Demystifying RNN with Mathematical and Graphical Insights with Application to Time Series Analysis and Natural Language Processing (NLP)

Salma Zafar, Amjad Ali*, Syeda Zahra Kazmi, Muhammad Zeeshan

Centre for Advanced Studies in Pure and Applied Mathematics (CASPAM)
Bahauddin Zakariya University (BZU), Multan, Pakistan.

*amjadali@bzu.edu.pk

Abstract

Recurrent Neural Networks (RNNs) are a type of neural network that maintains a hidden state, preserving information from previous inputs, which enables them to comprehend and generate sequences. RNNs excel in handling tasks involving sequential data over time, particularly in Natural Language Processing (NLP), including applications like voice recognition, music generation, and image captioning. Training RNNs with long sequences presents several challenges. To tackle these challenges, advanced techniques such as Long Short-Term Memory (LSTM), Gated Recurrent Units (GRU), and Bidirectional RNN (BRNNs) are available in the literature. This article comprehensively explains the fundamental processes of RNNs, including the variants LSTM, GRU, and BRNN, with mathematical and graphical insights. The process is explained using detailed mathematical expressions and algorithmic constructs. The article includes a hands-on worked-out example demonstrating the word prediction. Additionally, it includes an application involving sentiment analysis and compares the performance of simple RNNs, LSTM, GRU, and BRNNs using Transfer Learning. The article also includes an example of time series analysis problem.

Keywords: Deep Neural Networks, Artificial Neural Networks, Recurrent Neural Networks, RNNs, Cyclic Connections Networks, Sequence Models, Backpropagation Through Time (BPTT), Gradient Descent, Long Short-Term Memory (LSTM), Gated Recurrent Unit (GRU), Bidirectional RNN (BRNN), Natural Language Processing (NLP), Time Series Analysis

ACM Classification Codes (ccs98): **I.2.6**; **I.5.1**; **K.3.2**. MSC Codes (2020): **68T07**

1. Introduction

Deep learning an advanced subset of machine learning harnesses Artificial Neural Networks as its foundation for data-driven knowledge acquisition. Neural Networks draw inspiration from the intricate architecture of the human brain and these networks comprise interconnected layers of nodes each performing elementary mathematical operations. Their interconnection enables the acquisition of intricate data patterns. Notably, three primary forms of deep learning networks exist Artificial Neural Networks (ANNs) Convolutional Neural Networks (CNNs) and Recurrent Neural Networks (RNNs). Among these ANNs offer versatility and can adeptly address various problem domains including classification, regression and clustering. For image-centric tasks CNNs excel. They employ convolutional filters to extract salient features from images empowering them to excel in computer vision applications such as image classification object detection and image segmentation. In contrast, RNNs are tailor-made for sequential data. Equipped with feedback loops they handle long-term dependencies within data sequences finding applications in NLP tasks like machine translation, text generation and speech recognition.

As Kutyniok (2022) highlighted, the domain of Artificial Intelligence mathematics provides a foundational framework and discusses significant discoveries while shedding light on ongoing challenges. Traditional NLP methods such that N-grams have limitations particularly in capturing distant word relationships and handling variable-length sequences. These methods treat individual words discretely unable to grasp dynamic contextual nuances. RNNs on the other hand present a compelling remedy. RNNs excel in language understanding in NLP because they can adjust their inner workings based on what they observe and what they've seen before. They're also strong at processing long-distance connections between words and sentences, which helps with understanding the meaning of words. They find wide-ranging applications in everyday life constituting a cornerstone in deep learning (Bird *et al*., 2009). While ANNs and CNNs have made significant strides in handling multi-dimensional data with independent features certain data types involve continuous connections between attributes such as continuous Time Series, Language/Text data and biological data. To address these dependencies the Elman network a specific category of RNNs developed by Jeff Elman in 1990 was introduced. Based on Vanilla RNN, this architecture employs a straightforward recurrent structure where the current network input incorporates information from the preceding time step. RNNs characterized by their short-term memory can retain prior input states to generate subsequent outputs in a sequence. They maintain state variables that capture diverse sequential data patterns with training facilitated by optimization algorithms such as Gradient Descent (GD), Adam, Momentum GD and Adaptive learning rate methods (Marhon *et al*., 2013).

The primary distinction between ANNs, CNNs and RNNs is their structural design. ANNs adopt a feedforward architecture characterized by unidirectional data flow from the input to the output layer. In contrast, CNNs employ a convolutional architecture where data undergoes processing via convolutional filters. RNNs feature a recurrent architecture facilitating data feedback into the network at each time step. Furthermore, ANNs, CNNs and RNNs exhibit varied data processing capabilities. ANNs demonstrate versatility and can handle diverse data types although they are commonly applied to tabular data. RNNs offer a robust solution to various challenges encompassing machine translation, text generation, speech recognition, music composition, video captioning, question answering and sentiment analysis. Their intrinsic strength lies in their aptitude for processing sequential data as they maintain a memory of prior inputs enabling the capture of temporal dependencies in the data. Regarding network architecture ANNs feature connections between neurons in adjacent layers both forward and backwards. Conversely, RNNs incorporate recurrent connections allowing information to traverse from one step in the sequence to the next a crucial differentiator between the two.

Traditional feedforward neural networks can model sequential data but only when the data is represented using features that effectively capture the essential patterns within the sequence as observed by (Salehinejad *et al*., 2018). However, deriving such a functional representation can be a labor-intensive process. Another approach to modelling sequential data with a feedforward architecture involves creating a distinct set of parameters for each time or sequence position. Each set of parameters is responsible for learning patterns at a specific position substantially increasing the model's memory requirements. In contrast, RNNs adopt a more efficient strategy by sharing parameters across time steps facilitating learning and generalization from sequences of varying lengths. Specifically, the same set of parameters is applied at each time step to compute the output and adapt the network's hidden state. This parameter-sharing characteristic in RNNs enables the network to acquire a concise representation of the input sequence that captures its temporal dependencies. These networks are occasionally referred to as Spatio-Temporal Networks as highlighted by (Marhon *et al*., 2013). This sharing of weights and biases across time steps results in each time step serving as a condensed summary of the preceding inputs collectively constituting the sequence's information.

Furthermore, employing the same set of parameters at each time step offers two significant advantages. Firstly, it reduces the diversity of parameters that must be explicitly learned expediting the model training process. Secondly, instead of memorizing individual sequences this approach enhances the model's ability to generalize to longer sequences efficiently. Rather than committing specific sequences to memory the model learns to identify recurring patterns at various points within the sequence. RNNs can handle data of varying lengths without requiring padding or uniform sizing of input sequences. Traditionally, training RNNs posed challenges due to their substantial parameter count, often reaching millions. Nevertheless, recent advancements in network architectures, optimization techniques, and parallel computing have enabled successful large-scale training with RNNs. Architectures like LSTM and Bidirectional RNNs have exhibited groundbreaking performance across a spectrum of tasks, including handwriting recognition, language translation and image captioning as demonstrated by (Lipton *et al*., 2015).

Numerous resources, including books, articles, research papers, and other material, delve into RNNs. These encompass works by Werbos (1990), Schuster & Paliwal (1997), Hochreiter & Schmidhuber (1997), D'informatique *et al*. (2001), Bishop (2006), Graves (2012), Sutskever (2013), Sutskever *et al*. (2014), Hagan *et al*. (2014), LeCun *et al*. (2015), Goodfellow *et al*. (2016), Chen (2016), Hastie *et al*. (2016), Prabhavalkar *et al*. (2017), Aggarwal (2018), Ng (2018), Higham (2019), Strang (2019), Dawani (2020), Sarker (2021), Kutyniok (2022) and Alexender (2023). However, despite this wealth of resources there remains a need for a comprehensive mathematical exposition of backpropagation and improved visual representations.

RNNs belong to the family of deep-learning networks sharing this characteristic with ANNs and CNNs. The connection between RNNs, ANNs and CNNs opens the door to solving even more intricate problems. RNNs have gained popularity due to the evolution of backpropagation techniques, notably the gradient descent algorithm which has simplified their training process. To effectively model long-term dependencies RNNs must surmount challenges like the vanishing and exploding gradient problems as explored within this study. This paper is structured into distinct sections each addressing specific aspects of RNNs. Section 2 delves into the fundamental structure of RNNs while Section 3 expounds on mathematical working of RNNs. Section 4 delves into the strategies employed for training RNNs including Backpropagation through time. Section 5 gives overview of some obstacles in training of RNNs. Section 6 includes a hands-on exercise for guessing the following word using RNNs from a mathematical standpoint. Section 7 focuses into the advanced variation of RNN known as Long Short-Term Memory (LSTM), which is highly successful in dealing with long-term dependencies. Section 8 examines into the operation of GRU and BRNNs offering insight on their operating methods. Section 10 switches to the use of NLP techniques in predicting emojis providing insights into this exciting area of research and its practical ramifications.

2. Basic Structure of RNN

The fundamental components of an RNN are an input layer an, output layer and numerous hidden layers (sometimes referred to as recurrent layers). The sequence of inputs is received by the input layer which is processed by recurrent layers using feedback loops before producing the desired result at the output layer. There are three states of RNN:

- Input state: The input state captures all external/Internal information entering the system. The input layer comprises a set of cells/nodes/time steps each representing the input sequence's specific characteristic or feature/word/letter.
- Output state: The output state of the RNN is the final state of the network after processing a sequence of inputs with single/multiple nodes depending on the nature of the problem. For binary classification tasks the output layer will consist of a single output node that produces a 0 or 1 output. The output layer may consist of multiple nodes/time steps corresponding to a different possible output for multi-variable classification or time series forecasting.
- Recurrent state: The recurrent/hidden/memory state is the internal state of a Recurrent Neural Network (RNN) that stores information from previous inputs. This state allows the RNN to capture the temporal relationships in the input sequence and use them to output a meaningful prediction.

A schematic structure of unidirectional RNN is shown in Fig. 2.1.

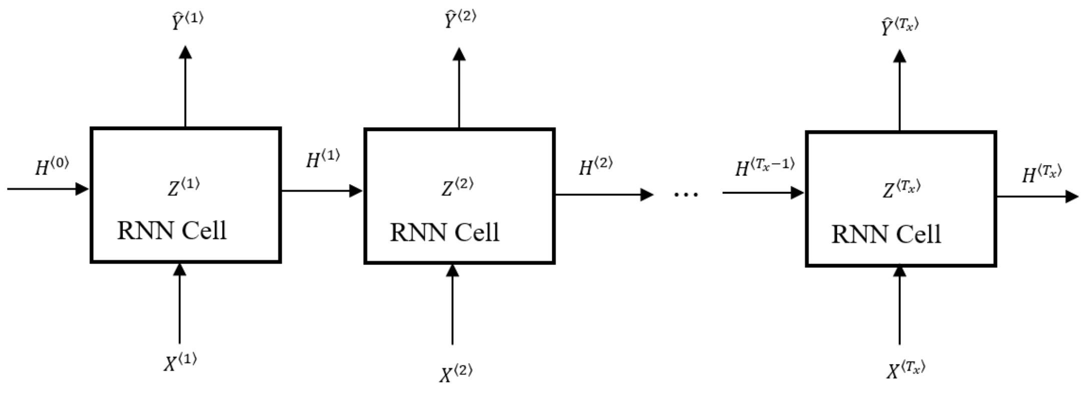
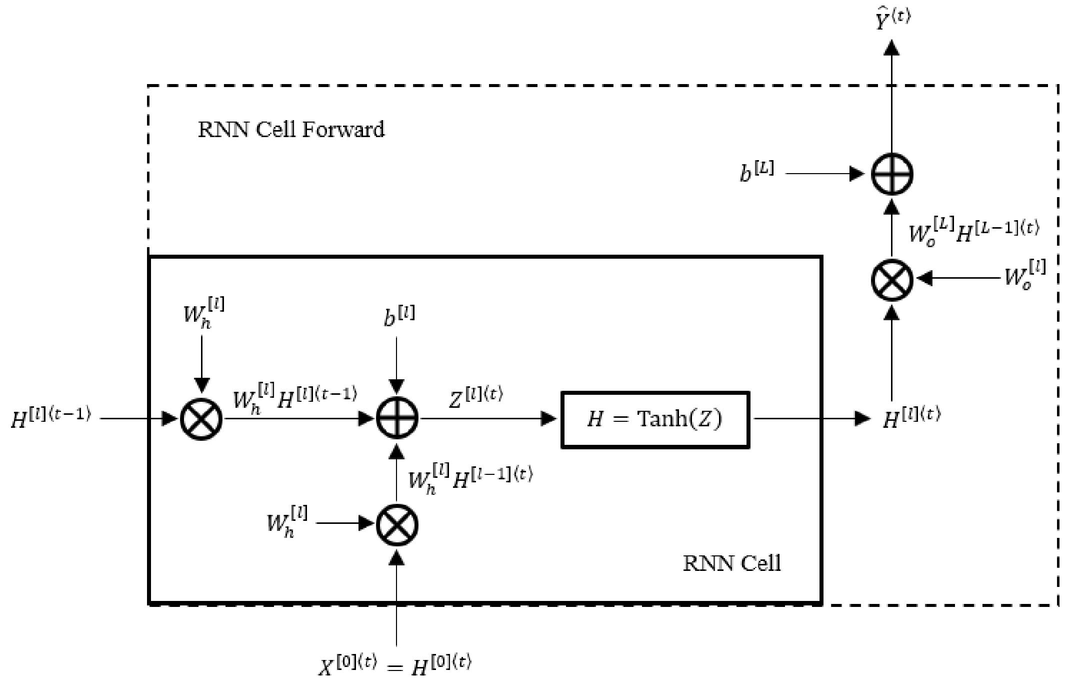

$\hat{Y}^{\langle 1 \rangle}$
$\hat{Y}^{\langle 2 \rangle}$
$\hat{Y}^{\langle T_x \rangle}$
$H^{\langle 0 \rangle}$
$Z^{\langle 1 \rangle}$
RNN Cell
$H^{\langle 1 \rangle}$
$Z^{\langle 2 \rangle}$
RNN Cell
$H^{\langle 2 \rangle}$
...
$H^{\langle T_x - 1 \rangle}$
$Z^{\langle T_x \rangle}$
RNN Cell
$H^{\langle T_x \rangle}$
$X^{\langle 1 \rangle}$
$X^{\langle 2 \rangle}$
$X^{\langle T_x \rangle}$

Fig. 2.1: Basic schematic structure for a unidirectional RNN

The same neural network can create RNNs at the word and character levels. The collection of basic symbols used to define the sequence is the only distinction between these two. While introducing the notations stick to the word-level RNN in this document (Aggarwal, 2018).

2.1. RNN Cells

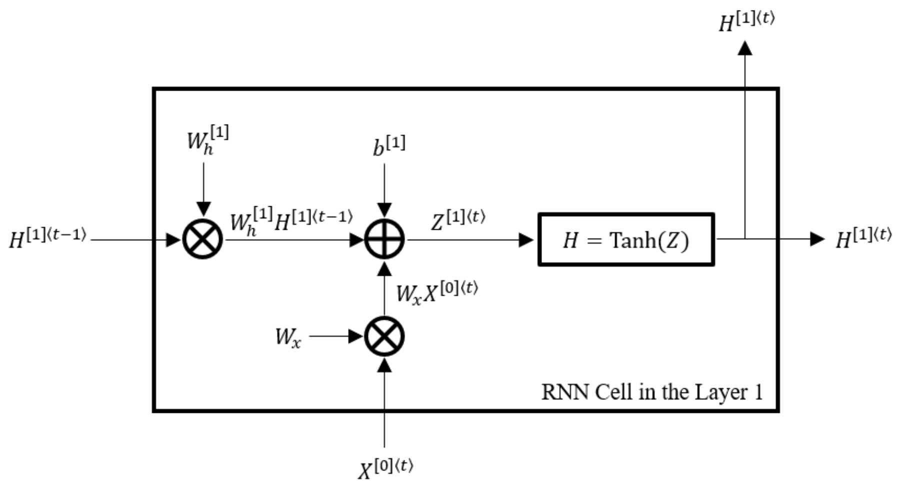
RNN replicate a single cell at a time. The schematic structure of a single time step of an RNN cell is shown in Fig. 2.2.



$\hat{Y}^{\langle t \rangle}$
RNN Cell Forward
$b^{[L]}$
$W_o^{[L]}H^{[L-1]\langle t \rangle}$
$W_o^{[l]}$
$W_h^{[l]}$
$b^{[l]}$
$H^{[l]\langle t-1 \rangle}$
$W_h^{[l]}H^{[l]\langle t-1 \rangle}$
$Z^{[l]\langle t \rangle}$
$H = \text{Tanh}(Z)$
$H^{[l]\langle t \rangle}$
$W_h^{[l]}H^{[l-1]\langle t \rangle}$
$W_h^{[l]}$
RNN Cell
$X^{[0]\langle t \rangle} = H^{[0]\langle t \rangle}$

Fig. 2.2: RNN cell for the forward pass

In Fig. 2.2, W_x, W_z and W_o represent the weight parameters shared at different time steps. $Z^{\langle t \rangle}$ is the intermediate state at time step t is the linear combination of weights, input and hidden states. $H^{\langle t \rangle}$ is the hidden state at time step t which lies in the range of tanh function. b_h and b_o are the recurrent and output state bias vectors respectively. $\hat{Y}^{\langle t \rangle}$ is the probabilistic output of the Softmax activation function. Schematic structures of the basic types of RNN cells for different layers are shown in Figs. 2.3-2.5.



$H^{[1]\langle t \rangle}$
$W_h^{[1]}$
$b^{[1]}$
$H^{[1]\langle t-1 \rangle}$
$W_h^{[1]}H^{[1]\langle t-1 \rangle}$
$Z^{[1]\langle t \rangle}$
$H = \text{Tanh}(Z)$
$H^{[1]\langle t \rangle}$
$W_x X^{[0]\langle t \rangle}$
W_x
RNN Cell in the Layer 1
$X^{[0]\langle t \rangle}$

Fig. 2.3: Basic RNN cell in the layer $l = 1$

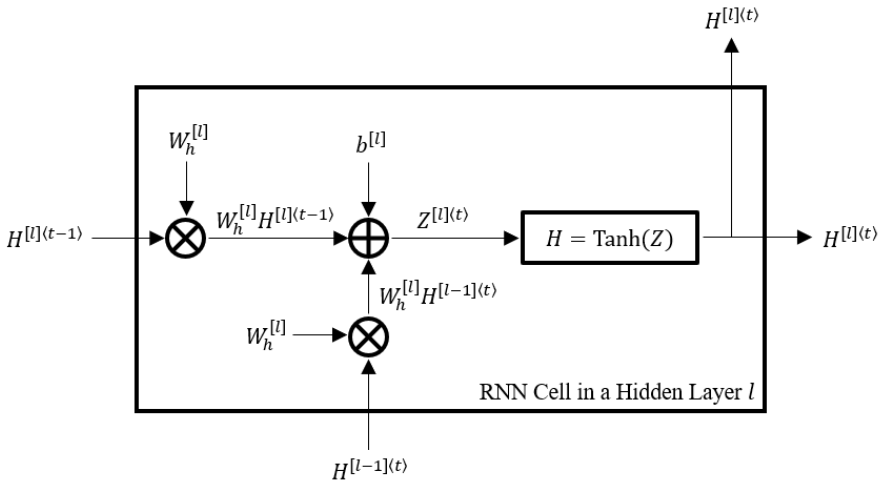

$H^{[l]\langle t\rangle}$
$W_h^{[l]}$
$b^{[l]}$
$H^{[l]\langle t-1\rangle}$
$W_h^{[l]}H^{[l]\langle t-1\rangle}$
$Z^{[l]\langle t\rangle}$
$H = \text{Tanh}(Z)$
$H^{[l]\langle t\rangle}$
$W_h^{[l]}H^{[l-1]\langle t\rangle}$
$W_h^{[l]}$
RNN Cell in a Hidden Layer l
$H^{[l-1]\langle t\rangle}$

Fig. 2.4: Basic RNN Cell in layer l, for any $l = 2,3,\cdots,L-1$

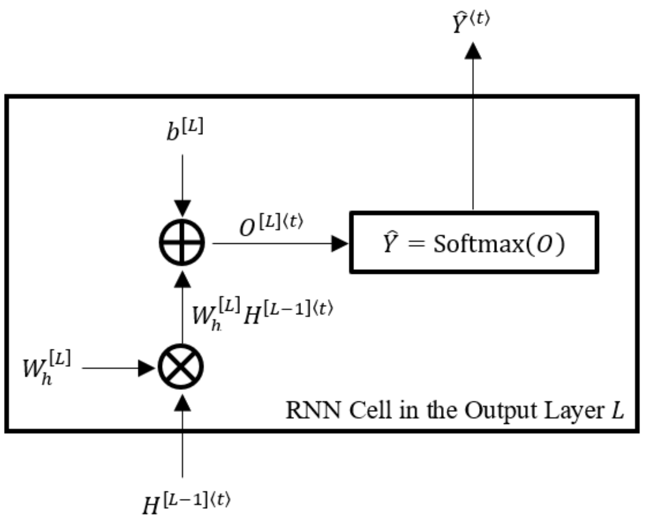

$\hat{Y}^{\langle t\rangle}$
$b^{[L]}$
$O^{[L]\langle t\rangle}$
$\hat{Y} = \text{Softmax}(O)$
$W_h^{[L]}H^{[L-1]\langle t\rangle}$
$W_h^{[L]}$
RNN Cell in the Output Layer L
$H^{[L-1]\langle t\rangle}$

Fig. 2.5: Basic RNN Cell in the output layer L

2.2. Structural Types of RNN

RNNs have different structural types, depending on the specific use case for which they are designed. There are four main structural types of RNNs: one-to-one, one-to-many, many-to-one, and many-to-many (Kedia & Rasu, 2018).

One-to-One: It is the most straightforward type of RNN, which uses a single input and output, as shown in Fig. 2.6. The current input is influenced by the inputs observed before it. Consider the difficulty of forecasting stock market developments. Such a case forecasts the current value based on previous inputs' behavior.

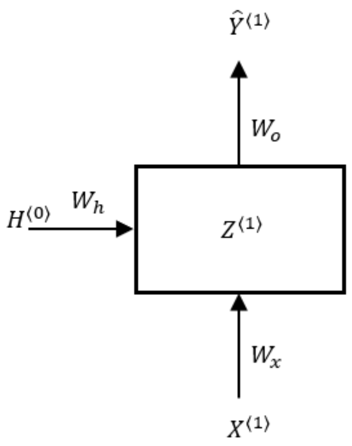


Fig. 2.6: One-to-one architecture

One-to-Many: It uses individual input to produce several outputs. For example, image captioning, in which the image is sent to CNNs for image feature extraction and RNNs for generating captions based on those image features, which generates a sentence of words. A schematic structure of one-to-many architecture is shown in Fig. 2.7.

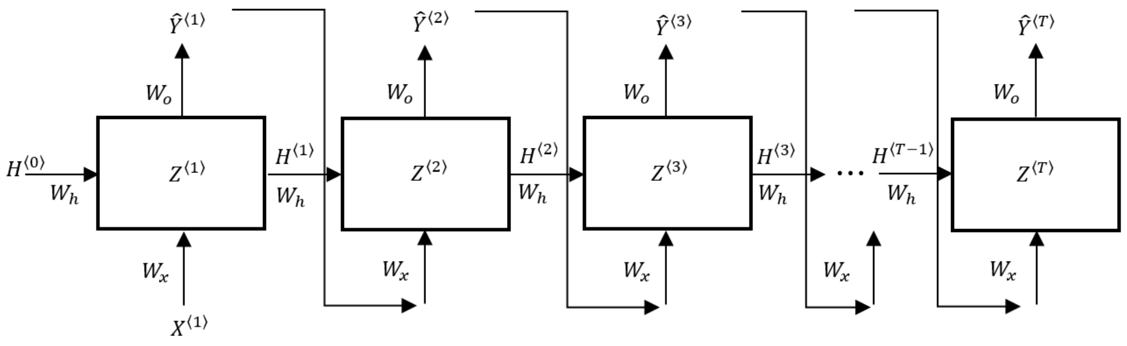


Fig. 2.7: One-to-many architecture

The image features and the earlier created words are used to generate each word in the caption one at a time. A feature extractor trained to extract features from images is used as the input image.

Output Caption: "A cat is sitting on a couch."

The input and output vectors can be represented mathematically in the following ways:

Image Feature Vector: The vector F is typically generated by a pre-trained CNNs applied to the input image.

$$F \;=\; [F_1, F_2, F_3, \cdots, F_n]$$

Vocabulary: $\{1\text{: "A"}, 2\text{: "cat"}, 3\text{: "is"}, 4\text{: "sitting"}, 5\text{: "on"}, 6\text{: "a"}, 7\text{: "couch"}\}$

The output caption is a string is a vocabulary of words, and each word is associated with an index that have been produced repeatedly. $W[t]$ represent the word at time step t defined mathematically as $[0, 0, I, 1, I, 0]$, where the dimension of $W[t]$ is equal to the vocabulary's size. The word created at time step t has an index of 1, represented by the value 1.

Many-to-One: Since its name indicates, this flavor of RNN has only one outcome and multiple inputs. This is depicted in Fig. 2.8.

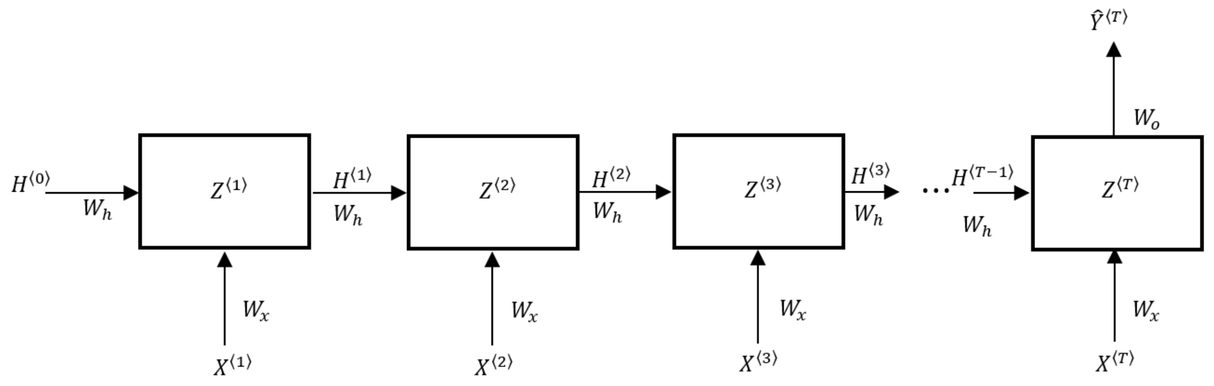


Fig. 2.8: Many-to-one architecture

Sentiment analysis serves as one of its uses, where the input is a sentence and the model determines whether it has a positive or negative sentiment.

Input Sentence: "I loved the movie. It was fantastic!"

Output Sentiment: The sentence's input is related to a favorable or bad feeling.

Output emotion is mathematically defined as either positive (encoded as 1) or negative (encoded as 0).

Vocabulary: {"I": "loved"; "the"; "movie"; "it"; "was"; "fantastic"}

The input sentence is made up of a list of words, each of which is represented by its vocabulary index.

