Recurrent Neural Networks (RNNs): A gentle Introduction and Overview

Robin M. Schmidt
Department of Computer Science
Eberhard-Karls-University Tübingen
Tübingen, Germany
rob.schmidt@student.uni-tuebingen.de

Abstract

State-of-the-art solutions in the areas of "Language Modelling & Generating Text", "Speech Recognition", "Generating Image Descriptions" or "Video Tagging" have been using Recurrent Neural Networks as the foundation for their approaches. Understanding the underlying concepts is therefore of tremendous importance if we want to keep up with recent or upcoming publications in those areas. In this work we give a short overview over some of the most important concepts in the realm of Recurrent Neural Networks which enables readers to easily understand the fundamentals such as but not limited to "Backpropagation through Time" or "Long Short-Term Memory Units" as well as some of the more recent advances like the "Attention Mechanism" or "Pointer Networks". We also give recommendations for further reading regarding more complex topics where it is necessary.

1 Introduction & Notation

Recurrent Neural Networks (RNNs) are a type of neural network architecture which is mainly used to detect patterns in a sequence of data. Such data can be handwriting, genomes, text or numerical time series which are often produced in industry settings (e.g. stock markets or sensors) [7, 12]. However, they are also applicable to images if these get respectively decomposed into a series of patches and treated as a sequence [12]. On a higher level, RNNs find applications in *Language Modelling & Generating Text*, *Speech Recognition*, *Generating Image Descriptions* or *Video Tagging*. What differentiates Recurrent Neural Networks from Feedforward Neural Networks also known as Multi-Layer Perceptrons (MLPs) is how information gets passed through the network. While Feedforward Networks pass information through the network without cycles, the RNN has cycles and transmits information back into itself. This enables them to extend the functionality of Feedforward Networks to also take into account previous inputs $\mathbf{X}_{0:t-1}$ and not only the current input $\mathbf{X}_t$. This difference is visualised on a high level in Figure 1. Note, that here the option of having multiple hidden layers is aggregated to one Hidden Layer block $\mathbf{H}$. This block can obviously be extended to multiple hidden layers.

We can describe this process of passing information from the previous iteration to the hidden layer with the mathematical notation proposed in [24]. For that, we denote the hidden state and the input at time step t respecively as $\mathbf{H}_t \in \mathbb{R}^{n \times h}$ and $\mathbf{X}_t \in \mathbb{R}^{n \times d}$ where n is number of samples, d is the number of inputs of each sample and h is the number of hidden units. Further, we use a weight matrix $\mathbf{W}_{xh} \in \mathbb{R}^{d \times h}$, hidden-state-to-hidden-state matrix $\mathbf{W}_{hh} \in \mathbb{R}^{h \times h}$ and a bias parameter $\mathbf{b}_h \in \mathbb{R}^{1 \times h}$. Lastly, all these informations get passed to a activation function ϕ which is usually a logistic sigmoid or tanh function to prepair the gradients for usage in backpropagation. Putting all these notations together yields Equation 1 as the hidden variable and Equation 2 as the output variable.

$$\mathbf{H}_t = \phi_h \left(\mathbf{X}_t \mathbf{W}_{xh} + \mathbf{H}_{t-1} \mathbf{W}_{hh} + \mathbf{b}_h \right) \tag{1}$$

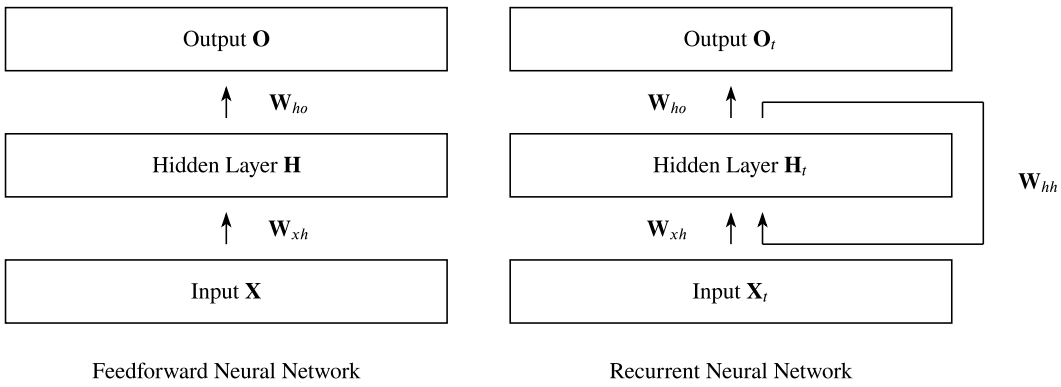

Figure 1: Visualisation of differences between Feedfoward NNs und Recurrent NNs

$$\mathbf{O}_t = \phi_o \left(\mathbf{H}_t \mathbf{W}_{ho} + \mathbf{b}_o \right) \tag{2}$$

Since $\mathbf{H}_t$ recursively includes $\mathbf{H}_{t-1}$ and this process occurs for every time step the RNN includes traces of all hidden states that preceded $\mathbf{H}_{t-1}$ as well as $\mathbf{H}_{t-1}$ itself.

If we compare that notation for RNNs with similar notation for Feedforward Neural Networks we can clearly see the difference we described earlier. In Equation 3 we can see the computation for the hidden variable while Equation 4 shows the output variable.

$$\mathbf{H} = \phi_h \left(\mathbf{X} \mathbf{W}_{xh} + \mathbf{b}_h \right) \tag{3}$$

$$\mathbf{O} = \phi_o \left(\mathbf{H} \mathbf{W}_{ho} + \mathbf{b}_o \right) \tag{4}$$

If you are familiar with training techniques for Feedforward Neural Networks such as backpropagation one question which might arise is how to properly backpropagate the error through a RNN. Here, a technique called Backpropagation Through Time (BPTT) is used which gets described in detail in the next section.

2 Backpropagation Through Time (BPTT) & Truncated BPTT

Backpropagation Through Time (BPTT) is the adaption of the backpropagation algorithm for RNNs [24]. In theory, this unfolds the RNN to construct a traditional Feedfoward Neural Network where we can apply backpropagation. For that, we use the same notations for the RNN as proposed before.

When we forward pass our input $\mathbf{X}_t$ through the network we compute the hidden state $\mathbf{H}_t$ and the output state $\mathbf{O}_t$ one step at a time. We can then define a loss function $\mathcal{L} \left(\mathbf{O}, \mathbf{Y} \right)$ to describe the difference between all outputs $\mathbf{O}_t$ and target values $\mathbf{Y}_t$ as shown in Equation 5. This basically sums up every loss term ℓ_t of each update step so far. This loss term ℓ_t can have different definitions based on the specific problem (e.g. Mean Squared Error, Hinge Loss, Cross Entropy Loss, etc.).

$$\mathcal{L} \left(\mathbf{O}, \mathbf{Y} \right) = \sum_{t=1}^{T} \ell_t \left(\mathbf{O}_t, \mathbf{Y}_t \right) \tag{5}$$

Since we have three weight matrices $\mathbf{W}_{xh}$, $\mathbf{W}_{hh}$ and $\mathbf{W}_{ho}$ we need to compute the partial derivative w.r.t. to each of these weight matrices. With the chain rule which is also used in normal backpropagation we get to the result for $\mathbf{W}_{ho}$ shown in Equation 6.

$$\frac{\partial \mathcal{L}}{\partial \mathbf{W}_{ho}} = \sum_{t=1}^{T} \frac{\partial \ell_t}{\partial \mathbf{O}_t} \cdot \frac{\partial \mathbf{O}_t}{\partial \phi_o} \cdot \frac{\partial \phi_o}{\mathbf{W}_{ho}} = \sum_{t=1}^{T} \frac{\partial \ell_t}{\partial \mathbf{O}_t} \cdot \frac{\partial \mathbf{O}_t}{\partial \phi_o} \cdot \mathbf{H_t} \tag{6}$$

For the partial derivative with respect to $\mathbf{W}_{hh}$ we get the result shown in Equation 7.

