
اللہ اکبر
عَلَّمَ مَالَمْ
الْاِنْسَانَ يَعْلَمْ
جامعہ اسلامیہ

# DEVANAGARI TO ROMAN TRANSLITERATION

## Minor Project Report

Submitted in the partial fulfilment of the requirements for the award of the degree of

BACHELOR OF TECHNOLOGY

IN

COMPUTER ENGINEERING


**Submitted By**

Almas Ansari (19BCS042)

Amir Javed (19BCS044)

Tanmay Vig (19BCS061)

**Under the supervision of**

Mr. Mohd Zeeshan Ansari

(Assistant Professor)

**DEPARTMENT OF COMPUTER ENGINEERING**

**FACULTY OF ENGINEERING & TECHNOLOGY**

**JAMIA MILLIA ISLAMIA**

**NEW DELHI (110025)**

**(YEAR - 2022)**

# CERTIFICATE

This is to certify that the project entitled “Devanagari to Roman Transliteration” by Almas Ansari (19BCS042), Amir Javed (19BCS044) and Tanmay Vig (19BCS061) is a record of bonafide work carried out by them, in the Department of Computer Engineering, Jamia Millia Islamia, New Delhi, under my supervision and guidance in partial fulfilment of requirements for the award of Bachelor Of Engineering in Computer Engineering, Jamia Millia Islamia in the academic year 2020.

**Prof. Bashir Alam**
(Head of the Department)
Department of Computer Engineering
Faculty of Engineering & Technology
JAMIA MILLIA ISLAMIA
NEW DELHI

**Mr. Mohd Zeeshan Ansari**
(Assistant Professor)
Department of Computer Engineering
Faculty of Engineering & Technology
JAMIA MILLIA ISLAMIA
NEW DELHI

# ACKNOWLEDGEMENT

A very sincere and honest acknowledgement to Mr. Mohammad Zeeshan Ansari, Assistant Professor, Department of Computer Engineering, Jamia Millia Islamia, New Delhi for his invaluable technical guidance, great innovative ideas and overwhelming support. We are very grateful to our HOD Prof. Bashir Alam for his valuable support throughout the project. We would also like to express our gratitude to the Department of Computer Engineering and entire faculty members, for their teaching, guidance and encouragement.

We are also thankful to our classmates and friends for their valuable suggestions and support whenever required. We regret any inadvertent omissions.

Almas Ansari (19BCS042)

Amir Javed (19BCS044)

Tanmay Vig (19BCS061)

Department of Computer Engineering
Faculty of Engineering & Technology
JAMIA MILLIA ISLAMIA
NEW DELHI

# ABSTRACT

With increasing globalisation, information access across language barriers has become important. Given a source term, machine transliteration refers to generating its phonetic equivalent in the target language. This is important in many cross-language applications.

Transliteration is converting text from one language into another using a pre-established mapping. When a user is conversant in a language but not proficient in writing its script, this is helpful. India's lingua-franca is Hindi. In India, it is the language that is written and spoken the most. When pronouncing words and names in foreign languages, transliteration is helpful. Only the letters or characters of the source language should be changed into their equivalent letters in the target language. Contrary to translation, which changes the spoken or written meanings of words or texts in a source language into a target language, it does not render meaning.

The model utilises an Encoder-Decoder Architecture to translate a word from Devanagari script (Hindi) into Roman script (English). The Devanagari input sequence is transformed into an encoded version of the sequence by the network component known as the encoder network. The encoded representation is then used by the decoder network to generate a Roman script output sequence. Transliteration is helpful when dealing with proper nouns. LSTM can be used to improve the precision of the models by incorporating the previous outputs with current learnings.

# TABLE OF CONTENTS


| S. No. | Contents | | Page No. |
|---|---|---|---|
| | **CERTIFICATE** | | 2 |
| | **ACKNOWLEDGEMENT** | | 3 |
| | **ABSTRACT** | | 4 |
| | **TABLE OF CONTENTS** | | 5 |
| | **LIST OF FIGURES** | | 7 |
| | **LIST OF TABLES** | | 9 |
| | | | |
| 1 | INTRODUCTION | | 10 |
| | 1.1 | What Is Transliteration | 10 |
| | 1.2 | Translation v/s Transliteration | 11 |
| | 1.3 | Challenges Faced in Transliteration | 12 |
| | 1.4 | Applications of Transliteration | 13 |
| | 1.5 | Introduction to Deep Learning in Transliteration | 14 |
| 2 | REVIEW OF LITERATURE | | 16 |
| 3 | THEORETICAL BACKGROUND | | 18 |
| | 3.1 | Neural Networks | 18 |
| | 3.2 | Neuron | 19 |
| | 3.3 | Propagation (How Neural Networks Work) | 20 |
| | 3.4 | Feed Forward Neural Networks | 22 |
| | 3.5 | Backpropagation In Neural Networks | 24 |
| | 3.6 | Recurrent Neural Networks (RNN) | 25 |
| | 3.7 | Long Short Term Memory (LSTM) | 27 |
| | 3.8 | Encoder-Decoder Architecture | 29 |

| | | | |
|---|---|---|---|
| 4 | PROPOSED METHODOLOGY | | 31 |
| | 4.1 | Basic Outline | 32 |
| | 4.2 | Dataset Used | 33 |
| | 4.3 | Implementation Strategy | 35 |
| | 4.4 | Model Architecture Used | 36 |
| | 4.5 | Training | 38 |
| 5 | EVALUATION METRICS | | 41 |
| | 5.1 | Word Error Rate | 41 |
| | 5.2 | Character Error Rate | 42 |
| 6 | ANALYSIS OF RESULTS | | 43 |
| 7 | CONCLUSION & FUTURE WORK | | 45 |
| 8 | PROGRAMMING ENVIRONMENT AND TOOLS USED | | 46 |
| 9 | REFERENCES | | 48 |

# LIST OF FIGURES


| Fig. No. | Description | Page No. |
|---|---|---|
| 1.1 | Transliteration v/s Translation | 12 |
| 1.2 | Graphical representation of relationship between various fields in artificial intelligence | 15 |
| 3.1 | A Simple Neural Network | 18 |
| 3.2 | A Biological Neuron (left) & An Artificial Neuron (right) | 20 |
| 3.3 | Weights and Biases | 21 |
| 3.4 | Tanh Function (Hyperbolic Tangent) | 22 |
| 3.5 | A Feed Forward Neural Network with one hidden layer (and 3 neurons) | 23 |
| 3.6 | Backpropagation in Neural Networks | 25 |
| 3.7 | Recurrent Neural Networks | 26 |
| 3.8 | Recurrent Neural Network v/s Feed-Forward Neural Network | 27 |
| 3.9 | A Standard LSTM Architecture | 28 |
| 3.10 | Encoder-Decoder Architecture | 29 |
| 4.1 | Shapes of input to the Encoder-Decoder Architecture | 32 |
| 4.2 | The dataset used in its raw form | 34 |

| | | |
|---|---|---|
| 4.3 | Dataset after preprocessing | 34 |
| 4.4 | Model Architecture Used | 36 |
| 4.4 | Encoder Architecture of the model | 37 |
| 4.5 | Decoder Architecture of the model | 38 |
| 4.6 | Categorical Cross Entropy | 39 |

# LIST OF TABLES


| Table No. | Description | Page No. |
|---|---|---|
| 5.1 | An example of Word Error Rate | 41 |
| 5.2 | An Example of Character Error Rate | 42 |
| 6.1 | Word Error Rate Accuracy | 43 |
| 6.2 | Character Error Rate Accuracy | 44 |

# 1. INTRODUCTION

## 1.1 What is Transliteration

Transliteration is a problem in natural language processing for transforming the text in one script into another script so as to preserve the phonetic structure of words as closely as possible. It is useful when a user knows a language but does not know how to write its script. Machine transliteration can play an important role in natural language processing applications such as information retrieval and machine translation, especially for handling proper nouns and technical terms, cross-language applications, data mining and information retrieval systems.

Transliteration is the conversion of a word from one language to another without losing its phonological characteristics. It is the practice of transcribing a word or text written in one writing system into another writing system. For instance, the English word school would be transliterated to the Hindi word कूल. Note that this is different from translation in which
the word school would map to पाठशाला (’paathshaala’).

Hence, transliteration can be understood as the process of entering data in one language using the script of another language. In general, the mapping between the alphabet of one language and the other in a transliteration scheme will be as close as possible to the pronunciation of the word. Transliteration is a grapheme-based conversion method which converts every grapheme of source language to its equivalent in target language.

Text transliteration has its use cases in many applications. For the task of identification of text on sign boards, transliteration can be effectively used. Since the road names, city names, organisation names, shop names etc have the same pronunciation in every language, transliteration can be used to bridge the gap between the two languages.

With increasing globalisation and the rapid growth of the web, a lot of information is available today. However, most of this information is present in a select number of languages. Effective knowledge transfer across linguistic groups requires bringing down language barriers. Automatic name transliteration plays an important role in

many cross- language applications. For instance, cross-lingual information retrieval involves keyword translation from the source to the target language followed by document translation in the opposite direction. Proper names are frequent targets in such queries. Contemporary lexicon-based techniques fall short as translation dictionaries can never be complete for proper nouns. This is because new words appear almost daily and they become unregistered vocabulary in the lexicon.

Text transliteration is extremely useful when a non-local person who has no prior knowledge of the local language, wants to convey his speech to a large gathering in their local language, he can use transliteration to read the other language words using the script written in his language. For example, When Prime Minister of India visits China, in order to speak Chinese, he is given a script written in Hindi but the actual meaning of the script is Chinese. So, both parties can communicate with each other in their own comfortable languages.

## 1.2 Translation v/s Transliteration

### 1.2.1 Defining Translation

Translation is taking the meaning of a word from the source text and providing an equivalent text in the target language. Many times when you use an online translator, what you are really getting is transliteration or transcription and not a true translation. That is why the results of the “translation” often do not make sense.

It should be noted that there are levels of translation. A simple word by word translation takes text from one language and changes it into a word with an identical meaning in the other language. This is great unless there are word strings or sentences in which case a word for word translation is not sufficient. For example, the English phrase “under the weather” is used to denote that somebody has fallen ill. The word-to-word Spanish translation for the same would be “por debajoel clima”, which does not render any meaning in this specific context and the translated message would make no sense at all in the target language. It did not consider sentence structure, grammar, cultural issues, context, or overall meaning.

