

1 CHAPTER 01 : INTRODUCTION

1.1 DEFINITION

Natural language processing is a sub-field of computer science that combines linguistics with artificial intelligence tools to comprehend, manipulate and generate natural text.

1.2 NATURAL LANGUAGE COMPONENTS

1.2.1 NATURAL LANGUAGE UNDERSTANDING VS NATURAL LANGUAGE GENERATION

Natural language understanding gives the ability to computers to extract and identify certain information and give them a meaning, using **probabilistic models** or **deep learning based models**.

Natural language generation on the other hand aims to transform other forms of data to text.

1.2.2 TEXT MINING VS NLP

	Text Mining	nlp
Objective	Text transformation	Text understanding.
Tools	uses different tools	Mainly deep & machine learning models are used
Semantic analysis	do not consider semantic analysis	considers semantic analysis
Data nature	data is large amount of documents (written text)	data can be of different nature : text, voice, images...etc
Evaluation	simple evaluation	complex evaluation depending on the task
Human intervention	doesn't require human intervention	requires human intervention
Product	produces model + words frequency	produces a model + grammatical structure

1.3 LEVELS OF LANGUAGE PROCESSING

1.3.1 LEXICAL ANALYSIS

Text segmentation into paragraphs, sentences and words and identify the lexical components (words, phrases,...) of a particular language.

1.3.2 SYNTACTIC ANALYSIS

consists of identifying more high level components (grouping words and phrases into categories) and identify the relations between those components.

1.3.3 SEMANTIC ANALYSIS

consists of understanding the meaning of words and phrases.

1.3.4 ANALYSIS OF DISCOURSE

Understanding the meaning of phrases in the context of a text and identify the relation between different phrases, like : co-reference resolution.

1.3.5 PRAGMATIC ANALYSIS

pragmatic analysis focuses on understanding the real meaning of natural text.

1.4 NATURAL LANGUAGE PROCESSING APPLICATIONS

- Sentiment analysis.
- Named entity recognition.
- Text translation.
- Part of speech tagging.
- co-reference resolution.
- Search engines & ranking.
- chat bots.
- text to speech and speech to text.

1.5 NATURAL LANGUAGE PROCESSING CHALLENGES

- Issues related to data : Absence of datasets and tools for minority languages,manual documents annotation.
- Issues related to the complexity of text data : due to phenomena like : polysemous,metaphors,irony...etc.
- Issues related to humans nature : things like personality,emotions,intention are hard to model.
- Evaluation : hard/time consuming for complex tasks.
- Ethic : Potential bias and discrimination resulting from data bias.

2 CHAPTER 02 : PREPROCESSING IN NLP

2.1 BASIC TEXT PROCESSING

2.1.1 REGULAR EXPRESSIONS

Regular expressions are a sequence of characters that describes a text fragments, allowing for fast and easy text manipulation and information extraction.

2.1.2 EDIT DISTANCE

Edit distance allows comparing tow sequences of characters by counting how the minimum number of operations are necessary to transform the first sequence to the second one,there four such operations : **insertion,suppression,substitution** and **transposition**,an edit distance may allow all or some of these operations.

Types of edit distance :

- **Hamming distance** : allows only substitution, strings must have the same length.

$$D_i = D_{i-1} + \begin{cases} 0 & \text{if } x_i = y_i \\ 1 & \text{otherwise} \end{cases}$$

- **Longest common subsequence** : allows insertion and suppression.
- **Levenshtein distance** :

$$D[i, j] = \min \begin{cases} D[i-1, j] + 1 \\ D[i, j-1] + 1 \\ D[i-1, j-1] + \begin{cases} 2 & \text{if } x[i] \neq y[i] \\ 0 & \text{otherwise} \end{cases} \end{cases}$$

With $D \in M_{n+1, m+1, n} = |x|, m = |y|$ and $D[0, 0] = 0$

- **Jaro distance** : allows transposition.
- **Damerau–Levenshtein distance** : allows all operations between tow adjacent characters.

Applications :

- Compare tow files.
- auto-correct.
- plagiarism detection.
- spam detection.
- computational biology.

Example (Levenshtein distance between "play" and "stay") :

	#	s	t	a	y
#	0	1	2	3	4
p	1	2	3	4	5
l	2	3	4	5	6
a	3	4	5	4	5
y	4	5	6	5	4

Table 1: The distance matrix D

$$D(stay, play) = D_{length(stay), length(play)} = D_{5,5} = 4$$

2.2 TEXT PROCESSING AND TOKENIZATION

2.3 DEFINITIONS

- **CORPUS** : a dataset, a collection of documents.
- **DOCUMENT** : a data point in corpus is called "document".
- **TOKEN** : a text unite, usually a word or a subword.
- **VOCABULARY** : the set of unique tokens.

2.3.1 TEXT NORMALIZATION

Text normalization includes :

- Removing repetitive characters.
- Handling homoglyphs.
- Handling special entries like : urls, mentions, hashtags, emails ...etc.
- Captilization.
- Punctuation removal.
- Replacing multiple consecutive spaces with only one.
- Stop words removal.
- Handling emojis.

Q : what normalization steps we should apply ?

Answer : it mainly depends on : task, dataset size and computational resources.

2.3.2 NAMED ENTITY RECOGNITION

Identifying and classifying named entities like : places, persons, events...etc, helps in understanding the semantic of words, three methods are mainly used :

- Dictionary based methods.
- Rule based methods (model based and context based).
- Machine learning based methods.

2.3.3 TOKENIZATION

Tokenization is the process of dividing a text into smaller entities called tokens, we distinguish three types of tokenization algorithms :

- **Space/punctuation based tokenization** : tokenize the text based on white spaces and punctuations, can not handle all languages and can not handle rare words.
- **Rule based tokenization** : tokenize the based on white spaces and punctuations and applies additional rules on individual tokens, can not handle all languages and can not handle rare words.
- **Subword tokenization** : subword tokenization algorithms further decompose rare words on subwords, examples of such tokenization algorithms are : Byte-pair encoding, wordpiece, Unigram language model and sentencepiece.
 - Byte pair encoding : starts by dividing words into characters and adding them to the vocabulary, and while there's still space left in the vocabulary : find the most frequent pair of symbols, fuse them and add them to the vocabulary, the size of the vocabulary is hyper parameter of the algorithm.
 - Wordpiece : similar to BPE but fuses the pair of symbols with the highest frequency divided by the frequency of each symbol.

- Unigram language model.
- Sentencepiece : uses either BPE or Wordpiece at character level (even white spaces) which means that text is tokenized independently of the language, which allows detokenisation, as a hyperparameter sentencepiece takes the size of the vocabulary, and operates on the corpus' documents without any preprocessing.

2.3.4 STEMMING AND LEMMATIZATION

Stemming only conserves the root of the words, three types of stemming algorithms exist : Based on a **database** (dictionary), based on **statistics** and based on **rules**.

Lemmatisation reduces a word to its canonical form, also called a lemma, for example : is \rightarrow be.

3 TEXT VECTORIZATION

3.1 BASIC TEXT VECTORIZATION TECHNIQUES

3.1.1 BAG OF WORDS

bag of words is a technique that represents each document in the corpus as a vector with dimension equal to the size of the vocabulary V ($\dim D_i = |V|$), based on the occurrences of each word in the document.

More formally the dataset is represented as a matrix $D \in M_{n,m}$, where n is the number of the documents in the dataset and $m = |V|$, such that :

$D_{i,j} ==$ the number of occurrences of the j^{th} word of the vocabulary in the i^{th} document of the dataset.