$$\frac{\partial \mathcal{L}}{\partial \mathbf{W}_{hh}} = \sum_{t=1}^{T} \frac{\partial \ell_t}{\partial \mathbf{O}_t} \cdot \frac{\partial \mathbf{O}_t}{\partial \phi_o} \cdot \frac{\partial \phi_o}{\partial \mathbf{H}_t} \cdot \frac{\partial \mathbf{H}_t}{\partial \phi_h} \cdot \frac{\partial \phi_h}{\partial \mathbf{W}_{hh}} = \sum_{t=1}^{T} \frac{\partial \ell_t}{\partial \mathbf{O}_t} \cdot \frac{\partial \mathbf{O}_t}{\partial \phi_o} \cdot \mathbf{W}_{ho} \cdot \frac{\partial \mathbf{H}_t}{\partial \phi_h} \cdot \frac{\partial \phi_h}{\partial \mathbf{W}_{hh}} \tag{7}$$

For the partial derivative with respect to $\mathbf{W}_{xh}$ we get the result shown in Equation 8.

$$\frac{\partial \mathcal{L}}{\partial \mathbf{W}_{xh}} = \sum_{t=1}^{T} \frac{\partial \ell_t}{\partial \mathbf{O}_t} \cdot \frac{\partial \mathbf{O}_t}{\partial \phi_o} \cdot \frac{\partial \phi_o}{\partial \mathbf{H}_t} \cdot \frac{\partial \mathbf{H}_t}{\partial \phi_h} \cdot \frac{\partial \phi_h}{\partial \mathbf{W}_{xh}} = \sum_{t=1}^{T} \frac{\partial \ell_t}{\partial \mathbf{O}_t} \cdot \frac{\partial \mathbf{O}_t}{\partial \phi_o} \cdot \mathbf{W}_{ho} \cdot \frac{\partial \mathbf{H}_t}{\partial \phi_h} \cdot \frac{\partial \phi_h}{\partial \mathbf{W}_{xh}} \tag{8}$$

Since each $\mathbf{H}_t$ depends on the previous time step we can substitute the last part from above equations to get Equation 9 and Equation 10.

$$\frac{\partial \mathcal{L}}{\partial \mathbf{W}_{hh}} = \sum_{t=1}^{T} \frac{\partial \ell_t}{\partial \mathbf{O}_t} \cdot \frac{\partial \mathbf{O}_t}{\partial \phi_o} \cdot \mathbf{W}_{ho} \sum_{k=1}^{t} \frac{\partial \mathbf{H}_t}{\partial \mathbf{H}_k} \cdot \frac{\partial \mathbf{H}_k}{\partial \mathbf{W}_{hh}} \tag{9}$$

$$\frac{\partial \mathcal{L}}{\partial \mathbf{W}_{xh}} = \sum_{t=1}^{T} \frac{\partial \ell_t}{\partial \mathbf{O}_t} \cdot \frac{\partial \mathbf{O}_t}{\partial \phi_o} \cdot \mathbf{W}_{ho} \sum_{k=1}^{t} \frac{\partial \mathbf{H}_t}{\partial \mathbf{H}_k} \cdot \frac{\partial \mathbf{H}_k}{\partial \mathbf{W}_{xh}} \tag{10}$$

The adapted part can then further be written as shown in Equation 11 and Equation 12.

$$\frac{\partial \mathcal{L}}{\partial \mathbf{W}_{hh}} = \sum_{t=1}^{T} \frac{\partial \ell_t}{\partial \mathbf{O}_t} \cdot \frac{\partial \mathbf{O}_t}{\partial \phi_o} \cdot \mathbf{W}_{ho} \sum_{k=1}^{t} \left(\mathbf{W}_{hh}^{\top}\right)^{t-k} \cdot \mathbf{H}_k \tag{11}$$

$$\frac{\partial \mathcal{L}}{\partial \mathbf{W}_{xh}} = \sum_{t=1}^{T} \frac{\partial \ell_t}{\partial \mathbf{O}_t} \cdot \frac{\partial \mathbf{O}_t}{\partial \phi_o} \cdot \mathbf{W}_{ho} \sum_{k=1}^{t} \left(\mathbf{W}_{hh}^{\top}\right)^{t-k} \cdot \mathbf{X}_k \tag{12}$$

From here, we can see that we need to store powers of $\mathbf{W}_{hh}^{k}$ as we proceed through each loss term ℓ_t of the overall loss function $\mathcal{L}$ which can become very large. For these large values this method becomes numerically unstable since eigenvalues smaller than 1 vanish and eigenvalues larger than 1 diverge [5]. One method of solving this problem is truncate the sum at a computationally convenient size [24]. When you do this, you're using Truncated BPTT [22]. This basically establishes an upper bound for the number of time steps the gradient can flow back to [15]. One can think of this upper bound as a moving window of past time steps which the RNN considers. Anything before the cut-off time step doesn't get taken into account. Since BPTT basically unfolds the RNN to create a new layer for each time step we can also think of this procedure as limiting the number of hidden layers.

3 Problems of RNNs: Vanishing & Exploding Gradients

As in most neural networks, vanishing or exploding gradients is a key problem of RNNs [12]. In Equation 9 and Equation 10 we can see $\frac{\partial \mathbf{H}_t}{\partial \mathbf{H}_k}$ which basically introduces matrix multiplication over the (potentially very long) sequence, if there are small values (< 1) in the matrix multiplication this causes the gradient to decrease with each layer (or time step) and finally vanish [6]. This basically stops the contribution of states that happened far earlier than the current time step towards the current time step [6]. Similarly, this can happen in the opposite direction if we have large values (> 1) during matrix multiplication causing an exploding gradient which in result values each weight too much and changes it heavily [6].

This problem motivated the introduction of the long short term memory units (LSTMs) to particularly handle the vanishing gradient problem. This approach was able to outperform traditional RNNs on a variety of tasks [6]. In the next section we want to go deeper on the proposed structure of LSTMs.

4 Long Short-Term Memory Units (LSTMs)

Long Short-Term Memory Units (LSTMs) [9] were designed to properly handle the vanishing gradient problem. Since they use a more constant error, they allow RNNs to learn over a lot more time steps (way over 1000) [12]. To achieve that, LSTMs store more information outside of the traditional neural network flow in structures called gated cells [6, 12]. To make things work in an LSTM we use an output gate $\mathbf{O}_t$ to read entries of the cell, an input gate $\mathbf{I}_t$ to read data into the cell and a forget

gate $\mathbf{F}_t$ to reset the content of the cell. The computations for these gates are shown in Equation 13, Equation 14 and Equation 15. For a more visual approach please see Figure 8 in Appendix A.

$$\mathbf{O}_t = \sigma\left(\mathbf{X}_t\mathbf{W}_{xo} + \mathbf{H}_{t-1}\mathbf{W}_{ho} + \mathbf{b}_o\right) \tag{13}$$

$$\mathbf{I}_t = \sigma\left(\mathbf{X}_t\mathbf{W}_{xi} + \mathbf{H}_{t-1}\mathbf{W}_{hi} + \mathbf{b}_i\right) \tag{14}$$

$$\mathbf{F}_t = \sigma\left(\mathbf{X}_t\mathbf{W}_{xf} + \mathbf{H}_{t-1}\mathbf{W}_{hf} + \mathbf{b}_f\right) \tag{15}$$

The shown equations use $\mathbf{W}_{xi}, \mathbf{W}_{xf}, \mathbf{W}_{xo} \in \mathbb{R}^{d\times h}$ and $\mathbf{W}_{hi}, \mathbf{W}_{hf}, \mathbf{W}_{ho} \in \mathbb{R}^{h\times h}$ as weight matrices while $\mathbf{b}_i, \mathbf{b}_f, \mathbf{b}_o \in \mathbb{R}^{1\times h}$ are their respective biases. Further, they use the sigmoid activation function σ to transform the output $\in (0, 1)$ which each results in a vector with entries $\in (0, 1)$.

