to Google Drive needs to be started. The following code shows how to mount the Google Drive files:

```
1 from google.colab import drive
2 drive.mount('/content/gdrive', force_remount=True)
```

Running these lines of code will show a link and an empty input field. When clicking on the link the permission window for the file stream is opened and the user merely has to login to a Google account and allow the connection. Then the displayed key has to be copied into the input field of the Colab Notebook.

Afterwards, the input file has to be opened applying the following code:

```
1 def load_doc(filename):
2     # open the file as read only
3     file = open('model_data/' + filename, 'r')
4     text = file.read()
5     file.close()
6     return text
```

The `open()` command uses UTF-8 as default encoding. But, as some text file are encoded in another format (as for example the Harry Potter file), these files have to be converted to UTF-8 before because otherwise the visualization, described in Chapter 7, can not display the characters in a proper way.

After loading the data, the sequences have to be *one-hot-encoded*. The one-hot-encoding is a representation of categorical variables as binary vectors. Each word or in this case each character gets a unique integer index assigned [11, p. 234]. Thus, two dictionaries have to be created: characters to indices and indices to characters. The first dictionary is used for the encoding of the characters to unique integers (*char to indices*). The second dictionary is needed to decode the indices of the predictions back to characters (*indices to char*), as described in Section 6.1.6. Afterward, these vectors have to be converted to binary class matrices. Keras already provides a function, called `to_categorical`, which takes off this work [62].

As next step, the data has to be split into training, test and validation sets. This can ensure that the model does not overfit. Overfitting means that a model fits the data too well. It learns the training data too good so that it performs poorly on unseen datasets. Hence, an overfitting model does not generalize well. In contrast, a model is underfitting when it does not approximate the data well and generalizes too much. In this case, it is too imprecise. The goal is to train a model in a way that it generalizes

[5] Read more about LaTeX in the following link: https://latex.tugraz.at/latex/warum.

the information as good as possible and performs well on more data from similar types [33, pp. 26, 283].

The character-level language model is categorized as *supervised learning* (described in Section 4.1), which means that the algorithm learns with pairs of input (`x`) and output (`y`) data. In this case, the input data is a sequence of characters (e.g., 20 characters) and the output (also called label) is one character. Firstly the whole dataset has to be split into sequences of 20 characters. Secondly, each of these sequences has to be labeled. This is just a split of the data into input and output. For instance, when the data is "Hello World" and the sequence length is 4, then the input would be "Hell" and output (label) would be "o". The split of the input sequence is shown in Line 9 and the output in Line 11 in the following code:

```
1  def dataGenerator(data):
2    batchnum = int((len(data) - seqlen) / batchsize)
3    while 1:
4      for i in range(batchnum):
5        X_batch = []
6        y_batch = []
7        for j in range(batchsize):
8            X_batch.append(
9              data[i*batchsize+j:i*batchsize+j+seqlen])
10           y_batch.append(
11             data[i*batchsize+j+seqlen:i*batchsize+j+seqlen+1])
12
13       X_batch = np.array(
14         [to_categorical(x, num_classes=vocab_size) for x in X_batch])
15       y_batch = np.array(to_categorical(y_batch, num_classes=vocab_size))
16
17       X_batch.reshape((len(X_batch), seqlen, vocab_size))
18       y_batch.reshape((len(X_batch), vocab_size))
19
20       yield (X_batch, y_batch)
```

The process of labeling, encoding and splitting sequences of the data is done on runtime (per training and validation epoch), due to too small memory. Unfortunately, this can not be avoided, even if it is an overhead on training steps. This process is done by the generator.

6.1.3 Create Model

The easiest way to create a model in Keras is by using the sequential model [61]:

```
1 model = Sequential()
2
3 model.add(LSTM(150, input_shape=(seqlen, vocab_size), name="lstm",
      return_sequences=True))
4 model.add(Dropout(0.2))
5 model.add(LSTM(100, return_sequences=True))
6 model.add(Dropout(0.2))
7 model.add(LSTM(80, return_sequences=True))
8 model.add(Flatten())
9 model.add(Dense(vocab_size, activation='softmax'))
```

It enables stacking layers on each other with a simple `add()` function. Hence, a sequential model is created and some LSTM layers are added. Additionally, to prevent overfitting the model is regularized. The regularization prevents the model to modify the gradient in the direction where it would overfit. The `Dropout()` layer is one possible mechanism which is used to improve the training process. It randomly removes a neuron, so that it will not take part in the forward pass and backpropagation [33, p. 103]. Moreover, as the whole output sequences are required for the visualization application, `return_sequences` is set to true for each LSTM layer. Hence, before adding the `Dense()` layer [59] a `Flatten()` layer is needed to reshapes the input. The `Flatten()` layer removes all of the dimensions except of one [59].

The model is defined as a multiclass classifier, whereas more than two classes can be distinguished [13, p. 93]. In this case, those classes are represented as the different characters in the dictionary.

Before the training can be started the learning process has to be configured. Hence, an `optimizer`, a `loss` function and the metric `accuracy` is defined, as seen in the following lines:

```
optimizer = RMSprop(lr= learning_rate)

model.compile(loss='categorical_crossentropy', optimizer=optimizer, metrics
    =['accuracy'])
```

The accuracy shows the proportion of correct predictions with respect to the targets [17, p. 20]. Those hyperparameters are described in the Sections 6.3 and 6.4 in detail.

6.1.4 Fit Model

As described before, there is a need for a generator. Thus, the `fit_generator()` method is used for training instead of the normally used `fit()` method:

```
train_generator = dataGenerator(train_data)
val_generator = dataGenerator(val_data)

history = model.fit_generator(train_generator,
        steps_per_epoch = train_batchnum,
        validation_data = val_generator,
        validation_steps = val_batchnum,
        epochs = 2,
        verbose=1,
        callbacks=[generate_text])
```

As it can be seen in this code segment, the validation data, the train data, as well as the number of epochs and the steps per epoch, are needed by the generator.

During *training* of the model the accuracy and the loss is shown in real time. At the end of each epoch, the validation accuracy and loss is shown for checking the quality of the training. Additionally, in order to analyze the quality of the training, sequences are generated after some epochs (e.g., after each second epoch). This is done with the callback function:

```
generate_text = LambdaCallback(on_epoch_end=on_epoch_end)
```

This is called after each training epoch, which enables to retrace the learning process. The text generation process is explained in a later section.

6.1.5 Visualizing the Training Process

The accuracy and loss of the training as well as of the evaluation is saved for each epoch. These values can be visualized by using the library Matplotlib (see Chapter 5):

```
# Plot training & validation accuracy
plt.plot(history.history['acc'])
plt.plot(history.history['val_acc'])
plt.title('Model accuracy')
plt.ylabel('Accuracy')
plt.xlabel('Epoch')
plt.legend(['Train', 'Test'], loc='upper left')
plt.show()

# Plot training & validation loss
plt.plot(history.history['loss'])
plt.plot(history.history['val_loss'])
plt.title('Model loss')
plt.ylabel('Loss')
plt.xlabel('Epoch')
plt.legend(['Train', 'Test'], loc='upper left')
plt.show()
```

The plotting helps to get an intuition about the performance of the model and the quality of the training.

Additionally, after training the model the *evaluation* is executed. This provides supplementary information for the training and if it does not overfit or underfit.

6.1.6 Text Generation

After the training, validation and evaluation, the model is ready to predict text, as done in the following code seqment:

```
# -------------
# 1. Predict from input
p = model.predict(pred_input, verbose=0)[0]

# -------------
# 2. Choose next char with highest prob or diversity
next_index = sample(p, diversity)
next_char = indices_char[next_index]

# -------------
# 3. Save data
[...]

# -------------
# 4. Append to input
whole_seq += next_char
curr_seq = curr_seq[1:] + next_char
```

This type of model is categorized as sequence prediction, which involves predicting the next value for a given input sequence [9]. Therefore, a random sequence of the initial data is chosen for the input sequence. Then, the data is given to the model to make a prediction, as shown in Line 3. Only one character is predicted at the time, so for generating a longer sequence the process has to be repeated several times. The part of saving data in Line 12 is described in the next Section 6.2.

Additionally, some temperature is added to the prediction to receive a text with a higher diversity. This helps to obviate repetitions and make the predictions of the model more varied. The function `sample()`, which is shown in code segment Line 7, is choosing one character based on the predicted probability with a specified temperature (also called diversity). Too low diversity generates repeating text and the model can be stuck in a loop, whereas too high diversity corresponds with more risk for errors. A minimal diversity of 0.5 seems to be a good balance [31].

6.2 Data Saving for Visualization

The missing part of the section before is the data saving step:

```
1 # -------------
2 # 3. Save data
3 a = get_activations(model, pred_input)
4 single_data = dict()
5 single_data["x"] = curr_seq
6 single_data["y"] = next_char
7 single_data["act"] = format_activations(a)
8 single_data["pred"]= get_highest_predictions(char_indices, p)
9 data_sequences.append(single_data)
```

This is a main step before the visualization can be implemented. After each generated character the following values are saved into a defined array: The input sequence (`x`), the output (`y`), the activation (`act`) of hidden neurons as well as the probability distribution (`pred`) for each character in the dictionary.

The generated object `data_sequences` has to be stored (described in Chapter 7). As the visualization is done with D3, the object is saved as JSON:

```
1 with open("./vis_data/"+file_name, 'w') as outfile:
2     print("writing file to: ",file_name)
3
4     json.dump({
5         "dictionaries": {
6             "char_indices": char_indices,
7             "indices_char": indices_char},
8         "sequences": seq_data},
9         outfile, indent=4, cls=NumPyArangeEncoder)
10 outfile.close()
```

The whole structure and values of the JSON data are described in Chapter 7.

6.3 Hyperparameter

While creating and especially for the training of the model, different hyperparameters were tested in order to find the optimal combination of values to receive a model with good metrics and predictions (called Hyperparameter tuning). This section describes the different model hyperparameters as well as the training hyperparameters used for all the models. There exist two types of parameters in Machine Learning: Model parameters and parameters related to the training of the network [33, p. 78]. This section is split into these parts and describes the single hyperparameters. Section 6.4 then contains the specific parameters used for the four different models.

6.3.1 Hyperparameters related to Network Structure

The following hyperparameters are used in relation to the network structure:

Layers

As described in the Chapter 4, a neural network consists of one input layer, several hidden layers and one output layer. The input and output layers are defined by how the model handles input and output. Therefore, the input layer has a layer size of 20 as this corresponds to the number of features (sequence length) in the input vector. For the output layer, it is the size of the dictionary created at the begin of the data preparing as this layer size corresponds to the number of classes that it is trying to predict [33, p. 100].

The hidden layers for the sequential model consist of several LSTM layers with different amount of hidden units. These are all different for the implemented models depending on the input data. Generally, it can be added an arbitrary number of neurons and layers as there are no rules for the size of a neural network [33, p. 100]. Additionally, for each of the LSTM layers the `return_sequences` parameter is set to true because all output values are needed for the visualization, as already described before. Hence, an additional `Flatten` layer is needed before the output can be created. The specific model structures for each of the trained models are shown the Section 6.4.

Dropout

As already described in Section 6.1 the dropout defines the percentage of values dropped out during the training process. It is a good regularization technique to prevent overfitting [33, p. 103]. In most of the models a dropout of about 0.5 is used after each LSTM layer. Currently, this seems to be the best option as it minimizes the overfitting in the best way. When using more dropout the learning progress would be too slow as too many neurons are set to zero.

Activation function

Activation functions are used to propagate the output of the layers nodes forward to the next layer. For the *output layer* the activation function *Softmax* is used. The output of a *Sigmoid* is a value between 0 or 1, but its sum does not need to be 1. Therefore, it is used for binary classification. But, all the probabilities of the Softmax sum up to

one and therefore Softmax is used in multi-class predictions [65]. The Softmax is often used in an output layer of a classifier [33].

The LSTM layer in general uses the `tanh` activation function. The main problem in a recurrent network is the vanishing gradient problem and to keep the gradient in the linear region of the activation function. To overcome these problems, there is a need for a function whose second derivative can sustain for a long range before going to zero. Hence, as an activation function `tanh` is used, which can slightly solve those problems [33, p. 67].

Loss

The *loss* is a measure on how good the model is at achieving the given objective. Therefore a metric is calculated based on the error observed out of the network's predictions (how far away the prediction is in comparison to the goal). These errors are aggregated over the entire dataset. The average of this number represents how close the network is to the ideal [33, p. 71].

For the loss functions[6] there are some common choices: `mean squared error` (MSE), `binary cross-entropy` and `categorical cross-entropy`. For the implemented models the categorical cross-entropy is chosen as loss function. This is the most common choice for multi-class classification problems and it is also the default choice in association with Softmax activation. A lower score calculated with categorical cross-entropy indicates that the model is performing better. Therefore the aim of the hyperparameter tuning is to decrease the loss as much as possible [17, p. 19].

Optimizer

Machine Learning can be seen as an equivalent to an optimization problem in which the loss function should be minimized with respect to the parameters of the prediction function. Hence, the `optimizer` is used to minimize the loss by updating the weights using the gradients. The selected loss function is then used by the optimizer to navigate the space of weights. The process of optimization is also defined as a process of loss minimization [33, p. 96]. In the implemented models, the `RMSprop` [60] optimizer is used, as it is a good choice for recurrent networks. It is a very effective, but currently unpublished adaptive learning rate method [33, p. 103]. Another possible optimizer for this type of model would be `Adam` [60]. As there is no rule of thumb which of the optimizer is the best and `RMSprop` performs better for the implemented models the choice fell on `RMSprop`.

6.3.2 Hyperparameters related to Training Algorithm

The following hyperparameters are optimized to improve the training of the network. This section, as done before in the previous section, describes solely the general settings. The detailed values for each of the models are described in the later Section 6.4.

[6] Loss function is also called objective functions or optimization score function. For more information, refer to https://keras.io/losses/.

Learning Rate

Another parameter that can be changed is the learning rate for the optimizer. It is one of the most important hyperparameters that can be set to tune a NN [33, p. 285].

The learning rate has an effect on the amount of adjustment of the parameters during the optimization, in order to minimize the error of the neural networks guesses. When using a too high learning rate it can save training time but may miss the local minimum. A too low learning rate might need a lot of training time and possibly does not find the global minimum, but only the local minimum [33, pp. 78–79].

Generally there is no specific learning rate that should be used as this always depends on the specific use case. But, the optimal learning rate is somewhere close to 0.001, which is the default learning rate for the optimizer. In all the models the learning rate is set between 0.001 and 0.0001 [33, p. 103].

Number of epochs

Once the model is compiled, it can be trained. Therefore, the number of epochs have to be defined. The epochs define how often the model is exposed to the training set. During each iteration, the optimizer tries to change the weights in a way that the loss is minimized [17, p. 20]. In the implemented models the number of epochs differs from 5 to a maximum of 50 epochs. This decision is based on the performance of the model and the degree of overfitting.

Batch size

Another parameter that has to be defined for the training is the batch size. It defines the number of training instances observed before the optimizer updates the weights and biases [17, p. 20]. During the training it manifests the following: The higher the batch size the slower but more accurate the model learns and the other way round. Therefore, if the batch size is too high it may stop learning.

6.4 Results and Evaluation

This section provides an overview of the results for each of the trained models. This overview contains the settings for the hyperparameters as well as an explanation of choosing them. Furthermore, the section shows some generated text to demonstrate how the models perform. For all the predictions of the models, a random sequence (20 characters) is chosen out of the input data and different values for the diversity (from 0.2 to 0.8) are tested. Then in most cases a text with 100 characters is predicted.

6.4.1 Model 1: JavaScript (small)

The first model is trained with JavaScript machine code. This was chosen due to the clear structure in comparison to natural language text. As already mentioned two models were trained with JavaScript code, one with an input file of about 170 KB (169,855 characters) and another one with an input file of 3.3 MB (3 million characters), which is then described in the next ection.

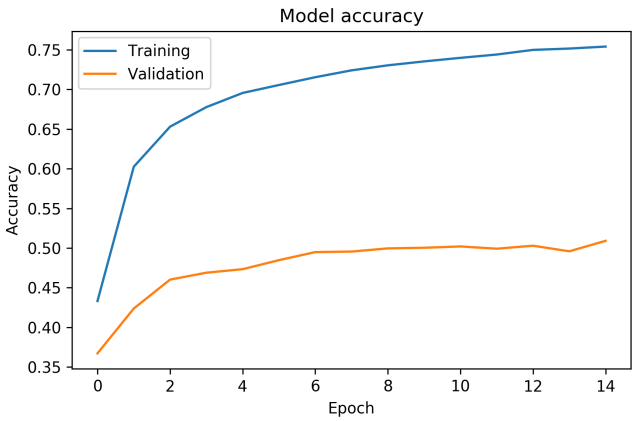


Figure 6.1: Learning curve of the model shown by the accuracy.

Figure 6.2: Learning curve of the model shown by the loss.

After training the model for 15 epochs it has a training accuracy of 75%, a validation accuracy of 50% and a test accuracy of 44%. After these 15 epochs, the model starts overfitting very much and the validation accuracy stays the same at this point. The line graph, created with Matplotlib, in Figure 6.1 shows the change of the training and validation accuracy over the epochs. It can be clearly identified that the model is overfitting as the training accuracy is so much higher than the validation accuracy and the test accuracy is also lower than the validation accuracy.