3.1.2 TF-IDF

Term Frequency-Inverse Document Frequency (TF-IDF) is another frequency-based feature extraction technique that assigns lower weights to words that appears in many documents, and produces a matrix $D \in M_{n,m}$

$$D_{i,j} = \text{TF}(d_i, t_j) \times \text{IDF}(t_j) = \text{TF}(d_i, t_j) \times \log \left(\frac{N}{\text{df}(t_j)} \right)$$

$\text{df}(t_j)$: the number of documents where the term t_j appears

N : total number of documents

$\text{TF}(d_i, t_j) = \frac{\text{the number of occurrences of the } j^{th} \text{ word of the vocabulary in the } i^{th} \text{ document of the dataset.}}{\text{Total number of terms in document } i}$

3.1.3 ONE HOT ENCODING

one hot encoding represents each word in the vocabulary as a vector v of dimension $|V|$, for i^{th} word in the vocabulary :

$$v_j = \begin{cases} 1 & \text{if } i = j \\ 0 & \text{otherwise} \end{cases}$$

3.1.4 PROBLEMS WITH BASIC VECTORIAL REPRESENTATIONS

- they do not take into account the context of words as they don't take their order in the sentence into account.
- result in a high dimensional vectors for a large vocabulary size.
- BOW doesn't take into account the relative importance of words.

3.2 VECTORS SIMILARITY

3.2.1 COSINE SIMILARITY

$$\cos(u, v) = \frac{u \cdot v}{\|u\| * \|v\|}$$

$-1 \leq \cos(u, v) \leq 1$ with 1 indicating identical meanings and -1 indicating opposite meanings.

3.2.2 SIMILARITY BETWEEN TOW WORDS WITH TF-IDF

Q : Given a dataset with n documents, how to represent words with vectors such that words with similar meaning have high cosine similarity.

ANSWER : Using the words' context, context = meaning.

METHOD :

- Choose a window size, and for each word in the vocabulary collect all the words in its context across all the documents, in some sort the words became the documents.
- feed the result of previous step to the Tf-Idf algorithm, since Tf-Idf yields a vector for each document, we will have a vector for each word in the dataset, which means that Tf-Idf will result in $|V| \times |V|$ matrix.
- calculate the cosine similarity between the vectors corresponding to the tow words you wish to know their similarity.

3.3 WORD EMBEDDING

3.3.1 WORD2VEC

Word2Vec are unsupervised algorithms based on deep learning (shallow networks) to generate word embedding, two main variants exist: **continuous bag of words** and **skip-gram**.

CONTINUOUS BAG OF WORDS

In **CBOW** architecture we use a shallow neural network with one hidden layer and output layer and train it to predict the **center word** given the **context words**, we aim to maximize :

$$S = \frac{1}{I} \sum_{i=1}^I \log p(m_i | m_{i-n}, m_{i-n+1}, \dots, m_{i-1}, m_{i+1}, \dots, m_{i+n-1}, m_{i+n})$$

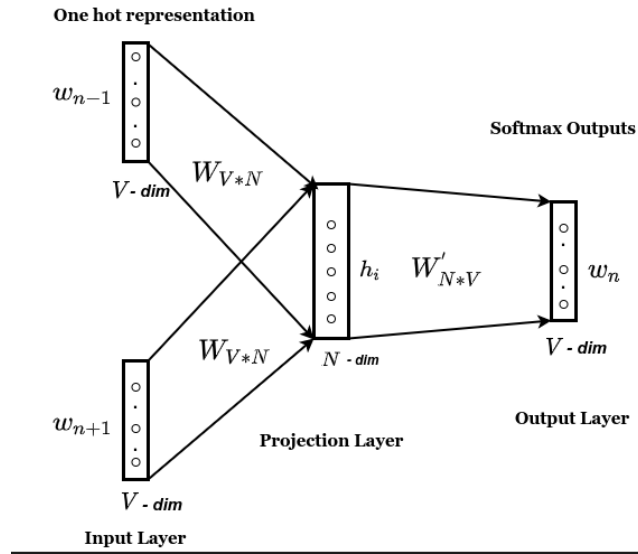


Figure 1: Continuous bag of words architecture, [source](#).

$$W \in M_{V,N} \text{ and } W' \in M_{N,V}$$

Dataset creation :

From raw documents, create a new dataset, pairs of contexts, and center words.

Forward pass :

Given a pair of (context, center word) with half window size of C :

- Lookup for the embedding for each word in the context using the matrix W , resulting in matrix of size $2 * C \times N$, let's call it A , which can also be achieved as multiplying the one hot encoded context word X of size $2 * C \times V$ with the matrix W .

$$A = X \times W$$

- Calculate the sum (or the mean) of the embeddings of the context words, this will result in a vector of size N , let's call it B .
- multiply the resulted vector from the previous step by the transpose of W' , resulting in a vector size V .

$$\text{logits} = B \times W'^T$$

- calculate the probabilities by applying the softmax function on the *logits* vector.

$$\hat{y} = \text{softmax}(\text{logits})$$

- calculate the cross entropy loss of y_{hat} and the center word.

$$loss = cross_entropy(\hat{y}, y)$$

SKIP GRAM

Does the inverse of CBOW it tries to predict the context words using the center word.

Forward pass :

Let X be a one hot vector of size V that represents the center word, W a matrix of size $V \times N$ and W' of size $N \times V$.

- We first compute $h = X \times W$, h is a vector of size N .
- We then compute $z = h \times W'$, z is a vector of size V .
- We apply the softmax function on the vector z , to get the predictions $\hat{y} = softmax(z)$.
- We then calculate the difference between \hat{y} and each one hot vector corresponding to a word in the context of X , which means that we'll end up with $2 * C$ error vectors that will get eventually sum up to get the final error vector e of size V .

Backward pass :

$$\begin{cases} \frac{\partial J}{\partial W} = x \times (W' \times e) \text{ Dimensions : } V \times (N, V \times V) = V \times N = V, N = \dim W \\ \frac{\partial J}{\partial W'} = h \times e \text{ Dimensions : } N \times V = N, V = \dim W' \end{cases}$$

Weights update :

$$\begin{cases} W = W - \alpha * \frac{\partial J}{\partial W} \\ W' = W' - \alpha * \frac{\partial J}{\partial W'} \end{cases}$$

SKIP-GRAM WITH NEGATIVE SAMPLING

The previously described skip-gram model is very computationally expensive to train to address this problem SGNS (skip-gram with negative sampling) is proposed.

The idea is instead of taking into account all the words in the vocabulary we sample for each center word K negative words (words that don't appear in the context of the center word), the loss and the gradients is then only calculated based on the context words and the negative samples and only one row (the one that corresponds to the center word) from the matrix W and $K + 1$ row from W' are updated each times.

when calculating the loss we include only the negative samples and the target positive word and we use a sigmoid instead of a softmax which can be seen as transforming the task from being multi-class to binary classification (given two words a and b , is the word b in the context of the word a ?).

Forward pass :

Let x be the center word encoded as one hot vector and A be the positive word and the negative samples one hot encoded in a matrix of size $(k+1) \times V$, W and W' are two matrices of size $V \times N$ called the embedding and the context matrices respectively.

- We calculate $h = x \times W$, h is a vector of size N .
- We calculate $W'' = A \times W'$, W'' is a matrix of size $(k+1) \times N$.
- We calculate $z = W'' \times h$, z is a vector of size $k+1$.
- We then apply sigmoid on each element of z to get the final predictions $\hat{y} = \text{sigmoid}(z)$, \hat{y} is of size $k+1$.
- finally we calculate the loss vector as the predictions minus the actual labels, $e = \hat{y} - y$, e is of size $k+1$.