Input Sequence: $X^{\langle 1 \rangle} = \begin{bmatrix}1\\0\\0\\0\\0\\0\\0\end{bmatrix}, X^{\langle 2 \rangle} = \begin{bmatrix}0\\1\\0\\0\\0\\0\\0\end{bmatrix}, X^{\langle 3 \rangle} = \begin{bmatrix}0\\0\\1\\0\\0\\0\\0\end{bmatrix}, X^{\langle 4 \rangle} = \begin{bmatrix}0\\0\\1\\0\\0\\0\\0\end{bmatrix}, X^{\langle 5 \rangle} = \begin{bmatrix}0\\0\\0\\0\\1\\0\\0\end{bmatrix},$

$$X^{\langle 6 \rangle} = \begin{bmatrix}0\\0\\0\\0\\0\\1\\0\end{bmatrix}, X^{\langle 7 \rangle} = \begin{bmatrix}0\\0\\0\\0\\0\\0\\1\end{bmatrix}$$

Many-to-many: This RNN network receives multiple inputs and produces multiple outputs. Regardless of whether the input size is equal to the size of the output it can take either of two different forms. It has two variations:

- Synchronous $(T_x = T_y)$: Here, the input and output sequence lengths are identical. Object identification (NER) is a common application. Consider a sentence:

"John works at Google in California"

To perform NER on this sentence using a synchronous model tokenize the sentence into individual words and assign each word a label indicating whether it is part of a named entity. A possible output sequence for this sentence might be:

Input sequence: *John works at Google in California*

Output sequence: *B-I E E B-Co E B-Geo*

Here, the label *"B-I"* represents the starting of an Individual entity (i.e., word *"John"*), *"B-Co"* tells the beginning of an organization/Company entity (i.e., word *"Google"*), and *"B-Geo"* is the starting point of a geographical location entity (i.e., word *"California"*). The label *"E"* is used for words that do not belong to any named entity. Synchronous RNN is represented in Fig. 2.9.

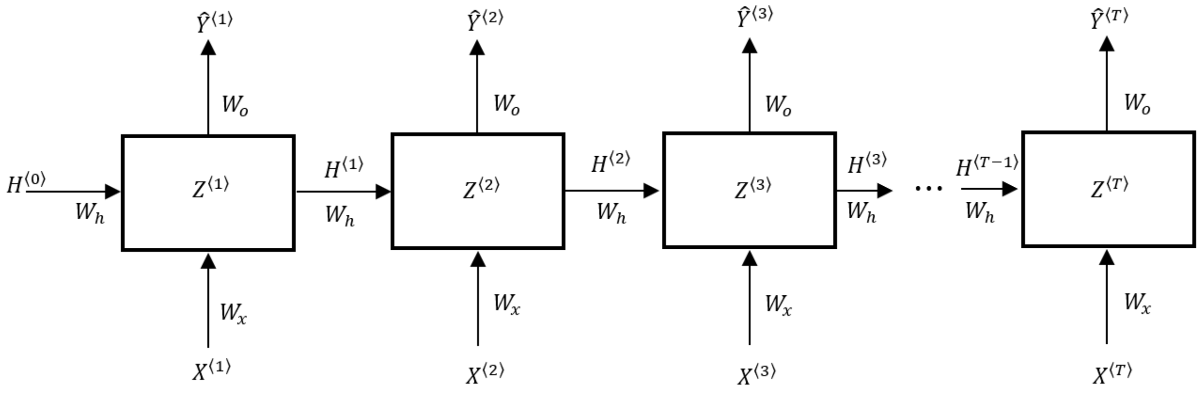


Fig. 2.9: Many-to-many architecture (synchronous)

- Asynchronous $(T_x \neq T_y)$: Asynchronous RNN is used in sequence-to-sequence challenges, such as machine translation the method which attempt to translate one language into another. The input sequence in one language and the output sequence in another are often of different lengths in such issues. Look into the English phrase "How are you". To get the output "wie gehts ", translate it into German. The output is a size of 2 $(T = 2)$ while the input contains a size of 3 $(T = 3)$ shown in Fig. 2.10.

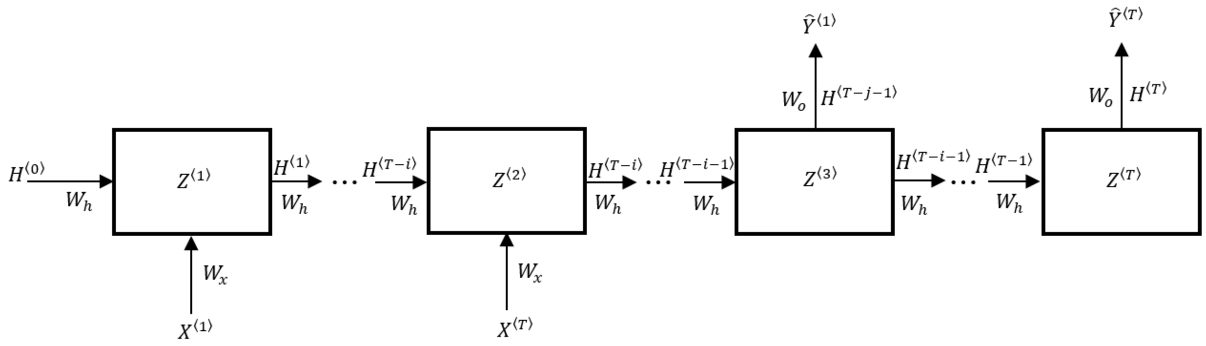


Fig. 2.10: Many-to-many (asynchronous)

2.3. Forward Pass: Finding Neuron Values in a Simple Recurrent Neural Network (RNN)

RNNs manage sequential data by propagating and storing information from one time step to the next, employing feedback loops. RNN involves an additional hidden state using the input at each time step and the hidden state from the previous time step which is then used in the subsequent time step. RNNs can process input sequences with varying lengths using their internal state (memory). The RNN cell will be duplicated five times if the data input sequence consists of five steps. Each cell generates an output hidden state $\left(H^{\langle t\rangle}\right)$ and a vector of probabilities $\left(\hat{Y}^{\langle t\rangle}\right)$ for this time-step by taking as inputs the hidden state from the previous cell $\left(H^{\langle t-1\rangle}\right)$ and the current time-step's input data $\left(X^{\langle t\rangle}\right)$.

The Input sequence $X=\left(X^{\langle 1\rangle}, X^{\langle 2\rangle}, \ldots, X^{\langle t\rangle}\right)$ is carried over T time steps and the network outputs $\hat{Y}=\left(\hat{Y}^{\langle 1\rangle}, \hat{Y}^{\langle 2\rangle}, \ldots., \hat{Y}^{\langle T\rangle}\right)$. Unfold the network for t time steps at every step to get the output at time step $t+1$. The feedforward neural network and the unfolded network are pretty similar. Therefore, using an activation function φ_h (Tanh) for t^{th} time step and p^{th} training instance:

$$Z^{(p)\langle t\rangle} \quad = \quad W_x X^{(p)\langle t\rangle} + W_h H^{(p)\langle t-1\rangle} + b_h \qquad ---(A.1)$$

$X^{\langle t\rangle} \in \mathbb{R}^{n_x}$ is the input at time step t.

$$H^{(p)\langle t\rangle} \quad = \quad \varphi_h\left(Z^{(p)\langle t\rangle}\right) \qquad ---(A.2)$$

$$O^{(p)\langle t\rangle} \quad = \quad W_o H^{(p)\langle t\rangle} + b_o$$

The output $\hat{Y}^{\langle t\rangle}$ at time t is computed as

$$\hat{Y}^{(p)\langle t\rangle} \quad = \quad \varphi_o\left(O^{(p)\langle t\rangle}\right)$$

Here, φ_o is the softmax activation function.

$$H^{\langle 1\rangle} \quad = \quad \varphi_h\left(X^{\langle 1\rangle}, H^{\langle 0\rangle}\right) \qquad ---(A.3)$$

Because of the recursive nature of Eq. $(A.3)$, the recurrent network can compute a function of variable length. Eq. $(A.1)$ can be expanded starting at h_0, which is a constant vector (such as zero vector) such as $H^{\langle 1\rangle}=\varphi_o\left(X^{\langle 1\rangle}, H^{\langle 0\rangle}\right)$ and $H^{\langle 2\rangle}=\left(\varphi_h\left(\varphi_h\left(X^{\langle 1\rangle}, H^{\langle 0\rangle}\right), X^{\langle 2\rangle}\right)\right)$. $H^{\langle 1\rangle}$ is a function of $X^{\langle 1\rangle}$, and $H^{\langle 2\rangle}$ is a function of both $X^{\langle 1\rangle}$ and $X^{\langle 2\rangle}$. $H^{\langle t\rangle}$ is a function of $X^{\langle 1\rangle}, X^{\langle 2\rangle}, \ldots, X^{\langle T\rangle}$. Since the output, $\hat{Y}^{\langle t\rangle}$ is the function of $X^{\langle 1\rangle}$ indirectly.

3. Mathematical Working of RNNs

Mathematics is required to understand the sophisticated mathematical ideas and algorithms behind RNNs like probability theory, calculus and linear algebra which are fundamental mathematical ideas incorporated into RNNs. RNNs execute computations on sequential input using matrix multiplication and vector addition and optimize the network's parameters during training using calculus and probabilistic models for sequential data to make predictions based on the likelihood of various events. Deep RNNs have L number of layers denoted by the superscript $[l]$ that the concerning value is used from layer l. To form the values for p^{th} training example/Instance at the t time step in a layer l, for $l = 1,2,3,\cdots,L$. The notations for expressions are defined as below.

- The matrix of input values at the t^{th} step from the previous layer $(l=0)$ is denoted by

$$X^{[0](p)\langle t\rangle} \quad = \quad \begin{bmatrix} x_1^{[0](p)\langle t\rangle} \\ x_2^{[0](p)\langle t\rangle} \\ \vdots \\ x_{n_x}^{[0](p)\langle t\rangle} \end{bmatrix}_{n_x\times 1} \qquad \text{for } t=1,2,\cdots,T,\;\; p=1,2,3,\cdots,N$$

$$\bar{X}^{[0]\langle p\rangle} = \begin{bmatrix} | & | & & | \\ X^{[0](p)\langle 1\rangle} & X^{[0](p)\langle 2\rangle} & \cdots & X^{[0](p)\langle T\rangle} \\ | & | & & | \end{bmatrix}_{n_x \times T}$$

Here, $\bar{X}^{[0](p)} \in \mathbb{R}^{n_x \times T}$.

- The weights matrices for generating the value at t^{th} time step $t = 1,2,3,\cdots,T$ (in layer l) are denoted by

  Input to Hidden state weight matrix for layer $l = 0$

  $$\left(W_x^{[0]}\right)_{n_h \times n_x} = \left(w_{j,i}^{[0]}\right) \qquad \begin{array}{l} i = 1,2,3,\cdots,n_x \\ j = 1,2,3,\cdots,n_h \end{array}$$

  Hidden to Hidden state weight matrix $l = 1,2,3,\cdots,L-1$

  $$\left(W_h^{[l]}\right)_{n_h \times n_h} = \left(w_{j,i}^{[l]}\right) \qquad \begin{array}{l} i = 1,2,3,\cdots,n_h \\ j = 1,2,3,\cdots,n_h \end{array}$$

  Hidden to Output state weight matrix $l = L$

  $$\left(W_o^{[L]}\right)_{n_y \times n_h} = \left(w_{r,s}^{[L]}\right) \qquad \begin{array}{l} r = 1,2,3,\cdots,n_y \\ s = 1,2,3,\cdots,n_h \end{array}$$

  Here,

  n_h = dimension of hidden vectors

  n_x = dimension of input vectors

  n_y = dimension of output vectors

- The bias vector for a single time step in a sequence for hidden and output connection $\left[b^{[l]}\right]_{n_h \times 1}, \left[b^{[L]}\right]_{n_y \times 1}$respectively, the matrix B for $l = 1,2,3,\cdots,L-1$ between hidden-hidden and hidden-output $(\bar{B})$ layer $l = L$ respectively represented by

$$b^{[l]\langle t\rangle} = \begin{bmatrix} b_1^{[l]} \\ b_2^{[l]} \\ \vdots \\ b_{n_h}^{[l]} \end{bmatrix}_{n_h \times 1}$$

$$B^{[l]} = \begin{bmatrix} | & | & & | \\ b^{[l]\langle 1\rangle} & b^{[l]\langle 2\rangle} & \cdots & b^{[l]\langle T\rangle} \\ | & | & & | \end{bmatrix}_{n_h \times T}$$

Here, $B \in \mathbb{R}^{n_h \times T}$. For $l = L$

$$b^{[l]\langle t\rangle} = \begin{bmatrix} b_1^{[l]} \\ b_2^{[l]} \\ \vdots \\ b_{n_y}^{[l]} \end{bmatrix}_{n_y \times 1}$$

$$\bar{B}^{[l]} = \begin{bmatrix} | & | & & | \\ b^{[l]\langle 1\rangle} & b^{[l]\langle 2\rangle} & \cdots & b^{[l]\langle T\rangle} \\ | & | & & | \end{bmatrix}_{n_y \times T}$$

Here, $\bar{B}^{[l]} \in \mathbb{R}^{n_y \times T}$.

- The hidden 2D structure in recurrent state for layer $l = 1,2,3,\dots,L-1,\ t = 1,2,\cdots,T$ is denoted by

$$H^{[l](p)\langle t\rangle} = \begin{bmatrix} h_1^{[l](p)\langle t\rangle} \\ h_2^{[l](p)\langle t\rangle} \\ \vdots \\ h_{n_h}^{[l](p)\langle t\rangle} \end{bmatrix}_{n_h \times 1}$$

$$H^{[l](p)} = \begin{bmatrix} | & | & & | \\ H^{[l](p)\langle 1\rangle} & H^{[l](p)\langle 2\rangle} & \cdots & H^{[l](p)\langle T\rangle} \\ | & | & & | \end{bmatrix}_{n_h \times T}$$

Here, $H^{[l](p)} \in \mathbb{R}^{n_h \times T}$.

- The matrix of the pre-activation recurrent state for generating the values at t time steps in layer l is denoted by

$$Z^{[l](p)\langle t\rangle} = \begin{bmatrix} z_1^{[l](p)\langle t\rangle} \\ z_2^{[l](p)\langle t\rangle} \\ \vdots \\ z_{n_h}^{[l](p)\langle t\rangle} \end{bmatrix}_{n_h \times 1} \qquad \text{for } t = 1,2,\cdots,T$$

$$\bar{Z}^{[l](p)} = \begin{bmatrix} | & | & & | \\ Z^{[l](p)\langle 1\rangle} & Z^{[l](p)\langle 2\rangle} & \cdots & Z^{[l](p)\langle T\rangle} \\ | & | & & | \end{bmatrix}_{n_h \times T}$$

Here, $\bar{Z}^{[l](p)} \in \mathbb{R}^{n_h \times T}$.

- The matrix of pre-activation outputs at the last layer $l = L$ is denoted by

$$O^{[l](p)\langle t\rangle} = \begin{bmatrix} o_1^{[l](p)\langle t\rangle} \\ o_2^{[l](p)\langle t\rangle} \\ \vdots \\ o_{n_y}^{[l](p)\langle t\rangle} \end{bmatrix}_{n_y \times 1}$$

$$\bar{O}^{[l](p)} = \begin{bmatrix} | & | & & | \\ O^{[l](p)\langle 1\rangle} & O^{[l](p)\langle 2\rangle} & \cdots & O^{[l](p)\langle T\rangle} \\ | & | & & | \end{bmatrix}_{n_y \times T}$$

Here, $\bar{O}^{[l](p)} \in \mathbb{R}^{n_y \times T}$.

- The matrix $\hat{Y}$ of probabilistic output values $\hat{Y}^{(p)} = W_o^{[l]} H^{[l](p)} + \bar{B}$ generated at the t time steps in layer $l = L$ is denoted by

$$\hat{Y}^{(p)\langle t\rangle} = \begin{bmatrix} \hat{y}_1^{(p)\langle t\rangle} \\ \hat{y}_2^{(p)\langle t\rangle} \\ \vdots \\ \hat{y}_{n_z}^{(p)\langle t\rangle} \end{bmatrix}_{n_y \times 1} \quad \text{for } t = 1,2,\cdots,T$$

$$\hat{\bar{Y}}^{(p)} = \begin{bmatrix} | & | & & | \\ \hat{Y}^{(p)\langle 1\rangle} & \hat{Y}^{(p)\langle 2\rangle} & \cdots & \hat{Y}^{(p)\langle T\rangle} \\ | & | & & | \end{bmatrix}_{n_y \times T}$$

Here, $\hat{\bar{Y}}^{(p)} \in \mathbb{R}^{n_y \times T}$.

- The matrix of targeted/actual values at the t time step in the output layer L is denoted by

$$Y^{(p)\langle t\rangle} = \begin{bmatrix} y_1^{(p)\langle t\rangle} \\ y_3^{(p)\langle t\rangle} \\ y_3^{(p)\langle t\rangle} \\ \vdots \\ y_{n_y}^{(p)\langle t\rangle} \end{bmatrix}_{n_y \times 1} \quad \text{for } t = 1,2,\cdots,T$$

$$\bar{Y}^{(p)} = \begin{bmatrix} | & | & & | \\ Y^{(p)\langle 1\rangle} & Y^{(p)\langle 2\rangle} & \cdots & Y^{(p)\langle T\rangle} \\ | & | & & | \end{bmatrix}_{n_y \times T}$$

Here, $\bar{Y}^{(p)} \in \mathbb{R}^{n_y \times T}$.

3.1. Forward Propagation in a Deep RNN

In a deep RNN, each layer receives the input from the previous layer and passes its output to the next layer. The input to the first layer is usually a sequence of vectors and the output from the last layer is also a sequence of vectors (maybe the same dimension). In a multilayer RNNs, forward propagation involves computing the output of each layer for every instance at each time step. Deep RNN is used for more complex tasks such as NLP and Time Series Analysis (TSA). Single-layer RNN structure is used only for learning to construct more complicated models in real-world applications Multilayer networks are used.

The deep RNN displayed in Fig. 3.1 has multiple hidden or recurrent layers, which can be more computationally expensive than Simple RNN due to the multiple hidden layers. When the network is trained to carry out a task that requires predicting the future from the past, the network typically learns to use hidden states $H^{\langle t\rangle}$, the past sequence of inputs up to T. Deep RNNs go one step further by stacking single layer RNN on top of each other. Since it converts a series of any length $\left(X^{\langle T\rangle}, X^{\langle T-1\rangle}, X^{\langle T-2\rangle}, \dots, X^{\langle 2\rangle}, X^{\langle 1\rangle}\right)$ to a vector of a fixed length $H^{\langle t\rangle}$, the vector for the hidden states at time-step t and layer l must be added as a superscript to the hidden state denoted by $H^{[l]\langle t\rangle}$ for the higher-up layers to distinguish between the hidden nodes. The weights are shared over various time steps (like in the single-layer RNN) but not across various layers, so the weight matrix for the lth hidden layer is represented as $W_h^{[l]}$.

For any instance p (having T time steps/words), a general RNN involving L layers is shown in Fig. 3.1.

For n_x- dimensional input vector dimensions are stated as follows:

- $X^{[0]\langle t\rangle} \in \mathbb{R}^{n_x}$ is the input at time step t.
- $\hat{Y}^{\langle t\rangle} \in \mathbb{R}^{n_y}$ is the predicted output of the network at time step t.
- $H^{[l]\langle t\rangle} \in \mathbb{R}^{n_h}$ vector stores the values of the hidden units/states at time t.
- $H^{[l]\langle 0\rangle}$ hidden vector at $t = 0$ initialized to zero.
- $W_x^{[0]} \in \mathbb{R}^{n_h \times n_x}$ are weights associated with the inputs-recurrent layer.
- $W_h^{[l]} \in \mathbb{R}^{n_h \times n_h}$ are weights associated with hidden units in the recurrent layers?
- $W_0^{[L]} \in \mathbb{R}^{n_y \times n_h}$ are weights associated with hidden units to output units.
- $b^{[l]} \in \mathbb{R}^{n_h}$ is the bias associated with the recurrent layer.
- $b^{[L]} \in \mathbb{R}^{n_y}$ is the bias associated with the output layer.

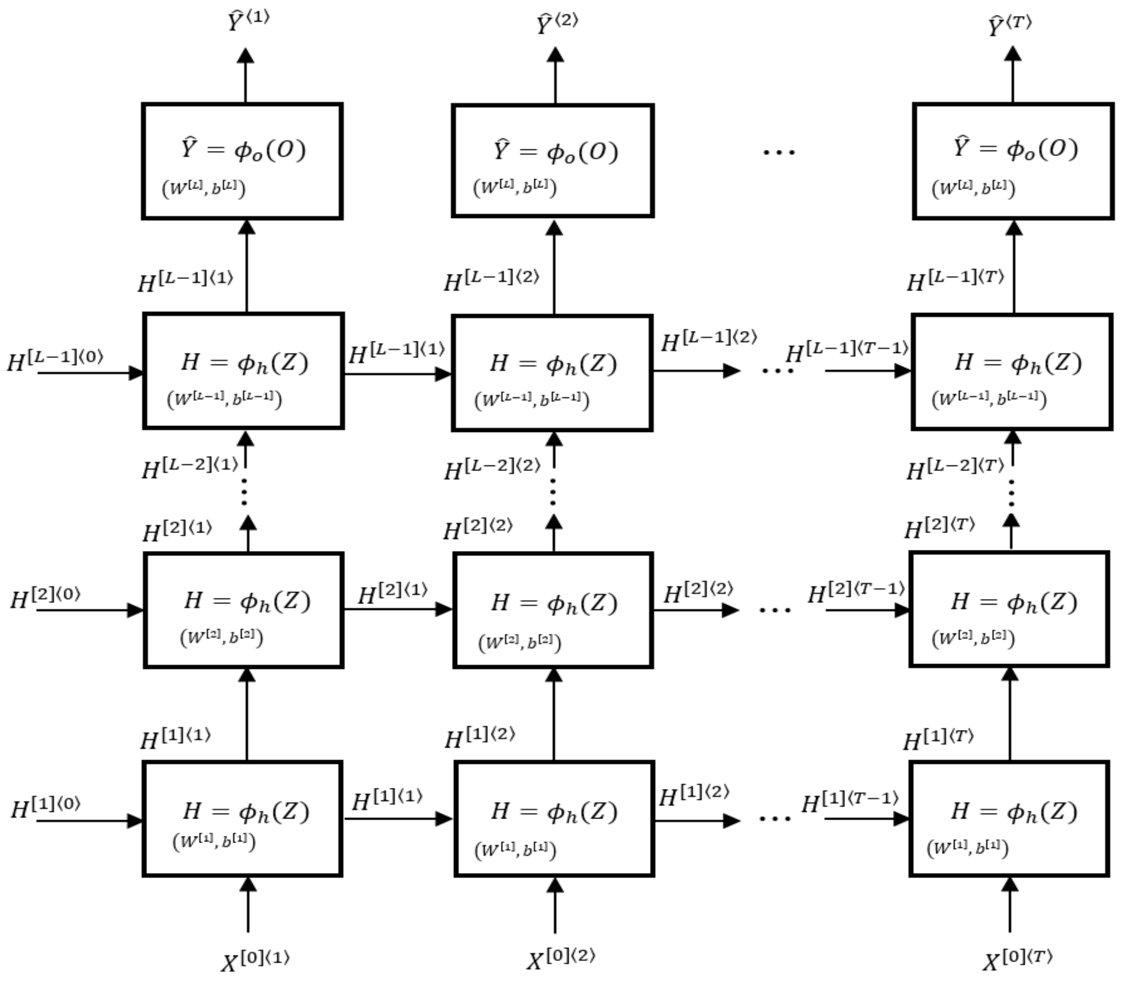

$\hat{Y}^{\langle 1 \rangle}$
$\hat{Y}^{\langle 2 \rangle}$
$\hat{Y}^{\langle T \rangle}$
$\hat{Y} = \phi_o(O)$
$(W^{[L]}, b^{[L]})$
$H^{[L-1]\langle 1 \rangle}$
$H^{[L-1]\langle 2 \rangle}$
$H^{[L-1]\langle T \rangle}$
$H^{[L-1]\langle 0 \rangle}$
$H^{[L-1]\langle T-1 \rangle}$
$H = \phi_h(Z)$
$(W^{[L-1]}, b^{[L-1]})$
$H^{[L-2]\langle 1 \rangle}$
$H^{[L-2]\langle 2 \rangle}$
$H^{[L-2]\langle T \rangle}$
$H^{[2]\langle 1 \rangle}$
$H^{[2]\langle 2 \rangle}$
$H^{[2]\langle T \rangle}$
$H^{[2]\langle 0 \rangle}$
$H^{[2]\langle T-1 \rangle}$
$(W^{[2]}, b^{[2]})$
$H^{[1]\langle 1 \rangle}$
$H^{[1]\langle 2 \rangle}$
$H^{[1]\langle T \rangle}$
$H^{[1]\langle 0 \rangle}$
$H^{[1]\langle T-1 \rangle}$
$(W^{[1]}, b^{[1]})$
$X^{[0]\langle 1 \rangle}$
$X^{[0]\langle 2 \rangle}$
$X^{[0]\langle T \rangle}$

Fig. 3.1: Schematic structure of a multilayer/deep RNN

3.2. Steps for Forward Pass of Deep RNNs

The input sequence is fed into the input layer of the network. Each input vector $X^{[0](p)\langle t \rangle}$ is passed to the first layer $l = 1$ of recurrent cells which computes its output based on the input vector and its previous hidden $H^{[l](p)\langle 0 \rangle}$ state of layer l. This output $H^{[l](p)\langle 1 \rangle}$ is passed to the next $l + 1$ layer of recurrent cells. The layer $l + 1$ of recurrent cells computes its output based on the first layer l's output and its previous state $H^{[l+1](p)\langle 0 \rangle}$. This output is passed to the third layer $l + 2$ of recurrent cells and so on. The output of the final layer of recurrent cells is the output sequence of the network that can be computed using the following steps:

Step 1 (Find the product of input, hidden vector, and corresponding weight matrices):

As a vectors (Embedding) representation of a word in data is represented as input vectors $X^{[0](p)\langle t \rangle}$ which will multiply $X^{[0](p)\langle t \rangle}$ with the weight matrix $\left(W_x^{[l]}\right)_{n_h \times n_x}$ and $H^{[l](p)\langle t-1 \rangle}$ with $\left(W_h^{[l]}\right)_{n_h \times n_h}$ and sum all the multiplied vectors and matrices (linear combination). A weight matrix represents the strength of the connection between cells and decides how much influence the given input will have on the cell's output.

1. For $t = 1,2,\cdots,T$, $l = 1$

$$Z^{[l](p)\langle t \rangle} = W_x^{[0]} X^{[0](p)\langle t \rangle} + W_h^{[l]} H^{[l](p)\langle t-1 \rangle} \qquad ---(F.1)$$

By stacking $X^{[0](p)\langle t\rangle}$ and $H^{[l](p)\langle t-1\rangle}$ together horizontally

$$Z^{[l](p)\langle t\rangle} = W^{[1]}\begin{bmatrix} X^{[0](p)\langle t\rangle} \\ H^{[l](p)\langle t-1\rangle}\end{bmatrix}$$

or,

$$\mathrm{N}^{[1](p)\langle t\rangle} = W^{[1]}\begin{bmatrix} | \\ X^{[0](p)\langle t\rangle} \\ | \\ \\ | \\ H^{[l](p)\langle t-1\rangle} \\ | \end{bmatrix}_{(n_x+n_h)\times 1} = W^{[1]}\mathrm{N}^{[1](p)\langle t\rangle}$$

Here, $W^{[1]}$ is the concatenating matrix stacking horizontally s.t:

$$(W_{xh})_{n_h\times(n_x+n_h)} = \left[W_x^{[0]} \mid W_h^{[l]}\right]_{n_h\times(n_x+n_h)}$$

For $l = 2,3,\dots,L-1$.

$$Z^{[l](p)\langle t\rangle} = W_h^{[l]}H^{[l-1](p)\langle t\rangle} + W_h^{[l]}H^{[l](p)\langle t-1\rangle} \qquad ---(F.2)$$

by stacking $H^{[l-1](p)\langle t\rangle}$ and $H^{[l](p)\langle t-1\rangle}$ together horizontally

$$Z^{[l](p)\langle t\rangle} = W^{[l]}\begin{bmatrix} H^{[l-1](p)\langle t\rangle} \\ H^{[l](p)\langle t-1\rangle}\end{bmatrix}$$

$$\mathrm{N}^{[l](p)\langle t\rangle} = W^{[l]}\begin{bmatrix} | \\ H^{[l-1](p)\langle t\rangle} \\ | \\ \\ | \\ H^{[l](p)\langle t-1\rangle} \\ | \end{bmatrix}_{2n_h\times 1} = W^{[l]}\mathrm{N}^{[l](p)\langle t\rangle}$$

Here, $W^{[l]}$ is the concatenating matrix stacking vertically.

$$(W_{hh})_{n_h\times 2n_h} = \left[W_h^{[l]} \mid W_h^{[l]}\right]_{n_h\times 2n_h}$$

The computation for the output layer $l = L$

$$O^{[l](p)\langle t\rangle} = W_o^{[l]}H^{[l-1](p)\langle t\rangle} \qquad ---(F.3)$$

Here, $O^{[l](p)\langle t\rangle}$ is the vector of intermediate/pre-activation/non-probabilistic values of output for instance p at time step t.

