HOCHSCHULE
MITTWEIDA
UNIVERSITY OF
APPLIED SCIENCES

PRESENTATION HANDOUT

Mr.
Thomas Davies

Transformer Neural Networks

An Introduction and Overview

2022

I. Contents

1 Recurrent Neural Networks

Recurrent Neural Networks (RNN) are state of the art models for learning on sequential data. For this matter we briefly introduce the concept of an RNN. Though RNNs have very specific flaws due to their architectural behavior, they help us understand the efforts which have been made to solve these RNN induced issues, which finally led to the development of Transformer Networks. First we define a RNN.

1.1 Introduction to Recurrent Neural Networks

Definition 1.1 Let $\boldsymbol{x}$ be a sequence of vectors, i.e. $\boldsymbol{x} = (\boldsymbol{x}_1, \boldsymbol{x}_2, \ldots, \boldsymbol{x}_t) = (\boldsymbol{x}_i)_{i=0}^{t \in \mathbb{N}}$ [1] with $\boldsymbol{x}_i \in \mathbb{R}^n$. A *Recurrent Neural Network* (RNN) is a neural network taking $\boldsymbol{x}$ as input and containing directed cycles when illustrated as a computational graph. Hence, such cycles can be expressed in terms of computations as *dynamic systems*[2].

$$\boldsymbol{s}_t = f(\boldsymbol{s}_{t-1}, \boldsymbol{x}_t; \boldsymbol{\Theta})$$

where $\boldsymbol{s}_t \in \mathbb{R}^m$ represents the node s at time t in the computational graph of the recurrent neural network and $\boldsymbol{x}_t \in \boldsymbol{x}$ the signal point at time t. In this notation $\boldsymbol{\Theta}$ represents all parameters we want to learn in $\boldsymbol{s}_t$. In regular feed forward neural networks we would consider this to be the weight vector $\boldsymbol{w}$ and the bias $\boldsymbol{b}$. Later we will define this parameter more precisely when going into more detail on RNNs.

From this definition we can tell that such a network uses the state of a neuron to calculate the next state. It is not uncommon to illustrate this using some sort of memory or delay cell for the redirection, such that the feedback happens after the next time step $t+1$. Figure 1.1 illustrates the basic idea of an RNN. Instead of $\boldsymbol{s}$ we use $\boldsymbol{h}$ representing a hidden neuron. At first glance the displayed RNN does not make much sense, but from the definition we can clearly identify it as an RNN which yields a hidden state $\boldsymbol{h}$.

[1] Note if you read different literature the notation might differ. It is also common to find the notation with curly brackets $\{\boldsymbol{x}_1, \boldsymbol{x}_2, \ldots, \boldsymbol{x}_t\}$ which could be confused with a set. Since set properties should not be applied on sequences (in sets equal elements are counted as one element and there is no order). To make the nature of the element more clear the sequence notation is chosen. Further note that we stack vectors of same size, such that $\boldsymbol{x}$ could also be written as a matrix $\boldsymbol{x}$ and sequences usually have an index like $\boldsymbol{x}_n = (\boldsymbol{x}_i)_{i=0}^{t}$. To be more consistent with the notation of machine learning we chose just to write $\boldsymbol{x}$ for the sequence. Be aware, at all times, $\boldsymbol{x}$ represents a sequence of vectors.

[2] A *dynamic system* is defined to be a set of quantities which contain all the necessary information from the past to be able to evaluate the systems future behavior [1, p.797]

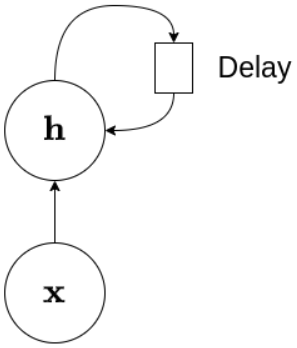

Figure 1.1: Basic RNN without Output

Remark 1.2 RNNs can be unfolded such that we get a feed forward neural network [2]. Figure 1.2 shows the basic RNN unfolded.

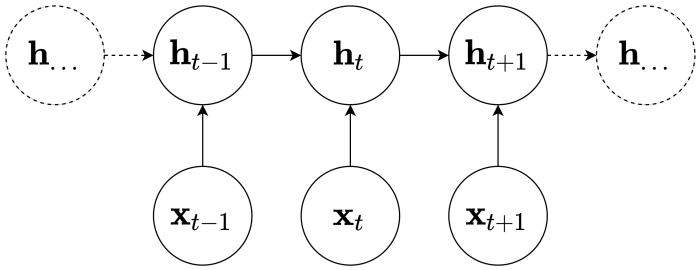

Figure 1.2: Basic RNN unfolded

Example 1.3 Let $\boldsymbol{x} = \{(\}\boldsymbol{x}_1, \boldsymbol{x}_2, \boldsymbol{x}_3)$ and $\boldsymbol{h}$ the hidden neuron of an RNN such that we have the following computation

$$\boldsymbol{h}_t = f(\boldsymbol{h}_{t-1}, \boldsymbol{x}_t; \boldsymbol{\Theta})$$

Since $\boldsymbol{x}_3$ is our furthest signal regarding time, we want to formulate our RNN with respect to $t = 3$. Hence

$$\boldsymbol{h}_3 = f(\boldsymbol{h}_2, \boldsymbol{x}_3; \boldsymbol{\Theta})$$

Due to the recursive nature of the RNN we can unfold each nested $\boldsymbol{h}_{t-1}$ until we reach the initial input $\boldsymbol{x}_1$.

$$\boldsymbol{h}_3 = f\Bigg(\underbrace{f\Big(\underbrace{f(\boldsymbol{h}_0, \boldsymbol{x}_1; \boldsymbol{\Theta})}_{=\boldsymbol{h}_1}, \boldsymbol{x}_2; \boldsymbol{\Theta} \Big)}_{=\boldsymbol{h}_2}, \boldsymbol{x}_3; \boldsymbol{\Theta} \Bigg)$$

Remark 1.4 Note that all the functions use the same trainable parameters $\boldsymbol{\Theta}$. This is called *parameter-sharing* [2, p.364]. It means that through all the time steps we use the same weights and biases. Also note that we need in the deepest level of our nested function the state h_0, which is not yet defined properly.

Definition 1.5 The first hidden state $\boldsymbol{h}_0$ of an RNN is called the *initial state*. The initial state is usually either initialized with $\mathbf{0}$ or randomly chosen.

Remark 1.6 Finding a good initial hidden state is a whole issue on its own [3]. For the system to be reliable with short-term sequences an optimal initial state must be found. This can be done by also learning the optimal initial state $\boldsymbol{h}^*(0)$.

