Final Report
Summer of Science 2024: Natural Language Processing

Mentee: Abhi Jain
23b0903

Mentor: Nilesh Choudhary

July 22, 2024

Contents

Chapter 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 encoded 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 don'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 noise. Hence, removing these less important words to extract only meaningful information from the given text could help.

In general, 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 do not 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 libraries 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 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 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 epitome of imagined punctuations is ignored in many NLP tasks as they do not contain and have no implied meaning in the text. The process of text filtration for 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

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 this 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:** Vocabulary 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 the document.

Let's take an example using which we will be understanding the different techniques. It's just random data created by me. The data contains 4 statements and the outputs are 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 the word in the Vocabulary and 0 everywhere else. Therefore, each document is essentially a vector of vectors i.e., a 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 in sparse storage and computation that if not well handled can cause inefficient use of available resources. 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 resulting in 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, the 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 that 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, and information retrieval in search engines.
- **Non-parametric Nature:** Requires minimal parameter tuning.

Disadvantages

- **Loss of Context:** It ignores 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 the 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 sentences into words we break them into pairs 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 the ability to comprehend context.
- **Flexibility:** The value of 'n' can be changed accordingly to capture more context.

Disadvantages of n-gram Models

- **Increased Dimensionality:** It is obvious that a 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 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 a 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 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 the value of each term in the vector for the document is $Tf(t,d) \cdot Idf(t)$

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 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 it 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 vectors are shorter, with dimensions of d ranging from 50-1000, rather than a 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 as 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 neighbouring 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 skip-gram 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 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. 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.

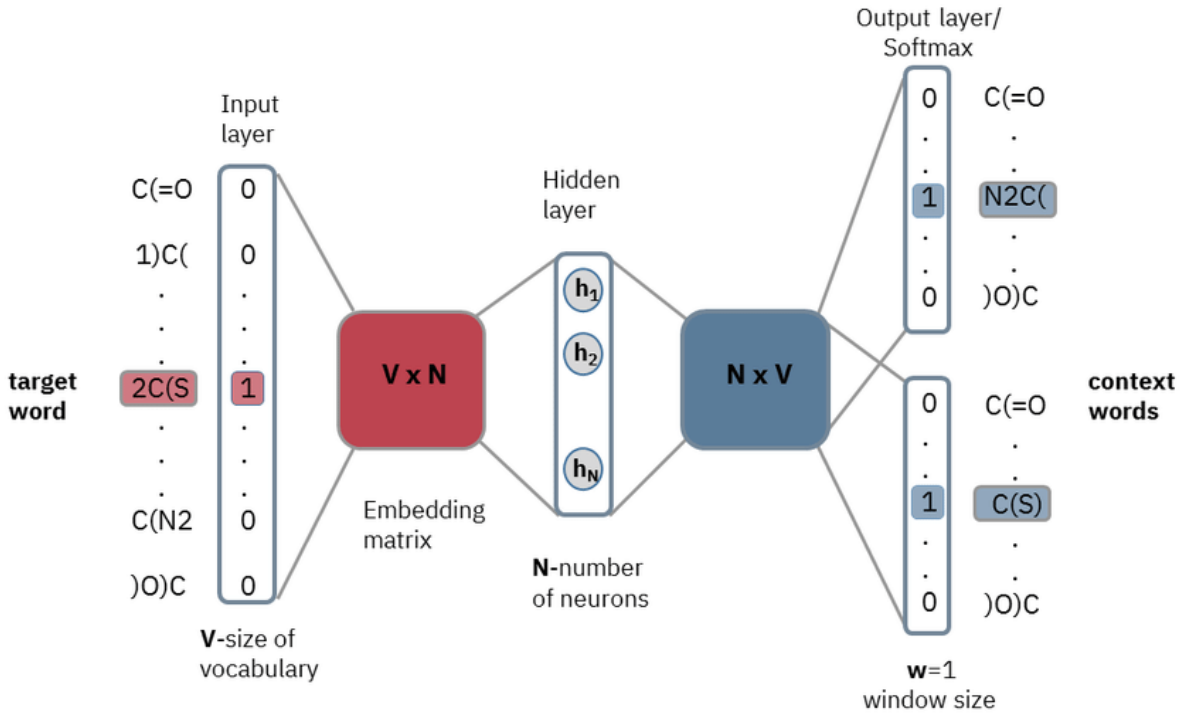


Figure 2.1: Skip-Gram Algorithm, in the figure N is the size of the word - embedding vector, $V \times N$ is the W Matrix and $N \times V$ is the C Matrix

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 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

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.

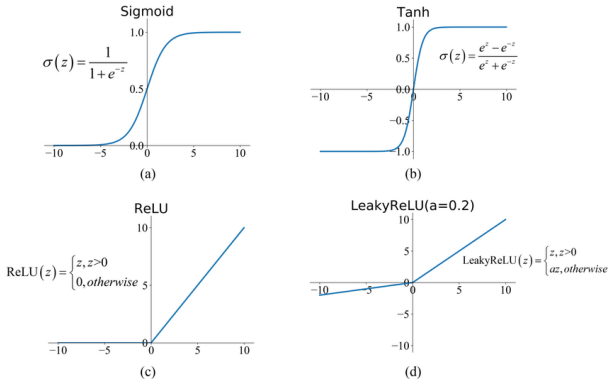


Figure 3.1: Popular Activation functions

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 backwards 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 kinds of nodes: Input Units, Hidden units, Output Units

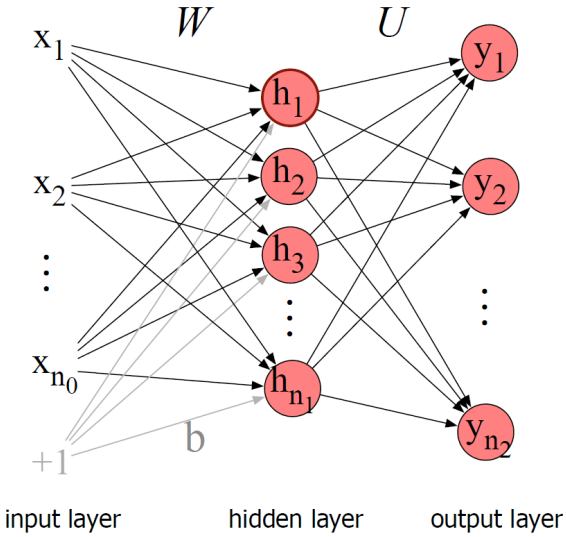


Figure 3.2: A simple 2-layer feedforward network, with one hidden layer, one output layer, and one input layer

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 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 the loss function with respect to any individual w or b but in neural networks, there are millions of parameters in many layers, so 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 operations like addition multiplication explicit then,

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

Therefore, the two derivatives are calculated using the 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.

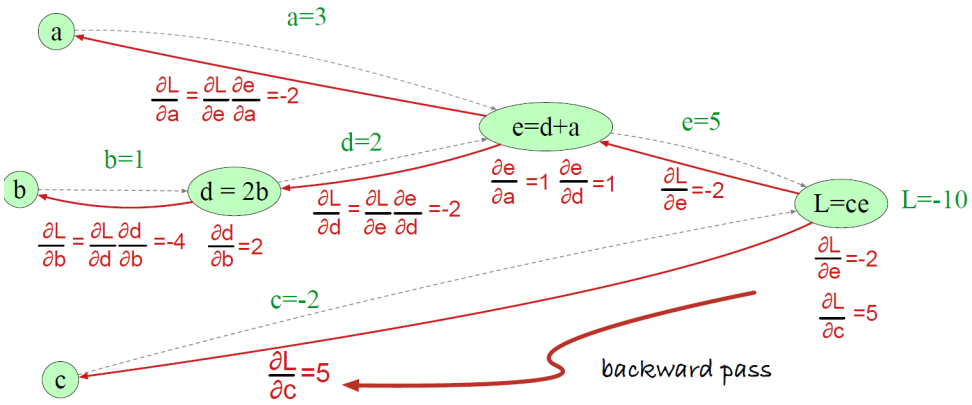


Figure 3.3: Computation graph for the function $L(a, b, c) = c(a + 2b)$, showing the backward pass computation

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 modelling 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 their network connections, allowing some units to be directly or indirectly dependent on earlier outputs.

3.2.1 Forward Inference

We use three weight matrices $\mathbf{W}$,$\mathbf{U}$ and $\mathbf{V}$ 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.

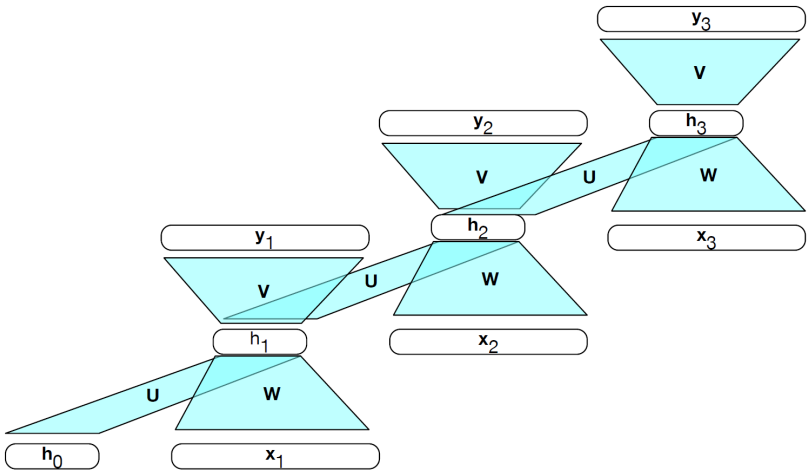


Figure 3.4: Simple RNN shown unrolled in time

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 backwards 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

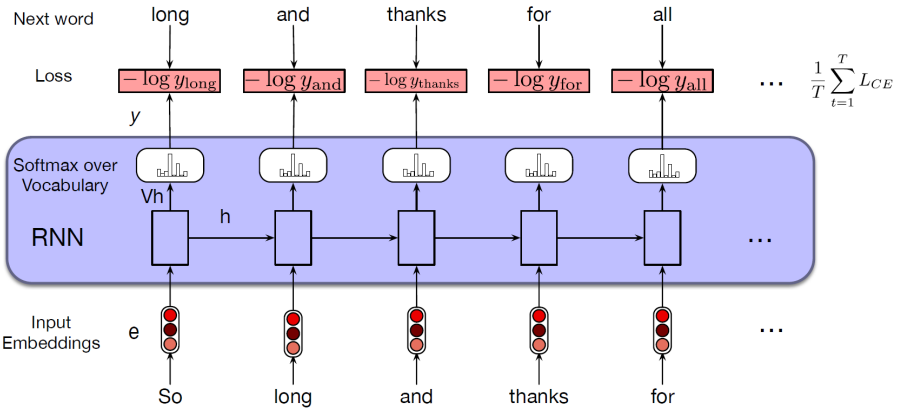


Figure 3.5: Training RNNs as language model

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.

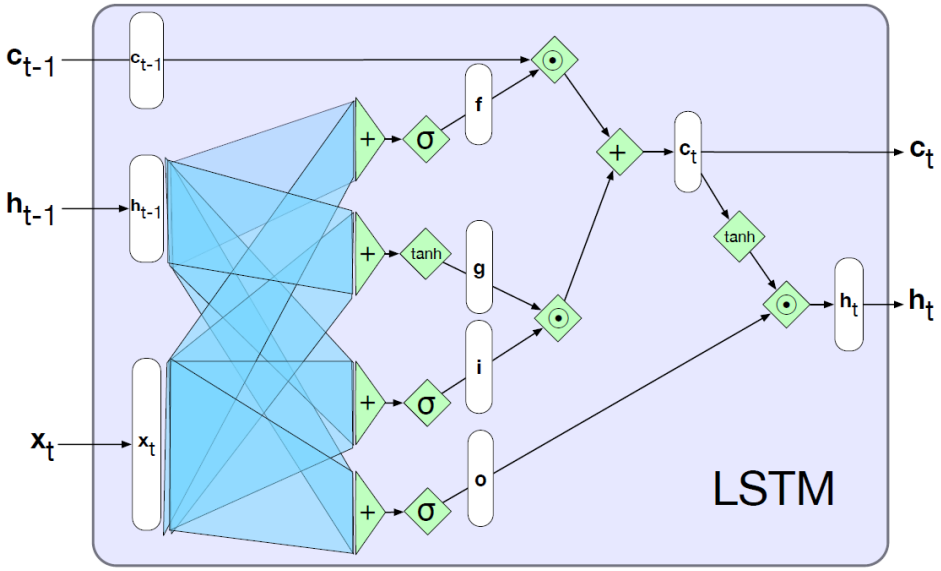


Figure 3.6: Single LSTM unit displayed as a computation graph

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.