Step 2 (Adding Biases)

Add bias vector b_h to the linear combination of input hidden vectors with weight matrices called Z and O for the hidden and output layers. Bias is also known as the offset to the activation function of each node in the RNN layers shifting the function up or down that is added to the hidden layers and the output layer to represent complex temporal patterns and dependencies in the input sequence even when the input is noisy or incomplete or zero. The network should often produce a non-zero output based on prior knowledge when all the input values are zero (no previous context or information).

- The vector $Z^{[l](p)\langle t\rangle}$ for generating the values in the layer $l = 1$ at each time step with bias term is denoted by

For Input-hidden layer

$$Z^{[l](p)\langle t\rangle} = W_x^{[0]} X^{[0](p)\langle t\rangle} + W_h^{[l]} H^{[l](p)\langle t-1\rangle} + b^{[l]} \qquad ---(F.4)$$

$$Z^{[l](p)\langle t\rangle} = W_{xh}^{[1]} \begin{bmatrix} X^{[0](p)\langle t\rangle} \\ H^{[l](p)\langle t-1\rangle} \end{bmatrix} + b^{[l]}$$

In matrix notation, Its written as:

$$\begin{bmatrix} z_1^{[l](p)\langle t\rangle} \\ z_2^{[l](p)\langle t\rangle} \\ \vdots \\ z_{n_h}^{[l](p)\langle t\rangle} \end{bmatrix}_{n_h\times 1} = \begin{bmatrix} w_{1,1}^{[l-1]} & w_{1,2}^{[l-1]} & \cdots & w_{1,i}^{[l-1]} \\ w_{2,1}^{[l-1]} & w_{2,2}^{[l-1]} & \cdots & w_{2,i}^{[l-1]} \\ \vdots & \vdots & \vdots & \vdots \\ w_{j,1}^{[l-1]} & w_{j,2}^{[l-1]} & \cdots & w_{j,i}^{[l-1]} \end{bmatrix}_{n_h\times n_x} \begin{bmatrix} x_1^{[0](p)\langle t\rangle} \\ x_2^{[0](p)\langle t\rangle} \\ \vdots \\ x_{n_x}^{[0](p)\langle t\rangle} \end{bmatrix}_{n_x\times 1}$$
$$+ \begin{bmatrix} w_{1,1}^{[l]} & w_{1,2}^{[l]} & \cdots & w_{1,i}^{[l]} \\ w_{2,1}^{[l]} & w_{2,2}^{[l]} & \cdots & w_{2,i}^{[l]} \\ \vdots & \vdots & \vdots & \vdots \\ w_{j,1}^{[l]} & w_{j,2}^{[l]} & \cdots & w_{j,i}^{[l]} \end{bmatrix}_{n_h\times n_h} \begin{bmatrix} h_1^{[l](p)\langle t\rangle} \\ h_2^{[l](p)\langle t\rangle} \\ \vdots \\ h_{n_h}^{[l](p)\langle t\rangle} \end{bmatrix}_{n_h\times 1} + \begin{bmatrix} b_1^{[l]} \\ b_2^{[l]} \\ \vdots \\ b_{n_h}^{[l]} \end{bmatrix}_{n_h\times 1}$$

After concatenation/stacking vertically in a large matrix $W_{xh}^{[l]} = \left[W_x^{[0]} \mid W_h^{[l]}\right]$
In matrix notation,

$$\left[W_{xh}^{[l]}\right]_{n_h\times(n_x+n_h)} = \begin{bmatrix} w_{1,1}^{[l-1]} & w_{1,2}^{[l-1]} & \cdots & w_{1,i}^{[l-1]} & w_{1,1}^{[l]} & w_{1,2}^{[l]} & \cdots & w_{1,i}^{[l]} \\ w_{2,1}^{[l-1]} & w_{2,2}^{[l-1]} & \cdots & w_{2,i}^{[l-1]} & w_{2,1}^{[l]} & w_{2,1}^{[l]} & \cdots & w_{2,i}^{[l]} \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ w_{j,1}^{[l-1]} & w_{j,2}^{[l-1]} & \cdots & w_{j,i}^{[l-1]} & w_{j,1}^{[l]} & w_{j,2}^{[l]} & \cdots & w_{j,i}^{[l]} \end{bmatrix}$$

For $l = 1$, In complex deep structure write it as for notational convenience.

$$\left[\bar{Z}^{[l](p)}\right]_{n_h\times T} = \left[\left[W_{xh}^{[l]}\right]_{n_h\times(n_x+n_h)} \left[\mathrm{N}^{[1](p)\langle t\rangle}\right]_{(n_x+n_h)\times T} + \left[B^{[l]}\right]_{n_h\times T}\right]_{n_h\times T}$$

The initial hidden layer is unique because it receives inputs from the adjacent hidden state from the prior time step and from the input layer at the same time step. The matrix $W_{xh}^{[l]}$ will have a size of $n_h \times (n_x + n_h)$ only for the 1st layer ($l = 1$). Note that n_h will not be the same as n_x.

For hidden-hidden layer: Let's emphasize all the recurrent layers l for $l \geq 2$. It is noticed that Eq. $(F.5)$ above is pretty similar to the recurrence condition for the layers with $l \geq 2$:

$$Z^{[l](p)\langle t\rangle} = W_h^{[l]} H^{[l-1](p)\langle t\rangle} + W_h^{[l]} H^{[l](p)\langle t-1\rangle} + b^{[l]} \qquad ---(F.5)$$

$$Z^{[l](p)\langle t\rangle} = W_{hh}^{[l]} \left[H^{[l](p)\langle t-1\rangle}, H^{[l-1](p)\langle t\rangle}\right]^{\mathbf{T}} + b^{[l]}$$

In matrix notation,

$$\begin{bmatrix} z_1^{[l](p)\langle t\rangle} \\ z_2^{[l](p)\langle t\rangle} \\ \vdots \\ z_{n_h}^{[l](p)\langle t\rangle} \end{bmatrix}_{n_h\times 1} = \begin{bmatrix} w_{1,1}^{[l-1]} & w_{1,2}^{[l-1]} & \cdots & w_{1,i}^{[l-1]} \\ w_{2,1}^{[l-1]} & w_{2,2}^{[l-1]} & \cdots & w_{2,i}^{[l-1]} \\ \vdots & \vdots & \vdots & \cdots \\ w_{j,1}^{[l-1]} & w_{j,2}^{[l-1]} & \cdots & w_{j,i}^{[l-1]} \end{bmatrix}_{n_h\times n_h} \begin{bmatrix} h_1^{[l-1](p)\langle t\rangle} \\ h_2^{[l-1](p)\langle t\rangle} \\ \vdots \\ h_{n_h}^{[l-1](p)\langle t\rangle} \end{bmatrix}_{n_h\times 1}$$
$$+ \begin{bmatrix} w_{1,1}^{[l]} & w_{1,2}^{[l]} & \cdots & w_{1,i}^{[l]} \\ w_{2,1}^{[l]} & w_{2,2}^{[l]} & \cdots & w_{2,i}^{[l]} \\ \vdots & \vdots & \vdots & \cdots \\ w_{j,1}^{[l]} & w_{j,2}^{[l]} & \cdots & w_{j,i}^{[l]} \end{bmatrix}_{n_h\times n_h} \begin{bmatrix} h_1^{[l](p)\langle t\rangle} \\ h_2^{[l](p)\langle t\rangle} \\ \vdots \\ h_{n_h}^{[l](p)\langle t\rangle} \end{bmatrix}_{n_h\times 1} + \begin{bmatrix} b_1^{[l]} \\ b_2^{[l]} \\ \vdots \\ b_{n_h}^{[l]} \end{bmatrix}_{n_h\times 1}$$

The concatenated matrix and vector for the hidden layers l $(l \geq 2)$ together a large matrix $W_{hh}^{[l]} = \left[W_h^{[l-1]} \mid W_h^{[l]}\right]$ that includes the column of $W_h^{[l]}$ and $W_h^{[l-1]}$.

$$\left(W_{hh}^{[l]}\right)_{n_h\times 2n_h} = \begin{bmatrix} w_{1,1}^{[l-1]} & w_{1,2}^{[l-1]} & \cdots & w_{1,i}^{[l-1]} & w_{1,1}^{[l]} & w_{1,2}^{[l]} & \cdots & w_{1,i}^{[l]} \\ w_{2,1}^{[l-1]} & w_{2,2}^{[l-1]} & \cdots & w_{2,i}^{[l-1]} & w_{2,1}^{[l]} & w_{2,1}^{[l]} & \cdots & w_{2,i}^{[l]} \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ w_{j,1}^{[l-1]} & w_{j,2}^{[l-1]} & \cdots & w_{j,i}^{[l-1]} & w_{j,1}^{[l]} & w_{j,2}^{[l]} & \cdots & w_{j,i}^{[l]} \end{bmatrix}$$

The vertically stacked matrix of hidden vectors is

$$\left[\bar{Z}^{[l](p)}\right]_{n_h\times T} = \left[\left[W_{hh}^{[l]}\right]_{n_h\times 2n_h} \left[\mathrm{M}^{[l](p)\langle t\rangle}\right]_{2n_h\times T} + \left[\bar{B}^{[l]}\right]_{n_h\times T}\right]_{n_h\times T}$$

In this case, the horizontally stacked matrix $W_{hh}^{[l]}$ size is $n_h \times (n_h + n_h) = n_h \times 2n_h$.

For hidden-output layer

$$O^{[L-1](p)\langle t\rangle} = W_o^{[L]} H^{[L-1](p)\langle t\rangle} + b^{[L]} \qquad ---(F.6)$$

Similarly, In matrix notation for $l = L$

$$\begin{bmatrix} o_1^{[L-1](p)\langle t\rangle} \\ o_2^{[L-1](p)\langle t\rangle} \\ \vdots \\ o_{n_{ly}}^{[L-1](p)\langle t\rangle} \end{bmatrix}_{n_y\times 1} = \begin{bmatrix} w_{1,1}^{[L]} & w_{1,2}^{[L]} & \cdots & w_{1,s}^{[L]} \\ w_{2,1}^{[L]} & w_{2,2}^{[L]} & \cdots & w_{2,s}^{[L]} \\ \vdots & \vdots & \vdots & \vdots \\ w_{r,1}^{[L]} & w_{r,2}^{[L]} & \cdots & w_{r,s}^{[L]} \end{bmatrix}_{n_y\times n_h} \begin{bmatrix} h_1^{[L-1](p)\langle t\rangle} \\ h_2^{[L-1](p)\langle t\rangle} \\ \vdots \\ h_{n_h}^{[L-1](p)\langle t\rangle} \end{bmatrix}_{n_h\times 1} + \begin{bmatrix} b_1^{[L]} \\ b_2^{[L]} \\ \vdots \\ b_{n_y}^{[L]} \end{bmatrix}_{n_y\times 1}$$

Step 3 (Applying Activation Function)

Pass the value of Z to an activation function $H = \varphi_h(Z)$ (Obtained by applying an appropriate activation function on Z). Activation functions introduce non-linearity in the cell output, without which the neural network will be a linear function. Also, it has the learning speed of the neural network. The activation function decides whether a node should be activated and whether the information the node is receiving is relevant to the given information or should be ignored. Several definitions of the activation functions are available in the literature, such as the Sigmoid, Tanh, ReLU, Leaky ReLU, Softmax functions etc. However, use the Tanh activation function here because it is non-linear and the values lie between -1 and 1 which helps prevent the gradient from exploding during the backpropagation but may cause the vanishing gradient. In addition, the Tanh function has a symmetric S-shaped curve, which means that it is centered around zero. It can help model data that has a mean of zero such as data that has been standardized or normalized. The vector $H^{[l](p)\langle t\rangle}$ of output values generated at the t cell in layer $l = 1,2,3,\ldots,L-1$ is denoted by.

$$H^{[l](p)\langle t\rangle} \quad = \quad \varphi_h\left(Z^{[l](p)\langle t\rangle}\right)$$

Here, φ_h represents the tanh activation function.
The transformation from hidden to output layer $l = L$ remains the same as in the FFNN network.

$$\hat{Y}^{[l](p)\langle t\rangle} \quad = \quad \varphi_o\left(O^{[L](p)\langle t\rangle}\right)$$

Here, φ_o represents the softmax activation function.

4. Learning Phase

The learning phase of an RNN refers to training the network using the backpropagation through time (BPTT) algorithm. During the learning phase, the RNN adjusts its internal parameters (weights and biases) to minimize the difference between the predicted and actual output for each input sequence in the training data. The RNN weights and biases are updated using an optimization algorithm such as minibatch gradient descent (MBGD) which adjusts the weights and biases in the direction of the negative gradient of the loss function. The learning phase is repeated for multiple epochs (one complete pass through the training data).

The learning phase involves the following steps:

4.1. Backward Pass

The gradient of the loss function concerning the RNN parameters is computed using the BPTT algorithm.

Backpropagation Through Time (BPTT): BPTT is an algorithm used to train RNNs. It is a generalization of the backpropagation algorithm used to train feedforward neural networks. The key idea behind BPTT is to apply the chain rule to calculate the gradients of a recurrent neural network concerning its weights (Goodfellow, 2016). The procedure of backpropagation is depicted in Fig. 4.1.

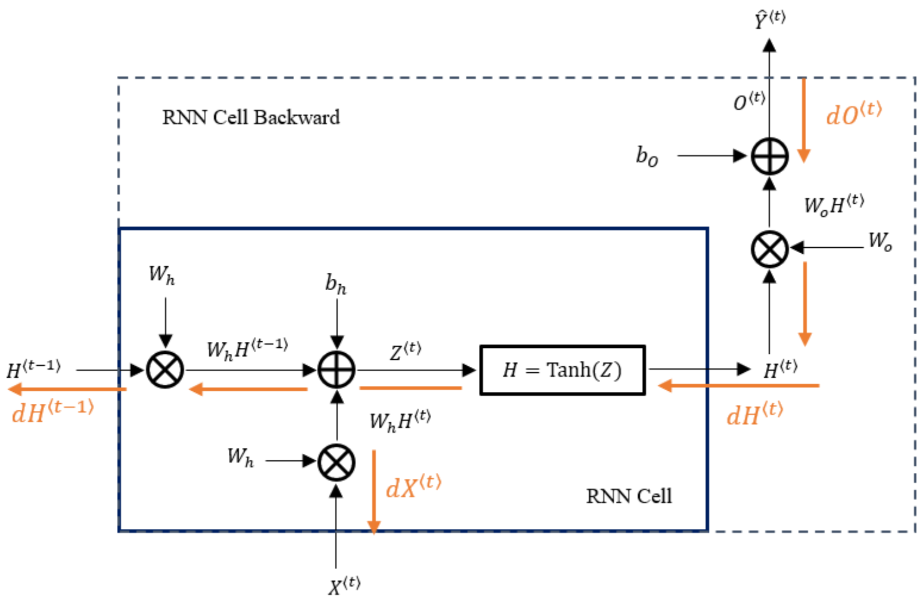


Fig. 4.1: Backpropagation through time; a single RNN cell

The RNN cells are connected cyclically, so the input to a node is a weighted sum of the outputs from the previous node in the network, and the output of a node is a non-linear transformation of its input. First, calculate the derivative of the loss function (the error) concerning the network weights. Calculate the derivative of the loss function concerning each weight by using the chain rule. For this purpose, start by computing the loss function's derivatives concerning the network's output node. During the backpropagation process in RNNs, the entire sequence rather than just a single input is taken into account. The stages of the process of calculating the gradients are shown in Fig. 4.2.

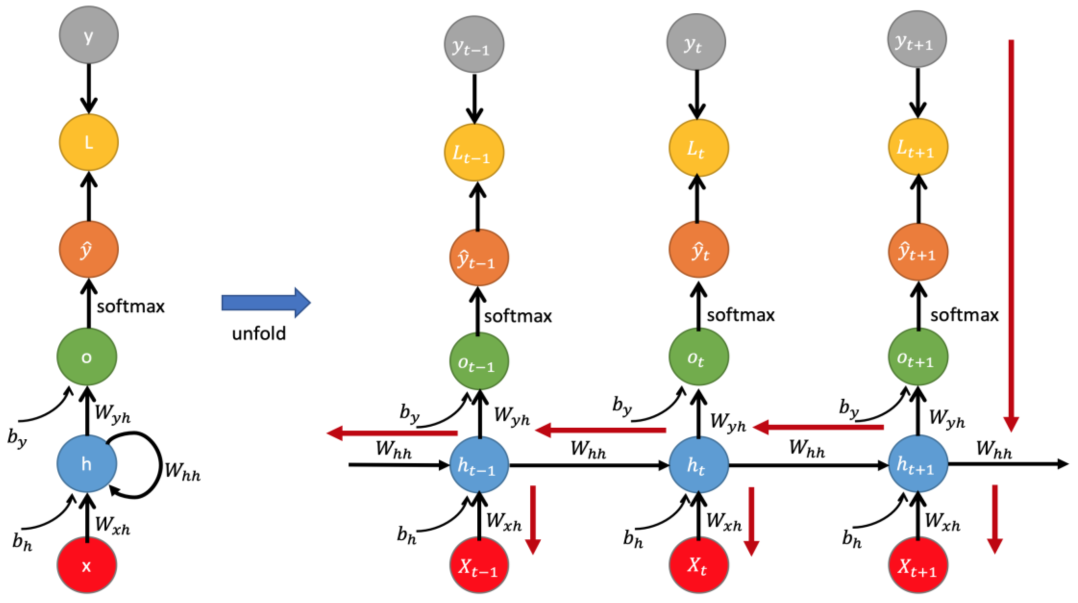


Fig. 4.2: Backpropagation Through Time (Image Credits: Arat, 2019)

4.2. Loss Computation

In loss computation, the predicted output is compared to the actual output at each time step. It measures the difference between predicted and actual output. It takes a target vector/word $\left(Y^{(p)\langle t\rangle}\right)$ and a predicted vector/word $\left(\hat{Y}^{(p)\langle t\rangle}\right)$ as input and gives the accuracy score. The higher this score, the worse the model's prediction is.

Categorical Cross-Entropy Loss: In RNNs the input sequence is fed into the network one timestep at a time and the output at each timestep is a probability distribution over the possible classes. Cross-entropy loss is used for multi-classification problems with three or more possible classes. It measures the difference between the predicted and actual probability distributions across all classes. The categorical cross-entropy loss is calculated for each timestep to compute the loss for the entire sequence and then summed over all timesteps.

Assuming a sequence of length T and K possible classes the loss function for a single sequence/ instance can be written as:

$$E^{\langle t\rangle}\left(Y^{(p)\langle t\rangle}, \hat{Y}^{(p)\langle t\rangle}\right) = -\sum_{t=1}^{T}\sum_{k=1}^{K} Y(t,k)\log\left(\hat{Y}^{(p)}(t,k)\right)$$

or

$$E^{\langle t\rangle}\left(Y^{(p)\langle t\rangle}, \hat{Y}^{(p)\langle t\rangle}\right) = -\sum_{t=1}^{T}\sum_{k=1}^{K} Y^{(p)\langle t\rangle}(k)\log\left(\hat{Y}^{(p)\langle t\rangle}(k)\right) \qquad ---(F.7)$$

$Y^{(p)\langle t\rangle}(k)$ is the actual label for the k^{th} class at timestep t and $\hat{Y}^{(p)\langle t\rangle}(k)$ is the predicted probability for the k^{th} class at timestep t. The negative sign in the categorical cross-entropy loss function ensures the loss value is always non-negative. An entire sequence (sentence) is considered as one training example so the total error is just the sum of the errors at each time step (word). The negative logarithm function has a range of $(-\infty, 0]$, so the negative sign is added to make the loss value positive and ensure that it can be used as a valid objective function to optimize during training. The optimization algorithms aim to minimize the loss value, so minimizing the negative of the loss value is equivalent to maximizing the actual loss value. The graphs of log(x) and -log(x) are shown in Fig. 4.3.

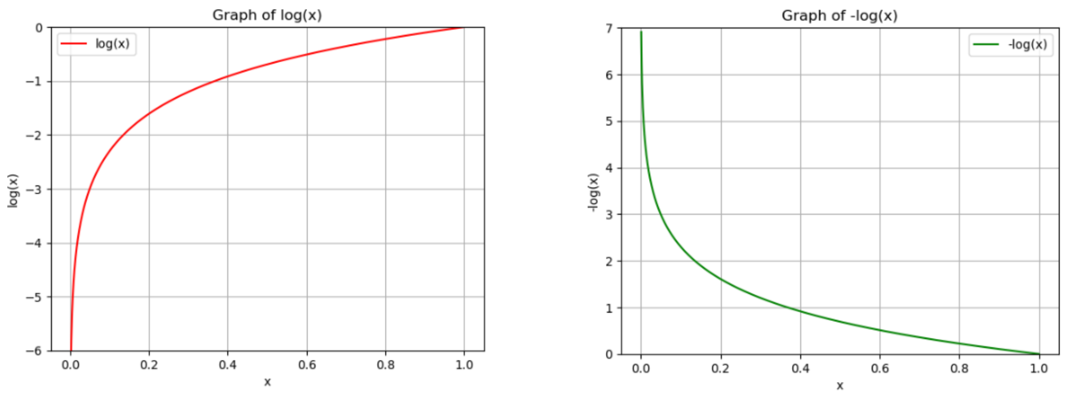


Fig. 4.3: Categorical cross entropy loss

Binary Cross-Entropy Loss: The Binary Cross-Entropy Loss function is a widely used metric in deep learning. It measures the difference between the predicted probabilities and actual binary labels in classification tasks where only two predicted outputs $(0 \; or \; 1)$. This function benefits binary classification problems where each example belongs to one of two complementary classes.

Premises of Binary Classification Problems

- The result of one observation does not affect the result of any other examples.
- Results of all instances/Observations in the dataset are independent.
- All instances are generated from the same underlying distribution.

If all the instances in data follow 2 and 3 premises which makes them independent and Identically Distributed (ID) which makes various statistical analyses more accessible and more robust. Considering the data's ID nature the calculations become much more straightforward.

Binary classification problems are used in predicting the probability that a given example belongs to the positive class (target class), while the probability of it belonging to the negative class (complementary class) be derived from it (Saxena, 2021). Therefore, its needed to predict the probability of one class, and the other class's probability can be inferred using the complement rule of probability $\left(1 - \hat{Y}^{(p)\langle t\rangle}\right)$.

Assuming a sequence of length T and a binary class label $Y^{(p)\langle t\rangle}$ ($Y^{(p)\langle t\rangle} = 0$ or 1), define the output of the RNN at timestep t as a scalar value $\hat{Y}^{[L](p)\langle t\rangle}$ which represents the probability of the positive class ($Y^{(p)\langle t\rangle} = 1$) at timestep t. The probability of the hostile class is ($Y^{(p)\langle t\rangle} = 0$).

$$P\left(Y^{(p)\langle t\rangle} = 1 | X^{[0](p)\langle t\rangle}\right) = \hat{Y}^{(p)\langle t\rangle}$$

$$P\left(Y^{(p)\langle t\rangle} = 0 | X^{[0](p)\langle t\rangle}\right) = 1 - P\left(Y^{(p)\langle t\rangle} = 1 | x\right) = 1 - \hat{Y}^{(p)\langle t\rangle}$$

In another form,

$$P\left(Y^{(p)\langle t\rangle} = 0 | X^{[0](p)\langle t\rangle}\right) = \begin{cases} \hat{Y}^{(p)\langle t\rangle} & \text{if } Y^{(p)\langle t\rangle} = 1 \\ 1 - \hat{Y}^{(p)\langle t\rangle} & \text{if } Y^{(p)\langle t\rangle} = 0 \end{cases}$$

Based on the above expression

$$P\left(Y^{(p)\langle t\rangle} = 0 | X^{[0](p)\langle t\rangle}\right) = \begin{cases} \text{Maximize } \hat{Y}^{(p)\langle t\rangle} & \text{if } Y^{(p)\langle t\rangle} = 1 \\ \text{Maximize } 1 - \hat{Y}^{(p)\langle t\rangle} \text{ or Minimize } \hat{Y}^{(p)\langle t\rangle} & \text{if } Y^{(p)\langle t\rangle} = 0 \end{cases}$$

Rather than having separate equations for the positive and negative classes create a simplified version of the Binary Cross-Entropy Loss function that works for both classes. This simplified version is formulated as follows:

$$P\left(Y^{(p)\langle t\rangle} | X^{[0](p)\langle t\rangle}\right) = \left(\hat{Y}^{(p)\langle t\rangle}\right)^{Y^{(p)\langle t\rangle}} * \left(1 - \hat{Y}^{(p)\langle t\rangle}\right)^{1-Y^{(p)\langle t\rangle}} \qquad ---(F.8)$$

If $Y^{(p)\langle t\rangle} = 1$,

$$P\left(Y^{(p)\langle t\rangle} = 1 | X^{[0](p)\langle t\rangle}\right) = \left(\hat{Y}^{(p)\langle t\rangle}\right)^{1} * \left(1 - \hat{Y}^{(p)\langle t\rangle}\right)^{1-1} = \hat{Y}^{(p)\langle t\rangle}$$

If $Y^{(p)\langle t\rangle} = 0$,

$$P\left(Y^{(p)\langle t\rangle} = 0 | X^{[0](p)\langle t\rangle}\right) = \left(\hat{Y}^{(p)\langle t\rangle}\right)^{0} * \left(1 - \hat{Y}^{(p)\langle t\rangle}\right)^{1-0} = 1 - \hat{Y}^{[L](p)\langle t\rangle}$$

The expression in Eq. $(F.8)$ is called the Bernoulli trial. Calculate the Bernoulli trial for each time step and multiply all to get the final loss. But it will create a problem of getting 0 at every step. To avoid this problem, perform -log Operation, which will convert multiplication into summation and find the optimum value for loss.

Applying the log, for all time steps $t = 1,2,3,\cdots,T$:

$$\log\left(P\left(Y^{(p)\langle t\rangle} | X^{[0](p)\langle t\rangle}\right)\right) = E^{\langle t\rangle}\left(Y^{(p)\langle t\rangle}, \hat{Y}^{(p)\langle t\rangle}\right)$$

$$E^{\langle t\rangle}\left(Y^{(p)\langle t\rangle}, \hat{Y}^{(p)\langle t\rangle}\right) = -\frac{1}{T}\sum_{t=1}^{T}\left[Y^{(p)\langle t\rangle}\ln\left(\hat{Y}^{(p)\langle t\rangle}\right) + \right]\left(1 - Y^{(p)\langle t\rangle}\right)\ln\left(1 - \hat{Y}^{(p)\langle t\rangle}\right) \qquad (F.9)$$

$$\left(Y^{(p)\langle t\rangle}, \hat{Y}^{[L](p)\langle t\rangle}\right) = \begin{cases} \log\left(\hat{Y}^{(p)\langle t\rangle}\right) & \text{when } Y^{(p)\langle t\rangle} = 1 \\ \log\left(Y^{(p)\langle t\rangle}, \hat{Y}^{(p)\langle t\rangle}\right) & \text{when } Y^{(p)\langle t\rangle} = 0 \end{cases}$$

Based on the simplified Eq. $(F.9)$ of the Binary Cross-Entropy Loss function minimize the values of both $\log\left(\hat{Y}^{(p)\langle t\rangle}\right)$ and $\log\left(Y^{(p)\langle t\rangle}, \hat{Y}^{(p)\langle t\rangle}\right)$.