To be effective, a translation must take the meaning behind the text and put it into the target language so that the intent of the message remains intact. In the dead meat example, an effective translation might be “sentirse enfermo” or “feeling ill”.

A comprehensive translation will consider the grammar, syntax, and local culture during the conversion process so that the final document reads as if a native speaker wrote it.

### 1.2.2 Defining Transliteration

Transliteration involves changing the script used to write words in one language to the script of another; taking the letters or characters from a word and changing them into the equivalent characters in another language. This process is concerned with the spelling and not the sound. The sound of the words is handled through yet another "trans" – transcription. When there is a word you don't want to be changed, interpreted, or explained, but only put into the characters of another language, your translator will use their transliteration skills. This is often the case when it comes to names, addresses and other such material. When you think about Japanese words written in Latin lettering, you are thinking about transliteration.

The challenge of transliteration comes when there is not an equivalent character as often happens in Chinese or Japanese. The translator will need to approximate the character and this can lead to several translators spelling the same word in different ways.

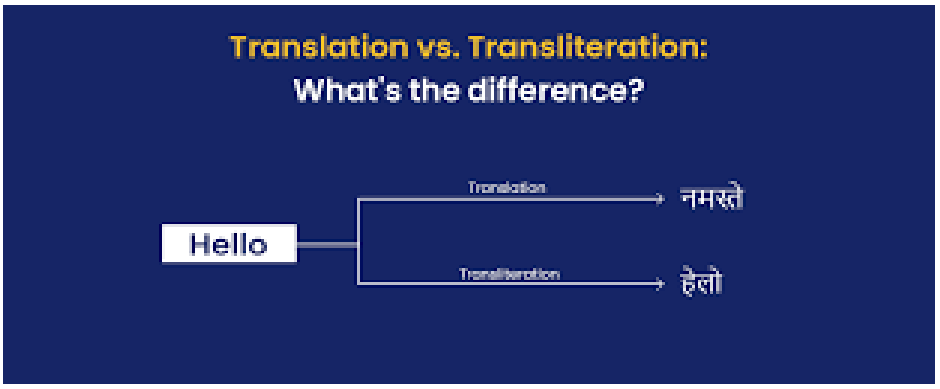


Fig. 1.1 Transliteration v/s Translation

## 1.3 Challenges Faced In Transliteration

A source language word can have more than one valid transliteration in the target language. For example, for the Hindi word below four different transliterations are possible:

गौतम - gautam, gotam, gautum, etc.

Therefore, in a CLIR context, it becomes important to generate all possible transliterations to retrieve documents containing any of the given forms.

Transliteration is not trivial to automate, but we will also be concerned with an even more challenging problem going from English back to Hindi, i.e., back-transliteration. Transforming target language approximations back into their original source language is called back-transliteration. The information-losing aspect of transliteration makes it hard to invert.

Back-transliteration is less forgiving than transliteration. There are many ways to write a Hindi word like मीनाक्षी (meenakshi, meenaxi, minakshi, minaakshi), all equally valid, but we do not have this flexibility in the reverse direction.

Besides this, specifically for Devanagari to Roman Transliteration, we run into the problem of the concept of "half-tokens" in the Devanagari script. To properly transliterate words like कच्चा, कप्तान and चम्मच, we need not only to convert to Roman token by token but at every point of time, we also need to look back at the transliterated tokens of the several previous tokens to be able to properly convert the word to Roman.

## 1.4 Applications Of Transliteration

- With the help of transliteration, we can convert proper nouns from one script to another. e.g. We wouldn't write ताज महल as Crown Palace, rather Taj Mahal. This is an important problem to consider while translating text, as the proper nouns must always be transliterated.

- Transliteration helps in defining pronunciation of words for non-native speakers of a language. We can see its application when we ask Google to give us the pronunciation of a word in a native language. It breaks the word into English tokens and presents the results in Roman. Similarly, for our specific use-cases, Devanagari to Roman Transliteration helps to render pronunciation of words written in the Devanagari script in Roman.

- While integrating Google Maps or Spotify APIs with chatbots in, e.g., Hindi, Tamil, Arabic, Chinese or Greek, you know they rarely produce acceptable results. With transliteration, you can extract entities like names, addresses, songs, etc. in your local language and convert them into the Latin (or English) alphabet to fully utilise APIs from Google Maps, Spotify and others.

- In content creation, something as simple as typing can be extremely challenging when we talk about languages spoken in the Middle East, Africa, India and South-East Asia. With governments and financial institutions mandating inclusion and content creators being more used to the Latin letter keypad, a tool like Transliteration can be very handy to make it easy to create local language content.
- Using Roman script as the derived script ensures that the transliterated texts can be read and understood by the majority as it is a universal script

## 1.5 Introduction To Deep Learning in Transliteration

Machine learning (ML) is the scientific study of algorithms and statistical models that computer systems use to perform a specific task without using explicit instructions, relying on patterns and inference instead. It is seen as a subset of artificial intelligence. Machine learning algorithms build a mathematical model based on sample data, known as "training data", in order to make predictions or decisions without being explicitly programmed to perform the task. Machine learning is closely related to computational statistics, which focuses on making predictions using computers. The study of mathematical optimization delivers methods, theory and application domains to the field of machine learning. "A computer program is said to learn from experience E with respect to some class of tasks T and performance measure P if its performance at tasks in T, as measured by P, improves with experience E.” This is Alan Turing’s definition of machine learning.

Deep learning is a class of machine learning algorithms that utilises a hierarchical level of artificial neural networks to carry out the process of machine learning. The artificial neural networks are built like the human brain, with neuron nodes connected together like a web. While traditional programs build analysis with data in a linear way, the hierarchical function of deep learning systems enables machines to process data with a nonlinear approach.

The word "deep" in "deep learning" refers to the number of layers through which the data is transformed. More precisely, deep learning systems have a substantial credit assignment path (CAP) depth. The CAP is the chain of transformations from input to output. CAPs describe potentially causal connections between input and output.

For a feedforward neural network, the depth of the CAPs is that of the network and is the number of hidden layers plus one (as the output layer is also parameterized). For recurrent neural networks, in which a signal may propagate through a layer more than once, the CAP depth is potentially unlimited.

Natural language processing (NLP) is a subfield of linguistics, computer science, information engineering, and artificial intelligence concerned with the interactions between computers and human (natural) languages, in particular how to program computers to process and analyse large amounts of natural language data.

Deep learning architectures such as deep neural networks, deep belief networks, recurrent neural networks and convolutional neural networks have been applied to fields including computer vision, speech recognition, natural language processing, audio recognition, social network filtering, machine translation, bioinformatics, drug design, medical image analysis, material inspection and board game programs, where they have produced results comparable to and in some cases superior to human experts.

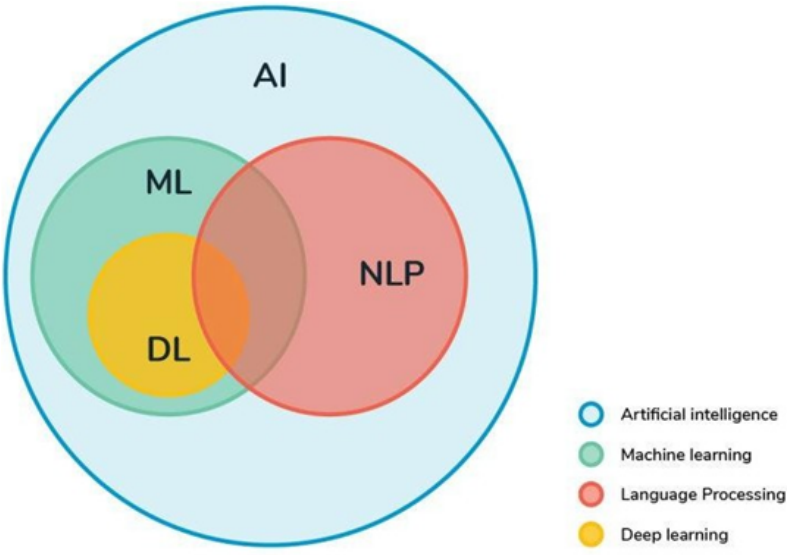


Fig. 1.2 Graphical representation of relationship between various fields in artificial intelligence

# 2. REVIEW OF LITERATURE

Deep learning systems are pragmatic for building a scalable pipeline, as they are completely data driven. Akshat Joshi, Kinal Mehta, Neha Gupta, Varun Kannadi Valloli experimented with Sequence to Sequence models to find an optimal model for the scalable pipeline by comparing the results. The results show Sequence to Sequence models are a better fit for this solution. They also discussed techniques for pre-processing the data and post processing the output for optimal performance.

P H Rathod, M L Dhore and RM Dhore have proposed the named entity transliteration for Hindi to English and Marathi to English language pairs using Support Vector Machine (SVM). In their proposed approach, the source named entity is segmented into transliteration units; hence the transliteration problem can be viewed as a sequence labelling problem. The classification of phonetic units is done by using the polynomial kernel function of Support Vector Machine (SVM). Their proposed approach uses phonetics of the source language and n-gram as two features for transliteration.

Sanjanashree and Anand Kumar presented a framework for bilingual machine transliteration for English and Tamil based on deep learning. The system uses Deep belief Network (DBN) which is a generative graphical model. The transliteration process consists of three steps viz. Preprocessing, Training using DBN and testing. The preprocessing phase does the Romanization of Tamil words. The data in both languages is converted to sparse binary matrices. Character padding is done at the end of every word to maintain the length of the words constant while encoding as sparse binary matrices. Deep Belief Network is a generative graphical model made up of multiple layers of Restricted Boltzmann Machine, a kind of Random Markov Field and Boltzmann Machine.

Mathur and Saxena have developed a system for EnglishHindi named entity transliteration using hybrid approach. The system first processes English words to extract phonemes using rules. After that statistical approach converts the English phoneme to equivalentHindi phoneme. The authors have used Stanford's NER for name entity extraction and extracted 42,371 name entities. Rules were applied to these entities and phonemes were extracted.
These English phonemes were transliterated to Hindi and a knowledgebase of English-Hindi phonemes was created.

