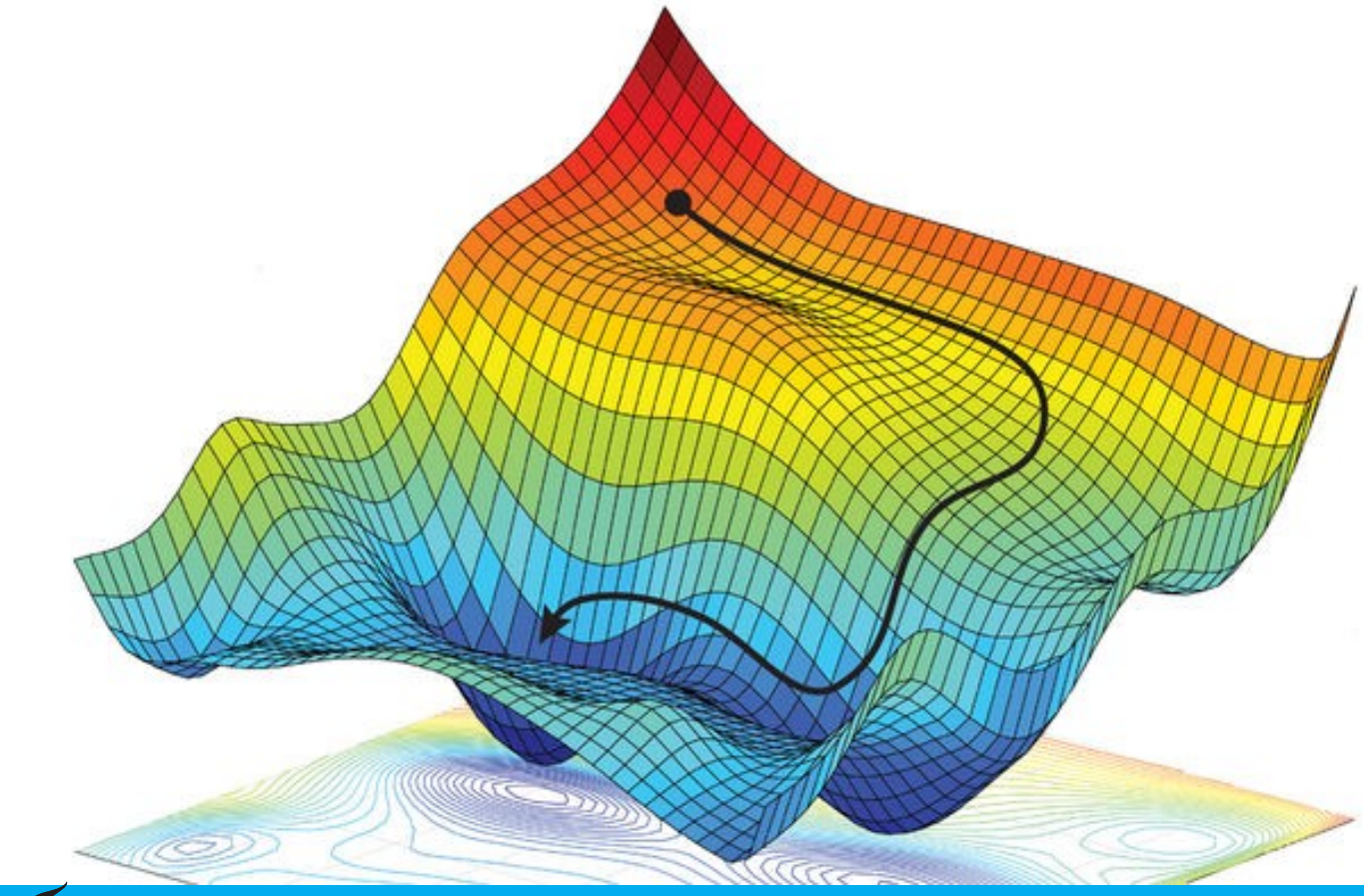
Loss functions and neural networks

Author: J. Kirchner
Supervisor: H.X. Lin

Comparing different loss functions
for NLP neural networks

Source: https://medium.com/analytics-vidhya/gradient-descent-b0dc1af33517



TU Delft

Abstract

Neural network is an active research field which involves many different (unsolved) issues, for example, different types of configuration of the network architectures, training strategies, etc. Amongst these active issues, the choice of loss (or cost) functions plays an important role in how a neural network model is to be optimized (trained) and how the model will perform after the training. Given the choice of measurement criteria, loss functions measure how far an estimated output is from its true value. And the measurement criteria can change depending on the task in hand and the goal to be met. The objective of this project is to understand the role of different loss functions and to evaluate the dependence of the performance on the loss functions using the language prediction problem.

Contents

1 Training a Neural Network **3**
1.0.1 Preparing the data . . . 3
1.1 Forward Pass . . . 4
1.2 Backward propagation . . . 4
1.2.1 Cross Entropy Loss . . . 5
1.2.2 Automatic Differentiation . . . 5
1.3 Optimization . . . 6

2 Recurrent Neural Networks **9**
2.1 Regular RNN . . . 9
2.1.1 Forward Pass . . . 9
2.1.2 Limitations . . . 10
2.2 Long Short-Term Memory . . . 11
2.2.1 Forward Pass . . . 11
2.2.2 Activation functions . . . 12

3 Loss functions **13**
3.1 Purpose of loss functions . . . 13
3.2 Interpreting the gradients . . . 13
3.2.1 Cross Entropy Loss . . . 14
3.2.2 Mean Squared Error . . . 16
3.2.3 Binary Cross Entropy Loss . . . 17
3.2.4 Smooth L1 Loss . . . 18

4 Numerical results **21**
4.1 Dataset . . . 21
4.2 Discussing the code . . . 22
4.3 SL1 loss . . . 22
4.4 Consistency . . . 23
4.5 Overfitting . . . 26
4.5.1 CEL and BCE . . . 26
4.5.2 MSE and SL1 . . . 27
4.5.3 Final results . . . 29
4.6 Whole dataset . . . 29

5 Conclusion **33**

6 Discussion **35**

A Derivations **37**
A.1 Derivation for equation 3.3 . . . 37
A.2 Derivation for equation 3.4 . . . 38

B Books **39**

C Code **43**

Introduction

A lot of research has been done on artificial intelligence (AI) in the last twenty to thirty years. To understand what is happening in this field now, it is first necessary to understand how it was designed. People wanted to make computers function like we do, therefore the most logical step would be to try to understand how our brain works. If one knows this, it could theoretically be implemented into a computer. Now, of course, none have been able to do this yet, however some basic elements are known. Learning for us takes repetition, by looking at questions, formulating an answer, and then comparing this answer to the real solution. In essence, this is also what artificial intelligence models do.

Formulating an answer for a neural network is often visualized like figure 1. Starting with a vector, in this case four dimensional and represented by the blue circles, linear functions are applied to this vector. First making it into a six dimensional hidden layer and then this hidden layer is again transformed into another six dimensional hidden layer by use of a linear function. Finally a last linear function is applied to arrive at the two dimensional output vector, represented by the green circles. Now one thing is missing, as a careful reader will have noticed, the network would be linear in this case. To circumvent this issue, non-linear functions are applied to the vectors. In the classical networks, mostly after each hidden layer. For a more detailed explanation of training neural networks and making predictions with them, see chapter 1.

The circles of the hidden layers represent neurons, which are connected by lines, these lines represent weights that can be altered during training.

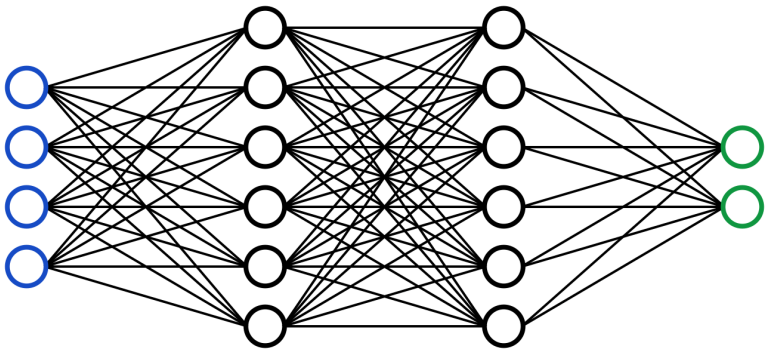


Figure 1: Visualization of a neural network.[17]
Blue are the input nodes and green the output.

Specifically, the network in figure 1 is a feed-forward neural network, however the network used in this report will not be of this kind. Neural networks can have different architectures, so not each neural network will look like the above figure. For language prediction, often recurrent neural networks (RNNs) are used, see chapter 2. These networks work better for time series, where new characters or numbers can depend on those that have already been parsed through the network.

After the forward step, so formulating an answer, it is time to check if the output is any good. To do this some sort of metric needs to be selected. This metric should convert the output and the target into a number. Function that do this are called loss functions and multiple of these functions are described in chapter 3.

Now to train the model, steps are taken to minimize the loss function. For the weights that can be changed, the gradients are calculated with respect to the loss function, this process is known as gradient decent, see section 1.2. And the weights are altered by an optimizer, see section 1.3.

With the focus of this report being the loss functions, the derivatives have been analytically calculated to compare them. This is done in chapter 3. The clearest difference found was the fact that some loss functions seemed to be more likely to arrive at local minima. Which means that they will likely get stuck when training for longer amounts of time and not yield low losses.

To train the network a dataset is needed. For this report, a selection of 96 free English books were

used.[1] For more information on this dataset, see section 4.1. In general, because the books are free, most of them are old. But this does not make them less relevant for this report. A full list of the books can be seen in appendix B.

Training the network on the entire dataset of 502,708 sentences took a lot of time, therefore some tests were done on part of this dataset. Nevertheless results were found, see chapter 4, which in some cases coincided with the hypotheses that arose from the analytical analysis, but in other cases returned unexpected results. Some of the tested loss functions were not designed for classification problems, namely the mean squared error and the smooth L1 loss. However these functions did yield interesting results.

Chapter 1 describes how a neural network is trained, where the focus will be on a feed forward neural network. It contains the three steps, with small explanations and some other important notes.

Chapter 2 addresses the chosen architecture for this report. Where first the general class of models is explained. And later on, the intuition behind some of the choices in the equations is discussed.

Chapter 3 contains the analytical analysis of the loss functions including all equations and some explanation as to why such a function would make sense. Furthermore it contains the analytically calculated gradients, which are compared with each other. Some hypotheses are made that are numerically tested in the next chapter.

Chapter 4 gives the numerical results of using the different loss functions for different problems. One of the functions, the smooth L1 loss, see section 3.2.4, needs to be analyzed first. Afterwards the functions are tested on consistency, the tendency of overfitting and just on the whole dataset.

Chapter 5 contains the conclusion of the numerical and analytical results. With chapter 6 containing the discussion of these results and some limitations of this report.

[1] https://www.gutenberg.org/

1

Training a Neural Network

As can be read in the introduction, a neural network consists of many linear and non-linear functions. The linear functions are the ones that can be changed to make sure the network preforms as it should. These functions contain variables, that in the first iteration often are chosen randomly. Therefore at the start of training, the network will give mostly arbitrary results. The training of a feed-forward neural network can be divided in to three steps.

- Forward pass
- Backward propagation
- Optimization

1.0.1. Preparing the data

Because the training set is very large, not all data can be used at once, as this would require too much memory. Therefore a different method needs to be used. The data is divided into mini-batches of a small number of sentences, in this case 128 sentences per mini-batch. These mini-batches are then fed into the network, one by one, and after each batch, the loss is calculated together with the gradients, and a step is taken. This process is called mini-batch gradient descent[15]. This makes it possible that the training loss does not decrease after a step, as a different part of the data was used to calculate it. The end of the first so called epoch is when all of the mini-batches have been used. Then the data set is shuffled and new mini-batches are made. This process can be repeated a lot of times and here two hyper-parameters become clear, the mini-batch size and the number of epochs.

To use a network, the sentences have to be converted into vectors or matrices. In this case each character was given a standard basis vector. For example, when considering only the lower-case characters and a space, the vectors will be 27 dimensional. Now to incorporate whole sentences, the vectors are put into a matrix. Every sentence can be converted into a $N \times C$ matrix, where N is the number of characters in a sentence and C the number of characters. For programming purposes, the sentences in the same mini-batch can all be put into one 3 dimensional array or matrix, so this will be a $B \times N \times C$ array, where B is the mini-batch size. But to do this, the vectors need to be of the same length. Therefore an end of sentence character is needed, in this report ”@” is used for this purpose. The sentences that are shorter than the longest sentence are filled with this character until they are of the same length.

For each sentence, there needs to be an input and a desired output. The used network returns for every character, the next character. This means that the targeted sentence should be the same, but shifted one character to the left. For example, the sentence ”I want to eat.”, will have ”I want to ea” as input, and ” want to eat” as target. Note here that spaces are also characters, so whenever a string is given in this report that starts or ends with a space, the quotation marks will be around that space as well.

1.1. Forward Pass

In the forward pass the content of the current batch is given to the network. The network then produces an output vector by using the different types of functions. Note that the first time the network is run, the output is mainly arbitrary. As an example one can see in equations (1.1) to (1.3) how a simple, 3-layer, feed-forward network would work with an arbitrary input vector and arbitrary linear functions. Note that this example can be visualized like figure 1, where v_{input} corresponds with the blue circles, the entries of the matrices A_1, A_2 and A_3 correspond with the black lines between the layers, v_1 and v_2 are the hidden layers and therefore the black circles of the figure and v_{out} is represented by the green circles at the end of the network.

Layer 1:

$$v_1 = F_1(v_{input}) = F_1\left(\begin{pmatrix}0\\1\end{pmatrix}\right) = A_1 \cdot \begin{pmatrix}0\\1\end{pmatrix} = \begin{pmatrix}1 & 3\\2 & -1\\1 & 3\end{pmatrix} \cdot \begin{pmatrix}0\\1\end{pmatrix} = \begin{pmatrix}3\\-1\\3\end{pmatrix} \tag{1.1}$$

Layer 2:

$$v_2 = F_2(ReLU(v_1)) = A_2\left(ReLU\left(\begin{pmatrix}3\\-1\\3\end{pmatrix}\right)\right) = \begin{pmatrix}2 & 0 & 0\\-1 & 2 & 0\\0 & 0 & 2\end{pmatrix} \cdot \begin{pmatrix}3\\0\\3\end{pmatrix} = \begin{pmatrix}6\\-3\\6\end{pmatrix} \tag{1.2}$$

Layer 3:

$$v_{out} = F_3(ReLU(v_2)) = A_3\left(ReLU\left(\begin{pmatrix}6\\-3\\6\end{pmatrix}\right)\right) = \begin{pmatrix}1 & 0 & 0\\-1 & 1 & 0\\0 & 0 & 1\end{pmatrix} \cdot \begin{pmatrix}6\\0\\6\end{pmatrix} = \begin{pmatrix}6\\-6\\6\end{pmatrix} \tag{1.3}$$

In equation (1.1), the input character is converted into the vector v_{input}. This is a standard basis vector e_n, where n is the assigned number of the character. In these equations F_1, F_2 and F_3 are different linear functions with variables that can be changed. By theorems from linear algebra, linear functions can always be written as some matrix multiplication in some subspace, hence here matrices are used in the calculations. All values in the matrices will be adjusted as the model trains, but are chosen here semi randomly. The $ReLU$ function[3] is used here as activation function, but any non-linear function will work. Note here that if such a non-linear function is not used, the neural network would have been able to be reduced to only one layer, as the matrices can then be multiplied by each other. This would decrease the number of variables, which would make the network not able to adapt as much. Also the entire network would now be linear, which heavily limits the performance of a network, as most things neural networks are used for are non-linear.

As seen from equation (1.3), the output now contains values outside of the interval [0,1]. For a classification problem, this makes little sense. Therefore often a function is used to project the output onto this interval and make the output into a probability vector. This can be any function, however to train the network it is required to be at least once piecewise differentiable, but a lot of the most used methods require this function to be twice piecewise differentiable. An example is the Softmax function, see equation (3.1).

1.2. Backward propagation

During backward propagation, the weights are changed to fit the network to the training data. To know how the weights need to be changed first the output vector needs to be compared to the targeted vector. For this some sort of error function is needed. In classification problems often Cross Entropy Loss is used[5]. To alter the network, the gradient with respect to the error, is calculated for all variables in the linear functions. This method is called gradient descent[13] and is used often in optimization.

Multiple types of gradient descent exist. The most straight forward one is called batch gradient descent. Here the entire dataset is used and the loss is calculated across this entire dataset, which is then used for the gradients. The advantage is that this method is guaranteed to converge to the global minimum for convex error surfaces and to a local minimum for non-convex error surfaces[15]. An error surface is a surface with dimension equal to the amount of parameters, where the error is present for each combination of parameter values. The problem with this method is that it requires too much memory when using large datasets.

Stochastic gradient descent (SGD) fixes this issue. Instead of using the whole dataset at once, it uses one sentence at a time. This makes the algorithm way faster, however it also causes a heavy fluctuation in the losses. [15] concludes that this may sometimes improve results as it can jump out of certain local minima, but it can also sometimes hurt results. Convergence is no longer guaranteed, but does seem to happen almost always when using the algorithm correctly.

Mini-batch gradient descent uses a combination of the previous two methods. Where here mini-batches are made containing a certain number of sentences. Each of these batches is fed into the network and the loss is calculated over a batch. This reduces the variance, while still being quick. It is the most common algorithm for neural networks and is therefore used in this report as well.