Next, we need a candidate memory cell $\tilde{\mathbf{C}}_t \in \mathbb{R}^{n\times h}$ which has a similar computation as the previously mentioned gates but instead uses a tanh activation function to have an output $\in (-1, 1)$. Further, it again has its own weights $\mathbf{W}_{xc} \in \mathbb{R}^{d\times h}$, $\mathbf{W}_{hc} \in \mathbb{R}^{h\times h}$ and biases $\mathbf{b}_c \in \mathbb{R}^{1\times h}$. The respective computation is shown in Equation 16. See Figure 9 in Appendix A for a visualisation of this enhancement.

$$\tilde{\mathbf{C}}_t = \tanh\left(\mathbf{X}_t\mathbf{W}_{xc} + \mathbf{H}_{t-1}\mathbf{W}_{hc} + \mathbf{b}_c\right) \tag{16}$$

To plug some things together we introduce old memory content $\mathbf{C}_{t-1} \in \mathbb{R}^{n\times h}$ which together with the introduced gates controls how much of the old memory content we want to preserve to get to the new memory content $\mathbf{C}_t$. This is shown in Equation 17 where $\odot$ denotes element-wise multiplication. The structure so far can be seen in Figure 10 in Appendix A.

$$\mathbf{C}_t = \mathbf{F}_t \odot \mathbf{C}_{t-1} + \mathbf{I}_t \odot \tilde{\mathbf{C}}_t \tag{17}$$

The last step is to introduce the computation for the hidden states $\mathbf{H}_t \in \mathbb{R}^{n\times h}$ into the framework. This can be seen in Equation 18.

$$\mathbf{H}_t = \mathbf{O}_t \odot \tanh\left(\mathbf{C}_t\right) \tag{18}$$

With the tanh function we ensure that each element of $\mathbf{H}_t$ is $\in (-1, 1)$. The full LSTM framework can be seen in Figure 11 in Appendix A.

5 Deep Recurrent Neural Networks (DRNNs)

Deep Recurrent Neural Networks (DRNNs) are in theory a really easy concept. To construct a deep RNN with L hidden layers we simply stack ordinary RNNs of any type on top of each other. Each hidden state $\mathbf{H}_t^{(\ell)} \in \mathbb{R}^{n\times h}$ is passed to the next time step of the current layer $\mathbf{H}_{t+1}^{(\ell)}$ as well as the current time step of the next layer $\mathbf{H}_t^{(\ell+1)}$. For the first layer we compute the hidden state as proposed in the previous models shown shown in Equation 19 while for the subsequent layer we use Equation 20 where the hidden state from the previous layer is treated as input.

$$\mathbf{H}_t^{(1)} = \phi_1\left(\mathbf{X}_t, \mathbf{H}_{t-1}^{(1)}\right) \tag{19}$$

$$\mathbf{H}_t^{(\ell)} = \phi_\ell\left(\mathbf{H}_t^{(\ell-1)}, \mathbf{H}_{t-1}^{(\ell)}\right) \tag{20}$$

The output $\mathbf{O}_t \in \mathbb{R}^{n\times o}$ where o is the number of outputs is then computed as shown in Equation 21 where we only use the hidden state of layer L.

$$\mathbf{O}_t = \phi_o\left(\mathbf{H}_t^{(L)}\mathbf{W}_{ho} + \mathbf{b}_o\right) \tag{21}$$

6 Bidirectional Recurrent Neural Networks (BRNNs)

Lets take an example of language modeling for now. Based on our current models we are able to reliably predict the next sequence element (i.e. the next word) based on what we have seen so far. However, there scenarios where we might want to fill in a gap in a sentence and the part of the sentence after the gap conveys significant information. This information is necessary to take into account to perform well on this kind of task [24]. On a more generalised level we want to incorporate a look-ahead property for sequences.

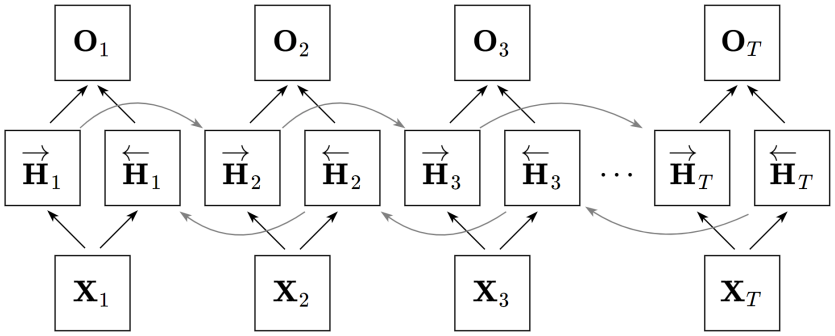

Figure 2: Architecture of a bidirectional recurrent neural network

To achieve this look-ahead property Bidirectional Recurrent Neural Networks (BRNNs) [14] got introduced which basically add another hidden layer which run the sequence backwards starting from the last element [24]. An architectural overview can is visualised in Figure 2. Here, we introduce a forward hidden state $\overrightarrow{\mathbf{H}}_t \in \mathbb{R}^{n \times h}$ and a backward hidden state $\overleftarrow{\mathbf{H}}_t \in \mathbb{R}^{n \times h}$. Their respective calculations are shown in Equation 22 and Equation 23.

$$\overrightarrow{\mathbf{H}}_t = \phi\left(\mathbf{X}_t \mathbf{W}_{xh}^{(f)} + \overrightarrow{\mathbf{H}}_{t-1} \mathbf{W}_{hh}^{(f)} + \mathbf{b}_h^{(f)}\right) \tag{22}$$

$$\overleftarrow{\mathbf{H}}_t = \phi\left(\mathbf{X}_t \mathbf{W}_{xh}^{(b)} + \overleftarrow{\mathbf{H}}_{t+1} \mathbf{W}_{hh}^{(b)} + \mathbf{b}_h^{(b)}\right) \tag{23}$$

For that, we have similar weight matrices as in definitions before but now they are seperated into two sets. One set of weight matrices is for the forward hidden states $\mathbf{W}_{xh}^{(f)} \in \mathbb{R}^{d \times h}$ and $\mathbf{W}_{hh}^{(f)} \in \mathbb{R}^{h \times h}$ while the other one is for the backward hidden states $\mathbf{W}_{xh}^{(b)} \in \mathbb{R}^{d \times h}$ and $\mathbf{W}_{hh}^{(b)} \in \mathbb{R}^{h \times h}$. They also have their respective biases $\mathbf{b}_h^{(f)} \in \mathbb{R}^{1 \times h}$ and $\mathbf{b}_h^{(b)} \in \mathbb{R}^{1 \times h}$. With that, we can compute the output $\mathbf{O}_t \in \mathbb{R}^{n \times o}$ with o being the number of outputs and $\frown$ denoting the concatenation of the two matrices on axis 0 (stacking them on top of each other).

$$\mathbf{O}_t = \phi\left(\left[\overrightarrow{\mathbf{H}}_t \frown \overleftarrow{\mathbf{H}}_t\right] \mathbf{W}_{ho} + \mathbf{b}_o\right) \tag{24}$$

Again, we have weight matrices $\mathbf{W}_{ho} \in \mathbb{R}^{2h \times o}$ and bias parameters $\mathbf{b}_o \in \mathbb{R}^{1 \times o}$. Keep in mind that the two directions can have different number of hidden units.

7 Encoder-Decoder Architecture & Sequence to Sequence (seq2seq)

The Encoder-Decoder architecture is a type of neural network architecture where the network is twofold. It consists of encoder network and a decoder network whose respective roles are to *encode* the input into a state and *decode* the state to an output. This state usually has shape of a vector or a tensor [24]. A visualisation of this structure is shown in Figure 3.

Based on this Encoder-Decoder architecture a model called Sequence to Sequence (seq2seq) [16] got proposed for generating a sequence output based on a sequence input. This model uses RNNs for the encoder as well as the decoder where the hidden state of the encoder gets passed to the hidden state of the decoder. Common applications of the model are Google Translate [16, 23], voice-enabled

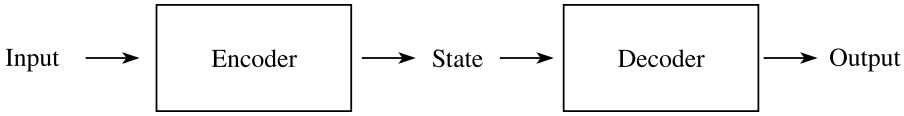

Figure 3: Encoder-Decoder Architecture Overview alternated from: [24]

devices [13] or labeling video data [18]. It mainly focuses on mapping a fixed length input sequence of size n to an fixed length output sequence of size m where $n \neq m$ can be true but isn't a necessity.