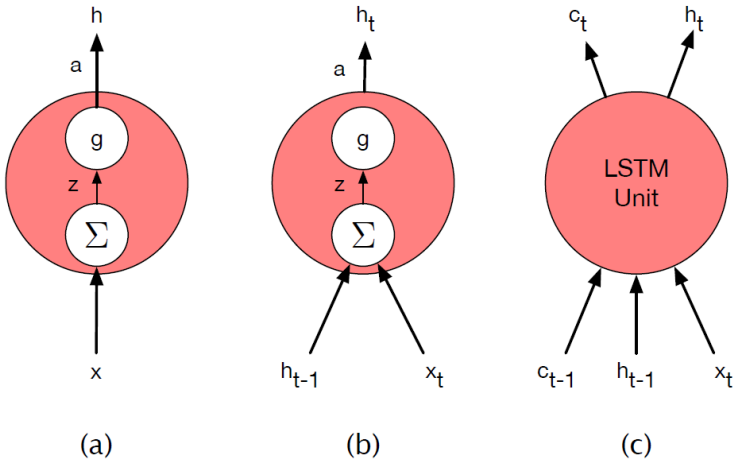


Figure 3.7: Basic neural units used in feedforward, simple recurrent networks (SRN), and long short-term memory (LSTM).

Chapter 4

Attention Mechanism, Seq2seq, and Encoder-decoder Models

4.1 Encoder-decoder and Seq2Seq Models

In these types of models, we take sequences as inputs and output a sequence only, the best example of this is language translation. The problem with this is that both the input and output sizes are variable in length. So, let us take the example of language translation and continue our understanding.

Since language translation does not 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 the 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.

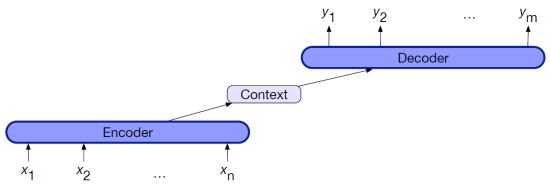


Figure 4.1: Encoder-Decoder architecture

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 in 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 modelling, 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:

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

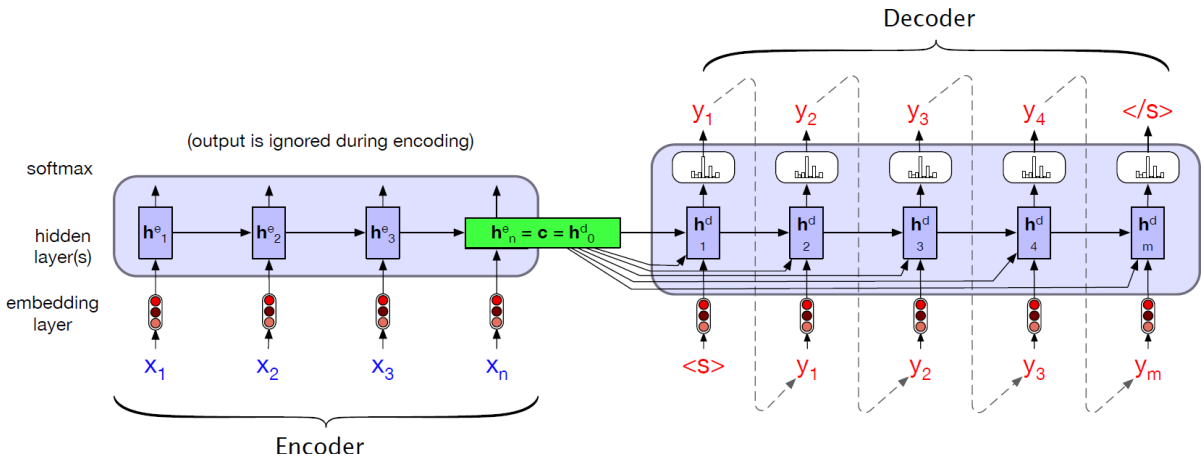


Figure 4.2: Translating a sentence at inference time in the basic RNN-based encoder-decoder architecture.

To translate the sentence through such networks, let us understand the architecture and forward inference of the network through the 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 the 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 the 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 arg max 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

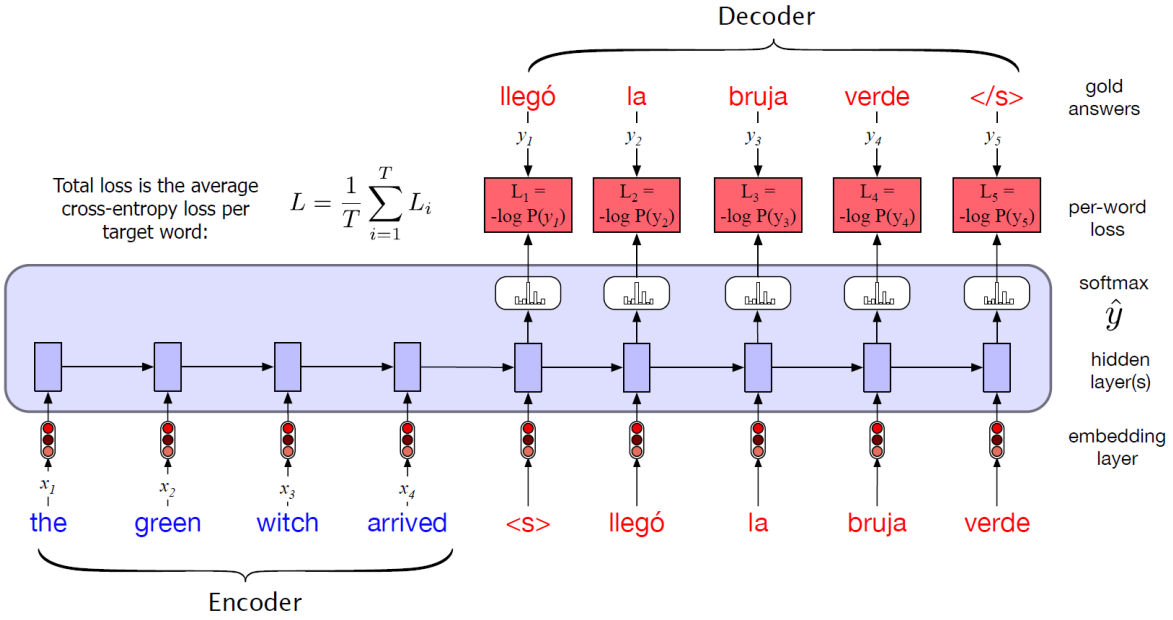


Figure 4.3: Training the basic RNN encoder-decoder approach to machine translation.

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 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 uses gates similarly.

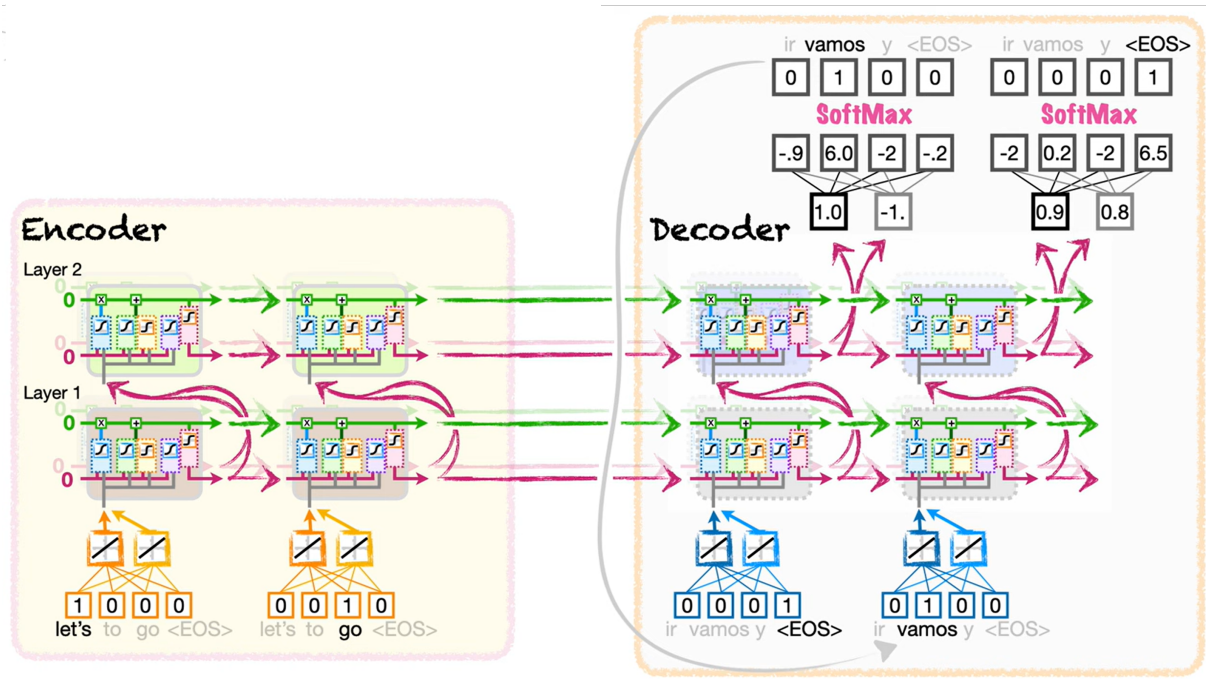


Figure 4.4: A Unfolded Encoder-decoder structure showing the translation of "let's go" in English to "vamos" in Spanish, $< EOS >$ represents the end of sentence

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 [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 layer k-1 forms the input for the context vectors for 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 meaning, it is just about obtaining the output by training the weights. It is 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. In addition, 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 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 in each decoding step and takes into account all the hidden encoder steps 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 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 the encoder has to be there, $(\mathbf{h}^d_{t-1}, \mathbf{h}^e_j)$. One such score is **dot product attention**.

$$\text{score}(\mathbf{h}^d_{t-1}, \mathbf{h}^e_i) = \mathbf{h}^d_{t-1} \cdot \mathbf{h}^e_i$$

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

$$\alpha_{ij} = \text{softmax}(\text{score}(\mathbf{h}^d_{i-1}, \mathbf{h}^e_j)) = \frac{\exp(\text{score}(\mathbf{h}^d_{i-1}, \mathbf{h}^e_j))}{\sum_{k=1}^{n} \exp(\text{score}(\mathbf{h}^d_{i-1}, \mathbf{h}^e_k))}$$

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

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

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.

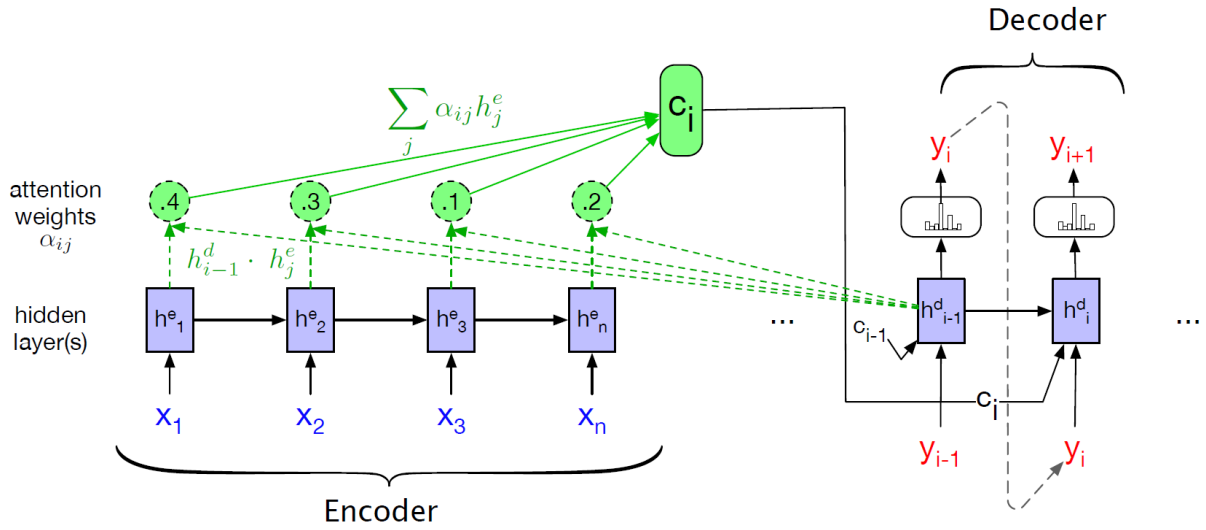


Figure 4.5: A sketch of the encoder-decoder network with attention, focusing on the computation of $\mathbf{c}_i$.

