# Midterm Report
# Summer of Science 2024: Natural Language Processing

Mentee: Abhi Jain
Mentor: Nilesh Choudhary

June 25, 2024

# Contents

# Chapter 1

# Week 1: Introduction to NLP and Text Preprocessing

## 1.1 Overview of NLP

Natural Language Processing (NLP) deals with getting computers to understand and process human language: unstructured and complex by nature. Computers are great at parsing structured data such as spreadsheets, but it gets much more difficult when you include the complexities of human language.

NLP seeks to bridge this gap by teaching models how to understand and manipulate huge volumes of natural language material, be it written or spoken words. NLP has since become more mainstream and approachable, ushering in newer models like ChatGPT. ChatGPT is a fine example of our vast advancements in reducing the gap between man-to-computer interaction. One can be blown away by its capabilities of understanding as well as producing human language.

NLP breaks the process into smaller manageable tasks, allowing effective model training and handling of language-related problems. Ultimately, NLP strives to make human-computer interaction more like human-human interaction in a way.

## 1.2 Tokenization

Tokenization deals with how to represent words in a way that computers can process them.

For example, the word "listen" can be represented by numbers using an encoding scheme. A popular one called ASCII can be used. This bunch of numbers can then define the word listen. But the word silent has the same letters and thus the same numbers, just in a different order. So it makes it hard for us to understand the sentiment of a word just by the letters in it. So, it might be easier to encode words instead of encoding letters.

Consider the sentence "I love c++" We can encode both words as well as letters but encoding words seems better. If we encode the words instead of the letters, we can assign a unique number to each word. Therefore, the sentence "I love c++" could be encoded as 1, 2, 3. Now, if we take another sentence like "I love python" the words "I love" have already been assigned 1, 2. We only need to encode the new word "python" which could be given the number 4. This method shows the similarity between the sentences. But

this method can't differentiate between the language python and snake python. Both will be encode the same. Now, if you add the word "python!" in a new sentence the tokenizer is smart enough to not create a new token for that, since a token for "python" is already present.

Tokenizer API from the Keras library can be used to achieve tokenization.

## 1.3 Stop-word Removal

Stop words are the commonly occurring words in the language that doesn't often add much meaning to the sentence but take a lot of computational resources. Words like "the", "is", "and", etc don't contribute much to the meaning as we all know, and hence these words are considered as noice. Hence, removing these less important words to extract only meaningful information from the given text could help.

In general, the libraries also ignore the "not" word by considering it a stop word. But in problems like sentiment analysis, the presence of not can highly affect a sentence. For example, "Python is an easy language" and "Python is not an easy language" have completely different meanings. So, in such cases, we don't consider "not" a stop word. Hence, in sentiment analysis tasks, careful consideration is given to the treatment of negations and other contextually important words, ensuring that the analysis accurately captures the intended sentiment expressed in the text.

In Python, several libraries are commonly used for natural language processing tasks like handling stop words. NLTK is one of the popular library for NLP tasks in Python. It provides various modules, including nltk.corpus.stopwords, which contains lists of stop words for different languages and hence can be used for stop word removal.

## 1.4 Stemming and Lemmatization

NLP uses two methods for text normalization: stemming and lemmatization. By reducing words to their base or root forms, these techniques aid in the reduction of vocabulary size. I can simplify "languages" to just "language," for instance.

### Stemming:

Reducing inflected (or occasionally derived) words to their most basic form is known as stemming. Prefixes and suffixes are eliminated in this procedure in an effort to compress related words to a single stem. Stemming can occasionally produce stems that are not real words because it is based on heuristic rules rather than linguistic expertise.

Example:

- Word: "running"
  Stem: "run"
- Word: "cats"
  Stem: "cat"
- Word: "better"
  Stem: "better" (no change, as stemming may not always reduce irregular forms)

### Lemmatization:

Contrarily, lemmatization is the process of finding the lemma or dictionary form of a word. Lemmatization applies more advanced rules in order to guarantee a root word to be part of the language, rather than stemming which betrays simplicity. It is the process of reducing words to their base or dictionary form taking into account the context of a word.

Example:

- Word: "better"
  Lemma: "good"
- Word: "cats"
  Lemma: "cat"
- Word: "running"
  Lemma: "run"

## 1.5 Punctuation Removal

Full stops, commas, question marks, exclamation marks, quotation marks, etc. work in natural language as interpretative and organisational signs. However, the peritome of imagined punctuations are ignored in many NLP tasks as they do not contain and have no implied meaning in the text. The process of text filtration for the purpose of subsequent analytical processing can be enhanced by the elimination of punctuation marks as these do not contribute to the identification of important keywords and barriers that form the basis of a text.

# Chapter 2

# Week 2: Text Representation

To train the Neural Networks to understand the text in NLP. We need to convert the raw text into numerical form. There are a lot of different techniques which can be used for the purpose. Extracting features is always a crucial part of ML. Before that let's try to understand some common terminologies :

- **Corpus:** Corpus is the collection of all the written and spoken material. In NLP, it is stored for training and evaluating models.
- **Vocabulary:** Vocaubulary is the set of all the unique words that appear in the Corpus.
- **Documents:** In the context of NLP, a document is a single piece of text within a corpus. It can be a sentence, a paragraph, an article, or any other segment of text.
- **Words:** Words as we all know are the basic units of document.

Let's take an example using which we will be understanding the different techniques. Its just a random data created by me. The data contains 4 statements and the outputs 0 or 1 (random for now as created by me). The documents have already been preprocessed using the techniques learnt earlier.

| D.No. | Documents | Output |
|---|---|---|
| D1 | `college student enjoy` | 0 |
| D2 | `school college chill` | 1 |
| D3 | `life enjoy life` | 1 |
| D4 | `student enjoy life` | 0 |

**Corpus(C):** `college student enjoy school college chill life enjoy life student enjoy life`
**Vocabulary(V):** `[ college, student, enjoy, school, chill, life ]`

## 2.1 One-Hot-Encoding

This technique involves coding each word in the vocabulary as a V-dimensional vector with 1 at the position of word in Vocabulary and 0 everywhere else. Therefore, each document is essentially a vector of vectors i.e., matrix of dimension : $N(D) \times V$.

For Example: D1 and D2 will be encoded as:

$$\mathbf{D1} = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \end{bmatrix}, \mathbf{D2} = \begin{bmatrix} 0 & 0 & 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \end{bmatrix} \tag{2.1}$$

### Advantages

- Easy to Implement
- Intuitive

### Disadvantages

- **Sparsity:** As for encoding given sets of words in documents, one-hot encoding leads to rather sparse vectors, in which most of the coordinates are equal to zero. This results to sparse storage and computation that if not well handled can cause inefficient use of available resource. This also leads to overfitting.
- **High Dimensionality:** The major issue arising from the high-dimensional vectors is that as the word vocabulary increases, the vectors tend to be very large, thus results into larger computations and storage requirements.
- **Lack of Semantic Meaning:** Another important aspect is that values of vectors obtained via one-hot encoding do not contain any information about semantic associations between words. For example, distance between the words "enjoy" and "chill" would be as much as the words "chill" and "college", but semantically "chill" is closer to "enjoy" rather than to "college".
- **Scalability Issues:** This becomes a problem as we go through the sizes of vocabulary because the size of the one-hot encoded vectors also ranks high, and thus the larger the data set, the harder it becomes to handle the models.
- **OOV:** Out of Vocabulary Problem, so if there is a word which is not in the vocabulary its meaning will be lost.
- **No fix Size:** All matrices for documents will be of different sizes,

## 2.2 Bag of Words(BOW)

In this method we store each document as a V-dimensional vector where each position represents a word's frequency. This method holds good in text - classification problems. Closeness between statements can be seen as the angle made between the Vectors in the V-dimensional space.

| D.No. | college | student | enjoy | school | chill | life |
|---|---|---|---|---|---|---|
| D1 | 1 | 1 | 1 | 0 | 0 | 0 |
| D2 | 1 | 0 | 0 | 1 | 1 | 0 |
| D3 | 0 | 0 | 1 | 0 | 0 | 2 |
| D4 | 0 | 1 | 1 | 0 | 0 | 1 |

### Advantages

- **Simplicity:** The concept of BOW is easy and intuitive.
- **Efficiency:** High computational efficiency for small to medium-sized datasets has been seen.
- **Baseline Performance:** It serves as a strong baseline for many NLP tasks, like text classification, information retrieval in search engines.
- **Non-parametric Nature:** Requires minimal parameter tuning.

### Disadvantages

- **Loss of Context:** It ignore the order of words which is a big problem. For example: "It's a pleasant day" and "It's not a pleasant day" will be very close vectors to each other due to a lot of common words but it isn't so.
- **High Dimensionality:** Results in high memory usage and computational costs for large vocabularies.
- **Sparsity:** Produces sparse vectors with many zeros.
- **Lack of Semantic Meaning:** Does not capture relationships between words.
- **OOV:** Cannot represent unseen words during inference.