4.3. Computation of the Gradients of Loss Function w.r.t Parameters

The gradient of the Cross-Entropy function E_t for the predicted value $\left(\hat{Y}^{(p)\langle t\rangle}\right)$ is as follows:

$$\frac{\partial E^{\langle t\rangle}}{\partial \hat{Y}^{(p)\langle t\rangle}} = -\frac{1}{T}\sum_{t=1}^{T}\frac{\partial}{\partial \hat{Y}^{(p)\langle t\rangle}}\left[Y^{(p)\langle t\rangle}\ln\left(\hat{Y}^{(p)\langle t\rangle}\right)+\left(1-Y^{(p)\langle t\rangle}\right)\ln\left(1-\hat{Y}^{(p)\langle t\rangle}\right)\right]$$

$$\frac{\partial E^{\langle t\rangle}}{\partial \hat{Y}^{(p)\langle t\rangle}} = -\frac{1}{T}\sum_{t=1}^{T}\left[Y^{(p)\langle t\rangle}\frac{\partial}{\partial \hat{Y}^{(p)\langle t\rangle}}\left(\ln\hat{Y}^{(p)\langle t\rangle}\right)+\left(1-Y^{(p)\langle t\rangle}\right)\frac{\partial}{\partial \hat{Y}^{(p)\langle t\rangle}}\ln\left(1-\hat{Y}^{(p)\langle t\rangle}\right)\right]$$

$$\frac{\partial E^{\langle t\rangle}}{\partial \hat{Y}^{(p)\langle t\rangle}} = -\frac{1}{T}\sum_{t=1}^{T}\left[\frac{Y^{(p)\langle t\rangle}}{\hat{Y}^{(p)\langle t\rangle}}+\frac{\left(1-Y^{(p)\langle t\rangle}\right)}{\left(1-\hat{Y}^{(p)\langle t\rangle}\right)}(-1)\right]$$

$$\frac{\partial E^{\langle t\rangle}}{\partial \hat{Y}^{(p)\langle t\rangle}} = -\frac{1}{T}\sum_{t=1}^{T}\left[\frac{Y^{(p)\langle t\rangle}}{\hat{Y}^{[L](p)\langle t\rangle}}-\frac{\left(1-Y^{(p)\langle t\rangle}\right)}{\left(1-\hat{Y}^{(p)\langle t\rangle}\right)}\right]$$

$$\frac{\partial E^{\langle t\rangle}}{\partial \hat{Y}^{[L](p)\langle t\rangle}} = -\frac{1}{T}\sum_{t=1}^{T}\left[\frac{Y^{(p)\langle t\rangle}\left(1-\hat{Y}^{(p)\langle t\rangle}\right)-\hat{Y}^{(p)\langle t\rangle}\left(1-Y^{(p)\langle t\rangle}\right)}{\hat{Y}^{(p)\langle t\rangle}\left(1-\hat{Y}^{(p)\langle t\rangle}\right)}\right]$$

$$\frac{\partial E^{\langle t\rangle}}{\partial \hat{Y}^{(p)\langle t\rangle}} = -\frac{1}{T}\sum_{t=1}^{T}\left[\frac{Y^{(p)\langle t\rangle}-Y^{(p)\langle t\rangle}\hat{Y}^{(p)\langle t\rangle}-\hat{Y}^{(p)\langle t\rangle}+Y^{(p)\langle t\rangle}\hat{Y}^{(p)\langle t\rangle}}{\hat{Y}^{(p)\langle t\rangle}\left(1-\hat{Y}^{(p)\langle t\rangle}\right)}\right]$$

$$\frac{\partial E^{\langle t\rangle}}{\partial \hat{Y}^{(p)\langle t\rangle}} = -\frac{1}{T}\sum_{t=1}^{T}\left[\frac{Y^{(p)\langle t\rangle}-\hat{Y}^{(p)\langle t\rangle}}{\hat{Y}^{(p)\langle t\rangle}\left(1-\hat{Y}^{(p)\langle t\rangle}\right)}\right]$$

Start backward pass from the end of the sequence by computing gradients for each time step. For time step $T, H^{\langle T\rangle}$ has only $O^{\langle T\rangle}$ as a descendent; its gradient will be simple

$$\frac{\partial E^{\langle T\rangle}}{\partial H^{\langle T\rangle}} = (W_o)^{\mathrm{T}}\frac{\partial E^{\langle T\rangle}}{\partial O^{\langle T\rangle}}$$

$t = T-1$ down to $t = 1$ having $O^{\langle t\rangle}$ and $H^{\langle t+1\rangle}$ as descendent so that the gradient will be

$$\frac{\partial E^{\langle t\rangle}}{\partial H^{\langle t\rangle}} = \left(\frac{\partial H^{\langle t+1\rangle}}{\partial H^{\langle t\rangle}}\right)^{\mathrm{T}}\times\frac{\partial E^{\langle T\rangle}}{\partial H^{\langle t+1\rangle}}+\left(\frac{\partial O^{\langle t\rangle}}{\partial H^{\langle t\rangle}}\right)^{\mathrm{T}}\times\frac{\partial E^{\langle t\rangle}}{\partial O^{\langle t\rangle}}$$

$$\frac{\partial E^{\langle t\rangle}}{\partial H^{\langle t\rangle}} = (W_h)^{\mathrm{T}}diag\left(1-\left(H^{\langle t+1\rangle}\right)^2\right)\times\frac{\partial E^{\langle T\rangle}}{\partial H^{\langle t+1\rangle}}+(W_o)^{\mathrm{T}}\times\frac{\partial E^{\langle t\rangle}}{\partial O^{\langle t\rangle}}$$

The Gradient of the Cost Function for the Weight Matrix at the Output Layer *L*

Let's calculate the gradient of the loss function $E^{\langle t\rangle}$ concerning the weights W_o and biases $b^{[L]}$using partial differentiation. By using the chain rule since the loss function is unrelated to the weights. For the gradient of $E^{\langle t\rangle}$:

$$\frac{\partial E}{\partial W_o} = \frac{1}{T}\sum_{t=1}^{T}\frac{\partial E^{\langle t\rangle}}{\partial \hat{Y}^{\langle t\rangle}}\times\frac{\partial \hat{Y}^{\langle t\rangle}}{\partial O^{[L]\langle t\rangle}}\times\left(\frac{\partial O^{[L]\langle t\rangle}}{\partial W_o}\right)^{\mathbf{T}} \qquad ---(B.1)$$

Now, need to find the following three gradients,

$$\frac{\partial E^{\langle t\rangle}}{\partial \hat{Y}^{\langle t\rangle}} =? \quad \frac{\partial \hat{Y}^{\langle t\rangle}}{\partial O^{[L]\langle t\rangle}} =? \quad \frac{\partial O^{[L]\langle t\rangle}}{\partial W_o} =?$$

The gradient of the predicted values $\hat{Y}^{\langle t\rangle}$ concerning the $O^{\langle t\rangle}$ can be written as:

$$\frac{\partial \hat{Y}^{\langle t\rangle}}{\partial O^{[L]\langle t\rangle}} = \frac{\partial}{\partial O^{[L]\langle t\rangle}} \varphi_o\left(\hat{Y}^{\langle t\rangle}\right) = \varphi_o'\left(O^{[L]\langle t\rangle}\right) \qquad ---(B.1a)$$

The gradient of $O^{[L]\langle t\rangle}$ concerning the weight W_o is given by

$$\frac{\partial O^{[L]\langle t\rangle}}{\partial W_o} = \frac{\partial}{\partial W_o}\left(O^{[L]\langle t\rangle}\right) = \frac{\partial}{\partial W_o^{[L]}}\left[\left(W_o^{[L]}\right)\left(H^{[L-1]\langle t\rangle}\right) + b_o\right] \qquad ---(B.1b)$$

$$\frac{\partial O^{[L]\langle t\rangle}}{\partial W_o} = H^{[L-1]\langle t\rangle} \qquad ---(B.1c)$$

Therefore, using Eqs. $(B.1a - B.1b)$ in Eq. $(B.1)$, the gradient of $E^{\langle t\rangle}$ concerning the weights W_o can be expressed as

$$\frac{\partial E}{\partial W_o} = \frac{1}{T}\sum_{t=1}^{T} \frac{\partial E^{\langle t\rangle}}{\partial \hat{Y}^{\langle t\rangle}} \times \varphi_o'\left(O^{[L]\langle t\rangle}\right) \times \left(H^{[L-1]\langle t\rangle}\right)^{\mathbf{T}} \qquad ---(B.2)$$

The Gradient of the Cost Function for Bias Vectors

The gradient of $E^{\langle t\rangle}$ concerning the biases $b_o^{[L]}$ is as follows. Derivative of the loss function $E^{\langle t\rangle}$:

$$\frac{\partial E^{\langle t\rangle}}{\partial b_o} = \frac{\partial E_t}{\partial \hat{Y}^{\langle t\rangle}} \times \frac{\partial \hat{Y}^{\langle t\rangle}}{\partial O^{[L]\langle t\rangle}} \times \frac{\partial O^{[L]\langle t\rangle}}{\partial b_o} \qquad ---(B.3)$$

The gradient of $O^{[L]\langle t\rangle}$ concerning the weight $b_o^{[L]}$ is given by

$$\frac{\partial O^{[L]\langle t\rangle}}{\partial b_o} = \frac{\partial}{\partial b_o}\left(O^{[L]\langle t\rangle}\right) = \frac{\partial}{\partial b_o}\left[\left(W_o^{[L]}\right)\left(H^{[L-1]\langle t\rangle}\right) + b_o\right] = 1 \qquad ---(B.3a)$$

Using Eqs. $(B.1a)$ and $(B.3a)$ in Eq. $(B.3)$.

$$\frac{\partial E^{\langle t\rangle}}{\partial b_o} = \frac{1}{T}\sum_{t=1}^{T} \frac{\partial E^{\langle t\rangle}}{\partial \hat{Y}^{\langle t\rangle}} \times \varphi_o'\left(O^{[L]\langle t\rangle}\right) \times 1$$

$$\frac{\partial E^{\langle t\rangle}}{\partial b_o} = \frac{1}{T}\sum_{t=1}^{T} \frac{\partial E^{\langle t\rangle}}{\partial \hat{Y}^{\langle t\rangle}} \times \varphi_o'\left(O^{[L]\langle t\rangle}\right) \qquad ---(B.4)$$

Gradient of Bias Vector in Recurrent Layers for $t = T$

$$\frac{\partial E^{\langle t\rangle}}{\partial b^{[L-1]}} = \left(\frac{\partial H^{[L-1]\langle t\rangle}}{\partial Z^{[L-1]\langle t\rangle}} \times \frac{\partial Z^{[L-1]\langle t\rangle}}{\partial b^{[L-1]\langle t\rangle}}\right)^{\mathbf{T}} \times \frac{\partial E^{\langle t\rangle}}{\partial \hat{Y}^{\langle t\rangle}} \times \frac{\partial \hat{Y}^{\langle t\rangle}}{\partial O^{[L]\langle t\rangle}} \times \frac{\partial O^{[L]\langle t\rangle}}{\partial H^{[L-1]\langle t\rangle}} \qquad ---(B.5)$$

$$\frac{\partial \hat{Y}^{\langle t\rangle}}{\partial O^{[L]\langle t\rangle}} = \varphi_o'\left(O^{[L]\langle t\rangle}\right) \qquad ---(B.5a)$$

$$\frac{\partial O^{[L]\langle t\rangle}}{\partial H^{[L-1]\langle t\rangle}} = W_o$$

$$\frac{\partial H^{[L-1]\langle t\rangle}}{\partial Z^{[L-1]\langle t\rangle}} = \varphi_h'\left(Z^{[L-1]\langle t\rangle}\right) = \left(1 - \varphi_h\left(Z^{[L-1]\langle t\rangle}\right)^2\right) = \operatorname{diag}\left(1 - \left(H^{[L-1]\langle t\rangle}\right)^2\right) \quad (B.5b)$$

$$\frac{\partial Z^{[L-1]\langle t\rangle}}{\partial b^{[L-1]\langle t\rangle}} = 1$$

$$\frac{\partial E^{\langle t\rangle}}{\partial b^{[L-1]\langle t\rangle}} = \operatorname{diag}\left(1-\left(H^{[L-1]\langle t\rangle}\right)^2\right)\times\frac{\partial E^{\langle t\rangle}}{\partial \hat{Y}^{\langle t\rangle}}\times\varphi_o'\left(O^{[L]\langle t\rangle}\right)\times W_o \qquad ---(B.6)$$

Gradient of Bias Vector in Recurrent Layers for $\boldsymbol{t=1,2,\cdots,T-1}$

$$\frac{\partial E^{\langle t\rangle}}{\partial b^{[L-1]\langle t\rangle}} = \left(\frac{\partial H^{[L-1]\langle t\rangle}}{\partial Z^{[L-1]\langle t\rangle}}\times\frac{\partial Z^{[L-1]\langle t\rangle}}{\partial b^{[L-1]\langle t\rangle}}\right)^{\mathbf{T}}\times \left[\frac{\partial E^{\langle t\rangle}}{\partial \hat{Y}^{\langle t\rangle}}\times\frac{\partial \hat{Y}^{\langle t\rangle}}{\partial O^{[L]\langle t\rangle}}\times\frac{\partial O^{[L]\langle t\rangle}}{\partial H^{[L-1]\langle t\rangle}}+\frac{\partial E^{\langle t+1\rangle}}{\partial \hat{Y}^{\langle t+1\rangle}}\times\frac{\partial \hat{Y}^{\langle t+1\rangle}}{\partial O^{[L]\langle t+1\rangle}} \times\frac{\partial O^{[L]\langle t+1\rangle}}{\partial H^{[L-1]\langle t+1\rangle}}\times\frac{\partial H^{[L-1]\langle t+1\rangle}}{\partial Z^{[L-1]\langle t+1\rangle}}\times\frac{\partial Z^{[L-1]\langle t+1\rangle}}{\partial H^{[L-1]\langle t\rangle}}\right] \qquad ---(B.7)$$

$$\frac{\partial \hat{Y}^{\langle t\rangle}}{\partial O^{[L]\langle t\rangle}} = \varphi_o'\left(O^{[L]\langle t\rangle}\right)$$

$$\frac{\partial O^{[L]\langle t\rangle}}{\partial H^{[L-1]\langle t\rangle}} = W_o$$

$$\frac{\partial H^{[L-1]\langle t\rangle}}{\partial Z^{[L-1]\langle t\rangle}} = \varphi_h'\left(Z^{[L-1]\langle t\rangle}\right)=\left(1-\varphi_h\left(Z^{[L-1]\langle t\rangle}\right)^2\right)=\operatorname{diag}\left(1-\left(H^{[L-1]\langle t\rangle}\right)^2\right)$$

$$\frac{\partial Z^{[L-1]\langle t\rangle}}{\partial b^{[L-1]\langle t\rangle}} = 1$$

$$\frac{\partial E^{\langle t+1\rangle}}{\partial H^{\langle t\rangle}} = \operatorname{diag}\left(1-\left(H^{[L-1]\langle t+1\rangle}\right)^2\right)\times\frac{\partial E^{\langle t+1\rangle}}{\partial \hat{Y}^{\langle t+1\rangle}}\times\varphi_o'\left(O^{[L]\langle t+1\rangle}\right)\times W_o$$

$$\frac{\partial E^{\langle t\rangle}}{\partial b^{[L-1]\langle t\rangle}} = \operatorname{diag}\left(1-\left(H^{[L-1]\langle t\rangle}\right)^2\right)\times\left[\left(W_o^{[L]}\right)^{\mathbf{T}}\times\frac{\partial E^{\langle t\rangle}}{\partial \hat{Y}^{\langle t\rangle}}\times\varphi_o'\left(O^{[L]\langle t\rangle}\right)+ \left(W_h^{[L-1]}\right)^{\mathbf{T}}\times\frac{\partial E^{\langle t+1\rangle}}{\partial \hat{Y}^{\langle t+1\rangle}}\times\varphi_o'\left(O^{[L]\langle t+1\rangle}\right)\times W_o^{[L]}\times\operatorname{diag}\left(1-\left(H^{[L-1]\langle t+1\rangle}\right)^2\right)\right]$$

For all time steps $t=1,2,\cdots,T$

$$\frac{\partial E}{\partial b^{[L-1]\langle t\rangle}} = -\frac{1}{T}\sum_{t=1}^{T}\frac{\partial E^{\langle t\rangle}}{\partial b^{[L-1]\langle t\rangle}}$$

For $t=T$

$$\frac{\partial E^{\langle t\rangle}}{\partial b^{[L-1]\langle t\rangle}} = \operatorname{diag}\left(1-\left(H^{[L-1]\langle t\rangle}\right)^2\right)\times\frac{\partial E^{\langle t\rangle}}{\partial \hat{Y}^{\langle t\rangle}}\times\varphi_o'\left(O^{[L]\langle t\rangle}\right)\times W_o$$

For $t=T-1.\cdots,1$

$$\frac{\partial E^{\langle t\rangle}}{\partial b^{[L-1]\langle t\rangle}} = \operatorname{diag}\left(1-\left(H^{[L-1]\langle t\rangle}\right)^2\right)\times\left[\left(W_o\ \right)^{\mathbf{T}}\times\frac{\partial E^{\langle t\rangle}}{\partial \hat{Y}^{\langle t\rangle}}\times\varphi_o'\left(O^{[L]\langle t\rangle}\right)+ \left(W_h^{[L-1]}\right)^{\mathbf{T}}\times\frac{\partial E^{\langle t+1\rangle}}{\partial \hat{Y}^{\langle t+1\rangle}}\times\varphi_o'\left(O^{[L]\langle t+1\rangle}\right)\times W_o\ \times\operatorname{diag}\left(1-\left(H^{[L-1]\langle t+1\rangle}\right)^2\right)\right]$$

The Gradient of Cost Function w.r.t Weight Matrices in Recurrent Layers for $t = T$

$$\frac{\partial E^{\langle t\rangle}}{\partial W_h^{[L-1]\langle t\rangle}} = \frac{\partial E^{\langle t\rangle}}{\partial \hat{Y}^{\langle t\rangle}} \times \frac{\partial \hat{Y}^{\langle t\rangle}}{\partial O^{[L]\langle t\rangle}} \times \frac{\partial O^{[L]\langle t\rangle}}{\partial H^{[L-1]\langle t\rangle}} \times \frac{\partial H^{[L-1]\langle t\rangle}}{\partial Z^{[L-1]\langle t\rangle}} \times \left(\frac{\partial Z^{[L-1]\langle t\rangle}}{\partial W_h^{[L-1]}}\right)^{\mathbf{T}}$$

$$\frac{\partial \hat{Y}^{\langle t\rangle}}{\partial O^{[L]\langle t\rangle}} = \varphi_o'\left(O^{[L]\langle t\rangle}\right)$$

$$\frac{\partial O^{[L]\langle t\rangle}}{\partial H^{[L-1]\langle t\rangle}} = W_o$$

$$\frac{\partial H^{[L-1]\langle t\rangle}}{\partial Z^{[L-1]\langle t\rangle}} = \varphi_h'\left(Z^{[L-1]\langle t\rangle}\right) = \left(1 - \varphi_h\left(Z^{[L-1]\langle t\rangle}\right)^2\right) = \mathrm{diag}\left(1 - \left(H^{[L-1]\langle t\rangle}\right)^2\right)$$

$$\frac{\partial Z^{[L-1]\langle t\rangle}}{\partial W_h^{[L-1]}} = H^{[L-1]\langle t-1\rangle}$$

$$\frac{\partial E^{\langle t\rangle}}{\partial W_h^{[L-1]\langle t\rangle}} = \mathrm{diag}\left(1 - \left(H^{[L-1]\langle t\rangle}\right)^2\right) \times \frac{\partial E^{\langle t\rangle}}{\partial H^{\langle t\rangle}} \times \left(H^{[L-1]\langle t-1\rangle}\right)^{\mathrm{T}} \qquad ---(B.8)$$

Gradient of Cost Function w.r.t Weight in Recurrent Layers for $t = 1, 2, \cdots, T-1$

$$\frac{\partial E^{\langle t\rangle}}{\partial W_h^{[L-1]\langle t\rangle}} = \left(\frac{\partial H^{[L-1]\langle t\rangle}}{\partial Z^{[L-1]\langle t\rangle}} \times \frac{\partial Z^{[L-1]\langle t\rangle}}{\partial W_h^{[L-1]}}\right)^{\mathbf{T}} \times \left[\frac{\partial E^{\langle t\rangle}}{\partial \hat{Y}^{\langle t\rangle}} \times \frac{\partial \hat{Y}^{\langle t\rangle}}{\partial O^{[L]\langle t\rangle}} \times \frac{\partial O^{[L]\langle t\rangle}}{\partial H^{[L-1]\langle t\rangle}} + \frac{\partial E^{\langle t+1\rangle}}{\partial \hat{Y}^{\langle t+1\rangle}} \times \frac{\partial \hat{Y}^{\langle t+1\rangle}}{\partial O^{[L]\langle t+1\rangle}} \times \frac{\partial O^{[L]\langle t+1\rangle}}{\partial H^{[L-1]\langle t+1\rangle}} \times \frac{\partial H^{[L-1]\langle t+1\rangle}}{\partial Z^{[L-1]\langle t+1\rangle}} \times \frac{\partial Z^{[L-1]\langle t+1\rangle}}{\partial H^{[L-1]\langle t\rangle}}\right] \qquad ---(B.9)$$

$$\frac{\partial \hat{Y}^{\langle t\rangle}}{\partial O^{[L]\langle t\rangle}} = \varphi_o'\left(O^{[L]\langle t\rangle}\right)$$

$$\frac{\partial O^{[L]\langle t\rangle}}{\partial H^{[L-1]\langle t\rangle}} = W_o$$

$$\frac{\partial H^{[L-1]\langle t\rangle}}{\partial Z^{[L-1]\langle t\rangle}} = \varphi_h'\left(Z^{[L-1]\langle t\rangle}\right) = \left(1 - \varphi_h\left(Z^{[L-1]\langle t\rangle}\right)^2\right) = \mathrm{diag}\left(1 - \left(H^{[L-1]\langle t\rangle}\right)^2\right)$$

$$\frac{\partial Z^{[L-1]\langle t\rangle}}{\partial W_h^{[L-1]\langle t\rangle}} = H^{[L-1]\langle t-1\rangle}$$

$$\frac{\partial E^{\langle t+1\rangle}}{\partial H^{\langle t\rangle}} = \mathrm{diag}\left(1 - \left(H^{[L-1]\langle t+1\rangle}\right)^2\right) \times \frac{\partial E^{\langle t+1\rangle}}{\partial \hat{Y}^{\langle t+1\rangle}} \times \varphi_o'\left(O^{[L]\langle t+1\rangle}\right) \times W_o$$

$$\frac{\partial E^{\langle t\rangle}}{\partial W_h^{[L-1]\langle t\rangle}} = \mathrm{diag}\left(1 - \left(H^{[L-1]\langle t\rangle}\right)^2\right) \times \frac{\partial E^{\langle t\rangle}}{\partial H^{\langle t\rangle}} \times \left(H^{[L-1]\langle t-1\rangle}\right)^{\mathbf{T}}$$

or

$$\frac{\partial E^{\langle t\rangle}}{\partial W_h^{[L-1]\langle t\rangle}} = \operatorname{diag}\left(1-\left(H^{[L-1]\langle t\rangle}\right)^2\right)\times\left[\left(W_o\ \right)^{\mathrm{T}}\times\frac{\partial E^{\langle t\rangle}}{\partial\hat{Y}^{\langle t\rangle}}\times\varphi_o'\left(O^{[L]\langle t\rangle}\right)+\right.$$
$$\left(W_h^{[L-1]}\right)^{\mathrm{T}}\times\frac{\partial E^{\langle t+1\rangle}}{\partial\hat{Y}^{\langle t+1\rangle}}\times\varphi_o'\left(O^{[L]\langle t+1\rangle}\right)\times$$
$$\left.\times W_o\ \ \times\operatorname{diag}\left(1-\left(H^{[L-1]\langle t+1\rangle}\right)^2\right)\right]\left(H^{[L-1]\langle t-1\rangle}\right)^{\mathrm{T}} \qquad ---(B.10)$$

For all time steps $t=1,2,\cdots,T$

$$\frac{\partial E}{\partial W_h^{[L-1]}} = -\frac{1}{T}\sum_{t=1}^{T}\frac{\partial E^{\langle t\rangle}}{\partial W_h^{[L-1]\langle t\rangle}}$$

For $t=T$

$$\frac{\partial E^{\langle t\rangle}}{\partial W_h^{[L-1]\langle t\rangle}} = \operatorname{diag}\left(1-\left(H^{[L-1]\langle t\rangle}\right)^2\right)\times\frac{\partial E^{\langle t\rangle}}{\partial H^{\langle t\rangle}}\times\left(H^{[L-1]\langle t-1\rangle}\right)^{\mathrm{T}}$$

For $t=T-1,\cdots,1$

$$\frac{\partial E^{\langle t\rangle}}{\partial W_h^{[L-1]\langle t\rangle}} = \operatorname{diag}\left(1-\left(H^{[L-1]\langle t\rangle}\right)^2\right)\times\left[\left(W_o\ \right)^{\mathrm{T}}\times\frac{\partial E^{\langle t\rangle}}{\partial\hat{Y}^{\langle t\rangle}}\times\varphi_o'\left(O^{[L]\langle t\rangle}\right)+\right.$$
$$\left(W_h^{[L-1]}\right)^{\mathrm{T}}\times\frac{\partial E^{\langle t+1\rangle}}{\partial\hat{Y}^{\langle t+1\rangle}}\times\varphi_o'\left(O^{[L]\langle t+1\rangle}\right)\times W_o^{[L]}$$
$$\left.\times\operatorname{diag}\left(1-\left(H^{[L-1]\langle t+1\rangle}\right)^2\right)\right]\times\left(H^{[L-1]\langle t-1\rangle}\right)^{\mathrm{T}} \qquad ---(B.11)$$

The Gradient of Cost Function w.r.t Weight Matrices between Input and Hidden Layer for $t=T$

$$\frac{\partial E^{\langle t\rangle}}{\partial W_x^{\langle t\rangle}} = \frac{\partial E^{\langle t\rangle}}{\partial\hat{Y}^{\langle t\rangle}}\times\frac{\partial\hat{Y}^{\langle t\rangle}}{\partial O^{[L]\langle t\rangle}}\times\frac{\partial O^{[L]\langle t\rangle}}{\partial H^{[L-1]\langle t\rangle}}\times\left(\frac{\partial H^{[L-1]\langle t\rangle}}{\partial Z^{[L-1]\langle t\rangle}}\times\frac{\partial Z^{[L-1]\langle t\rangle}}{\partial W_x}\right)^{\mathbf{T}} \qquad ---(B.12)$$

$$\frac{\partial\hat{Y}^{\langle t\rangle}}{\partial O^{[L]\langle t\rangle}} = \varphi_o'\left(O^{[L]\langle t\rangle}\right)$$

$$\frac{\partial O^{[L]\langle t\rangle}}{\partial H^{[L-1]\langle t\rangle}} = W_o$$

$$\frac{\partial H^{[L-1]\langle t\rangle}}{\partial Z^{[L-1]\langle t\rangle}} = \varphi_h'\left(Z^{[L-1]\langle t\rangle}\right)=\left(1-\varphi_h\left(Z^{[L-1]\langle t\rangle}\right)^2\right)=\operatorname{diag}\left(1-\left(H^{[L-1]\langle t\rangle}\right)^2\right)$$

$$\frac{\partial Z^{[L-1]\langle t\rangle}}{\partial W_x} = X^{[0]\langle t\rangle}$$

$$\frac{\partial E^{\langle t\rangle}}{\partial W_x^{\langle t\rangle}} = \operatorname{diag}\left(1-\left(H^{[L-1]\langle t\rangle}\right)^2\right)\times\frac{\partial E^{\langle t\rangle}}{\partial H^{\langle t\rangle}}\times\left(X^{[0]\langle t\rangle}\right)^{\mathrm{T}}$$

