UniTs - University of Trieste

Faculty of Data Science and Artificial Intelligence

Department of mathematics informatics and geosciences

Natural Language Processing

Lecturer:
Prof. Alberto Cazzaniga

Author:
Christian Faccio

October 7, 2025

github.com/christianfaccio christianfaccio@outlook.it

Preface

As a student of Data Science and Artificial Intelligence, I've created these notes while attending the **Natural Language Processing** course.

The course provides a comprehensive introduction to the field of natural language processing, covering both theoretical concepts and practical applications. The notes encompass a variety of topics, including:

- ML and DL Fundamentals
- Tokenization and Text Preprocessing
- Word Embeddings
- RNN for NLP
- Transformers and Attention Mechanisms
- Language Models (e.g., BERT, GPT)
- Understanding LLMs
- Visutal Language Models

While these notes were primarily created for my personal study, they may serve as a valuable resource for fellow students and professionals interested in natural language processing.

Contents

1 Tokenization

1.1 Basics

NLP deals with computer readable text.

Problem. computer readable text is of variable format:

1. "This is an interesting lecture"
2. "This lecture is cool :)#nlp proc"
3. "This lecture is also inter-interesting"
4. The same sentence written in chinese, japanese, arabic, etc.

There is the need for **standardization**. Regular Expressions (∼ 1950 Kleen) are a way to describe patterns in strings.

Example: ***Find all sentences starting with "the"***

1. `/the/` → matches "the" anywhere in the string (but not "The")
2. `/[Tt]the/` → matches "the" or "The"
3. `/[Tt]he` → **MISSING:** explanation
4. `/[â-zA-Z][Tt]he[â-zA-Z]/` →
5. Solve the fact that this misses beginning or end of a line

*For more information on Regular Expressions look at [1].

Basic entities Computer readable text is collected in **Corpora**.

Definition: ***Corpus***

The **Corpus** is a collector of computer readable text (or speech). See for example a document, a novel, ...

Corpora may vary for different elements:

- Length: # bytes to store it;
- Language: phonemic (It, En, ...), syllabic (Arabic, Japanese), morphosyllabic (Chinese):
  - Different languages with same character system;
  - Same language, same character systems but different dialect or slang (ex. "I don't" → "Iont");
- Genre: written and spoken genre;
- Demographic characteristics;
- Time;

Standardization comes with **Datasheets** (T.Gebru) or **Data Statement** (Bendor). Important is the *motivation* for that collection, the *situation*, the *language-variety*, *speaker/writer demographics*, the *collection process* and the *annotation*. This is done to make it **reproducible** for others.

The unit we would like to focus on are **WORDS**. A word is the single distinct meaningful element of speech or writing, typically shown with spaces on either side when written or printed and saw with others in a portion of text.

Again, **problems** arise:

- Punctuation: usually considered as a word (”,”, ”.”, ”:”, ...);
- Utterance: like ”uh”, ”um”, ”ah”, ...;

Actual elementary parts of a corpus will be defined/deduced but he corpus itself.

Word Types are the distinct words in a corpus ($\mathscr{V}$ is the vocaboulary). Capitalization is sometimes considered and sometimes not, depends on the situation. **Word Instances** represent a measure of length, indicating how many total words in a corpus there are (N). **Word Counting** indicates how many times a single and distinct word is repeated in a corpus.

	$\lvert\mathscr{V}\rvert$	N
Shakespeare	31k	884k
Google n-grams	13M	1B
Wikipedia	?	4.9B

Definition: ***Herdant's Law***

$|\mathscr{V}| = k|\mathscr{C}|^{\beta}$ with $k > 0$ and $0 < \beta < 1$.

A big problem is the cardinality of the vocaboulary, making tasks like next token prediction almost impossible to solve.