Note that in Example 1.3 we unfolded $\boldsymbol{h}_3$ by unfolding each nested function up until we got to the initial sequence vector $\boldsymbol{x}_1$. This also allows us to formulate a general $\boldsymbol{h}_t$ more statically by using a single function evaluating all sequence vectors [2].

$$\boldsymbol{h}_t = g_t(\boldsymbol{x}_t, \boldsymbol{x}_{t-1}, \ldots, \boldsymbol{x}_1)$$

From this we can state a rather obvious property explicitly.

Property 1.7 *The time parameter t is not fixed over different input vectors $\boldsymbol{w}$ and $\boldsymbol{v}$, such that*

$$\boldsymbol{w} = (\boldsymbol{w}_i)_{i=1}^{t}$$
$$\boldsymbol{v} = (\boldsymbol{v}_i)_{i=1}^{t'}$$

with $t' \leq t$. The input space of the sequence data can consist of sequences with variable length.

For the time being we have ignored the trainable parameters Θ and want to describe this parameter. Let us define three different matrices $\boldsymbol{U} \in \mathbb{R}^{m \times n}, \boldsymbol{V} \in \mathbb{R}^{l \times m}, \boldsymbol{W} \in \mathbb{R}^{m \times m}$ for mapping the vectors between the states and two biases $\boldsymbol{b} \in \mathbb{R}^n$ and $\boldsymbol{c} \in \mathbb{R}^l$. Then we can explicitly formulate all components of the network as follows [2, p.370] [1, p.798]

$$\boldsymbol{a}_t = \boldsymbol{b} + \boldsymbol{W}\boldsymbol{h}_{t-1} + \boldsymbol{U}\boldsymbol{x}_t \tag{1.1}$$
$$\boldsymbol{h}_t = \tanh(\boldsymbol{a}_t) \tag{1.2}$$
$$\boldsymbol{o}_t = \boldsymbol{c} + \boldsymbol{V}\boldsymbol{h}_t \tag{1.3}$$
$$\hat{\boldsymbol{y}}_t = \text{softmax}(\boldsymbol{o}_t) \tag{1.4}$$

with

$$\tanh(x) = \frac{1 - e^{-2x}}{1 + e^{-2x}} \tag{1.5}$$

Note that there can also different activation functions can be used. Common functions are the sigmoid and RELU function.The prediction of the network is $\hat{\boldsymbol{y}}_t$ which is a vector over normalized probabilities. Note that by our description the network outputs a prediction after each time step. Figure 1.3 illustrates our enhanced model description.

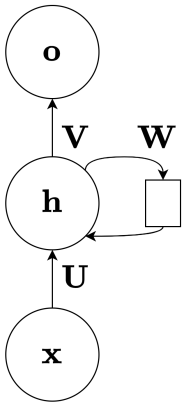

Figure 1.3: Basic RNN with Output and Weights

To actually be able to improve the performance of the RNN we need to also incorporate a loss function which measures the deviation of our prediction from the expected output.

$$L\Big((\boldsymbol{x}_i)_{i=0}^{t}, (\boldsymbol{y}_i)_{i=0}^{t}\Big) = \sum_t L_t \tag{1.6}$$

$$= -\sum_t \log \mathrm{p}\Big(y_t \mid (\boldsymbol{x}_i)_{i=0}^{t}\Big) \tag{1.7}$$

where $L(t)$ is the negative log-likelihood [2, p.371]. The negative log-likelihood is also known as the cross entropy. To optimize the neural network the partial derivative for the to be optimized components is calculated and these are updated accordingly. Still noteworthy is the fact we have to propagate over all time steps, which is why this process is also called *backpropagation through time*. Now we can complete our illustration of an RNN (Figure 1.4).

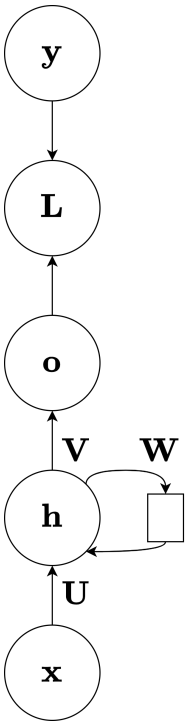

Figure 1.4: Basic RNN completed with Loss function

1.2 The Vanishing/Exploding Gradients Problem

Definition 1.8 Backpropagation through time (BTT) is the backpropagation of deep neural networks on an over time unfolded RNN.[3]

Recall that the property of parameter sharing makes each weight matrix the same over each time step t (or "fold" when thinking of unfolding the RNN).

For optimizing the network we want to calculate the gradient of our trainable parameters. For a complete overview of the derivatives of all trainable parameters refer to [2, p.374]. For the purpose of illustrating the issue with the vanishing/exploding gradient in RNNs we make use of our very basic RNN previously illustrated in figure 1.3. We will now have a closer look into the derivative of the weight matrix $\boldsymbol{W}$ with respect to the overall loss L. Therefore let t, t' be time steps such that $t' < t$. Since in our RNN architecture we produce an output with each time step, we can measure the loss at arbitrary time steps. We can therefore calculate the gradient of L with respect to $\boldsymbol{W}$ with

$$\Delta \boldsymbol{W} = \frac{\partial L(t', t)}{\partial \boldsymbol{W}} \tag{1.8}$$

[3] Note that there are also other methods of training RNNs such has *Real Time Recurrent Learning* (RTRL).

Since the overall loss is the sum of all losses for each time step, we have to sum all the losses for each input $\boldsymbol{x}_t$, therefore we can more precisely state

$$\Delta \boldsymbol{W} = \sum_{\mathcal{T}=t'}^{t} \frac{\partial L_{\mathcal{T}}}{\partial \boldsymbol{W}} \tag{1.9}$$

Doing the backpropagation on an RNN can be tricky on the first glance, therefore we illustrate the mechanism of the backpropagation in figure 1.5.

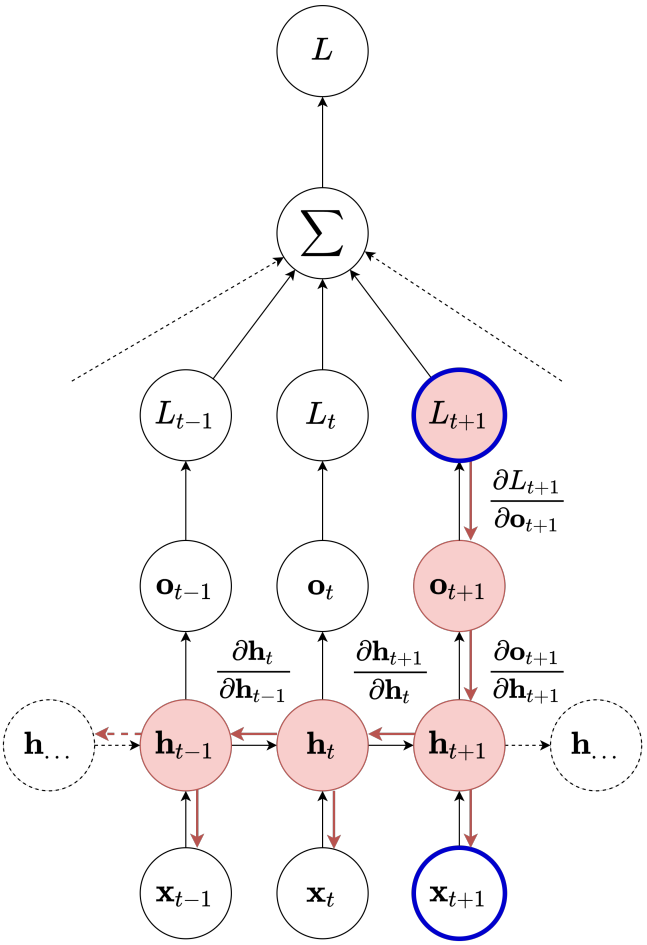

Figure 1.5: Backpropagation illustrated for input $\boldsymbol{x}_{t+1}$ and the respective loss L_{t+1} highlighted in blue. The backpropagation is highlighted in red.