A de-rellod visualisation of the proposed architecture is shown in Figure 4. Here, we have a encoder which consists of a RNN accepting a single element of the sequence $\mathbf{X}_t$ where t is the order of the sequence element. These RNNs can be LSTMs or Gated Recurrent Units (GRUs) to further improve performance [16]. Further, the hidden states $\mathbf{H}_t$ are computed according to the definition of the hidden states in the used RNN type (e.g. LSTM or GRU). The Encoder Vector (context) is a representation of the last hidden state of the encoder network which aims to aggregate all information from all previous input elements. This functions as initial hidden state of the decoder network of the model and enables the decoder to make accurate predictions. The decoder network again is built of a RNN which predicts an output $\mathbf{Y}_t$ at a time step t. The produced output is again a sequence where each $\mathbf{Y}_t$ is a sequence element with order t. At each time step the RNN accepts a hidden state from the previous unit and itself produces an output as well as a new hidden state.

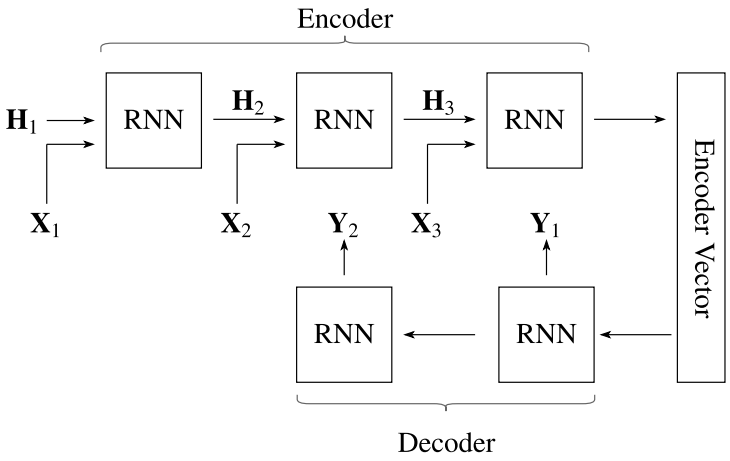

Figure 4: Visualisation of the Sequence to Sequence (seq2seq) Model

The Encoder Vector (context) was shown to be a bottleneck for these type of models since it needed to contain all all the necessary information of a source sentence in a fixed-length vector which was particularly problematic for long sequences. There have been approaches to solve this problem by introducing Attention in for example [4] or [10]. In the next section, we take a closer look at the proposed solutions.

8 Attention Mechanism & Transformer

The Attention Mechanism for RNNs is partly motivated by human visual focus and the peripheral perception [21]. It allows humans to focus on a certain region to achieve high resolution while adjacent objects are perceived with a rather low resolution. Based on these focus points and adjacent perception, we can make inference about what we expect to perceive when shifting our focus point. Similarly, we can transfer this method on our sequence of words where we are able to perform inference based on observed words. For example, if we perceive the word *eating* in the sequence "She is eating a green apple" we assume to observe a food object in the near future [21].

Generally, Attention takes two sentences and transforms them into a matrix where each sequence element (i.e. a word) corresponds to a row or column. Based on this matrix layout we can fill in the entries to identify relevant context or correlations between them. An example of this process can

be seen in Figure 5 where white denotes high correlation while black denotes low correlation. This method isn't limited to two sentences of a different languages as seen the example but can also be applied to the same sentence which is then called self-attention.

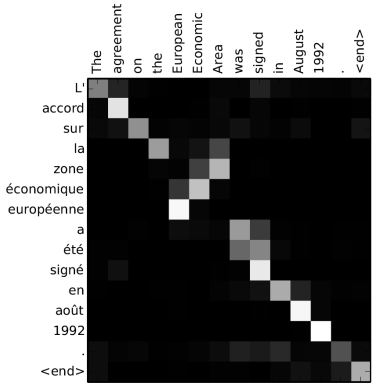

Figure 5: Example of an Alignment matrix of "L'accord sur la zone économique européen a été signé en août 1992" (French) and its English translation "The agreement on the European Economic Area was signed in August 1992": [4]

8.1 Definition

To help the seq2seq model to better deal with long sequences the attention mechanism got introduced. Instead of constructing the Encoder Vector out of the last hidden state of the encoder network, attention introduces shortcuts between context vector and the entire source input. A visualisation of this process can be seen in Figure 6. Here, we have source sequence $\mathbf{X}$ of length n and try to output a target sequence $\mathbf{Y}$ of size m. In that regard the formulation is rather similar to the one we described before in Section 7. We have an overall hidden state $\mathbf{H}_{t'}$ which is the concatenated version of the forward and backward pass as shown in Equation 25. Also, the hidden state of the decoder network is denoted as $\mathbf{S}_t$ while the encoder vector (context vector) is denoted as $\mathbf{C}_t$. Both of these are shown in Equation 26 and Equation 27 respectively.

$$\mathbf{H}_{t'} = \left[\overrightarrow{\mathbf{H}}_{t'} \overleftarrow{\mathbf{H}}_{t'}\right] \tag{25}$$

$$\mathbf{S}_t = \phi_d\left(\mathbf{S}_{t-1}, \mathbf{Y_{t-1}}, \mathbf{C_t}\right) \tag{26}$$

The context vector $\mathbf{C}_t$ is a sum of hidden states of the input sequence each weighted with an alignment score $\alpha_{t,t'}$ where $\sum_{t'=1}^{T} \alpha_{t,t'} = 1$. This is shown in Equation 27 as well as Equation 28.

$$\mathbf{C}_t = \sum_{t'=1}^{T} \alpha_{t,t'} \cdot \mathbf{H}_{t'} \tag{27}$$

$$\alpha_{t,t'} = \text{align}(\mathbf{Y}_t, \mathbf{X}_{t'}) = \frac{\exp\left(\text{score}(\mathbf{S}_{t-1}, \mathbf{H}_{t'})\right)}{\sum_{t'=1}^{T} \exp(\text{score}(\mathbf{S}_{t-1}, \mathbf{H}_{t'}))} \tag{28}$$

The alignment $\alpha_{t,t'}$ connects an alignment score for the input at position t' and the output at position t. This score is based on how well this pair matches [21]. The set of all alignment scores defines how much each source hidden state should be considered for each output [21]. Please see Appendix B for a more easy and visual explanation of the attention mechanism in the seq2seq model.

8.2 Different types of score functions

Generally, there are different implementations for this score function which have been used in various works. Table 1 gives an overview over their respective name, equation and the usage in publications. Here, we have two trainable weight matrices in the alignment model denoted as $\mathbf{v}_a$ and $\mathbf{W}_a$.

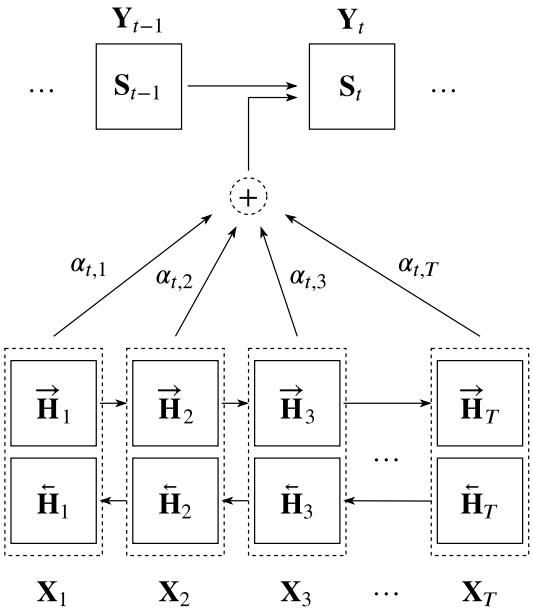

Figure 6: Encoder-Decoder architecture with additive attention mechanism alternated from: [4]

Name	**Equation for:** $\text{score}(\mathbf{S}_t, \mathbf{H}_{t'})$	**Used In**
Content-base	$\text{cosine}[\mathbf{S}_t, \mathbf{H}_{t'}]$	[8]
Additive	$\mathbf{v}_a^\top \tanh \mathbf{W}_a[\mathbf{S}_t; \mathbf{H}_{t'}]$	[3]
Location-Base	$\text{softmax}(\mathbf{W}_a \mathbf{S}_t)$	[11]
General	$\mathbf{S}_t^\top \mathbf{W}_a \mathbf{H}_{t'}$	[11]
Dot-Product	$\mathbf{S}_t^\top \mathbf{H}_{t'}$	[11]
Scaled Dot-Product	$\frac{\mathbf{S}_t^\top \mathbf{H}_{t'}}{\sqrt{n_{source}}}$	[17]

Table 1: Different score functions with their respective equations and usage alternated from: [21]

The Scaled-Dot-Product used in [17] scales the dot-product by the number of characters of the current word which is motivated by the problem that when the input is large, the softmax function may have an extremely small gradient which is a problem for efficient learning.