1. A first solution is to reduce the vocaboulary size by using **Stemming** or **Lemmatization**: Stemming reduces words to their base or root form (e.g., ”running” $\to$ ”run”), while lemmatization considers the context and converts words to their meaningful base form (e.g., ”better” $\to$ ”good”).
2. A better solution is to use **Subword Tokenization**, breaking words into smaller units (subwords) that can be combined to form words. This approach helps manage the vocabulary size while still capturing meaningful components of words. Examples include Byte Pair Encoding (BPE) and WordPiece.

Definition: ***BPE***

Byte-Pair-Encoding is a procedure that, given a new sentence, divides it into subwords and then encodes it as a sequence of tokens. The algorithm starts with a base vocabulary of characters and iteratively merges the most frequent pairs of tokens to create new subword tokens, until a predefined vocabulary size is reached.
Example: corpus is made of (A,B,D,C,A,B,E,C,A,B).

- (AB D C AB E C AB), $\mathscr{V} = \{A, B, C, D, E, AB\}$;
- (AB D CAB E CAB), $\mathscr{V} = \{A, B, C, D, E, AB, CAB\}$;
- ...
- **k** times;

Once we have that, how do we perform **inference**?

- Start from characters ($|\mathscr{C}_{test}|$)
- Merge them into subwords
- Merge subwords into other subwords
- ...

Distance The usual measure of distance used in NLP is the **Edit Distance** (or Levenshtein distance). It is defined as the minimum number of operations required to transform one string into another, where the allowed operations are insertion, deletion, or substitution of a single character.

2 Neural Networks for NLP

Neural networks are **universal function approximators** that can learn complex patterns in data. They have become a cornerstone of modern natural language processing (NLP) due to their ability to model the intricacies of human language. In this chapter, we will explore the fundamental concepts of neural networks and how they are applied in NLP tasks. In theory, just **one hidden layer** is sufficient to approximate any function (very very large though), but in practice, deeper networks often yield better performance. This is also true if we require hidden layers to be of fixed size N but we allow for arbitrary depth D.

Why are we using neural networks?

1. They go beyond the so called **feature engineering** paradigm, where we manually design features to represent the data. Neural networks can automatically learn relevant features from raw data, reducing the need for extensive feature engineering;
2. Inner layers construct **meaningful representations** of the data, only from raw inputs (+ labels).

2.1 Basic Concepts of Neural Networks

We start from a vector of inputs:

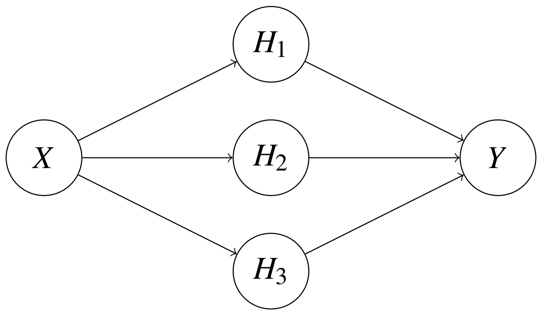
$$x \in \mathbb{R}^n$$

and we want to predict a vector of outputs:

$$y = f(z_x) = f(w \cdot x + b)$$

where:

- $w \in \mathbb{R}^m$ is a weight vector;
- $b \in \mathbb{R}$ is a bias term;
- f is a non-linear activation function.


$$\begin{tikzcd} & H_1 \arrow[dr] & \\ X \arrow[ur] \arrow[r] \arrow[dr] & H_2 \arrow[r] & Y \\ & H_3 \arrow[ur] & \end{tikzcd}$$

The **activation function** f introduces non-linearity into the model, allowing it to learn complex patterns. Common activation functions include:

- **ReLU** (Rectified Linear Unit): $f(z) = \max(0, z)$
- **Sigmoid**: $f(z) = \frac{1}{1+e^{-z}}$

Warning:

Due to **asymptotic properties**, ReLU is generally preferred over Sigmoid in hidden layers, as it helps mitigate the vanishing gradient problem and allows for faster training.

XOR is the classical example of a function that is not linearly separable, meaning that it cannot be represented by a single linear decision boundary. A neural network with at least one hidden layer can learn to represent the XOR function by combining multiple linear decision boundaries in a non-linear way.