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.

$$\text{score}(\mathbf{h}^d_{t-1}, \mathbf{h}^e_i) = \mathbf{h}^d_{t-1} \mathbf{W}_s \mathbf{h}^e_i$$

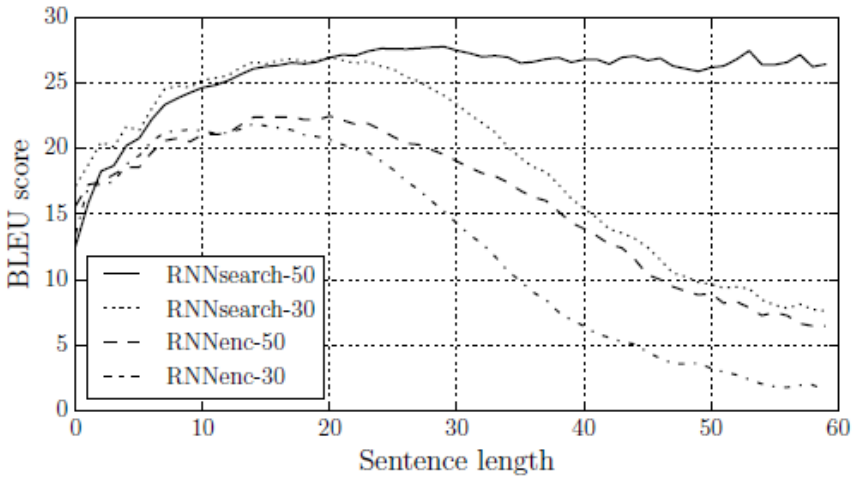


Figure 4.6: The graph from [15] represents the improvement in translation for long sentences after applying attention.

Chapter 5

Transformers

The seq2seq models are mostly based on complex RNNs and LSTMs involving encoder-decoder architecture. Even out of those, the best-performing models are based on an attention mechanism, connecting the encoder-decoder through attention. In the paper [16] a new model, the transformer was introduced which was simply based on attention.

The intuition for the transformer was based on how a human being learns. While reading a novel or a text there are often cases where we don't understand the exact word but still be able to grasp the meaning of the word based on the context. So, this idea of learning is called *distributional hypothesis*. Thus, we can learn word meanings without any grounding in the real world.

In the next chapter, we will see LLMs, which came into the limelight with *ChatGPT*. LLMs are pre-trained language models that learn knowledge about the language and the world from vast amounts of text. LLMs have been very transformative for tasks where we need to produce text, summarization, machine translation, question-answering or chatbots.

5.1 Transformers

The LLMs are based on the standard architecture of transformers. Transformer makes use of **self-attention**. Self-attention can be thought of as a way to build contextual representations of the meaning of a word that integrate information from surrounding words, helping the model to learn how words relate to each other in large text spans. We will see how to apply this to auto-regressive language models.

Transformers are made up of stacks of blocks, each of which maps an input sequence to an output sequence of the same length. These blocks are made by combining simple linear layers, feed-forward layers and self-attention layers.

5.1.1 Transformers: the intuition

The intuition of a transformer is that, in a series of layers, we build up richer and richer contextualized representations of the meanings of input words or tokens. So, in each layer of the transformer to compute the representation of a word, we use the representation of the word and its neighbours in the previous layer.

Now, consider the following sentence: The papers for submission are on the table. The word shows the linguistic relationship with the word papers. So, self-attention is just such a mechanism: it allows us to look broadly at the context and tells us how

to integrate the representation of words in that context from layer $k-1$ to build the representation for words in layer k.

The concept of context can be used in two ways in self-attention. In causal, or backward-looking self-attention, the context is any of the prior words. In general bidirectional self-attention, the context can include future words. In this chapter, we focus on causal, backward-looking self-attention.

5.1.2 Self-attention - formally

Let's say we have an input-sequence $\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_n$ and need to predict the output-sequence $\mathbf{a}_1, \mathbf{a}_2, \ldots, \mathbf{s}_n$. So, let us say I need to predict $\mathbf{a}_4$. Therefore the idea of self attention asks me to compute $\mathbf{a}_4$ based on the similarity of $\mathbf{x}_4$ with $\mathbf{x}_1$, $\mathbf{x}_2$, $\mathbf{x}_3$ and $\mathbf{x}_4$ itself.

Version 1

Here, We use the idea of dot-product similarity between $\mathbf{x}_i$ and $\mathbf{x}_j$.

$$\text{score}(\mathbf{x}_i, \mathbf{x}_j) = \mathbf{x}_i \cdot \mathbf{x}_j$$

We normalise these scores to get the $\mathbf{a}_i$ as the weighted average of input tokens.

$$\alpha_{ij} = \text{softmax}(\text{score}(\mathbf{x}_i, \mathbf{x}_j)) = \frac{\exp(\text{score}(\mathbf{x}_i, \mathbf{x}_j))}{\sum_{k=1}^{i} \exp(\text{score}(\mathbf{x}_i, \mathbf{x}_k))}$$

$$\mathbf{a}_i = \sum_{j=1}^{i} \alpha_{ij} \mathbf{x}_j$$

This kind of attention can be simple and we applied the same in the encoder-decoder architecture of LSTMs in the previous chapter. But the transformer allows us to use more sophisticated way.

Version 2

The transformer uses three matrices to embed the input word into three embeddings namely:

- **Query:** When we have to compare the similarity with the preceding inputs.
- **Key:** When in the role of preceding input, it has to be compared with the current focus of attention.
- **Value:** Used to compute the output for the current focus of attention.

We will have a dimension d for the input vector, d_k for the key and query vectors, and d_v for the value vector. [1]Therefore, we define three weight matrices to transform the input into these roles as $\mathbf{W^Q} \in \mathbb{R}^{d \times d_k}$, $\mathbf{W^K} \in \mathbb{R}^{d \times d_k}$ and $\mathbf{W^V} \in \mathbb{R}^{d \times d_v}$. These weights are then applied to each of the input tokens as follows:

$$\mathbf{q}_i = \mathbf{x}_i \mathbf{W^Q}; \; \mathbf{k}_i = \mathbf{x}_i \mathbf{W^K}; \; \mathbf{v}_i = \mathbf{x}_i \mathbf{W^V}$$

[1]In the original manuscript [16], the d was 512, d_k and d_v were both 64.

Now, the score while producing $\mathbf{a}_i$ is calculated as:

$$\text{score}(\mathbf{x}_i, \mathbf{x}_j) = \frac{\mathbf{q}_i \cdot \mathbf{k}_j}{\sqrt{d_k}}$$

The score is not just the dot product but is also divided by a factor related to the size of embeddings. This is done to scale down the dot-product because the dot-product can take large values and exponenting large-values can lead to numerical issues and can lead to significant loss of gradient while back-propagating.

$$\alpha_{ij} = \text{softmax}(\text{score}(\mathbf{x}_i, \mathbf{x}_j)) = \frac{\exp(\text{score}(\mathbf{x}_i, \mathbf{x}_j))}{\sum_{k=1}^{i} \exp(\text{score}(\mathbf{x}_i, \mathbf{x}_k))}$$

$$\mathbf{a}_i = \sum_{j=1}^{i} \alpha_{ij} \mathbf{x}_j$$

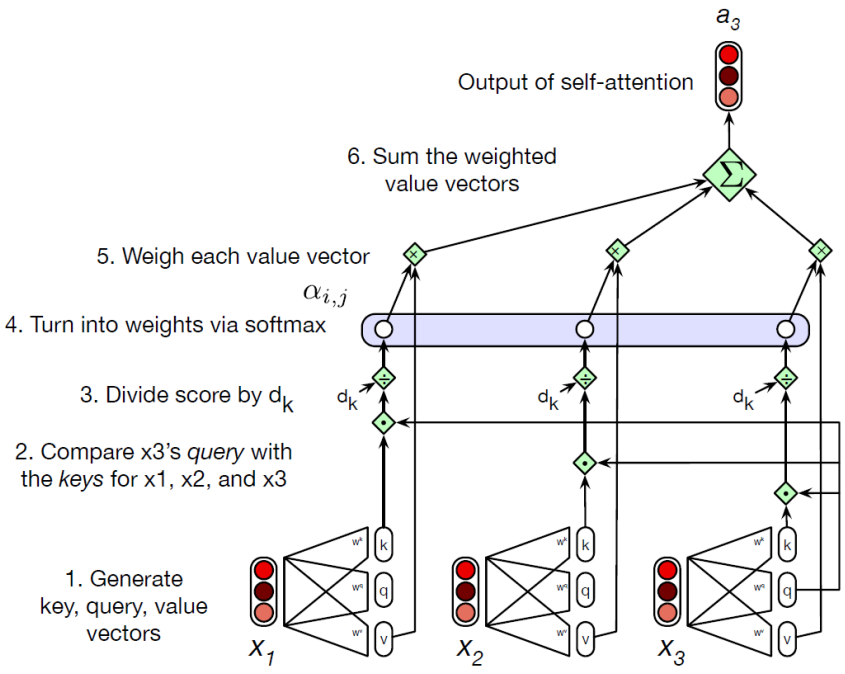


Figure 5.1: Calculating the value of a_3, the third element of a sequence using causal (left-to-right) self-attention

5.1.3 Using the power of Parallelisation with matrices

Since each output, is computed independently, this entire process can be parallelized, taking advantage of efficient matrix multiplication routines by packing the input embeddings of the N tokens of the input sequence into a single matrix $\mathbf{X} \in \mathbb{R}^{N \times d}$. We can then multiply this to key, query, weight matrices (all of dimension $d \times d$) to produce $\mathbf{Q} \in \mathbb{R}^{N \times d}$, $\mathbf{K} \in \mathbb{R}^{N \times d}$ and $\mathbf{V} \in \mathbb{R}^{N \times d}$. These matrices contain all the key, query and value vectors.

$$\mathbf{Q} = \mathbf{X}\mathbf{W}^{\mathbf{Q}}; \mathbf{K} = \mathbf{X}\mathbf{W}^{\mathbf{K}}; \mathbf{V} = \mathbf{X}\mathbf{W}^{\mathbf{V}}$$

Therefore, all query-key comparisons can be simultaneously performed by multiplying $\mathbf{Q}$ and $\mathbf{K}^T$ to form a $N \times N$ Matrix. The entire process has been reduced to the following computation:

$$\mathbf{A} = \text{SelfAttention}(\mathbf{Q}, \mathbf{K}, \mathbf{V}) = \text{softmax}\left(\frac{\mathbf{Q}\mathbf{K}^T}{\sqrt{d_k}}\right)\mathbf{V}$$

Since this process of simultaneous computation can be performed using the powers of GPU. This process can be made a lot faster compared to the recursive approach that we saw in the previous chapter.