## 2.3 n-gram Models (uni-gram, bi-gram, tri-gram)

n-gram model is very inspired by the BOW model. In n-gram model, we consider vocabulary to be made up of a sequence of n-continuous words. For example, in bi-gram rather than breaking sentence into words we break it into pair of words.

### How n-gram Models Work

- **Unigrams** (1-grams): It considers each word as an individual token. This is essentially the Bag of Words model.
- **Bigrams** (2-grams): It considers pairs of consecutive words. For example, in the D1 "college student enjoy" the bigrams are "college student" and "student enjoy". Here, we implement a model similar to BoW, the difference is that the Vocabulary(v) now consists of bigrams.
- **Trigrams** (3-grams): Similarly, it considers triplets of consecutive words.

### Advantages of n-gram Models

- **Captures Local Context:** Sometimes, the sequences of words do matter and may be effective in capturing the local context and relations.
- **Improved Performance:** Sometimes reflects higher levels of performance for operations that involve the arrangement of words in a particular sequence or ability to comprehend context.
- **Flexibility:** Th value of 'n' can be changed accordingly to capture more context.

### Disadvantages of n-gram Models

- **Increased Dimensionality:** It is obvious that larger value of n will result in exponential growth of vocabulary size.
- **Sparsity:** Higher-order n-grams result in even sparser representations.

Other advantages and disadvantages of BoW still continue to exist.

## 2.4 TF-IDF

TF-IDF (Term Frequency-Inverse Document Frequency) is a statistical measure used to evaluate the importance of a word in a document relative to corpus. So, it essentially gives more weightage to terms that appear less in the corpus as whole but more in the document. If a word occurs very often in the corpus then it is given less weightage.

Term frequency of term t in document d is given by:

$$Tf(t,d) = \frac{\text{Number of occurrences of term t in document d}}{\text{Total number of terms in document d}}$$

Inverse Document Frequency of term t is given by:

$$Idf(t) = \log\left(\frac{\text{Total number of documents in the corpus}}{\text{Number of documents with text t in them}}\right)$$

Note: In some of the python libraries 1 is added to the Idf so as to not eliminate the terms which occur in all the documents.

| | **college** | **student** | **enjoy** | **school** | **chill** | **life** |
|---|---|---|---|---|---|---|
| $Idf(t)$ | $\log(\frac{4}{2})$ | $\log(\frac{4}{3})$ | $\log(\frac{4}{3})$ | $\log(\frac{4}{1})$ | $\log(\frac{4}{1})$ | $\log(\frac{4}{2})$ |
| | 0.3 | 0.125 | 0.125 | 0.6 | 0.6 | 0.3 |

Therefore, now value of each term in the vector for document is now $Tf(t,d) \cdot Idf(t)$

### Advantages of TF-IDF

- **Simple and Effective:** It is very simple to perform and offers a clear approach for identifying a chief term in a document.
- **Highlights Rare but Important Terms:** This method down-weights words that often appear frequently and up-weights the less frequently occurring words in order to obtain more of the right words.
- **Widely Used:** A common method of performing information retrieval and text mining, often used as a benchmark method in many NLP problems.

### Disadvantages of TF-IDF

- **Sparsity:** The same disadvantage continues
- **Lack of Context:** It also has the same problem, where is doesn't account for the order of words.
- **Not Suitable for All NLP Tasks:** For activities that may involve analysis of semantics, contextual analysis or word relations, more complex models such as word embedding or even deep neural networks may be necessary.

## 2.5 Word Embeddings (Word2Vec)

Word2Vec is a popular algorithm developed by Tomas Mikolov and his team at Google in 2013. It represents words in a continuous vector space such that words that share common contexts are located near each other in the space.

### Advantages

- This method captures semantic relationships between words, enabling more sophisticated natural language processing (NLP) applications.
- Unlike the vectors we've seen so far, Word2Vec are short, with number of dimensions d ranging from 50-1000, rather than the much larger vocabulary size.
- The vectors are dense i.e., instead of vector entries being sparse, mostly-zero counts or functions of counts, the values will be real-valued numbers that can be negative.

The intuition for Word2Vec is that each word is a vector of small size say 300 for now. These 300 numbers can be thought of 300 features on which all numbers are categorised and given some number. How do we select such features? We use deep learning which automatically generates these numbers and features. The two common algorithms that we use are: CBOW(continuous bag of words) and Skip-Gram.

Word2vec embeddings are static embeddings, meaning that the method learns one fixed embedding for each word in the vocabulary.

### 2.5.1 Skip-Gram Algorithm

The intuition of skip-gram is:

- Treat the target word and a neighboring context word as positive examples.
- Randomly sample other words in the lexicon to get negative samples.
- Use logistic regression to train a classifier to distinguish those two cases.
- Use the learned weights as the embeddings.

### The classifier

Our goal is to train a classifier such that, given a tuple $(w; c)$ of a target word $w$ paired with a candidate context word $c$ it will return the probability that c is a real context word.

$$P(+|w, c) = \text{Probability that c is the real context word}$$

$$P(-|w, c) = \text{Probability that c is not the real context word}$$

The intuition of the skipgram model is to base this probability on embedding similarity i.e., if they have a high dot product.

$$Similarity(w; c) = \vec{w} \cdot \vec{c}$$

To turn the dot product into a probability, we'll use the logistic or sigmoid function.

$$\text{Therefore, } P(+|w, c) = \sigma(\vec{w} \cdot \vec{c}) = \frac{1}{1 + exp(\vec{w} \cdot \vec{c})} \tag{2.2}$$

$$\text{and } P(-|w, c) = 1 - P(+|w, c) = \sigma(\vec{w} \cdot -\vec{c}) = \frac{1}{1 + exp(\vec{w} \cdot -\vec{c})} \tag{2.3}$$

Skip-gram makes the simplifying assumption that all context words are independent, allowing us to just multiply their probabilities. So, if we are taking a window of size L i.e., checking L context words on both sides then:

$$P(+|w, c_{1:L}) = \prod_{i=1}^{L} P(+|w, c_i) = \prod_{i=1}^{L} \sigma(\vec{w} \cdot \vec{c}) \tag{2.4}$$

$$\log(P(+|w, c_{1:L})) = \sum_{i=1}^{L} \log(\sigma(\vec{w} \cdot \vec{c})) \tag{2.5}$$

In summary, skip-gram trains a probabilistic classifier that, given a test target word $w$ and its context window of $L$ words $c_{1:L}$, assigns a probability based on how similar this context window is to the target word. To compute this probability, we just need embeddings for each target word and context word in the vocabulary.

Skip-gram actually stores two embeddings for each word, one for the word as a target, and one for the word considered as context. Thus the parameters we need to learn are two matrices **W** and **C**, each containing an embedding for every one of the $|V|$ words in the vocabulary $V$.

### Learning skip-gram embeddings

The learning algorithm for skip-gram embeddings takes as input a corpus of text, and a chosen vocabulary size $|V|$.

For training a binary classifier we also need negative examples. In fact skip-gram with negative sampling (SGNS) uses more negative examples than positive examples (with the ratio between them set by a parameter k). So for each of these $(w; c_{pos})$ training instances we'll create k negative samples, each consisting of the target $w$ plus a 'noise word' $c_{neg}$. A noise word is a random word from the lexicon, constrained not to be the target word w.