Backward pass :

$$\left\{ \begin{array}{l} \frac{\partial J}{\partial W} = e \times W'' \\ \text{Dimensions : } 1, (k+1) \times (k+1), N = 1, N, \text{only the row that corresponds to the center word is updated.} \\ \frac{\partial J}{\partial W'} = e \times h \\ \text{Dimensions : } (k+1) \times N = (k+1), N, \text{only the rows that correspond to the one positive and the negative samples are updated.} \end{array} \right.$$

The window size and number of negative samples :

Usually a window size of 5 is used (half window size of 2), 2 words after + 2 words before + the center word.

In the original paper K is recommended to be 2-5 for large datasets and 5-20 for small datasets.

GLOVE (GLOBAL VECTORS FOR WORD REPRESENTATION)

GloVe is another unsupervised used for generating word embedding, based on co-occurrence matrix.

Notations :

- X co-occurrence matrix.
- $X_{i,j}$ the number of times the word j occurs in the context of the word i .
- $X_i = \sum_k X_{i,k}$ the number of times any word appeared in the context of the word i .
- $P_{i,j} = P(j|i) = \frac{X_{i,j}}{X_i}$: the probability that words j appears in the context of the word i .
- w_i, w_j : feature vectors for the words i and j .
- w_k : feature vector for the word k .

Idea :

The idea is to convert X into matrices W and V such that :

$$F(w_i, w_j, v_k) = \frac{P_{i,k}}{P_{j,k}}$$

is close to 1, if the word k either appears frequently or rarely appears in the context of both i and j .

$$F(w_i, w_j, v_k) = \frac{P_{i,k}}{P_{j,k}}$$

$$F(w_i - w_j, v_k) = \frac{P_{i,k}}{P_{j,k}}$$

$$F((w_i - w_j)^T v_k) = \frac{P_{i,k}}{P_{j,k}}$$

$$F(w_i^T v_k) = P_{i,k} = \frac{X_{i,k}}{X_i} \implies F((w_i - w_j)^T v_k) = \frac{F(w_i^T v_k)}{F(w_j^T v_k)}$$

if we choose F to be *exp* then :

$$w_i^T v_k = \log P_{i,k} = \log X_{i,k} - \log X_i$$

$\log X_i$ is independent of k then $\log X_i = b_i + \tilde{b}_i$:

$$w_i^T v_k + b_i + \tilde{b}_i = \log X_{i,k}$$

Loss function :

The loss function is a weighted sum squared error given by the following formula :

$$J = \sum_{i,j}^V F(x_i)(w_i^T v_j + b_i + \tilde{b}_i - \log X_{i,j})^2$$

With :

$$F(x) = \begin{cases} (\frac{x}{x_{max}})^\alpha & \text{if } x < x_{max} \\ 1 & \text{otherwise} \end{cases}$$

The recommended values for x_{max} and α are : 100 and $\frac{3}{4}$ respectively.

FASTTEXT

FastText uses either CBOW or skip-gram, but with an additional change that allows calculating word embeddings for unknown words.

- First we append a starting symbol to the beginning and an end symbol to the end of each token.
- We then generate n-grams from each token using a moving window of size n, n is recommended to be 3-6.
- to deal with the potential large number of unique n-grams we group them into buckets, using a hash function we calculate the bucket index of each n-gram.
- after applying this processing the difference is that the embedding for the center word is calculated using the sum of its n-grams, this processing is only applied to the center words and the embeddings for the context words/negative samples are extracted directly from the context matrix, which means that W is of size $B \times N$ and W' is of size $V \times N$, with B being the number of buckets.

SENTENCE EMBEDDING

To represent sentences with vectors two main approaches are used :

- Inference from the mean : the sentence embedding is either the average of the embedding of each word that appears in the sentence or a weighted average of them (usually Tf-Idf weights are used).
- Deep Learning based approaches that use sequence-to-sequence models.

4 NATURAL LANGUAGE PROCESSING WITH PROBABILISTIC MODELS

4.1 PART OF SPEECH TAGGING

Part of speech tagging is task that involve assigning a **grammatical class** to a word, which can be useful in tasks such as : **translation** and **text to speech**.

We distinguish tow types of grammatical classes :

- Open classes : classes that are frequently updated, like : noun, verb, adjective and adverbs.
- Closed classes : classes that are rarely/never updated, like : conjunction, pronouns, auxiliary.

Three main approaches are used :

- Markov chains.
- Hidden Markov chains.
- Deep learning.

4.2 MARKOV MODELS

4.2.1 STOCHASTIC PROCESS

We call stochastic (random) process a sequence of random variables q_1, q_2, \dots, q_T in a probability space ($q_i \in \Omega = S_1, S_2, \dots, S_N$), with q_t representing the system's state at instance t , a stochastic process is described by :

- The probability distribution of q_1 .
- The conditional probability $P(q_t = S_j | q_1 = S_{i_1}, q_2 = S_{i_2}, \dots, q_{t-1} = S_{i_{t-1}})$.

4.2.2 MARKOV CHAINS

Markov chains are a special type of stochastic process, that respects the following tow conditions :

- **Markov condition** : the value of the current state depends only on the previous state, which means we only need to know $P(q_t = S_i | q_{t-1} = S_j)$.
- **Stationary property** : the transition probability ($P(q_t = S_i | q_{t-1} = S_j)$) is constant over time.

A Markov chain can be represented by a transition matrix $A \in R^{N \times N}$ and an initial distribution $\pi \in R^N$, or represented graphically.

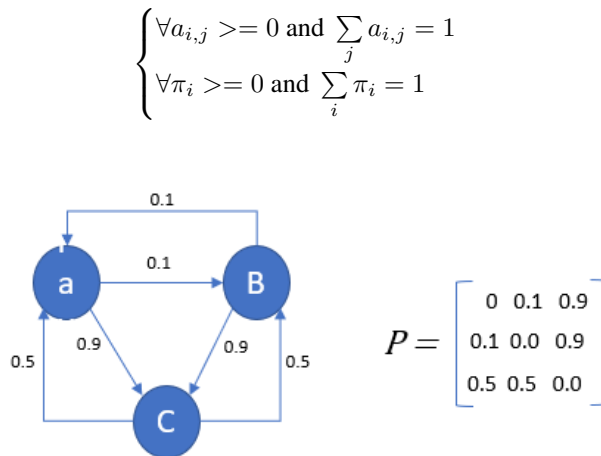


Figure 2: Markov models representation, [source](#).

4.2.3 HIDDEN MARKOV CHAINS

In a hidden Markov chain, the states are not directly observable; instead, they are hidden or latent, and you can only observe some output or emission that depends on the underlying state.

Hidden Markov chains can be described by a transition matrix A , an initial probability distribution π and an **emission matrix** b which summarizes the conditional probability $P(o_i|q_j)$.

In POS task : the initial distribution describes the probability of given state (grammatical class) to be the first one ($\pi_i = P(q_1)$), the transition matrix describes the probability of a grammatical class knowing the previous class ($a_{i,j} = P(q_j|q_i)$), and the emission matrix describes the probability of a word given the grammatical class ($b_{i,j} = P(w_j|q_i)$).