> There is a depth 2 NN with ReLU activation that approximates XOR.
>
> – GOODFELLOW (2016)

MISSING: graphs for AND, OR, XOR and the NN for XOR

Multiplying by the matrix

$$W = \begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix}$$

added to the bias $b = \begin{bmatrix} 0 \\ -1 \end{bmatrix}$ and then applying ReLU, we get a linearly separable space.

2.2 Feedforward Neural Networks

There are problems with classical NN:

- Leave the choice of non-linearity;
- Hidden representation can have arbitrary dimension and we want arbitraty number of layers;
- Dimension of output can vary depending on the number of classes;
- We want outputs to be **probabilities**.

Starting with the fourth point, we can use the **softmax** function to convert raw output scores (logits) into probabilities. The softmax function is defined as:

$$\text{softmax}(z_i) = \frac{e^{z_i}}{\sum_j e^{z_j}}$$

where z_i is the i-th element of the input vector z. This ensures that the output values are in the range (0, 1) and sum to 1, making them interpretable as probabilities.

Tip: ***Exponential function***

Why the exponential? It is **monotonic**, **non-negative** and **good for derivative**.

The parameters in the network are **weights** and **biases**, which are learned during training. First of all, we need data:

$$X_{train} = \{(x_1, y_1), (x_2, y_2), \dots, (x_n, y_n)\}, \quad X_{test} = \{(x_1, y_1), (x_2, y_2), \dots, (x_m, y_m)\}$$

We define a **loss function** $L(y, \hat{y})$ that measures the difference between the predicted output $\hat{y}$ and the true output y. At a certain point we will have

$$\underline{y} = softmax(\hat{y}), \quad \hat{y} \in \{0, 1, \dots, C\}$$

and we can use the **cross-entropy loss** for multi-class classification:

$$L(y, \hat{y}) = -\sum_{i=1}^{C} y_i \log(\hat{y}_i)$$

The goal of training is to minimize the loss function over the training data by adjusting the weights and biases in the network. This is typically done using optimization algorithms such as **Stochastic Gradient Descent** (SGD) or its variants (e.g., Adam, RMSprop). We approach stochasticity by using **mini-batches**, small random subsets of the training data, to compute the gradients and update the parameters. This helps to reduce the variance in the updates and can lead to faster convergence.

Example: ***Wrong sentiment classification with NN***

We have a *Corpus* $\mathscr{C}$, i.e. a collection of tweets, and we want to classify them as *positive*, *negative* or *neutral*.

$$\mathscr{C}_{train} = \{(x_1,y_1),(x_2,y_2),\dots,(x_n,y_n)\}, \quad \mathscr{C}_{test} = \{(x_1,y_1),(x_2,y_2),\dots,(x_m,y_m)\}$$

where x_i is a tweet and $y_i \in \{\text{positive}, \text{negative}, \text{neutral}\}$. The **idea** is to convert each tweet into a vector of features and then feed it into a neural network for classification.

- $x_1 = \log(\text{count of tokens})$
- $x_2 = \#$ of **bad** tokens
- $x_3 = \#$ of **good** tokens
- $x_4 = \begin{cases} 1 & \text{if 'no' in tweet} \\ 0 & \text{otherwise} \end{cases}$
- $x_5 = \begin{cases} 1 & \text{if '?' in tweet} \\ 0 & \text{otherwise} \end{cases}$
- $x_6 = $ count of 1^{st} person pronouns

The problem is that this model **will not perform well**.