Masking out the future

[] In the matrix product calculation of $\mathbf{Q}$ and $\mathbf{K^T}$, the query-key products are calculated for even the keys following the query, which isn't correct in the setting of language modelling. To fix this values of the upper triangular matrix are zeroed out i.e, made $(-\infty)$

$$\mathbf{QK^T} = \begin{bmatrix} \mathbf{q}_1 \cdot \mathbf{k}_1 & -\infty & -\infty & -\infty & -\infty \\ \mathbf{q}_2 \cdot \mathbf{k}_1 & \mathbf{q}_2 \cdot \mathbf{k}_2 & -\infty & -\infty & -\infty \\ \mathbf{q}_3 \cdot \mathbf{k}_1 & \mathbf{q}_3 \cdot \mathbf{k}_3 & \mathbf{q}_3 \cdot \mathbf{k}_3 & -\infty & -\infty \\ \mathbf{q}_4 \cdot \mathbf{k}_1 & \mathbf{q}_4 \cdot \mathbf{k}_4 & \mathbf{q}_4 \cdot \mathbf{k}_4 & \mathbf{q}_4 \cdot \mathbf{k}_4 & -\infty \\ \mathbf{q}_5 \cdot \mathbf{k}_1 & \mathbf{q}_5 \cdot \mathbf{k}_5 & \mathbf{q}_5 \cdot \mathbf{k}_5 & \mathbf{q}_5 \cdot \mathbf{k}_5 & \mathbf{q}_5 \cdot \mathbf{k}_5 \end{bmatrix}$$

5.2 Multi-head Attention

Transformers in reality compute a more complex kind of attention rather the single self-attention. This is because the different words relate to each other simultaneously in many ways, for example: syntactic, semantic, and discourse relationships can hold between the words in a sentence. A single self-attention model is not capable of learning all that. Hence, transformers use multihead self-attention layers. These layers reside in parallel at the same depth in a model. Each head has its own set of weight matrices.

$$\mathbf{Q} = \mathbf{X}\mathbf{W}_\mathbf{i}^\mathbf{Q};\ \mathbf{K} = \mathbf{X}\mathbf{W}_\mathbf{i}^\mathbf{K};\ \mathbf{V} = \mathbf{X}\mathbf{W}_\mathbf{i}^\mathbf{V}$$

$$\mathbf{head}_i = \text{SelfAttention}(\mathbf{Q}, \mathbf{K}, \mathbf{V})$$

Let us consider a model with h heads[2]. Now, each of the h heads is of shape $N \times d_v$. Therefore, the output of these matrices contains h such matrices, which can be concatenated to form a matrix of size $N \times hd_v$. Finally, we use another linear projection $\mathbf{W^O} \in \mathbb{R}^{hd_v \times d}$ that when multiplied with the concatenated matrix gives the final self-attention output of shape $N \times d$, suitable for passing through residual connections and layer norms.

$$\mathbf{A} = (\mathbf{head}_1 \oplus \mathbf{head}_2 \cdots \oplus \mathbf{head}_h)\mathbf{W^O}$$

[2]In the original paper [16] h = 8

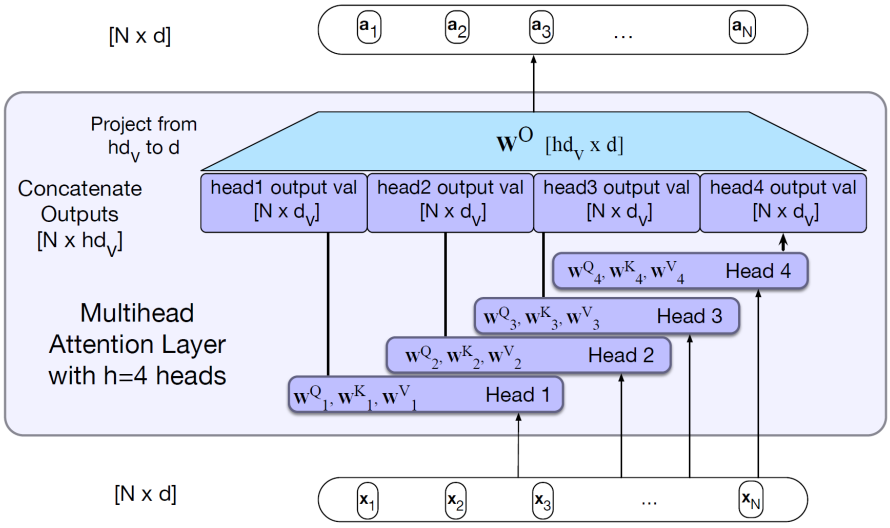


Figure 5.2: Multi head-attention using 4 heads

5.3 Transformer Blocks

The self-attention lies at the core of the transformation block, which in addition to the self-attention layer, includes 3 other kinds of layers:

1. Feedforward layer
2. Residual Connections
3. Layer Norm

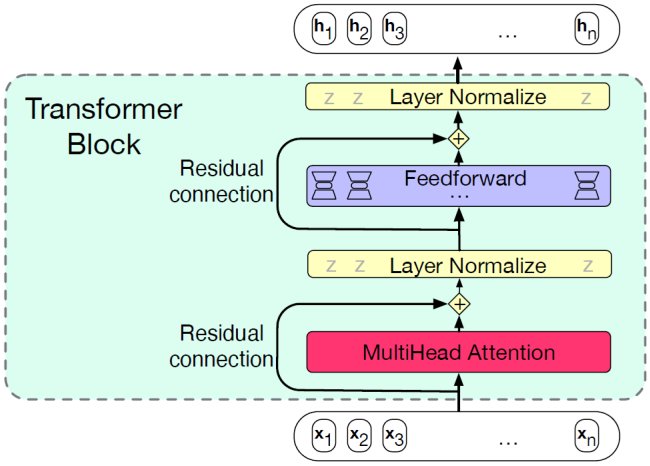


Figure 5.3: A transformer block showing all layers

Feedforward Layer

The feed-forward layer has N-position-wise networks, one for each position. Each is a fully connected network. The weights are the same for each position, but parameters are different from layer to layer. The feedforward networks can be computed in parallel as

they are independent for each position. It's common to make the dimensionality of the hidden layer d_H to be larger than the model dimensionality d. [3]

Residual Connections

They are the connections going from a lower layer to a higher layer without going through the intermediate layers. Since the information from a lower layer can go directly into a higher layer, the activation and backward propagation of errors have to skip a step which improves learning and has given better results. And, the higher layer has the advantage as it has access to the lower layer directly. The residual connections are just added, i.e., a layer's input vector is simply added to its output vector before passing it forward.

Layer Norm

These summed vectors are needed to be normalised to improve the training performance. Layer normalisation (also called Layer Norm) is used in deep neural networks to improve training performance by keeping the values of gradient in a range that facilitates gradient-based learning.

The layer norm takes an input of dimension d, then computes the mean and standard deviation to transform it into a vector with zero mean and standard deviation 1. The layer can be transformed as follows:

$$\mu = \frac{1}{d}\sum_{i=1}^{d} x_i$$

$$\sigma = \sqrt{\frac{1}{d}\sum_{i=1}^{d}(x_i - \mu)^2}$$

$$\hat{\mathbf{x}} = \frac{\mathbf{x} - \mu}{\sigma}$$

The standard implementation of layer normalisation introduces two learnable parameters γ and β, representing gain and offset values.

$$\text{LayerNorm} = \gamma\hat{\mathbf{x}} + \beta$$

Putting it all together

The function computed by a transformer block is simply:

$$\mathbf{O} = \text{LayerNorm}(\mathbf{X} + \text{SelfAttention}(\mathbf{X}))$$

$$\mathbf{H} = \text{LayerNorm}(\mathbf{O} + \text{FFN}(\mathbf{O}))$$

The input and output dimensions of the transformer blocks match as both $\mathbf{X}$ and $\mathbf{H}$ are of dimension $N \times d$, so they can be stacked. Transformers for large language models stack many of these blocks, from 12-layers (used in GPT-3 small language models) to 96-layers (used in GPT-3 large) to even more for recent models.

[3] In orginal model $d_H = 2048$ and $d = 512$.

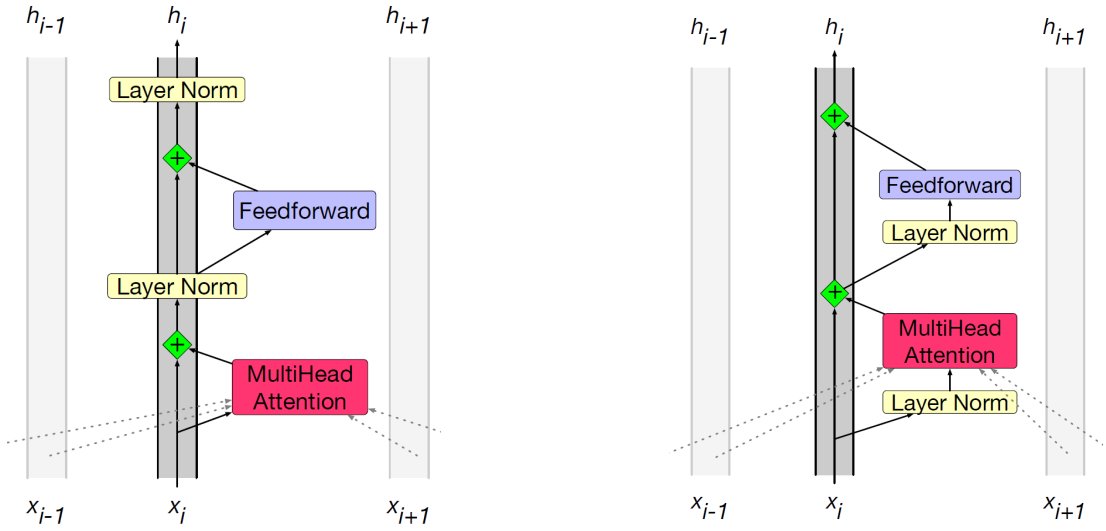
5.4 Residual Stream view of the Transformer block

The processing of the individual token through these layers as a stream of d-dimensional vectors is called the residual stream as in figure 5.4. The residual layers are constantly copying information up from earlier embeddings, so we can think of the other components as adding new views of this representation back into this constant stream. Feedforward networks add in a different view of the earlier embedding. We can view attention heads as literally moving attention from the residual stream of a neighbouring token into the current stream. In the earlier transformer blocks, the residual stream represents the current token. In the highest transformer blocks, the residual stream is usually the next token, as at the end it is being trained to predict the next token.

Pre-norm vs Post-norm

There is an alternative form of the transformer architecture that is commonly used because it performs better in many cases. In the prenorm transformer architecture, the layer norm happens in a slightly different place: before the attention layer and before the feedforward layer, rather than afterwards as in the figure 5.4(b). The one we discussed through back was the post-norm architecture.

The pre-norm transformer has one extra requirement: at the very end of the last (highest) transformer block, there is a single extra layer norm that is run on the last $\mathbf{h}_i$ of each token stream



(a) Post-norm architecture

(b) Pre-norm architecture

Figure 5.4: Nature of residual stream through the two types of transformer architectures

5.5 The input: embeddings for token and position

The input matrix $\mathbf{X}$ of dimensions $N \times d$ contains word embeddings for each word from context. The transformer computes two embeddings, input token embedding and input positional embedding. The embedding is similar to what we have learned in the Word2Vec representations. As we pass this embedding through different layers and transformation

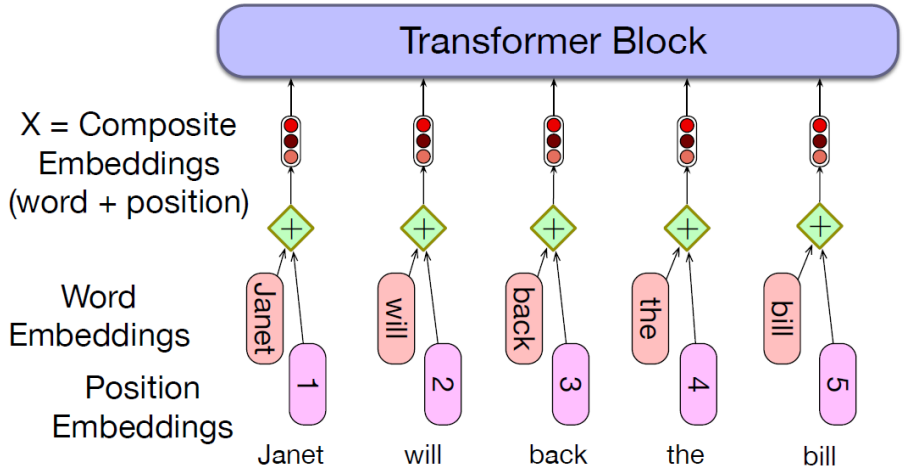


Figure 5.5: Adding positional embeddings and word embeddings to form the matrix $\mathbf{X}$

blocks, it keeps on changing, incorporating the context and depending on the kind of language model we are building.

For, token encoding we can use the one-hot vectors from the vocabulary and the embedding matrix to compute the token embeddings for each token.