Gradient of Cost Function w.r.t Weights between Input to Hidden Layer for $t=1,2,\cdots,T-1$

$$\frac{\partial E^{\langle t\rangle}}{\partial W_x^{\langle t\rangle}} = \left(\frac{\partial H^{[L-1]\langle t\rangle}}{\partial Z^{[L-1]\langle t\rangle}} \times \frac{\partial Z^{[L-1]\langle t\rangle}}{\partial W_x^{[0]\langle t\rangle}}\right)^{\mathbf{T}} \times \left[\frac{\partial E^{\langle t\rangle}}{\partial \hat{Y}^{\langle t\rangle}} \times \frac{\partial \hat{Y}^{\langle t\rangle}}{\partial O^{[L]\langle t\rangle}} \times \frac{\partial O^{[L]\langle t\rangle}}{\partial H^{[L-1]\langle t\rangle}} + \frac{\partial E^{\langle t+1\rangle}}{\partial \hat{Y}^{\langle t+1\rangle}} \times \frac{\partial \hat{Y}^{\langle t+1\rangle}}{\partial O^{[L]\langle t+1\rangle}} \times \frac{\partial O^{[L]\langle t+1\rangle}}{\partial H^{[L-1]\langle t+1\rangle}} \times \frac{\partial H^{[L-1]\langle t+1\rangle}}{\partial Z^{[L-1]\langle t+1\rangle}} \times \frac{\partial Z^{[L-1]\langle t+1\rangle}}{\partial H^{[L-1]\langle t\rangle}}\right] \quad ---(B.13)$$

$$\frac{\partial \hat{Y}^{\langle t\rangle}}{\partial O^{[L]\langle t\rangle}} = \varphi_o'\left(O^{[L]\langle t\rangle}\right)$$

$$\frac{\partial O^{[L]\langle t\rangle}}{\partial H^{[L-1]\langle t\rangle}} = W_o$$

$$\frac{\partial H^{[L-1]\langle t\rangle}}{\partial Z^{[L-1]\langle t\rangle}} = \varphi_h'\left(Z^{[L-1]\langle t\rangle}\right) = \left(1 - \varphi_h\left(Z^{[L-1]\langle t\rangle}\right)^2\right) = \operatorname{diag}\left(1 - \left(H^{[L-1]\langle t\rangle}\right)^2\right)$$

$$\frac{\partial Z^{[L-1]\langle t\rangle}}{\partial W_x^{[0]\langle t\rangle}} = X^{[0]\langle t\rangle}$$

$$\operatorname{diag}\left(1 - \left(H^{[L-1]\langle t+1\rangle}\right)^2\right) \times \frac{\partial E^{\langle t+1\rangle}}{\partial \hat{Y}^{\langle t+1\rangle}} \times \varphi_o'\left(O^{[L]\langle t+1\rangle}\right) \times W_o^{[L]}$$

$$\frac{\partial E^{\langle t\rangle}}{\partial W_x^{\langle t\rangle}} = \operatorname{diag}\left(1 - \left(H^{[L-1]\langle t\rangle}\right)^2\right) \times \frac{\partial E^{\langle t\rangle}}{\partial H^{\langle t\rangle}} \times \left(X^{[0]\langle t\rangle}\right)^{\mathrm{T}}$$

or

$$\frac{\partial E^{\langle t\rangle}}{\partial W_x^{\langle t\rangle}} = \operatorname{diag}\left(1 - \left(H^{[L-1]\langle t\rangle}\right)^2\right) \times \left[\left(W_o^{[L]}\right)^{\mathbf{T}} \times \frac{\partial E^{\langle t\rangle}}{\partial \hat{Y}^{\langle t\rangle}} \times \varphi_o'\left(O^{[L]\langle t\rangle}\right) + \left(W_h^{[L-1]}\right)^{\mathbf{T}} \times \frac{\partial E^{\langle t+1\rangle}}{\partial \hat{Y}^{\langle t+1\rangle}} \times \varphi_o'\left(O^{[L]\langle t+1\rangle}\right) \times W_o^{[L]} \times \operatorname{diag}\left(1 - \left(H^{[L-1]\langle t+1\rangle}\right)^2\right)\right] \times \left(X^{[0]\langle t\rangle}\right)^{\mathbf{T}}$$

For all time steps $t = 1,2,\cdots,T$

$$\frac{\partial E}{\partial W_x} = -\frac{1}{T}\sum_{t=1}^{T}\frac{\partial E^{\langle t\rangle}}{\partial W_x^{\langle t\rangle}}$$

For $t = T$

$$\frac{\partial E^{\langle t\rangle}}{\partial W_x^{\langle t\rangle}} = \operatorname{diag}\left(1 - \left(H^{[L-1]\langle t\rangle}\right)^2\right) \times \frac{\partial E^{\langle t\rangle}}{\partial \hat{Y}^{\langle t\rangle}} \times \varphi_o'\left(O^{[L]\langle t\rangle}\right) \times W_o$$

For $t = T-1,\cdots,1$

$$\frac{\partial E^{\langle t\rangle}}{\partial W_x} = \operatorname{diag}\left(1 - \left(H^{[L-1]\langle t\rangle}\right)^2\right) \times \left[\left(W_o\ \right)^{\mathbf{T}} \times \frac{\partial E^{\langle t\rangle}}{\partial \hat{Y}^{\langle t\rangle}} \times \varphi_o'\left(O^{[L]\langle t\rangle}\right) + \left(W_h^{[L-1]}\right)^{\mathbf{T}} \times \frac{\partial E^{\langle t+1\rangle}}{\partial \hat{Y}^{\langle t+1\rangle}} \times \varphi_o'\left(O^{[L]\langle t+1\rangle}\right) \times W_o\ \times \operatorname{diag}\left(1 - \left(H^{[L-1]\langle t+1\rangle}\right)^2\right)\right] \times \left(X^{[0]\langle t\rangle}\right)^{\mathbf{T}}$$

5. The Obstacles of Training RNNs

Sequence Models (RNN) can be challenging to train due to the deep structure of the network when considering the temporal dimension of the input sequence. This depth of temporal layering is input dependent which can lead to difficulties in training. Furthermore, RNNs use shared parameters for different temporal layers which can exacerbate the instability caused by the varying sensitivity of the loss function (how much the loss changes in response to changes in the network's parameters). Combining a shared set of parameters and varying sensitivity can lead to unstable effects during training such as exploding or vanishing gradients. Vanishing gradients in RNN can occur when the gradients are multiplied by the same weight matrix at each time step.

Consider an RNN architecture with L consecutive layers that employ the tanh activation function between each pair of layers. $\left(W_h^{[l]}\right)^T$ denotes the shared weight between a pair of hidden nodes.

The derivative of the activation function in hidden layer l can be seen by $\varphi_h'\left(H^{[l]\langle t\rangle}\right)$, and the hidden values in the recurrent layers are represented by $H^{[l]\langle 1\rangle},\ H^{[l]\langle 2\rangle}, \cdots,\ H^{[l]\langle T\rangle}$. Additionally, the following is a representation of the gradient of the entropy loss function about the hidden activation $H^{[l]\langle t\rangle}$ (Aggarwal, 2018):

$$\frac{\partial E}{\partial H^{[l-1]\langle t\rangle}} = \frac{\partial E}{\partial \hat{Y}^{[l]\langle t+1\rangle}} \times \frac{\partial \hat{Y}^{[l]\langle t+1\rangle}}{\partial H^{[l-1]\langle t+1\rangle}} \times \frac{\partial H^{[l-1]\langle t+1\rangle}}{\partial Z^{[l-1]\langle t+1\rangle}} \times \frac{\partial Z^{[l-1]\langle t+1\rangle}}{\partial H^{[l-1]\langle t\rangle}}$$

$$\frac{\partial E}{\partial H^{[l-1]\langle t\rangle}} = \frac{\partial E}{\partial \hat{Y}^{[l]\langle t+1\rangle}} \times \frac{\partial \hat{Y}^{[l]\langle t+1\rangle}}{\partial H^{[l-1]\langle t+1\rangle}} \times \varphi_h'\left(Z^{[l-1]\langle t+1\rangle}\right) \times \left(W_h^{[l]}\right)^{t+1}$$

Since the shared weights in each temporal layer are the same, the gradient will multiply with the same quantity $\left(W_h^{[l]}\right)^{t+1} = W_h$. When the largest eigenvalue of W_h is greater than 1, $(W_h > 1)$ the gradients $\frac{\partial E}{\partial H^{[l-1]\langle t\rangle}}$ can explode as they are multiplied by W_h during backpropagation. This is because the recurrent connections amplify the gradients at each time step, leading to exponentially large gradients. On the other hand, if the magnitude of the largest eigenvalue of W_h is less than 1, $(W_h < 1)$ then the gradients $\frac{\partial E}{\partial H^{[l-1]\langle t\rangle}}$ can become vanishingly small as they are multiplied by W_h during backpropagation. It is because the recurrent connections reduced the gradients at each time step, leading to exponentially small gradients. The choice of activation function used in a neural network architecture can significantly impact the network's ability to propagate gradients through the layers during backpropagation. When the tanh activation function is used, for which the derivative $\varphi_h'(.)$ is almost always less than 1, the likelihood of encountering the vanishing gradient problem increases.

One way to alleviate the vanishing gradient problem using the tanh activation function is to initialize the network weights carefully. It can be done using initialization techniques such as Xavier initialization or He initialization, which ensure that the weights are scaled appropriately to avoid the gradients becoming too small.

Also, one can use gradient clipping (setting a threshold value), which works by limiting the size of the gradients and using optimized activation functions such as ReLU and Leaky ReLU to help ensure that the gradients do not vanish or explode.

6. RNN Hands Work Out Example to Predict the Next Word

To demonstrate how an RNN works considered a simple example consisting of a single sequence composed of six words, as shown in Fig. 6.1.

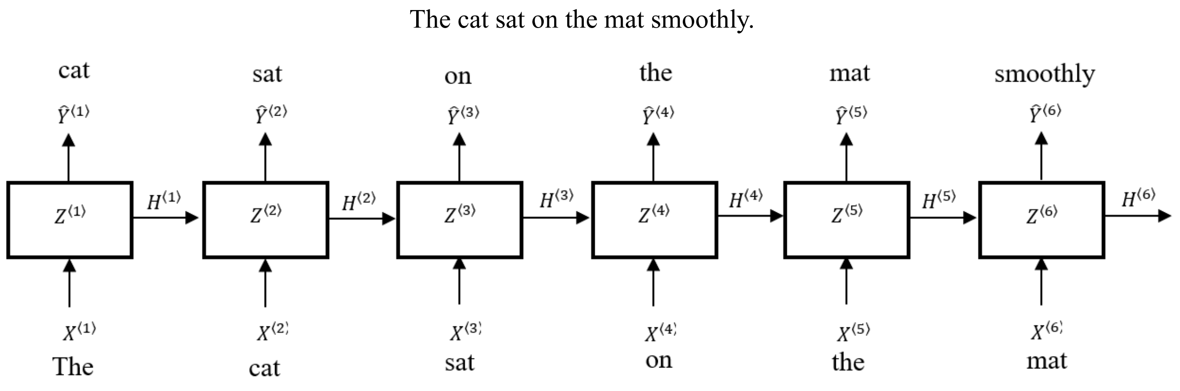


Fig. 6.1: Language Model Example Using RNNs

In this scenario, vocabulary consists of just six words, [*"cat", "sat", "on", "the", "mat", "smoothly"*]. At each time step from 1 to 6, RNNs processes the one-hot encoded input vector $X^{\langle t\rangle}$, mapping it to a hidden vector $H^{\langle t\rangle}$ of size 2 using the matrix W_x with size 2×6. The RNN then combines this hidden vector with the previous hidden state $H^{\langle t-1\rangle}$ represented by the matrix W_h and applies the tanh activation function to produce the new hidden state $H^{\langle t\rangle}$. Finally, the RNN computes the output $\hat{Y}^{\langle t\rangle}$ by multiplying the hidden state $H^{\langle t\rangle}$ with the matrix W_o. The matrices W_h and W_o sizes are 2×2 and 6×2, respectively.

Let's begin the example by converting the input sequence into a one-hot encoded vectors for $t = 1,2,3,4,5,6$ are $X^{\langle 1\rangle}, X^{\langle 2\rangle}, X^{\langle 3\rangle}, X^{\langle 4\rangle}, X^{\langle 5\rangle}, X^{\langle 6\rangle}$:

$$X^{\langle 1\rangle} = \begin{bmatrix}1\\0\\0\\0\\0\\0\end{bmatrix}, X^{\langle 2\rangle} = \begin{bmatrix}0\\1\\0\\0\\0\\0\end{bmatrix}, X^{\langle 3\rangle} = \begin{bmatrix}0\\0\\1\\0\\0\\0\end{bmatrix}, X^{\langle 4\rangle} = \begin{bmatrix}0\\0\\0\\1\\0\\0\end{bmatrix}, X^{\langle 5\rangle} = \begin{bmatrix}0\\0\\0\\0\\1\\0\end{bmatrix}, X^{\langle 6\rangle} = \begin{bmatrix}0\\0\\0\\0\\0\\1\end{bmatrix}$$

First sub list $[1, 0, 0, 0,0,0]$ represents "The", the first word in the vocabulary. The third sublist $[0, 0, 1, 0,0,0]$ represents "sat", the third word in the vocabulary. This one-hot encoded representation is input to a neural network for NLP tasks. However, more efficient ways exist to represent text data, especially for larger vocabularies, as it results in a very high-dimensional input. Other approaches, such as word embeddings or bag-of-words representations, are commonly used instead.

In the next step, define the model with three hidden units/memory cells and a vector dimension 2. Initializing weight matrices and bias vectors $W_x, W_h, W_o, b^{[l]}, b^{[L]}$. Weight matrix between input and hidden stat W_x:

$$\begin{bmatrix}1.624 & -0.612 & -0.528 & -1.073 & 0.865 & -2.302\\ 1.1745 & -0.761 & 0.319 & -0.249 & 1.642 & -2.06\end{bmatrix}$$

Weight matrix between hidden-to-hidden state W_h:

$$\begin{bmatrix} -0.322 & -0.384 \\ 1.134 & -1.1 \end{bmatrix}$$

Weight matrix between output to hidden state W_o:

$$\begin{bmatrix} -0.172 & -0.878 \\ 0.042 & 0.583 \\ -1.101 & 1.145 \\ 0.902 & 0.502 \\ 0.901 & -0.684 \\ -0.123 & -0.936 \end{bmatrix}$$

Bias Vector for hidden state b_h:

$$\begin{bmatrix} -0.268 \\ 0.53 \end{bmatrix}$$

Bias Vector for output state b_o:

$$\begin{bmatrix} -0.692 \\ -0.397 \\ -0.687 \\ -0.845 \\ -0.671 \\ -0.013 \end{bmatrix}$$

Using Eq. $(F.4)$, For $l = 1$, $t = 1$

$$Z^{[1]\langle 1\rangle} = W_x^{[0]} H^{[0]\langle 1\rangle} + W_h^{[1]} H^{[1]\langle 0\rangle} + b^{[l]}$$

$$Z^{[1]\langle 1\rangle} = \begin{bmatrix} 1.624 & -0.612 & -0.528 & -1.073 & 0.865 & -2.302 \\ 1.1745 & -0.761 & 0.319 & -0.249 & 1.642 & -2.06 \end{bmatrix} \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix} + \begin{bmatrix} -0.322 & -0.384 \\ 1.134 & -1.1 \end{bmatrix} \begin{bmatrix} 0 \\ 0 \end{bmatrix} + \begin{bmatrix} -0.268 \\ 0.53 \end{bmatrix}$$

$$Z^{[1]\langle 1\rangle} = \begin{bmatrix} 1.356 \\ 2.275 \end{bmatrix}$$

$$H^{[1]\langle 1\rangle} = \varphi_h\left(Z^{[1]\langle 1\rangle}\right)$$

$$H^{[1]\langle 1\rangle} = \begin{bmatrix} 0.875 \\ 0.979 \end{bmatrix}$$

$$O^{[2]\langle 1\rangle} = \left(W_o^{[2]}\right)^{\mathbf{T}} H^{[1]\langle 1\rangle} + b_o$$

$$O^{[2]\langle 1\rangle} = \begin{bmatrix} -0.172 & -0.878 \\ 0.042 & 0.583 \\ -1.101 & 1.145 \\ 0.902 & 0.502 \\ 0.901 & -0.684 \\ -0.123 & -0.936 \end{bmatrix}^{\mathbf{T}} \begin{bmatrix} 0.875 \\ 0.979 \end{bmatrix} + \begin{bmatrix} -0.692 \\ -0.397 \\ -0.687 \\ -0.845 \\ -0.671 \\ -0.013 \end{bmatrix}$$

$$O^{[2]\langle 1\rangle} = \begin{bmatrix} -1.702 \\ 0.211 \\ -0.529 \\ 0.436 \\ -0.552 \\ -1.037 \end{bmatrix}$$

The output Y at time $t = 1$ is computed as:

$$\hat{Y}^{[2]\langle 1\rangle} = \varphi_o\left(O^{[2]\langle 1\rangle}\right)$$

$$\hat{Y}^{[2]\langle 1\rangle} = \begin{bmatrix} 0.041 \\ 0.275 \\ 0.131 \\ 0.345 \\ 0.128 \\ 0.079 \end{bmatrix}$$

For $t = 2$, from Eq. $(F.4)$

$$Z^{[1]\langle 2\rangle} = W_x^{[0]} H^{[0]\langle 2\rangle} + W_z^{[1]} H^{[1]\langle 1\rangle} + b^{[l]}$$

$$Z^{[1]\langle 2\rangle} = \begin{bmatrix} 1.624 & -0.612 & -0.528 & -1.073 & 0.865 & -2.302 \\ 1.1745 & -0.761 & 0.319 & -0.249 & 1.642 & -2.06 \end{bmatrix} \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}$$

$$+ \begin{bmatrix} -0.322 & -0.384 \\ 1.134 & -1.1 \end{bmatrix} \begin{bmatrix} 0.875 \\ 0.979 \end{bmatrix} + \begin{bmatrix} -0.268 \\ 0.53 \end{bmatrix}$$

$$Z^{[1]\langle 2\rangle} = \begin{bmatrix} -0.88 \\ -0.231 \end{bmatrix}$$

$$H^{[1]\langle 2\rangle} = \varphi_h\left(Z^{[1]\langle 2\rangle}\right)$$

$$H^{[1]\langle 2\rangle} = \begin{bmatrix} -0.706 \\ -0.227 \end{bmatrix}$$

$$O^{[2]\langle 2\rangle} = \left(W_o^{[2]}\right)^{\mathbf{T}} H^{[1]\langle 2\rangle} + b^{[L]}$$

$$O^{[2]\langle 2\rangle} = \begin{bmatrix} -0.172 & -0.878 \\ 0.042 & 0.583 \\ -1.101 & 1.145 \\ 0.902 & 0.502 \\ 0.901 & -0.684 \\ -0.123 & -0.936 \end{bmatrix}^{\mathbf{T}} \begin{bmatrix} -0.706 \\ -0.227 \end{bmatrix} + \begin{bmatrix} -0.692 \\ -0.397 \\ -0.687 \\ -0.845 \\ -0.671 \\ -0.013 \end{bmatrix}$$

$$O^{[2]\langle 2\rangle} = \begin{bmatrix} -0.371 \\ -0.559 \\ -0.17 \\ -1.596 \\ -1.152 \\ 0.286 \end{bmatrix}$$

The output H at time $t = 2$ is computed as:

$$\hat{Y}^{[1]\langle 2\rangle} = \varphi_o\left(O^{[2]\langle 2\rangle}\right)$$

$$\hat{Y}^{[1]\langle 2\rangle} = \begin{bmatrix} 0.174 \\ 0.145 \\ 0.213 \\ 0.051 \\ 0.08 \\ 0.337 \end{bmatrix}$$

For $t = 3$, from Eq. $(F.4)$

$$Z^{[1]\langle 3\rangle} = W_x^{[0]} X^{[0]\langle 1\rangle} + W_z^{[1]} H^{[1]\langle 2\rangle} + b^{[l]}$$

$$Z^{[1]\langle 3\rangle} = \begin{bmatrix} 1.624 & -0.612 & -0.528 & -1.073 & 0.865 & -2.302 \\ 1.1745 & -0.761 & 0.319 & -0.249 & 1.642 & -2.06 \end{bmatrix} \begin{bmatrix} 0 \\ 0 \\ 1 \\ 0 \\ 0 \\ 0 \end{bmatrix}$$

$$+ \begin{bmatrix} -0.322 & -0.384 \\ 1.134 & -1.1 \end{bmatrix} \begin{bmatrix} -0.706 \\ -0.227 \end{bmatrix} + \begin{bmatrix} -0.268 \\ 0.53 \end{bmatrix}$$

$$Z^{[1]\langle 3\rangle} = \begin{bmatrix} -0.796 \\ 0.849 \end{bmatrix}$$

$$H^{[1]\langle 3\rangle} = \varphi_h\left(Z^{[1]\langle 3\rangle}\right)$$

$$H^{[1]\langle 3\rangle} = \begin{bmatrix} -0.662 \\ 0.691 \end{bmatrix}$$

$$O^{[2]\langle 3\rangle} = \left(W_o^{[2]}\right)^{\mathbf{T}} H^{[1]\langle 3\rangle} + b^{[L]}$$

$$O^{[2]\langle 3\rangle} = \begin{bmatrix} -0.172 & -0.878 \\ 0.042 & 0.583 \\ -1.101 & 1.145 \\ 0.902 & 0.502 \\ 0.901 & -0.684 \\ -0.123 & -0.936 \end{bmatrix}^{\mathbf{T}} \begin{bmatrix} -0.662 \\ 0.691 \end{bmatrix} + \begin{bmatrix} -0.692 \\ -0.397 \\ -0.687 \\ -0.845 \\ -0.671 \\ -0.013 \end{bmatrix}$$

$$O^{[2]\langle 3\rangle} = \begin{bmatrix} -1.185 \\ -0.022 \\ 0.833 \\ -1.095 \\ -1.74 \\ -0.578 \end{bmatrix}$$

The output Y at time $t = 3$ is computed as:

$$\hat{Y}^{[2]\langle 3\rangle} = \varphi_o\left(O^{[2]\langle 3\rangle}\right)$$

$$\hat{Y}^{[2]\langle 3\rangle} = \begin{bmatrix} 0.066 \\ 0.21 \\ 0.494 \\ 0.072 \\ 0.038 \\ 0.121 \end{bmatrix}$$

For $t = 4$, from Eq. $(F.4)$

$$Z^{[1]\langle 4\rangle} = W_x^{[0]} Z^{[1]\langle 4\rangle} + W_z^{[1]} Z^{[1]\langle 3\rangle} + b^{[l]}$$

$$Z^{[1]\langle 4\rangle} = \begin{bmatrix} 1.624 & -0.612 & -0.528 & -1.073 & 0.865 & -2.302 \\ 1.1745 & -0.761 & 0.319 & -0.249 & 1.642 & -2.06 \end{bmatrix} \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \\ 0 \\ 0 \end{bmatrix}$$

$$+ \begin{bmatrix} -0.322 & -0.384 \\ 1.134 & -1.1 \end{bmatrix} \begin{bmatrix} -0.662 \\ 0.691 \end{bmatrix} + \begin{bmatrix} -0.268 \\ 0.53 \end{bmatrix}$$

$$Z^{[1]\langle 4\rangle} = \begin{bmatrix} -1.341 \\ 0.281 \end{bmatrix}$$

$$H^{[1]\langle 4\rangle} = \varphi_h\left(Z^{[1]\langle 4\rangle}\right)$$

$$H^{[1]\langle 4\rangle} = \begin{bmatrix} -0.872 \\ 0.274 \end{bmatrix}$$

$$O^{[2]\langle 4\rangle} = \left(W_o^{[2]}\right)^{\mathbf{T}} H^{[1]\langle 4\rangle} + b^{[L]}$$

$$O^{[2]\langle 4\rangle} = \begin{bmatrix} -0.172 & -0.878 \\ 0.042 & 0.583 \\ -1.101 & 1.145 \\ 0.902 & 0.502 \\ 0.901 & -0.684 \\ -0.123 & -0.936 \end{bmatrix}^{\mathbf{T}} \begin{bmatrix} -0.872 \\ 0.274 \end{bmatrix} + \begin{bmatrix} -0.692 \\ -0.397 \\ -0.687 \\ -0.845 \\ -0.671 \\ -0.013 \end{bmatrix}$$

$$O^{[2]\langle 4\rangle} = \begin{bmatrix} -0.783 \\ -0.274 \\ 0.587 \\ -1.494 \\ -1.644 \\ -0.612 \end{bmatrix}$$

The output H at time $t = 4$ is computed as:

$$\hat{Y}^{[2]\langle 4\rangle} = \varphi_o\left(O^{[2]\langle 4\rangle}\right)$$

$$\hat{Y}^{[2]\langle 4\rangle} = \begin{bmatrix} 0.107 \\ 0.177 \\ 0.42 \\ 0.052 \\ 0.045 \\ 0.199 \end{bmatrix}$$

For $t = 5$, from Eq. $(F.4)$

$$Z^{[1]\langle 5\rangle} = W_x^{[0]} X^{[0]\langle 5\rangle} + W_z^{[1]} H^{[1]\langle 4\rangle} + b^{[l]}$$

$$Z^{[1]\langle 5\rangle} = \begin{bmatrix} 1.624 & -0.612 & -0.528 & -1.073 & 0.865 & -2.302 \\ 1.1745 & -0.761 & 0.319 & -0.249 & 1.642 & -2.06 \end{bmatrix} \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 1 \\ 0 \end{bmatrix}$$

$$+ \begin{bmatrix} -0.322 & -0.384 \\ 1.134 & -1.1 \end{bmatrix} \begin{bmatrix} -0.872 \\ 0.274 \end{bmatrix} + \begin{bmatrix} -0.268 \\ 0.53 \end{bmatrix}$$

$$Z^{[1]\langle 5\rangle} = \begin{bmatrix} 0.597 \\ 1.992 \end{bmatrix}$$

$$H^{[1]\langle 5\rangle} = \varphi_h\left(Z^{[1]\langle 5\rangle}\right)$$

$$H^{[1]\langle 5\rangle} = \begin{bmatrix} 0.535 \\ 0.963 \end{bmatrix}$$

$$O^{[2]\langle 5\rangle} = \left(W_o^{[2]}\right)^{\mathbf{T}} H^{[1]\langle 5\rangle} + b^{[L]}$$

$$O^{[2]\langle 5\rangle} = \begin{bmatrix} -0.172 & -0.878 \\ 0.042 & 0.583 \\ -1.101 & 1.145 \\ 0.902 & 0.502 \\ 0.901 & -0.684 \\ -0.123 & -0.936 \end{bmatrix}^{\mathbf{T}} \begin{bmatrix} 0.535 \\ 0.963 \end{bmatrix} + \begin{bmatrix} -0.692 \\ -0.397 \\ -0.687 \\ -0.845 \\ -0.671 \\ -0.013 \end{bmatrix}$$

$$O^{[2]\langle 5\rangle} = \begin{bmatrix} -1.63 \\ -0.187 \\ 0.173 \\ 0.121 \\ -0.848 \\ -0.98 \end{bmatrix}$$

The output Y at time $t = 5$ is computed as:

$$\hat{Y}^{[2]\langle 5\rangle} = \varphi_o\left(O^{[2]\langle 5\rangle}\right)$$

$$\hat{Y}^{[2]\langle 5\rangle} = \begin{bmatrix} 0.047 \\ 0.289 \\ 0.201 \\ 0.27 \\ 0.103 \\ 0.09 \end{bmatrix}$$

For $t = 6$, from Eq. $(F.4)$

$$Z^{[1]\langle 6\rangle} = W_x^{[0]} X^{[0]\langle 6\rangle} + W_z^{[1]} H^{[1]\langle 5\rangle} + b^{[l]}$$

$$Z^{[1]\langle 6\rangle} = \begin{bmatrix} 1.624 & -0.612 & -0.528 & -1.073 & 0.865 & -2.302 \\ 1.1745 & -0.761 & 0.319 & -0.249 & 1.642 & -2.06 \end{bmatrix} \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 1 \end{bmatrix}$$

$$+ \begin{bmatrix} -0.322 & -0.384 \\ 1.134 & -1.1 \end{bmatrix} \begin{bmatrix} 0.535 \\ 0.963 \end{bmatrix} + \begin{bmatrix} -0.268 \\ 0.53 \end{bmatrix}$$

$$Z^{[1]\langle 6\rangle} = \begin{bmatrix} -2.57 \\ -1.53 \end{bmatrix}$$

$$H^{[1]\langle 6\rangle} = \varphi_h\left(Z^{[1]\langle 6\rangle}\right)$$

$$H^{[1]\langle 6\rangle} = \begin{bmatrix} -0.988 \\ -0.91 \end{bmatrix}$$

$$O^{[2]\langle 6\rangle} = \left(W_o^{[2]}\right)^{\mathbf{T}} H^{[1]\langle 6\rangle} + b^{[L]}$$

$$O^{[2]\langle 6\rangle} = \begin{bmatrix} -0.172 & -0.878 \\ 0.042 & 0.583 \\ -1.101 & 1.145 \\ 0.902 & 0.502 \\ 0.901 & -0.684 \\ -0.123 & -0.936 \end{bmatrix}^{\mathbf{T}} \begin{bmatrix} -0.988 \\ -0.91 \end{bmatrix} + \begin{bmatrix} -0.692 \\ -0.397 \\ -0.687 \\ -0.845 \\ -0.671 \\ -0.013 \end{bmatrix}$$

$$O^{[2]\langle 6\rangle} = \begin{bmatrix} 0.247 \\ 0.071 \\ 0.099 \\ 0.021 \\ 0.073 \\ 0.489 \end{bmatrix}$$

The output Y at time $t = 6$ is computed as:

$$\hat{Y}^{[2]\langle 6\rangle} = \varphi_o\left(O^{[2]\langle 6\rangle}\right)$$

$$\hat{Y}^{[2]\langle 6\rangle} = \begin{bmatrix} 0.247 \\ 0.071 \\ 0.099 \\ 0.021 \\ 0.073 \\ 0.489 \end{bmatrix}$$

Hence, predicted probabilities vectors are $\hat{Y}^{[2]\langle 1\rangle}, \hat{Y}^{[2]\langle 2\rangle}, \hat{Y}^{[2]\langle 3\rangle}, \hat{Y}^{[2]\langle 4\rangle}, \hat{Y}^{[2]\langle 5\rangle}, \hat{Y}^{[2]\langle 6\rangle}$ given:

$$\hat{Y}^{[2]\langle 1\rangle} = \begin{bmatrix} 0.041 \\ 0.275 \\ 0.131 \\ 0.345 \\ 0.128 \\ 0.079 \end{bmatrix}, \hat{Y}^{[2]\langle 2\rangle} = \begin{bmatrix} 0.174 \\ 0.145 \\ 0.213 \\ 0.051 \\ 0.08 \\ 0.337 \end{bmatrix}, \hat{Y}^{[2]\langle 3\rangle} = \begin{bmatrix} 0.066 \\ 0.21 \\ 0.494 \\ 0.072 \\ 0.038 \\ 0.121 \end{bmatrix}$$

$$\hat{Y}^{[2]\langle 4\rangle} = \begin{bmatrix} 0.107 \\ 0.177 \\ 0.42 \\ 0.052 \\ 0.045 \\ 0.199 \end{bmatrix}, \hat{Y}^{[2]\langle 5\rangle} = \begin{bmatrix} 0.047 \\ 0.289 \\ 0.201 \\ 0.27 \\ 0.103 \\ 0.09 \end{bmatrix}, \hat{Y}^{[2]\langle 6\rangle} = \begin{bmatrix} 0.247 \\ 0.071 \\ 0.099 \\ 0.021 \\ 0.073 \\ 0.489 \end{bmatrix}$$

Description of Resulting Probabilities: The expected probability distribution over the vocabulary represents the RNN's outputs for each time step. Given the input sequence up to that time step each element in the probability distribution represents the possibility that the appropriate word from the vocabulary will appear as the subsequent word.