We can observe that when backpropagating all predecessors have to be taken into account. Since the hidden states are encapsulated in each other due to their recursive construction we have to apply the chain rule for $\frac{\partial \boldsymbol{o}_{t+1}}{\partial \boldsymbol{h}_{t+1}}$. Hence,

$$\frac{\partial L_{\mathcal{T}}}{\partial \boldsymbol{W}} = \frac{\partial L_{\mathcal{T}}}{\partial \boldsymbol{o}_{\mathcal{T}}} \frac{\partial \boldsymbol{o}_{\mathcal{T}}}{\partial \boldsymbol{h}_{\mathcal{T}}} \frac{\partial \boldsymbol{h}_{\mathcal{T}}}{\partial \boldsymbol{W}} \tag{1.10}$$

Depending on the time span $t - t'$ the expression $\frac{\boldsymbol{h}_{\mathcal{T}}}{\partial \boldsymbol{W}}$ can be further expanded [4] [5]. Looking at figure 1.5 we see that the derivative can be unfolded to the product of the derivatives if $t - t'$ is greater than one, otherwise there are no predecessors. Therefore we write

$$\frac{\partial L_{\mathcal{T}}}{\partial \boldsymbol{W}} = \begin{cases} \frac{\partial L_{\mathcal{T}}}{\partial \boldsymbol{o}_{\mathcal{T}}} \frac{\partial \boldsymbol{o}_{\mathcal{T}}}{\partial \boldsymbol{h}_{\mathcal{T}}} \frac{\partial \boldsymbol{h}_{\mathcal{T}}}{\partial \boldsymbol{W}} & \text{if } t - t' = 1 \\ \frac{\partial L_{\mathcal{T}}}{\partial \boldsymbol{o}_{\mathcal{T}}} \frac{\partial \boldsymbol{o}_{\mathcal{T}}}{\partial \boldsymbol{h}_{\mathcal{T}}} \left(\prod_{k=t'+1}^{t} \frac{\partial \boldsymbol{h}_k}{\partial \boldsymbol{h}_{k-1}} \right) \frac{\partial \boldsymbol{h}_{t'}}{\partial \boldsymbol{W}_{t'}} & \text{if } t - t' > 1 \end{cases} \tag{1.11}$$

Note in the second case of equation 1.11 the expression $\boldsymbol{W}_{t'}$ highlights the time step k, though $\boldsymbol{W}$ is static because of the *parameter sharing* property. This notation is only used for illustration purposes. Further note that this model is extremely simple, since we don't have any deeper structure for the hidden states. In more elaborate literature on this topic also deeper architectural structures are considered.

The key to understand the behavior of the gradient is the expression $\prod_{k=t'+1}^{t} \frac{\partial \boldsymbol{h}_k}{\partial \boldsymbol{h}_{k-1}}$. If we do the partial derivative we get

$$\begin{aligned} \frac{\partial \boldsymbol{h}_k}{\partial \boldsymbol{h}_{k-1}} &= \frac{\partial \tanh(\boldsymbol{a}_k)}{\partial \boldsymbol{h}_{k-1}} && \text{(equation 1.2)} \\ &= \frac{\partial \tanh(\boldsymbol{b} + \boldsymbol{W}\boldsymbol{h}_{k-1} + \boldsymbol{U}\boldsymbol{x}_t)}{\partial \boldsymbol{h}_{k-1}} && \text{(equation 1.1)} \\ &= \boldsymbol{W} \cdot \frac{d}{d\boldsymbol{h}_{k-1}} \tanh(\boldsymbol{a}_k) && \text{(differentiation chain rule)} \end{aligned}$$

Note in the last line we left the derivative of tanh untouched, since we do not need to solve it for understanding the vanishing gradient. More important is the factor $\boldsymbol{W}$ yielded by applying the chain rule in the second line. Putting this expression in the product over our time span we get

$$\prod_{k=t'+1}^{t} \frac{\partial \boldsymbol{h}_k}{\partial \boldsymbol{h}_{k-1}} = \prod_{k=t'+1}^{t} \boldsymbol{W} \cdot \frac{d}{d\boldsymbol{h}_{k-1}} \tanh(\boldsymbol{a}_k) \tag{1.12}$$

$$= \boldsymbol{W}^{t-1} \prod_{k=t'+1}^{t} \frac{d}{d\boldsymbol{h}_{k-1}} \tanh(\boldsymbol{a}_k) \tag{1.13}$$

In equation 1.13 we see the matrix $\boldsymbol{W}$ to the power of $t-1$. Here we have either an exponential growth or decay of the matrix depending on the time span $t-t'$.

If we consider the weight matrix $W \in \mathbb{R}$ to be a real number, then it is easy to see, that if $W > 1$ the gradient will explode and if $W < 1$ then the gradient will vanish. Since $\boldsymbol{W}$ is a matrix the assessment is a bit more complicated. Here the *spectral radius* needs to be considered as we did earlier with W. Since the spectral radius is also a real number we can evaluate its value similarly. There are optimization techniques built around the idea to stabilize the weight matrix with respect to its spectral radius [6].

1.3 Recurrent Neural Network Types

There are different types of RNNs which differ on how the kind of output of the network and how the recurrence is constructed. These types are just briefly mentioned for getting a better intuition on the different architectures being used in these kinds of networks.

1. The first architectural type has already been discussed so far. The network outputs after every time step and the recurrence can be found from the hidden state to itself.

2. Also here the RNN outputs after every time step with the difference that the recurrence is constructed from the output to the hidden state.

3. Lastly the RNN only outputs after all time steps t have been applied. The recurrence is from the hidden state to itself.

Other than that we can also categorize RNNs on the type of input they receive and the kind of output they generate. While regular deep neural networks have a *one to one* architecture, RNNs can have different architectures. In the previous observations we used a *many to many* architecture, where from an input an instant output followed. There is also a *many to many* architecture where the output sequence is delay until the very last input has entered the network. Further there are *one to many* and *many to one* where from single inputs sequences are generated or vice versa. Figure 1.6 illustrates these architectures.