To represent the position of each token in the sequence we combine these token embeddings with the positional embeddings. The final input matrix is an $N \times d$ in which each row is the representation of i^{th} token in the input, computed by adding $\mathbf{E}[id(i)]-$ the embedding of the id of the token that appeared at position i, to $\mathbf{P}[i]-$the positional embedding of position i. There is a problem with the absolute positional embedding approach in the training. These embeddings may be poorly trained and cannot be generalised during testing. An alternative approach to absolute positional embeddings is to choose a static function that maps integer inputs to real-valued vectors in a way that captures the inherent relationships among the positions. It may allow the model to extrapolate to sequences longer than the ones encountered during training. The original transformer work used a combination of sine and cosine functions with differing frequencies.

5.6 Summarizing the Transformer Architecture

- The encoder in the transformer takes an input sequence of the form $x_1, x_2, \ldots, x_n$ and converts it into a sequence of the form $z_1, z_2, \ldots, z_n$. The decoder uses this sequence to generate the output sequence of the form $y_1, y_2, \ldots, y_m$
- At each step the model is auto-regressive, consuming the previously generated symbols as additional input when generating the next.
- In the encoder, the sequence is first converted into word embeddings using the embedding matrix which when combined with positional embeddings are passed on into the transformer blocks.

Encoder

- The encoder contains a stack of N[4] transformer blocks. Each block has two sub-layers.
- The first is a multi-head self-attention. Each input token has self-attention linked to all the other input tokens, as in this case, we know all the input tokens already.
- The next sub-layer in the transformer block is a fully connected feed-forward neural network.
- There are residual connections around each of the two sub-layers, which then are followed by layer normalisation

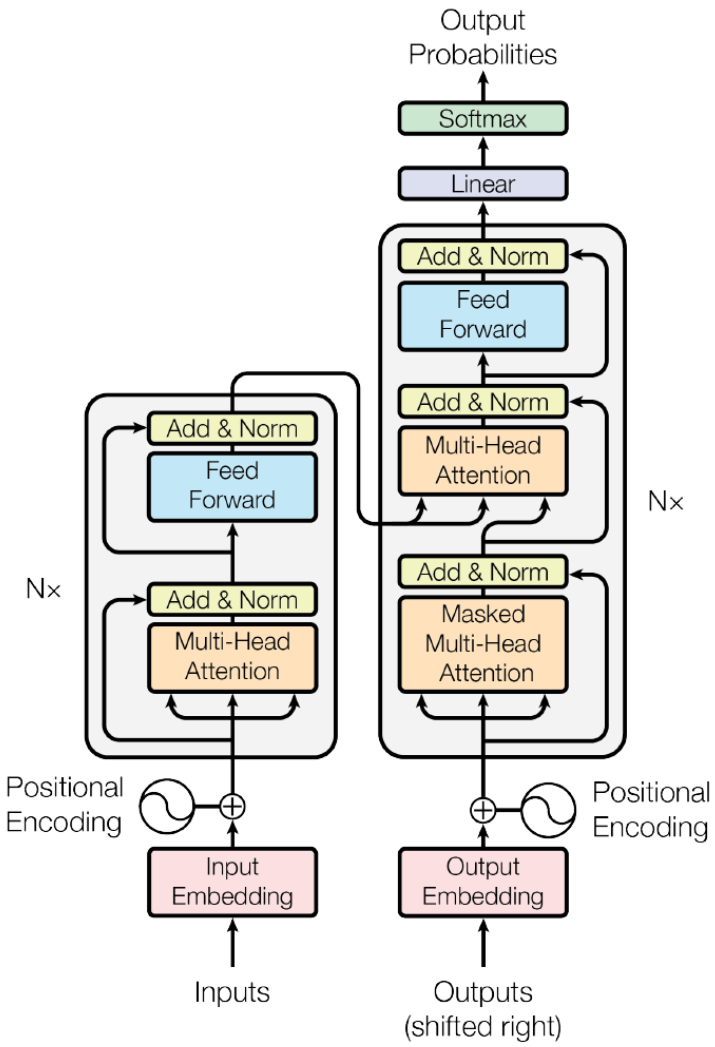


Figure 5.6: The transformer-model architecture

Decoder

- The decoder is also composed of N identical transformer blocks. The transformer blocks in the decoder are composed of three sub-layers.

[4]In the original paper, N was 6 both in encoder and decoder

- The first sub-layer is again a multi-head attention but since we don't know the upcoming output values. Therefore, we use masked attention as we discussed in 5.1.3 to ensure that token i is predicted from the tokens less than i.
- The second sub-layer performs attention over the output sequence $z_1, z_2, \ldots, z_n$ from the encoder and then passes it into the feed-forward sub-layer.
- Similar to the encoder, the residual connections and Layer normalisation are applied.

The sequence received after passing through transformer blocks is then transformed using Language Modelling as per the requirements of the model.

5.7 The Language Modeling Head

The last component of the Transformer is the language modeling head which is the circuitry we need to do language modeling. The language modelling allows us to give conditional probabilities like $P(w_{N+1}|w_{1:N})$.

The job of the language modelling head is to take the output of the final transformer layer ($\mathbf{h_N^L} \in \mathbb{R}^d$) from the last token N and use it to predict the upcoming word at position N +1. For this purpose, it uses the unembedding layer ($\mathbf{E}^T \in \mathbf{R}^{d \times |V|}$) to produce the logit vector ($\mathbf{u} \in \mathbb{R}^{|V|}$). The logit vector is then followed by the softmax function to generate a probability distribution over the entire vocabulary, as we have done in the earlier seq2seq models.

$$\mathbf{u} = \mathbf{h_N^L E}^T$$

$$\mathbf{y} = \text{softmax}(\mathbf{u})$$

We can use this probability in various ways, one of usage is in generating text which we will do by sampling.

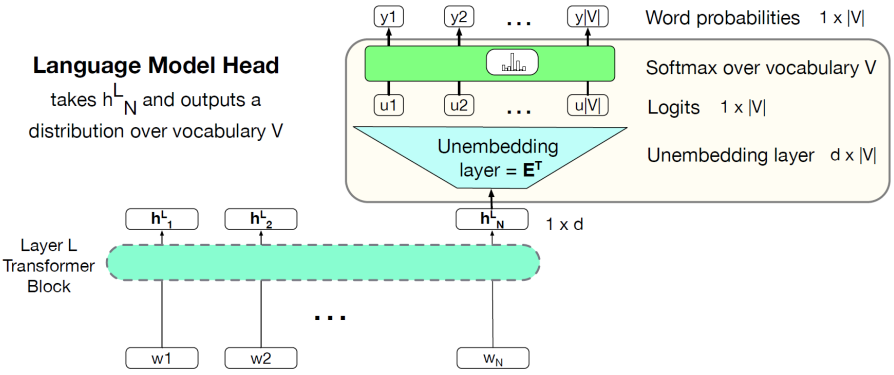


Figure 5.7: The language modelling head: the circuit at the top of a transformer

Chapter 6

LLMs with Transformers

Since we have developed a basic model for transformers, thus, now we will look at how we use it in language modelling tasks. We will look into some general things and then move into sampling and training of transformers. The Bi-directional transformers, transfer learning, and fine-tuning of transformers. These aspects have been covered in the next chapter. They were planned before this, but when I started studying the transformer and moved on bi-directional, it required the basic knowledge of LLMs since we had a buffer week in this report. That portion has been covered later.

6.1 Brief

Transformers were first introduced in the 2017 paper [16]. The focus of the original paper was on translation tasks, but then several influential models followed that architecture.

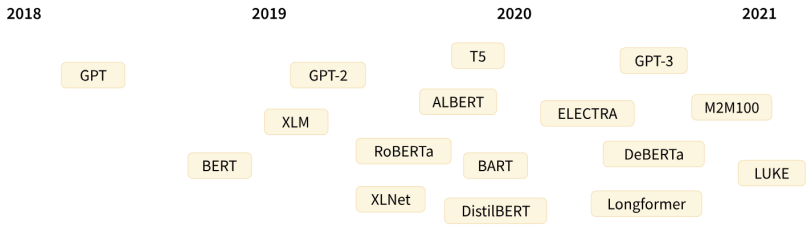


Figure 6.1: Short History of Transformer Models

Broadly the transformers can be classified into 3-categories:

- GPT-like (also called *auto-regressive* Transformer models)
- BERT-like (also called *auto-encoding* Transformer models)
- BART/T5-like (also called *sequence-to-sequence* Transformer models)

6.1.1 Applying Transformer Models to NLP tasks

All the NLP tasks are just examples of conditional generation. This task is conditioned on an input of text or prompt.

The ability of transformers to read the previous context and the generated output and to maintain it over a large context(even up to 1024 or 4096 tokens) makes them powerful.

Why do we need to predict the upcoming words? The idea is that almost all NLP tasks can be cast as word prediction. For-example:

- To see the sentiment of the sentence *"I love cricket"*, we need to compare the conditional probability of the words *"positive"* and *"negative"* to see which is higher.

$$P(\textit{positive}|\textit{The sentiment of the sentence "I love cricket" is:})$$

$$P(\textit{negative}|\textit{The sentiment of the sentence "I love cricket" is:})$$

- We can even ask questions, for example: "Who is the first PM of India?", this can be modelled by looking into the probability of the word w as:

$$P(w|\textit{"Q: Who is the first PM of India? A:"})$$

  By seeing which words w have higher probability we can expect w to be *"Jawahar"*, then we will look at:

$$P(w|\textit{"Q: Who is the first PM of India? A: Jawahar"})$$

  and so on we will have our answer *"Jawahar Lal Nehru"*

- Conditional generation can be used even for tasks that generate longer responses. For example: **text summarization**. We can cast summarization as language modelling by giving a large language model a text, and following the text with a token like tl;dr; this token is short for something like 'too long; don't read' and in recent years people often use this token, especially in informal work emails, when they are going to give a short summary. We can then do conditional generation: give the language model this prefix, and then ask it to generate the following words, one by one, and take the entire response as a summary.

6.1.2 Greedy Decoding

How to decide which word to generate on each step? One intuitive idea could be to just take the word with maximum probability i.e., being greedy for each next token.

$$\hat{w}_t = \text{argmax}_{w \in V} P(w|\mathbf{w}_{<t})$$

But the problem with this idea is that it will always take the word that is locally most optimal, not necessarily be the best choice for the overall text. In practice, we don't generally use greedy encoding as with it the generated text is extremely predictable, and the resulting text is generic and often quite repetitive. It is very deterministic, if the context is identical and the probabilistic models are identical then it always generates the same text.

In machine translation, we use a modified version of greedy encoding called beam search.

In practice, we prefer to use more sophisticated methods, called sampling, that introduce a bit more diversity into the generations.

6.2 Generation by Sampling

The idea of text generation is to choose a word based on the probabilities assigned by the model. This process is often called decoding. Decoding in a left-to-right manner, and generating the next word is called **auto-regressive generation** or **casual LM generation**.

One of the intuitive ideas that we saw was greedy decoding, another very intuitive idea is **random sampling** where we generate the next word based on the probabilities assigned by the model. So, at each step, we will sample words according to their probability conditioned on our previous choices. And among these words choose randomly with their probability till we don't get ¡EOS¿ token.

There is a higher chance that we will generate contextually important next word, but still, we have rare words in our sequence. This gives us diversity, but having rare words in the sentences and there can be many of them, decreases the quality of output.

So, while sampling we have to ensure the right trade-off between quality and diversity. If we only go on to choose the highest probable words, then we will end up producing factually correct, but boring and repetitive outputs. Methods that tend to take middle words, could produce diversity but can be less-factual, incoherent or low-quality.

6.2.1 Top-k Sampling

The method of top-k sampling works as follows:

1. Choose in advance a number $= k$.
2. For each word in the vocabulary, use the language model to calculate the probability $P(w|\mathbf{w}_{<t})$
3. Now, sort the probability to get the top-k words.
4. Renormalize the probability distribution using the current probabilities.
5. Now, do as we did in random sampling by choosing the words according to their probability.

$k = 1$ is what greedy encoding is. Setting an appropriate k value can lead us to generate words that are not highly probable but still quite probable, finally leading to the generation of diverse high-quality text.

6.2.2 Nucleus or Top-p Sampling

The problem with the probability of top k is that it chooses a fixed number. Now, top-k words may account for the maximum probability mass, but they might not account for enough probable mass, e.g.: when the probability distribution is flatter.