In the example from section 1.1, this means that v_{out} is compared to a vector which is the desired output, by using an error function like the mean squared error. Then for all values in the matrices A_1, A_2 and A_3, the gradients are calculated with respect to the error. This error function is often called a loss function when talking about neural networks.

1.2.1. Cross Entropy Loss

One of the most widely used loss functions for language modelling is the Cross Entropy loss function[1]. It is based on the principle that in language some sort of distribution is present, as to what the next character in a sequence is going to be. This of course can depend on the characters already present in the string. A perfect model will return the same distribution for each character it has to find. So in English if the string is ”Th”, the model should return a very high probability for ”e”, however ”a” should still have a probability greater than zero as the word ”That” can be made.

The loss function is defined as the sum of the Shannon Entropy of the perfect distribution p and the Kullback-Leibler divergence of p and the modelled distribution q, see equation (1.4).

$$H(p,q) = H(p) + D_{KL}(p||q) = -\sum_{x \in X} p(x) \log q(x) \tag{1.4}$$

The problem when using this function for language processing is that the perfect distribution p is not known. Therefore some sort of estimation needs to be used. The Shannon-McMillan-Breiman theorem[9], is a well-known theorem in information theory that estimates this function, see equation (1.5)

$$H(p,q) \approx -\frac{1}{N} \sum_{i=1}^{N} \log(q(x_i)) \tag{1.5}$$

Here N is the amount of characters and should be large enough as it is approximating a limit to infinity.

Note that this entropy can also be seen as the bits required to store a character; Bits per Character (BPC). This metric is often used in comparing different language models. However for this to work, the base of the logarithm should be 2.

1.2.2. Automatic Differentiation

In a normal neural network, which will be larger than just three matrices, it would take too long to calculate each gradient. Because in such a network it is not unlikely that millions of these gradients have to be calculated. Therefore it is necessary to use an efficient algorithm, which calculates these gradients. In the numerical mathematics, such an algorithm is used and is called Automatic Differentiation[16]. The promise of this algorithm is that the differentiation should have the same time complexity as the forward step. There are multiple variants, however for neural network training the Reverse Automatic Differentiation is the most common[4]. This algorithm works best for problems from $\mathbb{R}^n \to \mathbb{R}$ as the time complexity is then the same as the forward step.

Intuitively the method works by dividing the derivative in small parts. While the function is calculated with specific values, the derivative of each operation is also calculated. The best way to see this in formulas, is by using an example.

$$z = x \cdot y + e^{x+y} \tag{1.6}$$

[1] https://leimao.github.io/blog/Entropy-Perplexity/

In equation (1.6), the variable z depends on x and y. However they are not using basic operations, therefore the function needs to be split in to more variables.

$$a = x \cdot y$$
$$b = x + y$$
$$c = e^b$$
$$z = a + c$$

For each of these, the derivatives w.r.t. each part is calculated, so

$$\frac{\partial a}{\partial x} = y \qquad \frac{\partial a}{\partial y} = x$$
$$\frac{\partial b}{\partial x} = \frac{\partial b}{\partial y} = 1 \qquad \frac{\partial c}{\partial b} = e^b$$
$$\frac{\partial z}{\partial a} = 1 \qquad \frac{\partial z}{\partial c} = 1$$

Using the chain rule these functions can be rewritten into

$$\frac{\partial z}{\partial x} = \frac{\partial z}{\partial a}\frac{\partial a}{\partial x} + \frac{\partial z}{\partial c}\frac{\partial c}{\partial b}\frac{\partial b}{\partial x} = y + e^b = y + e^{x+y}$$
$$\frac{\partial z}{\partial y} = \frac{\partial z}{\partial a}\frac{\partial a}{\partial y} + \frac{\partial z}{\partial c}\frac{\partial c}{\partial b}\frac{\partial b}{\partial y} = x + e^b = x + e^{x+y}$$

When keeping track of the operations, this method can also be applied to more complicated problems. Now most of the time it is not necessary to find the formula for the derivative. Using this method this is not necessary as the derivatives can be found by plugging x and y in.

1.3. Optimization

With all the gradients calculated, the logical step would be to adjust the variables accordingly. However, gradient descent may only lead to a local minimum, but this might not be what is wanted. Therefore an optimization function needs to be used. This function adjusts the steps in a way, often stochastically, such that local minima are avoided as much as possible.

Another problem is that when large steps are taken, the method might not reach a minimum at all. The only thing the derivative does, is tell in what direction the slope is, and how steep it is. Therefore the derivative is multiplied by a parameter called the learning rate. But this is not enough, as the optimal learning rate might change during training. Optimizers therefore also often change the learning rate. A widely used optimizer is the Adam[10] optimizer. It is a stochastic optimizer that also changes the learning rate adaptively when training.

Figure 1.1: Visualization of the effect of different learning rates, when using gradient descent.
The loss function is on the y-axis, with a parameter on the x-axis.
The good learning rate represents a variable learning rate that decreases over time.

The learning rates can be visualized like figure 1.1. Here on the left a small learning rate is used, and one can note that this approaches a minimum, however it finds a local minimum. Also it takes a lot of steps to get there.

On the far right is a high learning rate, which takes large steps. This causes the loss to overshoot the minimum a lot of times. Also it has a good chance of getting stuck, when it ”bounces” left and right indefinitely.

The middle figure represents a good variable learning rate. It takes large steps at the start to be able to skip local minima, but decreases the size of these steps after a while. This causes the model to not need too many steps and can still converge to a minimum.

These figures are of course a rough sketch of what is really going on in a neural network, and is not realistic to assume that the optimizer will always find the global minimum. But it is worth keeping into consideration that the left and right scenario are to be avoided.

2

Recurrent Neural Networks

While feed-forward neural networks work well for certain types of applications, they have limited use when predicting the next element of a series. These networks only take one input and cannot differentiate between time or order. The feed-forward network only bases its output on what it learned during training and not what it is told during prediction.

This is where recurrent neural networks (RNN) shine. The main difference between the two types is that recurrent neural networks have a feedback loop after every process. Therefore it does something with the output it produces. For example when giving a word, character by character, to such a network, it can remember the first few characters of the word, even though they have already been passed through the network.

In general this works by having a so-called hidden state. This hidden state is the output of the network after each character is passed through. It is therefore a vector with a certain dimension. This dimension determines how much information can be stored in the vector, which makes for a new hyperparameter: ”Hidden size”. Note that in most models, this hidden state vector is multiplied with a weight matrix. Therefore when increasing the hidden size, the number of parameters often also increases. Sometimes these networks use different layers, meaning that after the hidden state or output is calculated, this output is again fed into a different recurrent neural network. This can be done multiple times, creating a new hyperparameter: ”number of layers”.

2.1. Regular RNN

2.1.1. Forward Pass

With this explained we can go into a bit more detail on how the network works. The example from section 1.1 can be changed a bit to clarify what the RNN does. The main difference being the hidden component. In order for the network to ”remember” the past characters in a sentence, the characters are given to the network one at a time. Each character is made into a standard basis vector of $\mathbb{R}^C$, where C is the number of different characters. The network itself is often visualized like figure 2.1[1]. And the calculations are preformed following equations (2.1) to (2.3).[7]

[1]Source: `https://gotensor.com/2019/02/28/recurrent-neural-networks-remembering-whats-important/`

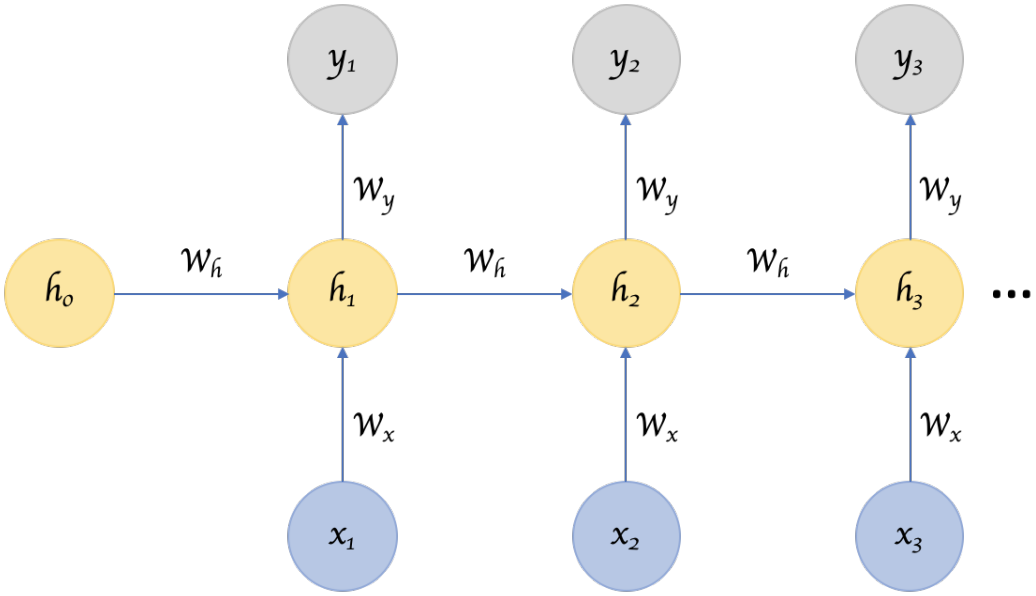


Figure 2.1: Visualization of a recurrent neural network.
Blue are the input nodes, yellow the hidden state and grey the output.

$$\textbf{For n=1: } h_t^1 = \phi(W_{ih^1}x_t + W_{h^1h^1}h_{t-1}^1 + b_h^1) \tag{2.1}$$

$$\textbf{For n} \geq \textbf{2: } h_t^n = \phi(W_{ih^n}x_t + W_{h^{n-1}h^n}h_t^{n-1} + W_{h^nh^n}h_{t-1}^n + b_h^n) \tag{2.2}$$

$$y = f(b_y + \sum_{n=1}^{N} W_{hy}h_T^n) \tag{2.3}$$

Here h_t^n represents the hidden state of the n-th layer from the t-th character in the sequence of T characters. ϕ is a non-linear activation function like $ReLU$. W_{xy} are weight matrices from x to y where i is for input, h^n the n-th hidden layer and y the output. b_h^n is the bias for the hidden state of the n-th layer and b_y the bias in the output. f is a function that can be preformed to get the required output. This can be linear or non-linear.

For these equations to make sense, the first hidden state h_0^1 needs to be set. This vector is often initialized as a zero vector. Then this hidden matrix is multiplied by a ”hidden to hidden” matrix $W_{h^1h^1}$, which contains variables that can be changed during training. The input vector of the first character in this case is multiplied by the ”input to hidden” matrix W_{ih^1}. The resulting two vectors are added together with a bias, where afterwards a nonlinear activation function is applied. This result is then used in the next layer as h_0^2, and the process is repeated for each layer. Note that the visualization in figure 2.1 is only one layer. The final result will be used as the hidden state for the next calculation h_1^1 with the next character. The process is repeated until all characters of the input sequence have been used and the final output can then be transformed into the predicted next character using f.

2.1.2. Limitations

While in this case the RNN will preform better because it can keep some of the information of a new sequence in storage by using the hidden state. It fails on multiple levels at predictions. This is because of a few reasons, where the two most important reasons are discussed:

Short Term Memory: With the hidden state the RNN can store some data such that it can be used later, but as sentences get longer, it is not enough. The method only remembers the last few characters and can therefore not complete long sentences. The size of the hidden state is not the main problem however. The hidden state is also not used very well, therefore the method is considered to have short term memory. It also has no way of filtering the input, this means that combinations of characters like ”he” or ”it” are just as important as some key words in a sentence.

Vanishing gradients or exploding gradients[14]: To make the predictability of neural networks better, often larger networks are considered. These networks have more variables and can as such be adjusted more accurately, in theory. However, when using such networks with gradient descent, something different happens. Gradients for variables at the beginning of the network will often get very high or very low. This means that either too small steps are taken or too large steps, which decreases the networks performance. The phenomenon is called vanishing gradients or exploding gradients and is a really important problem for training neural networks.

2.2. Long Short-Term Memory

To overcome these problems a new type of network was constructed, The Long Short-Term Memory (LSTM) network[8]. This network is also a recurrent neural network, and therefore it uses a hidden state. The only difference being how the hidden state is calculated.[11] This model uses different types of gates to more clearly process information and divide tasks.

2.2.1. Forward Pass

To explain a step in this network it might be easy to first look at a single cell in figure 2.2[2] and equations 2.4 to 2.9.

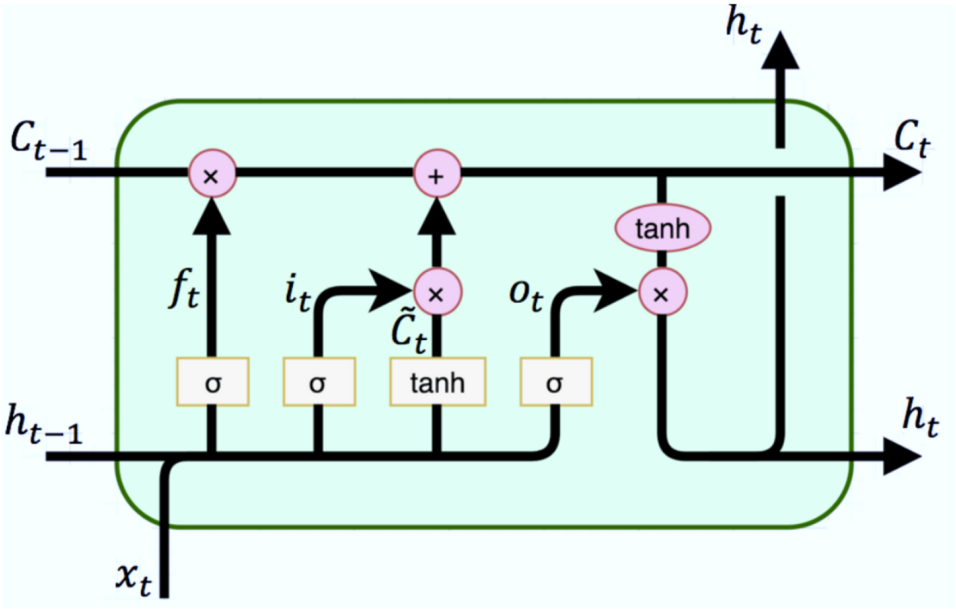


Figure 2.2: Visualization of a single LSTM cell.
Yellow blocks represent that a weight is used first. Red are only using the functions without weights.

$$f_t = \sigma(W_{xf}x_t + W_{hf}h_{t-1} + b_f) \tag{2.4}$$
$$i_t = \sigma(W_{xi}x_t + W_{hi}h_{t-1} + b_i) \tag{2.5}$$
$$\tilde{C}_t = \tanh(W_{xc}x_t + W_{hc}h_{t-1} + b_c) \tag{2.6}$$
$$C_t = f_t \otimes C_{t-1} + i_t \otimes \tilde{C}_t \tag{2.7}$$
$$o_t = \sigma(W_{xo}x_t + W_{ho}h_{t-1} + b_o) \tag{2.8}$$
$$h_t = o_t \otimes \tanh(C_t) \tag{2.9}$$

Here x_t is the t-th character in the sequence. h_t represents the hidden state for character t. W_{ab} is the weight matrix for a in gate b and b_b is the bias vector for gate b. σ is a non-linear activation function,

[2]Source:
https://abdullahsaka.medium.com/long-short-term-memory-lstm-model-in-stock-prediction-d0e1c93717b3

but almost always the logistic sigmoid function. f, i and o are respectively the forget, input and output gates. While $\tilde{C}_t$ is the input modulation gate and C_t the cell state. $\otimes$ is the Hadamard product, which is just the element-wise product.

When trying to understand the model, it is important to remember that each gate fulfills a different task. In each of these gates the input x_t is multiplied by some matrix with variables that can be changed to train the network. The same holds for the previous hidden state h_{t-1}, which holds the ”memory” of the network. For each gate and formula a small explanation will be given.

Forget gate(f_t): After each iteration, so after each character it is necessary to check if something that is memorized, needs to be forgotten. For example when the network encounters a space, it needs to predict a character that does not belong to the word is was just finishing.

Input gate(i_t): Unlike the name suggest, the input gate is not for the input x_t. Input here is used as it manages what should be put in the hidden state. The input gate therefore determines which information should be remembered.

Input modulation gate($\tilde{C}_t$): For the information that is to be stored, it is not only necessary to know which should be remembered, but also how much and in what way. So the input modulation gate calculates for each value in what way it should change.

Cell(C_t): This is the information of the cell, which is equal to the previous cell, but forgets everything that is not necessary and remembers everything that is.

Output gate(o_t): From the information that is stored in the cell C_t, some will be used as output of the cell. This gate determines what information is needed for the output and how much.

2.2.2. Activation functions

As activation functions, it was previously mentioned that the sigmoid function is often used. In figure 2.2, these can be seen as the yellow blocks with a σ inside. This functions is described by equation (2.10). One can note that from this definition the range of the function will be $(0, 1)$.

$$\sigma(x) = \frac{1}{1 + e^{-x}} \tag{2.10}$$

However this is not the only activiation function used in the LSTM architecture. In equations (2.6) and (2.9), the $\tanh(x)$ function is used. This is a common function in mathematics and has a range of $(-1, 1)$. Both of the mentioned functions are defined on $\mathbb{R}$.

The functions are used in different ways in this model. The sigmoid function is used when things need to be chosen. For example in the forget gate, the model needs to decide what it needs to forget. If it wants to forget things it can make sure those get low values, which results in the sigmoid returning a number close to zero. And if not it can return a high number, such that the sigmoid function returns a number close to one.