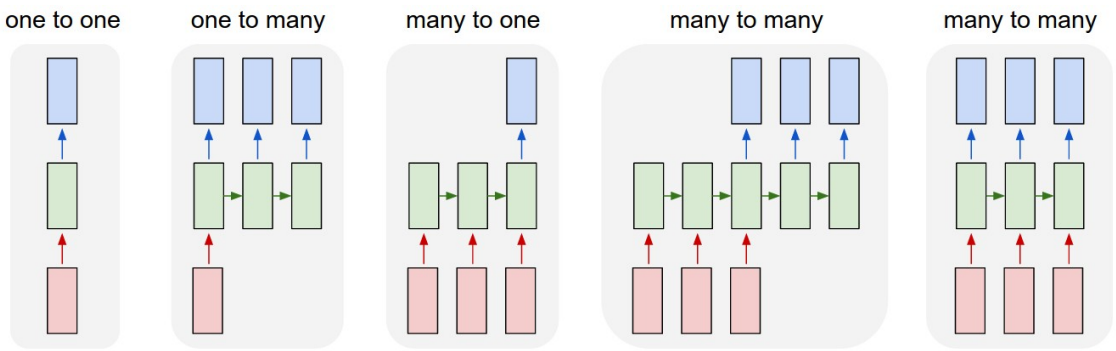

Figure 1.6: Different RNN Architectures with respect to the Input/Output [7]

1.4 Conclusion

To conclude the introduction to RNNs we will mention the challenges this architecture introduces. The recurrence is the elementary nature of the RNN which also yields an uncomfortable property. Since each successor state is reliant on its predecessor we have a chain of layers we need to calculate iteratively. Therefore it is not possible to parallelize over the sequence. This means we have $\mathcal{O}(t)$ computational complexity. Please note, that this does not include the complexity of each hidden layer itself, which can be composed of a complete feed forward network itself. Since we need for the backpropagation through time all the values of the hidden layers the memory complexity can be stated as $\mathcal{O}(t)$ as well. Though we need the backpropagation only for the training, recall that the training is the computational challenging part of creating these models. The computational complexity for the backpropagation through time obviously is also dependent on the time steps, such that we can state $\mathcal{O}(t)$. One of the biggest challenges is the already known problem of *vanishing* gradient (and also exploding gradients). Since these networks can grow fast to a very deep feed forward network when unfolded, we are challenged with these kind of problems here too. This property makes these basic RNNs very sensitive to long sequences. One solution to this is the introduction of *Long Short Term Memory cells* (LSTM-cells).

2 Long Short Term Memory

LSTM cells were introduced by Hochreiter and Schmidhuber in 1997 to solve the problem of vanishing gradients [8]. We will not focus on their original architecture. In 2000 Gers and Schmidhuber introduced an extension of the original LSTM cell including forget gates [9]. This chapter will focus on the latter architecture. A great summary on the improvement of this extension can be found here [10].

2.1 Architecture

In an LSTM-RNN each hidden state is replaced by a LSTM cell which is also referred as memory cell. These cells consist of different gates. A gate opens a flow of information if a condition is met. This condition is mathematically constructed such that it can be calculated instead of performing an algorithmic decision. If the value of a gate cell becomes zero it stops the information flow of another cell and the closer it is to one the more of the signal is passed through (equal to one means the signal is passed through completely) [10].

There are many different ways LSTM networks are illustrated in literature. In this introductory paper we will stick to an unfolded RNN to make the flow of information clearer. Other literature use the recursive illustration [8] [9] [10]. The notation in literature can be vastly different especially in the original papers. To be consistent with the former chapter we will stick as far as possible with already introduced notations. Figure 2.1 gives an overview of such a network.

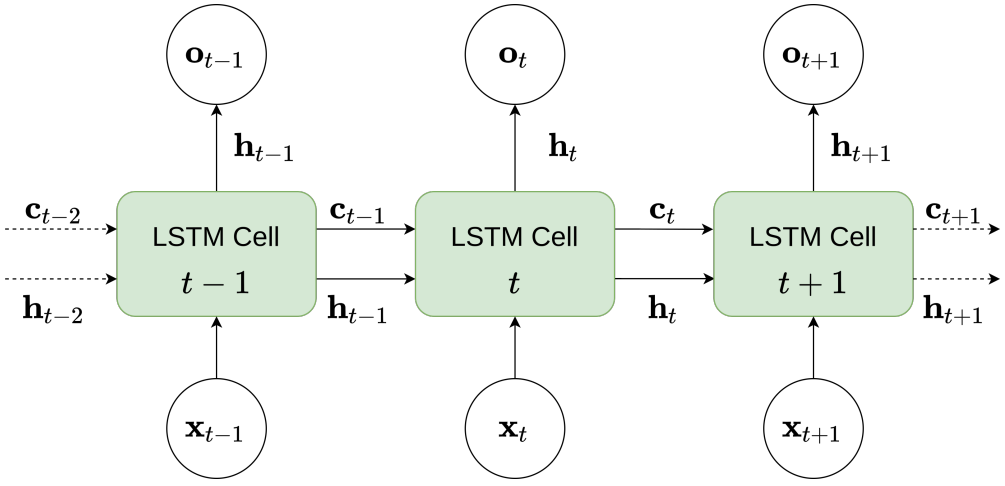

Figure 2.1: LSTM Network unfolded

Contrary to the regular RNN we now use the cell state $\boldsymbol{c}_{t-1}$ and the output $\boldsymbol{h}_{t-1}$ to