So, the idea is not to keep the top k words, but to keep the words that account for the top-p per cent of the probability mass. The rest of the idea is the same. Hence, we choose the smallest set of $V^{(p)}$ words over the distribution such that:

$$\sum_{w \in V^{(p)}} P(w|\mathbf{w}_{<t}) > p$$

6.2.3 Temperature Sampling

In temperature sampling, we don't truncate the probability distribution. Rather, we take the idea of thermodynamics. We know that at higher temperatures, a system is very flexible and can exist in many states, while a system at a lower temperature will exist in just a few low-energy states. We have seen the degrees of freedom of gases change with temperature.

So, in low-temperature sampling, we simply increase the probability of higher-probability words and decrease the probability of the rare words. In high-temperature sampling, as is obvious, we do the opposite. We simply achieve this by dividing the logit $\mathbf{u}$ used to generate probabilities before softmax by τ.

$$\mathbf{y} = \text{softmax}\left(\frac{\mathbf{u}}{\tau}\right)$$

For low-temperature sampling, $\tau \in (0, 1]$ and for high-temperature sapling, $\tau > 1$. For $\tau \to 0$, the method is the same as a greedy encoding (vibes of 0K).

6.3 Training Transformers

How does a transformer learn? Using what algorithms and data do we train it?

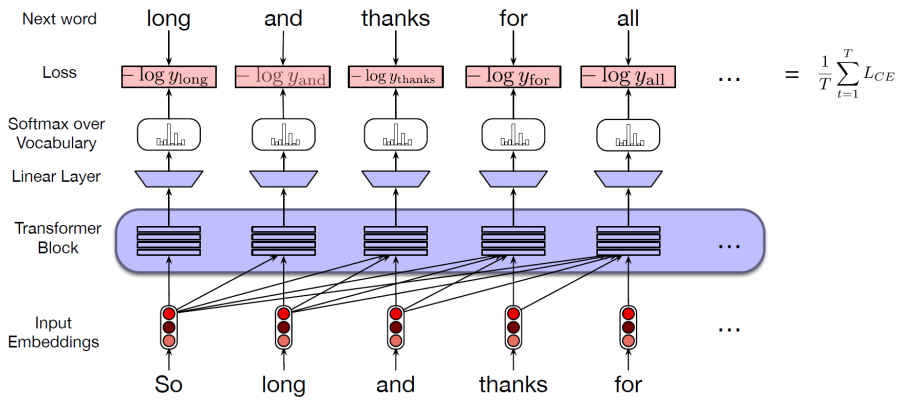


Figure 6.2: Training a transformer

6.3.1 Self-supervised Training

Self-supervised learning is simply learning by the natural sequence of words under its own supervision. We simply try to minimize the error in predicting the true next word in the next sequence.

The cross-entropy loss is:

$$L_{CE} = -\sum_{w \in V} \mathbf{y}_t[w] \log \hat{\mathbf{y}}_t[w]$$

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

Therefore, at each position, we take as input the sequence of tokens $w_{1:t}$ to predict the model's loss for the next token w_{t+1}. Then, we ignore whatever has been produced, and use the words $w_{1:t+1}$ to produce the token w_{t+2} and so on i.e., we use **teacher forcing**.

As with RNNs, the loss for a training sequence is the average cross-entropy loss over the entire sequence. The weights in the network are adjusted to minimize the average CE loss over the training sequence via gradient descent. But the key advantage over the RNN is that the calculation rather than being serial can be carried out in parallel for the whole sequence as the output for each element is independent as we are using teacher forcing.

6.3.2 Training Corpora for LLMs

From where does the training data for LLMs come from? Large language models are mostly trained using online scraped text, supplemented with more meticulously selected data. Due to the size of these training corpora, many natural instances that can be useful for NLP tasks will probably be present, such as question and answer pairs (from FAQ lists, for example), sentence translations between different languages, summaries of documents, and so on.

Web text is typically extracted from automatically crawled web pages such as the common crawl, which is a collection of billions of web pages. There are several ways to clean up typical crawl data, and the data is filtered. What information is contained in these data? An analysis suggests that in large part it's patent text documents, Wikipedia, and news sites. Wikipedia plays a role in lots of language model training, as do corpora of books. The GPT3 models, for example, are trained mostly on the web (429 billion tokens), some text from books (67 billion tokens) and Wikipedia (3 billion tokens).

6.3.3 Scaling Laws

The performance of LLMs is mainly determined by 3 factors:

1. Model size (the number of parameters not counting embeddings)
2. Dataset size (the amount of training data)
3. Amount of computer used for training

The relationships between these factors and performance are known as scaling laws. The following three relationships for loss L as a function of the number of non-embedding parameters N, the dataset size D, and the compute budget C, for models training with limited parameters, dataset, or compute budget, if in each case the other two properties are held constant:

$$L(N) = \left(\frac{N_C}{N}\right)^{\alpha_N}$$

$$L(D) = \left(\frac{D_C}{D}\right)^{\alpha_D}$$

$$L(C) = \left(\frac{C_C}{C}\right)^{\alpha_C}$$

The N can be roughly computed as follows:

$$N \approx 2dn_{layer}(2d_{attn} + d_{ff}) \approx 12d^2 n_{layer} \left[\text{assuming } d_{attn} \approx \frac{d_{ff}}{4} \approx d\right]$$

The values of N_c, D_c, C_c, α_N, α_D, and α_C depend on the exact transformer architecture, tokenization, and vocabulary size, so rather than all the precise values, scaling laws focus on the relationship with loss (L). Scaling laws can help determine, how much data will we need to increase the model size or to predict loss if we increase the data size or model size.

6.4 Practical Problems with LLMs

Since these LLMs are trained on a big bunch of data. So, if we use pre-trained models, they come with their limitations and biases.

```
from transformers import pipeline

unmasker = pipeline("fill-mask", model="bert-base-uncased")
result = unmasker("This man works as a [MASK].")
print([r["token_str"] for r in result])

result = unmasker("This woman works as a [MASK].")
print([r["token_str"] for r in result])
```

```
['lawyer', 'carpenter', 'doctor', 'waiter', 'mechanic']
['nurse', 'waitress', 'teacher', 'maid', 'prostitute']
```

Figure 6.3: See, the example of the bias of BERT model. The model gives only one gender-free answer (waiter/waitress). The others are work occupations usually associated with one specific gender and prostitute ended up in the top 5 possibilities the model associates with "woman" and "work." This happens even though BERT is one of the rare Transformer models not built by scraping data from all over the internet, but rather using apparently neutral data (it's trained on the English Wikipedia and BookCorpus datasets).

Following are some of the common problems:

- **Hallucination:** A problem where they can provide false information. The training algorithms don't have anything to ensure that the generated text is true.
- **Can generate toxic language:** LLMs can even generate toxic languages and harsh texts over non-toxic prompts. They can produce stereotypes and negativity towards demographies.
- **Privacy Issues:** They can leak information about their training data. Hence, it's possible that if there were some personal details, or contact information in the training data, the model could leak it.
- **Misinformation:** Language models can also be used by malicious actors to generate text for misinformation, phishing, or other socially harmful activities. We have seen in the news nowadays very often the exploitation of generative AI in spreading hatred, and deep fakes.

The training data is one cause of bias; LLMs' training datasets contain offensive content that is collected from websites that are prohibited. Aside from toxicity, there are other biases as well: a disproportionate number of writers from the US and other developed nations provide the training data. The ensuing generation is probably skewed toward the opinions or subjects of this group alone due to such biased demographic samples. Language models can also magnify biases in training data related to demographics and other factors.

6.4.1 Carbon Footprints of Transformers

Another problem is that transformers are really big models and training them becomes costly in terms of time and compute resources. Unfortunately, this has a huge environmental impact. Running a lot of trials to get the best parameters would have even higher carbon emissions.

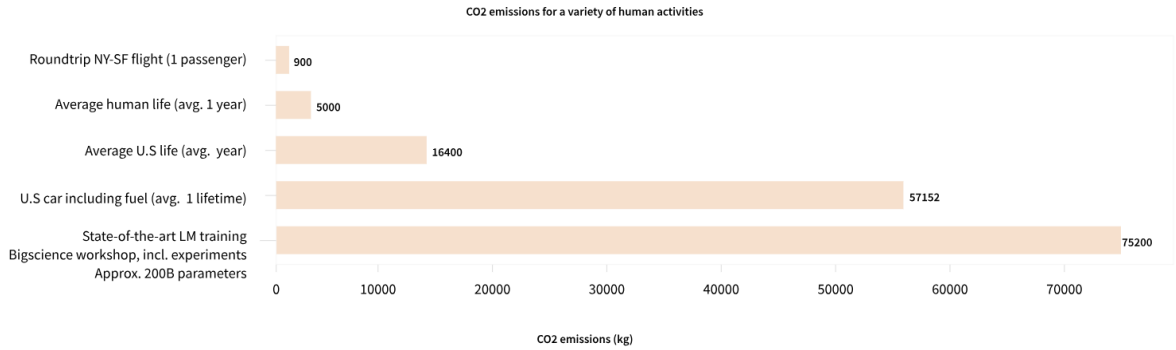


Figure 6.4: CO_2 Emissions

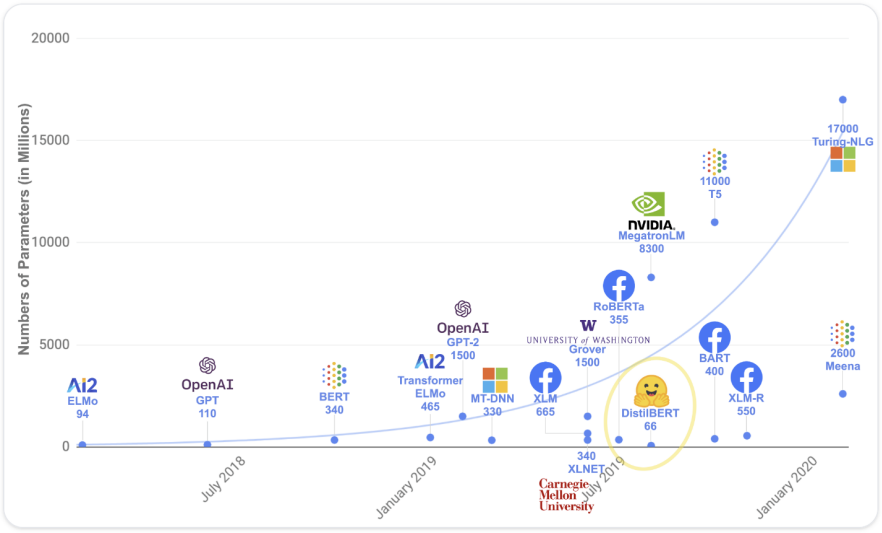


Figure 6.5: Increase in size of transformer models across years

So, if everyone trains models from scratch it will have huge impacts. Hence, sharing the model is the key, where we train over the already trained weights. This process is called Transfer Learning, Fine-Tuning of Language Models which we will be discussing

in the next chapter. However, using a pre-trained model has its limitations as the pre-trained model will come with its own biases and we need to keep in mind that it can generate sexist, racist, or homophobic content. Fine-tuning the model on your data won't make this intrinsic bias disappear.

Chapter 7

Fine-tuning and Bi-directional Transformer

The models we discussed before were casual or left-to-right transformer models. In this chapter, we will first discuss **a bi-directional transformer** encoder, trained via **masked language modeling**. We will see the most used masked language modelling architecture, the BERT model.