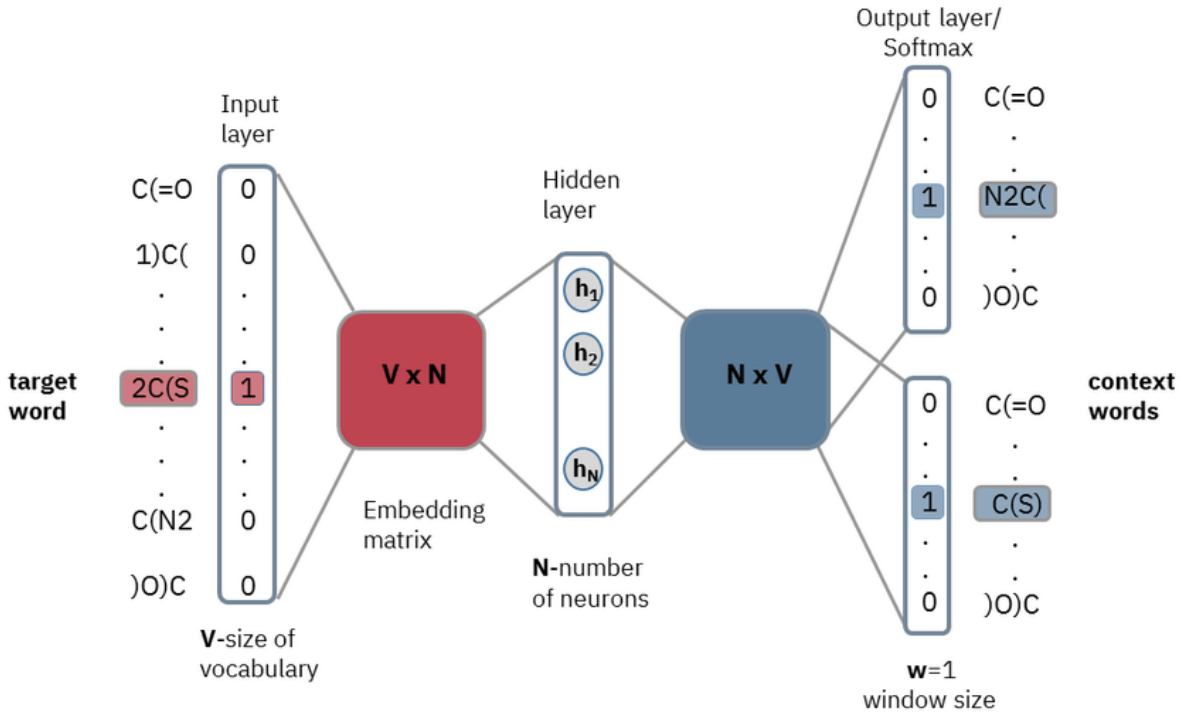


Figure 2.1: Skip-Gram Algorithm, in the figure N is the size of word - embedding vector, $V \times N$ is the $W$ Matrix and $N \times V$ is the $C$ Matrix

Given the set of positive and negative training instances, and an initial set of embeddings, the goal of the learning algorithm is to adjust those embeddings to

- Maximize the similarity of the $(w; c_{pos})$ pairs from the positive examples.
- Minimize the similarity of the $(w; c_{neg})$ pairs from the negative examples.

If we consider one word/context pair $(w; c_{pos})$ with its k noise words $c_{neg_1} \ldots c_{neg_1}$, we can express these two goals as the following loss function to be minimized (The minus sign is is to make the value positive);

$$L_{CE} = -\log\left(P(+|w, c_{pos}) \prod_{i=1}^{k} P(-|w, c_{neg_i})\right) \tag{2.6}$$

$$L_{CE} = -\log(\sigma(\vec{w} \cdot \vec{c_{pos}})) - \left(\sum_{i=1}^{k} \log(\sigma(\vec{w} \cdot -\vec{c_{neg_i}}))\right) \tag{2.7}$$

We minimize this loss function using stochastic gradient descent. The updated equations going from time step t to t +1 in stochastic gradient descent are thus:

$$\vec{c_{pos}}^{t+1} = \vec{c_{pos}}^{t} - \eta\left(\frac{\partial L_{CE}}{\partial c_{pos}}\right) \tag{2.8}$$

$$\vec{c_{neg}}^{t+1} = \vec{c_{neg}}^{t} - \eta\left(\frac{\partial L_{CE}}{\partial c_{neg}}\right) \tag{2.9}$$

$$\vec{w}^{t+1} = \vec{w}^{t} - \eta\left(\frac{\partial L_{CE}}{\partial w}\right) \tag{2.10}$$

Just as in logistic regression, then, the learning algorithm starts with randomly initialized $W$ and $C$ matrices, and then walks through the training corpus using gradient descent to move $W$ and $C$ so as to minimize the loss in 2.7 by making the updates in 2.8 - 2.10.

Hence, the skip-gram model learns two separate embeddings for each word $i$: the target embedding $w_i$ and the context embedding $c_i$, stored in two matrices, the embedding target matrix $W$ and the context matrix $C$.

# Chapter 3

# Week 3: Deep Learning for NLP

In this chapter we aim to delve into the application of deep learning techniques to NLP tasks, highlighting the capabilities of various neural network architectures in capturing the intricacies of natural language.

Once we have preprocessed and prepared the input data, transforming it into a numerical form that models can process, we leverage machine learning and deep learning models to perform various NLP tasks. The choice of model is influenced by the nature of the input data and the specific requirements of the task at hand. Currently, in SOS we will primarily focus on Neural Networks, Recurrent Neural Networks (RNNs), and Long Short-Term Memory Networks (LSTMs).

## 3.1 Neural Networks

Neural networks are a subset of machine learning and are inspired by the structure and function of the human brain. They are composed of interconnected units called neurons, which work together to process and learn from data.

Neural networks can capture complex patterns and representations, making them suitable for various applications, including image and speech recognition, and especially natural language processing (NLP).

### 3.1.1 Units

unit takes a set of real-valued numbers as input, performs some computation on them, and produces an output. Given a set of inputs $x_1 \dots x_n$, a unit has a set of corresponding weights $w_1 \dots w_n$ and a bias $b$, so the weighted sum z can be represented as:

$$z = \vec{w} \cdot \vec{x} + b$$

Instead of using z, a linear function of x, as the output, neural units apply a non-linear function f to z. We refer to output as the activation value for the unit and the function f as the activation function. Some popular examples of activation functions include sigmoid, ReLU, tanh, etc (See 3.1)

$$y = a = f(z)$$

These activation functions have different properties that make them useful for different language applications or network architectures.

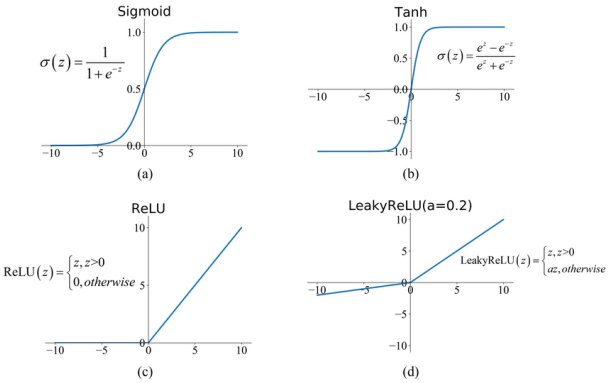


Figure 3.1: Popular Activation functions

The derivative in the case of sigmoid, tanh is very close to 0 for high values of y and hence y becomes saturated. But while training networks by propagating an error signal backward multiplying gradients from each layer of the network, gradients that are almost zero cause the error signal to get smaller. A problem called the **vanishing gradient problem**. Rectifiers don't have this problem, since the derivative of ReLU for high values of z is 1 rather than very close to 0.

### 3.1.2 Feedforward Neural Networks

It is one of the very simplest kinds of Neural networks. A feedforward network is a multi-layer network in which the units are connected with no cycles, the outputs from units in each layer are passed to units in the next higher layer, and no outputs are passed back to lower layers.

It has 3 kind of nodes: Input Units, Hidden units, Output Units

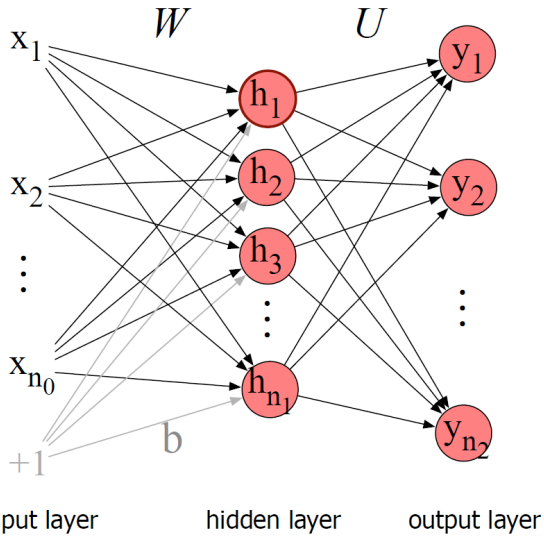


Figure 3.2: A simple 2-layer feedforward network, with one hidden layer, one output layer, and one input layer

In the standard architecture, each layer is **fully connected**, meaning that each unit in each layer takes as input the outputs from all the units in the previous layer, and there

is a link between every pair of units from two adjacent layers. Thus each hidden unit sums over all the input units. Each element $W_{ji}$ of the weight matrix $\mathbf{W}$ represents the weight of the connection from the $i^{th}$ input unit $x_i$ to the $j^{th}$ hidden unit $h_j$. Using a weight matrix has the advantage that we can use single matrix multiplication to feed the next layer. In vector $\mathbf{b}$, $i_th$ element corresponds to the bias for the element $h_i$.