Dhore et al. proposed Hindi to English transliteration of Named entities using Conditional random Fields. Indian places names are taken as input in Hindi language using Devanagari script by the system and transliterated into English. The input is provided in the form of syllabification in order to apply the n-gram techniques. This syllabification retains the phonemic features of the source language Hindi into transliterated form of English. The aim is to generate transliteration of a named entity given in Hindi into English using CRF as a statistical probability tool and n-gram as a feature set. The proposed system was tested using a bilingual corpus of 7251 named entities created from web resources and books. The commonly used performance evaluation parameter was "word accuracy?. The system has received very good accuracy of 85.79% for the bi-grams of source language Hindi

Lehal and Saini have developed "Sangam: A Perso-Arabic to Indic Script Machine Transliteration Model". Sangam is a hybrid system which combines rules as well as word and character level language models to transliterate the words. The system has been successfully tested on Punjabi, Urdu and Sindhi languages and can be easily extended for other languages likeKashmiri andKonkani. The transliteration accuracy for the three scripts ranges from 91.68% to 97.75%, which is the best accuracy reported so far in literature for script pairs in Perso-Arabic and Indic script

Deep and Goyal have developed a Rule based Punjabi to English transliteration system for common names. The proposed system works by employing a set of character sequence mapping rules between the languages involved.To improve accuracy, the rules are developed with specific constraints. This system was trained using 1013 preson's names and tested using different person names, city names, river names etc. The system has reported the overall accuracy of 93.22%

# 3. THEORETICAL BACKGROUND

## 3.1 Neural Networks

A neural network is a network or circuit of neurons, or in a modern sense, an artificial neural network, composed of artificial neurons or nodes. Thus, a neural network is either a biological neural network, made up of real biological neurons, or an artificial neural network, for solving artificial intelligence (AI) problems. The connections of the biological neuron are modelled as weights. A positive weight reflects an excitatory connection, while negative values mean inhibitory connections. All inputs are modified by a weight and summed. This activity is referred to as a linear combination. Finally, an activation function controls the amplitude of the output. For example, an acceptable range of output is usually between 0 and 1, or it could be -1 and 1.

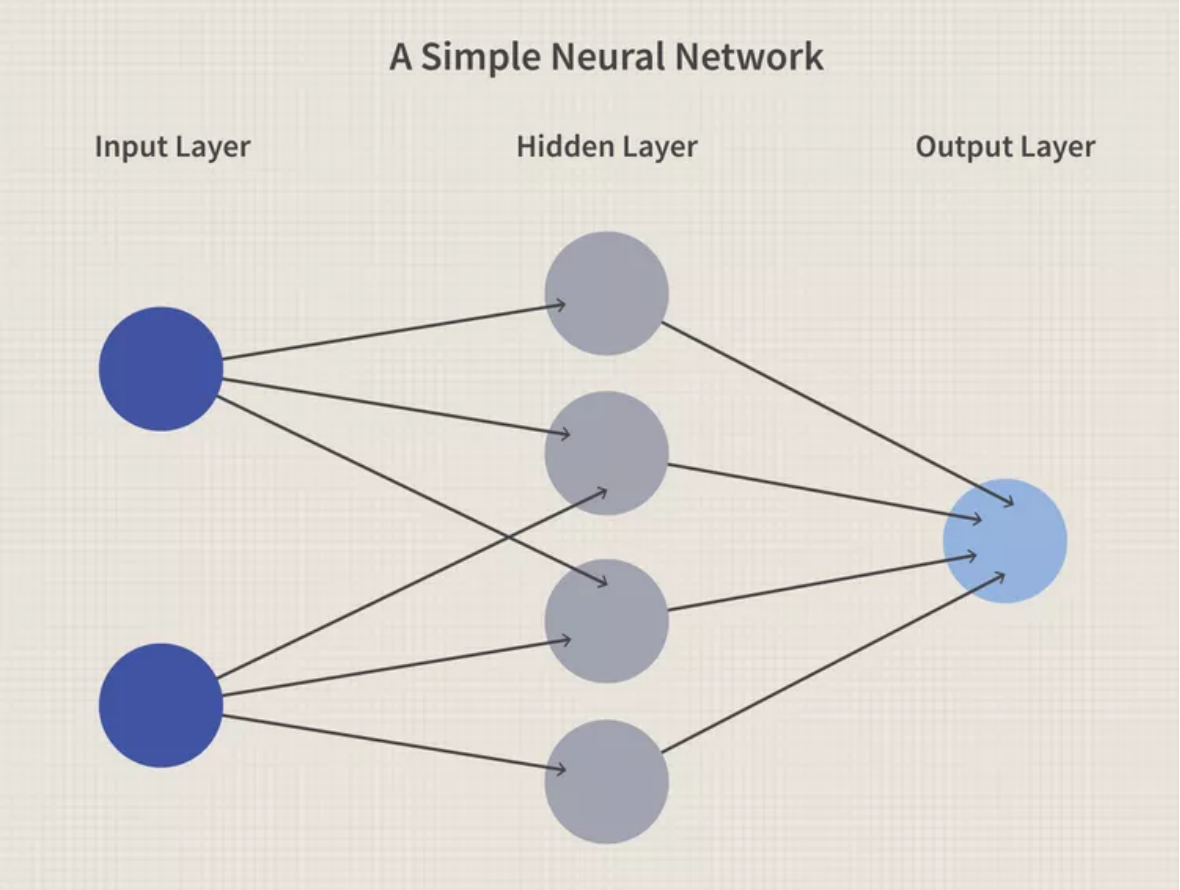


Fig. 3.1 A Simple Neural Network

These artificial networks may be used for predictive modelling, adaptive control and applications where they can be trained via a dataset. Self-learning resulting from experience can occur within networks, which can derive conclusions from a complex and seemingly unrelated set of information.

A biological neural network is composed of a group or groups of chemically connected or functionally associated neurons. A single neuron may be connected to many other neurons and the total number of neurons and connections in a network may be extensive. Connections, called synapses, are usually formed from axons to dendrites, though dendrodendritic synapses and other connections are possible. Apart from the electrical signalling, there are other forms of signalling that arise from neurotransmitter diffusion.

Artificial intelligence, cognitive modelling, and neural networks are information processing paradigms inspired by the way biological neural systems process data. Artificial intelligence and cognitive modelling try to simulate some properties of biological neural networks. In the artificial intelligence field, artificial neural networks have been applied successfully to speech recognition, image analysis and adaptive control, in order to construct software agents (in computer and video games) or autonomous robots.

## 3.2 Neuron

The basic unit of computation in a neural network is the neuron, often called a node or unit. It receives input from some other nodes, or from an external source and computes an output. Each input has an associated weight (w), which is assigned on the basis of its relative importance to other inputs. The node applies a function f (defined below) to the weighted sum of its inputs as in figure below.

An artificial neuron is a connection point in an artificial neural network. Artificial neural networks, like the human body's biological neural network, have a layered architecture and each network node (connection point) has the capability to process input and forward output to other nodes in the network. In both artificial and biological architectures, the nodes are called neurons and the connections are characterised by synaptic weights, which represent the significance of the connection.

As new data is received and processed, the synaptic weights change and this is how learning occurs.

Artificial neurons are modelled after the hierarchical arrangement of neurons in biological sensory systems. In the visual system, for example, light input passes through neurons in successive layers of the retina before being passed to neurons in the thalamus of the brain and then on to neurons in the brain's visual cortex. As the neurons pass signals through an increasing number of layers, the brain progressively extracts more information until it is confident it can identify what the person is seeing. In artificial intelligence, this fine tuning process is known as deep learning.
In both artificial and biological networks, when neurons process the input they receive, they decide whether the output should be passed on to the next layer as input. The decision of whether or not to send information on is called bias and it's determined by an activation function built into the system. For example, an artificial neuron may only pass an output signal on to the next layer if its inputs (which are actually voltages) sum to a value above some particular threshold value. Because activation functions can either be linear or nonlinear, neurons will often have a wide range of convergence and divergence. Divergence is the ability for one neuron to communicate with many other neurons in the network and convergence is the ability for one neuron to receive input from many other neurons in the network.

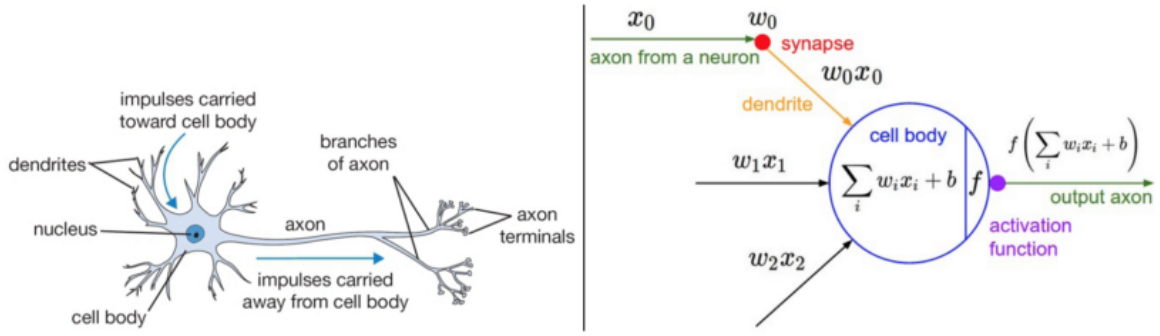


Fig. 3.2 A Biological Neuron (left) & An Artificial Neuron (right)

# 3.3 Propagation (How Neural Networks Work)

## 3.3.1 Weights and Biases

Weight is the parameter within a neural network that transforms input data within the network's hidden layers. A neural network is a series of nodes, or neurons. Within each node is a set of inputs, weight, and a bias value. As an input enters the node, it gets multiplied by a weight value and the resulting output is either observed,

or passed to the next layer in the neural network. Often the weights of a neural network are contained within the hidden layers of the network. Within a neural network there's an input layer that takes the input signals and passes them to the next layer. Next, the neural network contains a series of hidden layers which apply transformations to the input data. It is within the nodes of the hidden layers that the weights are applied. For example, a single node may take the input data and multiply it by an assigned weight value, then add a bias before passing the data to the next layer. The final layer of the neural network is also known as the output layer. The output layer often tunes the inputs from the hidden layers to produce the desired numbers in a specified range.

Weights and bias are both learnable parameters inside the network. A teachable neural network will randomise both the weight and bias values before learning initially begins. As training continues, both parameters are adjusted toward the desired values and the correct output. The two parameters differ in the extent of their influence upon the input data. Simply, bias represents how far off the predictions are from their intended value. Biases make up the difference between the function's output and its intended output. A low bias suggests that the network is making more assumptions about the form of the output, whereas a high bias value makes less assumptions about the form of the output. Weights, on the other hand, can be thought of as the strength of the connection. Weight affects the amount of influence a change in the input will have upon the output. A low weight value will have no change on the input, and alternatively a larger weight value will more significantly change the output.