Next, we see two more ideas that are used with these types of language models, first one is **fine-tuning**. It is just training the model over the last layer for the specific task required. This is what we commonly call **Transfer Learning** in ML. The second idea is the idea of **contextual embeddings.**

7.1 Bi-directional Transformer Encoder

The bi-directional encoder transformer is the underlying model for BERT and it's descendants. In the transformer model, the model that we discussed can be easily applied to tasks with auto-regressive generation, summarization and machine translation. However, when applied to sequence classification and labelling problems. these models have their shortcomings since they can only perform left-to-right processing of inputs. So, in the previous model, the hidden state computation at each time is solely based on the current and earlier elements while in this bi-directional model, it takes information from the right to tokens too which can have the useful information that the previous model was ignoring. The casual models are sometimes called **decoder-only**; the models of this chapter are sometimes called **encoder-only** because they produce an encoding for each

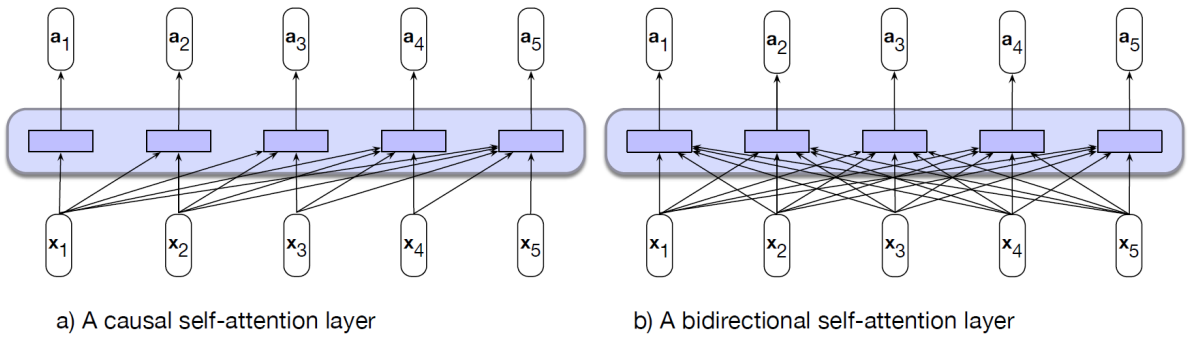


Figure 7.1: Difference in the self-attention of the two models.

input token but generally aren't used to produce running text by decoding/sampling.

7.1.1 Architecture of Bi-directional models

The architecture of the bi-directional model is exactly similar to what we use in casual or decoder-only models. Here, are the equations for the same:

$$\mathbf{Q} = \mathbf{X}\mathbf{W}^{\mathbf{Q}}; \mathbf{K} = \mathbf{X}\mathbf{W}^{\mathbf{K}}; \mathbf{V} = \mathbf{X}\mathbf{W}^{\mathbf{V}}$$

$$\text{SelfAttention}(\mathbf{Q}, \mathbf{K}, \mathbf{V}) = \text{softmax}\left(\frac{\mathbf{Q}\mathbf{K}^T}{\sqrt{d_k}}\right)\mathbf{V}$$

The difference is just in how self-attention is calculated, the summation limits are from 1 to n.

$$\alpha_{ij} = \text{softmax}(\text{score}(\mathbf{x}_i, \mathbf{x}_j)) = \frac{\exp(\text{score}(\mathbf{x}_i, \mathbf{x}_j))}{\sum_{k=1}^{n}\exp(\text{score}(\mathbf{x}_i, \mathbf{x}_k))}$$

$$\mathbf{a}_i = \sum_{j=1}^{n}\alpha_{ij}\mathbf{x}_j$$

7.1.2 Training Bidirectional Encoders

We train casual models by passing them a sequence asking to predict the next word on each step. But, with the bi-directional models, the task of the next-word identification is easy as the model already knows the context of upcoming tokens. So, we use another intuitive approach where we ask the model to perform a fill-in-the-blank approach (technically known as a cloze task). So, instead of predicting the next task we ask the model to predict the blank as in `"Jawahar Lal Nehru was the _____ prime minister of India"`. Therefore, given an input sequence with one or more words missing the model gives probability distribution over the entire vocabulary for each word.

Masking Words

The original approach to train bi-directional encoders is called **Masked Language Modelling(MLM)**. Here, a random sample of tokens from each training sequence is given to the model to utilize in the learning task. After selected, a token can be employed in one of three ways:

- It is replaced with the unique vocabulary token `[MASK]`.
- It is replaced with another token from the vocabulary, randomly sampled based on token unigram probabilities.
- It is left unchanged.

In BERT, 15% of the input tokens in a training sequence are sampled for learning. Of these, 80% are replaced with `[MASK]`, 10% are replaced with randomly selected tokens, and the remaining 10% are left unchanged.

The objective is to predict the original inputs for each of the replaced tokens. The cross-entropy loss from these predictions is used to train the model. All tokens are used for attention but only sampled tokens are used for learning.

For each of the masked tokens, a probability distribution over vocabulary is generated over the entire vocabulary after passing through transformer layers. The probability distribution $\mathbf{y}_i$ is generated from the last output $\mathbf{z}_i$ of transformer blocks using a weight matrix ($\mathbf{W}_V \in \mathbb{R}^{|V| \times d_h}$)

$$\mathbf{y}_i = \text{softmax}(\mathbf{W}_V \mathbf{z}_i)$$

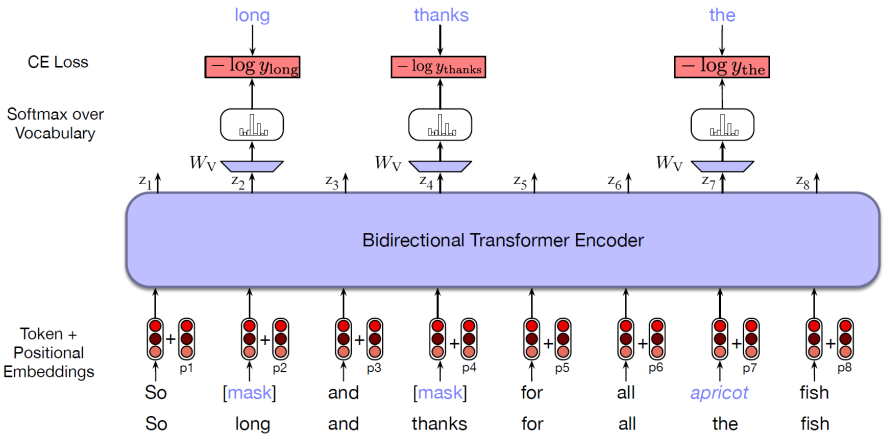


Figure 7.2: Masked language model training. In this example, three of the input tokens are selected, two of which are masked and the third is replaced with an unrelated word.

With this probability distribution, we can calculate the cross-entropy loss. Let M be the set of tokens that are masked. Hence, for a given token x_i, its loss is the probability of the correct word, given x^{mask}(as summarised by the single output vector $\mathbf{z}_i$).

$$L_{MLM} = -\frac{1}{|M|} \sum_{i \in M} L_{MLM}(x_i) = -\frac{1}{|M|} \sum_{i \in M} \log P(x_i | \mathbf{z}_i) = -\frac{1}{|M|} \sum_{i \in M} \log \mathbf{y}_i[x_i]$$

This loss now can be used to train the weights.

Next Sentence Prediction

Mask-based learning aims to generate efficient word-level representations by predicting words from surrounding contexts. However, figuring out the relationship between sentence pairs is a significant class of applications. These include tasks like discourse coherence, entailment, and paraphrase detection.

For these applications, the BERT family include a second learning objective called **Next Sentence Prediction (NSP)**. In this task, the model is presented with pairs of sentences and is asked to predict whether each pair consists of an actual pair of adjacent sentences from the training corpus or a pair of unrelated sentences. The NSP loss is based on how well the model can distinguish true pairs from random pairs.

BERT introduces two new tokens:

- The token `[CLS]` is prepended to the input sentence pair
- The token `[SEP]` is placed between the sentences and after the final token of the second sentence.

Finally, word embeddings and positional embeddings are added to the input sequences.

During training, the output vector from the final layer associated with the `[CLS]` token represents the next sentence prediction. As with the MLM, a learned set of classification weights $\mathbf{W_{NSP}} \in \mathbb{R}^{2\times d_h}$ is used to produce a two-class prediction from the raw `[CLS]` vector.

$$y_i = \text{softmax}(\mathbf{W_{NSP}}\mathbf{h}_i)$$

Cross entropy loss for each sentence pair is then used to train the model.

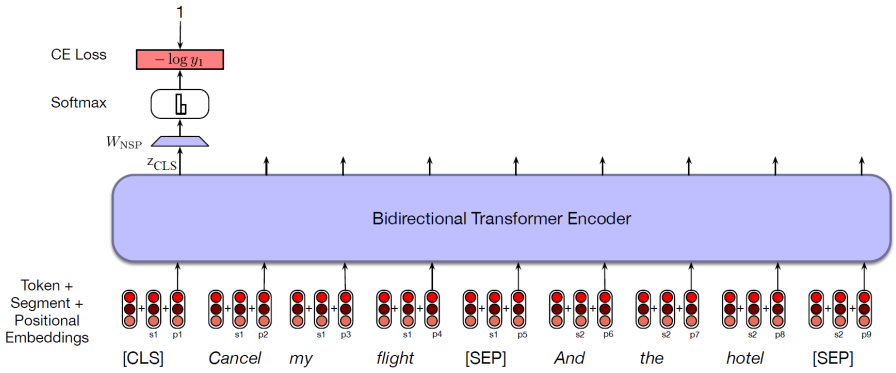


Figure 7.3: NSP loss calculation example

7.1.3 Training Regimes

BERT and other early transformer-based language models were trained on about 3.3 billion words (a combination of English Wikipedia and a corpus of book texts called BooksCorpus). Modern masked language models are now trained on much larger datasets of web text, filtered a bit, and augmented by higher-quality data.

MultiLingual models

What text should be used to learn this multilingual tokenization, given that it's easier to get much more text in some languages than others? One option would be to create this vocabulary-learning dataset by sampling sentences from our training data, randomly. In that case, we will choose a lot of sentences from languages like languages with lots of web representation like English, and the tokens will be biased toward rare English tokens instead of creating frequent tokens from languages with less data.

So, instead what we do is we divide the corpora into N languages, compute the number of sentence n_i of each language and readjust the probability to increase the probability of lesser-known languages. The new probability of selecting a sentence from each of the N languages (whose prior frequency is n_i) is $\{q_i\}_{i=1...N}$, where:

$$q_i = \frac{p_i^\alpha}{\sum_{j=1}^{N} p_j^\alpha} \text{ with } p_i = \frac{n_i}{\sum_{j=1}^{N} n_j}$$

A value of α between 0 and 1 will give higher probabilities to lower-probability samples.

Multilingual models can improve performance in low-resourced languages by leveraging linguistic information from a similar language in the training data that happens to have more resources. Some of the problems faced by these models are: **Curse of multilinguality** (the performance on each language degrades compared to a model training

on fewer languages), the vast amount of English language in training makes the model's representations for low-resource languages slightly more English-like

7.2 Contextual Embeddings

For any task that would require a model of word meaning, we can employ contextual embeddings as representations of word meanings in context, much as we did with static embeddings like word2vec. Therefore, contextual embeddings provide a single vector for each instance of that word type in its sentential context, whereas word2vec has a single vector for each word type. Thus, contextual embeddings are helpful in linguistic tasks requiring word meaning models, such as determining the semantic similarity of two words in context.

Contextual Embeddings and Word Sense

Since the same word can have different meanings: mouse can both mean a computer mouse or the animal that we know, similarly gay can mean both happy or a person who is attracted to another person of the same sex. We say that these words are polysemous. A sense (or word sense) is a discrete representation of one aspect of the meaning of a word. We can represent each sense with a superscript: gay^1 and gay^2, mouse^1 and mouse^2. These senses can be found listed in online thesauruses.

WSD-Word Sense Disambiguation

WSD algorithms take as input a word in context and a fixed inventory of potential word senses (like the ones in WordNet) and then output the correct word sense in context.

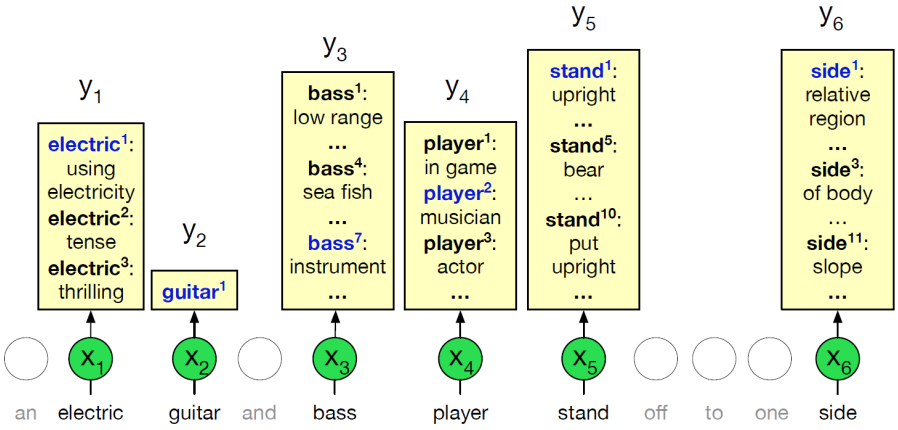


Figure 7.4: WSD task, words mapped from input words (x) to WordNet senses(y)

The best-performing WSD algorithm is a simple 1-nearest-neighbor algorithm using contextual word embeddings. At training time we pass each sentence in some sense-labelled dataset (like the SemCore or SenseEval datasets in various languages) through any contextual embedding (e.g., BERT) resulting in a contextual embedding for each labelled token.