Therefore, Output $\mathbf{h}$ can be written as, where g is any activation function:

$$\mathbf{h} = g(\mathbf{W}\mathbf{x} + \mathbf{b})$$

and this produces an output z which can be written as (ignoring the bias for this example):

$$\mathbf{z} = g(\mathbf{U}\mathbf{h})$$

However, $\mathbf{z}$ can't be the output of the classifier, since it's a vector of real-valued numbers, while what we need for classification is a vector of probabilities. Therefore, we normalize the vector using softmax functions as:

$$softmax(z_i) = \frac{exp(z_i)}{\sum_{j=1}^{n2} exp(z_j)}$$

Therefore, the final output $\mathbf{y}$ is:

$$\mathbf{y} = softmax(\mathbf{z})$$

The sizes of different vectors and matrices are:

$$\mathbf{x} \in \mathbb{R}^{n_0}, \mathbf{W} \in \mathbb{R}^{n_0 \times n_1}, \mathbf{b} \in \mathbb{R}^{n_1}, \mathbf{h} \in \mathbb{R}^{n_1}, \mathbf{U} \in \mathbb{R}^{n_1 \times n_2} \text{ and } \mathbf{y} \in \mathbb{R}^{n_2},$$

This is a 2-layer network. So, by terminology, logical regression is a 1-layer network.

The algorithm for computing the forward step in an n-layer feedforward network, given the input vector $a^{[0]}$ is thus simply:
**for** i **in** 1...n
$\mathbf{z}^{[\mathbf{i}]} = \mathbf{W}^{[\mathbf{i}]}\mathbf{z}^{[\mathbf{i}-\mathbf{i}]} + \mathbf{b}^{[\mathbf{i}]}$
$\mathbf{a}^{[\mathbf{i}]} = g^{[i]}\mathbf{z}^{[\mathbf{i}]}$
$\hat{\mathbf{y}} = \mathbf{a}^{[\mathbf{n}]}$

**Important Notes**

- If we did not use nonlinear activation functions for each layer in a neural network the resulting network is exactly equivalent to a single-layer network.
- **Replacing the bias term:** We will often use a slightly simplified notation that represents the same function without referring to an explicit bias node b. Instead, we add a dummy node $a_0$ to each layer whose value will always be 1.

### 3.1.3 Training Neural Networks

The goal of the training is to learn the Weight Matrices and the bias vector for each layer so as to produce the output as close to the actual output.

We will need a **loss function** and we will apply the **gradient descent** optimization. For that we will need the vector that contains the partial derivative of the loss function with respect to each of the parameters. In logistic regression we can calculate the derivative of loss function with respect to any individual $w$ or $b$ but in neural networks there are millions of parameters in many layers, it's much harder to see how to compute the partial derivative of some weight in layer 1 when the loss is attached to some much later layer. The answer is the algorithm called **error backpropagation** or **backward differentiation**.

### Loss Function

The cross-entropy loss is simply the negative log of the output probability corresponding to the correct class, and we therefore also call this the **negative log likelihood loss**:

$$L_{CE}(\mathbf{y}, \hat{\mathbf{y}}) = -\log \hat{\mathbf{y}}_c = -\log \left( \frac{exp(\mathbf{z}_c)}{\sum_{i=1}^{K} exp(\mathbf{z}_i)} \right) \tag{3.1}$$

,where $K$ is the number of classes, $c$ is the correct class, $\mathbf{y}$ is the correct output vector, $\hat{\mathbf{y}}$ is the received output vector after normalisation

### Backpropogation

Back-propagation is a critical process in training neural networks where weights and biases are adjusted to improve the network's fitting to the training data. It involves calculating gradients of the loss function with respect to each weight and bias using partial derivatives. Backwards differentiation makes use of the chain rule in calculus. This process could be understood with the analogy of the following function:

$$L(a, b, c) = c(a + 2b)$$

If we make each of the operation like addition multiplication explicit then,

$$d = 2 * b, e = a + d \text{ and } L = c * e$$

Therefore, the two derivatives are calculated using chain rule as follows:

$$\frac{\partial L}{\partial a} = \frac{\partial e}{\partial a}\frac{\partial L}{\partial e}$$

$$\frac{\partial L}{\partial b} = \frac{\partial d}{\partial b}\frac{\partial e}{\partial d}\frac{\partial L}{\partial e}$$

See the figure 3.3 to understand the back propagation using the analogy.

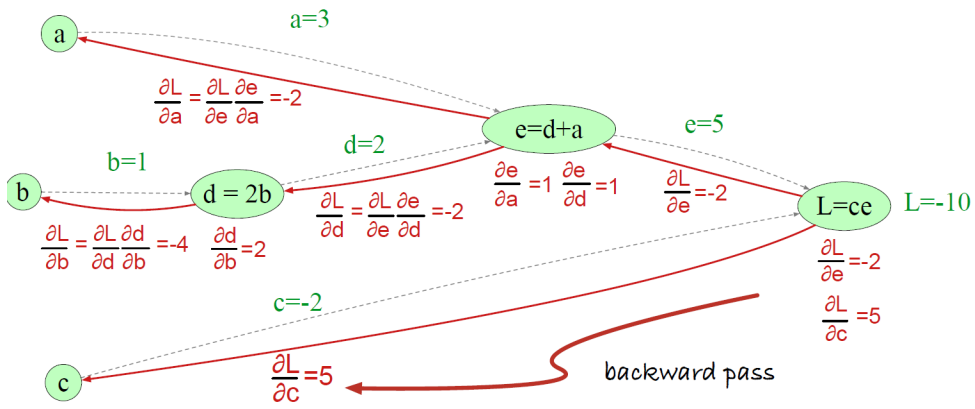


Figure 3.3: Computation graph for the function $L(a, b, c) = c(a + 2b)$, showing the backward pass computation

### 3.1.4 Optimising the Learning

- In neural networks we need to initialize the weights with small random numbers.
- **Dropout:** Various forms of regularization are used to prevent over-fitting. One of them is dropout i.e., randomly dropping some units and their connections from the network during training.
- **Tuning hyperparameters:** The parameters of a neural network are the weights $W$ and biases $b$; those are learned by gradient descent. The hyperparameters are things that are chosen by the algorithm designer; optimal values are tuned on a dev set rather than by gradient descent learning on the training set. Hyperparameters include the learning rate h, the mini-batch size, the model architecture (the number of layers, the number of hidden nodes per layer, the choice of activation functions), how to regularize, and so on.

## 3.2 Recurrent Neural Networks (RNNs)

We can apply feedforward neural networks to language modeling by making them look at a fixed-size window of words and then by sliding this window over the input, making independent predictions along the way. Through, RNNs and LSTMs we see a different approach to representing time. RNNs mechanism allows us to deal with the sequential nature of language, without using arbitrary windows. It occurs through the recurrent connections in the RNN, allowing the model to take decisions from the words from the past.

RNN networks contain a cycle within its network connections, allowing some units to be directly or indirectly dependent on earlier outputs.

### 3.2.1 Forward Inference

We use three weight matrices **W**,**U** and **V** for the purpose. Network layers are recalculated for each time step, while the weight matrices are shared across all the steps. The

forward computation occurs as follows:

$$\mathbf{h_t} = g(\mathbf{U}\mathbf{h_{t-1}} + \mathbf{W}\mathbf{x_t})$$

$$\mathbf{y_t} = f(\mathbf{V}\mathbf{h_t})$$

The sizes of the matrix are per the size of the layers involved. See the figure 3.4 showing computations over time.

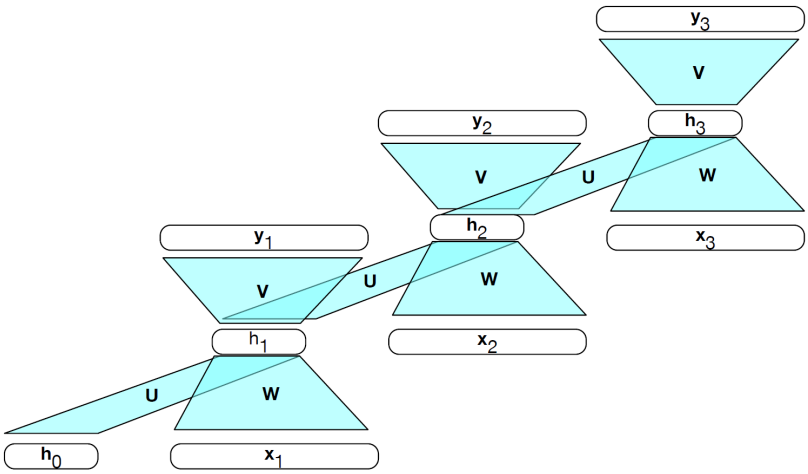


Figure 3.4: Simple RNN shown unrolled in time

### 3.2.2 Training

Now, we have three wight-matrices to be updated. While backpropagating in RNN following points are to be taken care of:

1. To compute the loss function for the output at time $t$ we need the hidden layer from time $t-1$.

2. The hidden layer at time $t$ influences both the output at time $t$ and the hidden layer at time $t+1$ (and hence the output and loss at $t+1$).

So, the backpropagation algorithm involves two-passed for training the weights in RNNs as follows:

1. In the first pass, we perform forward inference, computing $\mathbf{h_t}$ and $\mathbf{y_t}$, accumulating the loss at each step in time, saving the value of the hidden layer at each step for use at the next time step.