One-Hot-Encoding is a way to represent categorical variables as binary vectors. Each category is represented by a vector of length equal to the number of categories, with a 1 in the position corresponding to the category and 0s elsewhere. For example, if we have three categories: *cat*, dog, and *bird*, we can represent them as:

$$\text{cat} = [1,0,0], \quad \text{dog} = [0,1,0], \quad \text{bird} = [0,0,1]$$

It follows the tokenization of the corpus:

$$\mathscr{C} = \text{corpus} \rightarrow^{tokenization(BPE)} \mathscr{V} = \text{vocaboulary} \rightarrow^{OHE} \text{vector} \in \mathscr{R}^{|\mathscr{V}|}$$

Observation:

Input is in $\mathscr{R}^{|\mathscr{V}|\cdot n}$. The n is **variable** and depends on the number of tokens in the tweet. There are different solutions:

- **Pad** the input to a fixed length (e.g., 50 tokens) and truncate longer tweets. This allows us to have a fixed-size input vector, but it may lead to loss of information for longer tweets.
- **Cut** the input into fixed-size chunks (e.g., 10 tokens each) and process each chunk separately. This allows us to handle variable-length inputs, but it may lead to loss of context between chunks.
- **Average** the embeddings of all tokens in the tweet to obtain a fixed-size vector. This allows us to capture the overall meaning of the tweet, but it may lose information about the order of tokens.

They will return an **Embedding**, a dense vector representation of the input text, which can be fed into the neural network.

3 N-Gram Language Models

Theorized around 1980-1990, they are a probabilistic model for language. The objective is to construct a generative model able to produce sentences which are likely to be drawn from the distribution of a corpus $\mathscr{C}$.

Mathematically speaking, we want to estimate the probability of a sequence of words:

$$P(y_1 = w_1, y_2 = w_2, \ldots, y_n = w_n)$$

where y_i is a random variable representing the i-th word in a sentence and w_i is a specific word in the vocabulary $\mathscr{V}$.

Since predicting the entire sequence directly is complex, the idea is to decompose this joint probability using the **chain rule** of probability:

$$\begin{aligned} P(y_1, y_2, \ldots, y_n) &= P(y_1)P(y_2|y_1)P(y_3|y_1, y_2)\cdots P(y_n|y_1, y_2, \ldots, y_{n-1}) \\ &= \prod_{i=1}^{n} P(y_i|y_1, y_2, \ldots, y_{i-1}) \end{aligned}$$

The question now is: how do we approximate these conditional probabilities? In our hands we have the **corpus** $\mathscr{C}$, so we can just count how many times a sequence of words appears in it and then divide it by the sum of all the possible continuations:

$$P(y_i|y_1, y_2, \ldots, y_{i-1}) \approx \frac{\text{Count}(y_1, y_2, \ldots, y_{i-1}, y_i)}{\sum_{w\in\mathscr{V}} \text{Count}(y_1, y_2, \ldots, y_{i-1}, w)}$$

⚠ Warning:

The problem that arises now is that if the sequence has a high length, then it is very unlikely that it appears in the corpus, leading to a zero probability estimate. This is known as the **sparsity problem**.

3.1 N-Gram Models

This models are a first approximation to the problem of predicting the next word in a sentence if never seen before. A simple example is the following:

- ”John wears a mask” $\in \mathscr{C}$;
- ”Mary wears a mask” $\notin \mathscr{C}$;
- $P(mask|Marywearsa) = 0$.

We use the **Markov Assumption** to simplify the problem:

$$P(y_i|y_1, y_2, \ldots, y_{i-1}) \approx P(y_i|\underbrace{y_{i-(n-1)}, \ldots, y_{i-1}}_{n-1})$$

This means that the probability of the next word depends only on the previous $n-1$ words. This leads to the definition of **N-Gram Models**, where n is a hyperparameter that defines the size of the context window.

The most common choices are:

- **Unigram** ($n = 1$): assumes that each word is independent of the others. The probability of a word is estimated based on its overall frequency in the corpus.
- **Bigram** ($n = 2$): considers the previous word to predict the next one. The probability is estimated based on the frequency of word pairs.
- **Trigram** ($n = 3$): takes into account the two preceding words. The probability is estimated based on the frequency of word triplets.