Furthermore, Figure 6.2 shows the change of the training and validation loss. The curve also demonstrates that the model is overfitting because the validation loss (that should decrease) is increasing after the third epoch.

Hyperparameter

The Figure 6.3 shows an overview of the network structure. Generally, in the beginning, the model learned too fast and therefore not that good. Hence, the idea was to increase

Layer (type)	Output Shape	Param #
lstm (LSTM)	(None, 20, 246)	332592
dropout_1 (Dropout)	(None, 20, 246)	0
lstm_2 (LSTM)	(None, 20, 246)	485112
dropout_2 (Dropout)	(None, 20, 246)	0
flatten_1 (Flatten)	(None, 4920)	0
dense_1 (Dense)	(None, 91)	447811

Figure 6.3: This Figure shows the structure of the small JavaScript model.

the batch size so that the model learns slower but better. This resulted in the problem that the models stopped learning anything. So, the network size has to be increased as well to reach the desired effect. This helped a lot but unfortunately, the model never had more than 50% validation accuracy.

In the end, a batch size of 64 was used, because when using a higher or lower batch size the model starts to overfit much earlier. So, with a lower batch size, the model has a validation accuracy of only 47% and when choosing a higher batch size the validation accuracy stopped at 45% to increase. Furthermore, two LSTM layers with 246 hidden nodes were added to the model. This layer size was chosen because with more nodes the model does not get better and with more layers, it overfits more than before. Furthermore, during the training, it seems to be a connection between the batch size and the network size. So, when increasing the batch size it was often necessary to increase the network size as well. This is maybe due to the fact that the model with a higher batc hsize needs more nodes to process the data. As previously described in the implementation, the model is regularized with a `Dropout()` layer. So, a dropout with 0.5 was added after each LSTM layer. This decreases the degree of overfitting a little bit but when using a higher dropout the model learns too slow. The learning rate for the optimizer is set to 0.003. With a lower learning rate, the model needs much more epochs to learn nearly the same but it starts overfitting earlier at 0.46. With a higher learning rate, it learns too fast which also leads to more overfitting.

Predictions

After training the model, some JavaScript code is generated. This is done to measure the models performance in comparison to the metrices. So, despite the small accuracy, the model is able to predict almost good JavaScript code:

```
fault class Comparator {
  /**
   * @param {*} value
   * @return {boolean
   */
   constr
```

This is a predicted text generated by the JavaScript model during the learning process. The sequence was predicted with a diversity of 0.2, which was randomly chosen. Another

predicted text shows that the model sometimes forgets to close brackets:

```
----- Generating with seed: "y(a, b) {
   if (th"
----- diversity: 0.2
y(a, b) {
   if (this._db.get$(`{this._dbaget);
```

This is a generated sequence of the JavaScript model after finishing the training.

Thus, these results are not the best but still useful for the explanation visualization in a later step. The language model is able to generate words separated by empty spaces. It opens and closes brackets and it is also able to define the end of the comment over several lines. Nevertheless, it sometimes makes errors and for example, forgets to close a bracket.

6.4.2 Model 2: JavaScript (large)

As described before another model is trained with JavaScript code as input data. This time with a larger amount of data with the intent to receive a better model. Unfortunately, during training, a problem occurred. Google Colab has a limit in memory space and when training a model with more data the following error was printed to the output: "Buffered data was truncated after reaching the output limit". This error occurs because RAM or GPU memory was full and could not process new data. This already occurred after 5 epochs of training with the same hyperparameters as for the model with the small amount of input data. To solve this problem the model has to be saved after just a few epochs and then the training has to be started again from this point.

While trying many different hyperparameters the best result does not seem to be better than the model with a small dataset. This is probably because due to the fact that the data is more complex and so more classes has to be distinguished. After training the model twice for 5 epochs a training accuracy of 74% and a validation accuracy 46% cannot be beaten. Figure 6.4 shows the graph with the training and validation accuracy changing over the first 5 epochs. For the next 5 epochs, the accuracy only improves by 2% and then stops improving. These results are indicators for overfitting.

This can also be proven with the loss curve shown in Figure 6.5. The training loss nearly gets under 1 but the validation loss does not get under 3, which indicates that the model is overfitting very much.

Hyperparameter

The before described results were accomplished with the following hyperparameters. Figure 6.6 shows the structure of the model. It consists of three LSTM-layers with 600 hidden nodes. Those were chosen because during training it manifests that the structure is so complex that more nodes are needed to get better results. But when increasing the network size (adding more nodes and layers) a higher batch size is needed as otherwise, the model would overfit very fast. Additionally, as for all other models, a dropout of 0.5 was added after the LSTM layers. Lastly, a learning rate of 0.001 seems to be the best solution for this model. Increasing this rate would make the model learn too fast and therefore too "sloppy". Decreasing the learning rate would only let the model learn very slow but do not improve the results.

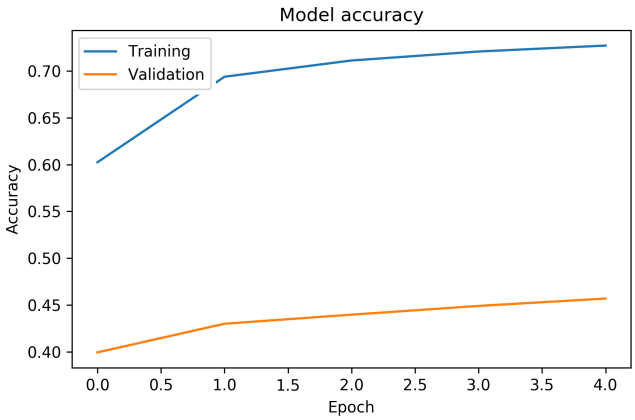


Figure 6.4: Learning curve of the model shown by the accuracy over the first 5 epochs.

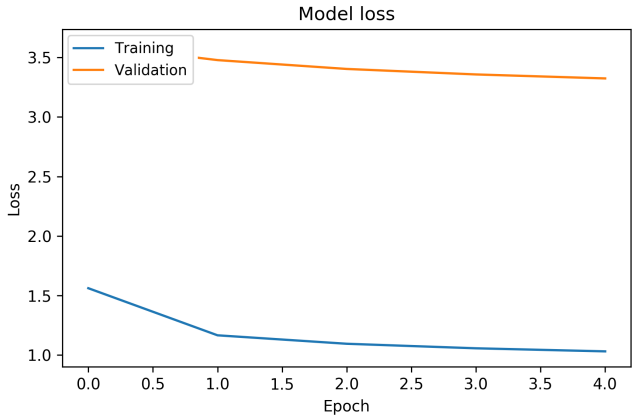


Figure 6.5: Learning curve of the model shown by the loss over the first 5 epochs.

Layer (type)	Output Shape	Param #
lstm (LSTM)	(None, 20, 600)	1752000
dropout_1 (Dropout)	(None, 20, 600)	0
lstm_1 (LSTM)	(None, 20, 600)	2882400
dropout_2 (Dropout)	(None, 20, 600)	0
lstm_2 (LSTM)	(None, 20, 600)	2882400
dropout_3 (Dropout)	(None, 20, 600)	0
flatten_1 (Flatten)	(None, 12000)	0
dense_1 (Dense)	(None, 129)	1548129

Figure 6.6: Structure of the bigger JavaScript Model.

Predictions

As for all other models, some sequences are generated to measure the performance of the model. As the following example shows, the model is able to predict very good JavaScript code:

```
----- Generating with seed: "if(this.count >= 209"
----- diversity: 0.4
if(this.count >= 209) {
            return this._isValid() : this.isUTC ? false;
```

This text was predicted with a diversity of 0.4, which was randomly chosen. Another predicted text is predicted with a diversity of 0.5:

```
----- Generating with seed: "tartAngle + t * delt"
----- diversity: 0.5
tartAngle + t * delta;

        if (!isNaN(input) !== 0) {
               if (match[i]) {
```

As the model before, the results are not perfect but still very good and useful for the explanation visualization in a later step. The model is able to generate words, start if-clauses, it opens and closes brackets. Nevertheless, it makes some mistakes and is not that good in predicting the right intention. Even if the accuracy is not better than the one trained with fewer input data the predictions of the model trained with more data seems to predict better sequences. The model makes fewer errors. So, maybe with more data and a lower learning rate, the model could be improved even more.

6.4.3 Model 3: Harry Potter

The third model is trained with natural language to be able to compare the differences in the structure and in their interpretability. Therefore, as described before in the implementation, a Harry Potter book was used as input data.

Training the model for 30 epochs resulted in a training accuracy of 65%, a validation accuracy of 64% and a test accuracy of 64%. Figure 6.7 shows the change of the training and validation accuracy over the epochs. These curves show that at the beginning the validation accuracy is higher than the training accuracy. This is probably because the network is struggling to fit the training data. Here it might help to use a bigger network.

The Figure 6.8 shows how the training and validation loss changes through the epochs. Here, the validation loss is also better than the training loss. But the validation loss stops improving after the 15th epoch. This is an indicator for overfitting so the training is stopped after the 25th epoch.

Generally, this is a better result than for the JavaScript model but it is also improvable. Maybe it can be improved with more input data but unfortunately, it was not possible to test it because of the Google Colab timeouts due to the lack of memory space.

Hyperparameter

The network structure for this model is presented in Figure 6.9. It shows that the model has three LSTM layers with 546 hidden nodes. A smaller network has performed worse

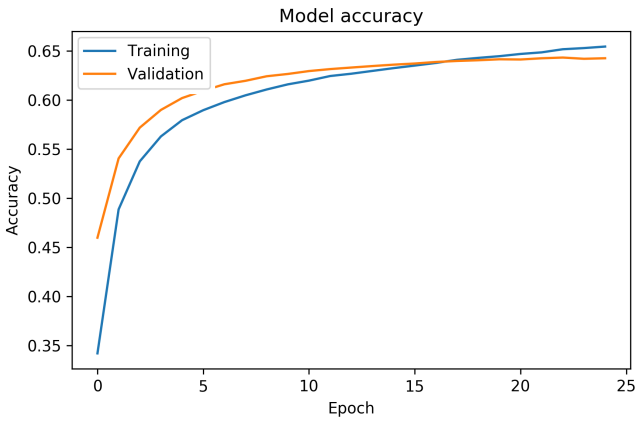


Figure 6.7: Learning curve of the model shown by the accuracy.

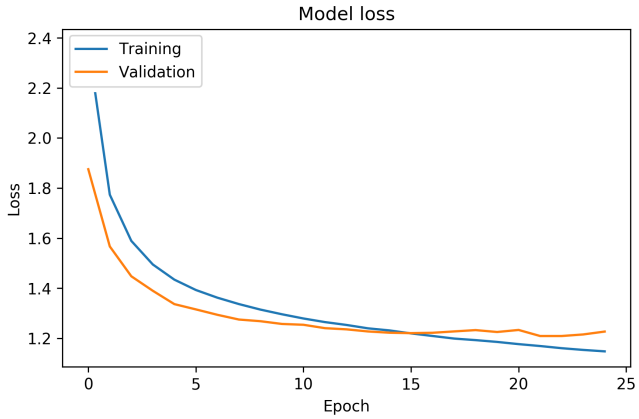


Figure 6.8: Learning curve of the model shown by the loss.

as it overfits much more and the model did not reach the validation accuracy of 64%. With a larger network, the same problem occurred. In combination with this network size, a batch size of 312 seemed to fit best. As for the other models a 0.5 dropout was added after each LSTM layer. Moreover, the learning rate of 0.001 was chosen because it performs best with it. With a higher learning rate the model overfits too much and with a lower learning rate it did not reach the maximum of 64% validation accuracy. These hyperparameters results in a model that is able to generate really good sequences, as the next section proves.

Predictions

After training the model with the hyperparameters described above, some sequences with different input sequences were predicted by the model. To analyse the efficiency of the model there were generated sequences several times and the following examples show the results of this process. Sometimes the predictions of the model are really good

Layer (type)	Output Shape	Param #
lstm (LSTM)	(None, 20, 546)	1369368
dropout_1 (Dropout)	(None, 20, 546)	0
lstm_1 (LSTM)	(None, 20, 546)	2387112
dropout_2 (Dropout)	(None, 20, 546)	0
lstm_2 (LSTM)	(None, 20, 546)	2387112
dropout_3 (Dropout)	(None, 20, 546)	0
flatten_1 (Flatten)	(None, 10920)	0
dense_1 (Dense)	(None, 80)	873680

Figure 6.9: Structure of Harry Potter Model.

as a sequence demonstrates:

```
----- Generating with seed: "rum's head. Several "
----- diversity: 0.5
rums's head. Several group of the fire.

"What sooner what the words were back in the party, and it was pretty
that champ
```

Nonetheless, sometimes there where repeating phrases, such as in the following sequence:

```
----- Generating with seed: "ike trying to dance "
----- diversity: 0.5
ike trying to dance to the ball with the table and saw the start of the castle
with the back of the way to the ball with
```

The problem is that the model always predicts the same phrases after the other (e.g., “and the stairs and the stairs and ...”) and it seems like a never-ending sentence. This is maybe due to a too small diversity and that the sentences in Harry Potter are much longer than 100 character. So, a solution was to use a higher diversity and to let the model predict a longer sequence of 300 characters, as the following example sequence shows:

```
----- Generating with seed: "eeting Ron for the f "
----- diversity: 0.5
eeting Ron for the first task. It was heading hard to bed before the clothes
were still closed in the end of the common room.

The gallent look had been having the students too. Professor McGonagall in the
entrance hall with the boy who was a lot many hat, but this was a student that
was so that about the ruin, and the
```

These predictions look much better than the examples before as it also predicts an end of the sentence. Nevertheless, the sentences do not really make sense.

These results are of course improvable but for the size of input data and training time, it is a really good result. The model was able to generate words, sets commas, quotation marks and it knows which words should be written in uppercase and lowercase.

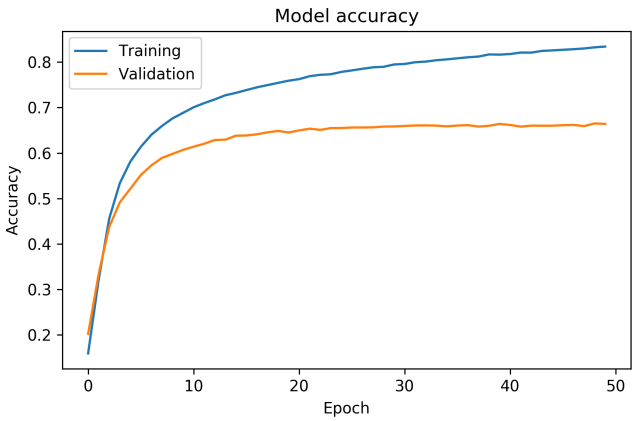


Figure 6.10: Learning curve of the model shown by the accuracy.

Furthermore, it learns to write in the same way as it saw it in the input data. The model writes very long sentences and predicts quotation marks at the correct position, as seen in this predictions:

```
----- Generating with seed: "hose girls got to?"
"
----- diversity: 0.5
hose girls got to?"

"What? said Ron, looking at him. What is that was a bit of going that the
color carried and bushi
```

The only problem is that the sentences do not really make much sense. But, with more training data and a bigger network, this can probably be fixed. In general, it is more difficult to measure the performance of the model as there is less syntax in the natural language as in machine code. So, here it is only possible to look at correct words, points, commas, uppercase and lowercase. But to write a text with the right content is so much harder for a model.

6.4.4 Model 4: LaTeX

After training the third model (natural language), it became clear that machine code is better understandable in the context of explainable NNs. The clear structure of machine code makes the generated text easier to analyze the models quality of learning. Therefore, the last model was trained with a large file containing LaTeX machine code.

After training the model for 50 epochs the model results in a training accuracy of 83%, a validation accuracy of 66% and a test accuracy of 62%. This model has a much higher training accuracy but overfits more than the other models. Thus it is of couse also improvable. The change of the training and validation accuracy is shown in Figure 6.10. As seen in this graph, the training accuracy improves faster than the validation accuracy, which indicates that the model is overfitting.

This is also proven with the loss curve, as shown in Figure 6.11. Over the 10th epoch, the validation loss does not decrease anymore but increases steadily. Hence, the

Figure 6.11: Learning curve of the model shown by the loss.

Layer (type)	Output Shape	Param #
lstm (LSTM)	(None, 20, 300)	477600
dropout_1 (Dropout)	(None, 20, 300)	0
lstm_1 (LSTM)	(None, 20, 300)	721200
dropout_2 (Dropout)	(None, 20, 300)	0
flatten_1 (Flatten)	(None, 6000)	0
dense_1 (Dense)	(None, 97)	582097

Figure 6.12: Structure of LaTeX model.

training loss resulted in 0.55 whereas the validation loss was 1.68. As already seen in the accuracy this is a better result than with the before trained models. So, maybe a character-level language model with an LSTM is better in learning well-structured text.