2. In the second phase, we process the sequence in reverse, computing the required gradients as we go, computing and saving the error term for use in the hidden layer for each step backward in time.

## 3.3 RNNs as Language Models

Let's say we want to know what's the probability that "I am going to" ends with the word "Library". Language Models give us the ability to give a conditional probability to

every possible next word, giving a distribution over the entire vocabulary.

$$P(w_{1:n}) = P(w_n|w_{1:n-1}) \cdot P(w_{1:n-1}) = \prod_{i=1}^{n} P(w_i|w_{1:i-1})$$

RNN language models process the input sequence one word at a time, attempting to predict the next word from the current word and the previous hidden state.

### 3.3.1 Forward Inference in RNN language model

The input sequence $\mathbf{X} = [\mathbf{x_1}, \ldots, \mathbf{x_t}, \ldots, \mathbf{x_n}]$ consists of a series of words each represented as a one-hot vector of size $|V|$, and the output prediction $\mathbf{y}$ is a vector representing a probability distribution over the vocabulary. At each step, the model will use the word embedding matrix $\mathbf{E}$ to get the embedding for the current word, and then with the previous layer, it will create the new layer. The hidden layer is used to generate an output layer which is passed through softmax to generate a probability distribution over the entire vocabulary. At any time t:

$$\mathbf{e}_t = \mathbf{E}\mathbf{x}_t$$

$$\mathbf{h_t} = g(\mathbf{U}\mathbf{h_{t-1}} + \mathbf{W}\mathbf{e_t})$$

$$\mathbf{y_t} = softmax(\mathbf{V}\mathbf{h_t})$$

Therefore,

$$\mathbf{y_t}[k] = \text{Probability that } t+1^{th} \text{ word is } k^{th} \text{ component of } \mathbf{y_t} = P(w_{t+1} = k|w_{1:t})$$

$$P(w_{1:n}) = \prod_{i=1}^{n} P(w_i|w_{1:i-1}) = \prod_{i=1}^{n} \mathbf{y_i}[w_i]$$

### 3.3.2 Training a RNN language model

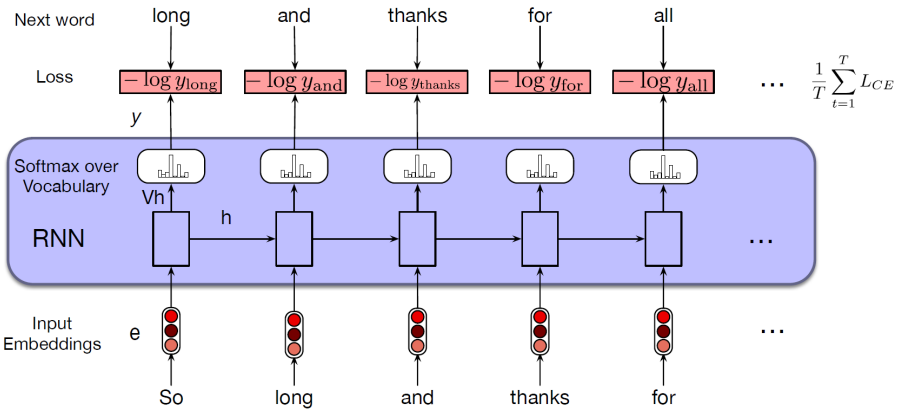


Figure 3.5: Training RNNs as language model

The method is the same, we take a corpus of text and ask the model to predict the next word. We train the model to minimize the loss function. We use cross-entropy loss which is the difference between a predicted probability distribution and the correct distribution.

So, at time $t$ the CE loss is the negative log probability the model assigns to the next word in the training sequence.

$$L_{CE}(\mathbf{y}, \hat{\mathbf{y}}) = -\log \hat{\mathbf{y}}_t[w_{t+1}]$$

This idea is that we always give the model the correct history sequence to predict the next word is called teacher forcing. The weights in the network are adjusted to minimize the average CE loss over the training sequence via gradient descent.

## 3.4 Long Short-Term Memory Networks (LSTMs)

In RNNs, it is quite difficult for the network to use information distant from the current point of processing. Despite having access to previous states, the information in the hidden states tends to be fairly local.

Consider the statement *"I went to the restaurant and ordered a cup of coffee. It was so hot that I had to wait for a few minutes before I could drink it."*

Here, to predict the word "drink", the context of coffee has to be remembered for quite a long time. The inability of RNN is due to two reasons:

- The weights that determine the hidden layer are made to do two tasks simultaneously: provide information for current decisions and update the information to be carried for future decisions.
- The Second difficulty is that we backpropagate the error back through time. The hidden layers are subject to repeated multiplication as determined by the sequence length for the backward pass of error. A frequent result of this process is that gradients are eventually driven to 0, a situation called the vanishing gradient problem.

To solve this issue, LSTM networks were discovered. It maintains two memories: one the long-term memory and the other the short-term memory, and that's why the name. It breaks down the process into two subproblems: removing the context no longer needed and adding the information likely to be needed in the future. The RNN architecture runs alongside.

LSTMs achieve this by adding another explicit context layer in addition to the usual recurrent hidden layer and it makes use of gates to control the flow of information between the units. The gates in an LSTM share a common design pattern; each consists of a feedforward layer, followed by a sigmoid activation function, followed by a Hadamard product (element-wise multiplication represented by $\odot$) with the layer being gated.

### 3.4.1 Forward Inference in LSTM

1. **Forget gate:** It gives the amount of information to be deleted from the context layer. It takes the weighted sum of the current input and the previous hidden layer to create a mask which when element-wise multiplied with the context removes the information that is no longer needed.

$$\mathbf{f}_t = \sigma(\mathbf{U}_f \mathbf{h}_{t-1} + \mathbf{W}_f \mathbf{x}_t)$$

$$\mathbf{k}_t = \mathbf{c}_{t-1} \odot \mathbf{f}_t \text{ (intermediate state between the two contexts)}$$

2. **Add gate:** First we use the previous hidden state and the current input to get the potential memory to be remembered and then create a mask using a similar procedure as the forget gate to get the percentage of the information to be added to the new context vector. Finally, add the vectors $\mathbf{k}_t$ and $\mathbf{j}_t$ to obtain the new context vector.

$$\mathbf{g}_t = \tanh(\mathbf{U}_g\mathbf{h}_{t-1} + \mathbf{W}_g\mathbf{x}_t) \text{ (potential memory)}$$

$$\mathbf{i}_t = \sigma(\mathbf{U}_i\mathbf{h}_{t-1} + \mathbf{W}_i\mathbf{x}_t) \text{ (mask)}$$

$$\mathbf{j}_t = \mathbf{g}_t \odot \mathbf{i}_t \text{ (new info to be remembered)}$$

$$\mathbf{c}_t = \mathbf{k}_t + \mathbf{j}_t \text{ (new context layer)}$$

3. **Output Gate:** Gives the information that is required for the current hidden state $\mathbf{h}_t$.

$$\mathbf{o}_t = \sigma(\mathbf{U}_o\mathbf{h}_{t-1} + \mathbf{W}_o\mathbf{x}_t)$$

$$\mathbf{h}_t = \mathbf{o}_t \odot \tanh(\mathbf{c}_t)$$

The LSTM then uses the current context vector and the output gate vector to generate the next hidden vector, So, we have 8 weight matrices in total inside the LSTM Gate architecture to be trained.

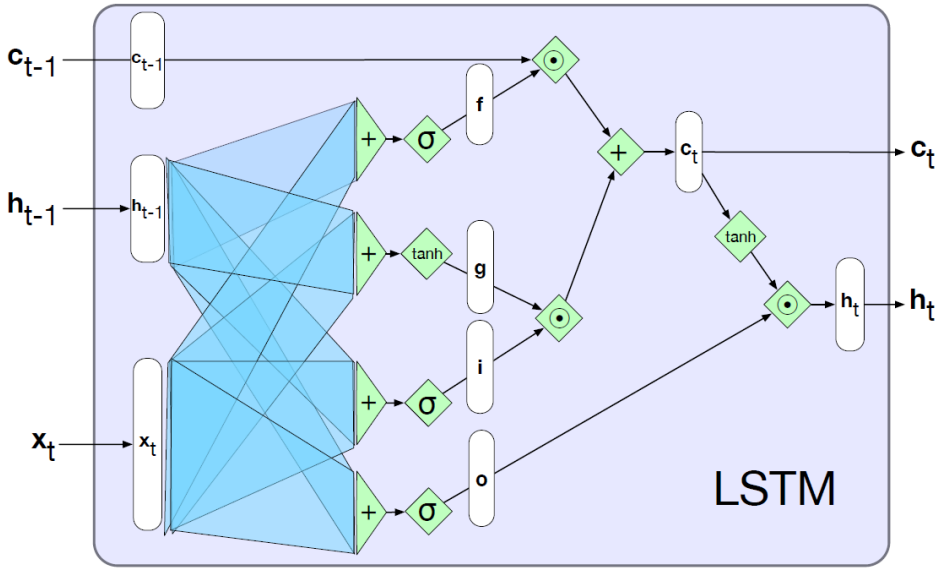


Figure 3.6: Single LSTM unit displayed as a computation graph

The increased complexity of the LSTM units is encapsulated within the unit itself. The only additional external complexity for the LSTM over the basic recurrent is the presence of the additional context vector as an input and output.