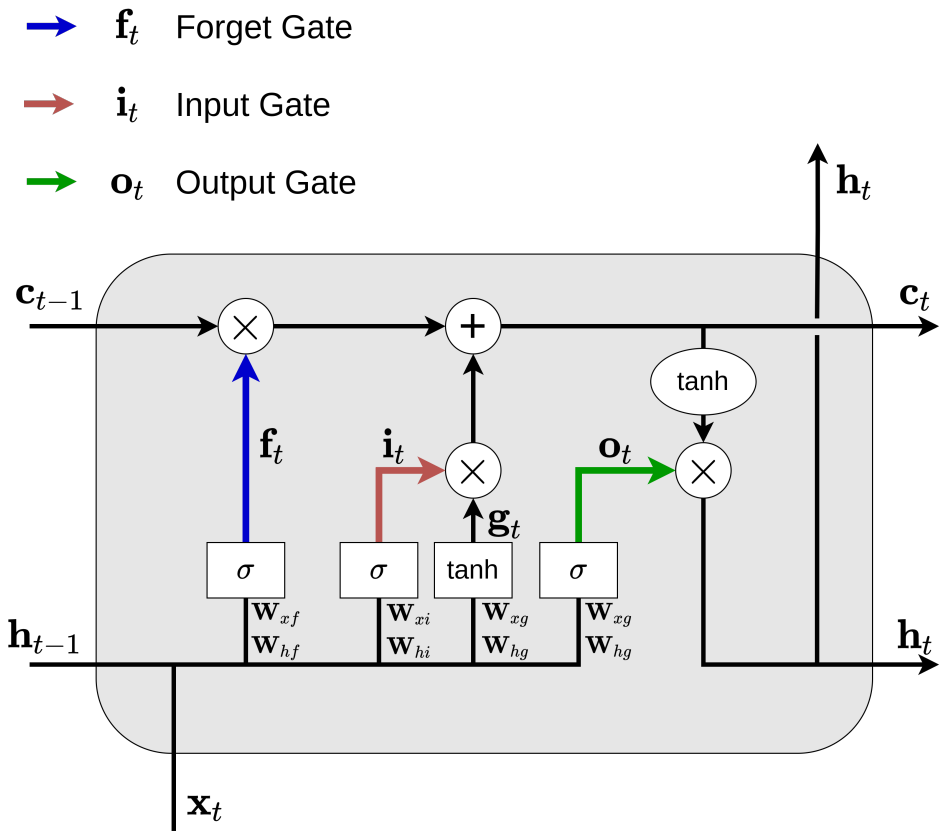

Figure 2.2: LSTM Cell with Gates Highlighted and Corresponding Weights

calculate $\boldsymbol{h}_t$. At this point we only briefly mention that $\boldsymbol{c}$ is the memory of the previous inputs. It takes importance of previous input into account and is filtered from input noise. The output $\boldsymbol{h}$ is basically what we have had so far and is the actual output of the hidden neuron or respectively RSTM cell.

Before further defining the connections between the cells we need to have a closer look at the gates [10] used in such a memory cell, because these are detrimental for understanding the cell state and the output.

Definition 2.1 [10]

1. The **Forget Gate** $\boldsymbol{f}_t$ determines how big the influence of the previous memory cell's state at $t-1$ for the new cell's state at t is.

$$\boldsymbol{f}_t = \sigma\left(\boldsymbol{W}_{xf}\boldsymbol{x}_t + \boldsymbol{W}_{hf}\boldsymbol{h}_{t-1} + \boldsymbol{b}_f\right) \tag{2.1}$$

   Note a weight matrix $\boldsymbol{W}_{xf}$ takes as input $\boldsymbol{x}$ and produces $\boldsymbol{f}$. This notation will be used for further matrices as well.

2. The **Input Gate** $\boldsymbol{i}_t$ determines how much the new input activation is being added to the new cell state $\boldsymbol{c}_t$. The input activation in the basic RNN was the hidden state $\boldsymbol{h}_t$ itself (see equation 1.2).

$$\boldsymbol{i}_t = \sigma\left(\boldsymbol{W}_{xi}\boldsymbol{x}_t + \boldsymbol{W}_{hi}\boldsymbol{h}_{t-1} + \boldsymbol{b}_i\right) \tag{2.2}$$

The input activation will be denoted as

$$\boldsymbol{g}_t = \tanh\left(\boldsymbol{W}_{xg}\boldsymbol{x}_t + \boldsymbol{W}_{hg}\boldsymbol{h}_{t-1} + \boldsymbol{b}_g\right) \tag{2.3}$$

3. The **Output Gate** $\boldsymbol{o}_t$ determines how much the new cell state influences the output of the memory cell.

$$\boldsymbol{o}_t = \sigma\left(\boldsymbol{W}_{xo}\boldsymbol{x}_t + \boldsymbol{W}_{ho}\boldsymbol{h}_{t-1} + \boldsymbol{b}_o\right) \tag{2.4}$$

Figure 2.2 illustrates the flow of information through such an RSTM cell. It highlights visually the role of these gates in such networks. The cell has three distinct inputs which are the previous cell's state $\boldsymbol{c}_{t-1}$, its output $\boldsymbol{h}_{t-1}$ and the data input $\boldsymbol{x}_t$. By definition 2.1 $\boldsymbol{f}_t$ preprocesses $\boldsymbol{c}_{t-1}$ and $\boldsymbol{h}_{t-1}$. Next $\boldsymbol{f}_t$ and $\boldsymbol{c}_{t-1}$ will be multiplied pointwise [4]. The flow of information of the *forget process* is highlighted in figure 2.3.

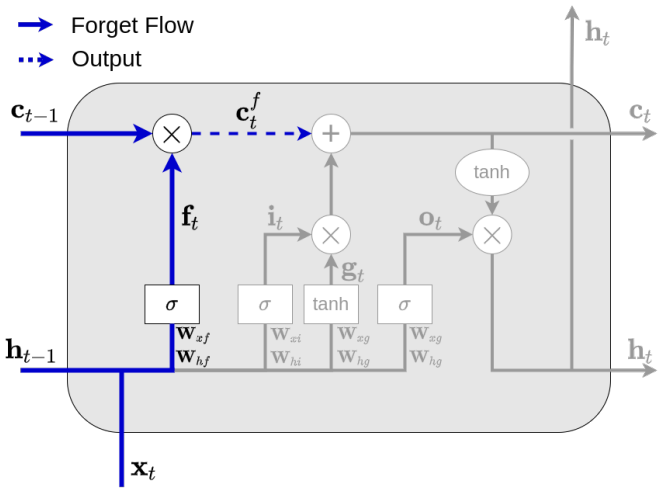

Figure 2.3: LSTM Cell with Forget Flow Highlighted

[4] We define the pointwise multiplication (Hadamard Product) of two vectors $\boldsymbol{a}, \boldsymbol{b} \in \mathbb{K}^n$ as

$$\boldsymbol{a} \odot \boldsymbol{b} = \begin{pmatrix} a_1 \cdot b_1 \\ a_2 \cdot b_2 \\ \dots \\ a_n \cdot b_n \end{pmatrix}$$

Note that if $\boldsymbol{f}$ is close or equal to $\mathbf{0}$ then the previous cell's state won't influence the succeeding calculations likewise if $\boldsymbol{f}_t$ is close or equal to $\mathbf{1}$ the previous cell's state will impact the succeeding calculations greatly. Further note that in the original work of Schmidhuber there was no forget gate $\boldsymbol{f}_t$ as mentioned before. To achieve the same behavior as in the original work we just set the forget gate to $\boldsymbol{f}_t = \mathbf{1}$ [10]. The processing at the multiplicative gate looks as follows