Hyperparameter

To achieve these results a batch size of 312 is used. A smaller batch size leads to a model with only 52% validation accuracy whereas a model with a higher batch size does not really changed the result. Moreover, the model is built with two LSTM layers with 300 hidden nodes. This is chosen because in combination with the batch size this seems to fit best for the input data. Furthermore, a 0.5 dropout is added after each LSTM layer as seen in the network structure presented in Figure 6.12. The dropout also improved the model and prevents it from overfitting too much. Finally, the last parameter which was tuned was the learning rate. For this, a value of 0.001 was chosen because when testing a higher or lower learning rate the model does not perform better or worse.

Predictions

The model, trained with the hyperparameters described above, is able to generate some really good LaTeX code. The following sequence is one example where the efficiency of the model is presented:

```
----- Generating with seed: "\overbrace{\rule(15m"
----- diversity: 0.5
\overbrace{\rule{15mm}{0pt}}^{\displaystyle\boldsymbol{b}_0 - \mathbf{B}$
bCect of $\mathbf{A}$ and the extrinsic parame
```

Another predicted sequence of the LaTeX model shows the ability of the model to open and close brackets in the correct order:

```
----- Generating with seed: "    \text{---} & \bolds"
----- diversity: 0.5
    \text{---} & \boldsymbol{v}_{1,1}(\mathbf{H})$
                           \Comment{$\boldsymbol{h}_2 = (H_{0,2}, H_{1,2},
                           H_{2,2}, H{2,2}, H_
```

As the examples above demonstrates, the LaTeX model is able to remember on important parts over a longer time range. Furthermore, it differentiates between code and text and separates the words with empty spaces if necessary. Thus, most of the time the predicted sequences look like real LaTeX code.

6.4.5 Evaluation

Generally, during the implementation and the training of the different models it turned out that at some point a bigger network (more layers or more hidden nodes) require a higher batch size, otherwise, it starts underfitting, overfitting or even learns nothing. Furthermore, the size of the network structure is very important. On the one hand, a too small network leads to a model that is not able to represent the problem in a proper way. So, it underfits and often does not get a high accuracy. On the other hand, a too large network leads to overfitting very much. Thus, the goal was to find parameters which fit best together and represent the input data in a good way.

At first glance, the first model (small JavaScript input) seems to be better than the second one (large JavaScript input) with more input data. The metrics for the first model are better but not the predictions. Thus, with more input data the predictions of a model look much better. This means that the metrics are not always that meaningful. The third model, which was trained with a Harry Potter book, has a lower training accuracy than the first two models, but has a much higher validation accuracy. This indicates that the model does not overfit that much. Nonetheless, a model with only 66% accuracy will probably only guess. Thus, it seems that the LSTM language model is better in learning machine code than natural language. If the input has a clear structure the model has the possibility to learn clear rules in contrast to a natural language where context is more important. Hence, the results of the last model are not surprising but affirmative. It is the best of the four trained models. The model is really good in learning the structure of the LaTeX code. It is of course overfitting but the predictions are comparable to real LaTeX code.

At least it is important to say, that all four models are able to predict really good sequences, especially the single words are right most of the time. It must be emphasized

that this is about character-level models and therefore these results are really good.

Nevertheless, the four models are of course improvable but as the focus is set on the explanation visualization, the performance of the model is not the priority. Therefore, the models were trained and adapted only so good that they generate text halfway similar to the structure of the input data. Those generated sequences are analyzed using the visualization described in the next chapter.

Some ideas to improve the models were for example to increase the network size, decrease the learning rate and the most promising improvement would be to use much more input data.

6.5 Design Decisions

- *Why use a GPU?* A CPU is used for more general computing workloads, whereas a GPU, in contrast, is less flexible. However, GPUs are designed to perform in parallel the same kind of computation. A NN is structured in a very uniform way, such that each layer of the network contains thousands of identical artificial neurons which perform the same computation. Hence, the parallel processing capabilities of GPUs can accelerate the NN training and inference processes [69]. Furthermore, GPUs are the standard for LSTM usage [64].
- *Why use Google Colab?* The main reason for using Colab instead of the local Jupyter notebook is that there is a need for a powerful GPU to train NNs. Hence, Google Colab was a good solution, as it provides a free Tesla K80 GPU. An additional advantage of Colab is that the model can be trained on each computer without any installation effort. Furthermore, it enables to use and save the files on Google Drive in an easy way [51].
- *Why use Keras?* The project's focus lies on the visualization and therefore easy and fast prototyping is needed (through user friendliness, modularity and extensibility). Furthermore, there is the requirement of being able to create a sequential model, especially an LSTM. As Keras fits all those desires, it is the best solution for this project.
- *Why use an LSTM?* There is no CNN used because it is unable to learn from the previous events as the information does not pass from one step to the next [15]. Furthermore, the main reason why there is no RNN used is that it forgets old information after a certain amount of time, no matter how important it was. As described in the Chapter 4, an LSTM fixes this issue using a considerably more complex memory. It contains four interacting elements in a repeating module, the part of the model which passes information on to the next time step. Hence, it makes use of a cell state, which passes straight through each instance of the network, and allows important information to pass onto the next time step largely unchanged. So, it effectively lengthens the network's long term memory duration [18]. Furthermore, RNNs in general and LSTMs, in particular, received the most success when working with sequences of words and paragraphs, generally called natural language processing [49]. Finally, the main reason for choosing an LSTM is that it is the only network that is able to discover learning patterns, as Hendrik Strobelt [44] said:

> For instance, it has been shown that RNNs can count parentheses or match quotes, but is unclear whether RNNs naturally discover aspects of language such as phrases, grammar, or topics.

So, since the project is a language model with sequential data, an LSTM is used for the project.

- *Why use JSON?* JSON (JavaScript Object Notion) is a JavaScript data structure where the indices are named [55]. For nested data or for passing around data it is hard to beat JSON. Furthermore, it has become the language of the internet for good reason. It's easy to understand, write and parse. Virtually all modern programming languages support it in one form or another. Therefore also D3 supports the import of data in JSON format. And, as python is also supporting JSON by providing a library to jsonify an object and saving it, there is no reason for not using it.

6.6 Challenges

During the implementation, training and hyperparameter tuning of the model some problems and challenges occured. Those are described and their solution are presented.

- *Data preparing for LSTM:* As described in Chapter 6, there is a big amount of text needed for the training (about 1MB). But due to the fact that Google Colab provides too small memory space, the encoding of the data was a challenge. Therefore, the solution was to encode and label the data on runtime with a `DataGenerator`, as described in Section 6.1. Despite the usage of the generator, there is too small memory space available. As described in the Section 6.4, when using a bigger batch size the RAM was still to small. There where just two possible solutions for it. The first solution was to minimize the batch size and the second solution was to stop the model after a few epochs, save the model and start the training again from this point.
- *Generating Text with Diversity:* Sometimes the model is predicting the same character or words more often and gets stuck in a loop (e.g., this.s = this.s = this.s). For preventing the model to generate always the same data and to make the predictions more "creative" a temperature is added. This makes the predictions a little bit more random as it is not always choosing the prediction with the highest probability but with a specified diversity. A lower temperature (e.g., 0.2) will generate more "safe" predictions, whereas a temperature above 1.0 will start to generate "riskier" guesses. This was described in detail in Section 6.1.6.
- *Google Colab Reconnecting:* Google Colab is intended for interactive use, which results in long-running background computations, particularly on GPU, that are stopped after some time [51]. Hence, during the whole training time (which are more than 4 hours) someone has to scroll a little bit in the notebook or click on "reconnect". This makes the training of the model in Colab a little bit annoying because the notebook session died for no reason and this leads to a loss of the trained model and the work of a day.
- *Overfitting problems:* During the training itself several problems were encountered. First of all, there were some overfitting problems due to too few data.

Therefore a solution was found with using the JavaScript libraries code (described in Chapter 6). Nevertheless, there was still too few data but as mentioned before, the RAM was to small so it was not even possible to put more data in it. Secondly, it was really hard to find the right values for hyperparameters e.g., the batch size, learning rate and the number of layers as well as hidden unit size. But in the end the four models where able to generate sequences in a rarely good way even though they could be improved.

Chapter 7

Language Model Explanation

This chapter gives a description of the architecture, features and technical design of the visualization. Finally, design decisions, the results, as well as the challenges faced during building the visualization, are explained.

7.1 Design Result

The following section provides an overview of the results, the design as well as the main features of the visualization.

7.1.1 Design

First of all, in Figure 7.1 the generated text is shown on the x-axis. To make the visualization easier understandable both features (predictions, activations) are deactivated. The text is always generated one character after the other, therefore when talking about "time" the generation of one character is meant. The x-axis also contains the input sequence, which is represented by the first 20 characters.

Secondly, when enabling the prediction selection, the dictionary, the y-axis (left) and the line chart appears, as seen in Figure 7.2. The line shows the probability of the selected character over time. There are no predictions shown for the first characters because during the input sequence (first 20 characters), no predictions are generated.

Thirdly, when enabling the activation selection, the layer and cell (hidden unit) selection, the y-axis (right) and the bar chart is shown (see Figure 7.3). If the user selects one cell the bar chart is updating and the activation of this cell over time is shown. The y-axis on the right side demonstrates that the activation has a range of "-1" to "+1", whereas all negative values are colored red and all positive are colored green.

Additional to the selection of the layer and cell a network structure visualization, shown in Figure 7.4, is provided for the user. In the visualization only LSTM layer activations can be shown, therefore only these layers are colored (as they are clickable).

Finally, when enabling both features (activation and prediction selection), the two charts are superimposed. As described before this enables the user to compare and find similarities of the neurons' activations and predictions. An example is shown in Chapter 8.

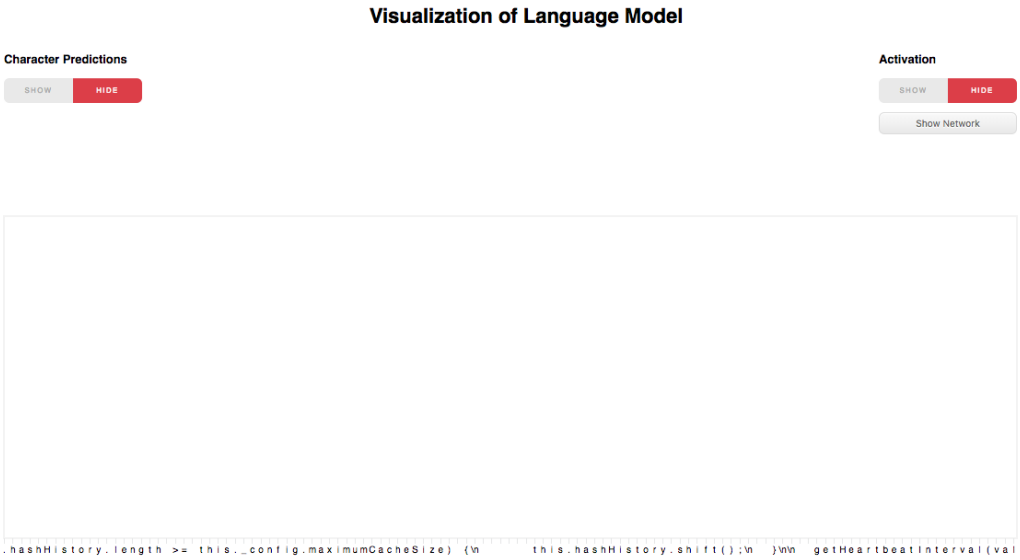


Figure 7.1: This is a screenshot of the visualization which shows the generated text on the bottom. All features are deactivated, as indicated by the left and right toggle buttons.

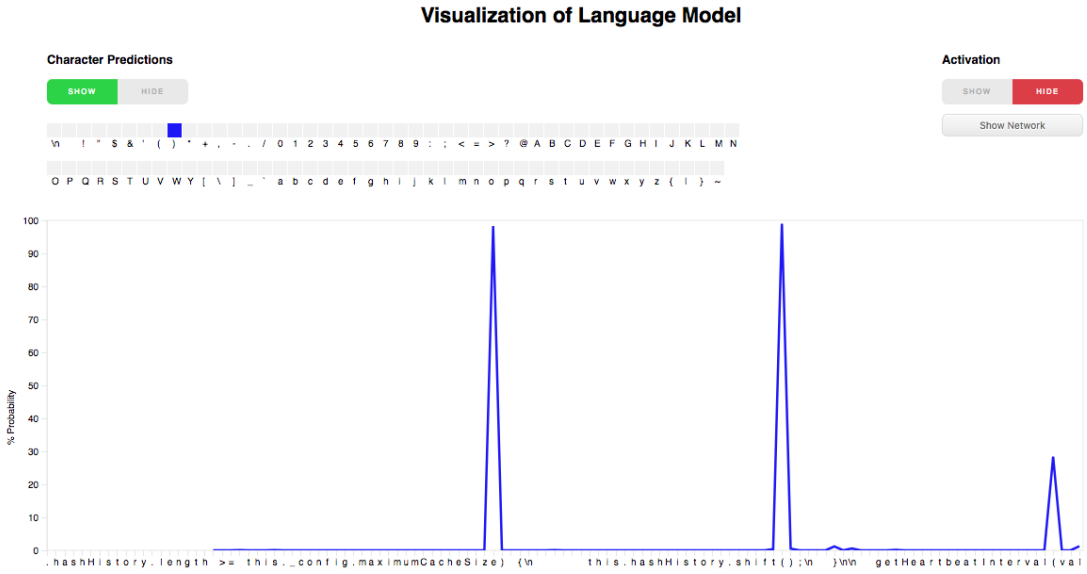


Figure 7.2: A screenshot of the visualization where the predictions are activated.

7.1.2 Features

As described in the section before, the visualization contains several features.

- *Character Selection:* The user has the possibility to choose one or more (but a maximum of 4) characters from the dictionary. Those probabilities are shown as a

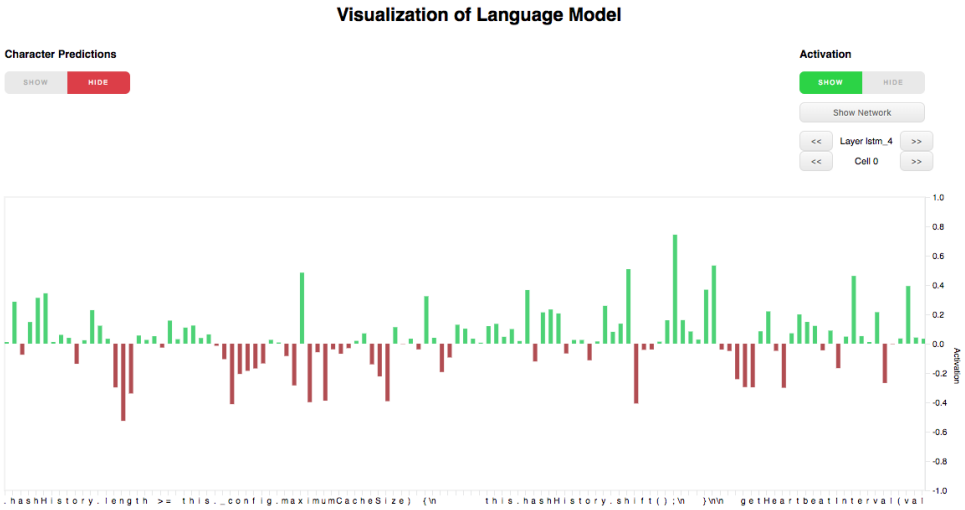


Figure 7.3: Screenshot of the visualization where the activation selection is activated.

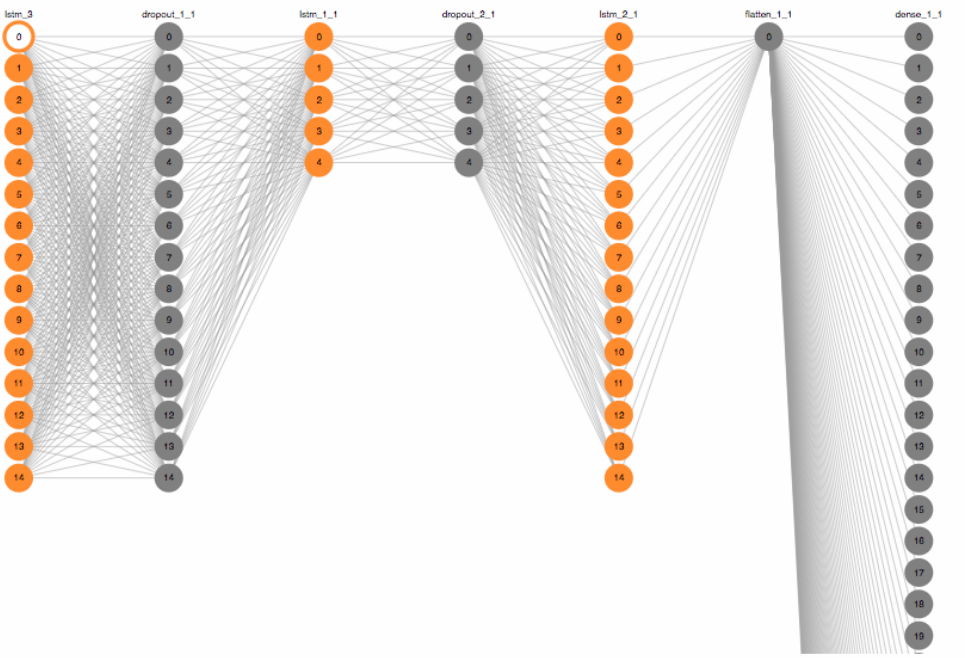


Figure 7.4: A screenshot of the network structure visualization. The selected hidden node is highlighted with a orange border.

line in the chart. Therefore the user can see how the probability is changing over time.