There are various ways to compute this contextual embedding v_i for a token i; for BERT it is common to pool multiple layers by summing the vector representations of i

from the last four BERT layers. Then for each sense s of any word in the corpus, for each of the n tokens of that sense, we average their n contextual representations v_i to produce a contextual sense embedding v_s for s:

$$\mathbf{v}_s = \frac{1}{n}\sum_i \mathbf{v}_i \ ,\forall \mathbf{v}_i \in \text{tokens}(s)$$

At test time, given a token of a target word t in context, we compute its contextual embedding t and choose its nearest neighbour sense from the training set, i.e., the sense whose sense embedding has the highest cosine with t:

$$\text{sense}(t) = \text{argmax}_{s \in \text{senses}(t)} \cos(\mathbf{t}, \mathbf{v}_s)$$

7.3 Fine-Tuning Language Models

In fine-tuning, we create applications on top of pre-trained models by adding a small set of application-specific parameters. The fine-tuning process consists of using labeled data about the application to train these additional application-specific parameters. The following sections introduce fine-tuning methods for the most common applications including sequence classification, sequence labeling and sentence-pair inference.

7.3.1 Sequence Classification

An additional vector is added to the model to stand for the entire sequence, it is also called the **sentence embedding**. The `[CLS]` token functions as this embedding in BERT. During pretraining and encoding, this special token is appended to the beginning of every input sequence and added to the lexicon. The whole input sequence is represented by the output vector in the last layer of the model for the `[CLS]` input, which is then used as input by the classifier head—a logistic regression or neural network classifier—to determine the pertinent decision.

$$\mathbf{y} = \text{softmax}(\mathbf{W_C z_{CLS}})$$

Fine-tuning the values in $\mathbf{W_C}$ requires supervised training data consisting of input sequences labeled with the appropriate class. Training proceeds in the usual way using cross-entropy loss between the expected output and received output.

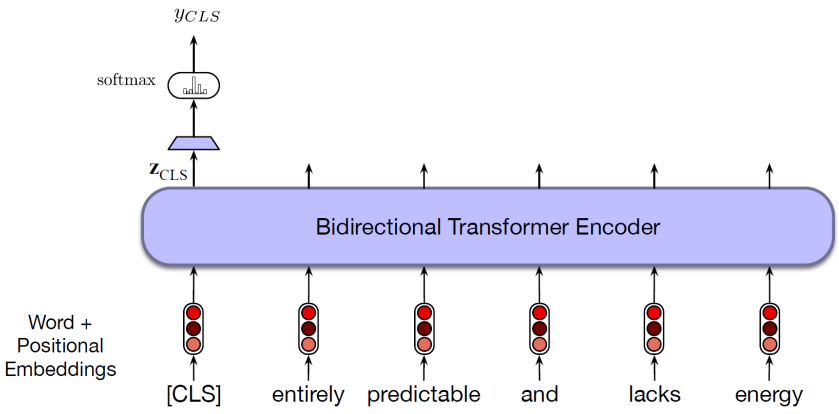


Figure 7.5: Sequence classification with bi-directional transformer encoder

7.3.2 Pair-Wise Sequence Classification

Practical applications that fall into this class include paraphrase detection (are the two sentences paraphrases of each other?), logical entailment (does sentence A logically entail sentence B?), and discourse coherence (how coherent is sentence B as a follow-on to sentence A?).

This is exactly similar to how we discussed in the 7.1.2 (Next Sentence Prediction - NSP). Fine-tuning for such a model we use the same procedure to start with the `[CLS]` token then the first sentence, followed by a `[SEP]` token to start the second sentence. And again a `[SEP]` token to end it. The output from the `[CLS]` vector is multiplied by waits to do the classification. The model can be trained by having such a dataset where pairs of sentences are labelled according to their classification.

For example: In the MultiNLI dataset, a pair of sentences is given 3 labels: *entails*, *contradicts* and *neutral*. These labels describe the relationship between the first statement (premise) and the second sentence (hypothesis). To fine-tune a classifier for the MultiNLI task, we pass the premise/hypothesis pairs through a bidirectional encoder as described above and use the output vector for the `[CLS]` token as the input to the classification head.

7.3.3 Sequence Labelling

Sequence labelling tasks, such as part-of-speech tagging or BIO-based named entity recognition(NER), follow the same basic classification approach. Here, the final output vector corresponding to each input token is passed to a classifier that produces a softmax distribution over the possible set of tags.

$$\mathbf{y_i} = \text{softmax}(\mathbf{W_K z_i})$$

$$\mathbf{t_i} = \text{argmax}_k(\mathbf{y_i})$$

Again this can be trained using cross-entropy loss over the outputs.

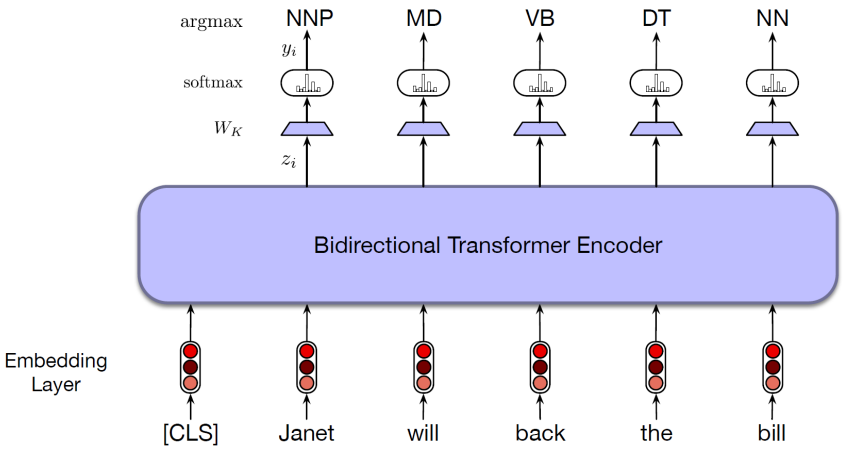


Figure 7.6: Sequence labelling for part-of-speech tagging with a bidirectional transformer encoder.

A complication of this approach comes with the use of sub-word tokenisation. To deal with this misalignment, we need a way to assign BIO tags to sub-word tokens

during training and a corresponding way to recover word-level tags from sub-words during decoding. For training, we can just assign the gold-standard tag associated with each word to all of the sub-word tokens derived from it. For decoding, the simplest approach is to use the arg max BIO tag associated with the first sub-word token of a word.

Chapter 8

NLP Pipeline

The Natural Language Processing or NLP pipeline is a set of steps or initiatives that are used to process raw text data before feeding it into the machine learning algorithms. In this chapter, we briefly describe and summarise the typical stages in the creation of an NLP pipeline. After this, we will be analysing a full NLP project. This is how we begin and understand our project flow as a NLP project involves a lot of steps. So, this pipeline summarises the processes for any general NLP project.

8.1 Data Acquisition

Data acquisition is the very initial process of NLP work and is often regarded as the most important step to create a great model. It includes acquiring the relevant information that is needed to solve the given NLP problem. **Sources**:

- Internal databases (e.g., MongoDB, MySQL)
- Public datasets (e.g., Google Dataset Search, U.S. Census Bureau)
- Web scraping
- Product intervention
- Data augmentation

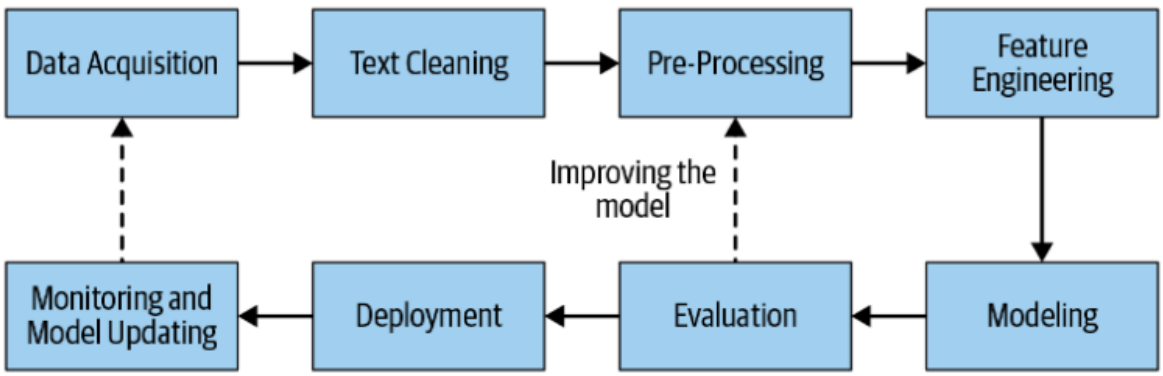


Figure 8.1: NLP Pipeline

8.2 Text Extraction and Cleanup

The next process is extracting the information and pre-processing the text after attaining the data.

- We need to discard the irrelevant Information.
- We need to extract the required fields, merge them and apply queries according to the requirement.
- Spelling errors should be fixed.
- Unnecessary new line characters should be eliminated.

8.3 Text Preprocessing

Text preprocessing is a crucial step in the NLP pipeline. It involves cleaning and preparing the text data for further analysis. The main steps include:

- Tokenization
- Stop-word Removal
- Stemming and Lemmatization
- Punctuation Removal

8.4 Text Representation

Therefore, the next steps for this textual data are to transform the text into a format within which a machine learning algorithm can work. Common techniques include:

- One-Hot Encoding
- Bag of Words (BOW)
- N-gram Models
- TF-IDF
- Word Embeddings

8.5 Model Building

The next process toward accomplishing this goal is to establish machine learning models that will be capable of learning from the textual information. This involves:

- Feature Extraction
- Model Selection: We need to choose the appropriate algorithms like neural networks, recurrent neural networks (RNNs), or transformers according to the requirement.

- Hyperparameter Tuning: We need to select the appropriate hyperparameters,it is one of the most important portion in Machine Learning.
- Training
- Evaluation

8.6 Model Deployment

When the model is built and tested, the last process is deployment. This involves:

- Integration: Including your model into an application or system where new textual data have to be processed according to this model.
- Monitoring: Overseeing the model's performance sufficiently to guarantee that it will continue being useful.
- Maintenance: Periodically re-calibrating the model and incorporating new data into the learning process if ever necessary.

8.7 Conclusion

It is worth stressing that the creation of an NLP application is a rather long and gradual process which includes several stages starting from the data collection to the monitoring. All of these are important to make certain that the model will work as intended and flexible to new data set and/or new demands. For more information, refer to The book [22].

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
- **Midterm Report Submission:** Late June
- **Week 5:** Transformers
  - Introduction to Transformers
  - Attention is all we need!
- **Week 6:** Large Language Models (LLMs)
  - Understanding LLMs

  - Potential harms

- **Week 7:** BERT and Transfer Learning
  - BERT (Bidirectional Encoder Representations from Transformers)
  - Contextual Embeddings
  - Transfer Learning with BERT
- **Week 8:** Applications of NLP
  - Overview of NLP applications
  - NLP Pipeline
  - Project analysis and workflow explanation
- **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.

[17] Hugging Face Course. Introduction to nlp, 2024.

[18] Serrano.Academy YT Channel. What are transformer models and how do they work?

[19] StatQuest YT Channel. Transformer neural networks, chatgpt's foundation.

[20] codebasics YT Channel. Nlp pipeline: Nlp tutorial for beginners in python.

[21] CampusX YT Channel. End to end nlp pipeline — nlp pipeline.

[22] Bodhisattwa Majumder Anuj Gupta Harshit Surana Chowdhury, Sowmya Vajjala. *Practical Natural Language Processing: A Comprehensive Guide to Building Real-World NLP Systems*. O'Reilly Media, Inc., 2020.