8.3 Transformer

By encorporating this Attention Mechanism the Transformer [17] got introduced which achieves parallelization by capturing recurrence sequence with attention but at the same time encoding each item's position in the sequence based on the encoder-decoder architecture [24]. In fact, for that it doesn't use any recurrent network units and entirely relies on the self-attention mechanism to improve performance. The encoding part of the architecture is made out of several encoders (e.g. six encoders in [17]) while the decoder part consists out of decoders with the same amount as the encoders. A general overview over the architecture is illustrated in Figure 7.

Here, each encoder component consists out of two sub-layers which are Self-Attention and a Feed Forward Neural Network. Similarly, those two sub-layers are found in each decoder component but with a Encoder-Decoder Attention sub-layer in between them which works similarly to the Attention used in the seq2seq model. The deployed Attention layers are not your ordinary attention layers but a method called Multi-Headed Attention which improves performance of the attention layer. This allows the model to jointly attend to information from different representation subspaces at different positions which in easier terms runs different chunks in parallel and concatenates the

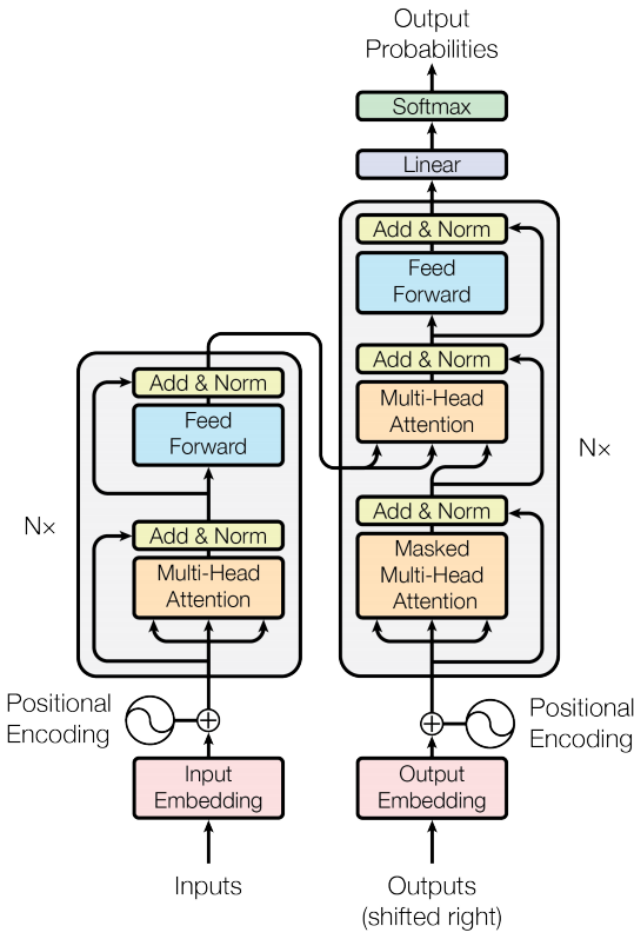

Figure 7: Model Architecture of the Transformer: [17]

results [17]. Unfortunately, explaining the design choices and mathematical formulations contained in multi-headed attention would be to much details at this point. Please refer to the original paper [17] for more information. The architecture shown in Figure 7 also deploys skip connections and layer normalisation for each sub-layer of the encoder as well as the decoder. One thing to note is that the input as well as the output get embedded and a positional encoding is applied which represents the proximity of sequence elements (see Appendix C).

The final linear and softmax layer turn the vector of floats which is the output of the decoder stack into a word. This is done by transforming the vector through the linear layer into a much larger vector called a logits vector [1]. This logits vector has the size of the learned vocabulary from the training dataset where each cell corresponds to the score of a unique word [1]. By applying a softmax function we turn those scores into probabilities which sum up to 1 and therefore we can choose the cell (i.e. the word) with the highest probability as output for this particular time step.

9 Pointer Networks (Ptr-Nets)

Pointer Networks (Ptr-Nets) [19] adapt the seq2seq model with attention to improve it by not fixing the discrete categories (i.e. elements) of the output dictionary *a priori*. Instead of yielding an output sequence generated from an input sequence, a pointer network creates a succession of pointers to the elements of the input series [25]. In [19] they show that by using Pointer Networks they can solve combinatorial optimization problems such as computing planar convex hulls, Delaunay triangulations and the symmetric planar Travelling Salesman Problem (TSP).

Generally, we apply additive attention (from Table 1) between states and then normalize it by applying the softmax function to model the output conditional probability as seen in Equation 29.

$$\mathbf{Y}_t = \text{softmax}\left(\text{score}\left(\mathbf{S}_t, \mathbf{H}_{t'}\right)\right) = \text{softmax}\left(\mathbf{v}_a^\top \tanh \mathbf{W}_a[\mathbf{S}_t; \mathbf{H}_{t'}]\right) \tag{29}$$

The attention mechanism is simplified, as Ptr-Net does not blend the encoder states into the output with attention weights. In this way, the output only responds to the positions but not the input content [21].

10 Conlusion & Outlook

In this work we gave an introduction into fundamentals for Recurrent Neural Networks (RNNs). This includes the general framework for RNNs, Backpropagation through time, problems of traditional RNNs, LSTMS, Deep and Bidirectional RNNs as well as more recent advances such as the Encoder-Decoder Architecture, seq2seq model, Attention, Transformer and Pointer Networks. Most topics are only covered conceptionally and don't go too deep into implementation specifications. To get a broader understanding of the covered topics, we recommend looking into some of the cited original papers. Additionally, most recent publications use some of the presented concepts so we recommend taking a look at such papers.

One recent publication which uses many of the presented concepts is "Grandmaster level in StarCraft II using multi-agent reinforcement learning" by Vinyals et al. [20]. Here, they present their approach to train agents to play the real-time strategy game Starcraft II with great success. If the presented concepts were a little too theoretical for you we recommend reading that paper to see LSTMs, the Transformer or Pointer Networks in a setting which can be deployed in a more practical environment.

References

[1] Jay Alammar. *The Illustrated Transformer*. 2018.

[2] Jay Alammar. *Visualizing A Neural Machine Translation Model (Mechanics of Seq2seq Models With Attention)*. 2018.

[3] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. *Neural Machine Translation by Jointly Learning to Align and Translate*. 2014.

[4] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. "Neural Machine Translation by Jointly Learning to Align and Translate". In: *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*. Ed. by Yoshua Bengio and Yann LeCun. 2015.

[5] Y. Bengio, Patrice Simard, and Paolo Frasconi. "Learning long-term dependencies with gradient descent is difficult". In: *IEEE transactions on neural networks / a publication of the IEEE Neural Networks Council* 5 (Feb. 1994), pp. 157–66.

[6] Gang Chen. *A Gentle Tutorial of Recurrent Neural Network with Error Backpropagation*. 2016.

[7] Junyoung Chung et al. "Gated Feedback Recurrent Neural Networks". In: *Proceedings of the 32Nd International Conference on International Conference on Machine Learning - Volume 37*. ICML'15. Lille, France: JMLR.org, 2015, pp. 2067–2075.

[8] Alex Graves, Greg Wayne, and Ivo Danihelka. *Neural Turing Machines*. 2014.

[9] Sepp Hochreiter and Jürgen Schmidhuber. "Long Short-term Memory". In: *Neural computation* 9 (Dec. 1997), pp. 1735–80.

[10] Minh-Thang Luong, Hieu Pham, and Christopher D. Manning. *Effective Approaches to Attention-based Neural Machine Translation*. 2015.