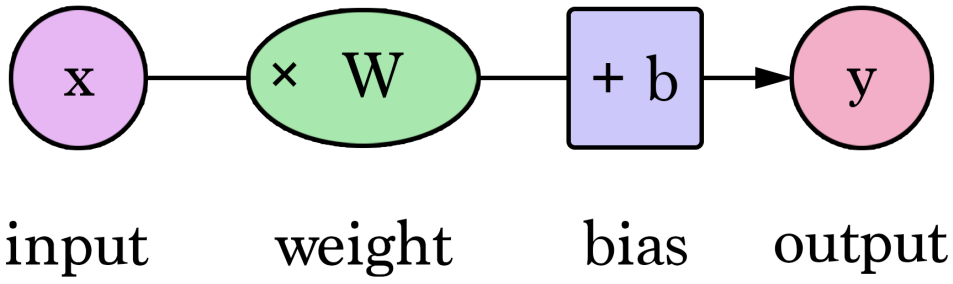


Fig. 3.3 Weights and Biases

### 3.3.2 Activation Function

An Activation Function decides whether a neuron should be activated or not. This means that it will decide whether the neuron's input to the network is important or not in the process of prediction using simpler mathematical operations. The role of the Activation Function is to derive output from a set of input values fed to a node (or a layer).

The primary role of the Activation Function is to transform the summed weighted input from the node into an output value to be fed to the next hidden layer or as output. Examples of activation functions include Binary Step Activation Function, Linear Activation Function and Non-Linear Activation Functions like Sigmoid Activation Function and Tanh Activation Function.

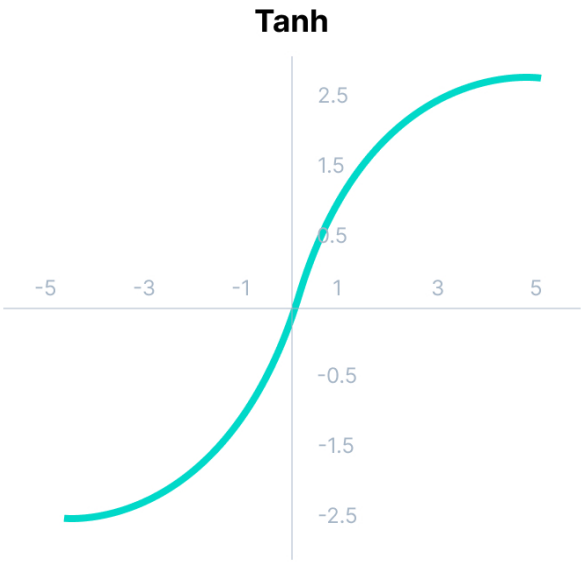


Fig. 3.4 Tanh Function (Hyperbolic Tangent)

## 3.4 Feed Forward Neural Network

A Feed Forward Neural Network is an artificial neural network in which the connections between nodes do not form a cycle. The opposite of a feed forward neural network is a recurrent neural network, in which certain pathways are cycled. The feed forward model is the simplest form of neural network as information is only processed in one direction. While the data may pass through multiple hidden nodes, it always moves in one direction and never backwards.

A feedforward neural network is a type of artificial neural network in which nodes' connections do not form a loop. Often referred to as a multi-layered network of neurons, feedforward neural networks are so named because all information flows in a forward manner only. The data enters the input nodes, travels through the hidden layers, and eventually exits the output nodes. The network is devoid of links that would allow the information exiting the output node to be sent back into the network. The purpose of feedforward neural networks is to approximate functions.

### 3.4.1 How does a Feed Forward Neural Network work?

A Feed Forward Neural Network is commonly seen in its simplest form as a single layer perceptron. In this model, a series of inputs enter the layer and are multiplied by the weights. Each value is then added together to get a sum of the weighted input values. If the sum of the values is above a specific threshold, usually set at zero, the value produced is often 1, whereas if the sum falls below the threshold, the output value is -1. The single layer perceptron is an important model of feed forward neural networks and is often used in classification tasks. Furthermore, single layer perceptrons can incorporate aspects of machine learning. Using a property known as the delta rule, the neural network can compare the outputs of its nodes with the intended values, thus allowing the network to adjust its weights through training in order to produce more accurate output values. This process of training and learning produces a form of a gradient descent. In multi-layered perceptrons, the process of updating weights is nearly analogous, however the process is defined more specifically as back-propagation. In such cases, each hidden layer within the network is adjusted according to the output values produced by the final layer.

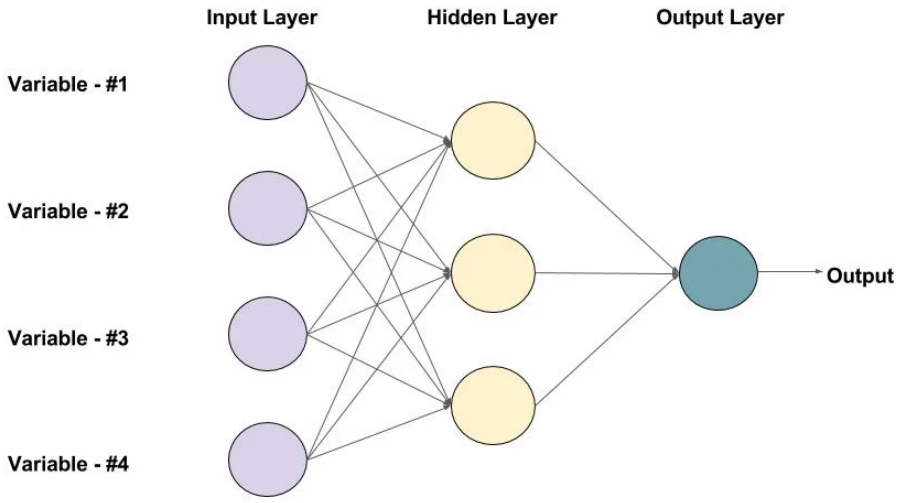


Fig. 3.5 A Feed Forward Neural Network with one hidden layer (and 3 neurons)

### 3.4.2 Applications of Feed Forward Neural Networks

While Feed Forward Neural Networks are fairly straightforward, their simplified architecture can be used as an advantage in particular machine learning applications. For example, one may set up a series of feed forward neural networks with the intention of running them independently from each other, but with a mild intermediary for moderation. Like the human brain, this process relies on many individual neurons in order to handle and process larger tasks. As the individual networks perform their tasks independently, the results can be combined at the end to produce a synthesised, and cohesive output.

## 3.5 Backpropagation In Neural Networks

Backpropagation is the essence of neural network training. It is the method of fine-tuning the weights of a neural network based on the error rate obtained in the previous epoch (i.e., iteration). Proper tuning of the weights allows you to reduce error rates and make the model reliable by increasing its generalisation.

Backpropagation in neural networks is a short form for “backward propagation of errors.” It is a standard method of training artificial neural networks. This method helps calculate the gradient of a loss function with respect to all the weights in the network.

The Back propagation algorithm in neural networks computes the gradient of the loss function for a single weight by the chain rule. It efficiently computes one layer at a time, unlike a native direct computation. It computes the gradient, but it does not define how the gradient is used. It generalises the computation in the delta rule.

Most prominent advantages of Backpropagation are:

- Backpropagation is fast, simple and easy to program
- It has no parameters to tune apart from the numbers of input
- It is a flexible method as it does not require prior knowledge about the network
- It is a standard method that generally works well

- It does not need any special mention of the features of the function to be learned.

Two Types of Backpropagation Networks are:

- Static Backpropagation
- Recurrent Backpropagation

### 3.5.1 Static Backpropagation

It is one kind of backpropagation network which produces a mapping of a static input for static output. It is useful to solve static classification issues like optical character recognition.

### 3.5.2 Recurrent Backpropagation

Recurrent Back propagation in data mining is fed forward until a fixed value is achieved. After that, the error is computed and propagated backward.
The main difference between both of these methods is: that the mapping is rapid in static back-propagation while it is nonstatic in recurrent backpropagation.

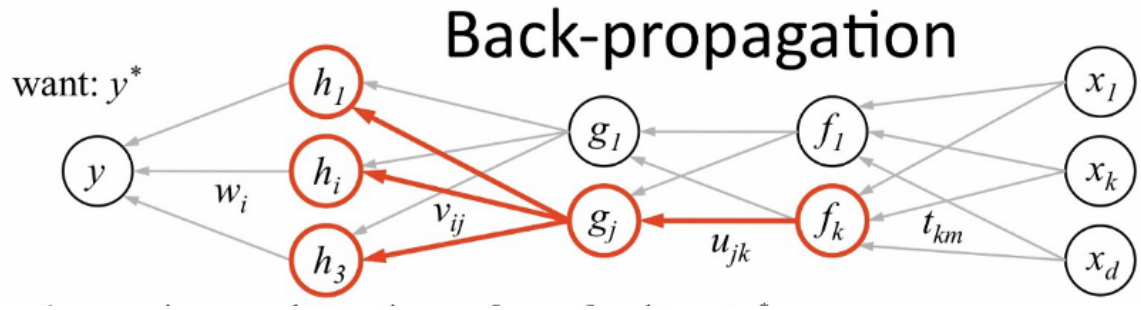


Fig. 3.6 Backpropagation in Neural Networks

## 3.6 Recurrent Neural Networks

### 3.6.1 Introduction To RNN

A Deep Learning approach for modelling sequential data is Recurrent Neural Networks (RNN). RNNs were the standard suggestion for working with sequential data before the advent of attention models. Specific parameters for each element of

the sequence may be required by a deep feedforward model. It may also be unable to generalise to variable-length sequences.

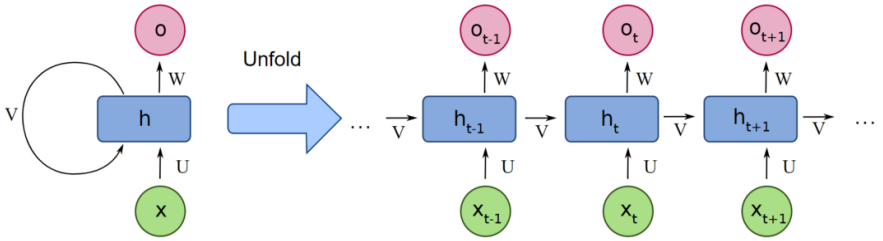


Fig. 3.7 Recurrent Neural Networks

Recurrent Neural Networks use the same weights for each element of the sequence, decreasing the number of parameters and allowing the model to generalise to sequences of varying lengths. RNNs generalise to structured data other than sequential data, such as geographical or graphical data, because of its design.