For each time step in this example, the outputs are shown as probability vectors, as follows:
$\hat{Y}^{[2]\langle 1\rangle}$= (corresponding to the predicted probabilities for the first word after "The")
$\hat{Y}^{[2]\langle 2\rangle}$= (corresponding to the predicted probabilities for the second word after "The cat")
$\hat{Y}^{[2]\langle 3\rangle}$ = (corresponding to the predicted probabilities for the third word after "The cat sat")
$\hat{Y}^{[2]\langle 4\rangle}$= (corresponding to the predicted probabilities for the fourth word after "The cat sat on")
$\hat{Y}^{[2]\langle 5\rangle}$ = (corresponding to the predicted probabilities for the fifth word after "The cat sat on the mat")
$\hat{Y}^{[2]\langle 6\rangle}$= (corresponding to the predicted probabilities for the sixth word after "The cat sat on the mat smoothly")

These probabilities are obtained by applying the Softmax function to the continuous values produced by the RNN at each time step. The highest probability in each predicted probability distribution represents the most likely next word in the sequence.

For example, in the first-time step, the word “on” has the highest probability of 0.345 which indicates that the RNN predicts “on” to be the most likely next word in the sequence after “The cat sat on”. However, RNN also assigns some probability to other words, such as "cat" with a probability of 0.275, indicating some uncertainty in the prediction. It is expected because the RNN is still learning to make accurate predictions based on the input sequence.

As the RNN is trained using backpropagation over several iterations, it modifies the weights and biases of the network to reduce prediction mistakes and improve prediction accuracy. The RNN is anticipated to eventually produce more accurate predictions and give the subsequent word in the sequence a higher probability.

7. Long Short-Term Memory (LSTM)

A memory cell has been added to the RNN that stores data in memory for a prolonged time, known as Long Short-Term Memory (LSTM). An amended version of RNN was created in 1997 by Hochreiter and Schmindhuber. The LSTM can pick up on Long-term dependencies between data pieces and make inferences from them. They can also solve issues when sequential multiplication of the weight matrix $W^{[l]}$ in recurrent layers results in an unstable gradient that either vanishes or explodes.

It can be used for issues like TSA, speech recognition, and translating languages. Many tasks related to Natural Language Processing (NLP) such as Sentiment Assessment (SA), Question Answering (QA) and Automated Translation (AT or MT) have been accomplished using LSTM networks. RNNs cannot determine what memory to store and what to dispose of because the hidden state must be updated at every time step, but LSTM has a far more comprehensive range of parameters and better control over this. This specialized neural network design uses a series of gates to regulate the information flow permitting it to retain and gain access to data over a prolonged time (Kedia & Rasu, 2020). The three gates manage the information flow across arbitrary time intervals. A basic schematic structure of LSTM architecture is shown in Fig. 7.1 (Santos, 2021).

7.1. Forward Propagation

- Each LSTM cell is mainly composed of five different states:
- Cell State/ Memory Cell: The memory cell stores information, allowing LSTM to remember information for long periods.
- Hidden State: The hidden state is the output of the LSTM network at each time step. It is used as the input for the next time step and can be used as the network's final output. It is calculated by applying the output gate and the Tanh function to the current cell state.
- Input Gate/update Gate: The input gate stores the information and controls whether to accept new input into the memory cell.
- Forget Gate: The forget gate controls whether to retain or forget information from the previous memory cell state. It contains the information that is no longer important.
- Output Gate: The output gate manages the information moving from the memory cell to the output layer. This gate stores the information to use at the current step.
- An example of an LSTM cell is shown in Fig. 7.3.

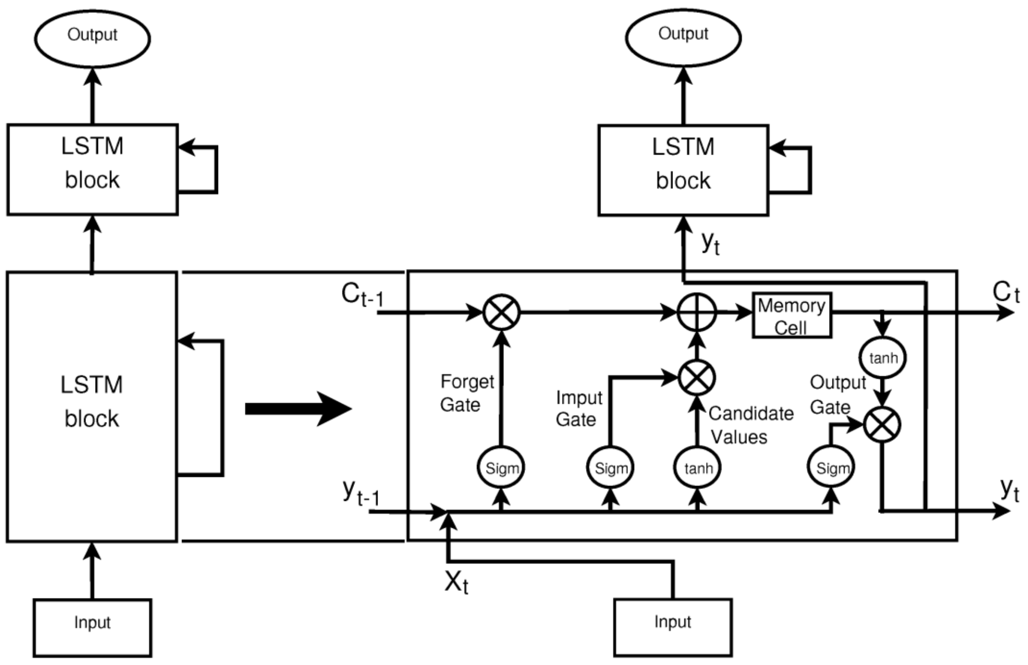


Fig. 7.1: Schematic Structure of a LSTM cell (Image Credits: Santos, 2021)

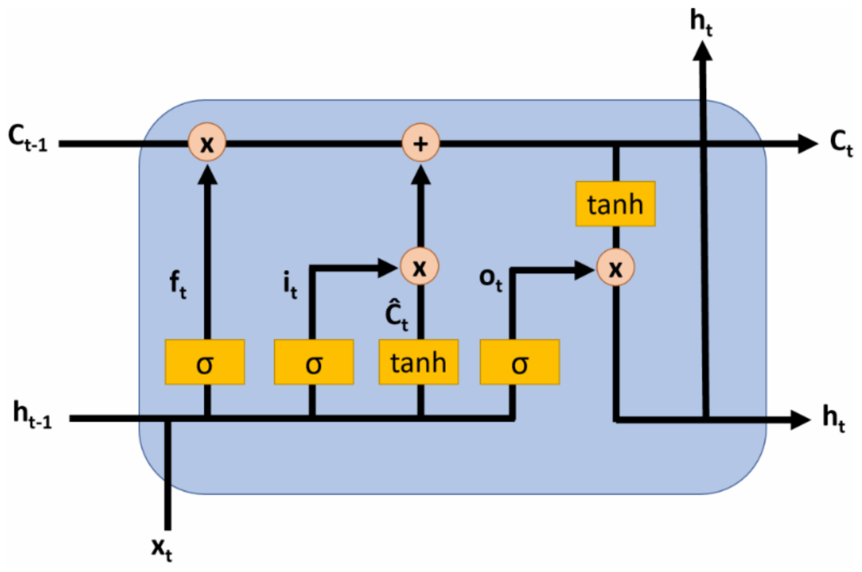


Fig. 7.3: Gates of LSTM (Image Credits: Tayeh, 2022)

Let's explain the working of each of these gates individually to clarify the concept of the forward pass of an LSTM.

7.1.1. Forget Gate

The Forget Gate is the LSTM cell's earliest synapse. First, the forget gate receives the concatenation vector $[H^{[l]\langle t\rangle}, X^{[0]\langle t\rangle}]$ from the current state's input and the prior state's output. Which values from the previous cell state should be preserved and which should be forgotten is decided by the forget gate $\Gamma^{\{f\}\langle t\rangle}$. It carries out this job using Sigmoid activation.

Its output is a vector of values between 0 and 1 and its inputs are the weighted input vector plus the previous hidden state. The product will be close to zero if any values in $\Gamma^{\{f\}\langle t\rangle}$ are zero or almost zero. As a result, the data kept in the associated unit $C^{[l]\langle t-1\rangle}$ has been preserved for the time step that follows. Corresponding to this, if a value is approximately 1 the result is comparable to its original value in the prior cell state. For use in a subsequent time step the LSTM will retain the data from the associated unit of $C^{[l]\langle t-1\rangle}$.

$$\begin{aligned}\left[Z^{\{f\}[l]\langle t\rangle}\right]_{n_h\times 1} &= \left[W^{\{f\}[l]}\right]_{n_h\times 1}\left[H^{[l]\langle t-1\rangle}\right]_{n_h\times 1} \\ &\quad+\left[W^{\{f\}[l]}\right]_{n_h\times n_x}\left[X^{[0]\langle t\rangle}\right]_{n_x\times 1}+\left[b^{\{f\}[l]}\right]_{n_h\times 1} \qquad ---(7)\end{aligned}$$

After applying activation function and concatenating hidden state and input state vectors.

$$\left[\Gamma^{\{f\}\langle t\rangle}\right]_{n_h\times 1} = \sigma\left(\boldsymbol{W}^{\{f\}[l]}\left[H^{[l]\langle t-1\rangle},X^{[0]\langle t\rangle}\right]^{\mathbf{T}} + b^{\{f\}[l]}\right) \qquad ---(8)$$

The previously mentioned cell state's $C^{[l]\langle t-1\rangle}$ dimension is similar to the forget gate. Where:

σ is the sigmoid activation function used to make each gate tensor's values $\Gamma^{\{f\}\langle t\rangle}$ range from 0 to 1.

- $\left[X^{[l]\langle t\rangle}\right]_{n_x\times 1}$ is the encoded input vector.
- $\left[W^{\{f\}[l]}\right]_{n_h\times(n_h+n_x)}$ is the weight matrix that governs the forget gate's behaviour.
- $\left[H^{[l]\langle t-1\rangle}\right]_{n_h\times 1}$ is the hidden state at time $t-1$.
- $\left[b^{\{f\}[l]}\right]_{n_h\times 1}$ is the forget gate's bias vector.

For layer $l = 1,2,\cdots,L-1$,

$$\begin{aligned}\left[Z^{\{f\}[l]\langle t\rangle}\right]_{n_h\times 1} &= \left[W^{\{f\}[l]}\right]_{n_h\times 1}\left[H^{[l]\langle t-1\rangle}\right]_{n_h\times 1} \\ &\quad+\left[W^{\{f\}[l]}\right]_{n_h\times n_h}\left[H^{[l-1]\langle t\rangle}\right]_{n_h\times 1}+\left[b^{\{f\}[l]}\right]_{n_h\times 1} \qquad ---(9)\end{aligned}$$

After applying the activation function and concatenating hidden state and input state vectors.

$$\left[\Gamma^{\{f\}\langle t\rangle}\right]_{n_h\times 1} = \sigma\left(\boldsymbol{W}^{\{f\}[l]}\left[H^{[l]\langle t-1\rangle},H^{[l-1]\langle t\rangle}\right]^{\mathbf{T}} + b^{\{f\}[l]}\right) \qquad ---(10)$$

7.1.2. Input Gate/Update Gate

The input gate in LSTM is a Sigmoid layer, which determines the values from the input vector $X^{\langle t\ that\rangle}$ will be allowed to pass into the cell state $C^{\langle t\rangle}$. It decides which values from the input should be "remembered" (closer to 1) by the cell state and which values should be "forgotten" (closer to 0). This task uses the same activation function as in the forget gate. For example

"Lionel Messi is good football player. Robert is also central player."

The input gate will ensure that model will remember "*Robert"* as it approaches the second sentence, while the forget gate helped to forget about *Lionel Messi*.

The input gate which consists of two sections aids in determining what has to be memorized and how much must be maintained at once. Next, let's examine the roles played by these two components in more detail.

Part 1 by producing values between 0 and 1 this part employs a Sigmoid activation function to determine which portion of the input data has to be preserved. A value of 0 would indicate that nothing from the inputs to this state needs to be kept but a value of 1 would indicate that everything must be remembered.

For layer $l = 0$,

$$\begin{aligned}\left[Z^{\{i\}[l]\langle t\rangle}\right]_{n_h\times 1} &= \left[W^{\{i\}[l]}\right]_{n_h\times 1}\left[H^{[l]\langle t-1\rangle}\right]_{n_h\times 1} \\ &\quad +\left[W^{\{i\}[l]}\right]_{n_h\times n_x}\left[X^{[0]\langle t\rangle}\right]_{n_x\times 1}+\left[b^{\{i\}[l]}\right]_{n_h\times 1} \qquad ---(11)\end{aligned}$$

$$\left[\Gamma^{\{i\}\langle t\rangle}\right]_{n_h\times 1} = \sigma\left(\boldsymbol{W}^{\{i\}[l]}\left[H^{[l]\langle t-1\rangle}, X^{[0]\langle t\rangle}\right]^{\mathbf{T}} + b^{\{i\}[l]}\right) \qquad ---(12)$$

For layer $l = 1,2,\cdots,L-1$,

$$\begin{aligned}\left[Z^{\{i\}[l]\langle t\rangle}\right]_{n_h\times 1} &= \left[W^{\{i\}[l]}\right]_{n_h\times 1}\left[H^{[l]\langle t-1\rangle}\right]_{n_h\times 1} \\ &\quad +\left[W^{\{i\}[l]}\right]_{n_h\times n_x}\left[H^{[l-1]\langle t\rangle}\right]_{n_x\times 1}+\left[b^{\{i\}[l]}\right]_{n_h\times 1}\end{aligned}$$

$$\left[\Gamma^{\{i\}\langle t\rangle}\right]_{n_h\times 1} = \sigma\left(\boldsymbol{W}^{\{i\}[l]}\left[H^{[l]\langle t-1\rangle}, H^{[l-1]\langle t\rangle}\right]^{\mathbf{T}} + b^{\{i\}[l]}\right) \qquad ---(13)$$

Part 2 uses the Tanh activation function which helps us to figure out relevant information from the current state. This component is also usually referred to as the candidate vector because it holds the values that the memory cell might be updated accordingly. A tensor between -1 and 1 makes up the candidate value's output range. Here, $p\tilde{C}^{\langle t\rangle}$ denotes candidate vector and $\left[W^{\{i\}[l]}\right]_{n_h\times(n_h+n_x)}$ denotes candidate value weight matrix

A tensor (Candidate value) of data from the most recent time step that could be kept in the present cell state $C^{[l]\langle t\rangle}$. The candidate state $\tilde{C}^{\langle t\rangle}$ is separated from the cell state $C^{[l]\langle t\rangle}$ using the tilde "$\sim$".

7.1.3. Output Gate

One of the three primary gates in a LSTM network is the output gate $\Gamma^{[o]\langle t\rangle}$. Its function regulates the information flow from the memory cell to the LSTM network's output. The following equation describes the output gate computation:

For layer $l = 0$

$$\begin{aligned}\left[Z^{\{o\}[l]\langle t\rangle}\right]_{n_z\times 1} &= \left[W^{\{o\}[l]}\right]_{n_h\times 1}\left[H^{[l]\langle t-1\rangle}\right]_{n_h\times 1} \\ &\quad +\left[W^{\{o\}[l]}\right]_{n_h\times n_x}\left[X^{[0]\langle t\rangle}\right]_{n_x\times 1}+\left[b^{\{o\}[l]}\right]_{n_h\times 1}\end{aligned}$$

$$\left[\Gamma^{\{o\}\langle t\rangle}\right]_{n_h\times 1} = \sigma\left(\boldsymbol{W}^{\{o\}[l]}\left[H^{[l]\langle t-1\rangle}, X^{[0]\langle t\rangle}\right]^{\mathbf{T}} + b^{\{o\}[l]}\right) \qquad ---(14)$$

For layer $l = 1,2,\cdots,L-1$

$$\begin{aligned}\left[Z^{\{o\}[l]\langle t\rangle}\right]_{n_z\times 1} &= \left[W^{\{o\}[l]}\right]_{n_h\times 1}\left[H^{[l]\langle t-1\rangle}\right]_{n_h\times 1} \\ &\quad +\left[W^{\{i\}[l]}\right]_{n_h\times n_x}\left[H^{[l-1]\langle t\rangle}\right]_{n_x\times 1}+\left[b^{\{o\}[l]}\right]_{n_h\times 1}\end{aligned}$$

$$\left[\Gamma^{\{o\}\langle t\rangle}\right]_{n_h\times 1} = \sigma\left(\boldsymbol{W}^{\{o\}[l]}\left[H^{[l]\langle t-1\rangle}, X^{[0]\langle t\rangle}\right]^{\mathbf{T}} + b^{\{o\}[l]}\right) \qquad ---(15)$$

7.1.4. Cell State

The "memory" that gets passed on to upcoming time steps is the cell state. The candidate value $C^{[l]\langle t\rangle}$ and the prior cell state $C^{[l]\langle t-1\rangle}$ combine to form the new cell state $C^{[l]\langle t\rangle}$. Tensor multiplication $\Gamma^{\{f\}\langle t\rangle} * C^{[l]\langle t\rangle}$ is equivalent to putting the mask of cells over the preceding cell state.
The cell state is updated as follows:

$$\left[C^{[l]\langle t\rangle}\right]_{n_h\times 1} = \left[\Gamma^{\{i\}\langle t\rangle}\right]_{n_h\times 1} * \left[\tilde{C}^{\langle t\rangle}\right]_{n_h\times 1} + \left[\Gamma^{\{f\}\langle t\rangle}\right]_{n_h\times 1} * \left[C^{[l]\langle t\rangle}\right]_{n_h\times 1}$$

$$\left[C^{[l]\langle t\rangle}\right]_{n_h\times 1} = \Gamma^{\{i\}\langle t\rangle} * \tilde{C}^{\langle t\rangle} + \Gamma^{\{f\}\langle t\rangle} * C^{[l]\langle t-1\rangle} \qquad ---(16)$$

Here, $*$ indicates the element-wise multiplication.

7.1.5. Hidden State

The hidden state is used to determine the next step's three gates and for prediction $H^{[L-1]\langle t\rangle}$ which is determined by cell state in combination with output gate $\Gamma^{\{o\}\langle t\rangle}$. This cell state is passed through the tanh to rescale values between -1 and 1. This gate acts like a "*mask*" that either preserves the values of tanh $\left(C^{[l]\langle t\rangle}\right)$ or keeps those values from being included in the hidden state $H^{[l]\langle t\rangle}$.

$$\left[H^{[l]\langle t\rangle}\right]_{n_h\times 1} = \left[\Gamma^{\{o\}\langle t\rangle}\right]_{n_h\times 1} + \mathrm{Tanh}\left[C^{[l]\langle t\rangle}\right]_{n_h\times 1}$$

$$\left[H^{[l]\langle t\rangle}\right]_{n_h\times 1} = \Gamma^{\{o\}\langle t\rangle} * \mathrm{Tanh}\left(C^{[l]\langle t\rangle}\right)$$

The purpose of the output gate is to selectively output only relevant information from the memory cell while suppressing irrelevant information. It allows the LSTM to effectively remember and use relevant information from previous time steps while ignoring noise or irrelevant information. It is advantageous in speech recognition and language translation applications, where the LSTM needs to output relevant information while processing sequential data selectively.

7.1.6. Prediction

For prediction in LSTM the softmax activation function is used.

$$\left[\hat{Y}^{\langle t\rangle}\right]_{n_x\times 1} = \mathrm{Softmax}\left(\left[W^{\{y\}[l]}\right]_{n_x\times n_h}\left[H^{[L-1]\langle t\rangle}\right]_{n_h\times 1} + \left[b^{\{y\}[L]}\right]_{n_x\times 1}\right)_{n_x\times 1} \qquad ---(17)$$

Here, $W^{\{y\}[l]} \in \mathbb{R}^{n_x\times n_h}$, bias vector $b^{\{y\}[l]} \in \mathbb{R}^{n_x}$, hidden state vector $H^{[l]\langle t\rangle} \in \mathbb{R}^{n_h}$.

8. Gated Recurrent Unit (GRU)

GRU is a simplified version of the LSTM that does not use explicit cell states. The GRU uses a single reset gate to accomplish the same task as the LSTM which uses separate forget and output gates to regulate the amount of information modified in the hidden state. The critical feature of GRUs is gating mechanisms to selectively update and reset the hidden state at each time step. Specifically, a GRU has two gates. One is a reset gate and the other is an update gate. The update gate (long-term memory) controls how much of the new input and the reset gate's output should be utilized to update the hidden state while the reset gate (short-term memory) controls how much of the prior hidden state should be forgotten. With these gates, the GRU can identify long-term dependencies in sequential data and avoid the vanishing gradient issue that might arise in conventional/Vanilla RNNs. A single GRU cell is shown in Fig. 8.1.

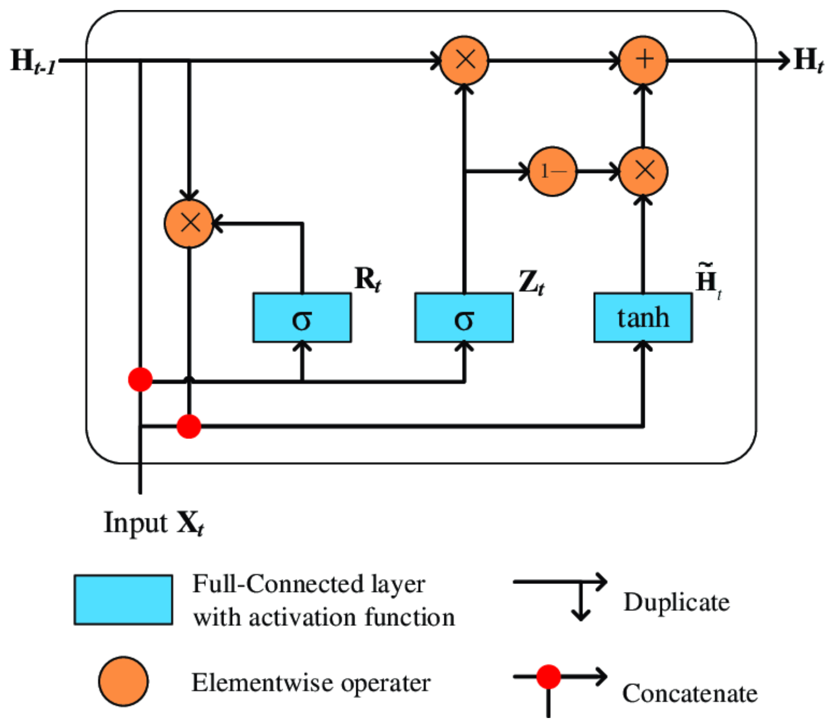


Fig. 8.1: The architecture of a gated recurrent unit (GRU) cell (Image Credits: Fang, 2021)

8.1. Reset Gate

The reset gate decides how much of the prior memory state will be erased at the current time step. It returns a value between 0 and 1 for each member in the memory state vector using the current input and the previous hidden state as inputs. An element should be completely overlooked if its value is 0 but entirely remembered if it is 1. Mathematically, its written as:

For layer $l = 1$,

$$\begin{aligned}\left[Z^{\{r\}[l]\langle t\rangle}\right]_{2n_h\times 1} &= \left[W^{\{r\}[l]}\right]_{2n_h\times n_h}\left[H^{[l]\langle t-1\rangle}\right]_{n_h\times 1} + \left[W^{\{r\}[l-1]}\right]_{2n_h\times n_x}\left[X^{[0]\langle t\rangle}\right]_{n_x\times 1}\\ &\quad +\left[b^{\{r\}[l]}\right]_{2n_h\times 1}\\ \left(\boldsymbol{W}^{\{r\}[l]}\right)_{2n_h\times(n_h+n_x)} &= \left[W^{\{r\}[l]}\middle|W^{\{r\}[l-1]}\right]\end{aligned}$$

After applying the activation function and concatenating hidden state and input state vectors:

$$\Gamma^{\{r\}\langle t\rangle} = \sigma\left(\boldsymbol{W}^{\{r\}[l]}\left[H^{[l]\langle t-1\rangle}, X^{[0]\langle t\rangle}\right]^{\mathbf{T}} + b^{\{r\}[l]}\right) \qquad ---(18)$$

For layer $l = 2,3,\ldots,L$

$$\begin{aligned}\left[Z^{\{r\}[l]\langle t\rangle}\right]_{2n_h\times 1} &= \left[W^{\{r\}[l]}\right]_{2n_h\times n_h}\left[H^{[l]\langle t-1\rangle}\right]_{n_h\times 1} + \left[W^{\{r\}[l-1]}\right]_{2n_h\times n_h}\\ &\quad +\left[b^{\{r\}[l]\langle t\rangle}\right]_{2n_h\times 1}\\ \left(\boldsymbol{W}^{\{r\}[l]}\right)_{2n_h\times 2n_h} &= \left[W^{\{r\}[l]}\middle|W^{\{r\}[l-1]}\right]\end{aligned}$$

After applying the activation function and concatenating hidden state vectors:

$$\Gamma^{[r]\langle t\rangle} = \sigma\left(W^{\{r\}[l]}\left[H^{[l]\langle t-1\rangle}H^{[l-1]\langle t\rangle}\right]^{\mathbf{T}} + b^{\{r\}[l]}\right) \qquad ---(19)$$

8.2. Update Gate

The update gate decides the amount of the new memory state that should be based on the current input and prior memory state. It returns a value between 0 and 1 for each member in the memory state vector using its present input and the earlier hidden state as inputs. A value of 0 indicates that the element should entirely depend on the prior state of the memory. In contrast, a value of 1 indicates that the element should entirely depend on the current input.