[11] Thang Luong, Hieu Pham, and Christopher D. Manning. "Effective Approaches to Attention-based Neural Machine Translation". In: *Proceedings of the 2015 Conference on Empirical Methods in Natural Language Processing*. Lisbon, Portugal: Association for Computational Linguistics, Sept. 2015, pp. 1412–1421.

[12] Chris Nicholson. *A Beginner's Guide to LSTMs and Recurrent Neural Networks*. `https://skymind.ai/wiki/lstm`. Accessed: 06 November 2019. 2019.

[13] Rohit Prabhavalkar et al. “A Comparison of Sequence-to-Sequence Models for Speech Recognition”. In: *INTERSPEECH*. 2017.

[14] Mike Schuster and Kuldip K. Paliwal. “Bidirectional recurrent neural networks”. In: *IEEE Trans. Signal Processing* 45 (1997), pp. 2673–2681.

[15] Ilya Sutskever. “Training Recurrent Neural Networks”. AAINS22066. PhD thesis. Toronto, Ont., Canada, 2013.

[16] Ilya Sutskever, Oriol Vinyals, and Quoc V Le. “Sequence to Sequence Learning with Neural Networks”. In: *Advances in Neural Information Processing Systems 27*. Ed. by Z. Ghahramani et al. Curran Associates, Inc., 2014, pp. 3104–3112.

[17] Ashish Vaswani et al. “Attention is All you Need”. In: *Advances in Neural Information Processing Systems 30*. Ed. by I. Guyon et al. Curran Associates, Inc., 2017, pp. 5998–6008.

[18] S. Venugopalan et al. “Sequence to Sequence – Video to Text”. In: *2015 IEEE International Conference on Computer Vision (ICCV)*. 2015, pp. 4534–4542.

[19] Oriol Vinyals, Meire Fortunato, and Navdeep Jaitly. “Pointer Networks”. In: *Advances in Neural Information Processing Systems 28*. Ed. by C. Cortes et al. Curran Associates, Inc., 2015, pp. 2692–2700.

[20] Oriol Vinyals et al. “Grandmaster level in StarCraft II using multi-agent reinforcement learning”. In: *Nature* 575.7782 (Oct. 2019), pp. 350–354.

[21] Lilian Weng. “Attention? Attention!” In: *lilianweng.github.io/lil-log* (2018). Accessed: 09 November 2019.

[22] R. J. Williams and J. Peng. “An Efficient Gradient-Based Algorithm for On-Line Training of Recurrent Network Trajectories”. In: *Neural Computation* 2.4 (1990), pp. 490–501.

[23] Yonghui Wu et al. *Google’s Neural Machine Translation System: Bridging the Gap between Human and Machine Translation*. 2016.

[24] Aston Zhang et al. *Dive into Deep Learning*. `http://www.d2l.ai`. 2019.

[25] Z. Zygmunt. *Introduction to pointer networks*. Accessed: 22 November 2019. 2017.

Appendices

A Visual Representation of LSTMs

In this section we consecutively construct the full architecture of Long Short-Term Memory Units (LSTMs) explained in Section 4. For a description what is changing between each step please read Section 4 or refer to the source of the illustrations [24].

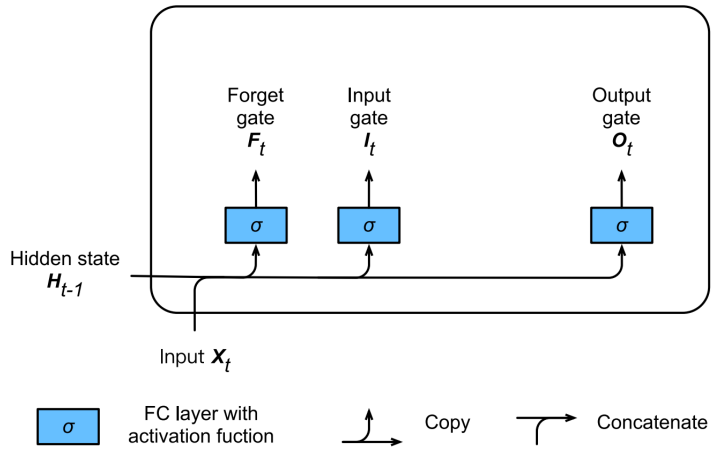

Figure 8: Calculation of input, forget, and output gates in an LSTM: [24]

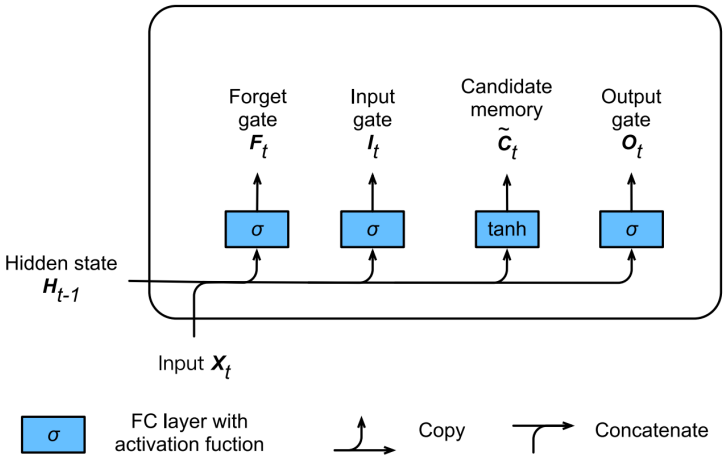

Figure 9: Computation of candidate memory cells in LSTM: [24]

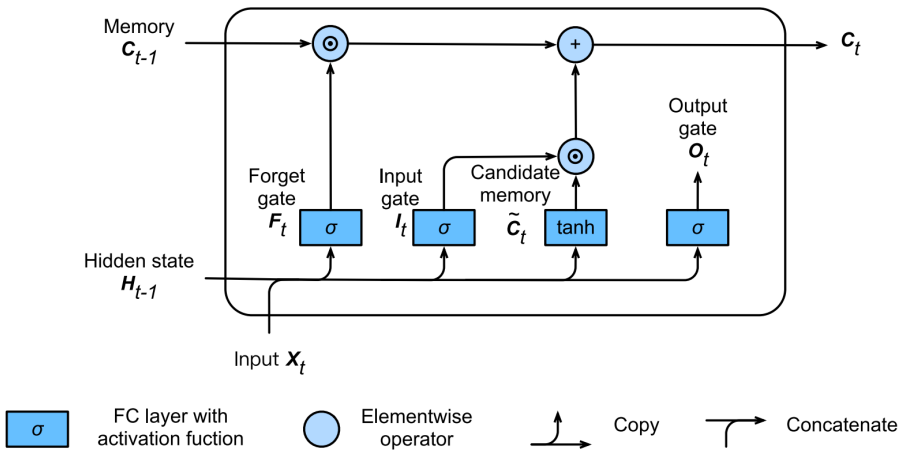

Figure 10: Computation of memory cells in an LSTM: [24]

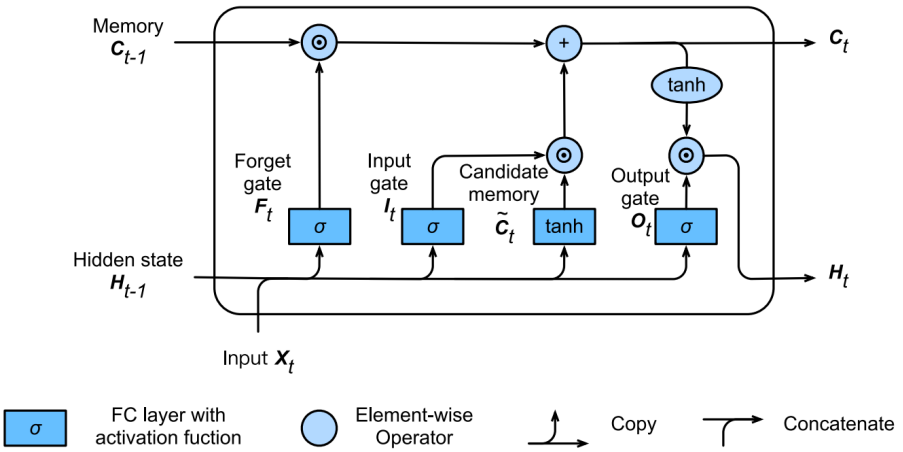

Figure 11: Computation of the hidden state in an LSTM: [24]

B Visual Representation of seq2seq with Attention

The seq2seq model with attention passes a lot more data from the encoder to the decoder than the regular seq2seq model. Instead of passing the last hidden state of the encoding stage, the encoder passes all the hidden states to the decoder. The first step of the decoder part in the seq2seq model with attention is illustrated in Figure 12 where we pass "*I am a student*" to the encoder and expect a translation to french producing "*je suis un étudiant*". Here, all the hidden states of the encoder $\mathbf{H}_1$, $\mathbf{H}_2$, $\mathbf{H}_3$ are passed to the attention decoder as well as the embedding from the $< End >$ token and an initial decoder hidden state $\mathbf{H}_{init}$.