Recurrent neural networks, like many other deep learning techniques, are relatively old. They were first developed in the 1980s, but we didn't appreciate their full potential until recently.

RNNs are a type of neural network that can be used to model sequence data. RNNs, which are formed from feedforward networks, are similar to human brains in their behaviour. Simply said, recurrent neural networks can anticipate sequential data in a way that other algorithms can't.

All of the inputs and outputs in standard neural networks are independent of one another, however in some circumstances, such as when predicting the next word of a phrase, the prior words are necessary, and so the previous words must be remembered. As a result, RNN was created, which used a Hidden Layer to overcome the problem. The most important component of RNN is the Hidden state, which remembers specific information about a sequence.

### 3.6.2 Difference between Feed Forward Neural Networks and RNN

Feed-forward neural networks have no memory of the input they receive and are bad at predicting what's coming next. Because a feed-forward network only considers the current input, it has no notion of order in time. It simply can't remember

anything about what happened in the past except its training. In a RNN the information cycles through a loop. When it makes a decision, it considers the current input and also what it has learned from the inputs it received previously.

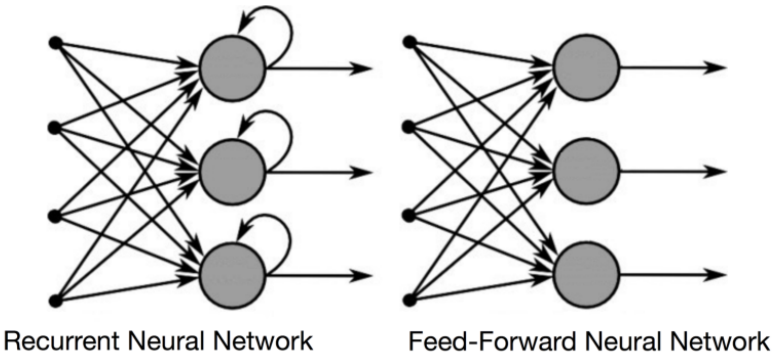


Fig. 3.8 Recurrent Neural Network v/s Feed-Forward Neural Network

## 3.7 Long Short Term Memory (LSTM)

Long Short Term Memory networks – usually just called “LSTMs” – are a special kind of RNN, capable of learning long-term dependencies. They were introduced by Hochreiter & Schmidhuber (1997), and were refined and popularised by many people in following work.1 They work tremendously well on a large variety of problems, and are now widely used.

LSTMs are explicitly designed to avoid the long-term dependency problem. Remembering information for long periods of time is practically their default behaviour, not something they struggle to learn!

All recurrent neural networks have the form of a chain of repeating modules of neural network. In standard RNNs, this repeating module will have a very simple structure, such as a single tanh layer.

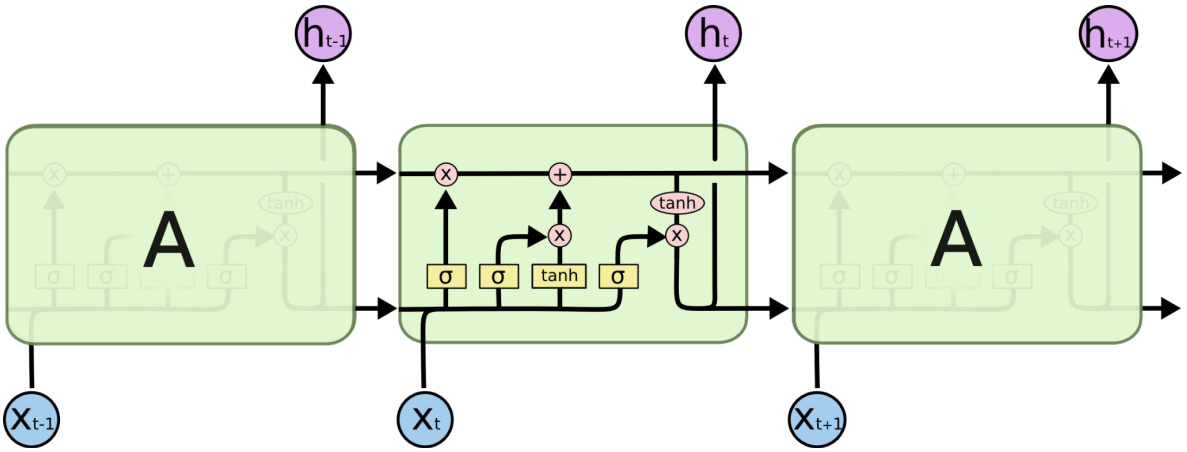


Fig. 3.9 A Standard LSTM Architecture

The key to LSTMs is the cell state, the horizontal line running through the top of the diagram. The cell state is kind of like a conveyor belt. It runs straight down the entire chain, with only some minor linear interactions. It's very easy for information to just flow along it unchanged.

The LSTM does have the ability to remove or add information to the cell state, carefully regulated by structures called gates. Gates are a way to optionally let information through. They are composed out of a sigmoid neural net layer and a pointwise multiplication operation.

The sigmoid layer outputs numbers between zero and one, describing how much of each component should be let through. A value of zero means "let nothing through," while a value of one means "let everything through!"

An LSTM has three of these gates, to protect and control the cell state.

## 3.8 Encoder-Decoder Architecture

In order to fully understand the model's underlying logic, we will go over the below illustration:

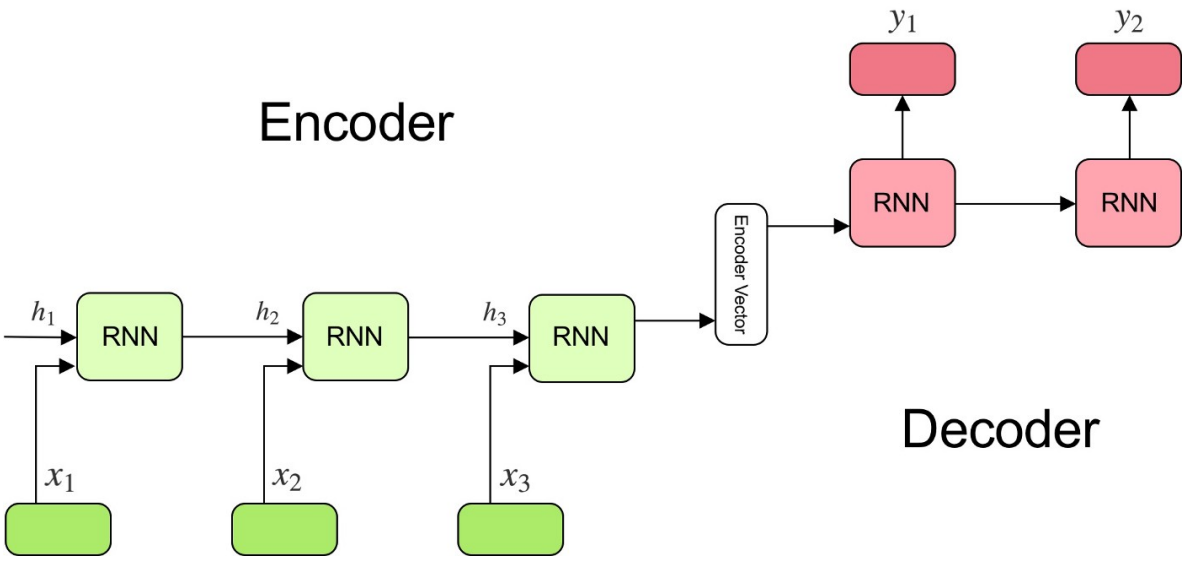


Fig. 3.10 Encoder-Decoder Architecture

### 3.8.1 Encoder

It is a stack of several recurrent units (LSTM or GRU cells for better performance) where each accepts a single element of the input sequence, collects information for that element and propagates it forward. In question-answering problem, the input sequence is a collection of all words from the question. Each word is represented as x_i where i is the order of that word. The hidden states h_i are computed using the formula:

$$h_t = f(W^{(hh)} h_{t-1} + W^{(hx)} x_t)$$

This simple formula represents the result of an ordinary recurrent neural network. As you can see, we just apply the appropriate weights to the previous hidden state h_(t-1) and the input vector x_t.

### 3.8.2 Encoder Vector

This is the final hidden state produced from the encoder part of the model. It is calculated using the formula above. This vector aims to encapsulate the information

for all input elements in order to help the decoder make accurate predictions. It acts as the initial hidden state of the decoder part of the model.

### 3.8.3 Decoder

A stack of several recurrent units where each predicts an output y_t at a time step t.
Each recurrent unit accepts a hidden state from the previous unit and produces an output as well as its own hidden state. In the question-answering problem, the output sequence is a collection of all words from the answer. Each word is represented as y_i where i is the order of that word. Any hidden state h_i is computed using the formula:

$$h_t = f(W^{(hh)} h_{t-1})$$

As you can see, we are just using the previous hidden state to compute the next one. The output y_t at time step t is computed using the formula:

$$y_t = softmax(W^S h_t)$$

We calculate the outputs using the hidden state at the current time step together with the respective weight W(S). Softmax is used to create a probability vector which will help us determine the final output (e.g. word in the question-answering problem). The power of this model lies in the fact that it can map sequences of different lengths to each other. As you can see the inputs and outputs are not correlated and their lengths can differ. This opens a whole new range of problems which can now be solved using such architecture.

# 4. PROPOSED METHODOLOGY

The primary motivation behind taking up this project for our major thesis work was that with the rise in popularity of sequence-to-sequence models, which seemed to be performing quite well in machine translation in recent years, and the new methodologies and latest research work in the field of sequence-to-sequence models, we planned to implement the same in the field of transliteration. In case of Devanagari to Roman Transliteration, we need a method which can not only convert individual Hindi tokens to English, rather we need the output produced in the earlier steps to factor into the output generation of the current token. Sequence-to-sequence architecture allows us to take into consideration all the niche aspects of the Devanagari script like the concepts of “half-tokens”, “bindu’s”, “chandra-bindu’s”, etc.