$$\boldsymbol{c}_t^f = \boldsymbol{c}_{t-1} \odot \boldsymbol{f}_t \tag{2.5}$$

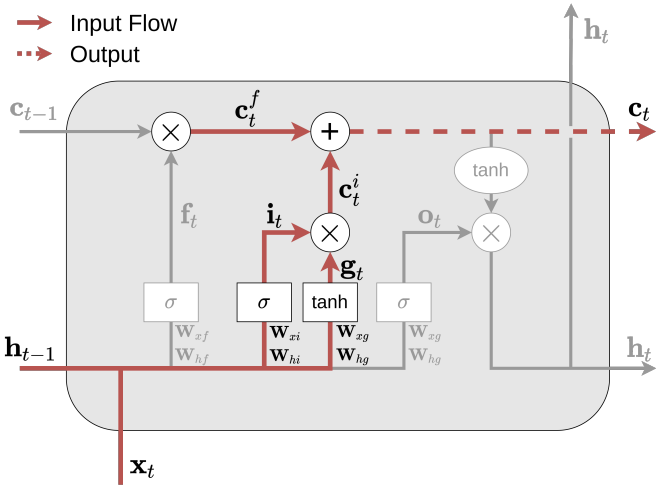

Figure 2.4: LSTM Cell with Input Flow Highlighted

Next we will discuss the input gate. Figure 2.4 highlights the flow of information regarding the input gate and its influence on the cell's state. As mentioned in definition 2.1 the input gate determines the impact of the recurrence $\boldsymbol{h}_{t-1}$ and $\boldsymbol{x}_t$ on the cell's state. Therefore we take the input gate $\boldsymbol{i}_t$ (equation 2.2) and input activation $\boldsymbol{g}_t$ and multiply them pointwise.

$$\boldsymbol{c}_t^i = \boldsymbol{i}_t \odot \boldsymbol{g}_t \tag{2.6}$$

The input gate's output $\boldsymbol{c}_t^i$ is a fitted version of the inputs activation regarding its importance with respect to its inputs $\boldsymbol{h}_{t-1}$ and $\boldsymbol{x}_t$.

Finally we can now compute the new cell's state $\boldsymbol{c}_t$. We now add what we determined to be important from the previous cell's state from the forget gate $\boldsymbol{c}_t^f$ and what we determined to be important from the recent input $\boldsymbol{c}_t^i$ such that we yield

$$\boldsymbol{c}_t = \boldsymbol{c}_t^f + \boldsymbol{c}_t^i \tag{2.7}$$

Lastly the output gate $\boldsymbol{o}_t$ does basically the same as the other gates did but for the output of the memory cell which is $\boldsymbol{h}_t$. Therefore the input $\boldsymbol{h}_{t-1}$ of each cell at time step

t is a gated output from the previous cell. The gated output determines how important the cell's state is for the output. If an LSTM cell's state is not relevant for the sequence in itself, the gate will filter the flow depending on the grade of importance **Hochreiter**. Figure 2.5 illustrates the information flow.

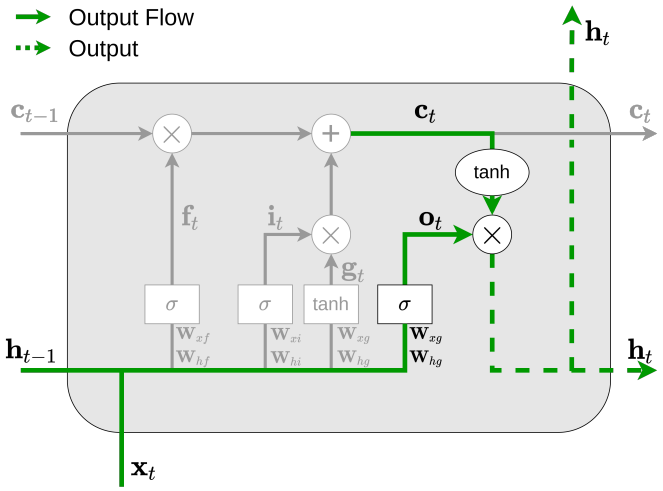

Figure 2.5: LSTM Cell with Output Flow Highlighted

Therefore we define the output of the LSTM cell at time step t to be

$$\boldsymbol{h}_t = \tanh(\boldsymbol{c}_t) \odot \boldsymbol{o}_t \tag{2.8}$$

2.2 Training LSTM RNNs

Next we try to understand what has changed such that an LSTM RNN improves on the issue of vanishing/exploding gradients. If we have a look again at the architecture of an LSTM cell (see figure 2.2) we see that after passing the gates we don't see any weights embedded in the illustration. These are omitted because the weights are being set to $\mathbf{1}$ to guarantee a *constant error flow* through the complete network [8]. This path of constant error flow through the network is denoted as the ***Constant Error Carousel* (CEC)** (see figure 2.6).

Definition 2.2 The **CEC** is a recurrent connection of a hidden state to itself with a linear unit[5], such that the error flow through the network is guaranteed to be constant.

[5] The linear unit is the addition of the gated state of the memory cell $\boldsymbol{c}_t^f$ and the gated state of the input $\boldsymbol{c}_t^i$. See equation 2.7 and figure 2.4.

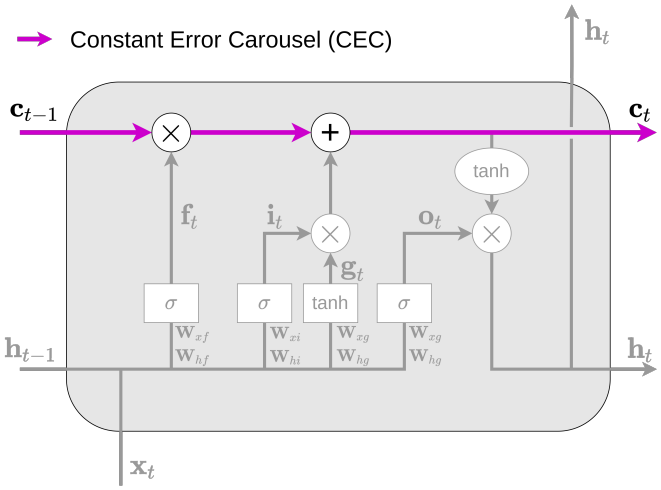

Figure 2.6: LSTM Cell with Constant Error Carousel Highlighted

Training an LSTM Network is in fact a complicated procedure. We will try to unravel the convoluted process of the optimization into a more easily comprehensible form. First we need to define the principle of *error truncation*, which is one of common solutions for reducing the effects of vanishing/exploding gradients and reducing computational complexity in regular RNNs [11].

Definition 2.3 Let $\tau > 0$ denote a constant of time steps. We call the BPTT algorithm error truncated (Truncated Backpropagation Through Time) by τ if we truncate, i.e. cut off, the backpropagation after τ time steps.

Remark 2.4 If we truncate the BPTT algorithm after τ time steps we don't calculate the furthest gradients such that vanishing/exploding gradient effect is limited by the choice of τ dependent of the weight matrices.