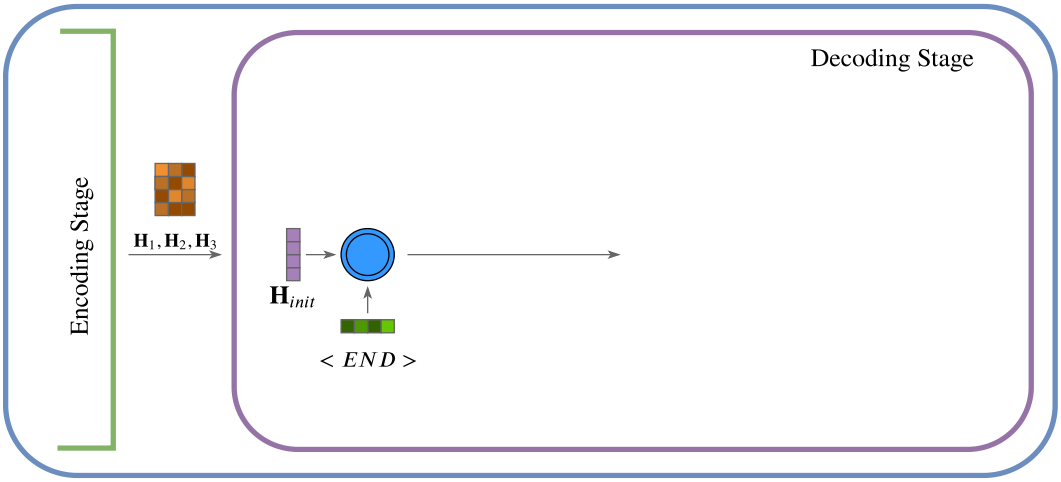

Figure 12: Seq2Seq Model with Attention Mechanism Step 1 alternated from: [2]

Next, we produce an output and a new hidden state vector $\mathbf{H}_4$. However, the output is discarded. This can be seen in Figure 13.

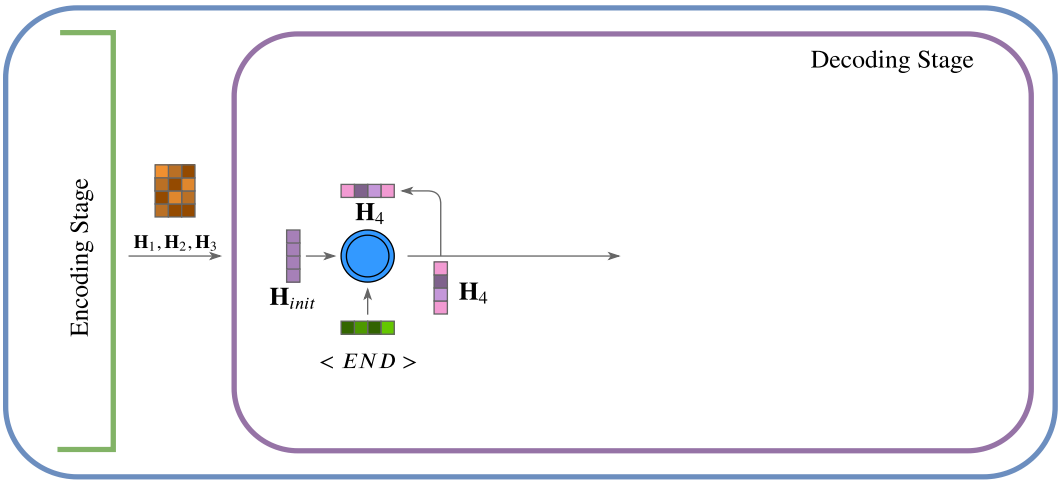

Figure 13: Seq2Seq Model with Attention Mechanism Step 2 alternated from: [2]

For the attention step we use this produced hidden state vector $\mathbf{H}_4$ and the hidden states from the encoder $\mathbf{H}_1$, $\mathbf{H}_2$, $\mathbf{H}_3$ to produce a context vector $\mathbf{C}_4$ (blue). This process can be seen in Figure 14. Each encoder hidden state is most associated with a certain word in the input sentence [2]. When we give these hidden states scores and apply a softmax to it we generate probability values. These probabilities are represented with by the three-element pink vector where light values stand for high probabilities while dark values denote low probabilities. Next, we apply each hidden state vector $\mathbf{H}_1$,

$\mathbf{H}_2$, $\mathbf{H}_3$ by its softmaxed score which increases hidden states with high scores, and decreases hidden states with low scores. This is visualised by graying out the hidden states $\mathbf{H}_2$ and $\mathbf{H}_3$ while keeping $\mathbf{H}_1$ in solid color.

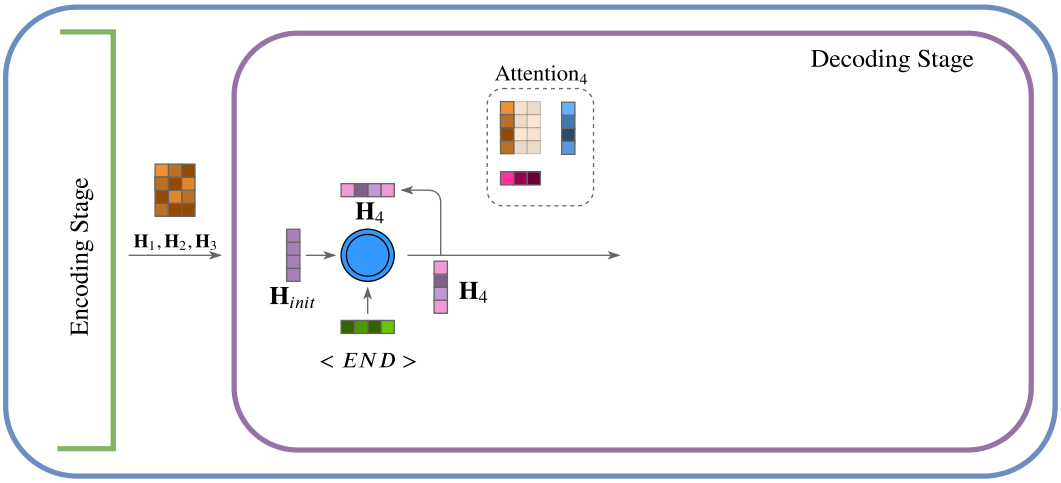

Figure 14: Seq2Seq Model with Attention Mechanism Step 4 alternated from: [2]

After that, we concatenate this produced context vector $\mathbf{C}_4$ with the produced hidden state $\mathbf{H}_4$. One can see this process in Figure 15. This process just stacks the two vectors on top of each other.