The tanh function is used when deciding how much and in what way something needs to be changed or saved. For example in the input gate, the sigmoid function is used to decide what to remember. And in the input modulation gate it is decided how much of each value should be changed and if it needs to add or subtract.

3

Loss functions

Having selected a model architecture, it is now time to look at the different loss functions and uncover their differences. These loss functions can work in different ways, sometimes using all the returned probabilities, and sometimes using only the targeted one. However when training the model using any of these function, it quickly became clear that all of them were having trouble with the ”@” symbol at the ends of sentences. With every sentence having a lot of these characters, a very deep local minima will be just guessing only this symbol. This of course is not ideal. Therefore each loss function was used on the characters that were supposed to be an ”@” separate from the other characters. Then the losses were added together, but the loss with the ”@” symbol was multiplied by 0.1, while the loss of the other characters was multiplied by 0.9. In the loss functions below, this was not taken into consideration, but it should not matter that much. The only thing it does is add another case for this character, which has the exact same derivative, apart from the scalar in front.

3.1. Purpose of loss functions

The loss function or error function determines the direction that the network trains in. Ideally the function should be minimized for the perfect model. To create such a function, there needs to be some idea of what a good model should do. In this report, the perfect model will probably never perfectly predict a sentence. This is also impossible. However it should minimize some error. Now this is not as straight forward as it may seem. As for classification problems, the worst a model can preform is when it returns 1, where it should return 0, or the other way around. For non-classification problems, this might not be the case, as those outputs can have an unbounded range. Therefore for classification models certain loss functions were created to penalize this more. The cross entropy loss and binary cross entropy loss, see sections 3.2.1 and 3.2.3 respectively, are examples of such functions. They use a logarithm, which is unbounded on (0,1).

On the other hand, it might still be interesting to see how other functions preform, which were not designed for classification problems. Therefore the mean squared error and smooth L1 loss, see sections 3.2.2 and 3.2.4 respectively, were tested here. These functions contain a more intuitive error term: target output minus output of the model. This error can therefore never exceed 1. In non-classification problems, squaring the error will often cause high errors to be even higher, whereas here it mainly does not reduce high errors as much as low errors.

3.2. Interpreting the gradients

When computing the different gradients, it is interesting to see what different loss functions do. Changing the loss function causes the derivatives to be different. So from the formulas of the derivatives, advantages and disadvantages, may become clear.

Now to use a model, LSTM is not enough to make it work nicely, as it returns a vector with size equal to the hidden size H. Therefore the last layer will be a linear layer, like in a regular feed-forward network. This causes the output vector to be the size of the number of characters C, with each element representing a character. This output vector does not yet contain probabilities, therefore a function

needs to be applied to transform the output into numbers between 0 and 1. Multiple function can be used for this, however here the Softmax function was used, see equation (3.1).

$$S(x_{i,c}) = \frac{\exp(x_{i,c})}{\sum_{\tilde{c}=1}^{C} \exp(x_{i,\tilde{c}})} \tag{3.1}$$

This function transforms the output into probabilities, where the probabilities sum to 1.

When calculating the gradients to the matrices from the LSTM layer, only a part of the derivative is changed when using different loss functions. This can be seen by using the chain rule, see equation (3.2). This equation is an example for one of the matrices.

$$\frac{\partial \mathcal{L}(q)}{\partial (W_{xo})_{ab}} = \sum_{i=1}^{N} \frac{\partial \mathcal{L}(q)}{\partial (h_i)_a} \frac{\partial (h_i)_a}{\partial (o_i)_a} \frac{\partial (o_i)_a}{\partial (W_{xo})_{ab}} \tag{3.2}$$

Here $\mathcal{L}$ is the loss and q the returned probability vector. W_{xo} is the weight matrix in the output layer, see equation (2.8).$(h_i)_a$ is element a of the hidden state for the i-th character. $(o_i)_a$ represents element a of the output layer for the i-th character and N is the number of characters in the sentence.

The derivatives to all of the other weight matrices will be of the same form, where $\frac{\partial \mathcal{L}(q)}{\partial (h_i)_a}$ is present and the same each time. While it is interesting to see all of these function, this is not the focus of this report, as it a characteristic of the model architecture and not of the loss function. Therefore only this one is used and explained, see equation (3.4), for the derivative using Cross Entropy loss and an explanation.

3.2.1. Cross Entropy Loss

As was discussed before the most common loss function used in neural networks is the Cross Entropy Loss, see section 1.2.1. With this function and a working neural network, it is interesting to see why such a function works. To do this the gradients can be calculated, to maybe see something from the formula.

Before calculating the gradients inside the LSTM part, it might be interesting to first calculate the gradients of the linear layer. This linear layer is a $C \times H$ matrix with variable entries. Differentiating equation (1.5) with respect to each of these elements yields equation (3.3).

$$\frac{\partial H(p,q)}{\partial A_{ab}} = -\frac{1}{N} \sum_{i=1}^{N} (h_i)_b \left(-S(x_{i,a}) \mathbf{1}_{y_i \neq a} + (1 - S(x_{i,a})) \mathbf{1}_{y_i = a} \right) \tag{3.3}$$

Here N is the number of characters or words in a sentence, depending if a word based algorithm was used or a character based. $H(p,q)$ is the estimated Shannon Entropy. A_{ab} is the entry on row a and column b of the linear layer. $(h_i)_b$ is the b-th entry in the returned hidden state vector from the LSTM layer. $x_{i,c}$ is the predicted value for the i-th character being character c. This value is not a probability, but S is the Softmax function, which makes it a probability. y_i is the target character for character i.

One thing to note is that when using gradient descent, the negative gradient is used to calculate the step in each of the variables. Therefore we can ignore the minus sign at the beginning of the equation. Inspecting this function, one can note that the function is divided into two parts. Each of them is multiplied by $(h_i)_b$, which is the result of the LSTM network and makes sure it is scaled correctly. So when the LSTM network returns something negative, it should increase the matrix entries to get lower output probabilities, instead of decreasing these entries.

One of the parts is for when the targeted class is not the same as the column in A, this is the left part. That part part then becomes $-S(x_{i,a})$. This is the guessed probability for character a, which in this case was not the targeted one. Therefore the more the guess is wrong, the larger this probability becomes. This can be seen as taking large steps in decreasing the guessed probability if the prediction of wrong answers is very wrong and taking small steps if the prediction of wrong answers is close to 0.

The right part is used when the column is not the same as the targeted output. Then it does the same thing, but now inverted. So when the guess is correct and close to one, it does not increase the gradient much. But when the guess is wrong, the gradient will become larger. Note that this part is

always positive, and will therefore try to increase the entry in the matrix and the output probability for this character.

This is done for each character in the sentence and the average error is used as the final gradient. Which is really simple, when compared to the other loss functions.

Calculating the derivative for the matrix inside the LSTM part, equation (3.2), was used.

$$\frac{\partial H(p,q)}{\partial (W_{xo})_{ab}} = -\frac{1}{N}\sum_{i=1}^{N} \frac{\sum_{c=1}^{C}(A_{y_i a} - A_{ca})\exp(x_{i,c})}{\sum_{c=1}^{C}\exp(x_{i,c})} \cdot \tanh((c_i)_a) \cdot \frac{(x_i)_b}{\exp(\alpha_{i,a}) + \exp(-\alpha_{i,a}) + 2} \quad (3.4)$$

where: $\alpha_{i,a} = (W_{xo})_a x_i + (W_{ho})_a h_{i-1} + (b_o)_a$

Here N is the number of characters in the sentence. $(\cdot)_a$ denotes the a-th entry in a vector or the a-th row in a matrix. c_i represents the cell state for the i-th character, calculated using equation (2.7). y_i is the target character for character i in the sentence. x_i is the input vector at the i-th character. W_{xo} and W_{ho} are the weight matrices from equation (2.8), which are the variables that can be changed together with the bias b_o. h_i is the hidden state for the i-th character. A_{ca} represents the entry on row c and column a of the matrix of the linear layer. $x_{i,c}$ is the predicted value for the i-th character being character c.

While this function looks a bit more complicated, it is worth noting that a lot of times the summation in this function will equal zero. This is because of the $(x_i)_b$ term being zero, when the input character does not equal character b. Therefore, it again becomes clear that each column in this matrix W_{xo} represents a single character.

Furthermore it is good to remember that the point of the output layer, where this matrix appears, is to remember how much of everything needs to be remembered. The cell-state c_i contains information about all values. The output gate decides what information to use for the hidden layer and output. The $\tanh((c_i)_a)$ part is the information it got from the cell. The

$$\frac{1}{N}\frac{\sum_{c=1}^{C}(A_{y_i a} - A_{ca})\exp(x_{i,c})}{\sum_{c=1}^{C}\exp(x_{i,c})}$$

part looks a bit like a part of equation (3.3), this is a large value if the guesses that should return low values, return high values. So when the model is performing like it should, the derivative will be small as not much needs to change. The $(A_{ay_i} - A_{ac})$ part scales it in a way such that it takes steps that minimize the guesses for the other characters on average.

In the case that $(x_i)_b = 1$, the only part left to analyze will be $\frac{1}{\exp(\alpha_{i,a})+\exp(-\alpha_{i,a})+2}$. Plotting this function with respect to α yields figure 3.1. Here it can be noted that the function returns low values when α is high or low. When α is close to zero, the function returns its highest values. Because it is the case that the input character corresponds with the column of the matrix, something should be remembered or forgotten. If α is close to zero, it will not forget and not remember much. Therefore the derivative will be high to change this.

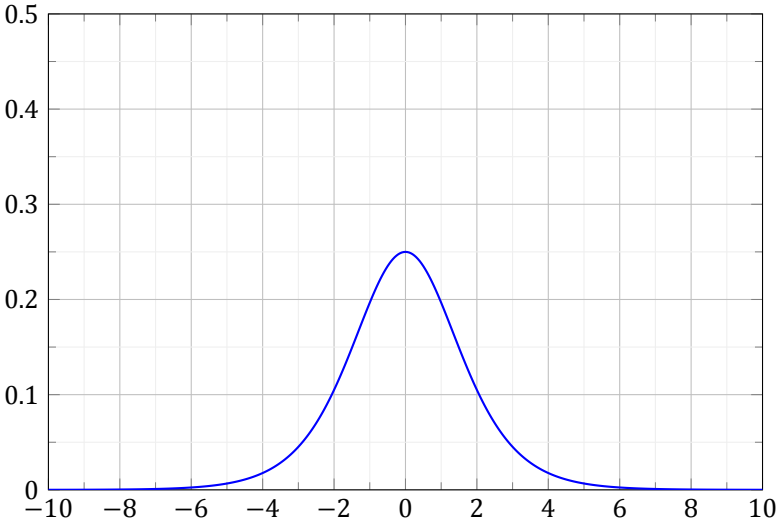


Figure 3.1: Plot of $y = \frac{1}{\exp(x)+\exp(-x)+2}$

3.2.2. Mean Squared Error

The most well known error function or loss function is probably the Mean Squared Error (MSE) function. This function does exactly what is described in the name. It computes average of the squared differences, see equation (3.5). Note that here are two sums, one for the characters in the sentence and the other for the guesses of the different characters. This is different from the cross entropy loss, as that loss function only summed over the characters that should be guessed.

$$MSE(q) = \frac{1}{N}\sum_{i=1}^{N}\frac{1}{C}\sum_{i=1}^{C}\left(\frac{\exp(x_{i,c})}{\sum_{\tilde{c}=1}^{C}\exp(x_{i,\tilde{c}})} - y_{i,c}\right)^2 \tag{3.5}$$

Here $x_{i,c}$ represents the output of the model for the i-th character in the sentence to be character c. $y_{i,c} \in \{0, 1\}$ equals 1 if the i-th character in the target is character c and 0 otherwise.

Again first the gradients of the linear layer are calculated.

$$\frac{\partial MSE(q)}{\partial A_{ab}} = -\frac{2}{N}\sum_{i=1}^{N}(h_i)_b S(x_{i,a})\frac{1}{C}\sum_{c=1}^{C}\beta_{i,c}\cdot\left(-S(x_{i,c})\mathbf{1}_{c\neq a} + (1-S(x_{i,c}))\mathbf{1}_{c=a}\right)$$

$$\text{where } \beta_{i,c} = y_{i,c} - \frac{\exp(x_{i,c})}{\sum_{\tilde{c}=1}^{C}\exp(x_{i,\tilde{c}})} = y_{i,c} - S(x_{i,c}) \tag{3.6}$$

In this equation A_{ab} is the a-th row and column b of the linear layer matrix. $(h_i)_b$ is entry b of the output or hidden state of the LSTM part for the i-th character in the sentence. S is the Softmax function defined in equation (3.1).

This equation is similar to the one found using Cross Entropy Loss, see equation (3.3). The main difference being the second summation. When looking at the part where $c = y_i$, it is almost the same apart from the added $\beta_{i,c}$, which represents the error made for the correct character. This seems like a logical inclusion that cannot do much wrong. Also the $S(x_{i,a})$ part is added, which causes this derivative to be small when the guessed value is low. This can cause problems, as if the model is trained a bit, this causes it to be hard to change further. Which might mean that it gets stuck in a local minimum.

Now the whole summation over c, can be seen as a weighted average over how bad and how good guesses are and what needs to change to make it better. If a guess is high, when it should be low, it can be decreased by increasing the other guesses. This is because the Softmax function causes all the guesses to sum to 1.

With this derivative analyzed, it is time for the derivative inside the LSTM layer. Again equation (3.2) is used. This time a big part of the equation is the same, therefore only the new part is written out.

$$\frac{\partial MSE(q)}{\partial (W_{xo})_{ab}} = -\frac{2}{NC}\sum_{i=1}^{N}\sum_{c=1}^{C}\beta_{i,c}S(x_{i,c})\frac{\sum_{\check{c}=1}^{C}(A_{ca}-A_{\check{c}a})\exp(x_{i,\check{c}})}{\sum_{\check{c}=1}^{C}\exp(x_{i,\check{c}})}\frac{\partial (h_i)_a}{\partial (o_i)_a}\frac{\partial (o_i)_a}{\partial (W_{xo})_{ab}} \tag{3.7}$$

Here the $\beta_{i,c}$ appears again, which is not that bad. However the $S(x_{i,c})$ part is not good in certain scenarios. This part causes the whole gradient to be small if the predicted value is small. Seeing this in both of the equations, the hypothesis becomes that using this loss function will probably cause the model to get stuck in local minima. And it will be hard for the model to get out of these local minima, as the gradients become smaller. At the start if the probabilities are close to 0.5, the loss will work fine as the derivatives are not that small. Therefore it might be important to choose the initial values in the weights well, if possible, when using the mean squared error.

The fact that the second summation is present, might contain advantages. This is good if wrong guesses are important, because this way the loss increases if other guesses have high probability, not only if the correct guess has low probability. However this might cause the model to over fit faster.

3.2.3. Binary Cross Entropy Loss

Even though the mean squared error is not really meant for classification problems, it does propose an interesting question. The second summation, to also sum over the wrong predictions, might have some useful properties. Therefore the next loss functions, being analyzed is the Binary Cross Entropy Loss (BCE), see equation (3.8). This loss function very much looks like the regular cross entropy loss, but with an added summation for the rest of the characters.

$$\begin{aligned} BCE(q) &= -\frac{1}{N}\sum_{i=1}^{N}\frac{1}{C}\sum_{c=1}^{C} y_{i,c}\log\big(S(x_{i,c})\big) + (1-y_{i,c})\log\big(1-S(x_{i,c})\big) \\ &= -\frac{1}{N}\sum_{i=1}^{N}\frac{1}{C}\log\big(S(x_{i,y_i})\big) + \frac{1}{C}\sum_{c=1,c\neq y_i}^{C}\log\big(1-S(x_{i,c})\big) \end{aligned} \tag{3.8}$$

In this equation S is the Softmax function, however all functions with a range of $(0,1)$, the whole of $\mathbb{R}$ as domain and preferably at least twice piecewise differentiable, can be used.

With this function being split into two parts, and one part being the same as equation (1.5), the derivative is split into two parts with the first part almost the same as equation (3.3).

$$\frac{\partial BCE(q)}{\partial A_{ab}} = -\frac{1}{NC}\sum_{i=1}^{N}(h_i)_b\left(-2S(x_{i,a})\mathbf{1}_{y_i\neq a} + (1-S(x_{i,a}))\mathbf{1}_{y_i=a} + S(x_{i,a})\sum_{c=1,c\neq y_i,c\neq a}^{C}\frac{S(x_{i,c})}{1-S(x_{i,c})}\right) \tag{3.9}$$

The obtained equation is a lot nicer than the one from the mean squared error. This equation contains three parts. To start with it is good to look at the last part, as this part will always be used. It sums over the characters that were not used for the row in A and not the targeted character. So each of the $S(x_{i,c})$ should be a low value. The fraction returns a low value if this is the case, which is good as nothing needs to change. However if the guess for the character is high, this fraction approaches infinity, causing the derivative to get larger. This is good as if the probability for other characters increases, the probability for the wrong character decreases.

Now the other part is almost the same as equation (3.3). Under this equation it is explained why this is good. The only difference is the 2 in front of the $\mathbf{1}_{y_1\neq a}$, this comes from the fact that it is double counting it. If $y_i \neq a$, the summation will include $c = a$, in which case that part can be added. This also causes the amount of additions taking place, will be equal to C if the 2 is counted as 2 additions. Which of course is logical as an average over each character is taken.