Populating the matrices A and B and the distribution π given an annotated corpus :

Formulas :

$$\begin{cases} \pi_i = P(q_i | \langle \text{sos} \rangle) = \frac{C(\langle \text{sos} \rangle, q_i)}{C(\langle \text{sos} \rangle)} \\ a_{i,j} = P(q_j | q_i) = \frac{C(q_i, q_j)}{\sum_j C(q_i, q_j)} = \frac{C(q_i, q_j)}{C(q_i)} \\ b_{i,j} = P(w_j | q_i) = \frac{C(q_i, w_j)}{\sum_j C(q_i, w_j)} = \frac{C(q_i, w_j)}{C(q_i)} \end{cases}$$

Example :

- un/D ordinateur/N peut/V vous/P aider/V
- il/P veut/V vous/P aider/V
- il/P veut/V un/D ordinateur/N
- il/P peut/V nager/V

D	$\frac{1}{4}$
N	0
V	0
P	$\frac{3}{4}$

Table 2: Initial distribution

	D	N	V	P
D	0	$\frac{2}{2}$	0	0
N	0	0	$\frac{1}{1}$	0
V	$\frac{1}{4}$	0	$\frac{1}{4}$	$\frac{2}{4}$
P	0	0	$\frac{5}{5}$	0

Table 3: Transition matrix

	un	ordinateur	peut	vous	aider	il	veut	nager
D	$\frac{2}{2}$	0	0	0	0	0	0	0
N	0	0	$\frac{2}{2}$	0	0	0	0	0
V	0	0	$\frac{2}{7}$	0	$\frac{2}{7}$	0	$\frac{2}{7}$	$\frac{1}{7}$
P	0	0	0	$\frac{2}{5}$	0	$\frac{3}{5}$	0	0

Table 4: Emission matrix

4.2.4 VITERBI ALGORITHM

Viterbi algorithm is an efficient algorithm that uses dynamic programming to find the most probable sequence of hidden states. The algorithm is composed of three steps :

- Initialization : calculate the initial probability for each hidden state.
- Forward pass : calculate the probability for each hidden state for each of the following stats.
- Backward pass : find the most probable sequence.

Initialization :

Initialize tow matrices C and D of zeros. $C, D \in R^{N \times K}$, with N being the number of grammatical classes and K the number of words in the sentence to annotate.

C is used to store the intermediate probabilities and D to store the intermediate indices

$$C_{i,1} = \pi_i * b_{i, index(w_1)}$$

Note : the function index returns the index of the word in the emission matrix for example : $index(il) = 6$.

Forward step :

In the forward step we populate the matrices C and D column after column using the following formulas :

$$\begin{cases} C_{i,j} = \max_k C_{k,j-1} * A_{k,i} * B_{i, index(w_j)} \\ D_{i,j} = \operatorname{argmax}_k C_{k,j-1} * A_{k,i} * B_{i, index(w_j)} \end{cases}$$

Backward step :

Algorithm 1 Backward

<pre> tags ← array(k) s ← argmax_i C_{i,K} tags[K] ← s for i ← K − 1 down to 1 do tags[i] ← D[s,i+1] s ← tags[i] end for return tags </pre>	<p>▷ initialize an array of size K</p> <p>▷ Get the index of the row with largest probability in the last column</p> <p>▷ Use the matrix D to get the remaining tags</p> <p>▷ the array tags contains the indices of the actual tags.</p>
---	--

Applying these three steps on the model found in the previous example on the sentence : "un ordinateur veut nager", note that \odot represents element wise multiplication and it is broadcastable.

Initialization :

$$C_{-,1} = \pi \odot B_{-,index(un)} = \pi \odot B_{-,1} = \begin{bmatrix} 0.25 \\ 0 \\ 0 \\ 0.75 \end{bmatrix} \odot \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \end{bmatrix} = \begin{bmatrix} 0.25 \\ 0 \\ 0 \\ 0 \end{bmatrix}$$

Forward :

Column 2 :

$$\begin{aligned} (C_{-,1} \odot A) \odot B_{-,index(ordinateur)}^T &= (C_{-,1} \odot A) \odot B_{-,2}^T = \left(\begin{bmatrix} 0.25 \\ 0 \\ 0 \\ 0 \end{bmatrix} \odot \begin{bmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0.25 & 0 & 0.25 & 0.5 \\ 0 & 0 & 1 & 0 \end{bmatrix} \right) \odot [0 \quad 1 \quad 0 \quad 0] \\ &= \begin{bmatrix} 0 & 0.25 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix} \odot [0 \quad 1 \quad 0 \quad 0] = \begin{bmatrix} 0 & 0.25 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix} \end{aligned}$$

After applying column wise *max* and *argmax* (from each column get the maximum value and the index of the maximum value) :

$$C_{-,2} = \begin{bmatrix} 0 \\ 0.25 \\ 0 \\ 0 \end{bmatrix}, D_{-,2} = \begin{bmatrix} 1 \\ 1 \\ 1 \\ 1 \end{bmatrix}$$

Column 3 :

$$\begin{aligned} (C_{-,2} \odot A) \odot B_{-,index(veut)}^T &= (C_{-,2} \odot A) \odot B_{-,7}^T = \left(\begin{bmatrix} 0 \\ 0.25 \\ 0 \\ 0 \end{bmatrix} \odot \begin{bmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0.25 & 0 & 0.25 & 0.5 \\ 0 & 0 & 1 & 0 \end{bmatrix} \right) \odot [0 \quad 0 \quad \frac{2}{7} \quad 0] \\ &= \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0.25 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix} \odot [0 \quad 0 \quad \frac{2}{7} \quad 0] = \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0.071428 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix} \end{aligned}$$

After applying column wise *max* and *argmax* :

$$C_{-,3} = \begin{bmatrix} 0 \\ 0 \\ 0.071428 \\ 0 \end{bmatrix}, D_{-,3} = \begin{bmatrix} 1 \\ 1 \\ 2 \\ 1 \end{bmatrix}$$

Column 4 :

$$\begin{aligned} (C_{-,3} \odot A) \odot B_{-,index(nager)}^T &= (C_{-,3} \odot A) \odot B_{-,8}^T = \left(\begin{bmatrix} 0 \\ 0 \\ 0.071428 \\ 0 \end{bmatrix} \odot \begin{bmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0.25 & 0 & 0.25 & 0.5 \\ 0 & 0 & 1 & 0 \end{bmatrix} \right) \odot [0 \quad 0 \quad \frac{1}{7} \quad 0] \\ &= \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0.01785 & 0 & 0.01785 & 0.03571 \\ 0 & 0 & 0 & 0 \end{bmatrix} \odot [0 \quad 0 \quad \frac{1}{7} \quad 0] = \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0.002551 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix} \end{aligned}$$

After applying column wise *max* and *argmax* :

$$C_{-,4} = \begin{bmatrix} 0 \\ 0 \\ 0.002551 \\ 0 \end{bmatrix}, D_{-,4} = \begin{bmatrix} 1 \\ 1 \\ 3 \\ 1 \end{bmatrix}$$

Finally :

$$C = \begin{bmatrix} 0.25 & 0 & 0 & 0 \\ 0 & 0.25 & 0 & 0 \\ 0 & 0 & 0.071428 & 0.002551 \\ 0 & 0 & 0 & 0 \end{bmatrix}, D = \begin{bmatrix} 0 & 1 & 1 & 1 \\ 0 & 1 & 1 & 1 \\ 0 & 1 & 2 & 3 \\ 0 & 1 & 1 & 1 \end{bmatrix}$$

Backward :

$$C = \begin{bmatrix} 0.25 & 0 & 0 & 0 \\ 0 & 0.25 & 0 & 0 \\ 0 & 0 & 0.071428 & \textcolor{red}{0.002551} \\ 0 & 0 & 0 & 0 \end{bmatrix} \Rightarrow s = 3$$

We use s to select the element from the last column of D , which is 3, then 3 is used to select the element from penultimate column and so on :

$$D = \begin{bmatrix} 0 & 1 & 1 & 1 \\ 0 & \textcolor{red}{1} & 1 & 1 \\ 0 & 1 & \textcolor{red}{2} & \textcolor{red}{3} \\ 0 & 1 & 1 & 1 \end{bmatrix}$$

The sequence is : 1,2,3,3 which corresponds to D,N,V,V.