The CEC will be backpropagated using the usual backpropagation algorithm while the input feed of the former cell's output $\boldsymbol{h}$ is being truncated (see figure 2.7). Since we assume this to be the same simple network architecture as in chapter 1 with the hidden state becoming the LSTM cell. Though in forward propagation we have two outgoing signals, when reversing the signal flow for the backpropagation we have a very similar network architecture as in the simple RNN due to the truncation. Therefore we can use equation 1.11 and just change $\boldsymbol{h}$ to $\boldsymbol{c}$ such that

$$\frac{\partial L_{\mathcal{T}}}{\partial \boldsymbol{W}} = \frac{\partial L_{\mathcal{T}}}{\partial \boldsymbol{o}_{\mathcal{T}}} \frac{\partial \boldsymbol{o}_{\mathcal{T}}}{\partial \boldsymbol{c}_{\mathcal{T}}} \left(\prod_{k=t'+1}^{t} \frac{\partial \boldsymbol{c}_k}{\partial \boldsymbol{c}_{k-1}} \right) \frac{\partial \boldsymbol{c}_{t'}}{\partial \boldsymbol{W}_{t'}} \tag{2.9}$$

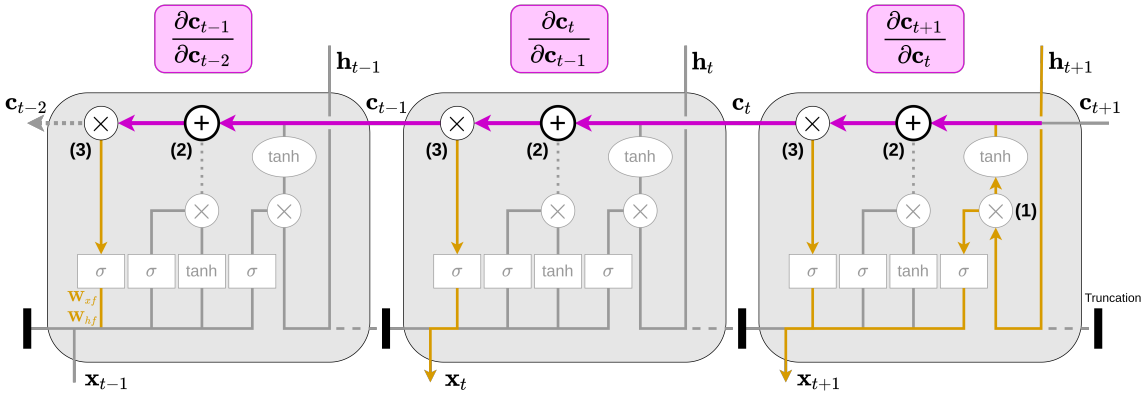

Figure 2.7: Backpropagation Through Time in an LSTM Network. The constant error flow through time is highlighted in pink, whereas the truncated backpropagation with $\tau = 1$ is highlighted in orange. The truncation is highlighted as thick blocking line at the bottom between two cells, canceling the $\boldsymbol{h}$ signal flow. This example highlights the backpropagation for $\frac{\partial \boldsymbol{L}_{t+1}}{\partial [\boldsymbol{W}_{xf}^{t-1}, \boldsymbol{W}_{hf}^{t-1}]}$.

First we will discuss the backpropagation for $\frac{\partial L_t}{\partial [\boldsymbol{W}_{xf}^{t'}, \boldsymbol{W}_{hf}^{t'}]}$. Note that we will have a look for both matrices $\boldsymbol{W}_{xf}$ and $\boldsymbol{W}_{hf}$ at the same time since they share the same signal path. Further note we keep the notation from section 1.2 of $t' < t$ such that we back-propagate over at least one but possibly multiple time steps. Figure 2.7 illustrates the backpropagation from $t+1$ to $t-1$.

Following from the right side of the illustration we see that first $\frac{\partial \boldsymbol{h}_{t+1}}{\partial \boldsymbol{o}_{t+1}}$ and $\frac{\partial \boldsymbol{h}_{t+1}}{\partial \tanh(\boldsymbol{c}_{t+1})}$ is being calculated at mark **(1)**. Recall that we have the Hadamard product at $\boldsymbol{o}$ (see equation 2.8). The derivative can be calculated using the product rule. For more insight on differentiating the Hadamard product see this article [12]. Note that after entering the output gate the backpropagation does not leave the memory cell following this path, except for the input which is $\boldsymbol{x}_{t+1}$ in our case. Following the path of $\tan(\boldsymbol{c}_{t+1})$ leads to the CEC.

Now considering the CEC, recall that the definition 2.3 required a linear activation. This linear activation at mark **(2)** prevents the backpropagation from entering preceding LSTM cell's input paths when propagating through the network. Only the forget gate is being considered due to the product rule of the Hadamard product at mark **(3)**. Mathematically this can be seen as follows [13].

$$\frac{\partial \boldsymbol{c}_{t+1}}{\partial \boldsymbol{c}_t} = \frac{\partial \boldsymbol{c}_t^f}{\partial \boldsymbol{c}_t} + \frac{\partial \boldsymbol{c}_t^i}{\partial \boldsymbol{c}_t} \qquad \text{(equation 2.7)}$$

$$= \frac{\partial \boldsymbol{c}_t \odot \boldsymbol{f}_{t+1}}{\partial \boldsymbol{c}_t} + \frac{\partial \boldsymbol{i}_{t+1} \odot \boldsymbol{g}_{t+1}}{\partial \boldsymbol{c}_t} \qquad \text{(equations 2.5 \& 2.6)}$$

$$= \frac{\partial \boldsymbol{c}_t \odot \boldsymbol{f}_{t+1}}{\partial \boldsymbol{c}_t} + \underbrace{\frac{\partial \boldsymbol{i}_{t+1} \odot \boldsymbol{g}_{t+1}}{\partial \boldsymbol{c}_t}}_{\text{input path}} \qquad \text{(differentiation sum rule)}$$

$$= \boldsymbol{c}_t \cdot \underbrace{\frac{\partial \boldsymbol{f}_{t+1}}{\partial \boldsymbol{c}_t}}_{=0} + \boldsymbol{f}_{t+1} \cdot \underbrace{\frac{\partial \boldsymbol{c}_t}{\partial \boldsymbol{c}_t}}_{=1} \qquad \text{(differentiation product rule)}$$

$$= \boldsymbol{f}_{t+1} \tag{2.10}$$

Substituting the found expression from equation 2.10 into the product $\prod_{k=t'+1}^{t} \frac{\partial \boldsymbol{c}_k}{\partial \boldsymbol{c}_{k-1}}$ gives us

$$\prod_{k=t'+1}^{t} \frac{\partial \boldsymbol{c}_k}{\partial \boldsymbol{c}_{k-1}} = \prod_{k=t'+1}^{t} \boldsymbol{f}_k \tag{2.11}$$

It can be observed that contrary to a regular RNN where we had the exponential growth on the weight matrix $\boldsymbol{W}$ (see equation 1.13) we here are completely independent of any weight matrix on the CEC signal path which is the illustration of the product in equation 2.11.