Differentiating with respect to the hidden layer, yields equation (3.10). Where one can see a com-

bination of the last two loss functions.

$$\frac{\partial BCE(q)}{\partial (W_{xo})_{ab}} = -\frac{1}{NC}\sum_{i=1}^{N}\left(\frac{\sum_{\check{c}=1}^{C}(A_{y_i a} - A_{\check{c}a})\exp(x_{i,\check{c}})}{\sum_{\check{c}=1}^{C}\exp(x_{i,\check{c}})} - \sum_{c=1,c\neq y_i}^{C}\frac{S(x_{i,c})}{1-S(x_{i,c})}\cdot\frac{\sum_{\check{c}=1}^{C}(A_{ca} - A_{\check{c}a})\exp(x_{i,\check{c}})}{\sum_{\check{c}=1}^{C}\exp(x_{i,\check{c}})}\right)\frac{\partial (h_i)_a}{\partial (o_i)_a}\frac{\partial (o_i)_a}{\partial (W_{xo})_{ab}} \tag{3.10}$$

To little surprise, this loss function starts the same as the regular cross entropy loss function of equation (3.4). However the target character is not the only one that is incorporated into the loss. The second part of this derivative does the other characters. It basically works in the same way, apart from the added error term, which was also seen in equation (3.9). Like in that equation, here it promises to do good as the equation seems logical.

Now comparing the CEL and the BCE, it is hard to arrive at a concrete hypothesis. In this report the functions will be used in a language prediction model. For predicting the next character in a sequence, it is impossible to predict with a high percentage of certainty, what the next character will be. Exceptions to this may be inside words. Therefore, it is not weird to assume that multiple characters will have a somewhat high probability of being next. The BCE model punishes that, which might cause it to arrive at very similar models after training, when starting with different random weights. However the CEL model does not care much if another character also has a high probability, which may cause the model to be more influenced by the starting weights. But this can also make the model reflect reality a bit more in this case.

3.2.4. Smooth L1 Loss

With the mean squared error using the L2 norm, it is logical to also test the L1 norm. However an improvement for this function has been suggested[6]. The derivative of the normal L1 loss function is of course mainly constant w.r.t. $(x_i - y_i)$. This can cause problems as the function is no longer convex. Also this function is not differentiable on $(x_i - y_i) = 0$. Therefore this adjusted L1 loss function was created. It combines the L1 norm and L2 norm such that it is less sensitive to outliers and can prevent exploding gradients, see equation (3.11). This loss has a hyperparameter β, which decides when the loss should use the L2 norm and when the L1 norm.

$$SL1(p,q) = \frac{1}{N}\sum_{i=1}^{N}\frac{1}{C}\sum_{c=1}^{C}\frac{0.5\delta_{i,c}^2}{\beta}\mathbf{1}_{|\delta_{i,c}|<\beta} + (|\delta_{i,c}| - 0.5\beta)\mathbf{1}_{|\delta_{i,c}|\geq\beta}$$
$$\text{where: } \delta_{i,c} = S(x_{i,c}) - y_{i,c} \tag{3.11}$$

One can check that this function is continuous by realizing that the only points that have to be checked are $|\delta_{i,c}| = |S(x_{i,c}) - y_{i,c}| = \beta$. For continuous differentiability, first the derivative needs to be calculated. Note that in the equations below, only the part after the summations is used, as this is enough.

$$\lim_{|\delta_{i,c}|\uparrow\beta} SL1(p,q) = 0.5\beta \qquad \lim_{|\delta_{i,c}|\downarrow\beta} SL1(p,q) = 0.5\beta$$

$$\frac{\partial SL1(p,q)}{\partial \delta_{i,c}} = \begin{cases} \frac{\delta_{i,c}}{\beta} & -\beta < \delta_{i,c} < \beta \\ 1 & \delta_{i,c} \geq \beta \\ -1 & \delta_{i,c} \leq \beta \end{cases} \quad \Longrightarrow \quad \begin{cases} \lim_{\delta_{i,c}\to\beta}\frac{\partial SL1(p,q)}{\partial \delta_{i,c}} = 1 \\ \lim_{\delta_{i,c}\to-\beta}\frac{\partial SL1(p,q)}{\partial \delta_{i,c}} = -1 \end{cases}$$

Now this function is not twice continuously differentiable, however the second derivative is piecewise continuous.

To analyze the effect it has on the derivatives w.r.t. the linear layer matrix. The derivatives are again calculated.

$$\frac{\partial SL1(p,q)}{\partial A_{ab}} = -\frac{1}{N}\sum_{i=1}^{N}(h_i)_b S(x_{i,a})\frac{1}{C}\sum_{c=1}^{C}\left(\frac{y_{i,c} - S(x_{i,c})}{\beta}\mathbf{1}_{|\delta_{i,c}|<\beta} + (-1)^{1-y_{i,c}}\mathbf{1}_{|\delta_{i,c}|>\beta}\right)(-S(x_{i,c})\mathbf{1}_{c\neq a} + (1-S(x_{i,c}))\mathbf{1}_{c=a}) \tag{3.12}$$

Here it can be seen that if the guess is close to the target, so when $|\delta_{i,c}| < \beta$, then this derivative is the same as the one from the mean squared error up to a scalar, see equation (3.6). This makes sense of course, but now if the guess is not good and differs more than beta from the target, it does something different. It basically uses the most extreme error in that case, for the first part. This could make it easier for the loss function to get out of local minima. However the loss is still multiplied by $S(x_{i,a})$, which can get really close to zero. In which case the loss will be small, as all parts of the equation are bounded by 1.

$$\frac{\partial SL1(p,q)}{\partial (W_{xo})_{ab}} = -\frac{1}{N}\sum_{i=1}^{N}\frac{1}{C}\sum_{c=1}^{C}\left(\frac{y_{i,c}-S(x_{i,c})}{\beta}\mathbf{1}_{|\delta_{i,c}|<\beta} + (-1)^{1-y_i}\mathbf{1}_{|\delta_{i,c}|>\beta}\right)S(x_{i,c})\frac{\sum_{\check{c}=1}^{C}(A_{ca}-A_{\check{c}a})\exp(x_{j,\check{c}})}{\sum_{\check{c}=1}^{C}\exp(x_{j,\check{c}})}\frac{\partial (h_i)_a}{\partial (o_i)_a}\frac{\partial (o_i)_a}{\partial (W_{xo})_{ab}} \tag{3.13}$$

This equation is nothing spectacular, as here the same can be seen. When the error is small enough, this derivative is equal to the mean squared error, up to an scalar. When it is not, it uses the extreme case where $y_{i,c} - S(x_{i,c})$ becomes 1 or -1 depending on if the character is the targeted value.

With the derivatives discussed, the hypothesis is that this function will probably mostly perform like the mean squared error. It might be a little bit better at avoiding local minima, or climbing out of them, however it will probably still suffer from this issue.

4

Numerical results

Having analyzed all the loss functions analytically, it is now time to verify and evaluate these results by comparing them to a numerical analysis. The results will be made with the model described in section 3.2. So first a one layer LSTM network, with a hidden size of 1500, then a feed-forward layer, which is a matrix of $C \times 1500$, where C is the number of different characters. The optimizer which is used is the Adam optimizer[10], with a learning rate of 0.001. All initial weights of the models were drawn randomly from a uniform distribution $U(-\frac{1}{\sqrt{hidden\_size}}, \frac{1}{\sqrt{hidden\_size}})$. Worth noting is that all analytically found gradients, were checked numerically, to see if this would yield the same answers. This was the case for each gradient. Note throughout this chapter that a model of this size is very small compared to some of the well-known models. Therefore some results could be attributed to this fact instead of what is stated here. More analysis about this can be found in the discussion.

4.1. Dataset

To discuss the numerical results, first the dataset needs to be discussed. As mentioned before, for the dataset multiple books were used. These books are available online from Project Gutenberg[1]. The selected books are all the top 100 books of July 2022, with the exception of the non English books, for a total of 94 books. A full list of the books can be found in the appendix, where also all the authors are mentioned. Because these books are all free to download, most books are old.

To make these books readable for the model, a few adjustments were made. First sentences with fewer than five words and more than fifty words were not used, to try to avoid outliers. Secondly all letters were decapitalized. Then, to distinguish between sentences and words, the Python NLTK library[1] was used. The regular sentencing from this library was used for the sentences. For the words a selection of characters were used: ”!”, ”,”, ”.”, ”:”, ”?”, ”””, ””””, ”-” and all the non-capitalized letters, which with the RegexpTokenizer function were considered part of words. These words were then put into the sentence again with spaces in between. This is done for each book and all of these sentences are stored. When all the books are done, the largest sentence is found and every other sentence is filled with ”@” characters to make each sentence of the same length. This causes the number of characters C to be 36.

Not all of the sentences were used for each numerical result, however the sentences were always split into a training set, a validation set and a test set. Which contain 80%, 10% and 10% of the used sentences respectively. The whole dataset contains 502,708 sentences. How the data is fed into the network can be read in section 1.0.1.

The division into the three sets is common everywhere in machine learning. Because for many applications the model will be applied to scenarios that are not in the dataset, it is important to know how the model will perform on cases it has not seen before. The training dataset is self-explanatory and is only used for training the model. Now the validation set, which is important to be completely different form the training set, is for evaluating the model on data it has not seen before. During training of models, it is common to first see both the training loss decrease, as well as the validation loss.

[1] https://www.gutenberg.org

But after a bit of training, the validation loss will likely start to increase. This is because the model is overfitting the training set. It will start to learn patterns that only appear in that specific subset of the dataset. The last set, which is not really used in this report, is the test set. When training a model all the parameters are optimized, however a lot of models have hyper-parameters, which need to be set before training. When trying to optimize these hyper-parameters, one often tries to optimize the validation loss. But when this is done, the evaluation becomes biased like before. Therefore the test set is there to compare the models without bias and should therefore never be optimized towards.

4.2. Discussing the code

The code used for the numerical analysis can be found in appendix C. Although the functions are explained there briefly, some parts may deserve a little more attention. The main thing to note here is that the code was made in a IPython Notebook (.ipynb) file. This works by having different cells that can be run independently. Every part between "#%%", is a separate cell, however if run from top to bottom, the code should work.

A few libraries were used, some are really common in Python programs, like NumPy, pandas and matplotlib, others a little less popular. The NLTK library[1], mentioned in the previous section is one of them. Also the PyTorch[2] library is not commonly used. This library contains methods to develop neural networks and other machine learning models. For this task it is relatively popular. A lot of the equations of the different loss functions came from the documentation of this library[12]. The library makes it easy to implement these functions and use them effectively. The last uncommon libarary used, is the tqdm[3] library, which just prints progress bars for loops, such that the training time can be measured.

Before training the model, it is useful to have a dataloader such that all the data can be fed into the network in the right way. With a dataloader the user has more control over what the network receives. This is needed for training as this data needs to be shuffled randomly before beginning a new epoch. The class used for this can be found in lines 104 to 141.

After being able to stream the data correctly, the full model class can be created, see lines 158 to 305. Here the PyTorch library is mainly used, and this model therefore is an extension on the already present module in the library. For calculating the losses, different functions were made inside this class. The MSE, BCE and SL1 losses work in the same way and require the Softmax function to be explicitly called beforehand. Therefore one function was used for these three losses, here called BCE, lines 196 to 210. The CEL already has a Softmax function built in, when using this specific library. It therefore required a separate function.

Like mentioned before, the characters which would need to be an "@" were first found. The loss is calculated separately on these characters and multiplied by 0.1, while the loss of the other characters is multiplied by 0.9, before adding them together. Calculating the gradients is completely done by the library and requires only one line of code in this program, see line 258. The same holds for taking the step and adjusting all the weights, which is done in the next line.

A few more functions and cells were used to extract all the results. Lines 308 to 326 contain two functions that can, when given a sentence, convert that sentence to vectors and then input it into the model. From the model it can translate the returned vectors to sentences again, by taking the character with highest probability each time. The other cells are self-explanatory.

4.3. SL1 loss

The SL1 loss described in section 3.2.4 has a hyper-parameter β, which needs to be analyzed first. From the analytical results and the reason why this parameter exists, it is known that this determines when the SL1 loss uses a mean squared error and when a mean absolute error. For this problem, the returned probabilities are between 0 and 1, therefore if a β of 1 is used, this loss just becomes the mean squared error. Similarly for β close to 0, this loss will almost be a regular mean absolute error loss.

To choose the correct β, a few are tried, namely: 0.75, 0.5, 0.25, 0.1. With these models the network is trained on 40,000 randomly selected sentences from the first 50,000 sentences of the dataset. This is to reduce computing time, also these models were trained with a hidden size of 1000. For each

[2] https://pytorch.org/
[3] https://github.com/tqdm/tqdm

β, the model is trained 3 times, randomizing the starting weights each time. And all training was done for 3 epochs. Note that because the models were only trained 3 times, coincidences may occur. Nevertheless a clear pattern appears from the results of table 4.1. Note that in the table an interval is given, which denotes the highest found loss of the three times, and the lowest one.

Loss\β	**0.75**	**0.5**	**0.25**	**0.1**
CEL	[2.13,3.12]	[2.19,2.34]	[2.44,2.58]	[2.72,2.75]
MSE ($\times 10^{-2}$)	[1.90,2.52]	[1.86,2.06]	[2.02,2.15]	[2.08,2.13]
BCE ($\times 10^{-2}$)	[8.05,11.32]	[8.16,8.74]	[8.98,9.46]	[9.90,10.01]

Table 4.1: Training losses obtained by training the network described in section 3.2 for 3 epochs, using the SL1 loss function of section 3.2.4, with a variable beta. All data is from training the network three times, each time randomizing the starting weights. Highest and lowest loss of these three times were noted as an interval.

Interestingly it seems that when the β parameter is decreased, the model is trained more consistently as the lowest and highest loss differ less. However this comes at a price, as the average loss does seem to increase. Now this is in agreement with what was found analytically. With a high value for β, it should behave almost like the mean squared error, where the hypothesis was that it got easily stuck in a local minimum and therefore strongly dependent on the starting weights. A large spread in the calculated losses would indicate this.

To now choose a specific β is highly dependent on the problem. Knowing something about the problem, could help with choosing starting weights instead of just randomly choosing them. In that case high values of β can be used. Also if one has a lot of time, the model can be trained with a high value β multiple times and then the lowest loss can be chosen. For this report a value of 0.5 is used, as the mean squared error is already being used, so with this value, this loss differs still. Furthermore when using the Softmax function in classification problems, the 0.5 has a special meaning, as when the targeted value has a probability of 0.5 or higher, it means that it has the highest probability of all characters. When predicting a sentence, the character with the highest probability will be chosen, it will therefore always be this character.

4.4. Consistency

Before checking how the networks do after training for a long time, it is maybe interesting to see how they compare on consistency. By consistency, it is meant to compare how the networks can differ from different initial weights. In most cases these weights will be chosen randomly at the start. It can therefore be useful to know if the trained model is very dependent on these weights or not. This is the same kind of test as done in the previous section. The training set will again be 40,000 sentences randomly chosen from the first 50,000 sentences of the dataset. Training over this dataset will be done for three epochs and is done ten times starting with randomized weights each time. After each trained model, the model is applied to the start of a sentence: ”how much”, to compare the differences in grammar, spelling and choice of words.

Loss function\Training function	**CEL**	**BCE**	**MSE**	**SL1**
CEL	2.08 (0.104)	2.21 (0.361)	3.27 (0.960)	2.48 (0.658)
BCE ($\times 10^{-2}$)	7.99 (0.345)	8.40 (1.22)	11.55 (2.859)	9.05 (2.11)
MSE ($\times 10^{-2}$)	2.01 (0.068)	2.08 (0.250)	2.25 (0.245)	1.94 (0.322)
SL1 ($\times 10^{-2}$)	1.67 (0.048)	1.72 (0.180)	1.84 (0.176)	1.57 (0.241)

Table 4.2: Sample mean of training losses after training the model described in section 3.2 ten times for 3 epochs, using different loss functions. In brackets is the sample standard deviation of the respective model.

Comparing the different models by the average result, one can see that the CEL and BCE seem to be really close in all the tested metrics. This was expected as these methods are meant to be used in classification problems and are very similar. The MSE and SL1 methods both have a little higher average, when comparing with the CEL or BCE metrics. However the SL1 method does really well in the other two metrics on average. The MSE model stands out with the highest sample mean, but this could be a coincidence. Testing these methods all ten times took about twelve hours, hence there was not enough time to do more. Values might therefore be a little inaccurate, but they should give some

sort of indication. An interesting result here is that the SL1 method has the lowest sample mean when comparing with the MSE and SL1 metric. These metrics use a different way of calculating the losses, as they do not use logarithms, but instead use a squared or absolute error. This means that the loss added for a single guess cannot exceed 1, while the loss, when calculated by the logarithm, can get infinitely high. This means that the obtained probabilities from the model when trained with the SL1 loss or the MSE loss, might be really polarized. Some are probably really high while others really low as it is not punished as much if a returned probability differs a lot from the targeted one.