4.2.5 CALCULATING THE PROBABILITY OF A SEQUENCE OF OBSERVATIONS

Let $\lambda = (A, B, \pi)$ a hidden Markov model with N states and $o = (o_1, o_2, \dots, o_T)$ a sequence of observations of length T , the task is to evaluate the probability of this sequence $P(o|\lambda)$:

$$\begin{cases} P(o|\lambda) = \sum_q P(o|q, \lambda) * P(q|\lambda) \\ P(q|\lambda) = \pi_{q_1} * A_{q_1, q_2} * A_{q_2, q_3} * \dots * A_{q_{T-1}, q_T} \\ P(o|q, \lambda) = B_{q_1, o_1} * B_{q_2, o_2} * \dots * B_{q_T, o_T} \end{cases}$$

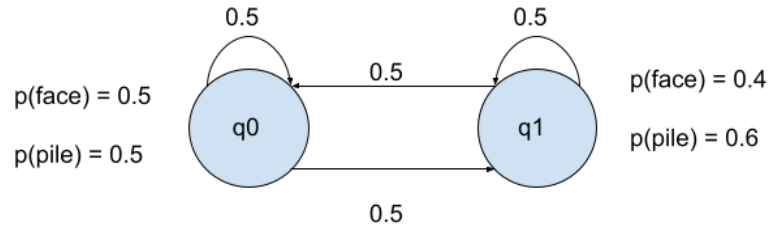
4.2.6 NAIVE APPROACH

- Enumerate all possible sequences of states q of length T .
- calculate $P(o|q, \lambda)$ and $P(q|\lambda)$ for each possible combination (sequence).
- calculate the product $P(o|q, \lambda) * P(q|\lambda)$ for each sequence.
- sum the products to get the final result.

Complexity :

$$(2 * T - 1) * N^T \text{ multiplications} \\ N^T - 1 \text{ additions}$$

Example $o = (face, pile, pile)$:



The following model corresponds to :

$$\pi = \begin{bmatrix} q_0 : 0.5 \\ q_1 : 0.5 \end{bmatrix}$$

$$A = \begin{bmatrix} q_0 & q_1 \\ q_0 : 0.5 & 0.5 \\ q_1 : 0.5 & 0.5 \end{bmatrix}$$

$$B = \begin{bmatrix} face & pile \\ q_0 : 0.5 & 0.5 \\ q_1 : 0.4 & 0.6 \end{bmatrix}$$

Calculations for the fourth sequence $q = (q_0, q_1, q_1)$:

$$P(q|\lambda) = \pi_{q_0} * A_{q_0, q_1} * A_{q_1, q_1} = 0.5 * 0.5 * 0.5 = 0.125$$

$$P(o|q, \lambda) = B_{q_0, face} * B_{q_1, pile} * B_{q_1, pile} = 0.5 * 0.6 * 0.6 = 0.18$$

$$P(o|q, \lambda) * P(q|\lambda) = 0.18 * 0.125 = 0.0225$$

Result :

states	$P(q \lambda)$	$P(o q, \lambda)$	Result
q_0, q_0, q_0	0.125	0.125	0.015625
q_0, q_0, q_1	0.125	0.15	0.01875
q_0, q_1, q_0	0.125	0.15	0.01875
q_0, q_1, q_1	0.125	0.18	0.0225
q_1, q_0, q_0	0.125	0.1	0.0125
q_1, q_0, q_1	0.125	0.12	0.015
q_1, q_1, q_0	0.125	0.12	0.015
q_1, q_1, q_1	0.125	0.125	0.018
sum			0.136125

$$P(o|\lambda) = 0.136125$$

4.2.7 FORWARD ALGORITHM

the Forward algorithm is an algorithm that uses the principle of dynamic programming to solve the problem of evaluating the probability a sequence of observations more efficiently, The algorithm is composed of three steps :

- Initialization : $\alpha_1(i) = \pi_i B_{i,o_1}$.
- Induction : $\alpha_{t+1}(j) = [\sum_{i=1}^N \alpha_t(i) * a_{i,j}] * B_{j,o_{t+1}}$
- Stop : $P(o|\lambda) = \sum_{i=1}^N \alpha_{T,i}$

Complexity :

$$N * ((N + 1) * T - 1) + N \text{ multiplications}$$

$$N * (N - 1) * (T - 1) \text{ additions}$$

Example : (the previous example)

2 states, sequence of length 3 \implies solution can be represented by a 2×3 matrix.

Initialization :

$$\alpha_{-,1} = \pi * B_{-,face} = \begin{bmatrix} 0.5 \\ 0.5 \end{bmatrix} \odot \begin{bmatrix} 0.5 \\ 0.4 \end{bmatrix} = \begin{bmatrix} 0.25 \\ 0.2 \end{bmatrix}$$

Induction :

Column 02 :

state q_0 :

$$\alpha_{-,1} * A_{-,q_0} = \begin{bmatrix} 0.25 \\ 0.2 \end{bmatrix} \odot \begin{bmatrix} 0.5 \\ 0.5 \end{bmatrix} = \begin{bmatrix} 0.125 \\ 0.1 \end{bmatrix}$$

$$\text{sum}(\alpha_{-,1} * A_{-,q_0}) = 0.225$$

state q_1 :

$$\alpha_{-,1} * A_{-,q_1} = \begin{bmatrix} 0.25 \\ 0.2 \end{bmatrix} \odot \begin{bmatrix} 0.5 \\ 0.5 \end{bmatrix} = \begin{bmatrix} 0.125 \\ 0.1 \end{bmatrix}$$

$$\text{sum}(\alpha_{-,1} * A_{-,q_1}) = 0.225$$

$$\alpha_{-,2} = \begin{bmatrix} 0.225 \\ 0.225 \end{bmatrix} \odot B_{-,pile} = \begin{bmatrix} 0.225 \\ 0.225 \end{bmatrix} \odot \begin{bmatrix} 0.5 \\ 0.6 \end{bmatrix} = \begin{bmatrix} 0.1125 \\ 0.135 \end{bmatrix}$$

Column 03 :

state q_0 :

$$\alpha_{-,2} * A_{-,q_0} = \begin{bmatrix} 0.1125 \\ 0.135 \end{bmatrix} \odot \begin{bmatrix} 0.5 \\ 0.5 \end{bmatrix} = \begin{bmatrix} 0.05625 \\ 0.0675 \end{bmatrix}$$

$$\text{sum}(\alpha_{-,2} * A_{-,q_0}) = 0.12375$$

state q_1 :

$$\alpha_{-,2} * A_{-,q_1} = \begin{bmatrix} 0.1125 \\ 0.135 \end{bmatrix} \odot \begin{bmatrix} 0.5 \\ 0.5 \end{bmatrix} = \begin{bmatrix} 0.05625 \\ 0.0675 \end{bmatrix}$$

$$sum(\alpha_{-,2} * A_{-,q_1}) = 0.12375$$

$$\alpha_{-,3} = \begin{bmatrix} 0.12375 \\ 0.12375 \end{bmatrix} \odot B_{-,pile} = \begin{bmatrix} 0.12375 \\ 0.12375 \end{bmatrix} \odot \begin{bmatrix} 0.5 \\ 0.6 \end{bmatrix} = \begin{bmatrix} 0.061875 \\ 0.07425 \end{bmatrix}$$

Stop :

$$P(o|\lambda) = sum(\alpha_{-,3}) = 0.061875 + 0.07425 = 0.136125$$

4.2.8 BACKWARD ALGORITHM

Backward algorithm is composed of three steps :

- Initialization : $\beta_T(i) = 1$.
- Induction : $\beta_t(i) = \sum_{j=1}^N A_{i,j} * B_{j,o_{t+1}} * \beta_{t+1}(j)$.
- Stop : $P(o|\lambda) = \sum_{i=1}^N \pi_i * B_{i,o_1} * \beta_1(i)$