For layer $l = 1$

$$\begin{aligned}\left[Z^{\{u\}[l]\langle t\rangle}\right]_{2n_h\times 1} &= \left[W^{\{u\}[l]}\right]_{2n_h\times n_h}\left[H^{[l]\langle t-1\rangle}\right]_{n_h\times 1} + \\ &\quad \left[W^{\{u\}[l-1]}\right]_{2n_h\times n_x}\left[X^{[0]\langle t\rangle}\right]_{n_x\times 1} + \left[b^{\{u\}[l]}\right]_{2n_h\times 1}\\ \left(\boldsymbol{W}^{\{u\}[l]}\right)_{2n_h\times(n_h+n_x)} &= \left[W^{\{u\}[l]}|W^{\{u\}[l-1]}\right]\end{aligned}$$

After applying the activation function and concatenating hidden state and input state vectors:

$$\Gamma^{[u]\langle t\rangle} = \sigma\left(\boldsymbol{W}^{\{u\}[l]}\left[H^{[l]\langle t-1\rangle}, X^{[0]\langle t\rangle}\right]^{\mathbf{T}} + b^{\{u\}[l]}\right) \qquad ---(20)$$

For layer $l = 2,3,\ldots,L$

$$\begin{aligned}\left[Z^{\{u\}[l]\langle t\rangle}\right]_{2n_h\times 1} &= \left[W^{\{u\}[l]}\right]_{2n_h\times n_h}\left[H^{[l]\langle t-1\rangle}\right]_{n_h\times 1} + \\ &\quad \left[W^{\{u\}[l]}\right]_{2n_h\times n_x}\left[H^{[l-1]\langle t\rangle}\right]_{n_x\times 1} + \left[b^{\{u\}[l]}\right]_{2n_h\times 1}\\ \left(\boldsymbol{W}^{\{u\}[l-1]}\right)_{2n_h\times 2n_h} &= \left[W^{\{u\}[l]}|W^{\{u\}[l]}\right]\end{aligned}$$

After applying the activation function and concatenating hidden state and input state vectors:

$$\Gamma^{\{u\}\langle t\rangle} = \sigma\left(\boldsymbol{W}^{\{u\}[l]}\left[H^{[l]\langle t-1\rangle}, H^{[l-1]\langle t\rangle}\right]^{\mathbf{T}} + b^{\{u\}[l]}\right) \qquad ---(21)$$

8.3. Candidate Activation

The candidate activation state is an intermediate state computed during the network's forward pass before the final output is produced. The candidate activation state is used to update the network's hidden state and compute the final output. During each time step of the GRU the candidate activation state is computed by combining the current input with the previous hidden state passing this combination through a non-linear activation function, such as the hyperbolic tangent (Tanh) function. The candidate activation state is then combined with the previous hidden state using the update gate which determines how much of each to retain.

For layer $l = 1$

$$\begin{aligned}\left[Z^{\{c\}[l]\langle t\rangle}\right]_{n_h\times 1} &= \left[W^{\{c\}[l]}\right]_{n_h\times n_h}\left[\left[H^{[l]\langle t-1\rangle}\right]_{n_h\times 1} * \Gamma^{\{r\}\langle t\rangle}\right] + \\ &\quad \left[W^{\{c\}[l-1]}\right]_{n_h\times n_x}\left[X^{[0]\langle t\rangle}\right]_{n_x\times 1} + \left[b^{\{c\}[l]}\right]_{n_h\times 1}\\ \left(\boldsymbol{W}^{\{c\}[l]}\right)_{n_h\times(n_h+n_x)} &= \left[W^{\{c\}[l]}|W^{\{c\}[l-1]}\right]\end{aligned}$$

After applying the activation function and concatenating hidden state and input state vectors:

$$\widetilde{H}^{[l]\langle t\rangle} = \mathrm{Tanh}\left(\boldsymbol{W}^{\{c\}[l]}\left[H^{[l]\langle t-1\rangle}, X^{[0]\langle t\rangle}\right]^{\mathbf{T}} + b^{\{c\}[l]}\right) \qquad ---(22)$$

For layer $l = 2,3,\ldots,L$

$$\left[Z^{\{c\}[l]\langle t\rangle}\right]_{n_h\times 1} = \left[W^{\{c\}[l]}\right]_{n_h\times n_h}\left[\left[H^{[l]\langle t-1\rangle}\right]_{n_h\times 1} * \Gamma^{\{r\}\langle t\rangle}\right] + \left[W^{\{c\}[l-1]}\right]_{n_h\times n_h}\left[H^{[l-1]\langle t\rangle}\right]_{n_h\times 1} + \left[b^{\{c\}[l]}\right]_{n_h\times 1}$$

$$\left(\boldsymbol{W}^{\{c\}[l-1]}\right)_{n_z\times 2n_h} = \left[W^{\{c\}[l]} | W^{\{c\}[l-1]}\right]$$

After applying the activation function and concatenating hidden state and input state vectors:

$$\widetilde{H}^{[l]\langle t\rangle} = \mathrm{Tanh}(\boldsymbol{W}^{\{c\}[l-1]}\left[\Gamma_t^r * H^{[l]\langle t-1\rangle}, H^{[l-1]\langle t\rangle}\right]^{\mathbf{T}} + b^{\{c\}[l]} \qquad ---(23)$$

8.4. Update Hidden State

$$H^{[l]\langle t\rangle} = \Gamma^{\{u\}\langle t\rangle} * H^{[l]\langle t-1\rangle} + \left(1 - \Gamma^{\{u\}\langle t\rangle}\right) * \widetilde{H}^{[l]\langle t\rangle} \qquad ---(24)$$

The update hidden state combines the previous and candidate activation states determined by the update gate. The update hidden state is used to compute the final output of the GRU at each time step.

Reset gate $\Gamma^{\{r\}\langle t\rangle}$ controls how much of the previous hidden state to forget and how much to maintain for the current time step in the GRU architecture. Update gate $\Gamma^{\{u\}\langle t\rangle}$ controls how much of the previous hidden state to update. The GRU's update gate $\Gamma^{\{u\}\langle t\rangle}$ controls how much the prior concealed state and the matrix-based update contribute to the final result. Based on the current input, it chooses how much of the former concealed state to keep and how much to update. It is similar to how the input and forget gates work in the LSTM, but in the GRU the update gate performs both functions simultaneously. However, the reset $\Gamma^{\{r\}\langle t\rangle}$ and update $\Gamma^{\{u\}\langle t\rangle}$ gates in the GRU are intermediate "scratch-pad" variables used to update the hidden state $H^{[l]\langle t\rangle}$. They allow the GRU to selectively forget or remember information from the previous hidden state based on the current input. It is essential for modelling long-term dependencies in sequential data.

9. Bidirectional RNNs (BRNNs)

Its already seen that RNNs are used for processing sequential data. However, one of their major limitations is that the state at a particular time step can only have information about the past inputs up to a certain point in the sequence and has no knowledge about future states. It can limit the effectiveness of RNNs for specific applications such as language modelling where knowing both past and future states can lead to improved results.

For example, in handwriting recognition knowing the past and future symbols can provide a better idea of the underlying context which can be crucial for accurately recognizing the intended symbols. By processing the input sequence in both forward and backward directions, bidirectional RNNs can capture dependencies and patterns that may be missed by a unidirectional RNN leading to improved performance in tasks like language modelling, speech recognition and sentiment analysis.

The basic structure of BRNNs is shown in Fig. 9.1.

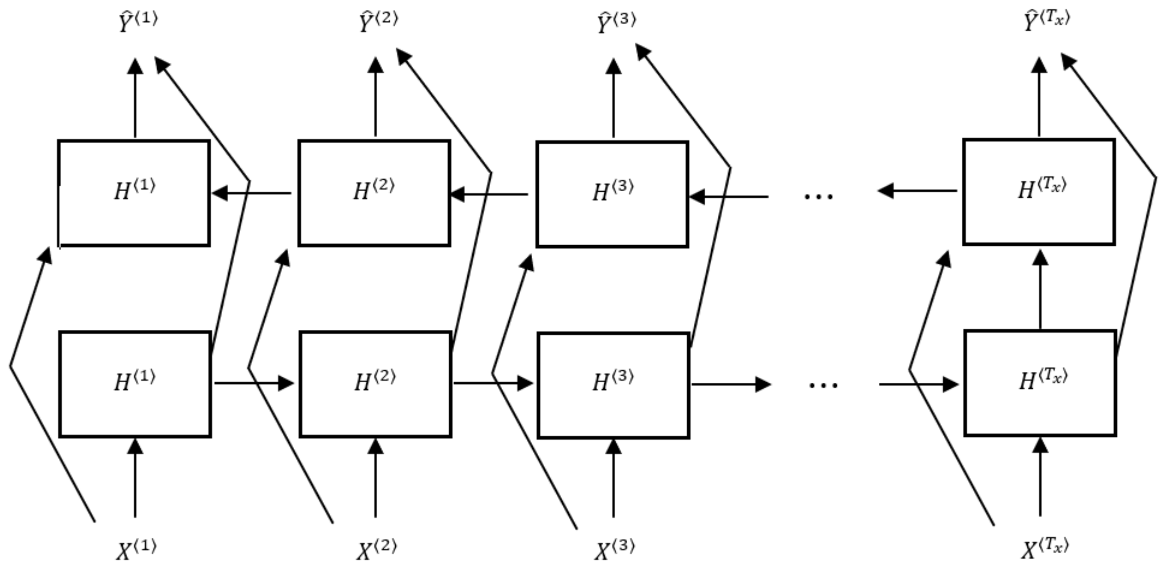


Fig. 9.1: Bidirectional RNNs Basic Structure

9.1. Working of BRNNs

In a bidirectional RNN, the input sequence is processed forward and backward by two RNNs, which are combined to produce the final output. The mathematical equations for a bidirectional RNN are:

9.1.1. Forward Hidden Layer

The intermediate/Pre-activation state of BRRNs can be written as:

$$Z^{\{f\}[l]\langle t\rangle} = W_x^{\{f\}[l]} X^{\langle t\rangle} + W_h^{\{f\}[l]} H^{\{f\}\langle t-1\rangle}$$

$H^{\{f\}\langle t\rangle} = \sigma(X^{\langle t\rangle}, H^{\{f\}\langle t-1\rangle})$ represents that the forward hidden state is the function of the current input node and the previous forward hidden node.
After applying the activation function

$$H^{\{f\}[l]\langle t\rangle} = \tanh\left(W_x^{\{f\}[l]} X^{\langle t\rangle} + W_h^{\{f\}[l]} H^{\{f\}\langle t-1\rangle}\right) \qquad ---(25)$$

9.1.2. Backwards Hidden Layer

The backward hidden state in BRNN is the function of current input $X^{\langle t\rangle}$ and the next hidden state $H^{\{b\}\langle t+1\rangle}$ in the backward direction.

$$Z^{\{b\}[l]\langle t\rangle} = W_x^{\{b\}[l]} X^{\langle t\rangle} + W_h^{\{b\}[l]} H^{\{b\}\langle t+1\rangle}$$

$$H^{\{b\}[l]\langle t\rangle} = \sigma\left(W_x^{\{b\}[l]} X^{\langle t\rangle} + W_h^{\{b\}[l]} H^{\{b\}\langle t+1\rangle}\right) \qquad ---(26)$$

9.1.3. Output Layer

The output late of BRNN is the function of both forward and backwards hidden states at the previous and current time steps, respectively.

$$\begin{aligned} O^{\{f\}[l]} &= W_o^{\{f\}[l]} H^{\{f\}\langle t-1\rangle} + W_o^{\{b\}[l]} H^{\{b\}\langle t\rangle} \\ \hat{Y}^{\langle t\rangle} &= \varphi_o\left(W_o^{\{f\}} H^{\{f\}\langle t-1\rangle} + W_o^{\{b\}} H^{\{b\}\langle t\rangle}\right) \qquad ---(27) \end{aligned}$$

Here,

- $\hat{Y}^{\langle t\rangle} = S\left(H^{\{f\}\langle t-1\rangle}, H^{\{b\}\langle t\rangle}\right)$ is the function of forward and backwards hidden state at time step t.
- $X^{\langle t\rangle}$ represents the input vector at time step t.
- $H^{\{f\}\langle t-1\rangle}$ and $H^{\{b\}\langle t\rangle}$ represent the hidden states of the forward and backward RNNs, respectively, at time step t.
- σ is the activation function of the RNN, which computes the hidden state at each time step based on the input and the previous hidden state. φ_o is the output function which combines the forward and backward RNNs outputs to produce the final output at time step t.

10. Sentiment Analysis with Simple RNN, LSTM, GRU and BRNNs using Transfer Learning: Mapping Emotions to Emojis

10.1. Problem Description

The Emojifier problem is a specific application of sentiment analysis which identify the sentiment or emotion conveyed in each text. In this problem, the goal is to predict an appropriate emoji to represent the emotion of a given text input (Sanyamdhawan, 2020). For example, given the sentence "Let's go see the baseball game tonight!", the Emojifier should ⚾ suggest the baseball emoji. To build the Emojifier model, Glove word vectors embedding used which is mathematical representations of words that capture their semantic meaning. The main benefit of using word vector representation is that even if the training set explicitly relates only a few words to a particular emoji the algorithm will be able to generalize and associate words in the test set to the same emoji even if those words do not even appear in the training set. It allows us to build an accurate classifier mapping from sentences to emojis even using a small training set.

10.2. Existing Studies

Ramaswamy et al. (2019) propose a federated learning approach for emoji prediction on a mobile keyboard. Their approach uses a novel "federated transfer learning" technique to transfer knowledge from a global model to local models on individual devices. It allows them to achieve better accuracy than previous FL approaches for emoji prediction. They evaluate their approach on a real-world dataset of keyboard input logs and show that it can achieve comparable accuracy to a centralized model trained on the same dataset. They also show that their approach is more privacy-preserving than the centralized model, as it does not require users to share their local data with the cloud. Sutskever et al. (2022) propose a deep learning approach for contextual emotion detection in text. Their approach uses a combination of word embeddings, LSTM networks, and attention mechanisms to capture the context of the text and identify the underlying emotion. They evaluate their approach on a real-world dataset of text messages and show that it can achieve an accuracy of 85%. Kone et al. (2023) propose a bi-directional LSTM model for emoji prediction. Their approach uses a bidirectional LSTM to learn the context of the text, both before and after the current word. They evaluate their approach on a Twitter dataset and show that it can achieve an accuracy of 94%. Acheampong et al. (2020) review the advances, challenges, and opportunities in text-based emotion detection. They discuss text-based emotion detection approaches, including lexicon-based, machine-learning, and deep-learning approaches. They also discuss the challenges of text-based emotion detection, such as the ambiguity of language, the lack of labelled data, and the need for context-aware emotion detection. Finally, they discuss the opportunities for text-based emotion detection such as its use in customer service, healthcare, and education.

10.3. Challenges in the Emojifier Problem

One of the main challenges in the Emojifier problem is that the meaning and usage of emojis can be subjective and context-dependent making it challenging to define explicit mappings between textual inputs and emojis. Additionally, an emoji's meaning can vary across cultures and languages making it essential to ensure that the model is culturally appropriate and sensitive. Another challenge is that multiple emojis can correspond to a given sentiment or emotion making it challenging to determine which emoji is the most appropriate. To address these challenges researchers have developed a variety of approaches for solving the Emojifier problem. The traditional hybrid approach is a rule-based method which relies on explicit rules or heuristics to make decisions. Rule-based and knowledge-based approaches may be more efficient and easier to interpret, but they have some limitations in that they may perform less well as deep learning models on complex datasets.

Choosing the best model for solving the Emojifier problem depends on several factors such as the dataset's size and complexity the desired accuracy level and the available computational resources. RNNs are often favored due to their ability to capture sequential and contextual information in the text input. Still, they can be computationally expensive and require copious amounts of training data.

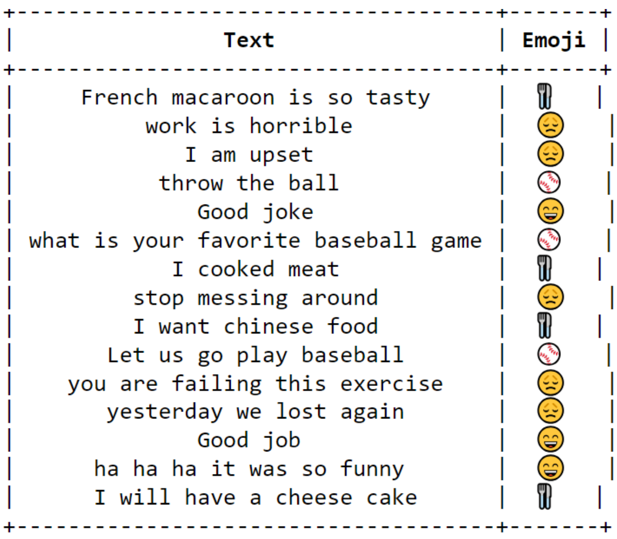
10.4. Methodology

The open-source data used to train the Emojifier includes pairs of sentences and their corresponding emojis available on Kaggle. It contains 183 Training instances and 56 testing data. For example, one instance might be "I love you so much" paired with the emoji ❤️ . The data is represented in Table 10.1.

X (Sentences)	***Y (Labels)***
I love you so much	0 ❤️
I think it will end up alone	3 😞
Miss you so much	0 ❤️
Food is life	1 🍴
I want to go play	4 ⚾
Never talk to me again	3 😞

Table 10.1: Emojify Data

The model uses this data to learn the relationship between particular words and emojis and then applies that knowledge to new sentences it has not seen before. By carefully examined the dataset and extracted useful information using visualization approaches. Five different classes are represented visually in Fig. 10.1.



```
+--------------------------------------+-------+
|                 Text                 | Emoji |
+--------------------------------------+-------+
|      French macaroon is so tasty     |  🍴   |
|          work is horrible            |  😞   |
|             I am upset               |  😞   |
|            throw the ball            |  ⚾   |
|              Good joke               |  😄   |
| what is your favorite baseball game  |  ⚾   |
|            I cooked meat             |  🍴   |
|          stop messing around         |  😞   |
|          I want chinese food         |  🍴   |
|        Let us go play baseball       |  ⚾   |
|      you are failing this exercise   |  😞   |
|        yesterday we lost again       |  😞   |
|               Good job               |  😄   |
|       ha ha ha it was so funny       |  😄   |
|        I will have a cheese cake     |  🍴   |
+--------------------------------------+-------+
```

Fig. 10.1: Visualization of data

10.5. Exploratory Data Analysis

Keras Mini-batching and Padding: Keras ought to be trained using mini-batches. The majority of deep learning frameworks demand that every sequence in a given mini-batch be identical in length, which makes vectorization possible. The computations required for a 4-word statement and a 5-word sentence are different from one another (one requires four steps of an LSTM the other needs five steps). Consequently, it would be unfeasible to handle both simultaneously. Padding is a frequent technique for dealing with sequences of various lengths. Specifically:

- Choose the lengthiest conceivable sequence.
- To make every sequence the same length, pad it.

Embedding Layer

In this phase an `Embedding()` layer in Keras is developed with GloVe 50-dimensional vectors to initiate the Embedding layer. Because of the shortness of the training set same GloVe embedding is used in each iteration.

Components of the Embedding Layer's Inputs and Outputs:

- The input matrix size is (m, n_x)
- The vocabulary size should not exceed the input's largest possible integer (the maximum word position). A visual representation of the Embedding layer can be seen in Fig. 10.2
- The dimension of the output array is (m, n_x, d).
- n_x = maximum input length
- m = Batch Size
- d = Dimension of words vector

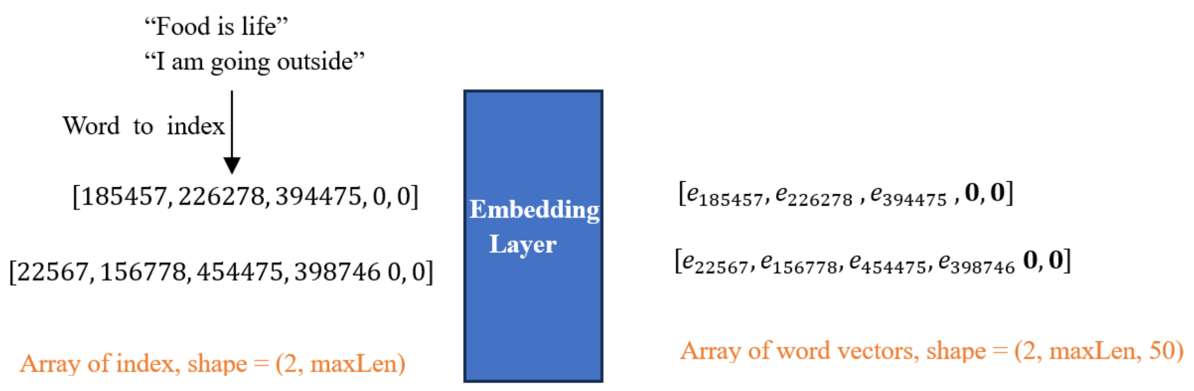

"Food is life"
"I am going outside"
Word to index
[185457, 226278, 394475, 0, 0]
[22567, 156778, 454475, 398746 0, 0]
Array of index, shape = (2, maxLen)
Embedding Layer
[e185457, e226278 , e394475 , 0, 0]
[e22567, e156778, e454475, e398746 0, 0]
Array of word vectors, shape = (2, maxLen, 50)

Fig. 10.2: Embedding Layer

Pretrained Embedding Layer

A trained embedding layer is essential in NLP activities. It is used to seed word embeddings with knowledge from big text corpora such as Word2Vec or GloVe word vectors and then fine-tune them for a specific downstream application. A trained embedding layer can be helpful in modifier data which entails interpreting and creating emojis or emoticons depending on text input. Even if the training data for the modifier data is minimal it allows the model to collect semantic information and links between words and emojis. The model can start with a base of general language understanding by exploiting pre-trained word embeddings, which can help it produce more contextually relevant emoji predictions. This method saves substantial training time and data while improving the modifie's effectiveness in comprehending and creating expressive emoticons in text-based interactions.

11. Emojifier-C_1: Baseline Model

A baseline and sequence model accomplish multiple purposes and have distinct features. The baseline model is the most straightforward or beginning entry point in machine learning. It is often a simple model that does not include complicated procedures or architectures. The algorithm cannot recognize the negation in a sentence like *"This movie is not good and not enjoyable,"* misclassifying it as a positive. It is because the model averages all of the word embeddings in the phrase together and ignores the words' order or the context in which they are used.

- Forward pass for Baseline Model

$$z^{(i)} = Wavg^{(i)} + b$$
$$h^{(i)} = \text{softmax}(z^{(i)})$$

- Loss Function

$$E^{(i)} = \sum_{k=0}^{n_y-1} Y_{oh,k}^{(i)} + \log\left(h_k^{(i)}\right)$$

Here, $Y_{oh,k}^{(i)}$ ("Y one hot") is the one-hot encoding of the output labels.

11.1. Model Architecture

The baseline model architecture uses a Softmax layer to train word vector representations. Xavier initialization begins by initializing model parameters such as weight matrices and bias vectors. The forward propagation algorithm computes the Softmax layer's output by performing a linear transformation on the input data followed by a softmax activation function. The cross-entropy-based cost function quantifies the difference between expected and accurate labels. Gradients are computed in backward propagation to update parameters using gradient descent. This process is repeated over a set number of training repetitions with batch updates where the model updates its parameters to minimize cost at each epoch. The architecture uses external functions to perform tasks such as converting labels to one-hot vectors and averaging word embeddings from input phrases. It also includes an evaluation stage in which predictions for the training data are generated regularly during training.

11.1.1. Training Results

The model has a high cost of 1046.09 initially (Epoch 0) indicating that it performs poorly in predicting the proper labels for the dataset. The accuracy is equally low at 0.57 implying that only 57% of the predictions match the actual labels. Its seen substantial development as the training proceeds. The cost has dropped substantially to 270.46 by Epoch 100 but the accuracy has increased to 94.54%. It shows that the model has trained to make considerably better predictions and can now accurately classify emojis. The cost continues to reduce slightly but slower in Epochs 200 and 300. The accuracy is continuously high with results ranging between 94.54% and 92.90%. These findings indicate that the model has converged and performs well on the training dataset.

11.1.2. Test Results

The baseline model's test results on the "Emojify" dataset reveal an accuracy of roughly 85.71% on the test set, which is a reasonable performance. Given that there are five classes, random guessing would only offer a 20% accuracy in this situation (1/5 = 20%). As a result, even with a minimal training dataset of 183 samples, the model's accuracy greatly surpasses random chance, indicating that it has learned meaningful correlations between words and emojis. With only 239 samples, the model could benefit from more data. The model's success depends on training data terms; unexpected words can confuse it. As a basic model, it may need help to understand intricate word-emoji relationships. The model faces difficulties due to the context-dependent meanings of emojis.

Confusion Matrix

The confusion matrix shown in Fig. 11.1 is intended to assess the effectiveness of Baseline model. The matrix compares the anticipated to the actual emojis $(0, 1, 2, 3, 4)$. The values are broken out as follows: Correct predictions are represented by the diagonal elements (from top-left to bottom-right). For example, the model predicted emoji 0 six times, emoji 1 eight times, emoji two twelve times, emoji three sixteen times and emoji 4 six times. Off-diagonal elements indicate misclassifications. The model predicted emoji 0 when the actual emoji was three once and emoji 4 when the real emoji was three once. The model predicted 11 emoji $0, 8$ emoji 1, 12 emoji $2, 19$ emoji 3 and 6 emoji 4. The model predicted 11 times for emoji $0, 8$ times for emoji $1, 12$ times for emoji $2, 19$ times for emoji 3 and 6 times for emoji 4.