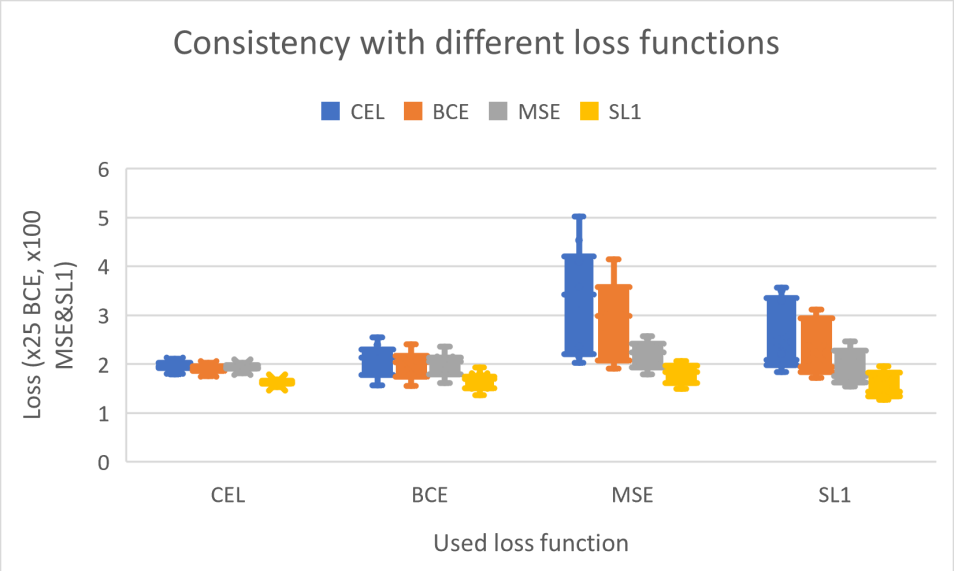
Looking at the sample standard deviations, a lot more diversity arises. The CEL and BCE methods here are definitely a lot more consistent than the other two methods. Interestingly the CEL method had a lower standard deviation than the BCE method on all tested metrics. This is unexpected, as the hypothesis was that the BCE method would be a little bit more consistent. Something different can be noted when comparing the MSE and SL1 methods. The SL1 method seems to have a lower standard deviation for the CEL and BCE metric, whereas the MSE method has a lower standard deviation when comparing with the MSE and SL1 metrics. This suggests that the hypothesis about the SL1 loss was correct. Because the SL1 loss uses a L1 term for guesses that are very wrong, it will try more to not guess very wrong. The CEL and BCE metric are really sensitive to these very wrong guesses.

Loss function\Training function	**CEL**	**BCE**	**MSE**	**SL1**
CEL	[1.92,2.24]	[1.64,2.69]	[2.08,4.97]	[1.94,3.55]
BCE ($\times 10^{-2}$)	[7.46,8.51]	[6.48,10.03]	[7.84,16.47]	[7.24,12.43]
MSE ($\times 10^{-2}$)	[1.90,2.11]	[1.68,2.42]	[1.85,2.55]	[1.62,2.48]
SL1 ($\times 10^{-2}$)	[1.60,1.75]	[1.42,1.96]	[1.55,2.05]	[1.32,1.94]

Table 4.3: Minimum and maximum losses after training the model described in section 3.2 ten times for 3 epochs, using different loss functions. Visualized in an interval, thus the loss of every trained model was inside this interval.

What is interesting to see from the minimum and maximum values of the losses, is that the SL1 does seem to have achieved a really low training loss, which is better on every metric except when comparing with the CEL and BCE metrics and trained on those respective losses. This would imply that this loss function can converge really fast, if it is given the right initial weights. The problem when looking at the minimum, might of course be that it is just a coincidence. Now when looking at the maximum, both the MSE and SL1 losses have reached a very high maximum, it is therefore not consistent.

The same ten models trained, were also tested on a validation set, see table 4.4. This table contains the average of the losses of the ten models in all the metrics. Also a figure was made to better of this data, to maybe better illustrate the tendencies using boxplots. Note that this figure multiplies the BCE loss with 25, to be able to better compare the results.



Loss function\Training function	CEL	BCE	MSE	SL1
CEL	1.97 (0.090)	2.07 (0.314)	3.29 (1.069)	2.45 (0.699)
BCE ($\times 10^{-2}$)	7.61 (0.302)	7.93 (1.071)	11.49 (3.189)	8.91 (2.238)
MSE ($\times 10^{-2}$)	1.94 (0.063)	1.99 (0.224)	2.19 (0.269)	1.89 (0.338)
SL1 ($\times 10^{-2}$)	1.62 (0.046)	1.65 (0.165)	1.79 (0.195)	1.53 (0.254)

Table 4.4: Sample mean of validation losses after training the model described in section 3.2 ten times for 3 epochs, using different loss functions. In brackets is the sample standard deviation of the respective model. Figure contains boxplots of the same data, where the BCE is multiplied by 25 instead of 100.

From this table not much new information can be gathered. It is almost the same as table 4.2. The same was seen when comparing the minima and maxima. This is probably because the models were not trained for long enough to see any overfitting, so the validation loss behaved almost the same as the training loss did.

As mentioned before, after the training of each model, it was applied to the start of a sentence: ”how much ”. Some interesting results were found here. The sentence can of course be completed in any number of ways, but because the network was not trained for long enough, most of the time it repeats words. In English the word ”the” is probably one of the most common words. When looking at the models created using the BCE loss, six out of the ten were repeating this word constantly. These models were all the ones with a higher than average loss. It seems that this is a local minimum, which this loss gets stuck in often. The remaining four models were still repeating words, where two were repeating ”the said”.

When looking at the models created using the CEL function, the minimum loss was also achieved by repeating ”the said”. Repeating the word ”the” only happened three out of ten times here. However once it kept repeating ”and”.

Now the by SL1 and MSE trained models were really different. First to go over the MSE model, twice it did not make words at all and just returned spaces, which looks like a local minimum as well. The lowest loss here was achieved by making the following sentence:

how much and the manter the was and the sain the preat and the sain the preat

Afterwards it continued to repeat the last five words. Having tested this model only on the mentioned input, it is not proven that the output changes depending on the input. But with the values of the losses it returned, it is highly unlikely that it did not change. This is therefore probably the first model that actually tried to make sentences instead of just repeating the same words.

This was extended further by the models trained using the SL1 loss, where the model with the lowest loss produced the following output:

how much say that she was an anster to make at the seath of the stor with an and and the seave of the stor

After which it repeated the last nine words. While not all words are real English words, it did find some word combinations that make sense. For example the ”to make” part are two words that are likely to appear together. ”say that” and ”she was”, are also examples of this.

4.5. Overfitting

In the previous section, the models were trained often, but not trained for a long time. In practice, there is a good chance that the model will be trained for a long time. If the loss keeps decreasing there is not really a point to stop training. However when training for long enough, the model should eventually start overfitting. When this happens, the model will start to learn patterns that only appear in the training dataset and will therefore hurt the predictability of the model. Overfitting can have multiple causes, with the main one being, the use of a small dataset. Therefore, in this section, the models were trained for a long time on a small dataset. This should cause the model to eventually overfit as it has seen the training data very often and with enough parameters can just guess from this, the best way to model those specific sentences. When this happens the validation loss should increase as these sentences were not used for training. It might be interesting to see which models start to overfit sooner. To achieve this the models were trained on 8000 sentences randomly selected from the first 10000 sentences of the dataset. The validation set contained another 1000 different sentences from the same subset of the dataset. This was done for a total of 50 epochs, so each sentence in the training set will have been passed into the model fifty times. After each epoch the training loss and validation loss were calculated and stored. Then after the fiftieth epoch, five sentences were given to the network to see what it would return: ”how ”, ”how much ”, ”how much more ”, ”a”, ”when was the last time that ”.

4.5.1. CEL and BCE

The CEL and BCE loss behaved very similarly, with the other two doing something different. Both the CEL and BCE loss did not manage to overfit in 50 epochs. This is really interesting, as this many epochs is not normally used. The validation loss did however almost reach a minimum as the decrease in loss each epoch is decreasing. Keep here in mind that the BCE loss was not very consistent and with this being trained only once, it is possible that in some scenarios the training process will be different.

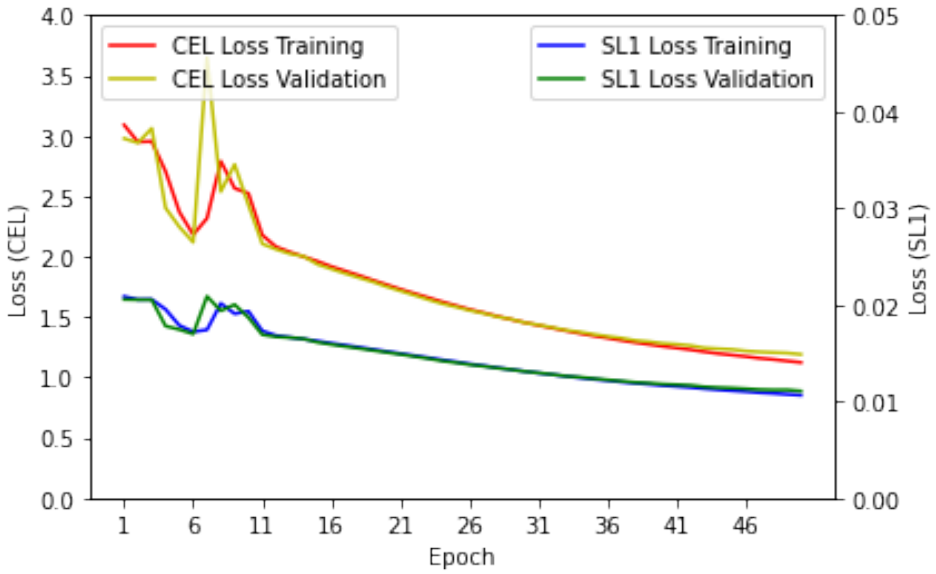


Figure 4.1: Losses from training the network with CEL function for 50 epochs on a dataset of size 8000.
The CEL metric and the SL1 metric are visualized.
Both the validation losses and the training losses were plotted after each epoch.
A learning rate of 0.001 was used and optimizer Adam.

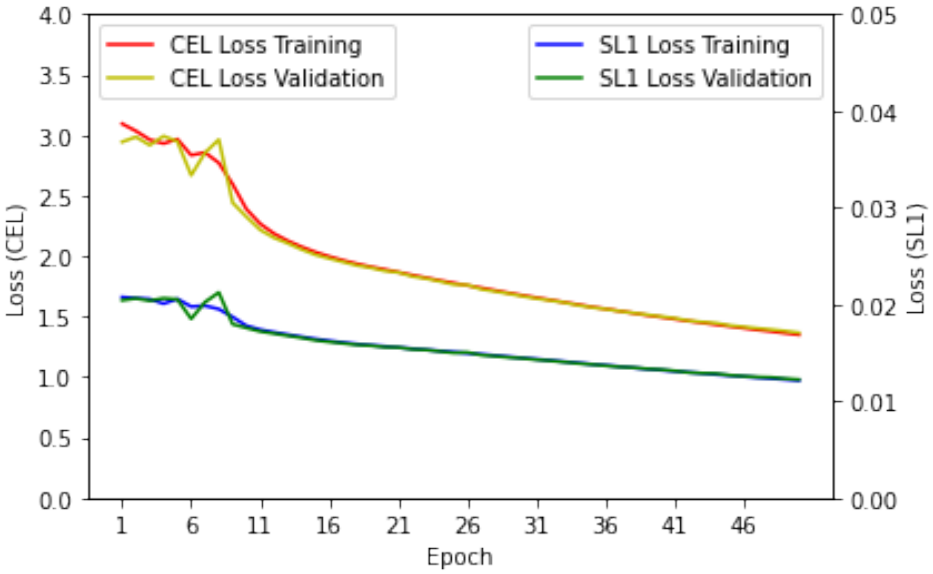


Figure 4.2: Losses from training the network with BCE loss function for 50 epochs on a dataset of size 8000.
The CEL metric and the SL1 metric are visualized.
Both the validation losses and the training losses were plotted after each epoch.
A learning rate of 0.001 was used and optimizer Adam.

Interestingly from this figure it becomes clear that in the beginning the losses had some trouble, where the CEL loss seems to fluctuate the most. After around 11 epochs, the training and validation losses are decreasing each epoch. Precise results of the last epoch can be found in table 4.5.

Looking at the sentences made by the models after the last epoch, it is clear that the BCE is still stuck in some local minimum. For each of the sentences, it returned a sentence that after a few words kept repeating ”of the same” or ”of the stairs”. For example, when given ”how ”, the model produced:

how i should not see her hand of the same of the same of the same of the same of the same of the same of the same

The same can be said about the sentences that the CEL model returned, but here it is a little bit more reasonable. The sentences continue a bit longer with the non-repeating part. Also the repeating part is a bit longer, however it does seem quite random. Good examples are the sentences that begin with ”how ”, where each one looked like this:

how i was a little too man be well, and the world with the countess of the starral some of the countess of the starral to the children in the countenance of the countess of the starral with the childr

So both of these models seem to reach a certain local minimum, where certain combinations of characters are learned, but not structures of sentences. Because the loss of the ”@” sign is reduced to account for only 10% of the total loss, the model will not be punished for letting the sentence go on forever. One could only look at the part that is not repeated and see a reasonable English sentence.

4.5.2. MSE and SL1

The MSE and SL1 losses behaved very different from the other two losses, but behaved similarly to each other. They did manage to overfit very fast and both had a way higher lowest validation loss. When analyzing the figures and data from these to models, keep in mind that they were really inconsistent. Therefore it could be that if trained multiple times, the models would be able to achieve lower losses. Nevertheless it is a good example of overfitting, see figures 4.3 and 4.4. The MSE network starts earlier after 17 epochs, where it reached its lowest CEL loss over the validation set of 1.599. The for SL1 loss optimized model obtained this minimum after 24 epochs, where it had a CEL loss of 1.773. The minima for the SL1 metric were reached after 19 epochs, with a value of 0.01133, and after 25 epochs, with a value of 0.01136, respectively.

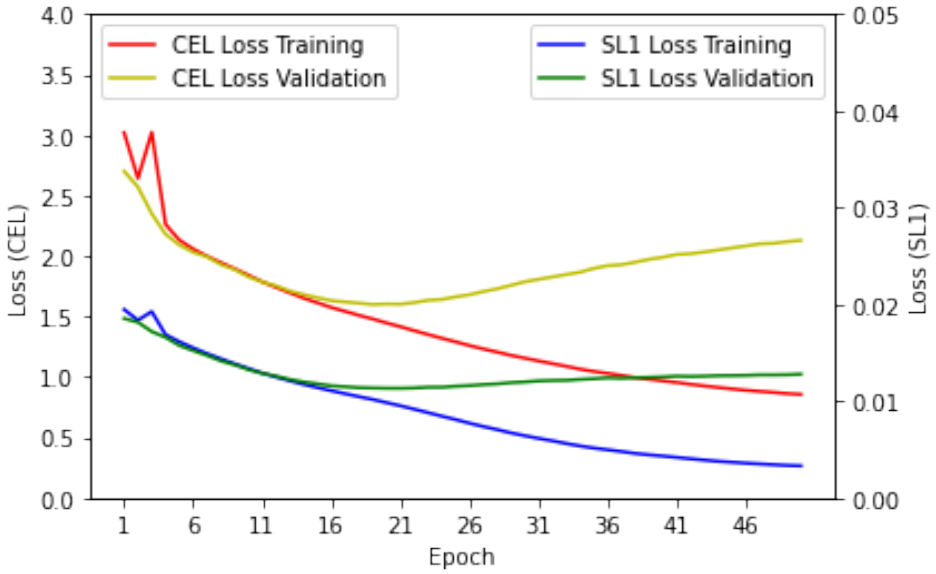


Figure 4.3: Losses from training the network with MSE loss function for 50 epochs on a dataset of size 8000.
The CEL metric and the SL1 metric are visualized.
Both the validation losses and the training losses were plotted after each epoch.
A learning rate of 0.001 was used and optimizer Adam.

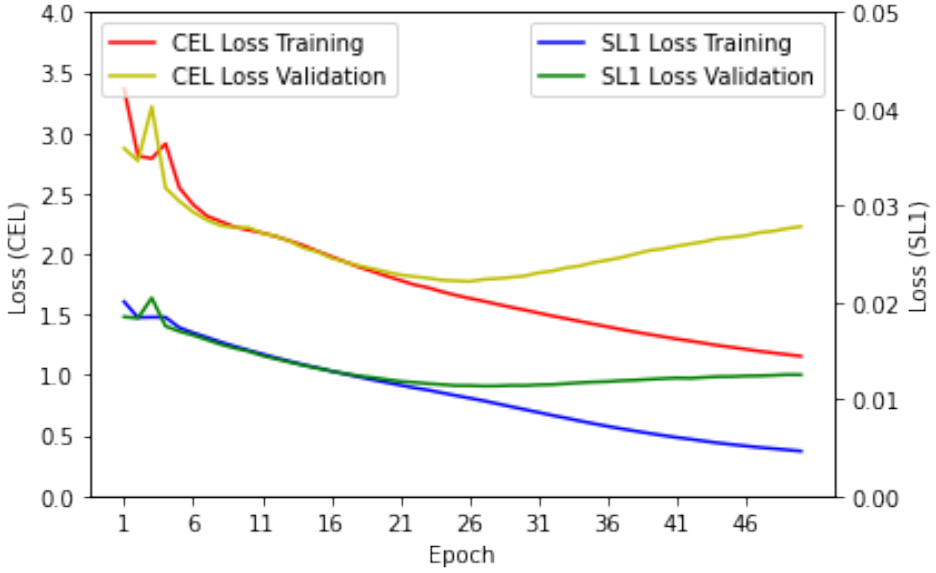


Figure 4.4: Losses from training the network with BCE loss function for 50 epochs on a dataset of size 8000.
The CEL metric and the SL1 metric are visualized.
Both the validation losses and the training losses were plotted after each epoch.
A learning rate of 0.001 was used and optimizer Adam.