- *Activation Selection:* The user is able to select one of the hidden units out of an LSTM layer in the selection menu at the upper right corner. The activation is shown in the chart as positive (green) or negative (red) bars for each time step.

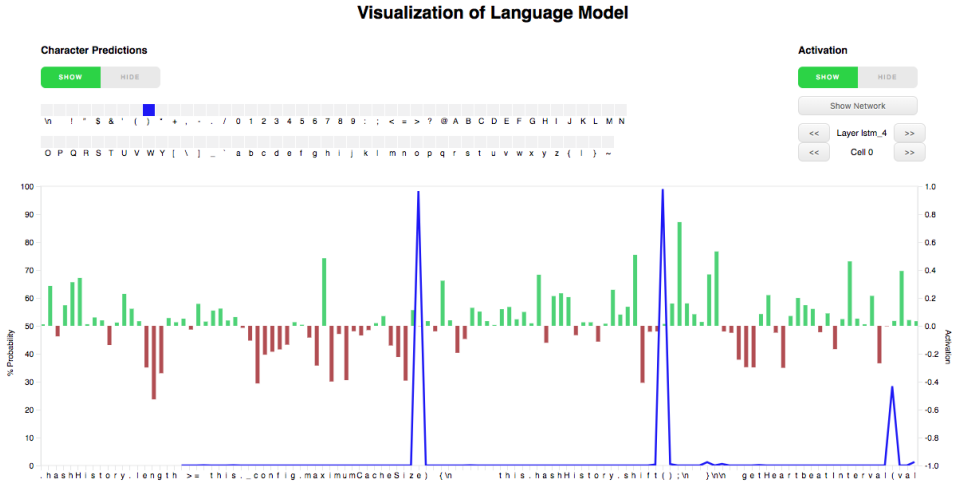


Figure 7.5: A screenshot of visualization where all features are activated and the two graphs are superimposed.

- *Feature Selection ON/OFF:* Additionally the user has the possibility to activate or deactivate the two features described above. This minimizes the complexity of the chart and enables the user to focus on a specific problem or solution.
- *NN Structure:* As the selection of the activation neurons are a bit confusing at first glance, a network structure visualization (see Figure 7.4) is provided. There, the user can see what the NN looks like. The whole network with the LSTM layers, the dropout layers, the flatten layer as well as the dense layer are drawn as circles on the screen. Only the LSTM layer circles are selectable and they trigger an update of the visualization.

7.2 Implementation

The following section is providing some details about the implementation and data structuring for the visualization.

The main steps for creating the chart are the following: First of all, the JSON (see Section 7.2.1), generated during the the text generation of the languge models, has to be loaded into the application. This JSON is then parsed into a structure needed for the visualization (Section 7.2.2). Afterwards

1. the SVG for charts,
2. the mouse tracker for the helper lines and
3. the objects for the tooltips, which shows the values of the lines or bars

have to be added to the HTML page. Furthermore, the objects for the activations (Section 7.2.3), the objects for the predictions (Section 7.2.4) as well as the objects for the network structure (Section 7.2.5) are created. Finally, the chart and the network structure can be drawn.

7.2.1 Data Structure

The JSON Object, created and saved by the model, has a fairly complicated structure:

```
1 "sequences": [
2     { // for each generated character (time)
3         "x": "if(",
4         "y": "t",
5         "act": [
6             { // for each layer
7                 "layer_name": "lstm_3/transpose_1:0",
8                 "activation_map": [
9                     [
10                        [ // Input Character "i"
11                            -0.021723484620451927,  // Hidden Node activation
12                            ...
13                        ],
14                        [ // Input Character "f"
15                            0.0015558326849713922,
16                            ...
17                        ],
18                        [ // Input Character "("
19                            -0.02718503028154373,
20                            ...
21                        ]
22                    ]
23                ]
24            },
25            { ... }
26        ],
27        "pred": [
28            { // for each character in the dictionary (sum up to 1)
29                "char": " ",
30                "prob": 0.20147760212421417
31            },
32            {...}
33        ]
34    },
35    { ... }
36 ]
```

The `sequences` array contains one object for each generated character. This object contains four values/objects: `x`, `y`, `act` and `pred`. The `x` represents the input sequence, whereas the `y` stands for the output character. The output character is the one prediction that is selected as the next character (described in Chapter 6). The `act` object contains the activation values (`activation_map`) for each layer. The `activation_map` is structured in a nested way. So, for each input character the activation of every hidden unit is saved, as shown in Line 10 to 24. When generating 100 characters with a model of three times 700 hidden nodes the JSON file has a capacity of 200 MB to 500 MB, which emphazises the amount of data saved for the visualization.

7.2.2 Data Preparing

The before described JSON is loaded into the application via D3 (described in Section 5.2). The response is parsed and restructured as objects within an array:

```
function parseData(data) {
    dict = data.dictionaries
    data = data.sequences
    var arr = []

    for(var t=0; t<data.length; t++) {
        time = data[t]
        x = time.x
        y = time.y
        act = time.act
        pred = time.pred
        arr.push({
            t: t,
            x: x,
            y: y,
            act: act,
            pred: pred
        })
    }
    return arr
}
```

The structure of the data is very complicated, as the example before has demonstrated. Therefore, the data has to be restructured for each chart type (bar chart and line chart). The main steps of the chart creation are described in the next sections.

7.2.3 Activation

For the activation chart the data has to be structured in a way that the activations for each cell in a layer are selectable, as the following code shows:

```
layers_cells.push({
    id: i,
    layer: layer_nr,
    layer_name: act_obj.layer_name.split("/")[0],
    cell: cell_nr,
    values: getActivationValues(cell_nr, layer_nr),
    visible: (curr_layer == layer_nr && curr_cell == cell_nr)
})
```

This object is used for creating bars for each cell in a layer. All the bar chart elements are created at the initial loading. This makes the initial start of the application very slow, but during the interaction, there are no performance problems.

7.2.4 Predictions

As for the activation chart, the data has to be restructured another time. Now, the probability at each time has to be saved for each character in the dictionary:

```
characters.push({
    name: char,
    id: key,
    values: getPredictionValues(char),
    color_obj: null,
    visible: (char == firstSelectedChar)
})
```

The generated object is then used for creating the lines for each character in the dictionary. As before, those lines are all created at the first loading and only their visibility is changed during runtime.

7.2.5 NN Structure

Finally, the last object that has to be created is for the network. As seen in Figure 7.4 the network contains circles as nodes and lines as connection between the nodes. Furthermore, the caption has to be added for the layers. An object for each layer is created with the attributes: layer number, layer name and cell size. It is necessary to distinguish between flatten, output layer and other layers because of their hidden node length:

```
nodes.forEach(function(act_obj, layer_nr){
    // FLATTEN
    if(layer_nr == nodes.length-2){
        network.push({
            layer: layer_nr,
            layer_name: act_obj.layer_name.split("/")[0],
            cellsize: 1
        })
    }
    //OUTPUT
    else if(layer_nr == nodes.length-1){
        network.push({
            layer: layer_nr,
            layer_name: act_obj.layer_name.split("/")[0],
            cellsize: act_obj.activation_map[0].length
        })
    }
    // OTHERS
    else {
        network.push({
            layer: layer_nr,
            layer_name: act_obj.layer_name.split("/")[0],
            cellsize: act_obj.activation_map[0][0].length
        })
    }
})
```

In this special case, the flatten layer is visualized as one neuron for representing the one dimension.

7.3 Design Decisions

The main decisions are justified as follows:

- *Why use D3?* The main focus of the project is to create an interactive visualization with a big amount of data, as already demonstrated in Section 7.2.1. On the one hand, D3 is widely used for the implementation of interactive data visualizations in the web and on the other hand it is able to handle big data. For this reason, the library is a good solution for this visualization as it solves all the necessary requirements.

- *How the chart design is chosen?* Based on the related work (Chapter 3), especially the research of Karpathy et al. [1] and Strobelt et al. [44], the decision for using the *probability* distribution and the *activations* of the hidden units was obvious. These attributes are the key values for the analysis. Therefore the idea was to visualize those two values in one chart to compare them and find connections between the prediction of a character and the activation of some hidden units. This makes it possible to demonstrate and prove that hidden units are specifying on different parts (patterns) of the text (e.g., brackets, text length, etc.).

7.4 Challenges

This section provides an overview of the challenges encountered during the design and implementation process of the visualization.

- *Data visualization:* The first big challenge was to find a good visualization for a neural network explanation. While reading about related work the idea was to visualize the prediction probability in comparison with the activation of the neurons. Afterward a challenge was the implementation of it. First of all, the decision of the design elements (line graph, bar chart). It was tricky with D3 to place them together into one graph with one x-axis and two y-axes because the two chart types had different scales. But in the end it works very well and also the feature with the on-off of the different charts works very good.
- *Data preparing and parsing:* As described in Section 7.2.1 the necessary data used for the visualization is a very large JSON file and the structure of the JSON is rarely complicated. Therefore, the implementation of the two charts was challenging, as the data had to be restructured several times for the creation of the charts. First, the structure have to be analysed and then the object has to be run through several loops to create new objects based on another structure: Once to be able to select a interesting character and show his probability and secondly to select a desired node within a specific layer to show his activation.
- *Visualize Network Structure:* The implementation of the network structure, shown in Figure 7.4, was difficult. A very challenging task was the calculation of the position of the nodes (circles) and the connections (lines) in an automatic operation. Another difficult part was the styling of these elements. Showing more than 100 hidden units on one browser page is very hard to implement as the screen is to small and scrolling should be avoided.
- *Visualization Performance:* One last big challenge was the performance of such a large visualization. The main problem is the huge amount of data and elements that are added to the SVG, especially for the network structure visualization. But, as the focus is not on the applications performance, the solution was to create all elements at the initial loading of the page. Even if this makes the loading of the application very slow.

Chapter 8

Results: Explanation Patterns

As described in Chapter 5, the last step in the explanation method is to load the predicted text with the concerning data (the JSON file generated by the model seen in Section 6.2) into the D3 application. This visualizes the available predictions as well as the activation in a interactive way. The application can then be used to find some patterns in the generated sequences for explaining the language model.

This chapter contains some of the patterns that are found during this process. For each of the four trained models, described in Section 6.4, different patterns were searched for. They are going to be explained to prove the efficient functioning of the visualization and of course to explain how the models learn.

For finding the explanation patterns, different sequences are predicted and loaded into the application. Then different settings are tested to find connections between the predictions of characters or also the activation of the neurons. First, it has to be said that the models are not that accurate and therefore the patterns are not always 100% unequivocally, which means that the neurons are activated during a special character but also in single cases also for other characters. But it must be differentiated between the activation level 0.3 or over 0.5. Thus, it must be considered that the activation under 0.3 is ignored most of the time.

8.1 Model 1: JavaScript (small)

The first language model, described in Section 6.4, is trained with JavaScript code. For this model, some really interesting explanation patterns were found.

For the first pattern, the activation section is deactivated and the two characters (closing bracket and open curly bracket) are selected in the character prediction section. This setting resulted in the graph shown in Figure 8.1. It contains a pattern which shows a relationship between normal brackets and curly brackets. This relationship can be identified by a raise of the probability for an open curly bracket (“{”) after a normal bracket was closed (“)”), as the line graph demonstrates. This absolutely resembles the structure of JavaScript and is therefore a good indicator that the model learns the right structure. Another graph (Figure 8.2) shows the correlation between opening and closing brackets. In the graph, there can be seen that the probability of the closing bracket is raising directly after the open bracket is predicted. Furthermore, as the closing bracket

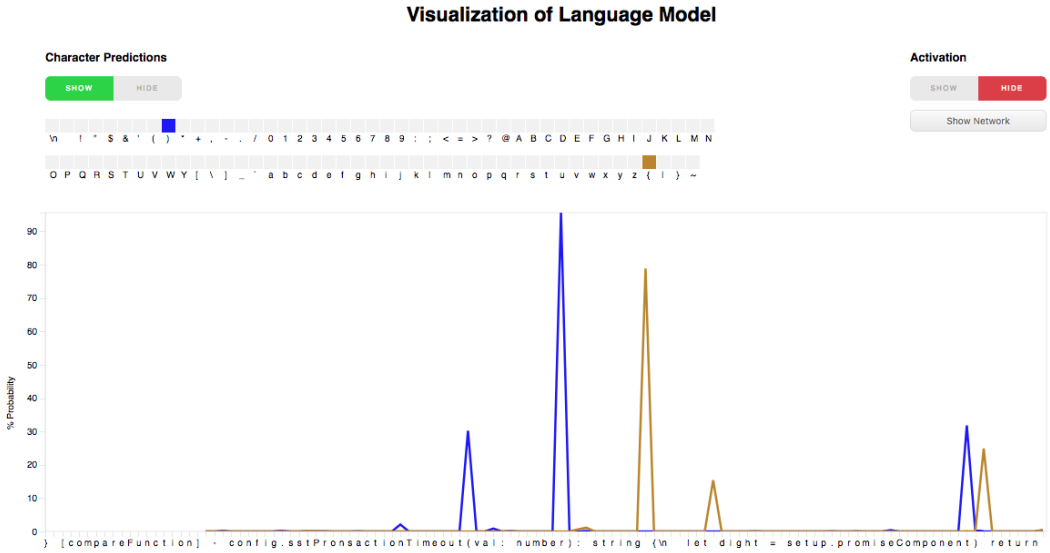


Figure 8.1: Pattern for opening curly brackets.

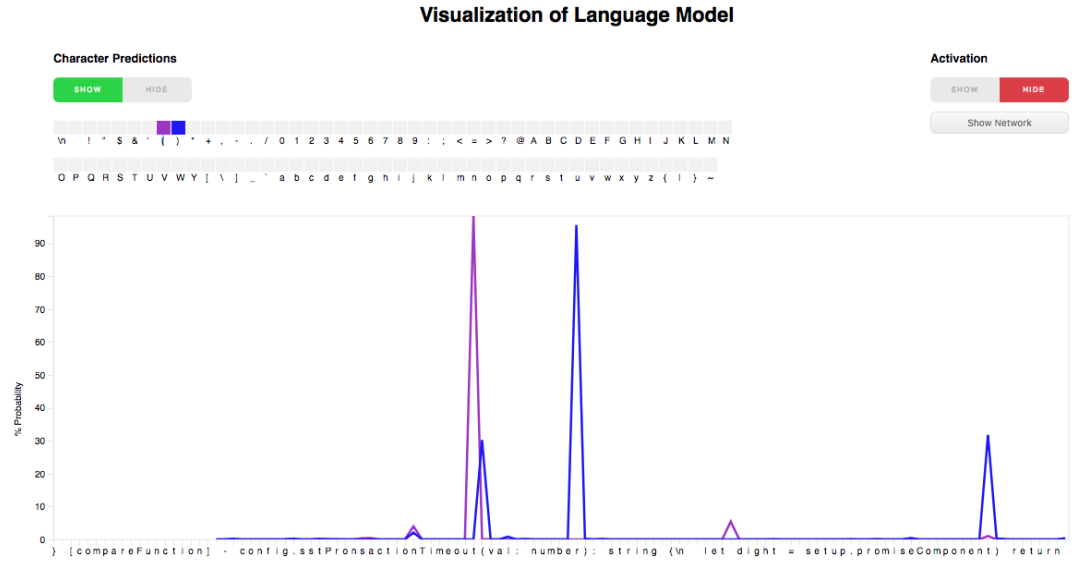


Figure 8.2: Pattern for opening and closing bracket.

was not predicted directly after the open bracket, the model predicts it another time not until a word is ended. Therefore there a correlation between brackets and the end of a word can be identified. The Figure 8.3 as well as the Figure 8.4 verifies this correlation between the end of a word and the probability of a bracket. The Figure 8.3 shows that the probability of an open square bracket does not raise until a words' end is reached. This can also be identified in Figure 8.4 where the probability of a closing bracket is increasing after a word ends.