To understand the methodologies we used and how they affected their respective outputs, we first need to discuss in brief about the evaluation metrics used for the results (which we will discuss in detail about in the Results section). The two evaluation metrics used are Word Error Rate and Character Error Rate. To compute Word Error Rate for the test data, we compare the output English word generated for each input Hindi word with the corresponding English word in the dataset. If the both words match completely, it is counted as a success, otherwise, the output is counted as a failure. On the other hand, to compute Character Error Rate, we take the ratio of the Longest Common Subsequence of the output English word and the given English word in the dataset with the maximum of the length of both those words. Generally, Character Error Rate is bound to be lesser than the Word Error Rate because even in case of words that could not completely match, it is still probable that at least some characters would still match and the Longest Common Subsequence will not be an empty string.

Our initial methodology was to use an Encoder-Decoder architecture without any memory cells to retain and improve from previously generated output. This method was good at predicting Hindi words with simple tokens without half-tokens or bindu’s. Eg. “मन”, “काम”, etc. i.e. “mann” and “kaam”, For such words, both the Character Error Rate and the Word Error Rate came out to be low (as expected). But for complex Hindi words, which contain half-tokens or bindu’s such as words like , this model performed poorly. Eg. for words like “त्वचा”, the corresponding word in English was “twacha”, while the output came out to be “tavacha”. The Character Error Rate generated was still relatively low, but the higher Word Error Rate forced us to switch to an Encoder-Decoder architecture with memory cells, like LSTM.

The sequence-to-sequence model used for our use case is the Encoder-Decoder model with LSTM which we will discuss in detail in this section. We will dive deep into the functioning and architecture of our model, discussing each of its layers, how LSTM played a major role in reducing not only the Character Error Rate, but also the Word Error Rate of the test outputs generated when compared with the dataset used. The Character Error Rate was significantly reduced and the Word Error Rate was also marginally reduced.

## 4.1 Basic Outline

The first step of this project was to find a relevant dataset that would contain word to word mapping from Devanagari (Hindi) to Roman (English). There were other datasets available that had the same mappings available but in a sentence-to-sentence or paragraph-to-paragraph format but the word-to-word approach was more suitable for our use-case and was faster to train over a neural network. This dataset was then pre-processed to first remove any unwanted characters and null values. In the later stages, the rows containing duplicate values were also dropped off. Start of word and End of word tags were also applied to input data to let the Encoder process it properly.

The input was encoded based on the number of characters in the longest word in the Devanagari column and the number of unique characters in the Devanagari script. The shape of the input to the Encoder was (1, 18, 66) where 66 is the number of unique characters in the Devanagari script and 18 is the maximum length of a Hindi word in the dataset. The shape of the input fed to the decoder was (1, 23, 28), where 23 is the maximum length of an English word in the dataset and 28 is the number of unique characters in the Roman script (26 + 2 characters for denoting Start of word and End of word characters).

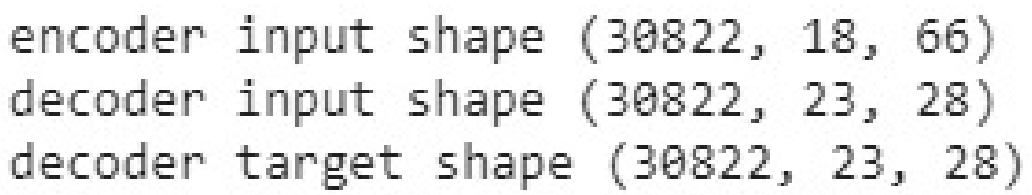


Fig. 4.1 Shapes of input to the Encoder-Decoder Architecture

The layers used in the model are Embedding Layer, Flatten Layer (Reshape Layer), Dense Layer, LSTM, etc, and all of them helped us to further reduce the Word Error Rate and Character Error Rate. We will discuss these layers in detail in the upcoming sections.

For the purpose of training, we found the suitable batch size, learning rate and number of epochs by trial and error methods, taking some help from our Supervisor and through research papers. We plotted the accuracy and loss functions on a graph to analyse our model.

Next step was inference. Here, we fed the states obtained from the trained encoder and an empty string to the decoder. The output obtained from this step was recurrently fed into an LSTM network to generate the final string character by character, with the LSTM network taking care of the memory of the previously generated characters into the generation of new characters.

## 4.2 Dataset Used

We needed a dataset that would contain mapping of words from the Devanagari script to Roman script. We explored several datasets, some having this mapping in a sentence wise manner, some having it on a paragraph level, while some being word-to-word mapped. We figured that for our use-case, the word-to-word approach was more suitable and was faster to train over a neural network. The dataset finally selected was a two dimensional dataset containing words in Devanagari script in the first column, and the direct mapping of those words in Roman script in the second column. The mapping in the dataset is merely word-to-word based. The model was trained to obtain a character-wise mapping as well. The size of the dataset is 30822 rows and 2 columns (30822 X 2). A snapshot of the dataset is attached below which shows how the dataset looked initially in its raw form before we had done any pre-processing on it.

| | hindi | english |
|---|---|---|
| 0 | खुशबू | khushboo |
| 1 | खुशबू | khushbuu |
| 2 | खुशबू | khushbu |
| 3 | खुशबू | khusbhu |
| 4 | तेरा | tera |

Fig 4.2 The dataset used in its raw form

### 4.2.1 Preprocessing the dataset

The first step in preprocessing involved removing all the rows with NaN values. Then, the white spaces were removed from each of the words in the dataset so that it does not become a hindrance for the model as otherwise, white spaces would have to be considered in vectorisation. Then, such rows were removed that contained obsolete characters from the Devanagari script or such rows that had any extra characters in either of the columns. Then, we padded the Roman words with a Start of word and End of word token. The start symbol used was "^" and the end symbol used was "$".

However, in the later stages, after obtaining the initial round of results on the test data, it was clear that the model was getting confused with the presence of multiple occurrences of the same Devanagari words that were mapped differently in Roman in each row, so as a final pre-processing step, we had to further filter the dataset and had to remove all the repeating occurrences of words in the Devanagari script. The snapshot below shows how the dataset looked after all the pre-processing was done and it was ready for a final round of training.

| hindi | english |
|---|---|
| खुशबू | khusbhu |
| तेरा | teraaa |
| बदन | badan |
| सुलगे | sulge |
| महके | maheke |

Fig. 4.3 Dataset after preprocessing

## 4.3 Implementation Strategy

To understand the implementation strategy, let's discuss each part in Fig. 3.10 and discuss in detail about how the Encoder-Decoder Architecture is used in our project.

### 4.3.1 Encoder

The left hand side part of the figure shows an Encoder. Formally, an Encoder is defined as a stack of several recurrent units (LSTM or GRU cells for better performance) where each accepts a single element of the input sequence, collects information for that element and propagates it forward. In our project, we scanned the training set and found the maximum number of characters among both the columns containing Devanagari and Roman words, i.e, the number of characters in the longest words of both the columns respectively, say d_max and r_max respectively. The input to the Encoder was then encoded based on d_max and the number of unique characters in the Devanagari script. Therefore, the shape of the input used for the Encoder turned out to be (1, 18, 66) where 66 is the number of unique characters in the Devanagari script and d_max was equal to 18.

The final hidden state produced from the encoder part of the model is the Encoder Vector which is then. This vector aims to encapsulate the information for all input elements in order to help the decoder make accurate predictions. It acts as the initial hidden state of the decoder part of the model.

### 4.3.2 Decoder

The right hand side of the Fig. 3.10 shows a Decoder. Formally, a Decoder is defined as a stack of several recurrent units where each predicts an output y_t at a time step t. Each recurrent unit accepts a hidden state from the previous unit and produces an output as well as its own hidden state. We take vectors formed from r_max as calculated above and the number of unique characters in the Roman script and feed them as inputs to the Decoder. Thus, the shape of the input fed to the Decoder turned out to be (1, 23, 28) where 28 is the number of unique characters in the Roman script (26 + 2 characters for denoting Start of word and End of word characters) and r_max was equal to 28.

## 4.4 Model Architecture Used

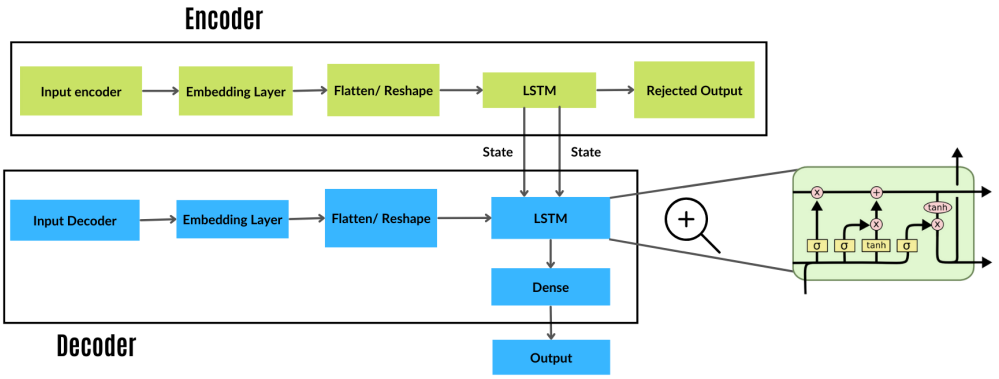


Fig. 4.4 Model Architecture Used

The input fed to the encoder was passed through an Input Layer. As stated earlier, the size of this input was (1, 18, 66) where 66 is the number of unique characters in the Devanagari script and 18 is the length of the longest word in the Devanagari column of the dataset.

The output from the Input Layer was passed to an Embedding Layer. Formally, the Embedding layer enables us to convert each word into a fixed length vector of defined size. The resultant vector is a dense one with real values instead of just 0's and 1's. The fixed length of word vectors helps us to represent words in a better way along with reduced dimensions. It is capable of understanding the context of words so that similar words have similar encodings. For our use-case however, the embedding layer was used to better understand the context of **characters** within these words so that similar characters have similar encodings for better lookup during the training phase. The shape of the output from the Embedding Layer (1, 18, 66, 100).

The output from the Embedding Layer was flattened with the help of Reshape Layer. Formally, the Reshape layer can be used to change the dimensions of its input, without changing its data. It changes only the dimensions of the input that is passed to it. Flattening of the input was necessary so that it can be processed by the LSTM network effectively. The shape of the output from the Reshape Layer (1, 18, 6600).