As with simple RNNs, multi-layered networks making use of gated units can be unrolled into deep feedforward networks and trained in the usual fashion with backpropagation. In practice, therefore, LSTMs rather than RNNs have become the standard unit for any modern system that makes use of recurrent networks.

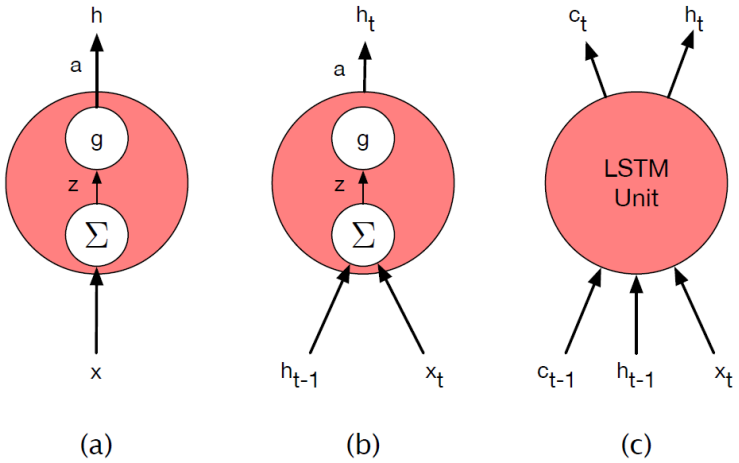


Figure 3.7: Basic neural units used in feedforward, simple recurrent networks (SRN), and long short-term memory (LSTM).

# Chapter 4

# Week 4: Attention Mechanism, Seq2seq, and Encoder-decoder Models

## 4.1 Encoder-decoder and Sequence-to-Sequence (Seq2seq) Models

In this type of models, we take sequence as inputs and output a sequence only, the best example of this is language translation. The problem with this is that both the size of input as well as output are of variable length. So, let's take the example of language translation and continue our understanding.

Since language translation doesn't involve word-to-word mapping, if the input sentence has 4 words the output sequence can still have any number of words depending on the language. So, in the encoder-decoder model the mapping in input token and the output token can be very indirect.

Encoder-decoder networks also known as sequence-to-sequence networks are models capable of generating contextually appropriate, arbitrary-length, output sequences, given an input sequence.

The main idea or intuition in this network is that we take an encoder which creates a context of the input sequence and the context vector is then passed to a decoder which converts it to an output sequence.

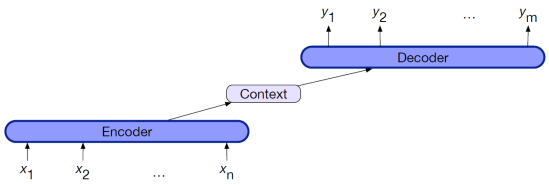


Figure 4.1: Encoder-Decoder architecture

RNNs, LSTMs, convolutional networks, and transformers can all be employed as encoders and decoders.

In this section, we'll describe an encoder-decoder network based on a pair of RNNs, but on the next subsection on the Seq2Seq model we will be seeing the model based on deep LSTMs based on the paper [1]

### 4.1.1 Encoder-Decoder model with RNNs

In RNN language modeling, at a particular time $t$, we pass the prefix of $t-1$ tokens through the language model, using forward inference to produce a sequence of hidden states, ending with the hidden state corresponding to the last word of the prefix. We then use the final hidden state of the prefix as our starting point to generate the next token.

$$\mathbf{h_t} = g(\mathbf{U}\mathbf{h_{t-1}} + \mathbf{W}\mathbf{x_t})$$

$$\mathbf{y_t} = f(\mathbf{V}\mathbf{h_t})$$

We only have to make one slight change to turn this language model into an encoder-decoder model which is a translation model that can translate from a source text in one language to a target text in a second language. Add a sentence separation marker ($< s >$) at the end of the source text, and then simply concatenate the target text.

Let $x$ = source-text, $y$ = target text, so our model just needs to compute the following probability:

$$P(y|x) = P(y1|x)P(y2|y1, x)P(y3|y_{1:2}, x)\ldots P(y_m|y_{1:m-1}, x)$$

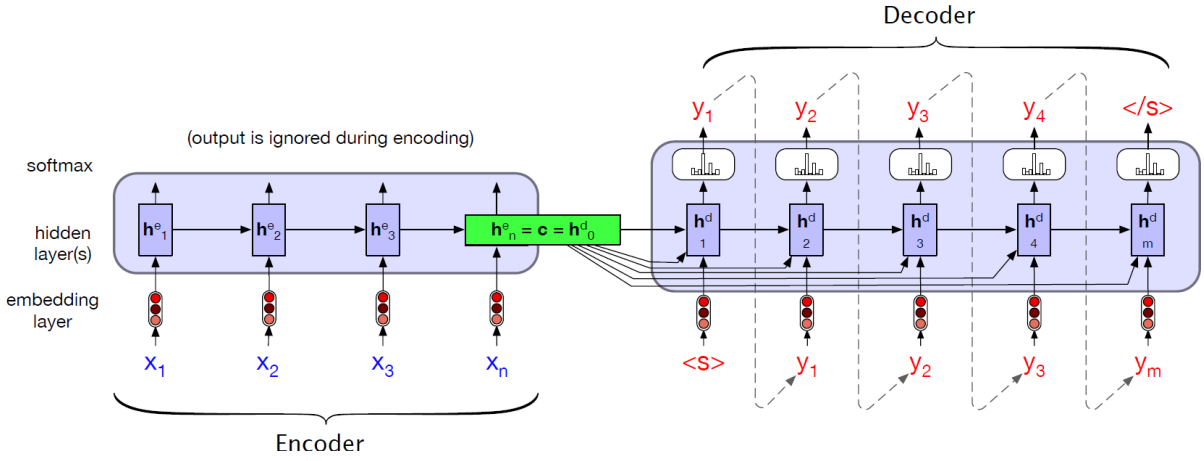


Figure 4.2: Translating a sentence at inference time in the basic RNN-based encoder-decoder architecture.

To translate the sentence through such networks, let's understand the architecture and forward inference of network through following points:

- First we pass the source text word by word, the words can be passed out as one-hot vectors and then converted to their word embeddings using an embedding layer.
- On each pass of word the next hidden state is calculated using the previous hidden state and the next word.
- As soon as we reach the end of the source text, the last hidden state is our context vector for the decoder. The entire purpose of the encoder is to generate a contextualized representation of the input. This representation is embodied in the final hidden state of the encoder($\mathbf{h}_n^e$).This representation, also called ($\mathbf{c}$) for context, is then passed to the decoder.

- Now, the decoder begins. The decoder then autoregressively generates a sequence of outputs, an element at a time until an end-of-sequence marker is generated.
- Each hidden state($\mathbf{h}_t^d$) is now a function of previous hidden state ($\mathbf{h}_{t-1}^d$), previous output($\hat{y}_{t-1}$), context vector($\mathbf{c}$). The context vector is available at each decoding step.

Following are the equations, for the process from the last hidden state:

$$\mathbf{c} = \mathbf{h}_n^e$$

$$\mathbf{h}_0^d = \mathbf{h}_n^e$$
$$\mathbf{h}_t^d = g(\mathbf{c}, \mathbf{h}_{t-1}^d, \hat{y}_{t-1})$$
$$\mathbf{z}_t = f(\mathbf{h}_t)$$
$$\mathbf{y}_t = \text{softmax}(\mathbf{z}_t)$$

The output $\mathbf{y}$ at each time step consists of a softmax computation over the set of possible outputs. We compute the most likely output at each time step by taking the argmax over the softmax output.

$$\hat{y}_t = \text{argmax}_{w \in V} P(w|y_{1:t-1}, x)$$

### Training the model

Each training example is a tuple of paired strings, a source and a target. Concatenated with a separator token, these source-target pairs can now serve as training data.