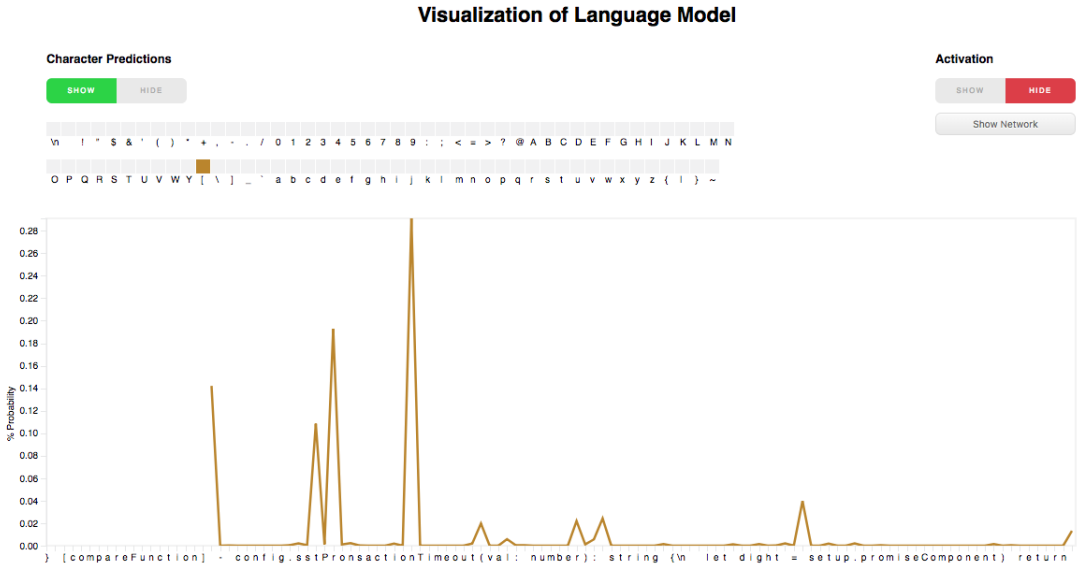


Figure 8.3: Pattern for opening square brackets.

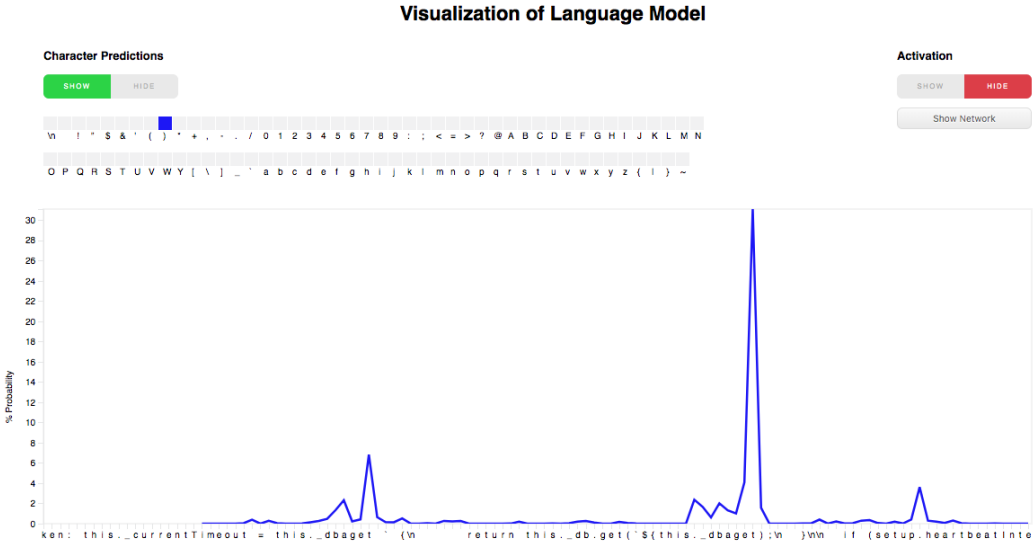


Figure 8.4: Pattern for a connection between end of word and brackets probability.

Finally, another pattern was found, which also shows the connection between the predictions and the activation of the neurons. Therefore, as described in Section 7.1.2, both features are activated in the application to show the character section and the activation section. First, one character was chosen in the character prediction section. Secondly, a neuron was chosen in the activation section. While clicking through the different neurons and layers, it turned out that many of the neurons do not specify on

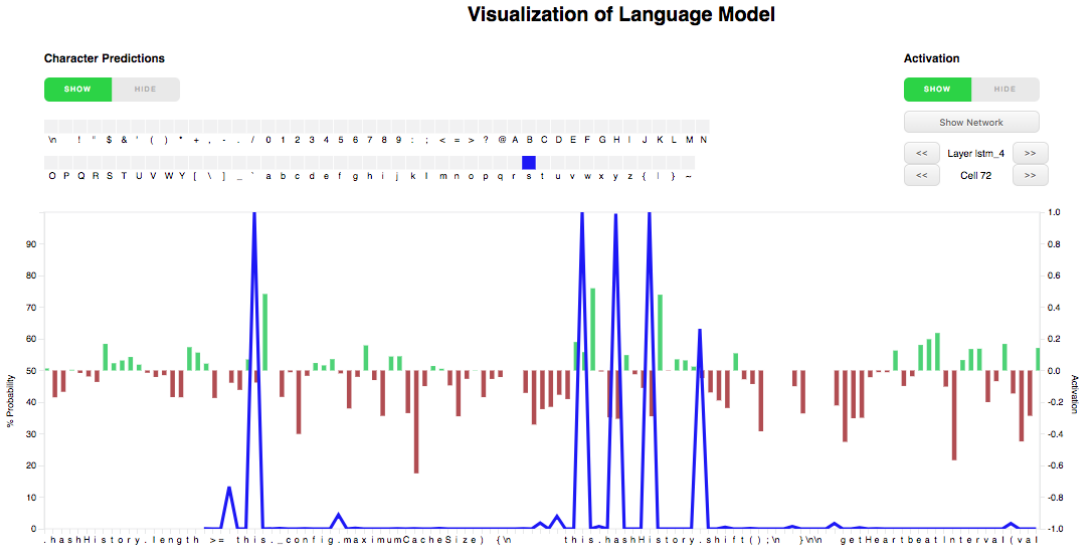


Figure 8.5: Pattern for the activation of a hidden unit when a “s” is predicted.

a specific pattern, character or anything else. However, there exist other neurons who seem to have a direct influence on the predictions and activate at the same time as a single character is predicted. So, there are some neurons who are directly related to the prediction of a character. The graph in Figure 8.5 shows, for example, that the hidden unit number 72 in the first LSTM layer is maybe directly connected to the prediction of the character “s”. The neuron gets activated when the characters’ probability raises. The same effect can be detected in Figure 8.6, where the hidden unit number 87 is activated when a “t” is predicted or when the probability of the “t” raises. This means that the neuron listens to the prediction of a single character or even causes the prediction of the character.

These examples demonstrate the main idea behind the visualization and design. Furthermore, they show in which way an LSTM learns to predict characters after some input sequences. Especially the example with the brackets proves that the model can detect long-term dependencies. Additionally, it has to be said that the patterns above are just a few small examples of what can be explained and how the visualization can be used. There are many more patterns found to explain this model’s behaviour.

8.2 Model 2: JavaScript (large)

After the first model, trained with JavaScript code, another model is trained with a bigger input file. The predictions of the model seem to be more accurate than the ones of the first model. The same result can be seen in the patterns as there are more specific patterns found. A specific pattern is, for instance, the activation of a neuron during a word sequence. Some of those patterns that were found are described below.

Figure 8.7 shows the activation of a neuron in combination with the probability of a new line (“\n”). The resulting graph shows that the probability of the new line is very

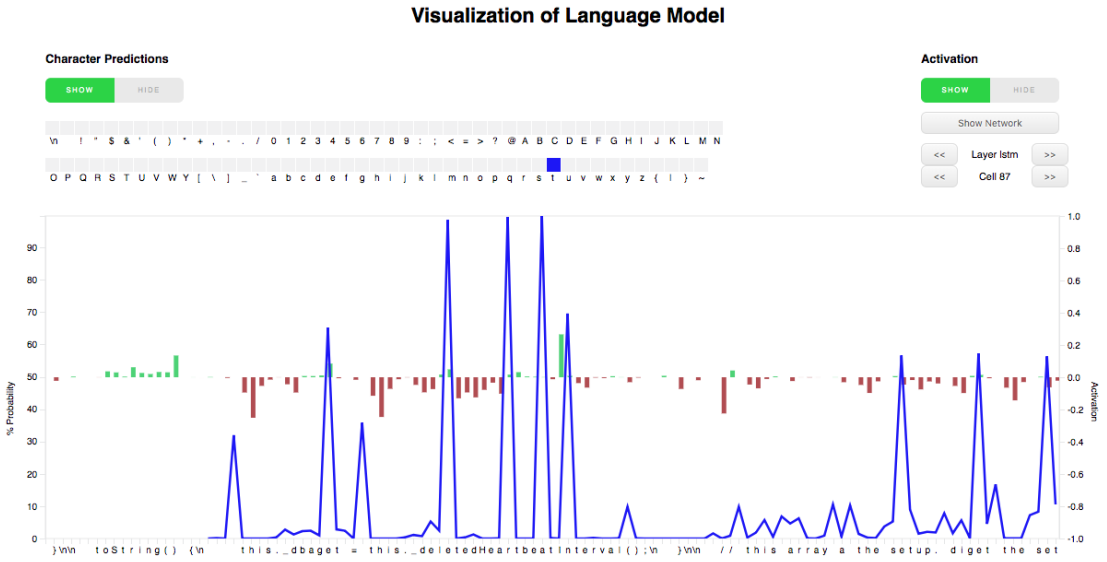


Figure 8.6: Pattern for the activation of a hidden unit when a "t" is predicted.

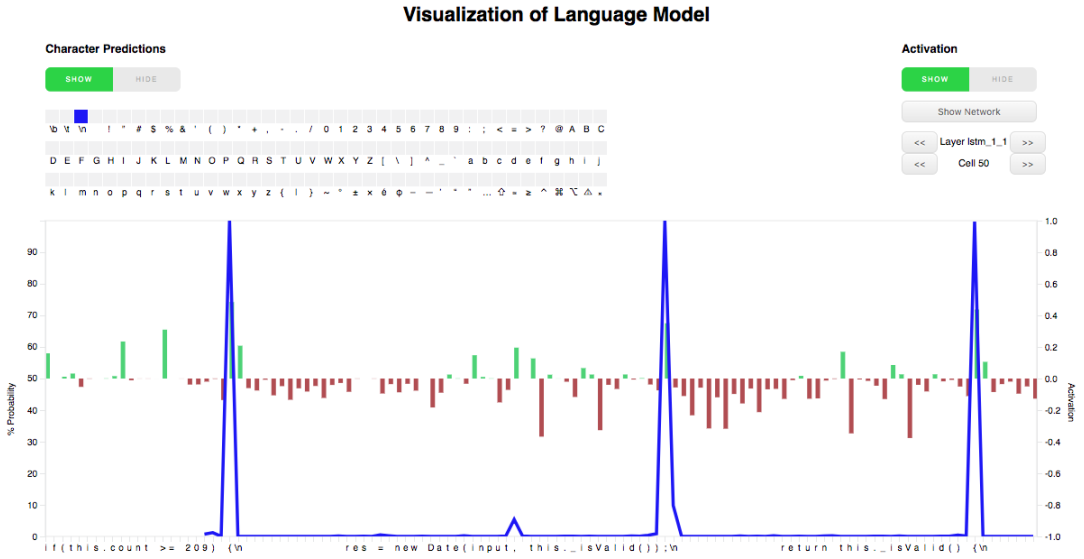


Figure 8.7: The neuron is active when a "\n" is predicted.

high at the same time when the activation of the selected neuron is high. This indicates that there exist neurons that specify on the predictions of new lines. Moreover, there is also a neuron which gets inactive when a new line is predicted. This pattern is shown in Figure 8.8 where the activation of the neuron is nearly minus one when a "\n" is predicted.

Another pattern shows the connection between space character and the activation of a neuron. There is, for example, one neuron that gets inactive when several space

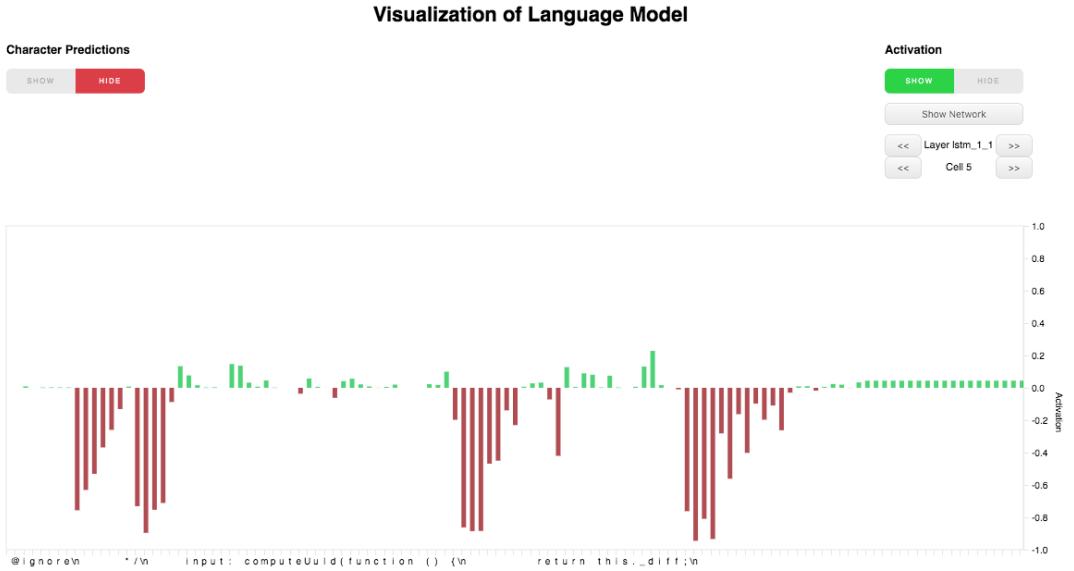


Figure 8.8: The neuron is inactive after a "\n" is predicted.

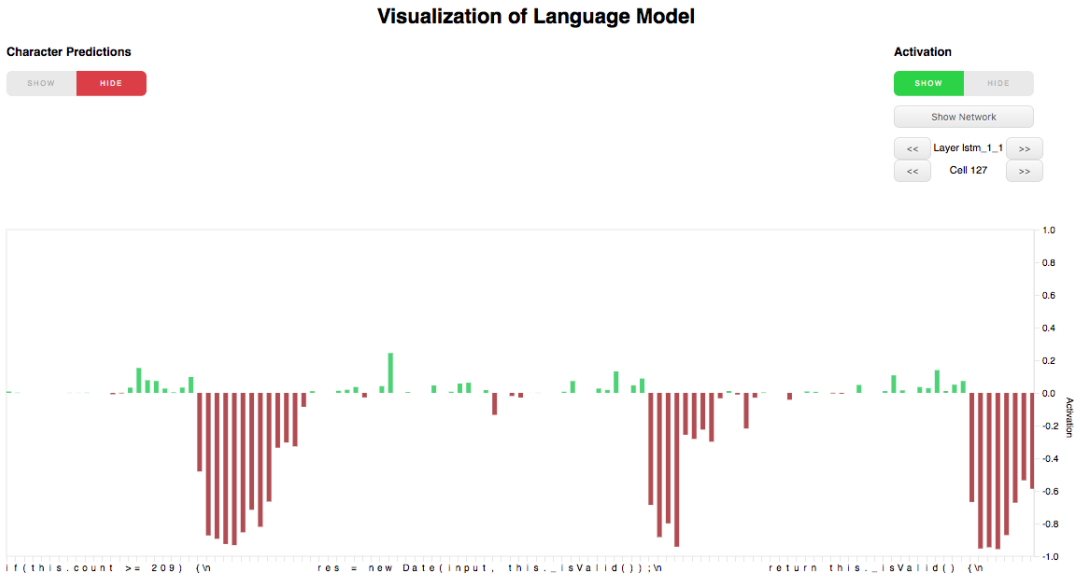


Figure 8.9: The neuron gets inactive when several space characters are predicted.

character are predicted. This pattern is shown in Figure 8.9. In addition, there is a neuron which gets activated by the predictions of space character. This is shown in the graph in Figure 8.10 where the neuron 176 is selected and which is only active during the space characters. Thus, this proves that neurons can specialize in different characters and also can be uninterested in a character.

Another pattern, that was already mentioned before, is the activation of neurons

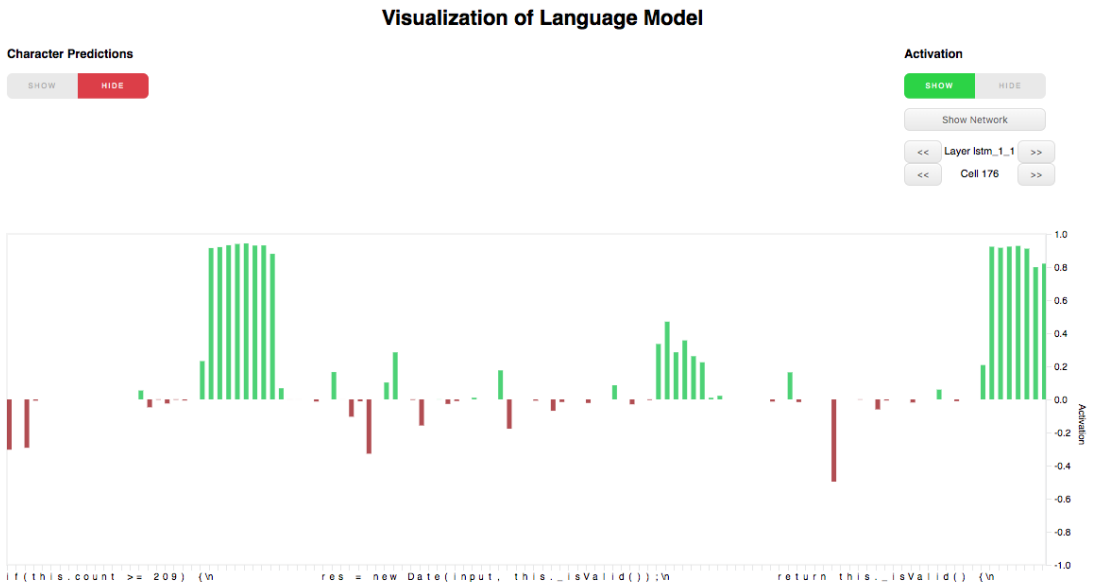


Figure 8.10: The neuron gets active when several space characters are predicted.