Note when learning the weights attached to the input gate, we apply the sum rule for derivatives but this time entering a different path at the point where we find the cell at the desired time step. This can be seen in figure 2.8. According to the same scheme the other weights can be learned.

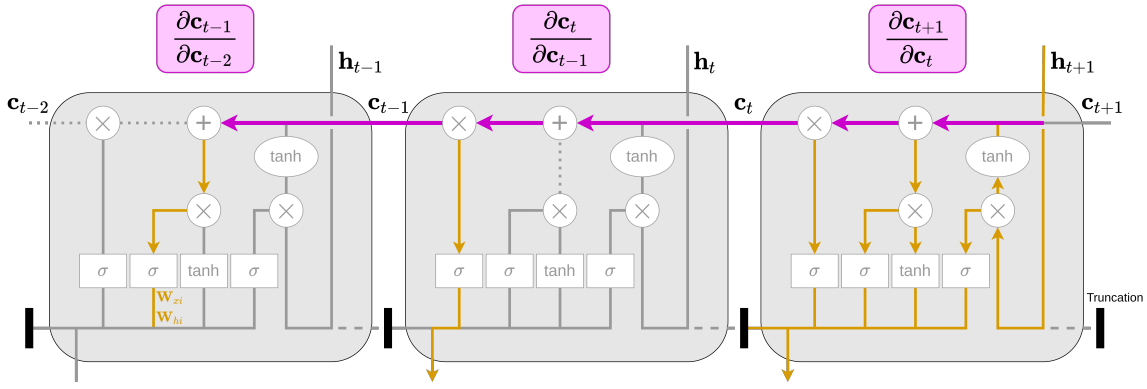

Figure 2.8: Backpropagation Highlighted for Learning Input Gate Weights

2.3 Conclusion

The LSTM architecture improves the behavior of the network over longer time spans. Noteworthy is that the gradients in LSTM networks can still vanish, but this happens at a much slower rate than in regular RNNs. Since the sequential data is still fed sequentially

into the network, we get the same computational complexity and memory consumption with respect to the length of the sequence [14]. Though there are deviations due to the network's architecture itself, the most impactful factor remains to be the length of the sequence for the computation.

3 Transformer Networks

Since the release of OpenAI's GPT-3 language model in 2021 the transformer has become very well known and showed very good results for natural language processing tasks [15]. Further the power of GPT-3 has also made mainstream headlines with the results of Dall-E in the natural language to artificial image generation [16]. Figure 3.1 shows examples what Dall-E can generate from a given input text.

(a) Input Text: *a shiba inu wearing a beret and black turtleneck*

(b) Input Text: *panda mad scientist mixing sparkling chemicals, artstation*

Figure 3.1: Dall-E Generated Images

The underlying success of these models is the transformer architecture. Following we will investigate briefly how the transformer architecture is setup.

3.1 Encoder Decoder Architecture

3.2 Attention Architecture

Bibliography

[1] S. S. Haykin, *Neural networks and learning machines*, Third. Upper Saddle River, NJ: Pearson Education, 2009.

[2] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*, ser. Adaptive Computation and Machine Learning series. MIT Press, 2016, ISBN: 9780262035613. [Online]. Available: https://books.google.de/books?id=Np9SDQAAQBAJ.

[3] N. Mohajerin and S. L. Waslander, "State initialization for recurrent neural network modeling of time-series data," in *2017 International Joint Conference on Neural Networks (IJCNN)*, 2017, pp. 2330–2337. DOI: 10.1109/IJCNN.2017.7966138.

[4] A. Kag and V. Saligrama, "Training recurrent neural networks via forward propagation through time," in *Proceedings of the 38th International Conference on Machine Learning*, M. Meila and T. Zhang, Eds., ser. Proceedings of Machine Learning Research, vol. 139, PMLR, 18–24 Jul 2021, pp. 5189–5200. [Online]. Available: https://proceedings.mlr.press/v139/kag21a.html.

[5] R. C. Staudemeyer and E. R. Morris, "Understanding LSTM - a tutorial into long short-term memory recurrent neural networks," *CoRR*, vol. abs/1909.09586, 2019. arXiv: 1909.09586. [Online]. Available: http://arxiv.org/abs/1909.09586.

[6] M. Henaff, A. Szlam, and Y. LeCun, "Orthogonal rnns and long-memory tasks," *CoRR*, vol. abs/1602.06662, 2016. arXiv: 1602.06662. [Online]. Available: http://arxiv.org/abs/1602.06662.

[7] A. Karpathy. "The unreasonable effectiveness of recurrent neural networks." (2015), [Online]. Available: http://karpathy.github.io/2015/05/21/rnn-effectiveness/ (visited on 08/09/2022).

[8] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural computation*, vol. 9, pp. 1735–80, Dec. 1997. DOI: 10.1162/neco.1997.9.8.1735.

[9] F. Gers, J. Schmidhuber, and F. Cummins, "Learning to forget: Continual prediction with lstm," vol. 12, Oct. 2000, pp. 2451–71. DOI: 10.1162/089976600300015015.

[10] Z. C. Lipton, "A critical review of recurrent neural networks for sequence learning," *CoRR*, vol. abs/1506.00019, 2015. arXiv: 1506.00019. [Online]. Available: http://arxiv.org/abs/1506.00019.

[11] H. Tang and J. R. Glass, "On training recurrent networks with truncated backpropagation through time in speech recognition," *CoRR*, vol. abs/1807.03396, 2018. arXiv: 1807.03396. [Online]. Available: http://arxiv.org/abs/1807.03396.

[12] T. Parr and J. Howard, "The matrix calculus you need for deep learning," *CoRR*, vol. abs/1802.01528, 2018. arXiv: 1802.01528. [Online]. Available: http://arxiv.org/abs/1802.01528.

[13] J. Bayer, P. van der Smagt, and J. Schmidhuber, *Learning Sequence Representations*. Universitätsbibliothek der TU München, 2015. [Online]. Available: https://books.google.de/books?id=GRHLzQEACAAJ.

[14] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin, "Attention is all you need," *CoRR*, vol. abs/1706.03762, 2017. arXiv: 1706.03762. [Online]. Available: http://arxiv.org/abs/1706.03762.

[15] R. Dale, "Gpt-3: What's it good for?" *Natural Language Engineering*, vol. 27, no. 1, pp. 113–118, 2021. DOI: 10.1017/S1351324920000601.

[16] A. Ramesh, P. Dhariwal, A. Nichol, C. Chu, and M. Chen, *Hierarchical text-conditional image generation with clip latents*, 2022. DOI: 10.48550/ARXIV.2204.06125. [Online]. Available: https://arxiv.org/abs/2204.06125.

Erklärung

I declare that I completed this paper independently and used only these materials that are listed. All materials used, from published as well as unpublished sources, whether directly quoted or paraphrased, are duly reported.

Furthermore I declare that this paper, or any abridgment of it, was not used for any other degree seeking purpose.

Mittweida, 27 July 2022