Initialization :

$$\beta_{-,3} = \begin{bmatrix} 1 \\ 1 \end{bmatrix}$$

Induction :

Column 02 :

State q_0 :

$$A_{q_0} \odot B_{-,pile} \odot \beta_{-,3} = \begin{bmatrix} 0.5 \\ 0.5 \end{bmatrix} \odot \begin{bmatrix} 0.5 \\ 0.6 \end{bmatrix} \odot \begin{bmatrix} 1 \\ 1 \end{bmatrix} = \begin{bmatrix} 0.25 \\ 0.3 \end{bmatrix}$$

$$sum(A_{q_0} \odot B_{-,pile} \odot \beta_{-,3}) = 0.25 + 0.3 = 0.55$$

State q_1 :

$$A_{q_1} \odot B_{-,pile} \odot \beta_{-,3} = \begin{bmatrix} 0.5 \\ 0.5 \end{bmatrix} \odot \begin{bmatrix} 0.5 \\ 0.6 \end{bmatrix} \odot \begin{bmatrix} 1 \\ 1 \end{bmatrix} = \begin{bmatrix} 0.25 \\ 0.3 \end{bmatrix}$$

$$sum(A_{q_1} \odot B_{-,pile} \odot \beta_{-,3}) = 0.25 + 0.3 = 0.55$$

So :

$$\beta_{-,2} = \begin{bmatrix} 0.55 \\ 0.55 \end{bmatrix}$$

Column 03 :

State q_0 :

$$A_{q_0} \odot B_{-,pile} \odot \beta_{-,2} = \begin{bmatrix} 0.5 \\ 0.5 \end{bmatrix} \odot \begin{bmatrix} 0.5 \\ 0.6 \end{bmatrix} \odot \begin{bmatrix} 0.55 \\ 0.55 \end{bmatrix} = \begin{bmatrix} 0.1375 \\ 0.165 \end{bmatrix}$$

$$sum(A_{q_0} \odot B_{-,pile} \odot \beta_{-,2}) = 0.1375 + 0.165 = 0.3025$$

State q_1 :

$$A_{q_1} \odot B_{-,pile} \odot \beta_{-,3} = \begin{bmatrix} 0.5 \\ 0.5 \end{bmatrix} \odot \begin{bmatrix} 0.5 \\ 0.6 \end{bmatrix} \odot \begin{bmatrix} 0.55 \\ 0.55 \end{bmatrix} = \begin{bmatrix} 0.1375 \\ 0.165 \end{bmatrix}$$

$$sum(A_{q_1} \odot B_{-,pile} \odot \beta_{-,2}) = 0.1375 + 0.165 = 0.3025$$

So :

$$\beta_{-,1} = \begin{bmatrix} 0.3025 \\ 0.3025 \end{bmatrix}$$

Stop :

$$P(o|\lambda) = sum(\pi * \beta_{-,1} * B_{-,face}) = sum\left(\begin{bmatrix} 0.5 \\ 0.5 \end{bmatrix} \odot \begin{bmatrix} 0.3025 \\ 0.3025 \end{bmatrix} \odot \begin{bmatrix} 0.5 \\ 0.4 \end{bmatrix}\right) = sum\left(\begin{bmatrix} 0.075625 \\ 0.0605 \end{bmatrix}\right) = 0.075625 + 0.0605 = 0.136125$$

4.2.9 EVALUATION FORWARD-BACKWARD

$$P(o|\lambda) = \sum_{i=1}^N \alpha_t(i) * \beta_t(i)$$

Note :

We can chose any column t for the evaluation.

Example:

$$\alpha = \begin{bmatrix} 0.25 & 0.1125 & 0.061875 \\ 0.2 & 0.135 & 0.07425 \end{bmatrix}$$

$$\beta = \begin{bmatrix} 0.3025 & 0.55 & 1 \\ 0.3025 & 0.55 & 1 \end{bmatrix}$$

$$\alpha \odot \beta = \begin{bmatrix} 0.25 & 0.1125 & 0.061875 \\ 0.2 & 0.135 & 0.07425 \end{bmatrix} \odot \begin{bmatrix} 0.3025 & 0.55 & 1 \\ 0.3025 & 0.55 & 1 \end{bmatrix} = \begin{bmatrix} 0.075625 & 0.061875 & 0.061875 \\ 0.0605 & 0.07425 & 0.07425 \end{bmatrix}$$

After applying a column wise sum :

$$\begin{bmatrix} 0.136125 & 0.136125 & 0.136125 \end{bmatrix}$$

5 Sequence to Sequence Models

5.1 Language models

A Language model is a probabilistic model that model the distribution of a sequence of words and are capable of understanding and generating natural language, They are used in many tasks such as : Machine translation, speech recognition, text generation...etc, and they are usually trained on vast amount of data.

5.2 Recurrent neural networks

Recurrent neural networks (RNNs) are a type of neural networks that designed to handle sequences $X = \{x_1, x_2, \dots, x_t\}$ such as text, it processes the input sequence taking into the account their order and allow previous outputs to be used as inputs while having hidden states h_t .

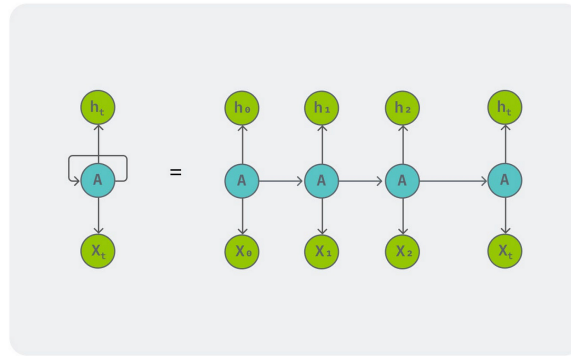


Figure 3: RNN representation

5.2.1 Traditional FNNs Vs RNNs

FNNs	RNNs
Fixed input & output length	Variable input & output length
Doesn't take into account the order of the data	Processes the input sequence in order.

Table 5: Traditional FNNs Vs RNNs

5.2.2 Traditional RNNs

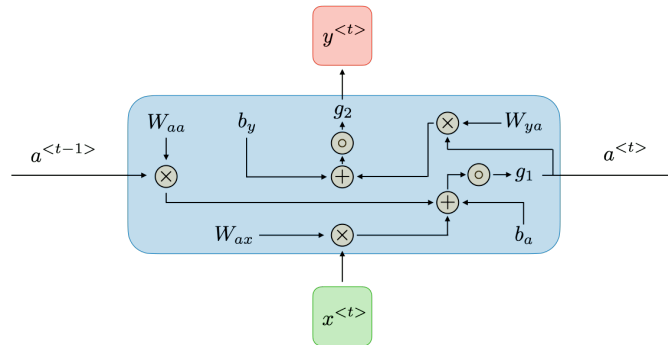


Figure 4: A Traditional RNN Cell

$$\begin{cases} a^{<t>} = g_1(W_{aa} * a^{<t-1>} + W_{ax} * x^{<t>} + b_a) \\ y^{<t>} = g_2(W_{ya} * a^{<t>} + b_y) \end{cases}$$

Where W_{aa} , W_{ya} , W_{ax} , b_a , b_y are **shared** temporally which means that the model's size doesn't increase with the increase of the sequence length and g_1 , g_2 activation functions, usually *sigmoid* and *softmax* are used respectively.

For $t = 0$ the hidden state is **initialized randomly**.

5.2.3 RNN Types

Depending on the task different Lengths for input & output sequences can be used.