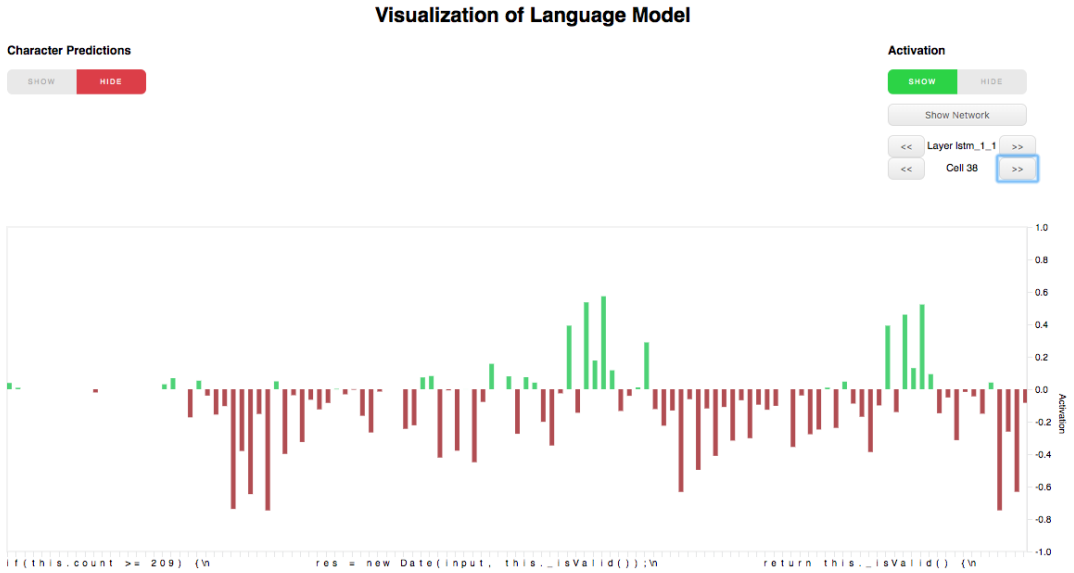


Figure 8.11: The neuron is active when the word “valid” is predicted.

during a special sequence of characters - a word. The graph in Figure 8.11 shows the activation of a special neuron. The green bars indicate that the neuron is always active when the word “valid” is predicted. Thus, neurons specify not only on single characters but also on sequences.

A last pattern remembers on Figure 8.3 as well as Figure 8.4 of the first model. The graph in Figure 8.12 shows the probability of a closing bracket during the time.

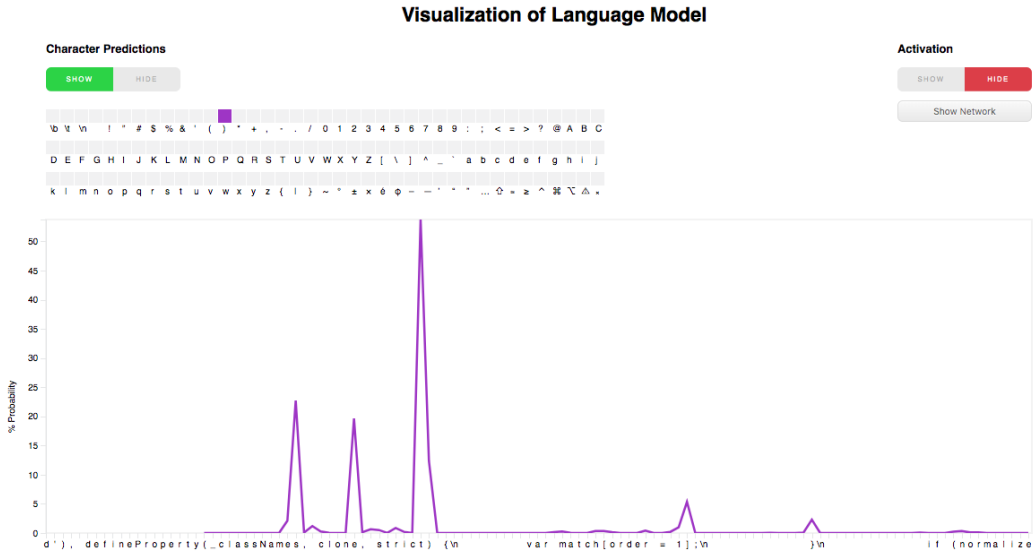


Figure 8.12: The neuron predicts closing brackets after a word ends.

It indicates a connection between words and closing brackets. In the graph, it can be seen that the likelihood of a closing bracket always increases after the completion of a word and then decreases when the next word starts. This makes it seem as if the model knows that there is no bracket in the middle of a word.

To conclude, these results show that there much more interesting patterns can be found when the model's predictions are better. A better model must have more neurons that specialize in different patterns to predict more accurately.

8.3 Model 3: Harry Potter

For the one model, that was trained with natural language, some interesting explanation patterns were found. But, for this model, the analysis was a bit more difficult, as a natural language text has not that much structure which can be looked at. Nevertheless, some interesting neuron activation's were found during the analyzation process.

The graph in Figure 8.13 shows the activation of the neuron number 43. It appears that the neuron gets more active when the character "a" within the word "Harry" is predicted. This is the reason why there is no character chosen in the character prediction section. The neuron is not always active when an "a" is predicted but only with a condition. This can be proven by a high activation (the large green bar chart) at the same time when the x-axis shows the character "a" in "Harry". This can probably mean that the neuron is responsible for the prediction of an "a" after the "H".

Furthermore, there exists another neuron that specifies on the prediction of words. The pattern, shown in Figure 8.14, indicates that the selected neuron is more active during the word "know". This can be identified by the raise of activation over 0.5 when the word is predicted.

Another interesting behaviour of the model can be seen in the graph in Figure 8.15.

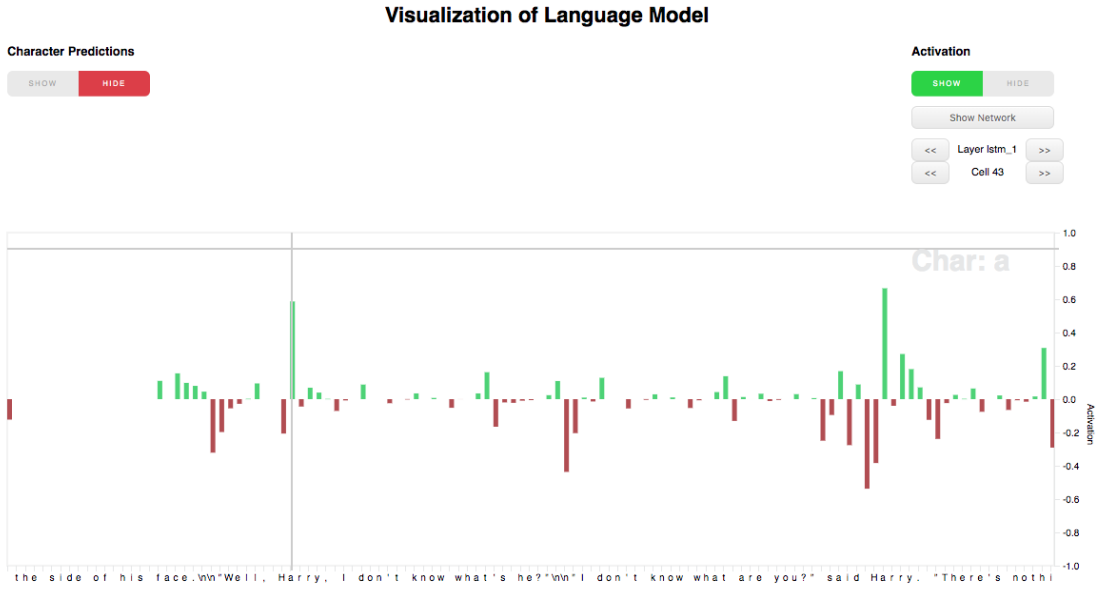


Figure 8.13: A neuron is active at the same time when the "a" is predicted within the word "Harry".

It shows that there is a neuron which listens to or feels responsible for the prediction of new lines. For this setting, the activation of the neuron 218 is shown and the character "\n" is selected in the prediction section. The graph then shows that the neuron is active at the same time when the "\n" is predicted.

The last pattern is shown in Figure 8.16. Even if the activation of this neuron looks very random at first glance, it specifies on an important part: the separations of words. The graph shows that this neuron is active during the prediction of a word and gets inactive when the space character is predicted. This means that it takes part in the prediction of the words and maybe another neuron is responsible for the prediction of the space characters, as already seen in the models before.

Hence, these patterns prove that the different neurons specify on different parts of the text generation and all the neurons together can yield such impressive results. With high probability, a model trained with more data will have much more neurons that specify on a pattern and therefore the model gets much better.

8.4 Model 4: LaTeX

For the last model, which was trained with LaTeX code, the same process was run through to find some explanations for the models' predictions. During this process, several patterns were found and the patterns described below are the ones most interesting and meaningful.

The first pattern, which was found for the LaTeX model, demonstrates, that a neuron is reacting when a "tabulator" ("\t") is predicted. The two Figures, 8.17 and 8.18, show that the character "\t" is selected in the character prediction section and a single neuron

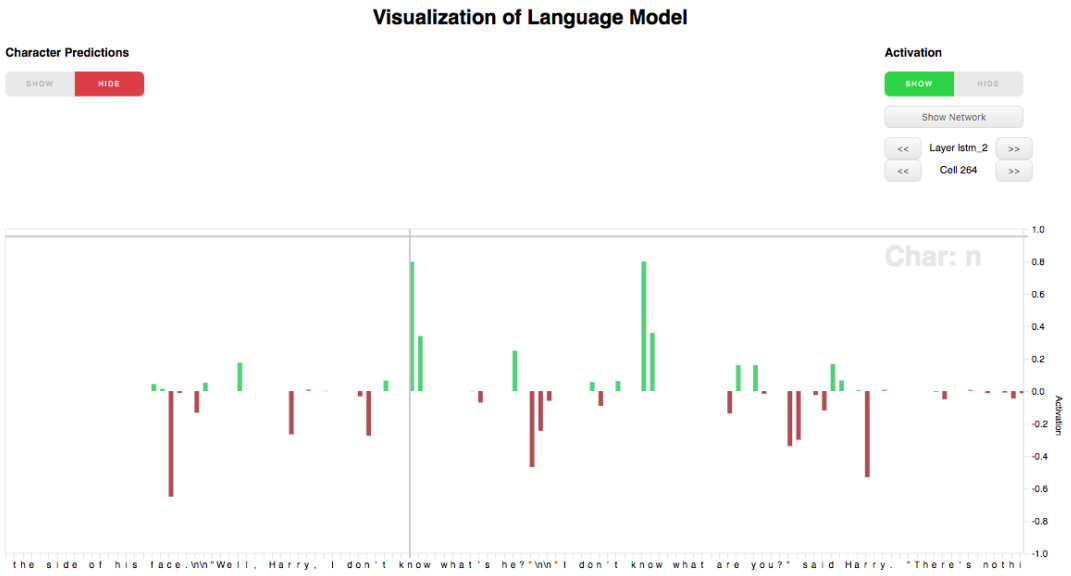


Figure 8.14: The graph shows that the neuron is more active when the word "know" is predicted.

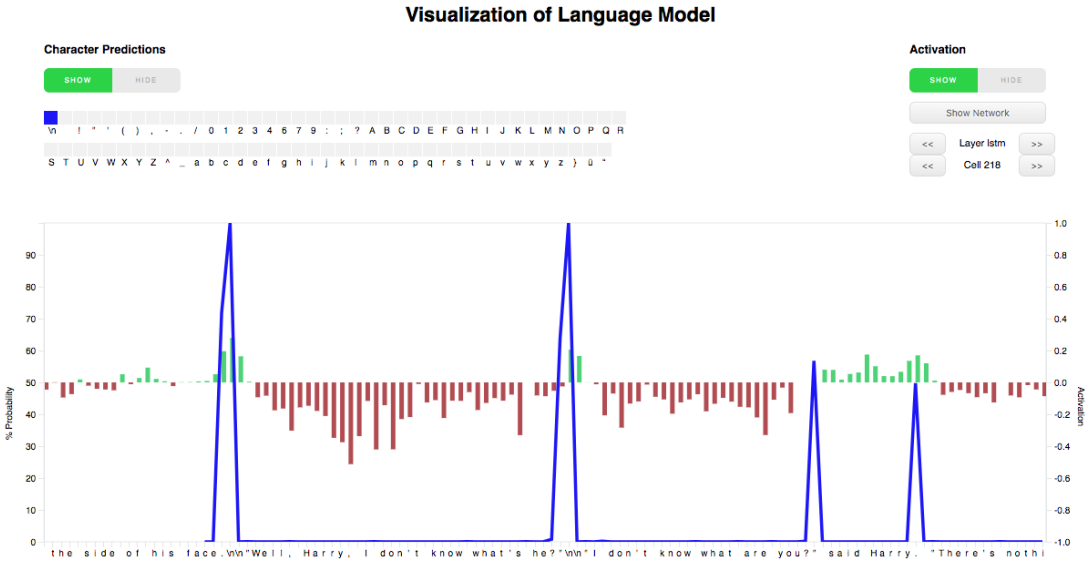


Figure 8.15: The graph shows that the neuron is active when the probability of a new line increases.

is selected in the activation section. Those two graphs together can indicate connections between the prediction of a character and the activation of the neuron. For instance, the graph in Figure 8.17 shows that a neuron gets inactive when a "\t" is predicted. Whereas another neuron does exactly the opposite, as Figure 8.18 shows. There the

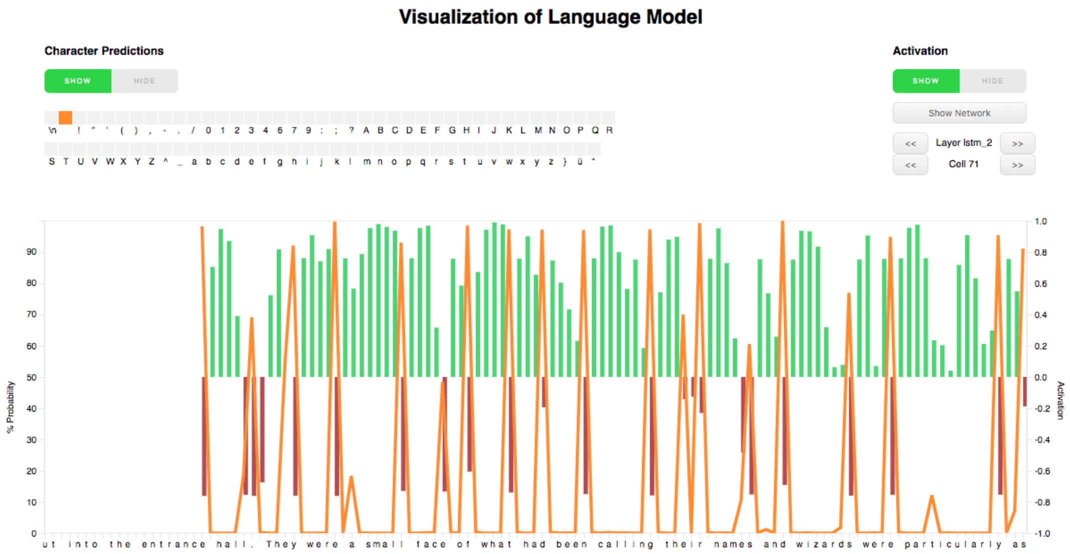


Figure 8.16: The graph shows that the neuron is active within a word and gets inactive immediately when the space character is predicted.