When looking at the sentences, that were made when the model was trained for fifty epochs, some interesting characteristics appear. These models had a considerably lower training loss at this point when compared with the CEL and BCE models. It now makes sentences that look really close to English, but do not really have any meaning. Also the sentences end more often with the end of sentence character. The SL1 model still had some trouble with the words and some words are not spelled correctly, however this may be due to bad scanning of older books. Because when older books are scanned often ”e” can become a ”c” as the line in the middle is not always clear. An example is the following sentence:

how is it theord and so these at itmass we's belicves that are to herself, "i'm but an ingerveve could be see more than beant...." "is that i'm about the world you have been done and manting this levi

Note that this sentence ends with half a word, this is because the sentences were only created for 200 characters, the model could continue for longer, maybe forever if not stopped. This is the sentence which began with "how ", which of course is a short start. There are multiple sentences in the dataset that start like this. Theoretically it could choose one of these sentences and return that. However each sentence in the training set is weighted evenly and therefore it makes a combination of the sentences which does not turn out to be English. Interestingly adding one word, making the input: "how much ", caused the model to return: "*how much i'm mad and deal, it for?"*@". So this is a sentence that ends very quickly. Another interesting result is when only "a" was given as input, this returned the only correct English sentence: "*and the prince went out of the room.."*@". This sentence appears in one of the books, but here only the "and the prince went" part is written.

The MSE model did finish more sentences and made less spelling mistakes. For example the sentence it made from "how " was:

how is it the-reast and benind the heart of her."@

Which does not make much sense and does contain two words that do not exist in English. It does start the same as the SL1 model does, which is probably because "how is it" is the most common combination starting with "how ". Both afterwards continue with "the", which does not fit well. Again the sentence starting with just an "a" is interesting:

and she drew madeline toward the door. @

This is a correct English sentence, which is an exact copy of one that is present in one of the books. It is not clear why it chose this one, as it only appears once.

4.5.3. Final results

When looking purely at the results after fifty epochs, table 4.5 can be made. One can note that here the MSE metric has the lowest training loss by far. Because the model was trained only once, this can be a coincidence.

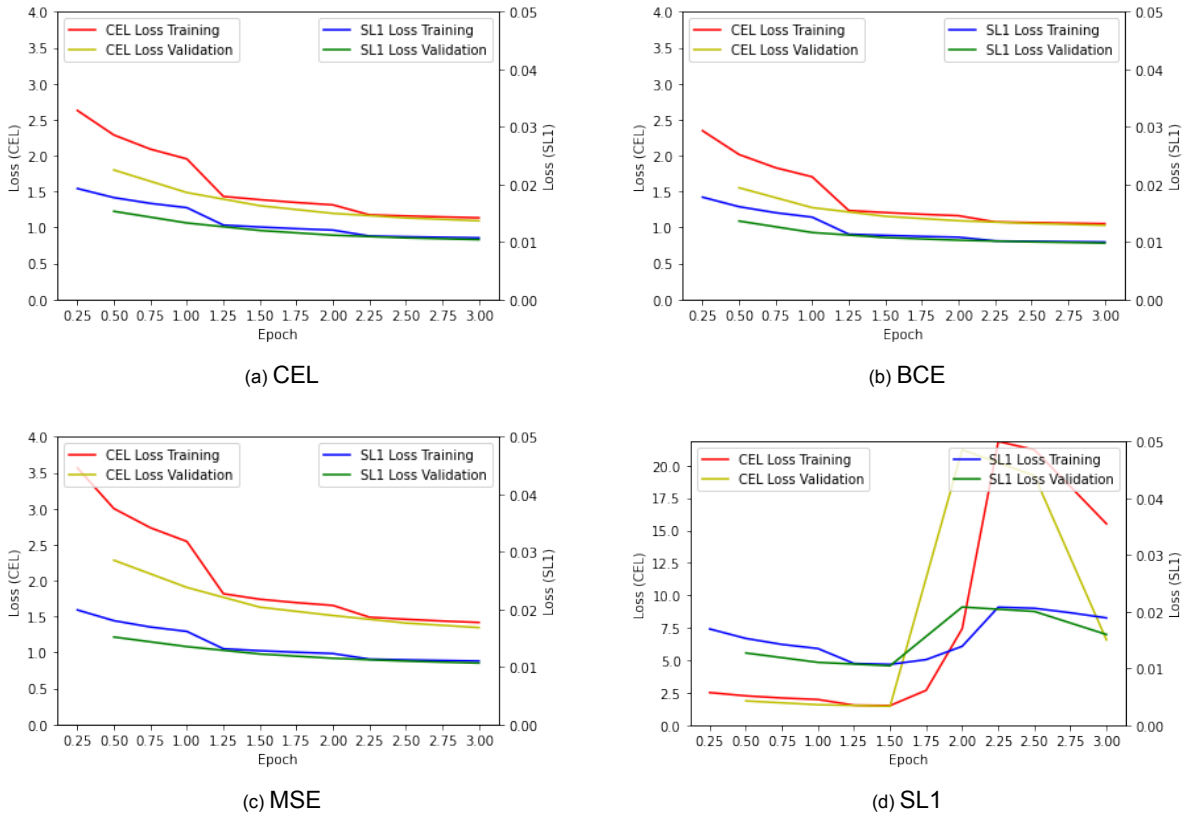
	Training CEL	Validation CEL	Training SL1 ($\times 10^{-2}$)	Validation SL1 ($\times 10^{-2}$)
CEL	1.123	1.190	1.065	1.108
BCE	1.349	1.365	1.215	1.224
MSE	0.856	2.130	0.332	1.279
SL1	1.155	2.228	0.463	1.252

Table 4.5: Training and validation losses of all four models after 50 epochs

The validation losses are of course a lot worse for the MSE and SL1 models as they overfitted. However the minimum CEL losses they achieved were also a lot higher than those of the CEL and BCE model. Interestingly the MSE model did manage to get a lower SL1 loss than the BCE model.

4.6. Whole dataset

In all previous sections only a part of the dataset was used. To train the model on the entire dataset took a lot of time, so the results might be comparable to the overfitting section. However the key difference here is that it trains on many different sentences of course, which causes overfitting to be really difficult. Like said before the training set contains 502,708 sentences, of which 402,168 were randomly selected as training set and 50,270 as validation set. The training of the models was done for 3 epochs, which took around 3 hours per model. Because some models were inconsistent, the same starting weights were used for each model, which were still taken from a uniform distribution. This still could favor some models, but will probably take away some variance. Training took long and only noting the training loss once per epoch, would have been too little data. Therefore the training loss was calculated four times per epoch. Because the validation set is also large now, the validation loss took a while to compute. This loss was only calculated twice per epoch.



(a) CEL (b) BCE

(c) MSE (d) SL1

Figure 4.5: Training and validation losses, calculated with the CEL and SL1 loss metrics. Training was performed with respect to the loss function below each figure. The whole dataset described in section 4.1 was used for 3 epochs. A learning rate of 0.001 was used with the Adam optimizer. Training losses were plotted four times per epoch and validation losses twice per epoch.

	Training CEL	Validation CEL	Training SL1 ($\times 10^{-2}$)	Validation SL1 ($\times 10^{-2}$)
CEL	1.135	1.093	1.071	1.040
BCE	1.053	1.030	0.998	0.980
MSE	1.418	1.346	1.102	1.066
SL1	15.51	6.565	1.891	1.598

Table 4.6: Training and validation losses of all four models after 3 epochs

It can be seen, from figures 4.5, that the CEL, BCE and MSE models trained all very similarly. With the CEL and BCE models, both having a lower loss than the MSE model at every measured moment. Now the SL1 model did something strange, see figure 4.5d. For some reason the losses started to increase here after 1.5 epochs. No real reason can be found for this, apart from it probably getting a few non-representative sentences, where it trained the wrong way. With a dataset of this size, the probability for this is really low. Therefore another explanation is probably correct. It is worth noting that when making this report, the loss would sometimes do this seemingly random.

Comparing the BCE and CEL models, it is interesting that here the BCE model actually achieved lower training and validation losses at all tested moments. None of the models managed to overfit, as 3 epochs not enough time for this to happen, which can be concluded from section 4.5.

After training the model, the same five sentences used in section 4.5, were given to the model to see what they would return. All of the models had the problem of repeating the same things again. Starting with the CEL model, the model kept repeating ”of the sea” or ”of the consequences”, for the first three sentences. Starting with only an ”a”, did make it so that the sentence stopped at some point, but it still repeated itself a little bit. Starting with ”when was the last time that”, produced the only decent sentence:

when was the last time that he had seen him and the strength of the constant street which he had seen

him and the son of a man who had been struck on his hands and she said that he was so far as the s

Which does not make much sense, but it does not really repeat the same words.

For the BCE model, which had the lowest losses, did a little worse if judged by just how it looks like a normal sentence. For example, when using the same start as before, it produced:

when was the last time that the most interesting state of the power of the power of the sense of the princess was seen to be able to prove that he had not seen him and the servant of the congregation

Where for all the other starts it just started repeating words after three of four unique ones.

The MSE model repeated for the first four tested sentences ”of the street”, after around four unique words. The last sentence produced the following:

when was the last time that he was to be so that the countess was the same time that he was to be so that the countess was to be so that he was to be so that he was to be so that he was to be so that

Now the SL1 model produced random characters for the most part, which was to be expected when looking at the losses.

5

Conclusion

First of all, comparing the CEL and BCE loss functions. From section 3.2.3 the hypothesis was that the BCE function would allow less diversity in the returned probabilities as the gradients increase by a lot for each incorrect guess. For this application, having a character appear in a sentence next to another character, does not mean that any of the other characters should never appear there as well. So trying to make every character have probability zero apart from the targeted character, which will have probability one, is not necessary. However it did cause the model to be able to train very fast at the start of training, as can be seen from the results that trained for a low number of epochs. But this caused the model to mainly select the most common words, which did not depend much on the sentence, as can be seen in section 4.4. The model repeats words in all of the cases. It did reach a lower loss than the CEL function did in section 4.6, albeit still repeating words. It seems that, when given the right starting weights, it can converge very fast to a local minimum. Even when trying to overfit the model by training it for a long time on a smaller dataset, it returned mainly repeating sentences. If the goal is to make real looking English sentences, this loss function does not seem to work well.

The CEL function was also prone to repeating the same things, as can be seen in chapter 4. However a little bit less so than the BCE loss. When comparing CEL analytically to the MSE and SL1 losses, it can be seen that because the CEL function uses logarithms, the derivative gets really simple. It causes some terms to disappear. In this case, this means that if a guess is low for a targeted character, the gradients will be bigger. Now this causes the same things to happen in the BCE loss, as making errors is punished hard. Therefore, when looking at the numerical results of section 4.5, the validation losses reach a lower minimum. On average the model performs better on sentences it has never seen before. But the training loss is higher, as it has not learned many of the training sentences word for word, which the other two methods did do. Now depending on the task, this is not a bad thing. For predicting sentences it has not seen, it definitely performs better, but this is at the cost of having actual English being produced. Maybe when trained longer, on a model with more parameters, it can learn better.

Now comparing the two discussed models to the MSE and SL1 models, the main difference analytically is of course the appearance of the $S(x_{i,c})$ or $S(x_{i,a})$ part in the derivatives. This part causes the guesses that are low to have a low impact on the gradient. It therefore matters less if a guess is very incorrect, as it has a low guessed probability. So the MSE and SL1 models can guess higher, which makes them less good on average, but they are better at fitting the training set. This can be seen as they overfit earlier and their training losses were lower.

Because the SL1 loss is the same as the MSE loss if the guess is close to being correct, the method should analytically behave the same once the model has been trained for some time. Because after some time of training the model should have learned the basic rules and can reliably guess within 0.5 distance. This was not seen from section 4.5, as here the SL1 method did not reach anything close to the training loss of the MSE model. Now this experiment was done only once, so it could be that this is because the initial weights were unfavorable for the SL1 loss. However in section 4.6, the same can be seen. Here of course the training loss started to increase a lot, which is weird. An explanation could be that because many of the characters were already being predicted quite well, it started to focus on those that were not yet. It therefore changed the model a lot into predicting those characters more. This

is however just speculation and no definitive proof has been found, as to why this happened. During the making of this report, it did however happen more than once, so that suggests it has something to do with the loss. From section 4.4, it becomes clear that it is a little bit more consistent when compared with the MSE model when trained for a short time. This is probably because it can take bigger steps at the beginning and fix some of the extreme cases more quickly because the gradient will not be dependent on the difference between the correct value and the guessed value, but be 1 or -1 which is the maximum difference.

Just an interesting find, which was not meant to be researched here, is that the size of the training set was less important than the amount of epochs it trained for. The models from section 4.5 took around an hour to train, whereas the models from section 4.6 took three hours to train. Looking at the training losses, which admittedly have been calculated with a different training set, they are very comparable. The BCE loss, did preform better, but this has probably to do with the fact that it converges fast if trained for a low number of epochs. When looking at the sentences produced by the models which were trained for few epochs on a large dataset, one can note that they are worse than the sentences produced when trained for 50 epochs on a smaller dataset. Now if the whole dataset was used for 50 epochs, the models would probably have been better of course, but if no time is available to do so, taking a smaller dataset, might be beneficial, as it can be trained for more epochs, in which it converges faster to reasonable results.

6 Discussion

Having done the calculations on a personal computer, the training times were sometimes quite long. This meant that some of the calculations were not done completely, or not done at all. Training the network on the entire dataset for a few more epochs could maybe yield some interesting results and maybe some different characteristics appear when done so. Also, when comparing the consistency, the result was that some loss functions were less consistent. To train these models for a long time, will probably yield different results if done a few times with randomized starting values. In this report only a part of the dataset was used. Furthermore the figure that was made with the whole dataset was made only once, which could yield different results if done multiple times.

Another key limitation is the fact that only one dataset was used. While quite large, it contains books from different time periods, and with very different writing styles. For other problems, a dataset with more similar inputs, might also give different results. Some loss functions could behave differently in that case. Mainly the mean squared error and the smooth L1 loss can in that case probably perform better. Also for other problems, like for other languages, different loss function may preform better, although this seems unlikely.

One of the main problems with neural networks since the first ones were developed, was that it is very hard to understand why these models work. With millions or even billions of parameters, it is almost impossible to describe what each of them does. In this report, an attempt has been made to understand the gradients of each one. Which in turn, should give some insight in what the parameter itself does. However analytically it still is a bit of speculation.

Like mentioned before in this report, a little insight into a specific problem can maybe cause some things to change. For example, the loss functions mentioned are used here in their simplest form. The only difference being that the loss over the ”@” characters is calculated separately from the other characters. With knowledge into the problem, loss functions can be altered such that certain conditions are always met. This would mean that the calculated derivatives in this report are mostly useless. Also certain conditions can be implemented into the optimizer in a way that it cannot do certain steps if the condition no longer holds. This could mean that different steps are taken and maybe certain local minima avoided or that some other local minima are harder to avoid.

One thing that was not tried for this report, is using multiple loss functions. Seeing the MSE and SL1 losses converge quickly sometimes, but not be able to get out of local minima, would suggest that they may work as initial loss functions. Then if a local minimum is found, the CEL or BCE losses can be used to get out of it. Then afterwards maybe the MSE can again be used, and so on. It would be interesting to see if such a method could work. Also one could formulate a function, which uses multiple of the tested functions. Either with a coefficient or, like the SL1 loss, with a restriction. The derivatives would be easy to calculate with the results found in this report.

Comparing the models made in this report to other NLP models, for example the GPT-3[2]. There are of course a lot of differences. For one the current most used models use an architecture known as transformers, which works very differently from LSTM. However these do often still end with a linear layer, so the results of this report can still be applied to those networks. The main differences however are the size of the network and the size of the training set. This specific network, the GPT-3 model, has 175 billion parameters. While the largest network in this report did not even have 10 million parameters,

9.2 million to be precise. Of course this means that the network cannot come close to what is achieved by larger companies. But nevertheless some results can be applied to these larger models as well, as most of these larger models, like the GPT-3 model, do use the cross entropy loss as well.

Larger models also behave differently when looking at certain scenarios, such as overfitting. A larger model will of course be able to differentiate between more patterns, which causes the model to be able to pick up on more nuanced patterns. This can make sure that the final model will have a lower minimum validation loss, when compared with a smaller model. For this report it means that some of the repeating behavior can be attributed to this. It is possible that the model is not large enough to predict better than what it does at this moment. Still, different some loss functions do seem to repeat more often than others.