The output from the Reshape Layer was fed into a network of LSTM. Formally, an LSTM module has a cell state and three gates which provides them with the power to selectively learn, unlearn or retain information from each of the units. The cell state in

LSTM helps the information to flow through the units without being altered by allowing only a few linear interactions. LSTM generates one output and two states. The outputs obtained from LSTM are discarded and only the states are preserved.

The input given to Decoder is an empty string and the final states obtained from the Encoder. Using these initial states, the decoder starts generating the output sequence. The data (1, 23, 28) is then again passed through an Embedding Layer (1, 23, 28, 256), flattened with a Reshape Layer (1, 23, 7168). This output, along with two initial states from Encoder are passed onto an LSTM network. The output from the LSTM layer is recurrently fed (with new states generated from the Decoder itself) into the Decoder until we reach the end of word. Then the Dense Layer is used to produce the final output of shape (1, 23, 28).

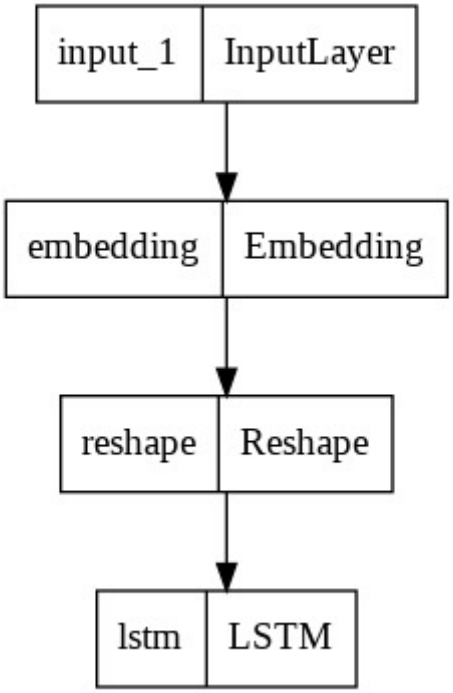


Fig. 4.5 Encoder Architecture of the model

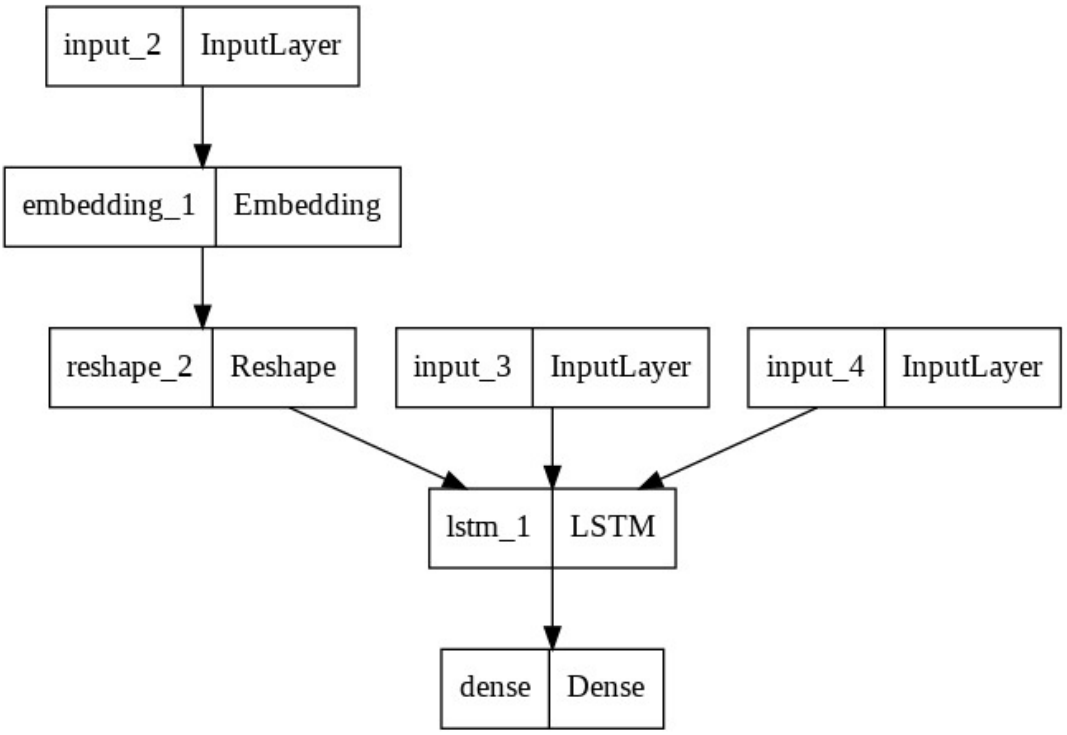


Fig. 4.6 Decoder Architecture of the model

## 4.5 Training

The training set was itself divided into a number of batches, each of batch size 64, where batch size signifies the number of training examples utilised in one iteration. It controls the accuracy of the estimate of the error gradient when training neural networks. The number of epochs was set to 500, where epochs signify the number of times a batch is passed forward and backward through the neural network only ONCE. The initial learning rate, denoted by the symbol α, is a tuning parameter in an optimization algorithm that determines the step size at each iteration while moving toward a minimum of a loss function. The initial learning rate in our case was set to 0.001.

An optimizer is a function or an algorithm that modifies the attributes of the neural network, such as weights and learning rate. Thus, it helps in reducing the overall loss and improving the accuracy. The optimiser that we used in our project is RMSprop optimiser. The RMSprop optimizer restricts the oscillations in the vertical direction. Therefore, we can increase our learning rate and our algorithm could take larger steps in the horizontal direction converging faster. The difference between RMSprop and gradient descent is on how the gradients are calculated. RMSProp is designed to

accelerate the optimization process, e.g. decrease the number of function evaluations required to reach the optima, or to improve the capability of the optimization algorithm, e.g. result in a better final result.

Loss is the penalty for a bad prediction. That is, loss is a number indicating how bad the model's prediction was on a single example. If the model's prediction is perfect, the loss is zero; otherwise, the loss is greater. Loss function is a method that evaluates how well the algorithm learns the data and produces correct outputs. It computes the distance between our predicted value and the actual value using a mathematical formula. A simple, and very common, example of a loss function is the squared-error loss, a type of loss function that increases quadratically with the difference, used in estimators like linear regression, calculation of unbiased statistics, and many areas of machine learning. The loss function used in our project is the Categorical Cross Entropy Function. Categorical Cross Entropy Loss, or log loss, measures the performance of the classification model whose output is a probability between 0 and 1. Cross entropy increases as the predicted probability of a sample diverges from the actual value. Therefore, predicting a probability of 0.05 when the actual label has a value of 1 increases the cross entropy loss. Cross-entropy is a measure from the field of information theory, building upon entropy and generally calculating the difference between two probability distributions. It is closely related to but is different from KL divergence that calculates the relative entropy between two probability distributions, whereas cross-entropy can be thought to calculate the total entropy between the distributions. Categorical Cross Entropy is also related to and often confused with logistic loss, called log loss. Although the two measures are derived from a different source, when used as loss functions for classification models, both measures calculate the same quantity and can be used interchangeably.

$$CCE(p,t) = -\sum_{c=1}^{C} t_{o,c} \log\left(p_{o,c}\right)$$

Fig. 4.7 Categorical Cross Entropy

We have also implemented Early Stopping in our project. Too many epochs can lead to overfitting of the training dataset, whereas too few may result in an underfit model. Early stopping is a method that allows you to specify an arbitrary large number of training epochs and stop training once the model performance stops improving on a hold out validation dataset. We can account for this by adding a delay to the trigger in terms of the number of epochs on which we would like to see no improvement. This

can be done by setting the “patience” argument. We set the patience argument to 20. The exact amount of patience will vary between models and problems. Reviewing plots of your performance measure can be very useful to get an idea of how noisy the optimization process for your model on your data may be.

Learning rate schedule is also implemented in our model for variable learning rate. It is a predefined framework that adjusts the learning rate between epochs or iterations as the training progresses. For the training process, this is good. Early in the training, the learning rate is set to be large in order to reach a set of weights that are good enough. Over time, these weights are fine-tuned to reach higher accuracy by leveraging a small learning rate. The mathematical form of time-based decay is lr = lr0/(1+kt) where lr, k are hyperparameters and t is the iteration number. Looking into the source code of Keras, the SGD optimizer takes decay and lr arguments and updates the learning rate by a decreasing factor in each epoch.

Learning Rate = (1 / (1 + self.decay * self.iterations))

All the above methods and concepts were utilised to train our model.

# 5. EVALUATION METRICS

The two evaluation metrics used are Word Error Rate and Character Error Rate.

## 5.1 Word Error Rate

Word Error Rate is a stricter metric as it compares two words and unless they match completely, i.e, the words are exactly the same and have no different characters among each other, it counts it as a failure. Only if the words match completely, it is counted as a success.

To compute Word Error Rate for the test data, we compare the output English word generated for each input Hindi word with the corresponding English word in the dataset. If the both words match completely, it is counted as a success, otherwise, the output is counted as a failure. Word Error Rate comes out to be minimal when the test data includes words containing simple and straight-forward Devanagari characters with no half-tokens or bindu's involved. But even in case of such words, the Word Error Rate is not too less because depending on the dataset used, words like "मन" can be transliterated to several Roman words like "man", "mann", "mun", etc, so it is not reasonable to expect a below par value for Word Error Rate as for words whose output don't match exactly with the corresponding Roman words, the model still performs well enough. Eg.

| Input | Predicted output | Target | flag |
|---|---|---|---|
| महल | mahal | mahal | True |
| हँसना | hansna | hasna | False |

Table 5.1 An example of Word Error Rate

## 5.2 Character Error Rate

Character Error is a more forgiving metric than the Word Error Rate as it is merely a measure of the similarities between two words with respect to their Longest Common Subsequence (LCS). Before we discuss Character Error Rate in detail, let's first briefly describe LCS.