Once we have a training set, the training itself proceeds as with any RNN-based language model. The network is given the source text and then starting with the separator token is trained autoregressively to predict the next word, as shown in 4.3

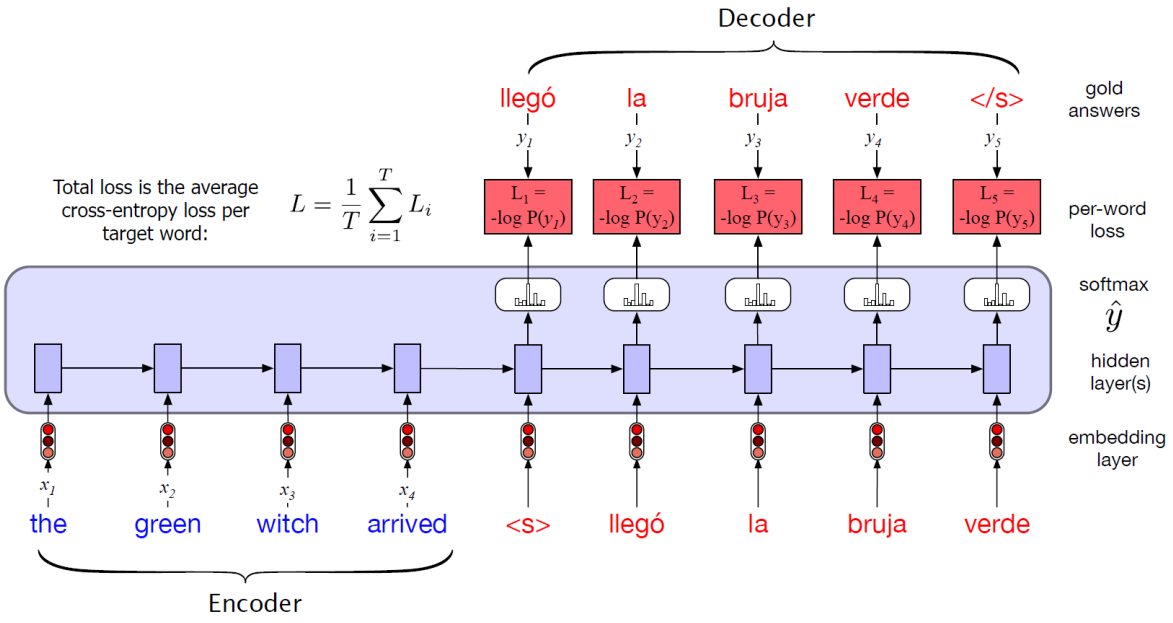


Figure 4.3: Training the basic RNN encoder-decoder approach to machine translation.

There is a difference between training and inference for the outputs at each time step. The decoder during inference uses its own estimated output $\hat{y}_t$ as the input for the next time step $t+1$. Thus the decoder will tend to deviate more and more from the gold target sentence as it keeps generating more tokens. In training, therefore, it is more common to use **teacher forcing** in the decoder. Teacher forcing means that we force the system to use the gold target token from training as the next input, rather than allowing it to rely on the decoder output $\hat{y}_t$. This speeds up training.

### 4.1.2 Seq2Seg model using deep LSTMs

Now, the problem with the model using RNNs is that it is unable to store context for too long. So, the final hidden state in RNNs is not always capable of carrying the whole context as what we have seen with RNNs in general. So, If we have to translate big paragraphs or something like a book ever, it's a very inefficient method. Hence, we use LSTMs, where the context vector passed is the same context vector that we have seen while studying LSTMs. This context vector similarly uses gates.

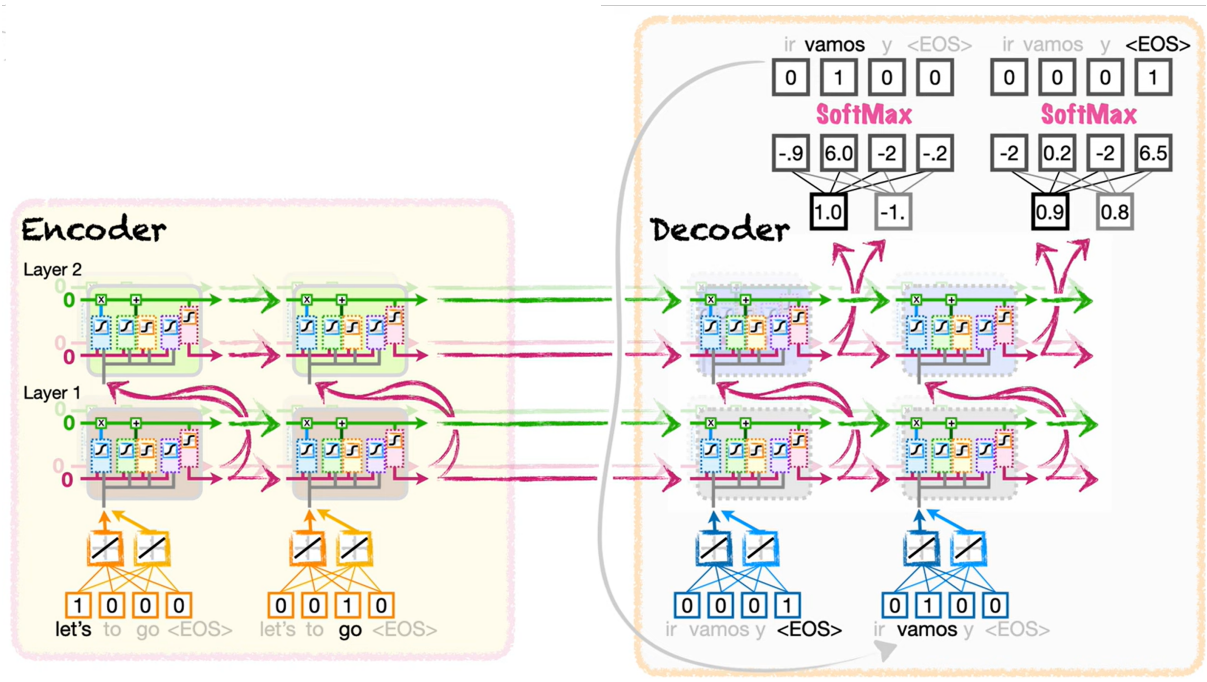


Figure 4.4: A Unfolded Encoder-decoder structure showing the translation of "let's go" in English to "vamos" in Spanish, $< EOS >$ represents the end of sentence

But a single LSTM is again also not enough sometimes and it has been observed that multiple LSTMs provide better results. Hence, we use a deep network of LSTMs as in the paper [11]. The deep network involves layers of LSTMs, where each layer passes its context vector and last hidden state to the decoder and the decoder also has layers of LSTMs to deal with it. The context vector from the k-1 layer forms the input for the context vectors for the layer k.

So, this could be understood as if there are 4 layers, the first layer will remember context at the word level, the second layer will remember context at the sentence level,

the third layer at the paragraph level, and the fourth level at the chapter or section level and so on. It's not how exactly it works but this way is how it could be understood.

From encoder to decoder hence, n context vectors and n hidden states are passed. A process similar to that we saw in the encoder-decoder model for RNN is applied for inference and training, where in training we will be using teacher forcing.

For better results, we can also do one more thing to have multiple LSTMs in each layer. For example in figure 4.4, we have two layers of LSTM and each layer has two LSTMs.

Also, it has been observed that in translation in certain languages reversing the source statement has helped because it makes the distance between the first word of the source and the target text less while increasing the distance between the last letters. For models, the order has no sense, it is just about obtaining the output by training the weights. It's not true for all language translations though.

Note: LSTM architecture used for each word in the figure 4.4 is the same in the encoder and similarly, each word's LSTM architecture is the same in the decoder, it is just that the LSTM model has been unfolded to obtain a computational graph.

The original manuscript ([11]) had an input vocabulary with 160,000 tokens and an output vocabulary with eighty thousand tokens. The original manuscript created one thousand embedding values per token and used four layers with one thousand LSTM cells per layer. Also, the output layer had one thousand inputs from the 1000 LSTM cells in the fourth layer and eighty thousand outputs to match the size of the output vocabulary. The model in the original manuscript had 384 million weights and biases to train and that gives you a sense of the scale that these models can have.

## 4.2 Attention Mechanism

The attention mechanism is an improvement in the encoder-decoder model. So, for discussion in this section, we will be considering the encoder-decoder applied through RNNs.

The simplicity of the encoder-decoder model is its clean separation of the encoder which builds a representation of the source text—from the decoder, which uses this context to generate a target text. This final hidden state thus represents everything about the source text in the form of a context vector. That's a pretty high load on the context vector as in the case of large statements, information at the beginning may not be equally well represented in the context vector.

The attention mechanism is a solution to the bottleneck problem, a way of mechanism allowing the decoder to get information from all the hidden states of the encoder, not just the last hidden state. For example, while translating languages a certain word in the output token may have more weightage than a particular word in the output token. So, in the attention mechanism rather than passing a common context vector $\mathbf{c}$ to all the hidden states in the decoder, we pass a context vector specific to that output token which kind of means that attention has been given to the word.