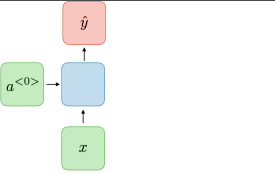
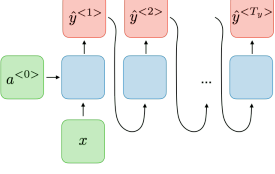
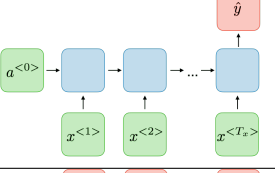
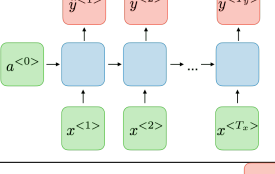
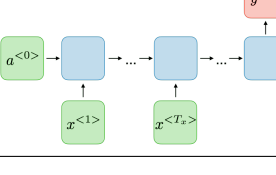
Type	Illustration	Example
One-To-One : $T_{in} = T_{out} = 1$		Traditional neural network
One-To-Many : $T_{in} = 1, T_{out} > 1$		Music generation
Many-to-one : $T_{in} > 1, T_{out} = 1$		Sentiment classification
Many-to-many : $T_{in} = T_{out}$		Name entity recognition
Many-to-many : $T_{in} \neq T_{out}$		Machine translation

Table 6: RNNs Types

5.2.4 Loss function

The loss function for a sequence to sequence model is the sum of the loss of each time step :

$$L(\hat{y}, y) = \sum_{t=1}^{T_{out}} L(\hat{y}^{<t>}, y^{<t>})$$

For example in the case of machine translation and if the used loss is the cross-entropy loss :

$$L(\hat{y}, y) = - \sum_{t=1}^{T_{out}} \sum_{k=1}^V y_k^{<t>} \log(\hat{y}_k^{<t>})$$

Where V is the size of the output language vocabulary.

5.2.5 Vanishing/exploding gradient

The vanishing and exploding gradient problems are often encountered in the context of RNNs. due to their limitations when it comes to capturing long term dependencies, because multiplying gradients can result in a very large or small value.

Gradient Clipping

It is a technique used to solve the problem of exploding gradient by limiting the value that a gradient can reach :

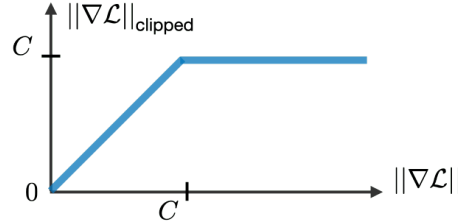


Figure 5: Gradient Clipping

5.2.6 LSTM & GRU

In order to solve the problem of vanishing gradient descent two types of RNN cells were introduced **LSTM** and **GRU**, these cells use **gates** to determine how much of the new information to store in the cell state and handle long term dependencies.

A gate is denoted by Γ and have their own learnable parameters W , U and b

$$\Gamma = \text{sigmoid}(W * x^{<t>} + U * a^{<t-1>} + b)$$

Types of gates

Gate Type	Role	Used In
Update Gate : Γ_u	How much past should matter now?	LSTM, GRU
Relevance Gate : Γ_r	Drop previous information?	LSTM, GRU
Forget Gate : Γ_f	Erase a cell or not?	LSTM
Output Gate : Γ_o	How much to reveal of a cell?	LSTM

Table 7: Caption

RNN/GRU

Characterization	GRU	LSTM
$\tilde{c}^{<t>}$	$\tanh(W_c[\Gamma_r \odot a^{<t-1>}, x^{<t>}] + b_c)$	$\tanh(W_c[\Gamma_r \odot a^{<t-1>}, x^{<t>}] + b_c)$
$c^{<t>}$	$\Gamma_u \odot \tilde{c}^{<t>} + (1 - \Gamma_u) \odot c^{<t-1>}$	$\Gamma_u \odot \tilde{c}^{<t>} + \Gamma_f \odot c^{<t-1>}$
$a^{<t>}$	$c^{<t>}$	$\Gamma_o \odot c^{<t>}$
Figure		

Table 8: GRU/LSTM

5.2.7 Bidirectional RNNs

Bidirectional RNNs can take into account both past and future input.

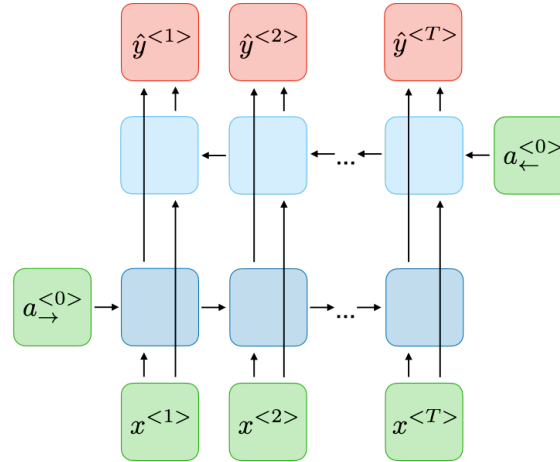


Figure 6: Bidirectional RNNs

5.2.8 RNN & LSTM & GRU (Summary & Comparison)

RNN	LSTM	GRU
Variable input length (+)	//	//
Shared weights	//	//
Takes into account the order of the input sequence	//	//
Slow computation	//	//
Can not handle long sequences	Can learn long range patterns	Can learn long range patterns
Uses only the past input	can be Bidirectional or Normal	can be Bidirectional or Normal
High risk of vanishing gradient	Low risk of vanishing gradient	Low risk of vanishing gradient
Low number of parameters	High number of parameters	Low number of parameters

Table 9: RNN & LSTM & GRU (Summary & Comparison)

5.3 Siamese Neural Networks

Siamese Neural Networks are a type of deep learning architectures that are used in **similarity learning** (a sub-type of supervised learning that aims to measure the similarity between two data points), it consists of two identical Neural networks used to represent inputs A and B (such as images) with embedding vectors $x_A = f(A)$, $x_B = f(B)$ and a **similarity metric** to measure the similarity between the two inputs.

5.3.1 Loss functions

Constructive Loss

$$\begin{cases} L = \frac{1}{2}(1 - Y) * D_w^2 + \frac{1}{2} * Y * \{max(0, m - D_w)^2\} \\ D_w = \sqrt{\{x_A - x_B\}^2} \end{cases}$$

Triplet Loss

With Triplet loss the input consists of three data points A, B and C such that A and B are very similar (for example two images that belongs to the same person) while A and C are dissimilar.

$$L(A, B, C) = max(0, d(A, B) - d(A, C) + m)$$

Where d is a distance metric, usually cosine similarity or L2 distance are used.

6 Attention Models

6.1 ENCODER-DECODER ARCHITECTURE

An Encoder-Decoder is a type of neural network architecture used in sequence-to-sequence learning. It comprises two main components: an **encoder** and a **decoder**. This architecture enables various tasks, including machine translation, text-to-speech, speech-to-text, and image captioning.

The encoder part takes the input sequence, process it and produces a context vector/or a set of context vectors which are then fed to **The decoder** part that which takes the the encoder's output and tries to construct the output sequence, the encoder and decoder parts are task-dependent for example in the case of machine translation both the encoder and decoder parts are RNN-based networks.

Attention mechanisms are usually used with Encoder-Decoder architectures to enhance the model's ability to capture complex relationships between the input and output sequences.

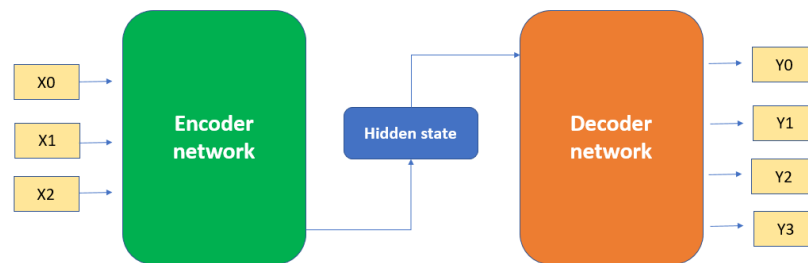


Figure 7: Encoder-Decoder Architecture, [source](#).