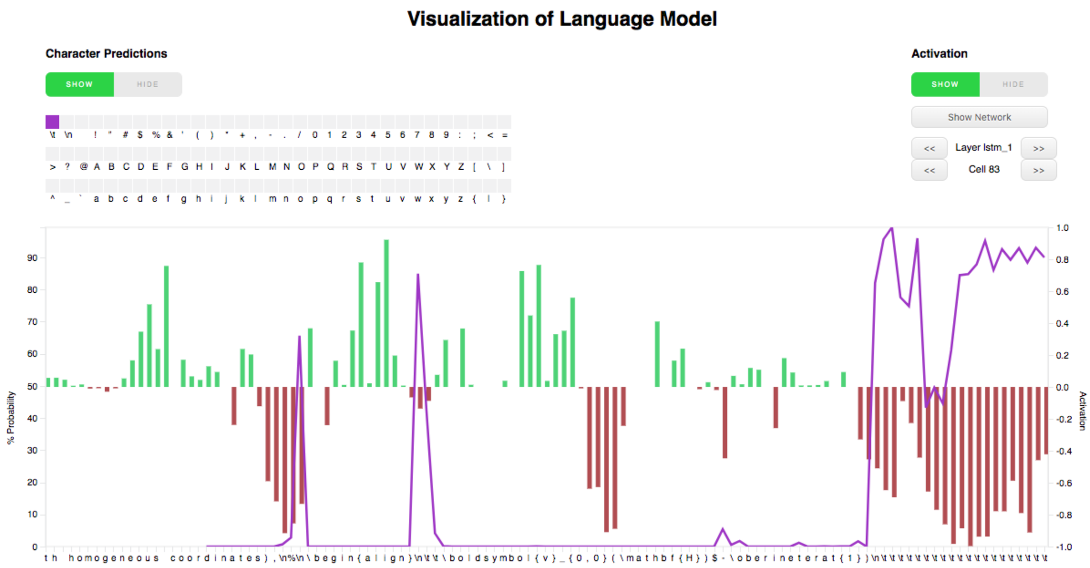


Figure 8.17: A neuron gets inactive when "\t" is predicted.

neuron always gets inactive when a "\t" is predicted. This can be clearly identified by the violet line. As the line goes up at the same time as the activation bar goes down.

Another pattern that explains the behaviour of this language model is shown in Figure 8.19. In this case, the probability of a backslash ("\") directly influences the activation of the neuron. This can be recognized due to the high activation at the same time as the probability of the "\" is increasing. This could mean that the neuron number

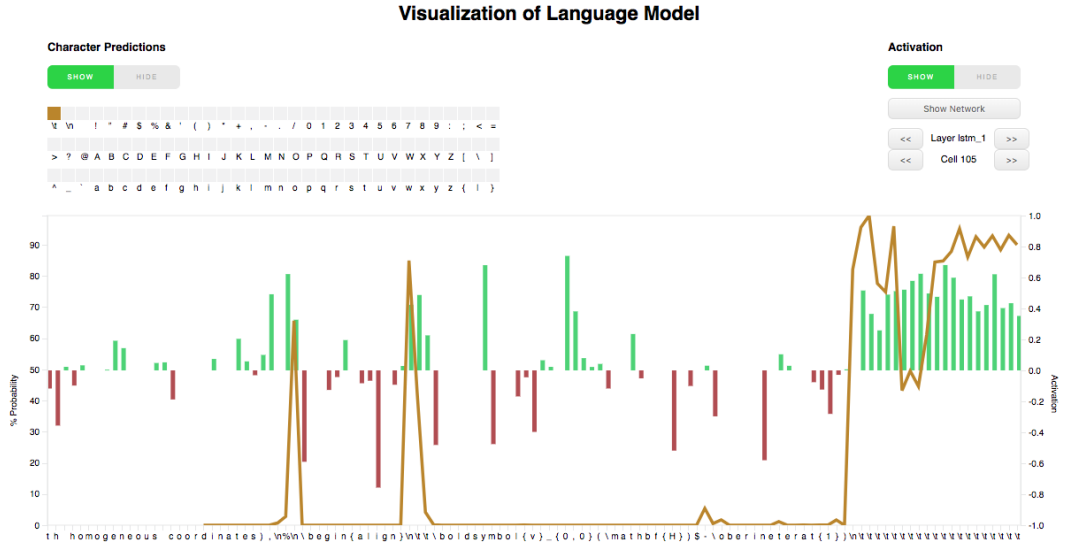


Figure 8.18: A neuron gets active when "\t" is predicted.

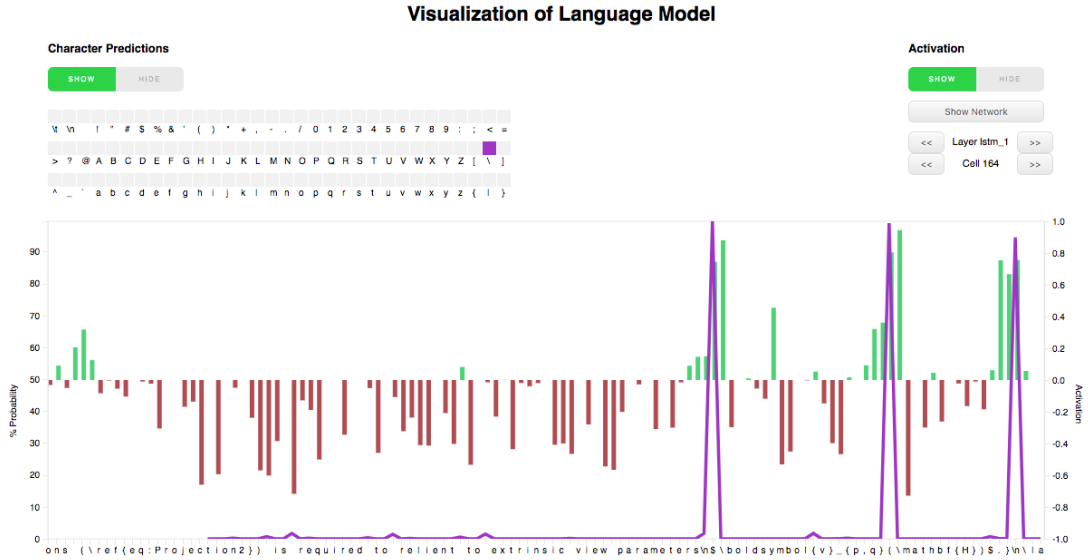


Figure 8.19: A neuron gets active after a "\" gets predicted.

164 in Layer "lstm_1" listens to the code segments of a LaTeX file and feels responsible for the predictions in these sequences. However, it could also mean that it is responsible for the predictions of the backslash itself.

In Figure 8.20 another pattern concerning brackets can be identified. This graph shows the probability of a closing bracket over time. It shows that a curly bracket is predicted after the word "boldsymbol" and therefore the probability of a closing curly bracket has increased marginally. Thus this proves that the prediction of an open curly

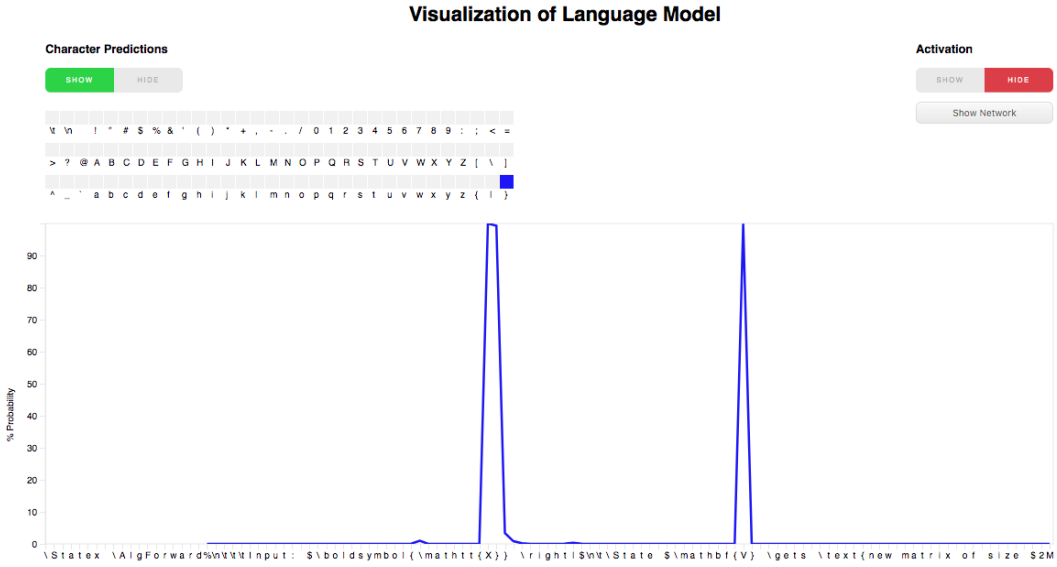


Figure 8.20: When predicting a open curly bracket the probability of a closing curly bracket immediately increases.

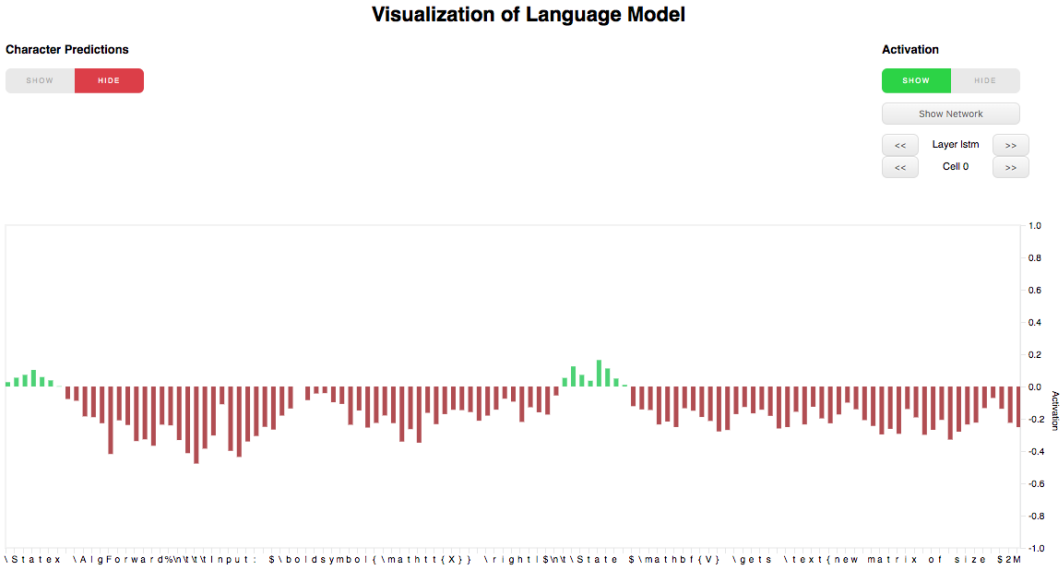


Figure 8.21: A single neuron listens to the prediction of the word “State”.

bracket influences the probability of a closing curly bracket. Furthermore, as it is not predicted at this point the model does not forget it but predicts it directly after the adjoining word is finished. This behaviour proves that the model knows that a bracket is never predicted in the middle of a word and that the open bracket requires a closing bracket.

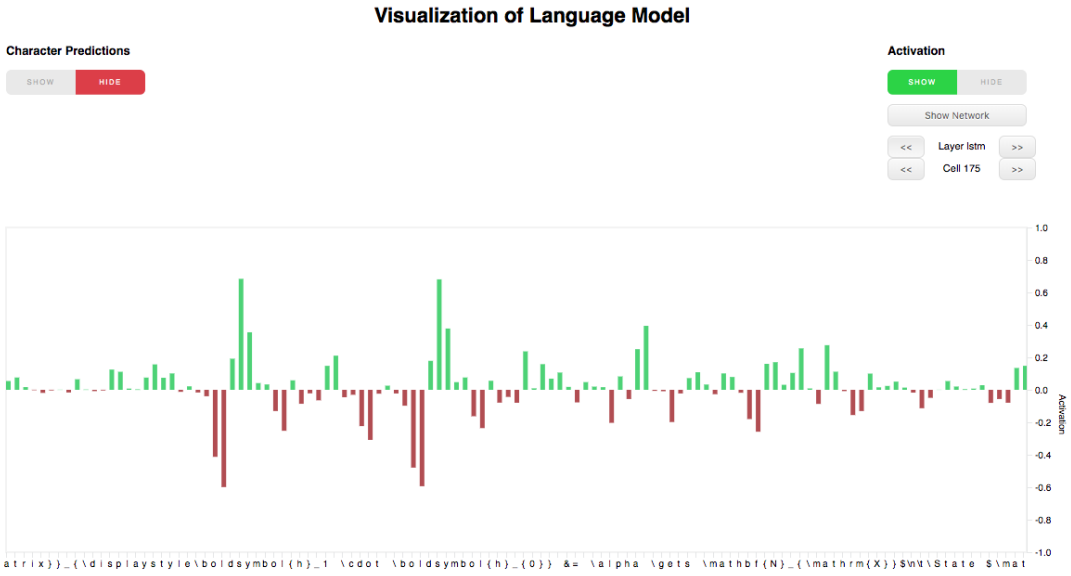


Figure 8.22: The neuron is inactive when the word “bold” is predicted and get active when the word “symbol” is predicted.

Figure 8.21 shows the activation of one single neuron in the first layer. From this graph, an interesting pattern of the model can be identified. This neuron is only active when the word “State” is predicted. The two green rashes in the graph prove this. Thus this maybe indicates that this neuron is related to the prediction of the word.

There are many different neurons that seem to listen to a single word as the one in Figure 8.21 does. There also exist several neurons that are all listening to the word “symbol”, as the one shown in Figure 8.22 does. This graph only shows the activation of a neuron and not the predictions as it does not listen to one single character but a sequence of characters. Furthermore, it can be identified that the neuron is always inactive when the word “bold” is predicted. The largest red bar proves this.

The Figure 8.23 shows the last interesting pattern. Unlike before this neuron is listening to single characters. The graph in Figure 8.23 shows that the neuron 193 in the first LSTM layer gets always active or is active when an “s” is predicted. Thus this probably means that the neuron causes the prediction or what character should be predicted after the “s”.

All in all, the model seems to be really trustworthy as the decisions are based on different patterns and were not randomly chosen.

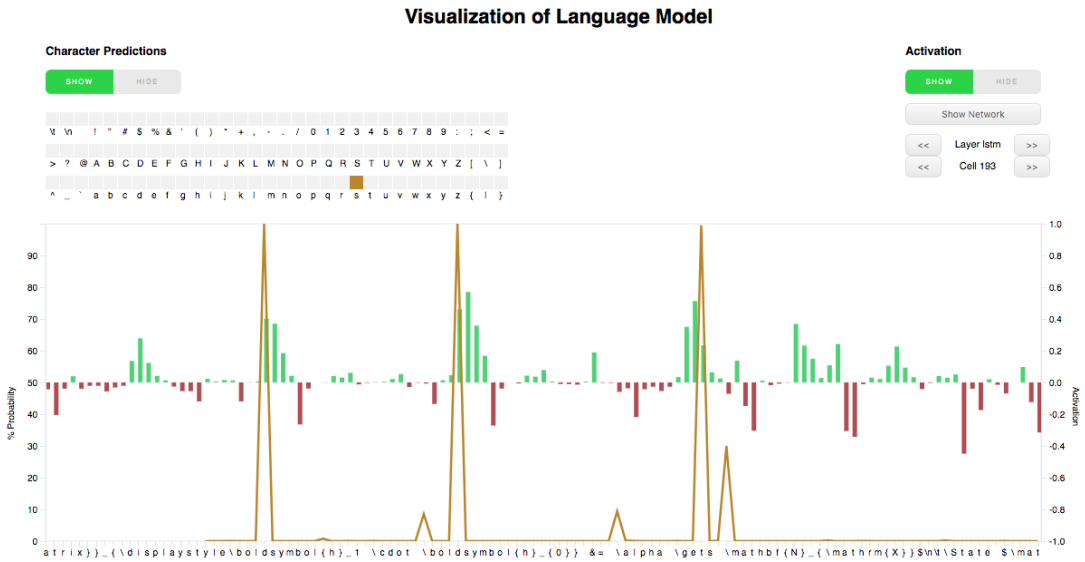


Figure 8.23: The neuron activates after the prediction of the character “s”.

Chapter 9

Conclusion

Explainability is a key issue in many problems. Using a black box as a machine learning model is no longer an option. Not only in the medical domain but also in other fields or for other critical applications. As shown in the first chapters the interpretation of a machine learning model is very useful. The explainability and the model interpretations allow the users and/or developers to:

- verify the predictions,
- improve and debug the model,
- rely on the models' predictions,
- understand what the system is doing,

and much more.

The goal of the thesis is to evaluate what a visualization of an LSTM language model can tell about the model's self-learned algorithm. Therefore the developed visualization (Chapter 7) is used to find patterns that explain the system's predictions and how they make their decisions. It enables to find patterns and allows to understand what the language model is doing and how it is doing.

The patterns, shown in Chapter 8, describe in which way the different language models can predict sequences that remember on the input data. All the different patterns of the four models are similar in some way. The patterns explain the system's predictions and the behaviour of the neurons. Thus, it can be concluded how a language model is making the predictions. The results prove that the models' predictions are not random but follow different patterns. Generally, there are single neurons concentrating on special characters or sequences. Furthermore, in all four models, patterns are found where the characters predictions are dependent on single input characters. For instance, closing brackets are directly connected to the prediction of open brackets. Additionally, as seen in the patterns the neurons in a language model also concentrate on the prediction of empty spaces or new lines. This list can be extended much further, but it only shows the following: Each neuron specifies itself on certain tasks and together they manage to predict a text that is almost identical to the input data.