### 5.2.1 Longest Common Subsequence (LCS)

The longest common subsequence (LCS) is defined as the longest subsequence that is common to all the given sequences, provided that the elements of the subsequence are not required to occupy consecutive positions within the original sequences. If S1 and S2 are the two given sequences then, Z is the common subsequence of S1 and S2 if Z is a subsequence of both S1 and S2. Furthermore, Z must be a strictly increasing sequence of the indices of both S1 and S2. In a strictly increasing sequence, the indices of the elements chosen from the original sequences must be in ascending order in Z.
eg. For S1 = "abcdef" and S2 = "acdehlmf", LCS = "abcef"

Now back to Character Error Rate. It is defined as the ratio of the LCS of two words divided by the length of the longer word. To compute Character Error Rate for test data, we take the ratio of the Longest Common Subsequence of the output English word and the given English word in the dataset with the maximum of the length of both those words. Generally, Character Error Rate is bound to be lesser than the Word Error Rate because even in case of words that could not completely match, it is still probable that at least some characters would still match and the Longest Common Subsequence will not be an empty string. Eg.

| Input | Predicted Output | Target | LCS | Accuracy |
|---|---|---|---|---|
| महल | mahal | mahal | mahal | 5/5 |
| हँसना | hansna | hasna | hasna | 5/6 |

Table 5.2 An Example of Character Error Rate

# 6. ANALYSIS OF RESULTS

In this section, we present the performance of our proposed method.

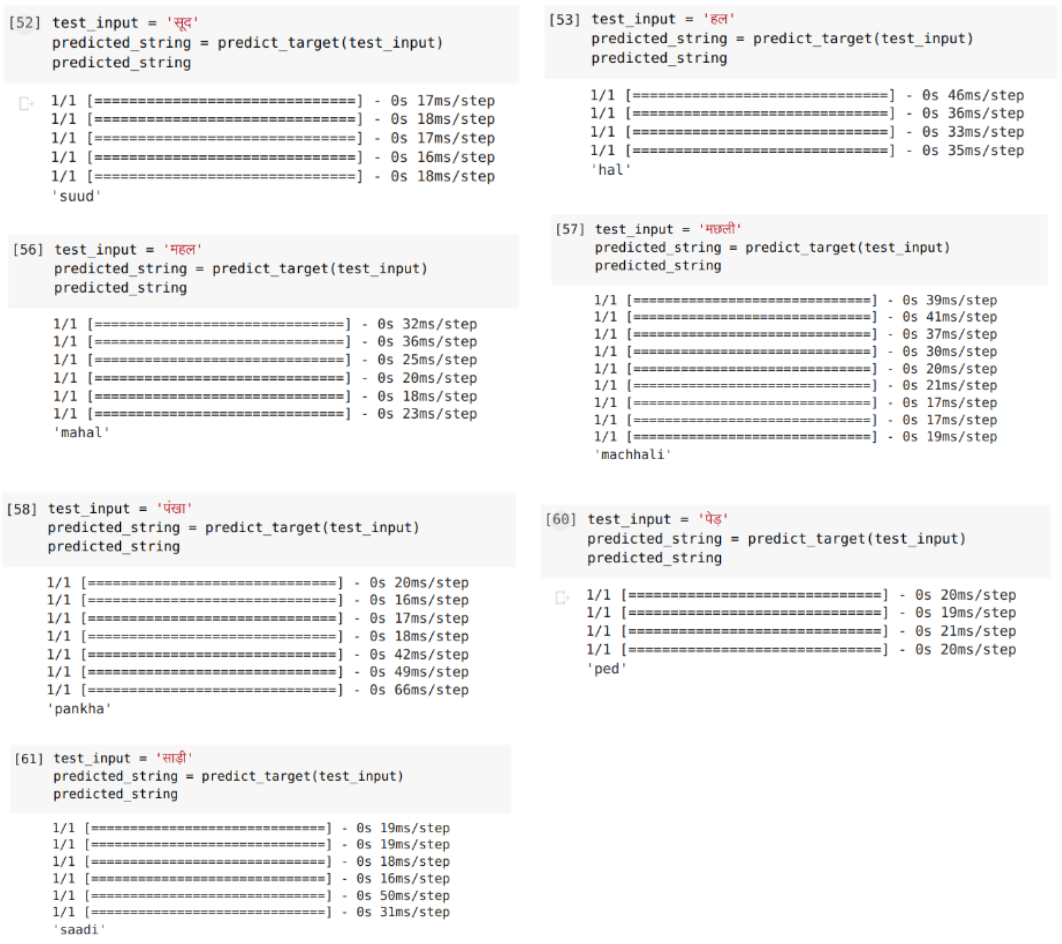


Fig. 6.1 A snapshot of Results

## Word Error Rate Accuracy

| Total Words | Wrongly Predicted Words | Word Error Rate |
|---|---|---|
| 3083 | 2029 | 65.85 |

Table 6.1 Word Error Rate Accuracy

## Character Error Rate Accuracy

| Total Characters | Wrongly Predicted Characters | Character Error Rate |
|---|---|---|
| 21357 | 2873 | 13.46 |

Table 6.2 Character Error Rate Accuracy

## Discussion

As expected, the Character Error Rate is much lesser than the Word Error Rate, but the Character Error Rate being so low indicates that the error in predictions produced by our model are minute and at least part of the outputs generated match with the Roman words in the dataset. The outputs might be off by a factor of about 0.13 when it comes to character-by-character matching but that was expected since a word in Devanagari does not have a single correct Roman transliterated version, rather, it can be converted to a number of outputs, all of them being equally valid.

# 7. CONCLUSION & FUTURE WORK

In this work, a neural network approach is introduced in machine transliteration, more specifically, an Encoder-Decoder architecture with LSTM memory cells are used to generate character by character output of a Devanagari word to its Roman counterpart by using memory of the previously generated characters using LSTM. The developed system learns the pattern of unknown characters from its own using training-corpus and it can be applied to any Devanagari-Roman pair of words. The proposed approach is tested on two different evaluation measures of Natural Language Processing (NLP), i.e, Word Error Rate and Character Error Rate. The Word Error Rate turned out to be 65.85 % and the Character Error Count turned out to be 13.46 %. The low Character Error Rate as compared to the Word Error Rate implies that our model is able to predict words with a minute error with respect to the similarity of characters in the target words and the predicted words.

There is a lot of scope for improvement in the dimensionality of the model. The major limitations of our model can be something to look forward to in our future work. The Character Error Rate can be further improved by implementing BERT as BERT has given proven results in the field of neural machine translation which can turn out to be similar to our domain transliteration. Furthermore, Our model can only process test data that has words not longer than the words in the training set. So, if a testing set is used having words with length greater than the length of the longest word in the training data, then this model might not produce optimum results. Also, the model proposed is language dependent right now, with future research models which are bilingual and bidirectional can also be explored.

# 8. PROGRAMMING ENVIRONMENT AND TOOLS USED

Python 3.x is used as the programming language to implement all the models and architecture of the Neural Networks. Python 3.x is an interpreted, object-oriented, high-level programming language with dynamic semantics. Its high-level built in data structures, combined with dynamic typing and dynamic binding, make it very attractive for Rapid Application Development, as well as for use as a scripting or glue language to connect existing components together. The Python 3.x interpreter and the extensive standard library are available in source or binary form without charge for all major platforms, and can be freely distributed.

**Google Colab** is used as the development environment for the implementation of the designed architecture. Google Colab is a free Jupyter notebook environment that runs entirely in the cloud. The notebooks can be simultaneously modified by team members and most importantly, it does not require any setup. Many common machine learning libraries are supported by Colab and can be quickly loaded into the notebook. A remote machine is allotted to the user and all the heavy work of processing and training the data can be done remotely on a remote machine which is allotted to the user while running the program. The model is trained and can be downloaded according to the user's satisfaction.

**Pandas** is a Python library used to load data and pre-process the data so that it is appropriate to be fed into the model such that the maximum accuracy is obtained. Pandas is an open-source library that is made mainly for working with relational or labelled data both easily and intuitively. It provides various data structures and operations for manipulating numerical data and time series. This library is built on top of the NumPy library. Pandas is fast and it has high performance & productivity for users. Through Pandas one can obtain a dataframe which is a two-dimensional size-mutable, potentially heterogeneous tabular data structure with labelled axes (rows and columns). A Data frame is a two-dimensional data structure, i.e., data is aligned in a tabular fashion in rows and columns. Pandas DataFrame consists of three principal components, the data, rows, and columns.

**Keras** is a Python Framework and High Level API used to implement the Architecture. Keras is a powerful and easy-to-use open-source Deep Learning library for Python. It allows one to easily build and train neural networks and deep learning

models. Keras is also one of the most popular Deep Learning frameworks among researchers and developers. Keras was developed and is maintained by a team of experienced developers and contributors. Keras is also backed by a large community of users and developers. Keras runs on top of TensorFlow, Theano, or CNTK thus providing a high accuracy.

**Matplotlib** is a Python library which is used to plot the accuracy and loss of the model. Matplotlib is a comprehensive library used for visualisation in Python. The library is mostly written in python, a few segments are written in C, Objective-C and Javascript for Platform compatibility.

# 9. REFERENCES


Laskar, Sahinur Rahman, et al. "Investigation of English to Hindi Multimodal Neural Machine Translation using Transliteration-based Phrase Pairs Augmentation." Proceedings of the 9th Workshop on Asian Translation. 2022.

Singh, Aryan, and Jhalak Bansal. "Neural Machine Transliteration Of Indian Languages." 2021 4th International Conference on Computing and Communications Technologies (ICCCT). IEEE, 2021.

Soni, Kartik. "SEQUENCE TRANSLITERATION USING ENCODER-DECODER MODELS." (2020).

Dhore, M. L., and P. H. Rathod. "Hindi and Urdu to English Named Entity Statistical Machine Transliteration Using Source Language Word Origin Context." Applied Machine Learning for Smart Data Analysis. CRC Press, 2019. 3-20.

K. Deep and V. Goyal, "Development of a Punjabi to English transliteration system,"International Journal of Computer Science and Communication, vol. 2, pp. 521- 526, 2011.

P. Antony, V. Ajith, and K. Soman, "Kernel method for english to kannada transliteration,"in Recent Trends in Information, Telecommunication and Computing (ITC), 2010 International Conference on, 2010, pp. 336-338.

M. L. Dhore, S. K. Dixit, and T. D. Sonwalkar, "Hindi to english machine transliteration of named entities using conditional random fields," International Journal of Computer Applications, vol. 48, pp. 31-37, 2012

P. Sanjanaashree, "Joint layer based deep learning framework for bilingual machine transliteration," in Advances in Computing, Communications and Informatics (ICACCI, 2014 International Conference on, 2014, pp. 1737-1743.

S. Mathur and V. P. Saxena, "Hybrid appraoch to EnglishHindi name entity transliteration," in Electrical, Electronics and Computer Science (SCEECS), 2014 IEEE Students' Conference on, 2014, pp. 1-5.