A

Derivations

A.1. Derivation for equation 3.3

$$H(p,q) = -\frac{1}{N}\sum_{i=1}^{N}\log(q(x_i)) \text{ where } q(x_i) = \frac{\exp(x_{i,y_i})}{\sum_{c=1}^{C}\exp(x_{i,c})} = \frac{\exp((Ah_i)_{y_i})}{\sum_{c=1}^{C}\exp((Ah_i)_c)}$$

$$\frac{\partial q(x_i)}{\partial A_{ab}} = \begin{cases} -(h_i)_b \frac{\exp(x_{i,y_i})\cdot\exp(x_{i,a})}{\left(\sum_{c=1}^{C}\exp(x_{i,c})\right)^2} & \text{if } y_i \neq a \\ \frac{(h_i)_b \exp(x_{i,y_i})\sum_{c=1,c\neq a}^{C}\exp(x_{i,c})}{\left(\sum_{c=1}^{C}\exp(x_{i,c})\right)^2} & \text{if } y_i = a \end{cases} \quad \text{(using the quotient rule)}$$

$$\begin{aligned} \frac{\partial H(p,q)}{\partial A_{ab}} &= -\frac{1}{N}\sum_{i=1}^{N}\frac{1}{q(x_i)}\cdot\frac{\partial q(x_i)}{\partial A_{ab}} \\ &= -\frac{1}{N}\sum_{i=1}^{N}\frac{\sum_{c=1}^{C}\exp(x_{i,c})}{\exp(x_{i,y_i})}\cdot\frac{\partial q(x_i)}{\partial A_{ab}} \\ &= -\frac{1}{N}\sum_{i=1}^{N}(h_i)_b\left(-\frac{\exp(x_{i,a})}{\sum_{c=1}^{C}\exp(x_{i,c})}\mathbf{1}_{y_i\neq a} + \frac{\sum_{c=1,i\neq a}^{N}\exp(x_{i,c})}{\sum_{c=1}^{C}\exp(x_{i,c})}\mathbf{1}_{y_i=a}\right) \end{aligned}$$

A.2. Derivation for equation 3.4

$$\frac{\partial (o_i)_a}{\partial (W_{xo})_{ab}} = \frac{\partial}{\partial (W_{xo})_{ab}} \left[\frac{1}{1+\exp(-\alpha_{i,a})} \right] = \frac{-(x_i)_b \exp(-\alpha_{i,a})}{(1+\exp(-\alpha_{i,a}))^2}$$
$$= \frac{(x_i)_b}{\exp(\alpha_{i,a}) + \exp(-\alpha_{i,a}) + 2}$$

$$\frac{\partial q(x_i)}{\partial (h_j)_a} = \begin{cases} 0 & \text{if } i \neq j \\ \exp\left(x_{j,y_j}\right) \frac{\sum_{c=1}^{C} A_{ay_j} \exp\left(x_{j,y_j}\right) - \sum_{c=1}^{C} A_{ac} \exp\left(x_{j,c}\right)}{\left(\sum_{c=1}^{C} \exp\left(x_{j,c}\right)\right)^2} & \text{if } i = j \end{cases}$$

$$\frac{\partial H(p,q)}{\partial (h_j)_a} = -\frac{1}{N} \sum_{i=1}^{N} \frac{1}{q(x_j)} \frac{\partial q(x_i)}{\partial (h_j)_a}$$
$$= -\frac{1}{N} \frac{\sum_{c=1}^{C} (A_{y_j a} - A_{ca}) \exp\left(x_{j,c}\right)}{\sum_{c=1}^{C} \exp\left(x_{j,c}\right)}$$

$$\frac{\partial (h_i)_a}{\partial (o_i)_a} = \tanh((c_i)_a)$$

$$\frac{\partial H(p,q)}{\partial (W_{xo})_{ab}} = \sum_{i=1}^{N} \frac{\partial H(p,q)}{\partial (h_i)_a} \frac{\partial (h_i)_a}{\partial (o_i)_a} \frac{\partial (o_i)_a}{\partial (W_{xo})_{ab}}$$
$$= -\frac{1}{N} \sum_{i=1}^{N} \frac{\sum_{c=1}^{C} (A_{ay_i} - A_{ac}) \exp\left(x_{i,c}\right)}{\sum_{c=1}^{C} \exp\left(x_{i,c}\right)} \cdot \tanh((c_i)_a) \cdot \frac{(x_i)_b}{\exp(\alpha_{i,a}) + \exp(-\alpha_{i,a}) + 2}$$

B

Books

Book title	Author
Alice's Adventures in Wonderland	Lewis Carroll
A Modest Proposal	Jonathan Swift
The Angel and the Demon	Timothy Shay Arthur
Anna Karenina	Leo Tolstoy
Around the World in Eighty Days	Jules Verne
A Study In Scarlet	Arthur Conan Doyle
The Awakening and Selected Short Stories	Kate Chopin
Autobiography of Benjamin Franklin	Benjamin Franklin
Beowulf	unknown
Beyond Good and Evil	Friedrich Nietzsche
Birds from North Borneo	Max C. Thompson
The Souls of Black Folk	W. E. B. Du Bois
The Call of the Wild	Jack London
Carmilla	Joseph Sheridan Le Fanu
A Christmas Carol	Charles Dickens
Common Sense	Thomas Paine
The Complete Works of William Shakespeare	William Shakespeare
Crime and Punishment	Fyodor Dostoevsky
David Copperfield	Charles Dickens
The American Diary of a Japanese Girl	Yone Noguchi
A Doll's House	Henrik Ibsen
The History of Don Quixote	Miguel de Cervantes Saavedra
The Picture of Dorian Gray	Oscar Wilde
Dracula	Bram Stoker
Dubliners	James Joyce
The Works of Edgar Allan Poe, Volume 2	Edgar Allan Poe
Emma	Jane Austen
Frankenstein	Mary Wollstonecraft (Godwin) Shelley
Great Expectations	Charles Dickens
The Great Gatsby	F. Scott Fitzgerald
Anne of Green Gables	Lucy Maud Montgomery
Grimms' Fairy Tales	Jacob Grimm and Wilhelm Grimm
Gulliver's Travels	Jonathan Swift
The Happy Prince	Oscar Wilde
Heart of Darkness	Joseph Conrad
The Hound of the Baskervilles	Arthur Conan Doyle
Adventures of Huckleberry Finn	Mark Twain (Samuel Clemens)
The Importance of Being Earnest	Oscar Wilde
Jane Eyre	Charlotte Brontë
Japanese Girls and Women	Alice Mabel Bacon
Life And Letters Of John Gay (1685-1732)	Lewis Melville
Josefine Mutzenbacher	Felix Salten
The Jungle Book	Rudyard Kipling
The Brothers Karamazov	Fyodor Dostoyevsky
The King in Yellow	Robert W. Chambers
The King James Bible	N/A
The Legend of Sleepy Hollow	Washington Irving
Leviathan	Thomas Hobbes
Little Women	Louisa May Alcott
Meditations	Marcus Aurelius
Metamorphosis	Franz Kafka
The Essays of Montaigne, Complete	Michel de Montaigne
Les Misérables	Victor Hugo
Moby-Dick; or The Whale	Herman Melville

Book title	Author
Moby Language, II	Grady Ward
The Count of Monte Cristo	Alexandre Dumas, père
The Mysterious Affair at Styles	Agatha Christie
Narrative of the Life of Frederick Douglass	Frederick Douglass
Notes from the Underground	Fyodor Dostoyevsky
The Odyssey	Homer
Old Granny Fox	Thornton W. Burgess
Oliver Twist	Charles Dickens
Persuasion	Jane Austen
Peter Pan	James M. Barrie
Pride and Prejudice	Jane Austen
The Republic	Plato
The Romance of Lust A classic Victorian erotic novel	Anonymous
The Confessions of Saint Augustine	Saint Augustine
Sense and Sensibility	Jane Austen
The Adventures of Sherlock Holmes	Arthur Conan Doyle
Simple Sabotage Field Manual	Office of Strategic Services
The Strange Case Of Dr. Jekyll And Mr. Hyde	Robert Louis Stevenson
The Iliad	Homer
A Pickle For The Knowing Ones	Lord Timothy Dexter
The Philippines A Century Hence	Jose Rizal
The Prince	Nicolo Machiavelli
The Prophet	Kahlil Gibran
The Scarlet Letter	Nathaniel Hawthorne
Thus Spake Zarathustra	Friedrich Nietzsche
The Time Machine	H. G. Wells
Treasure Island	Robert Louis Stevenson
Second Treatise of Government	John Locke
Twenty Thousand Leagues under the Sea	Jules Verne
A Tale of Two Cities	Charles Dickens
Ulysses	James Joyce
Uncle Tom’s Cabin	Harriet Beecher Stowe
The Kama Sutra of Vatsyayana	Vatsyayana
Walden	Henry David Thoreau
War and Peace	Leo Tolstoy
The War of the Worlds	H. G. Wells
Winnie-the-Pooh	A. A. Milne
The Wonderful Wizard of Oz	L. Frank Baum
Wuthering Heights	Emily Brontë
The Yellow Wallpaper	Charlotte Perkins Gilman