These results prove that the goal of the thesis is reached. The developed explanation method, which is an improvement of the method implemented by Karpathy et al. [1], performs very well. It is able to visualize the generated sequence and analyze the behaviour of different character-level language models during the predictions.

Finally, it has to be said that the ON/OFF for the selections (activation, predictions) proved as a very useful tool. It enabled to focus best on the goal and disable the unnecessary data at a specific point.

9.1 Future Prospect

In future work the explanation method can be improved in many different aspects: First of all, it is important to improve the model itself by using much more input data for the training. This can also lead to an improvement of the explanation patterns. A better model will probably have more neurons following more specific patterns: Not only characters or words but maybe also longer dependencies can be found. Secondly, the explanation method can be improved by implementing an automated feature recognition. Up to now, a user has to click through the characters or the neurons and search for interesting behaviours. This process can be simplified a lot by an algorithm that does this automatically. Finally, another helpful feature would be to add the visualization to the learning process of the model. At the moment, there are just predicted text outputted during the training to verify the quality of learning. But what if the training can be analyzed using the visualization too. For instance, this can help to find out if the patterns are more specific when the model is getting better.

All in all, these improvements together can lead to a much better explanation method and more ways to understand the inner workings of a language model.

Appendix A

DVD Contents

Format: DVD+RW, Single Layer, 4.7 GB

A.1 PDF

Path: /

Grafeneder2019.pdf . .	Thesis

A.2 Source Code

Path: /ProjectSourceCode

models.zip	Source Code of LSTM models
visualization.zip	Source Code of D3 application
readme.md	Instructions for usage

References

Literature

[1] A. Karpathy, J. Johnson, and L. Fei-Fei. "Visualizing and Understanding Recurrent Networks". In: *Proceedings of the International Conference on Learning Representations – ICLR '16*. 2016 (cit. on pp. 17, 18, 59, 75).

[2] L. Arras et al. "Explaining Predictions of Non-Linear Classifiers in NLP". In: *Proceedings of the 1st Workshop on Representation Learning for NLP – Rep4NLP '16*. 2016, pp. 1–7 (cit. on pp. 13, 14).

[3] L. Arras et al. "'What is Relevant in a Text Document?': An Interpretable Machine Learning Approach". *PLOS ONE* 12.8 (2016), pp. 1–23 (cit. on p. 13).

[4] B. Goodman and S. Flaxman. "European Union Regulations on Algorithmic Decision–Making and a 'Right to Explanation'". *AI Magazine* 38.3 (2016), pp. 50–57 (cit. on p. 3).

[5] S. Bach et al. "On Pixel-Wise Explanations for Non-Linear Classifier Decisions by Layer-Wise Relevance Propagation". *PLOS ONE* 10.7 (2015), pp. 1–46 (cit. on pp. 1, 9, 11–13).

[6] D. Baehrens et al. "How to Explain Individual Classification Decisions". *Journal of Machine Learning Research* 11 (2010), pp. 1803–1831 (cit. on pp. 5, 6).

[7] A. Binder et al. "Analyzing and Validating Neural Networks Predictions". In: *Proceedings of the Workshop on Visualization for Deep Learning at International Conference on Machine Learning – ICML '16*. 2016 (cit. on p. 12).

[8] M. Bojarski et al. "Explaining How a Deep Neural Network Trained with End-To-End Learning Steers a Car". 2017. URL: https://arxiv.org/pdf/1704.07911.pdf. Pre-published (cit. on p. 2).

[9] J. Brownlee. *Long Short-Term Memory Networks With Python – Develop Sequence Prediction Models With Deep Learning*. Machine Learning Mastery, 2017 (cit. on p. 35).

[10] R. Caruana et al. "Intelligible Models for Healthcare: Predicting Pneumonia Risk and Hospital 30-day Readmission". In: *Proceedings of the 21th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining – KDD '15*. 2015, pp. 1721–1730 (cit. on p. 2).

[11] F. Chollet. *Deep Learning mit Python und Keras*. Mitp-Verlag, 2018 (cit. on pp. 20–24, 31).

[12] A. P. Engelbrecht. “Sensitivity Analysis for Selective Learning by Feedforward Neural Networks”. *Fundamenta Informaticae XXI* 46.3 (2001), pp. 1001–1028 (cit. on p. 7).

[13] A. Géron. *Hands-On Machine Learning with Scikit-Learn & TensorFlow*. O’Reilly, 2017 (cit. on pp. 20, 33).

[14] M. Gevrey, I. Dimopoulos, and S. Lek. “Review and Comparison of Methods to Study the Contribution of Variables in Artificial Neural Network models”. *Ecological Modelling* 160.3 (2003), pp. 249–264 (cit. on p. 6).

[15] I. Goodfellow, Y. Bengio, and A. Courville. *Deep Learning*. MIT Press, 2016 (cit. on p. 49).

[16] A. Graves. “Generating Sequences With Recurrent Neural Networks”. Department of Computer Science University of Toronto. 2014. URL: https://arxiv.org/pdf/1308.0850.pdf. Pre-published (cit. on p. 24).

[17] A. Gulli and S. Pal. *Deep Learning with Keras: Implement neural networks with Keras on Theano and TensorFlow*. Packt Publishing, 2017 (cit. on pp. 21–24, 33, 37, 38).

[18] S. Hochreiter and J. Schmidhuber. “Long Short-Term Memory”. *Neural computation* 9.8 (1997), pp. 1735–1780 (cit. on pp. 22, 49).

[19] P. Joshi. *Artifical Intelligence with Python*. Packt Publishing, 2017 (cit. on p. 14).

[20] J. Kauffmann, K.-R. Müller, and G. Montavon. “Towards Explaining Anomalies: A Deep Taylor Decomposition of One-Class Models”. 2018. URL: https://arxiv.org/pdf/1805.06230.pdf. Pre-published (cit. on p. 9).

[21] J. Khan et al. “Classification and Diagnostic Prediction of Cancers using Gene Expression Profiling and Artificial Neural Networks”. *Nature Medicine* 7 (2001), pp. 673–679 (cit. on pp. 6, 7).

[22] L. Arras et al. “Explaining Recurrent Neural Network Predictions in Sentiment Analysis”. In: *Proceedings of the 8th Workshop on Computational Approaches to Subjectivity, Sentiment and Social Media Analysis – WASSA ’17*. 2017, pp. 159–168 (cit. on pp. 12, 13).

[23] W. Landecker et al. “Interpreting Individual Classifications of Hierarchical Networks”. In: *Proceedings of IEEE Symposium on Computational Intelligence and Data Mining – CIDM ’13*. 2013, pp. 32–38 (cit. on pp. 1, 11).

[24] J. Li et al. “Visualizing and Understanding Neural Models in NLP”. In: *Proceedings of the Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies – NAACL-HLT ’16*. 2016, pp. 681–691 (cit. on pp. 1, 18).

[25] Z. C. Lipton. “The Mythos of Model Interpretability”. In: *Proceedings of the ICML Workshop on Human Interpretability in Machine Learning – WHI ’16*. 2016, pp. 96–100 (cit. on p. 1).

[26] W. S. McCulloch and W. Pitts. “A Logical Calculus of the Ideas Immanent in Nervous Activity”. *Bulletin of Mathematical Biophysics* 5.4 (1943), pp. 115–133 (cit. on p. 20).

[27] T. M. Mitchell. *Machine Learning*. McGraw-Hill, 1997 (cit. on p. 19).

[28] G. Montavon, W. Samek, and K.-R. Müller. "Methods for Interpreting and Understanding Deep Neural Networks". *Digital Signal Processing: A Review Journal* 73 (2017), pp. 1–15 (cit. on pp. 1, 2, 5–7, 9–13).

[29] G. Montavon et al. "Deep Taylor Decomposition of Neural Networks". In: *Proceedings of the ICML Workshop on Visualization for Deep Learning – ICML '16*. 2016 (cit. on pp. 1, 8, 9).

[30] G. Montavon et al. "Explaining NonLinear Classification Decisions with Deep Taylor Decomposition". *Pattern Recognition* 65 (2017), pp. 211–222 (cit. on pp. 1, 7–11).

[31] D. Osinga. *Deep Learning Kochbuch: Praxisrezepte für einen schnellen Einstieg*. O'Reilly, 2019 (cit. on p. 35).

[32] P. E. Rauber et al. "Visualizing the Hidden Activity of Artificial Neural Networks". *IEEE Transactions on Visualization and Computer Graphics* 23.1 (2017), pp. 101–110 (cit. on p. 18).

[33] J. Patterson and A. Gibson. *Deep Learning: A Practitioner's Approach*. O'Reilly, 2017 (cit. on pp. 19, 21, 32, 33, 36–38).

[34] N. Poerner, B. Roth, and H. Schütze. "Evaluating Neural Network Explanation Methods using Hybrid Documents and Morphosyntactic Agreement". In: *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics – ACL '18*. 2018, pp. 340–350 (cit. on pp. 1, 5, 15).

[35] M. T. Ribeiro, S. Singh, and C. Guestrin. "'Why Should I Trust You?' Explaining the Predictions of Any Classifier". In: *Proceedings of the 22th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining – SIGKDD '16*. 2016 (cit. on pp. 2, 3, 15).

[36] S. J. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach*. Pearson Education, 2010 (cit. on pp. 13, 19–21).

[37] S. M. Lundberg and S. Lee. "A Unified Approach to Interpreting Model Predictions". In: *Proceedings of the 31st Conference on Neural Information Processing Systems – NIPS '17*. 2017, pp. 4765–4774 (cit. on pp. 1, 2).

[38] W. Samek, T. Wiegand, and K.-R. Müller. "Explainable Artifical Intelligence: Understanding, Visualizing and Interpreting Deep Learning Models". *ITU Journal: ICT Discoveries - Special Issue 1 - The Impact of Artificial Intelligence (AI) on Communication Networks and Services* 1 (2017), pp. 1–10 (cit. on pp. 11, 12).

[39] A. L. Samuel. "Some Studies in Machine Learning using the Game of Checkers". *IBM Journal of Research and Development* 3.3 (1959), pp. 210–229 (cit. on p. 19).

[40] S. Shalev-Shwartz and S. Ben-David. *Understanding Machine Learning: From Theory to Algorithms*. Cambridge University Press, 2014 (cit. on pp. 19, 20).

[41] A. Shrikumar, P. Greenside, and A. Kundaje. "Learning Important Features Through Propagating Activation Differences". In: *Proceedings of the 34th International Conference on Machine Learning – ICML '17*. Vol. 70. 2017, pp. 3145–3153 (cit. on p. 5).

[42] K. Simonyan, A. Vedaldi, and A. Zisserman. "Deep Inside Convolutional Networks: Visualising Image Classification Models and Saliency Maps". 2013. URL: https://arxiv.org/pdf/1312.6034.pdf. Pre-published (cit. on pp. 5–7).

[43] J. T. Springenberg et al. "Striving for Simplicity: The all Convolutional Net". In: *Proceedings of the International Conference on Learning Representations – ICLR '15*. 2014 (cit. on pp. 5, 11).

[44] H. Strobelt et al. "LSTMVis: A Tool for Visual Analysis of Hidden State Dynamics in Recurrent Neural Networks". *IEEE Transactions on Visualization and Computer Graphics* 24.1 (2018), pp. 667–676 (cit. on pp. 18, 49, 59).

[45] M. Sundararajan, A. Taly, and Q. Yan. "Axiomatic Attribution for Deep Networks". In: *Proceedings of the 34th International Conference on Machine Learning – ICML '17*. Vol. 70. 2017, pp. 3319–3328 (cit. on p. 5).

[46] Y. Yang et al. "Explaining Therapy Predictions with Layer-wise Relevance Propagation in Neural Networks". In: *IEEE International Conference on Healthcare Informatics – ICHI '19*. 2018, pp. 152–162 (cit. on p. 13).

[47] M. D. Zeiler and R. Fergus. "Visualizing and Understanding Convolutional Networks". In: *Proceedings of the European Conference on Computer Vision – ECCV '14*. 2014, pp. 818–833 (cit. on pp. 5, 11).

Online sources

[48] T. Broad. *Topological Visualisation of a Convolutional Neural Network*. 2015. URL: http://terencebroad.com/nnvis.html (visited on 01/02/2019) (cit. on p. 1).

[49] Jason Brownlee. *When to Use MLP, CNN, and RNN Neural Networks*. 2018. URL: https://machinelearningmastery.com/when-to-use-mlp-cnn-and-rnn-neural-networks/ (visited on 01/23/2019) (cit. on p. 49).

[50] E. Chen. *Exploring LSTMs*. 2018. URL: http://blog.echen.me/2017/05/30/exploring-lstms/ (visited on 12/29/2018) (cit. on p. 18).

[51] *Colaboratory*. URL: https://research.google.com/colaboratory/faq.html (visited on 01/25/2019) (cit. on pp. 29, 49, 50).

[52] *D3: Data-Driven Documents*. URL: https://d3js.org (visited on 01/22/2019) (cit. on p. 27).

[53] NumPy Developers. *NumPy*. 2018. URL: http://www.numpy.org (visited on 01/23/2019) (cit. on p. 28).

[54] L. Hulstaert. *Understanding Model Predictions with LIME*. 2018. URL: https://towardsdatascience.com/understanding-model-predictions-with-lime-a582fdff3a3b (visited on 03/09/2019) (cit. on pp. 14, 15).

[55] *Introducing JSON*. URL: https://www.json.org (visited on 01/23/2019) (cit. on p. 50).

[56] J. Hunter et al. *Matplotlib*. 2018. URL: https://matplotlib.org (visited on 01/23/2019) (cit. on p. 28).

[57] *Keract*. URL: https://pypi.org/project/keract/ (visited on 01/22/2019) (cit. on p. 28).

[58] *Keras Documentation*. URL: https://keras.io (visited on 01/22/2019) (cit. on p. 28).

[59] *Keras Layers Core*. URL: https://keras.io/layers/core/ (visited on 04/04/2019) (cit. on p. 33).

[60] *Keras Optimizer Documentation*. URL: https://keras.io/optimizers/ (visited on 09/27/2019) (cit. on p. 37).

[61] *Keras Sequential*. URL: https://keras.io/getting-started/sequential-model-guide/ (visited on 04/15/2019) (cit. on p. 32).

[62] *Keras Utils*. URL: https://keras.io/utils (visited on 04/15/2019) (cit. on p. 31).

[63] B. Kim. *Interpretable Machine Learning: The fuss, the concrete and the questions*. 2017. URL: http://people.csail.mit.edu/beenkim/papers/BeenK_FinaleDV_ICML2017_tutorial.pdf (visited on 03/10/2019) (cit. on p. 2).

[64] *Long Short-Term Memory (LSTM)*. URL: https://developer.nvidia.com/discover/lstm (visited on 01/25/2019) (cit. on p. 49).

[65] S. Polamuri. *Difference between Softmax Function and Sigmoid Function*. 2017. URL: http://dataaspirant.com/2017/03/07/difference-between-softmax-function-and-sigmoid-function/ (visited on 01/23/2019) (cit. on p. 37).

[66] M. T. Ribeiro, S. Singh, and C. Guestrin. *Local Interpretable Model-Agnostic Explanations (LIME): An Introduction*. 2016. URL: https://www.oreilly.com/learning/introduction-to-local-interpretable-model-agnostic-explanations-lime (visited on 04/11/2019) (cit. on pp. 5, 14–16).

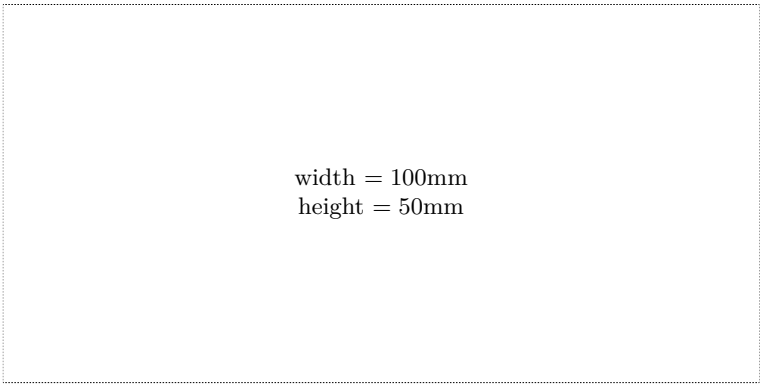
[67] W. Samek. *Interactive LRP Demos*. 2018. URL: http://www.heatmapping.org (visited on 01/07/2019) (cit. on pp. 13, 14).

[68] W. Samek, G. Montavon, and K.-R. Müller. *Interpretable Deep Learning: Towards Understanding and Explaining DNNs*. 2018. URL: http://iphome.hhi.de/samek/pdf/ICIP2018_1.pdf (visited on 03/09/2019) (cit. on p. 6).

[69] *Why and How are GPU's so Important for Neural Network Computations?* URL: https://www.quora.com/Why-and-how-are-GPUs-so-important-for-Neural-Network-computations-Why-cant-GPU-be-used-to-speed-up-any-other-computation-what-is-special-about-NN-computations-that-make-GPUs-useful (visited on 01/25/2019) (cit. on p. 49).

Check Final Print Size

— Check final print size! —



— Remove this page after printing! —