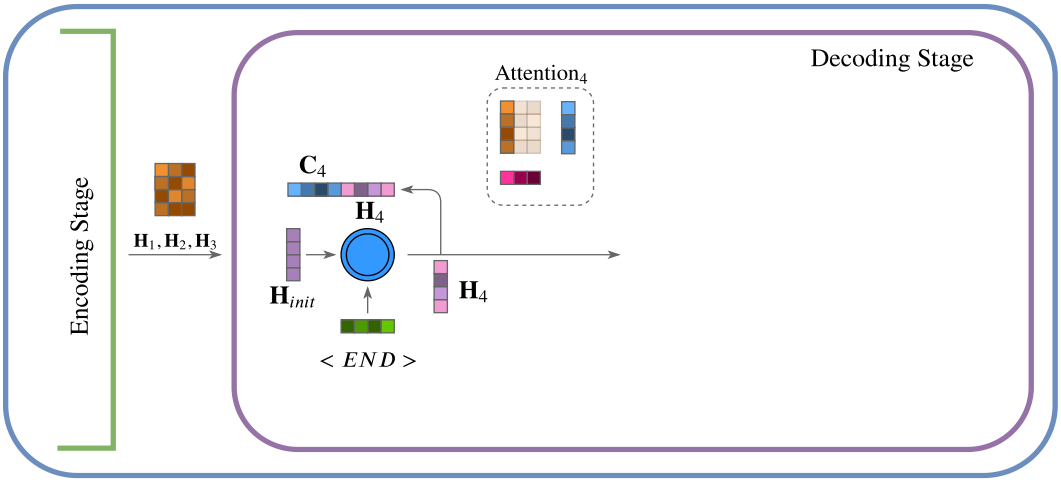

Figure 15: Seq2Seq Model with Attention Mechanism Step 5 alternated from: [2]

This concatenated version of hidden state $\mathbf{H}_4$ and context vector $\mathbf{C}_4$ is then passed into a jointly trained Feedforward Neural Network. This network is visualised by the red box with round edges in Figure 16. The output of this network then represents the output of the current time step t which in this case represents the word "*I*". This basically concludes all the steps needed at each iteration step.

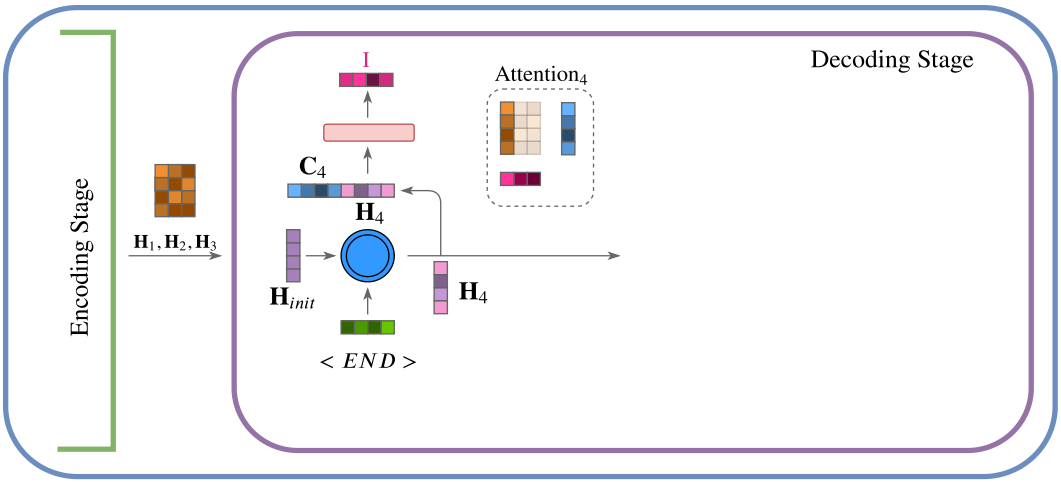

Figure 16: Seq2Seq Model with Attention Mechanism Step 6 alternated from: [2]

If we take a look at the next iteration step in Figure 17 we can see that the output from the previous hidden state $\mathbf{H}_4$ is passed instead of the $< END >$ token. All the other steps are equal from the previous iteration. However, we can see that the hidden state $\mathbf{H}_2$ has the best score during the attention stage. Again, this is represented by the lightest shade of pink in the score vector. By multiplying the scores with the hidden states we achieve two reduced hidden states $\mathbf{H}_1$ and $\mathbf{H}_3$ while keeping $\mathbf{H}_2$ as the most active hidden state. This results in the word "*am*" being produced as the output of the Feedforward Neural Network for this time step.

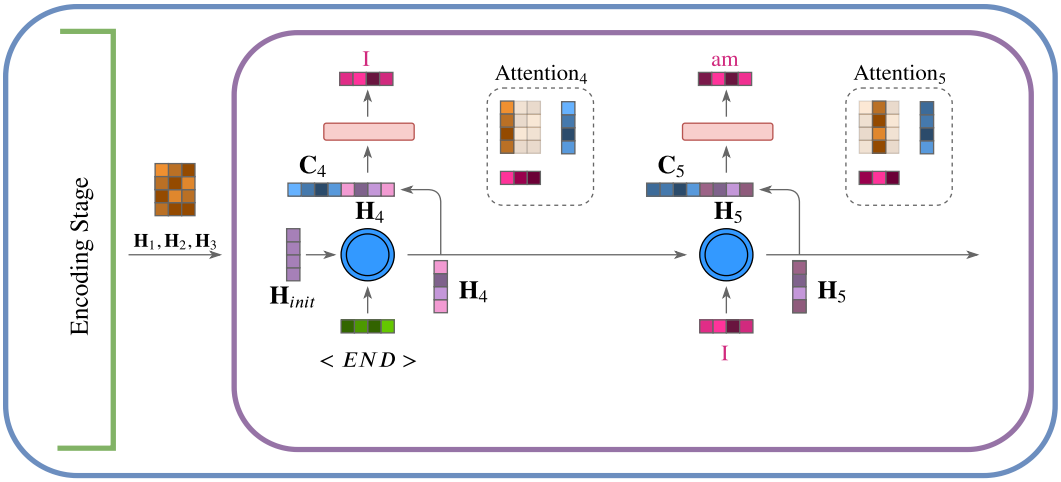

Figure 17: Seq2Seq Model with Attention Mechanism Step 7 alternated from: [2]

Obviously, there are still two more attention decoder time steps which are omitted here for illustration purposes. The functionality of each of those steps however would still be equivalent to the already seen time steps.

C Visual Representation of Positional Encodings used in the Transformer

On example of a positional encoding used inside the transformer is applying trigonometric functions as seen in Figure 18. Here, we have multiple trigonometric functions with different frequency. We also show the encoding for three words i.e. $\mathbf{X}_1, \mathbf{X}_2, \mathbf{X}_3$.

In principal the encoding for $\mathbf{X}_1$ is therefore high for the first curve (blue), mid for the second curve (red) and low for the last curve (green). Similarly, this applies for the other words as well. What

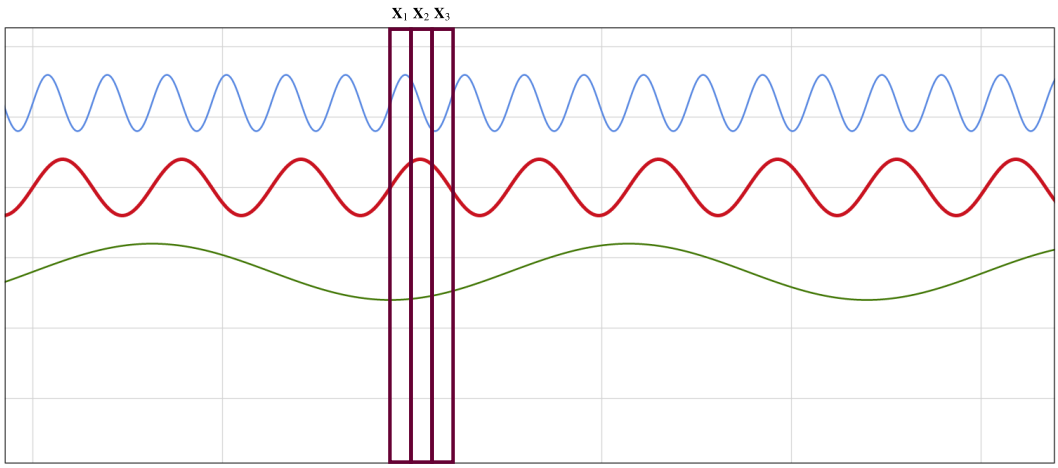

Figure 18: Positional Encoding Example based on trigonometric functions

we can see here is that close words have closer encodings while distant words have more different encodings. Generally, this is a method for binary encoding the position of a given sequence.

The choice of such a positional encoding algorithm definitely is not the main contribution of [17] but it is a relevant concept to at least understand in theory since this boosts performance.