A new context vector $\mathbf{c}_i$ is generated on each decoding step and takes all of the hidden encoder steps into account in its derivation.

$$\mathbf{c}_i = f(\mathbf{h}_1^e, \mathbf{h}_2^e, \mathbf{h}_3^e, \ldots, \mathbf{h}_n^e)$$

The next hidden state is thus the function of its previous hidden state, the new context,

and the output.

$$\mathbf{h}_t^d = g(\hat{y}_{t-1}, \mathbf{h}_{t-1}^d, \mathbf{c}_t)$$

Now, we need to decide how relevant each encoder step is to the decoder step captured in $\mathbf{h}_{t-1}$. A score for each hidden state $j$ in encoder has to be there, $(\mathbf{h}_{t-1}^d, \mathbf{h}_j^e)$. One such score is **dot-product attention**.

$$\text{score}(\mathbf{h}_{t-1}^d, \mathbf{h}_i^e) = \mathbf{h}_{t-1}^d \cdot \mathbf{h}_i^e$$

This score reflects the similarity between the two states. Now, to use these scores we normalize these scores by softmax function.

$$\alpha_{ij} = \text{softmax}(\text{score}(\mathbf{h}_{i-1}^d, \mathbf{h}_j^e)) = \frac{\exp(\text{score}(\mathbf{h}_{i-1}^d, \mathbf{h}_j^e))}{\sum_{k=1}^{n} \exp(\text{score}(\mathbf{h}_{i-1}^d, \mathbf{h}_k^e))}$$

Hence, the context vector is thus the weighted average of these scores.

$$\mathbf{c}_i = \sum_{k=1}^{n} \alpha_{ik} \cdot \mathbf{h}_k^e$$

With this, we finally have a fixed-length context vector that takes into account information from the entire encoder state that is dynamically updated to reflect the needs of the decoder at each step of decoding.

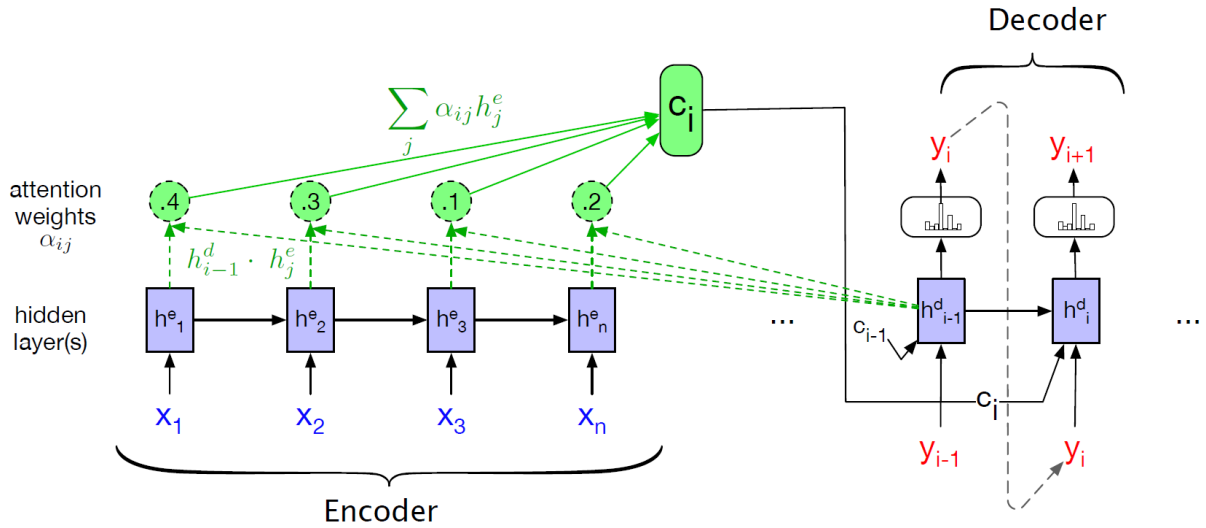


Figure 4.5: A sketch of the encoder-decoder network with attention, focusing on the computation of $\mathbf{c}_i$.

It's also possible to create more sophisticated scoring functions for attention models. Instead of simple dot product attention, we can get a more powerful function that computes the relevance of each encoder hidden state to the decoder hidden state by parameterizing the score with its own set of weights,$\mathbf{W}_s$. The weights $\mathbf{W}_s$, which are then trained during normal end-to-end training, give the network the ability to learn which aspects of similarity between the decoder and encoder states are important to the current

application. This also allows the use of different dimensional hidden states in encoder and decoder networks.

$$\text{score}(\mathbf{h}_{t-1}^d, \mathbf{h}_i^e) = \mathbf{h}_{t-1}^d \mathbf{W}_s \mathbf{h}_i^e$$

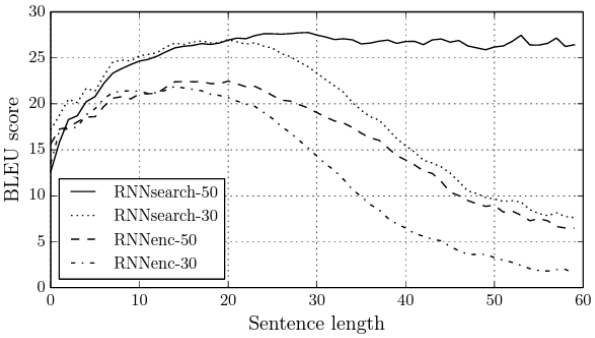


Figure 4.6: The graph from [15] represents the improvement in translation for long sentences after applying attention.

# Plan of Action

- **Week 1:** Introduction to NLP and Text Preprocessing
  - Overview of NLP
  - Tokenization
  - Stop-word removal
  - Stemming and Lemmatization
  - Punctuation removal
- **Week 2:** Text Representation
  - Bag of Words
  - Count Vectorization
  - TF-IDF
  - Word Embeddings (Word2Vec, Doc2Vec)
  - n-gram models (uni-gram, bi-gram, tri-gram)
- **Week 3:** Deep Learning for NLP
  - Neural Networks
  - Recurrent Neural Networks (RNNs)
  - Long Short-Term Memory Networks (LSTMs)
- **Week 4:** Attention Mechanism, Seq2seq, and Encoder-decoder models
  - Encoder-decoder architecture
  - Sequence-to-Sequence (Seq2seq) models
  - Understanding Attention Mechanism
- **Week 5:** Transformers and BERT
  - Introduction to Transformers
  - BERT (Bidirectional Encoder Representations from Transformers)
  - Transfer Learning with BERT
- **Midterm Report Submission:** Late June
- **Week 6:** Large Language Models (LLMs)

  - Understanding LLMs
  - Applications and case studies

- **Week 7:** Applications of NLP
  - Overview of NLP applications
  - Project analysis and workflow explanation
- **Week 8:** Advanced Topics and Additional Materials (if time permits)
  - Exploring additional advanced topics in NLP
  - Reviewing supplementary materials and lectures
- **Week 9:** Final Review and Preparation for Endterm Submission
  - Comprehensive review of all topics
  - Preparation for final report submission

**Final Report Submission:** Late July

# Bibliography


[1] Daniel Jurafsky and James H. Martin. *Speech and Language Processing: An Introduction to Natural Language Processing, Computational Linguistics, and Speech Recognition*. Stanford University and University of Colorado at Boulder, third edition, 2023. Draft edition.

[2] GeeksforGeeks. Introduction to natural language processing. Accessed: 2024-06-25.

[3] TensorFlow YT Channel. Natural language processing (nlp) zero to hero playlist.

[4] CampusX YT Channel. Text representation — nlp lecture 4 — bag of words — tf-idf — n-grams, bi-grams and uni-grams.

[5] CampusX YT Channel. Word2vec complete tutorial — cbow and skip-gram — game of thrones word2vec.

[6] Stanford University School of Engineering YT channel. Lecture 2 — word vector representations: word2vec.

[7] Stanford Online YT channel. Stanford cs224n - nlp w/ dl — winter 2021 — lecture 5 - recurrent neural networks (rnns).

[8] StatQuest YT Channel. Recurrent neural networks (rnns).

[9] StatQuest YT Channel. Long short-term memory (lstm).

[10] CampusX YT Channel. Encoder decoder — sequence-to-sequence architecture — deep learning — campusx.

[11] Ilya Sutskever, Oriol Vinyals, and Quoc V. Le. Sequence to sequence learning with neural networks. *arXiv preprint arXiv:1409.3215*, 2014. Available at `https://arxiv.org/abs/1409.3215`.

[12] StatQuest YT Channel. Sequence-to-sequence (seq2seq) encoder-decoder neural networks.

[13] CampusX YT Channel. Attention mechanism in 1 video — seq2seq networks — encoder decoder architecture.

[14] StatQuest YT Channel. Attention for neural networks.

[15] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. Neural machine translation by jointly learning to align and translate. *arXiv preprint arXiv:1409.0473*, 2014. Presented at ICLR 2015.

[16] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Advances in Neural Information Processing Systems*, pages 5998–6008, 2017.