6.2 ATTENTION MECHANISM

The encoder processes the input sequence and generates a **single** hidden state (the hidden state of the last RNN cell) vector, which doesn't help capturing all the patterns and often leads to a vanishing/exploding gradient problem.

In the very general sense, the attention mechanism is an improvement to the encoder-decoder architecture. An encoder-decoder model uses a neural network to translate one input to another through the use of encoded feature representation.

A high-level overview of the attention mechanism implementation is as follows :

- Assign a score to each state in the encoder state.
- Compute the attention weight.
- Compute the context vector : calculate the context vector based on step 1 and 2.
- Feedforward : pass the calculated context vector the decoder.

7 Transformers

7.1 TRANSFORMERS

Transformers architecture are designed for sequence-to-sequence tasks. They leverage multi-head attention mechanisms to handle long-range dependencies more effectively than previous models. Transformers have become foundational in natural language processing, excelling in tasks such as machine translation, text generation, and language understanding.

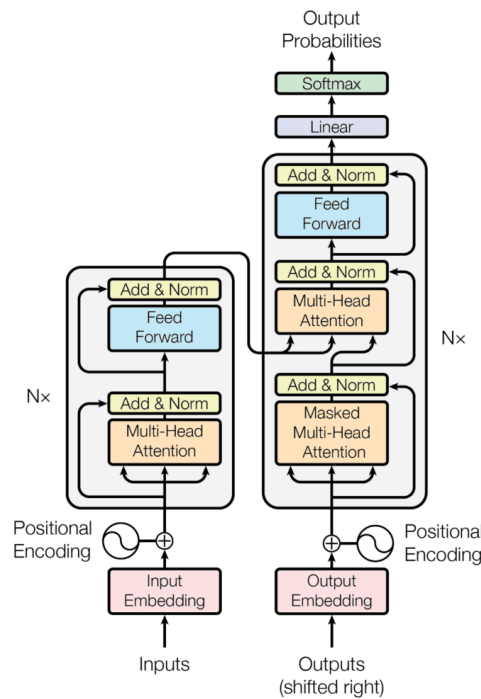


Figure 8: The transformer architecture, [source](#).

A transformer consists of N encoder and decoder layers. The first encoder layer takes as input the token embeddings with positional encoding, while subsequent encoder layers operate on the output of the previous encoder.

The first decoder layer takes as input the token embeddings with positional encoding of the output sequence as well as the output of the last encoder layer, while subsequent decoder layers operate on the output of the previous decoder layer as well as the output of the last encoder layer.

The outputs of the last decoder layer are then passed through a linear layer to map them to the dimension of the output sequence (the size of the vocabulary of the output sequences). Finally, softmax is applied to generate probabilities.

7.2 EMBEDDING LAYER

An Embedding Layer is essentially used to represent categorical/discrete features as continuous vectors. In nlp, embedding layers are used to represent each token in the vocabulary with a vector. The weight of an embedding layer is a matrix of dimension $V \times D$, where V is the size of the vocabulary and D is the dimension of the vector. Therefore, an embedding layer takes a sequence of length S and produces a matrix of dimensions $S \times D$.

The weight matrix is either initialized randomly and updated through backpropagation, or initialized with precomputed vectors such as GloVe, or a combination of both.

7.3 POSITIONAL ENCODING

Since transformers process the entire input sequence at the same time unlike recurrent units a form of ordering is needed in the architecture as it is clear that meaning of the sequences can change by changing the order of its elements.

A positional encoding layer maps each unique index to a vector of dimension D , so the weight of a positional encoding layer is a $S \times D$ matrix W where S is the size length of the input sequence and D is the dimension of the embedding vectors, it takes an input x of dimension $S \times D$, and produces the output $y = x + W$.

The weights of the positional encoding layer are not learnable and remain fixed; they are not updated during backpropagation, and they are initialized using the following formula :

$$\begin{cases} W_{i,2*j} = \sin(\frac{i}{N^{\frac{2*j}{D}}}) \\ W_{i,2*j+1} = \cos(\frac{i}{N^{\frac{2*j}{D}}}) \end{cases}$$

The value of N is usually set to 10000.

7.4 MULTI-HEAD ATTENTION & MASKED MULTI-HEAD ATTENTION

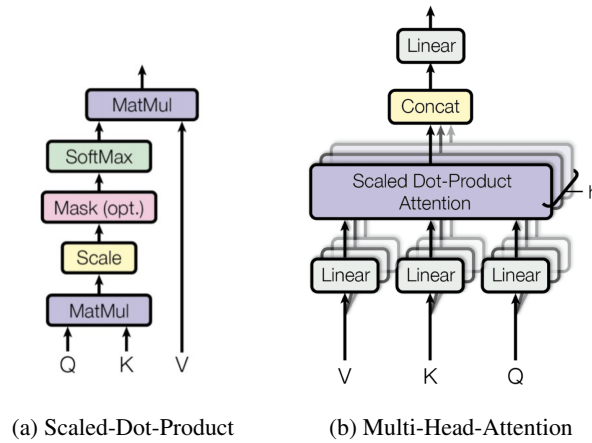


Figure 9: Multi-Head-Attention Mechanism

7.4.1 SELF-ATTENTION

VALUE, KEY AND QUERY :

A self-attention (or single-headed attention) layer is mainly composed of three linear layers called Value, Key, and Query, respectively. Their weights are all of dimension $D \times D_{att}$. Each layer is used to map the values of the input matrix (token embeddings + positional encoding) and/or change its dimension, meaning that each layer outputs a matrix of dimensions $S \times D_{att}$, let's call the resulted matrices V_{output} , K_{output} and Q_{output} respectively .

SCALED DOT-PRODUCT :

Then in **MatMul** stage the matrices V_{output} , K_{output} are multiplied, $y = Q_{output} \times V_{output}$, resulting in a $S \times S$ matrix that is often called **Attention Filter** which can be viewed as the attention that each token in the sequence is paying to other words in the sequence.

The attention scores are then scaled by dividing them by \sqrt{S} . This scaling is applied to prevent the issue of vanishing gradients. The scaled scores are then passed through a softmax layer to ensure that the values in each row of the attention matrix sum up to 1.

The attention Filter is then multiplied by the V_{output} matrix to produce a matrix of dimension $S \times D_{att}$.

$$ScaledDotProduct(Q, K, V) = softmax(\frac{Q \times K^T}{\sqrt{S}}) \times V$$

7.4.2 MULTI-HEAD ATTENTION

Unlike self-attention, multi-head attention learns multiple filters instead of one using the same mechanism described previously, so it can be viewed as h self-attention layers with an output dimension D_{att}/h where h is the number of heads.

The outputs of the Scaled Dot-Product stage of each head are then concatenated resulting in an $S \times D_{att}$ matrix, this matrix is then finally passed to a fully connected layer which maps the values and changes input dimension back to $S \times D$.

7.4.3 MASKED MULTI-HEAD ATTENTION

You may have noticed that in the figure above, a mask stage can be optionally performed after the Scale stage. The role of this stage is to prevent the model from looking into the future (i.e., paying attention to next tokens in the sequence) by masking (setting to zero) the upper triangular part of the attention filter.

7.5 ADD & NORM LAYER

7.5.1 Add

The add layer (also known as skip-connection or residual-connection) performs element-wise addition between the input and the output of the previous layer, which helps prevent the issue of vanishing gradient descent and allows for a better information flow.

7.5.2 Norm

After the addition operation, layer normalization is applied. Layer normalization normalizes the values across the feature dimension for each example in the mini-batch. This helps in stabilizing the learning process and improving generalization.