C

Code

```
#%%

import nltk
import numpy as np
import torch
import torch.nn as nn
from tqdm import tqdm
import random
import pandas as pd
import os

#%%

#Reads the books and separates the sentences, while removing unwanted characters
def txt_to_list(path, sentencing = nltk.sent_tokenize, wording = nltk.tokenize.RegexpTokenizer(r'[A-Za-z_]+').tokenize, lower = False, split=True, maxlen = 0, minlen=0):
    try:
        with open(path,"r", errors="ignore") as f:
            text = f.read()
    except:
        print(path)
        raise UnicodeDecodeError
    if lower:
        text = text.lower()
    if split:
        sentences = []
        text = text.split('\n')
        for sent in text:
            sentences += sentencing(sent)
    else:
        sentences = sentencing(text)
    words = []
    for sent in sentences:
        ToAdd = wording(sent)
        if len(ToAdd) >= minlen and (len(ToAdd) <= maxlen or maxlen == 0):
            words.append(' '.join(ToAdd))
    return words

#%%

#Reading all the books, putting all the sentences into a list and removing any unwanted characters.
ArgBooks = {"lower": True, "split": False, "minlen": 5, "maxlen": 50, "wording": nltk.tokenize.RegexpTokenizer(r"[A-Za-z!,.:?'\"-]+").tokenize}

books = []
for book in tqdm(os.listdir(r'Data/Books_Adjusted')):
    text = txt_to_list(r"Data/Books_Adjusted/"+book, **ArgBooks)
```

```
    books += text
#-------------------------------------------------------------------
#Filling all the sentences with the "@" character and creating the dictionaries that map each character to a number
maxlen = len(max(books, key=len))
minlen = len(min(books, key=len))
print(f"maxlen: {maxlen},    minlen: {minlen}")
for i in tqdm(range(len(books))):
    while len(books[i])<maxlen:
        books[i] += '@'
chars = set(''.join(books))
int2char = dict(enumerate(chars))
char2int = {char: ind for ind, char in int2char.items()}
print(f"Number of sentences: {len(books)}")
books = np.array(books)

#%%

#Splitting the dataset into a training set, a validation set and a test set
#right now it would only use the first 10,000 sentences
numbers = np.array(range(len(books)))[:10000]
random.Random(123).shuffle(numbers)
val_numbers, test_numbers, train_numbers = np.split(numbers, [len(numbers)//10, 2*(len(numbers)//10)])

train_input_seq = []
train_target_seq = []
val_input_seq = []
val_target_seq = []
test_input_seq = []
test_target_seq = []

for i in tqdm(numbers):
    inp = [char2int[character] for character in books[i][:-1]]
    tar = [char2int[character] for character in books[i][1:]]
    if i in test_numbers:
        test_input_seq.append(inp)
        test_target_seq.append(tar)
    elif i in val_numbers:
        val_input_seq.append(inp)
        val_target_seq.append(tar)
    else:
        train_input_seq.append(inp)
        train_target_seq.append(tar)

dict_size = len(char2int)
print(f"number of characters: {dict_size}")
#-----------------------------------------------------------
#Creating a function that gets as input a sequence of numbers and creates a matrix of standard basis vectors
def one_hot_encode(sequence, dict_size, seq_len, batch_size):
    features = np.zeros((batch_size, seq_len, dict_size), dtype=np.float32)
    for i in range(batch_size):
        for u in range(seq_len):
            features[i, u, sequence[i][u]] = 1
    return features

device = torch.device("cuda")

#%%

#Data has to be given to the model in batches, also prepares the data such that it is ready to be used by the model.
class Seq_Loader():
    def __init__(self, input_seq, target_seq, batchsize,
                 seed = None, shuffle = False):
        self.input = input_seq.copy()
        for k, seq in enumerate(self.input):
            seq.append(k)
        self.shuffle = shuffle
        self.batchsize = batchsize
```

```
        self.n = 0
        self.target = target_seq.copy()
        self.shuffled = self.input.copy()
        if seed is not None:
            self.random = random.Random(seed)
        else:
            self.random = random

    def __iter__(self):
        if len(self.shuffled) == 0:
            self.shuffled = self.input.copy()
        return self

    def __next__(self):
        if len(self.shuffled) == 0:
            self.shuffled = self.input.copy()
            raise StopIteration
        if self.input == self.shuffled and self.shuffle:
            self.random.shuffle(self.shuffled)
        outinp = []
        outtarg = []
        self.n = 0
        while len(self.shuffled) > 0 and self.n < self.batchsize:
            inp = self.shuffled.pop()
            targ = self.target[inp[-1]]
            outinp.append(inp[:maxlen-1])
            outtarg.append(targ[:maxlen-1])
            self.n += 1
        outinp = one_hot_encode(outinp,dict_size,maxlen-1,len(outinp))
        return torch.FloatTensor(outinp), torch.IntTensor(outtarg)

#-------------------------------------------------------------------------
#Creating the individual data loaders, where the training data loader is shuffled each time,
    but the others are not.
DL_train = Seq_Loader(train_input_seq, train_target_seq, 128, shuffle=True,
                seed = 123)
DL_val = Seq_Loader(val_input_seq, val_target_seq, 64, shuffle=False,
                seed = 123)
DL_test = Seq_Loader(test_input_seq, test_target_seq, 64, shuffle=False,
                seed = 123)
for inp, targ in DL_train:
    print(f"Input shape: {list(inp.shape)},     target shape: {list(targ.shape)}")
    break

#%%

#Class of the main LSTM model
class Model(nn.Module):
    def __init__(self, input_size, output_size, hidden_dim, n_layers, model = nn.LSTM,
                 optimizer=torch.optim.Adam, crit="MSE", lr=0.01, beta = 0.25):
        super(Model, self).__init__()
        self.hidden_dim = hidden_dim
        self.n_layers = n_layers
        self.lstm = model(input_size, hidden_dim, n_layers, batch_first=True)
        self.fc = nn.Linear(hidden_dim, output_size)
        self.optimizer = optimizer(self.parameters(), lr=lr)
        self.criterion = crit
        self.beta = beta

    #Forward pass of the model, first going through the lstm layer and then a linear layer (
    fc)
    def forward(self, x):
        out, hidden = self.lstm(x)
        out = self.fc(out.contiguous().view(-1, self.hidden_dim))
        out = out.to(device)
        return out,hidden

    #Function that given a dataloader, calculates the total loss across the data in the
    loader for all implemented metrics.
    def validate(self, DL_val, no_at):
        total_loss_mse = 0
```

```
180         total_loss_cel = 0
181         total_loss_bce = 0
182         total_loss_sl1 = 0
183         n = 0
184         for batch, target in DL_val:
185             input_seq = batch.to(device)
186             target_seq = target.to(device)
187             output, hidden = self(input_seq)
188             total_loss_mse += self.BCE(output, batch, target, target_seq, no_at, lossf=nn.
    MSELoss()).item()
189             total_loss_cel += self.CEL(output, batch, target, target_seq, no_at).item()
190             total_loss_bce += self.BCE(output, batch, target, target_seq, no_at,lossf=nn.
    BCELoss()).item()
191             total_loss_sl1 += self.BCE(output, batch, target, target_seq, no_at,lossf=nn.
    SmoothL1Loss(beta=self.beta)).item()
192             n += 1
193         return [total_loss_cel/n, total_loss_mse/n, total_loss_bce/n, total_loss_sl1/n]
194
195     #Function that calculates the loss for the MSE, BCE and SL1 losses. Where first a Softmax
     function is used.
196     def BCE(self,output, batch, target, target_seq, no_at, lossf=nn.BCELoss()):
197         output = nn.Softmax(dim=-1)(output)
198         targ = torch.Tensor(one_hot_encode(target, dict_size,maxlen-1,batch.shape[0])).to(
    device)
199         output = output.view(batch.shape[0],batch.shape[1],-1)
200         if no_at:
201             woAt = target_seq != char2int["@"]
202             if (woAt==True).all():
203                 loss = lossf(output[woAt], targ[woAt])
204             else:
205                 loss1 = lossf(output[woAt], targ[woAt])
206                 loss2 = lossf(output[~woAt], targ[~woAt])
207                 loss = 0.9*loss1+0.1*loss2
208         else:
209             loss = lossf(output, targ)
210         return loss
211
212     #Function that calculates the loss for the CEL.
213     def CEL(self,output, batch, target, target_seq, no_at):
214         if no_at:
215             woAt = (target_seq.view(-1).long() != char2int["@"]).to('cpu')
216             woAt = torch.BoolTensor(woAt)
217             lossf = nn.CrossEntropyLoss()
218             if (woAt==True).all():
219                 loss = lossf(output[woAt], target_seq.view(-1).long()[woAt])
220             else:
221                 loss1 = lossf(output[woAt], target_seq.view(-1).long()[woAt])
222                 loss2 = lossf(output[~woAt], target_seq.view(-1).long()[~woAt])
223                 loss = 0.9*loss1+0.1*loss2
224         else:
225             loss = nn.CrossEntropyLoss()(output, target_seq.view(-1).long())
226         return loss
227
228     #Function that trains the network, given a training data loader and a validation data
    loader.
229     def adapt(self, DL_train, DL_val, n_epochs, no_at = False, inb=True):
230         self.train()
231         n = 0
232         Train_Losses = []
233         Val_losses = []
234         for epoch in (range(1, n_epochs + 1)):
235             pr = 25
236             total_mse_loss, total_cel_loss, total_bce_loss, total_sl1_loss = 0, 0, 0, 0
237             for batch, target in DL_train:
238                 input_seq = batch.to(device)
239                 target_seq = target.to(device)
240                 self.optimizer.zero_grad() #First make sure the gradients are zero such that
    they can be cacluted.
241                 output, hidden = self(input_seq) #Forward pass, uses the above "forward"
    function.
242                 #Calculates the loss depending on the chosen criterion.
```

```
243                 if self.criterion == "MSE":
244                     loss = self.BCE(output, batch,target, target_seq, no_at, lossf=nn.MSELoss
    ())
245                     total_mse_loss += loss.item()
246                 elif self.criterion == "CEL":
247                     loss = self.CEL(output, batch,target, target_seq, no_at)
248                     total_cel_loss += loss.item()
249                 elif self.criterion == "BCE":
250                     loss = self.BCE(output, batch,target, target_seq, no_at, lossf=nn.BCELoss
    ())
251                     total_bce_loss += loss.item()
252                 elif self.criterion == "SL1":
253                     loss = self.BCE(output, batch,target, target_seq, no_at, lossf=nn.
    SmoothL1Loss(beta=self.beta))
254                     total_sl1_loss += loss.item()
255                 else:
256                     print(self.criterion)
257                     raise ValueError("Criterion not yet implemented.")
258                 loss.backward() #Calculates the gradients.
259                 self.optimizer.step() #Does the step according to the optimizer, with the
    calculated gradients.
260                 self.eval() #Makes sure no more gradients are calculated and no steps are
    taken
261                 #Calculates the losses using the other metrics
262                 if self.criterion != "MSE":
263                     loss = self.BCE(output, batch,target, target_seq, no_at, lossf=nn.MSELoss
    ())
264                     total_mse_loss += loss.item()
265                 if self.criterion != "CEL":
266                     loss = self.CEL(output, batch,target, target_seq, no_at)
267                     total_cel_loss += loss.item()
268                 if self.criterion != "BCE":
269                     loss = self.BCE(output, batch,target, target_seq, no_at, lossf=nn.BCELoss
    ())
270                     total_bce_loss += loss.item()
271                 if self.criterion != "SL1":
272                     loss = self.BCE(output, batch,target, target_seq, no_at, lossf=nn.
    SmoothL1Loss(beta=self.beta))
273                     total_sl1_loss += loss.item()
274                 #Prints and saves the losses every quarter of an epoch if enabled by "inb"
    variable
275                 if ((n*DL_train.batchsize*100)//len(train_numbers)>=pr and pr < 100) and inb:
276                     print(f"sentence: {n*DL_train.batchsize} of {len(train_numbers)}")
277                     Train_Losses.append([total_cel_loss/n,total_mse_loss/n,total_bce_loss/n,
    total_sl1_loss/n])
278                     print(f'Epoch: {epoch}/{n_epochs}...', end=' ')
279                     print(f"T Loss: (CEL):  {round(total_cel_loss/n,4)}...", end=' ')
280                     print(f"(MSE):  {round(total_mse_loss/n,4)}...", end=' ')
281                     print(f"(BCE):  {round(total_bce_loss/n,4)}...", end=' ')
282                     print(f"(SL1):  {round(total_sl1_loss/n,4)}...")
283                     #Only saves and prints validation losses every half of an epoch
284                     if pr == 50:
285                         val_loss = self.validate(DL_val, no_at)
286                         Val_losses.append(val_loss)
287                         print(f"Validation Loss:     {val_loss}...")
288                     pr += 25
289                 self.train()
290                 n += 1
291
292         self.eval()
293         #Prints and saves losses at the end of an epoch
294         val_loss = self.validate(DL_val, no_at)
295         Train_Losses.append([total_cel_loss/n,total_mse_loss/n,total_bce_loss/n,
    total_sl1_loss/n])
296         Val_losses.append(val_loss)
297         print(f'Epoch: {epoch}/{n_epochs}...', end=' ')
298         print(f"T Loss: (CEL):  {round(total_cel_loss/n,4)}...", end=' ')
299         print(f"(MSE):  {round(total_mse_loss/n,4)}...", end=' ')
300         print(f"(BCE):  {round(total_bce_loss/n,4)}...", end=' ')
301         print(f"(SL1):  {round(total_sl1_loss/n,4)}...")
302         print(f"Validation Loss:     {val_loss}...")
```

```
303             self.train()
304             n=0
305         return Train_Losses, Val_losses
306 
307 #Two functions that are needed to read the output of the model and finnish an input sentence.
308 def predict(model, character):
309     character = np.array([[char2int[c] for c in character]])
310     character = one_hot_encode(character, dict_size, character.shape[1], 1)
311     character = torch.from_numpy(character)
312     character = character.to(device)
313     out, hidden = model(character)
314     prob = nn.functional.softmax(out[-1], dim=0).data
315     char_ind = torch.max(prob, dim=0)[1].item()
316     return int2char[char_ind], hidden
317 
318 def sample(model, out_len, start='hey'):
319     model.eval()
320     start = start.lower()
321     chars = [ch for ch in start]
322     size = out_len - len(chars)
323     for i in range(size):
324         char, h = predict(model, chars)
325         chars.append(char)
326     return ''.join(chars)
327 
328 #Creating one instance of the model
329 lstm = Model(input_size=dict_size, output_size=dict_size, model=nn.LSTM,
330             hidden_dim=1500, n_layers=1, lr=0.001, crit="CEL", beta=0.5)
331 lstm = lstm.to(device)
332 training_loss = []
333 val_loss = []
334 
335 #%%
336 
337 #Training the model
338 t, v = lstm.adapt(DL_train, DL_val, 50, no_at=True, inb=False)
339 training_loss += t
340 val_loss += v
341 
342 #%%
343 
344 #Part that makes and saves the data for multiple runs of the training of the model using the
    different loss functions.
345 for method in ["CEL", "BCE", "MSE", "SL1"]:
346     tosave = ""
347     for i in tqdm(range(10)):
348         testm = Model(input_size=dict_size, output_size=dict_size, model=nn.LSTM,
349                     hidden_dim=1500, n_layers=1, lr=0.001, crit=method, beta=0.5)
350         testm.to(device)
351         t, v = testm.adapt(DL_train, DL_val, 3, no_at=True, inb=False)
352         tdf = pd.DataFrame(t, index=["ep_1","ep_2","ep_3"], columns=["CEL","MSE","BCE","SL1"
    ])
353         vdf = pd.DataFrame(v, index=["ep_1","ep_2","ep_3"], columns=["CEL","MSE","BCE","SL1"
    ])
354         if i==0:
355             traindf = tdf.copy()
356             valdf = vdf.copy()
357         else:
358             traindf = traindf.join(tdf,rsuffix=str(i))
359             valdf = valdf.join(vdf,rsuffix=str(i))
360         tosave += sample(testm, 150, start='how much ')+"\n"
361     traindf.to_csv(f"Report/{method}_1500_3epoch_10x_test.csv", sep=';')
362     valdf.to_csv(f"Report/{method}_1500_3epoch_10x_val.csv", sep=';')
363     with open(f"Report/{method}_1500_3epoch_10x.txt", "w") as f:
364         f.write(tosave)
365 
366 #%%
367 
368 #Part that calculates the losses when comparing the different values of beta for the SL1
    function.
369 for beta in [0.75,0.5,0.25,0.1]:
```

```
    tosave = ""
    for i in tqdm(range(3)):
        testm = Model(input_size=dict_size, output_size=dict_size, model=nn.LSTM,
                    hidden_dim=1000, n_layers=1, lr=0.001, crit="SL1", beta=beta)
        testm.to(device)
        t, v = testm.adapt(DL_train, DL_val, 3, no_at=True)
        tdf = pd.DataFrame(t, index=["ep_1","ep_2","ep_3"], columns=["CEL","MSE","BCE","SL1"
    ])
        vdf = pd.DataFrame(v, index=["ep_1","ep_2","ep_3"], columns=["CEL","MSE","BCE","SL1"
    ])
        if i==0:
            traindf = tdf.copy()
            valdf = vdf.copy()
        else:
            traindf = traindf.join(tdf,rsuffix=str(i))
            valdf = valdf.join(vdf,rsuffix=str(i))
        tosave += sample(testm, 150, start='how much ')+"\n"
    traindf.to_csv(f"Report/SL1_{str(beta)[2:]}_3epoch_3x_test.csv", sep=';')
    valdf.to_csv(f"Report/SL1_{str(beta)[2:]}_3epoch_3x_val.csv", sep=';')
    with open(f"Report/SL1_{str(beta)[2:]}_3epoch_3x.txt", "w") as f:
        f.write(tosave)

#%%

import matplotlib.pyplot as plt
#Creates a figure of the training process
def plot_losses(training_loss, val_loss, filename, ppe = 1):
    nt=len(training_loss)+1
    nv=len(val_loss)+1
    fig, ax1 = plt.subplots()
    ax1.plot(np.array(range(1,nt))/ppe, np.array(training_loss).T[0], 'r')
    ax1.plot(np.array(range(1,nv))*(nt-1)/(nv-1)/ppe, np.array(val_loss).T[0], 'y')
    plt.ylim(0,max(max(np.array(training_loss).T[0]),max(np.array(val_loss).T[0]),4))
    plt.xlabel("Epoch")
    plt.ylabel("Loss (CEL)")
    plt.xticks(ticks=np.array(range(1,nt,max(nt//10,1)))/ppe)
    ax2 = ax1.twinx()
    ax2.plot(np.array(range(1,nt))/ppe, np.array(training_loss).T[3], 'b')
    ax2.plot(np.array(range(1,nv))*(nt-1)/(nv-1)/ppe, np.array(val_loss).T[3], 'g')
    plt.ylim(0,max(max(np.array(training_loss).T[3]),max(np.array(val_loss).T[3]),0.05))
    plt.ylabel("Loss (SL1)")
    ax1.legend(["CEL Loss Training", "CEL Loss Validation"], loc=2)
    ax2.legend(["SL1 Loss Training", "SL1 Loss Validation"],loc=1)
    plt.savefig(f"Report/Figures/{filename}.png", bbox_inches='tight')

#%%

for method in tqdm(["CEL", "BCE", "MSE", "SL1"]):
    testm = Model(input_size=dict_size, output_size=dict_size, model=nn.LSTM,
                    hidden_dim=1500, n_layers=1, lr=0.001, crit=method, beta=0.5)
    testm.to(device)
    lstm=torch.load(f"Models/Baseline_3epoch_All.pt")
    testm.lstm.weight_ih_l0 = lstm.lstm.weight_ih_l0
    testm.lstm.weight_hh_l0 = lstm.lstm.weight_hh_l0
    testm.lstm.bias_hh_l0 = lstm.lstm.bias_hh_l0
    testm.lstm.bias_ih_l0 = lstm.lstm.bias_ih_l0
    testm.fc.weight = lstm.fc.weight
    testm.fc.bias = lstm.fc.bias
    testm.optimizer = torch.optim.Adam(testm.parameters(), lr=0.001)
    t, v = testm.adapt(DL_train, DL_val, 50, no_at=True, inb=False)
    traindf = pd.DataFrame(t, columns=["CEL","MSE","BCE","SL1"])
    valdf = pd.DataFrame(v, columns=["CEL","MSE","BCE","SL1"])
    tosave = sample(testm, 200, start='how ')+"\n"
    tosave += sample(testm, 200, start='how much ')+"\n"
    tosave += sample(testm, 200, start='how much more ')+"\n"
    tosave += sample(testm, 200, start='a')+"\n"
    tosave += sample(testm, 200, start='when was the last time that ')
    traindf.to_csv(f"Report/{method}_1500_50epoch_1x_test_n.csv", sep=';')
    valdf.to_csv(f"Report/{method}_1500_50epoch_1x_val_n.csv", sep=';')
    with open(f"Report/{method}_1500_50epoch_1x_n.txt", "w") as f:
        f.write(tosave)
```

```
    filename = f"{method}_50epoch_1500_10000_n"
    plot_losses(t, v, filename)
    filepath = f"Models/{filename}"
    torch.save(testm, filepath+".pt")
    with open(filepath+"_c2i.txt", "w") as f:
        f.write(str(char2int))
    print(f"Done: {filepath}")

#%%

lstm = Model(input_size=dict_size, output_size=dict_size, model=nn.LSTM,
                    hidden_dim=1500, n_layers=1, lr=0.001, crit="BCE", beta=0.5)
lstm.to(device)

filename = "Baseline_3epoch_All"
#Saves the model such that it can later be used again.
filepath = f"Models/{filename}"
torch.save(lstm, filepath+".pt")
with open(filepath+"_c2i.txt", "w") as f:
    f.write(str(char2int))
print(f"Done: {filepath}")

#%%

for method in tqdm(["BCE", "SL1"]):
    testm = Model(input_size=dict_size, output_size=dict_size, model=nn.LSTM,
                    hidden_dim=1500, n_layers=1, lr=0.001, crit=method, beta=0.5)
    lstm=torch.load(f"Models/Baseline_3epoch_All.pt")
    testm.to(device)
    testm.lstm.weight_ih_l0 = lstm.lstm.weight_ih_l0
    testm.lstm.weight_hh_l0 = lstm.lstm.weight_hh_l0
    testm.lstm.bias_hh_l0 = lstm.lstm.bias_hh_l0
    testm.lstm.bias_ih_l0 = lstm.lstm.bias_ih_l0
    testm.fc.weight = lstm.fc.weight
    testm.fc.bias = lstm.fc.bias
    testm.optimizer = torch.optim.Adam(testm.parameters(), lr=0.001)
    t, v = testm.adapt(DL_train, DL_val, 3, no_at=True, inb=True)
    traindf = pd.DataFrame(t, columns=["CEL","MSE","BCE","SL1"])
    valdf = pd.DataFrame(v, columns=["CEL","MSE","BCE","SL1"])
    tosave = sample(testm, 200, start='how ')+"\n"
    tosave += sample(testm, 200, start='how much ')+"\n"
    tosave += sample(testm, 200, start='how much more ')+"\n"
    tosave += sample(testm, 200, start='a')+"\n"
    tosave += sample(testm, 200, start='when was the last time that ')
    traindf.to_csv(f"Report/{method}_1500_3epoch_All_test.csv", sep=';')
    valdf.to_csv(f"Report/{method}_1500_3epoch_All_val.csv", sep=';')
    with open(f"Report/{method}_1500_3epoch_All.txt", "w") as f:
        f.write(tosave)
    filename = f"{method}_3epoch_1500_All"
    plot_losses(t, v, filename, ppe = 4)
    filepath = f"Models/{filename}"
    torch.save(testm, filepath+".pt")
    with open(filepath+"_c2i.txt", "w") as f:
        f.write(str(char2int))
    print(f"Done: {filepath}")

#%%

#Saves the training loss and test loss if needed
filename = "CEL_books_50epoch_1500_10000"
with open(f"Models/{filename}.txt", 'w') as f:
    f.write(str(training_loss) + "\n"+ str(val_loss))
```

Bibliography

[1] Steven Bird, Ewan Klein, and Edward Loper. *Natural Language Processing with Python: Analyzing Text with the Natural Language Toolkit*. 1st ed. O'Reilly Media, 2009.

[2] Tom B. Brown et al. *Language Models are Few-Shot Learners*. 2020. DOI: 10.48550/ARXIV.2005.14165. URL: https://arxiv.org/abs/2005.14165.

[3] Jason Brownlee. *A gentle introduction to the rectified linear unit (ReLU)*. Aug. 2020. URL: https://machinelearningmastery.com/rectified-linear-activation-function-for-deep-learning-neural-networks/.

[4] Justin Domke. *Lecture notes for Statistical Machine Learning: Automatic Differentiation and Neural Networks*. 2010.

[5] Krzysztof Gajowniczek et al. "Semantic and Generalized Entropy Loss Functions for Semi-Supervised Deep Learning". In: *Entropy* 22.3 (2020), p. 334. DOI: 10.3390/e22030334.

[6] Ross Girshick. *Fast R-CNN*. 2015. DOI: 10.48550/ARXIV.1504.08083. URL: https://arxiv.org/abs/1504.08083.

[7] Alex Graves. *Generating Sequences With Recurrent Neural Networks*. 2013. DOI: 10.48550/ARXIV.1308.0850. URL: https://arxiv.org/abs/1308.0850.

[8] Sepp Hochreiter and Jürgen Schmidhuber. "Long Short-Term Memory". In: *Neural Computation* 9.8 (Nov. 1997), pp. 1735–1780. ISSN: 0899-7667. DOI: 10.1162/neco.1997.9.8.1735. eprint: https://direct.mit.edu/neco/article-pdf/9/8/1735/813796/neco.1997.9.8.1735.pdf. URL: https://doi.org/10.1162/neco.1997.9.8.1735.

[9] Daniel James Martin Jurafsky. *Speech and Language Processing An Introduction to Natural Language Processing, Computational Linguistics, and Speech Recognition*. 1st ed. PEARSON INDIA, 2021. URL: https://web.stanford.edu/~jurafsky/slp3/3.pdf#page=21.

[10] Diederik P. Kingma and Jimmy Ba. *Adam: A Method for Stochastic Optimization*. 2014. DOI: 10.48550/ARXIV.1412.6980. URL: https://arxiv.org/abs/1412.6980.

[11] C Olah. *Understanding LSTM Networks – colah's blog*. Aug. 2015. URL: http://colah.github.io/posts/2015-08-Understanding-LSTMs/.

[12] *PyTorch documentation — PyTorch 1.12 documentation*. 2022. URL: https://pytorch.org/docs/stable/index.html.

[13] Ning Qian. *On the Momentum Term in Gradient Descent Learning Algorithms*. 1999.

[14] Antônio H. Ribeiro et al. *Beyond exploding and vanishing gradients: analysing RNN training using attractors and smoothness*. 2019. DOI: 10.48550/ARXIV.1906.08482. URL: https://arxiv.org/abs/1906.08482.

[15] Sebastian Ruder. *An overview of gradient descent optimization algorithms*. 2016. DOI: 10.48550/ARXIV.1609.04747. URL: https://arxiv.org/abs/1609.04747.

[16] P Ruffwind. *Reverse-mode automatic differentiation: a tutorial - Rufflewind's Scratchpad*. Dec. 2016. URL: https://rufflewind.com/2016-12-30/reverse-mode-automatic-differentiation.

[17] *Uva deep learning course*. URL: https://uvadlc.github.io/.