Observation:

There is the problem of dealing with words at the beginning and at the end of a sentence. Consider the following example:

- "<s> I am Sam </s>";
- "<s> Sam I am </s>";
- <s> I do not like this course </s>";

Then, considering a bigram model, we have:

- $P(I| < s >) = \frac{2}{3}$;
- $P(Sam| < s >) = \frac{1}{3}$
- $P(am|I) = \frac{\text{count(I am)}}{\text{count(I)}} = \frac{1}{3}$;

and so on.

Remark. When using n-grams with $n > 2$ and padding, we need to add at the beginning of the sentence the relative numbero of <s> tokens.

If the **context** of the n-gram is too long, we are just generating the sentences we saw, leading to probabilities close to 1 for those sentences and close to 0 for all the others. This is known as the **overfitting problem** and is a sign that n-grams don't generalize well. A common solution is the generation by **iterative sampling**: sample the next word from the predicted distribution and then use it as context for the next prediction. This sequence ends when we sample the end token </s>.

Another problem is the fact that we use **log probabilities**, leading to numerical instability when dealing with zeroes. This is particularly relevant since the n-gram probability vector is **sparse**.

MISSING: Example of trigram

To solve the problem of zero probabilities, we can also use

- the **Laplace Smoothing** technique, which consists in adding a small constant (usually 1) to the count of each n-gram. This ensures that no n-gram has a zero count, thus avoiding zero probabilities.

$$P(y_i|y_{i-(n-1)},\dots,y_{i-1}) \approx \frac{\text{Count}(y_{i-(n-1)},\dots,y_{i-1},y_i)+1}{\sum_{w\in\mathcal{V}}(\text{Count}(y_{i-(n-1)},\dots,y_{i-1},w)+1)}$$

- the **α-Smoothing** technique, which is a generalization of Laplace Smoothing. Instead of adding 1 to each count, we add a small constant $\alpha > 0$. This allows for more flexibility in controlling the amount of smoothing.

$$P(y_i|y_{i-(n-1)},\dots,y_{i-1}) \approx \frac{\text{Count}(y_{i-(n-1)},\dots,y_{i-1},y_i)+\alpha}{\sum_{w\in\mathcal{V}}(\text{Count}(y_{i-(n-1)},\dots,y_{i-1},w)+\alpha)}$$

- using **Interpolation** between different n-gram models. For example, we can combine a bigram and a unigram model to estimate the probability of the next word:

  $$P(y_i|y_{i-1}) = \lambda P_{bigram}(y_i|y_{i-1}) + (1-\lambda)P_{unigram}(y_i)$$

  where $\lambda \in [0,1]$ is a hyperparameter that controls the balance between the two models. This is useful since when we go into unigrams, we don't have zeroes anymore.
- using **Stupid Backoff**, which is a simpler version of interpolation. In this approach, if the higher-order n-gram has a zero count, we "back off" to the lower-order n-gram without any interpolation. For example, in a trigram model:

  $$P(y_i|y_{i-2},y_{i-1}) = \begin{cases} \frac{\text{Count}(y_{i-2},y_{i-1},y_i)}{\text{Count}(y_{i-2},y_{i-1})} & \text{if } \text{Count}(y_{i-2},y_{i-1}) > 0 \\ \alpha\frac{\text{Count}(y_{i-1},y_i)}{\text{Count}(y_{i-1})} & \text{otherwise} \end{cases}$$

  where α is a discount factor (usually set to 0.4) that reduces the probability mass when backing off.

∞-Gram *J.Liu et al,COLM,2024*

$$P_\infty(y_i|y_1,\ldots,y_{i-1}) = \frac{count(y_1,\ldots,y_{i-1},y_i)}{count(y_1,\ldots,y_{i-1})}$$

where

$$i = \max\{j|y_j \text{ is the last token before } y_i\}$$

⚠ Warning: ***Problems of N-Grams***

- Highly dependent on the dataset;
- Many reasonable sentences are rate 0-probable;
- Depend a lot on the choice of n;

Bibliography

[1] Daniel Jurafsky et al. *Speech and Language Processing: An Introduction to Natural Language Processing, Computational Linguistics, and Speech Recognition*.