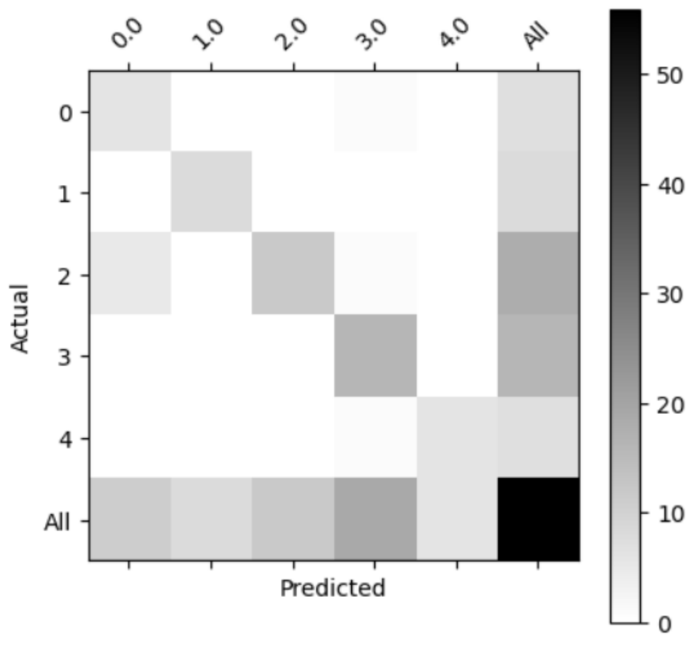


Fig. 11.2: Confusion Matrix

12. Emojifier-C_2: LSTM in Keras

On the other hand, LSTM is a specially designed RNN variant for dealing with sequential data with LSTM having more complicated memory abilities. To effectively address the intricacies of language including word selection and sentence composition a 2-layer LSTM structure is employed.

12.1. Model Architecture

The LSTM (Long Short-Term Memory) model architecture is built to classify text or sequence-to-sequence applications. The Architecture of LSTM used here is defined as:

- **Input Layer:** The model is fed a succession of input texts. In this example, each phrase is represented as a sequence of word indices with a maximum length of 10 words.
- **Embedding Layer:** The Embedding Layer is pre-trained using GloVe word vectors which are then utilized to turn the word indices into dense word embeddings. These embeddings capture semantic information about words and assist the model in comprehending the meaning of each word in the input phrases. Because it is initialized with pre-existing word vectors the embedding matrix is not trainable in this scenario.
- **LSTM Layers:** On top of the embedding layer two LSTM layers are stacked. These are RNNs that can recognize sequential dependencies and context within input texts. The first LSTM layer contains 200 units and returns sequences which outputs one for each word in the input sequence. The second LSTM layer has 200 units as well but it produces a single hidden state that summarizes the entire input sequence. This LSTM layer architecture enables the model to comprehend both individual word and sentence contexts.
- **Dropout Layers:** With a dropout rate 0.2 dropout layers are added after each LSTM layer. Dropout prevents overfitting by randomly assigning a percentage of the input units to zero during training forcing the model to acquire more resilient representations.

- **Dense Layer:** Following the LSTM layers is a dense (ultimately linked) layer with five units. The final classification is performed by this layer which maps the learnt features from the LSTM layers to the output classes. Because the activation function employed in this example is sigmoid' which is typically used for binary classification, it appears to be a binary classification task.
- **Activation Layer:** The final activation layer applies the sigmoid activation function to the dense layer output producing the model's predictions. Sigmoid works best for binary classification problems in which each of the five output units corresponds to a binary class (e.g., positive or negative emotion).

12.2. Training Results

The training results indicate the model's performance across 100 epochs on a training dataset, which means it went through the entire training dataset 100 times. The loss measures how well the model's predictions match the target values in the training dataset. It is a mathematical value that the model tries to minimize during training. In this example, the loss begins relatively high (about 1.55) in the first epoch and gradually reduces with each epoch. The model is learning to generate better predictions as training continues. Accuracy is a metric that expresses the proportion of correctly categorized examples in the training dataset as a percentage. The accuracy (about 27.32%) in the first epoch could be higher demonstrating that the model's initial predictions could be more reliable. However, as training progresses the model's accuracy rapidly rises and approaches 100% as it learns to categorize the training data correctly. Each epoch's duration is specified in seconds. The time required depends on the model's complexity and the size of the training dataset. The initial epoch took substantially longer (about 10 seconds) in this example, presumably due to some early setup or data loading. Following epochs are even faster lasting only a fraction of a second.

12.3. Test Result

The test results show the trained model's performance on a distinct dataset. The reported test accuracy is approximately 0.8393 which equates to approximately 83.93%. It means that when tested on the test dataset the model successfully identified about 83.93% of the samples. In other words, it performed admirably on this hitherto encountered data. A test accuracy of approximately 83.93% indicates that the model has acquired significant patterns and representations from the training data and can generalize pretty well to new previously unknown samples. However, analyzing the task's specific context is critical to determine whether this degree of accuracy is adequate. An accuracy of 83.93% may be excellent for some applications but it may require more development in others. It's also worth mentioning that the test loss which isn't expressly stated but usually goes hand in hand with accuracy is roughly 1.3550. The loss measures how well the model's predictions correspond to the actual labels in the test dataset. A reduced test loss means the forecasts and actual values are more aligned.

13. Emojifier-C_3: GRU in Keras

Let's build a GRU-based model for the Emojify dataset using pre-trained word embeddings and a GRU layer with 200 units. It has become a significant tool in NLP notably in sentiment analysis applications. GRUs are a recurrent neural network (RNN) architecture that has acquired prominence because of their ability to successfully capture long-range dependencies in sequential data while reducing some of traditional RNN shortcomings such as the vanishing gradient problem.

Training and Test Results: The GRU model's training results on the Emojifier dataset are pretty good with the model obtaining 100% accuracy after 100 epochs. This shows that the model learned to recognize and forecast the proper emojis based on input text during training, indicating a good ability to fit the training data. When it comes to test data the model's accuracy is slightly lower at around 83.93%. While this test accuracy remains good it is lower than the training accuracy. This performance gap between training and test accuracy implies that the model is slightly overfitting the training data which means it has learned to memorize the training examples rather than generalize to new previously unknown data.

Nonetheless, an accuracy of 83.93% on the test data indicates that the model is working well and capable of making reasonably accurate emoji predictions on new text inputs which is a good outcome for the Emojifier assignment. Further fine-tuning or regularization approaches may be able to close the gap between training and test accuracy allowing for even improved generalization.

14. Emojifier_C_4: Bidirectional RNNs

The Bidirectional RNN model consists of two LSTM layers: one processing the input sequence in a forward direction and the other in a backward direction. The outputs of these two layers are concatenated and passed through a dense layer to predict the most likely emoji for the input message. Build a Bidirectional RNN-based model for the Emojify dataset using pre-trained word embeddings and an LSTM layer with 200 units. Pre-trained word embedding is used to represent each word in the text as a fixed-length vector.

Training and Test Results: The accuracy of a bidirectional neural network model trained over 100 epochs improves significantly over time. Initially, the model had an accuracy of only 25.14% in the first epoch showing that it was performing poorly. However, as training went the accuracy steadily improved eventually reaching 100% by epoch 25. This implies that the model learns the patterns and features in the training data well and with near-perfect accuracy. The model was tested on a second dataset during the testing phase and achieved an accuracy of 83.93%. This result shows that the model generalized well to previously unseen data implying that it can make good predictions on new previously unseen examples.

15. Emojifier_C_4: Simple RNNs

To predict emoji labels for input texts, the Emojify-SimpleRNN model employs a series of layers. It begins by encoding words with pre-trained GloVe vectors. Two Simple RNN layers with 200 units capture the sentence's sequential information with dropout for regularization. The first layer returns concealed state sequences while the second summarizes the data into a fixed-size representation. Emojis are assigned probabilities via a dense layer with 5 units and Softmax activation allowing categorization. This architecture effectively uses recurrent layers to understand phrase context and predict emojis while avoiding overfitting via dropout.

Training and Test Results: The training results show that the Emojify-SimpleRNN model performs very well during training, obtaining 100% accuracy on the training data after 100 epochs. This indicates that the model learned to classify phrases into the relevant emoji groups during the training procedure. It is important to note that gaining high accuracy on training data does not necessarily guarantee equivalent performance on unknown data as overfitting can occur.

When tested on the test set, the model retains a relatively high accuracy of roughly 83.93%. This is a good indicator because it indicates that the model has generalized reasonably well to data that it hasn't seen during training. However, by apply the model to fresh previously unseen case some variation in performance is noticed. The programmed accurately predicts a happy emoji for the statement "I am not happy." However, it predicts a cheerful emoji for the statement "This movie was not good," which may not be fully accurate given the negative feeling in the sentence. In the example of "This movie is not good and not enjoyable," it predicts a sad emoji which corresponds to the negative attitude.

15.1. Results Comparison

The significant trends are observed in the performance of the different Emojifier models when compare their results. While the Simple RNN model achieves 100% accuracy on training data, it has certain limits when tested on unknown instances, indicating potential overfitting. The GRU and LSTM models on the other hand exhibit comparable and good performance with both obtaining an accuracy of around 87% on the test data, proving their robustness in generalizing to new samples as shown in Fig. 15.1. The Bidirectional RNN model has a fascinating behavior beginning with a lower accuracy of 25.14% on the first epoch but progressively improving with time eventually attaining 100% accuracy on the training data and roughly 83.93% accuracy on the test data. This shows that further training data or fine tuning may be required for the Bidirectional RNN to properly capture the patterns in the data.

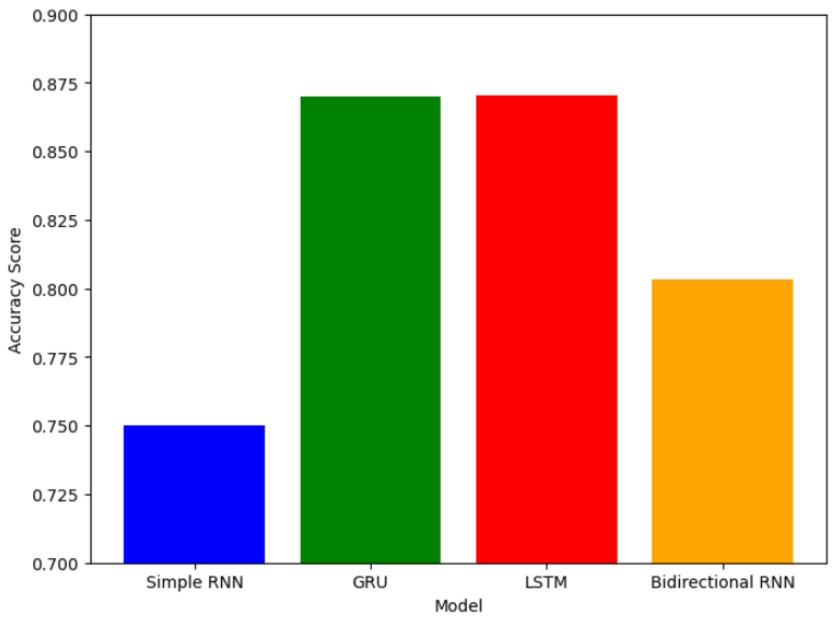


Fig. 15.1: Comparison of accuracy scores of various RNN Models for Emojifier dataset

The GRU and LSTM models come out as outstanding performers displaying their ability to handle sequential data and emoji categorization tasks. While the Simple RNN performs admirably during training it may require regularization approaches to increase its generalization to new cases as shown in Fig. 15.2.

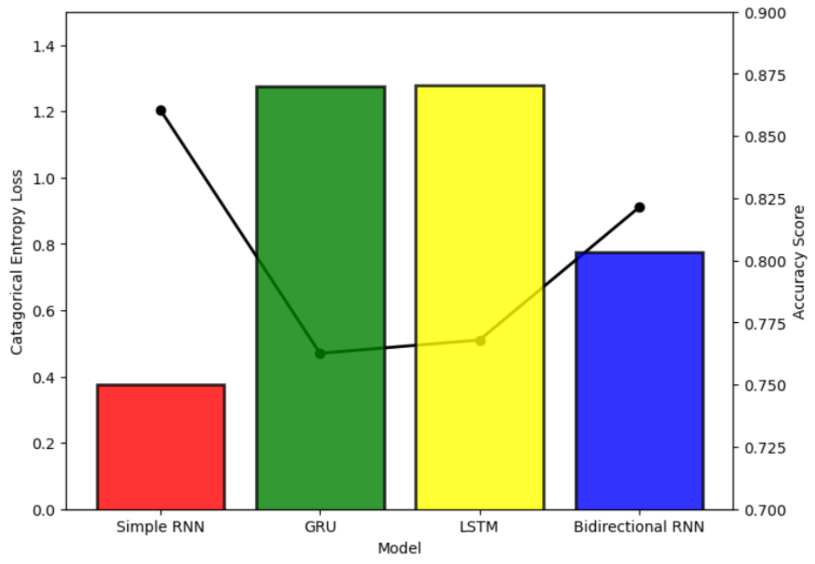


Fig. 15.2: Comparison of loss and accuracy score of various RNN Models for Emojifier dataset

16. Predicting Stock Price Using LSTM (A Time Series Analysis Example)

We analyze the share price history of 22 years of the dataset Pak Suzuki Motor Company Limited available from (Kaggle – Pakistan Stock Exchange Data, 2022). Our objective is to predict stock prices as a regression problem, using LSTM.

In this study, a robust approach for time-series prediction using Recurrent Neural Networks (RNNs) is proposed. The first step in the methodology involves preprocessing of the data to enhance model convergence and stability. The data is normalized to ensure that all values fall within a specified range. After preprocessing, the data is split into train and test sets with 95 percent data allocated to the training set. The training set is further divided into a reduced train set and a validation set. To create sequences suitable for time-series prediction, a data transformation step using a sliding window technique is introduced. The model architecture comprises an LSTM layer with 64 units, followed by two fully connected layers with 32 and 1 neurons, respectively. A tanh and ReLU activation functions are applied to the LSTM and the first fully connected layer, respectively. A sigmoid activation function is applied to the second fully connected layer. To optimize the model, the Adam optimizer with a 0.001 learning rate is employed. Mean squared error is adopted as the loss function for model training, and mean absolute error serves as a metric to assess model performance. The model is trained over 100 epochs with a batch size of 64, and training progress is monitored using both training and validation data. The graph for training and validation loss is shown in Fig. 16.1.

The trained model is utilized for predicting the target values on the test dataset. The comparison of predicted and true values is shown in Fig. 16.2. The Mean Absolute Error (MAE) and Root Mean Squared Error (RMSE) are 4.22109 and 5.68928, respectively.

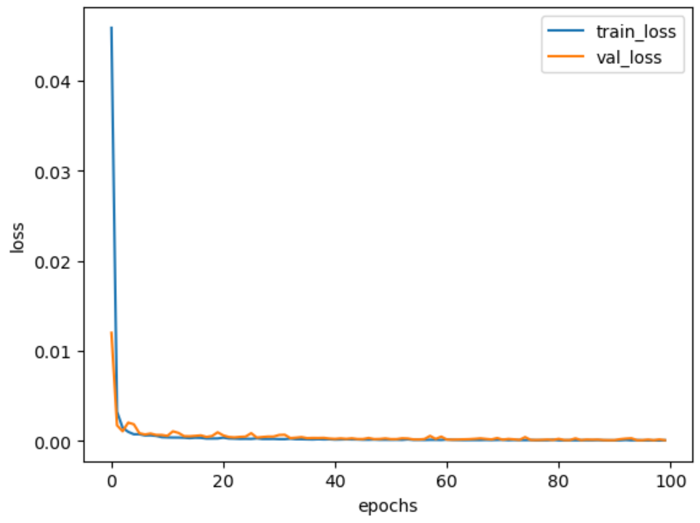


Fig. 16.1: Plot of train and validation loss functions

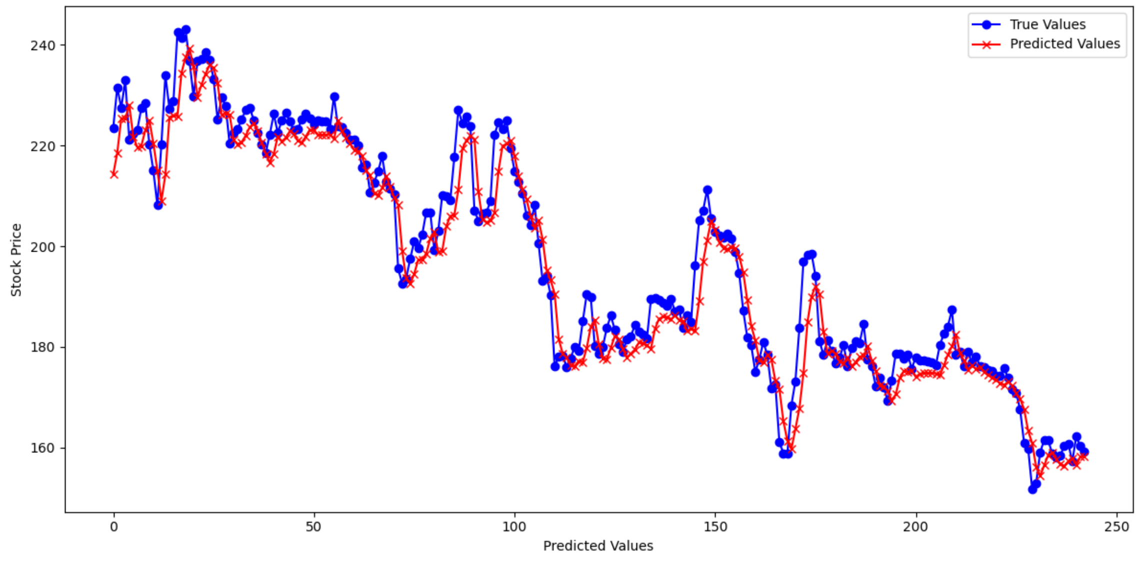


Fig. 16.2: Comparison of True and Predicted values of Stock prices

Conclusion

This article provides a detailed mathematical working of Recurrent Neural Networks (RNNs) and their different variants, such as LSTM, GRU, and BRNNs. The Backpropagation Through Time (BPTT) algorithm is explained, along with the computation and propagation of gradients through time for updating the parameters of the network. The article includes a worked-out simple example demonstrating word prediction. A sentiment analysis problem involving Emojifier is evaluated using simple RNNs, LSTM, GRU, and BRNNs with Transfer Learning. It is observed that LSTM gives the highest accuracy, demonstrating its efficiency in predicting the appropriate emoji for a given text input. The BRNN model has shown a slightly better performance than the LSTM and GRU models. The article aims to empower readers with the knowledge to develop a solid basis for sustained progress in deep learning using RNNs. Finally, an example time series analysis involving prediction of stock process using LSTM is presented.

Acknowledgements

The authors thank Prof. Dr. Muhammad Ali Ismail (Principal Investigator) and Mr Uzair Abid (Team Lead) at the National Center for Big Data & Cloud Computing, NED University of Engineering and Technology, Karachi, Pakistan, for their continuous guidance and support.

Conflicts of Interest: The authors have no conflicts of interest to declare relevant to this article's content.

Funding: No funding was received for this research work and the article.

References

Acheampong, F., Chen, W., Mensah, H.N., (2020). Text-Based Emotion Detection: Advances, Challenges and Opportunities.

Aggarwal, C. C. (2018). Neural Networks and Deep Learning: A Textbook. Springer.

Alexender, A. (2023). MIT 6.S191: Introduction to Deep Learning. YouTube. https://youtube.com/playlist?list=PLtBw6njQRU-rwp5__7C0oIVt26ZgjG9NI. http://introtodeeplearning.com

Bhavsar, K., & Dangeti, P. (2017). Natural Language Processing with Python Cookbook, Packt Publishing Ltd.

Bird, S., Klein, E., & Loper, E. (2009). Natural Language Processing with Python. O'Reilly Media.

Bishop, C. M. (2016). Pattern recognition and machine learning. Springer.

Chandra, P., Ahammed, T., Ghosh, S., Emon, R., Billah, M., Ahamad, M., Balaji, P. (2022). Contextual Emotion Detection in Text using Deep Learning and Big Data. 1-5. 10.1109/ICCSEA54677.2022.9936154.

Chen, G. (2016). A Gentle Tutorial of Recurrent Neural Network with Error Backpropagation.

Chung, J., Gulcehre, C., Cho, K., & Bengio, Y. (2014). Empirical Evaluation of Gated Recurrent Neural Networks on Sequence Modeling. ArXiv. /abs/1412.3555

Coşkun, M., Yildirim, O., Uçar, A., & Demir, Y. (2017). An overview of popular deep learning methods. European Journal of Technique, 7(2), 165–176 https://doi.org/10.23884/ejt.2017.7.2.11

Dawani, J. (2020). Hands-On Mathematics for Deep Learning: Build a solid mathematical foundation for training efficient deep neural networks. Packt Publishing Ltd.

Dey, R., & Salem, F. M. (2017). Gate-Variants of Gated Recurrent Unit (GRU) Neural Networks. ArXiv. /abs/1701.05923

D'informatique, D. N, E., Esent, P., Au, E., Gers, F., Hersch, P., Esident, P., Frasconi, P. (2001). Long Short-Term Memory in Recurrent Neural Networks. 10.5075/epfl-thesis-2366.

Fang, Y., Yang, S., Zhao, B., & Huang, C. (2021). Cyberbullying Detection in Social Networks Using Bi-GRU with Self-Attention Mechanism. Information, 12(4), 171. https://doi.org/10.3390/info12040171

Ganegedara, T. (2018). Natural Language Processing with TensorFlow: Teach language to machines using Python's deep learning library. Packt Publishing Ltd.

GitHub - Defcon27/Emoji-Prediction-using-Deep-Learning: https://github.com/Defcon27/Emoji-Prediction-using-Deep-Learning

Goodfellow, I., Bengio, Y., & Courville, A. (2016). Deep Learning. MIT Press

Graves, A. (2012). Supervised Sequence Labelling with Recurrent Neural Networks, Springer Berlin, Heidelberg. https://doi.org/10.1007/978-3-642-24797-2

Graves, A. (2013). Generating Sequences With Recurrent Neural Networks. ArXiv. /abs/1308.0850

Hagan, M., Demuth, H., Beale, M., & De Jesus, O. (2014). Neural Network Design, Second Edition. Martin Hagan.

Hastie, T., Tibshirani, R., & Friedman, J. H. (2009). The elements of statistical learning. In the Springer series in statistics. https://doi.org/10.1007/978-0-387-84858-7

Higham, C. F., Higham, D. J. (2019). Deep Learning: An Introduction for Applied Mathematicians. Siam Review, 61(3), 860–891. https://doi.org/10.1137/18m1165748

Hochreiter, S., Schmidhuber, J. (1997). Long Short-Term Memory. Neural Computation, 9(8), 1735–1780. https://doi.org/10.1162/neco.1997.9.8.1735

Hochreiter, S., Schmidhuber, J. (1997). Long Short-term Memory. Neural computation. 9. 1735-80. 10.1162/neco.1997.9.8.1735.

Kaggle - Pakistan stock exchange data (All Companies), (2022).

Karpathy. A., (2015). The Unreasonable Effectiveness of Recurrent Neural Networks. http://karpathy.github.io/2015/05/21/rnn-effectiveness/

Kedia, A., Rasu, M. (2020). Hands-On Python Natural Language Processing: Explore tools and techniques to analyze and process text with a view to building real-world NLP applications. Packt Publishing Ltd.

Kollipara, V. N. D. P., Kollipara, V. N. H., & Prakash, M. D. (2021). Emoji Prediction from Twitter Data using Deep Learning Approach. 2021 Asian Conference on Innovation in Technology (ASIANCON). https://doi.org/10.1109/asiancon51346.2021.9544680

Kone, V., Anagal, A., Anegundi, S., Jadekar, P., Patil, P., (2023). Emoji Prediction Using Bi-Directional LSTM. ITM Web of Conferences. 53. 02004. 10.1051/itmconf/20235302004.

Kutyniok, G. (2022). The Mathematics of Artificial Intelligence. ArXiv. /abs/2203.08890

LeCun, Y., Bngio, Y., Hinton, G. (2015). Deep Learning. Nature. 521. 436-44. 10.1038/nature14539.

Lipton, Z. C., Berkowitz, J., & Elkan, C. (2015). A Critical Review of Recurrent Neural Networks for Sequence Learning. ArXiv. /abs/1506.00019

Marhon, S.A., Cameron, C.J.F., Kremer, S.C. (2013). Recurrent Neural Networks. In: Bianchini, M., Maggini, M., Jain, L. (eds) Handbook on Neural Information Processing. Intelligent Systems Reference Library, vol 49. Springer, Berlin, Heidelberg. https://doi.org/10.1007/978-3-642-36657-4_2

Ng, A. (2018). Sequence Models. Deep learning. AI. https://www.coursera.org/learn/nlp-sequence-models/home

Pascanu, R. (2015). On Recurrent and Deep Neural Networks. Papyrus.bib.umontreal.ca. https://doi.org/1866/11452

Pascanu, R., Mikolov, T., & Bengio, Y. (2012). On the difficulty of training Recurrent Neural Networks. ArXiv. /abs/1211.5063

Prabhavalkar, R., Kanishka, R., Sainath, T., Li, B., Johnson, L., Jaitly, N. (2017). A Comparison of Sequence-to-Sequence Models for Speech Recognition. 939-943. 10.21437/Interspeech.2017-233.

Ramaswamy, S., Mathews, R., Rao, K., Beaufays, F. (2019). Federated Learning for Emoji Prediction in a Mobile Keyboard.

Salehinejad, H., Valaee, S., Dowdell, T., Colak, E., Barrett, J. (2018). Generalization of Deep Neural Networks for Chest Pathology Classification in X-Rays Using Generative Adversarial Networks. 10.1109/ICASSP.2018.8461430.

Sanyamdhawan. (2020). Emoji prediction using LSTM. Kaggle. https://www.kaggle.com/code/sanyamdhawan99/emoji-prediction-using-lstm

Sarker, I.H. Deep Learning: A Comprehensive Overview on Techniques, Taxonomy, Applications and Research Directions. SN COMPUT. SCI. 2, 420 (2021). https://doi.org/10.1007/s42979-021-00815-1

Saxena, S. (2023). Binary Cross Entropy/Log loss for binary classification. Analytics Vidhya. https://www.analyticsvidhya.com/blog/2021/03/binary-cross-entropy-log-loss-for-binary-classification/

Schmidt, R. M. (2019). Recurrent Neural Networks (RNNs): A Gentle Introduction and Overview. ArXiv. /abs/1912.05911

Schuster, A. Paliwal, K. K. "Bidirectional recurrent neural networks," in IEEE Transactions on Signal Processing, vol. 45, no. 11, pp. 2673-2681, Nov. 1997, doi: 10.1109/78.650093.

Sherstinsky, A. (2020). Fundamentals of Recurrent Neural Network (RNN) and Long Short-Term Memory (LSTM) network. Physica D: Nonlinear Phenomena, 404, 132306. https://doi.org/10.1016/j.physd.2019.132306

Srivastava, N., Hinton, G., Krizhevsky, A., Sutskever, I., Salakhutdinov, R., (2014). Dropout: A Simple Way to Prevent Neural Networks from Overfitting. Journal of Machine Learning Research. 15. 1929-1958.

Staudemeyer, R. C., & Morris, E. R. (2019). Understanding LSTM -- a tutorial into Long Short-Term Memory Recurrent Neural Networks. ArXiv. /abs/1909.09586

Strang, G. (2019). Linear Algebra and Learning from Data. Wellesley-Cambridge Press.

Sutskever, I. (2013). Training recurrent neural networks. University of Toronto, Toronto, Ont., Canada,

Sutskever, I., Martens, J., Hinton, G. (2011). Generating Text with Recurrent Neural Networks. Proceedings of the 28th International Conference on Machine Learning (ICML-11). 1017-1024.

Sutskever, I., Vinyals, O., & Le, Q. V. (2014). Sequence to Sequence Learning with Neural Networks. ArXiv. /abs/1409.3215

Santos, C. E. M., Da Silva, C. a. G., & Pedroso, C. M. (2021). Improving perceived quality of live adaptative video streaming. Entropy, 23(8), 948. https://doi.org/10.3390/e23080948

Tayeh, T., Aburakhia, S., Myers, R., & Shami, A. (2022). An Attention-Based ConvLSTM Autoencoder with Dynamic Thresholding for Unsupervised Anomaly Detection in Multivariate Time Series. (2022) Machine Learning and Knowledge Extraction, 4(2), 350–370. https://doi.org/10.3390/make4020015

Werbos, P. (1990). Backpropagation through time: what it does and how to do it. Proceedings of the IEEE. 78. 1550 - 1560. 10.1109/5.58337.