Natural Language Processing

Giovanni Novati

A.Y. 2025/2026

1 Introduction

1.1 The informative stratification of language

Natural language is made of elements, where each one is the abstraction of another one. The main elements are the following:

- **Alphabet**: the set of symbols on which a whole language is built on.
- **Word**: the smallest unit of meaning composed of one or more letters.
- **Morphology**: words with the same meaning can change form depending on context. ***E.g.*** *"to be", "are", "is". These are all different words with similar meaning, but that change based on the subject.*
- **Parts of speech**: each word has different functions into a sentence, based on the context and the position. ***E.g.*** *words can be nouns, verbs, prepositions, adjectives, etc.*
- **Syntax**: is the logical dependence between words in a sentence.
- **Semantics**: represents the meaning of a sentence.
- **Pragmatic and speech acts**: understand the context and the speaker intentions.

But what is the meaning? It can be obtained from real world experiences or can be retrieved using other data or concepts, with no reference to the outside world. The former method can be achieved using a *base knowledge system*, built on concepts using databases like Wikidata or Conceptnet. The problem with this method is that is almost impossible to encode the entirety of world knowledge into a machine, both in terms of completeness and correctness (as some data may become outdated). The latter can be achieved by describing words through other words and sentences. Associations between words can be obtained from precompiled lexical resources such as WordNet, or computed automatically from large corpora using methods like *word embeddings*.

It is difficult for a machine to understand the context and the speaker's intentions. Humans often express hidden sentiments and sarcasm while interacting with each other.

This happens more frequently when speaking, but also in text.

Word sense disambiguation: a word can have multiple and different meanings, or a single concept can be described by many different words. ***E.g.*** *the concept of father can be described using the words "dad", "daddy", "papa", etc.*

One of the first chatbots, ELIZA, relied only on patter matching expressions. These patterns may seem "natural enough" the first times, but after a few interactions you can start seeing them. Our main goal is building a model that can understand the context of a text without regular expressions. When we (humans) start learning the language, we first learn from examples. Only once we have familiarity with the language we start polishing it by studying grammar rules to polish it. That's *human learning*, and we want to adopt the same approach to computers using *machine learning*.

2 Tokenization

Suppose we have a list of m documents, and we want to split them into a list of *meaningful* substrings of equal or different length. These substrings are called *tokens*, that are saved in structures called *dictionaries* or *bags of words*.

A dictionary is a set of all the tokens extracted from the m documents, while the bag of words contains all the tokens with their absolute frequency.

Type-token ratio: is the number of unique tokens divided by the number of total tokens in the data. It represents how much variety there is in the data.

Given a set of m documents and a dictionary size of n computed on all m documents, we can create a matrix of size $m \times n$ where each entry x_{mn} represents the occurrences of the token n in the document m. This is called Document-Term Matrix (DTM).

Using this matrix, we can compare document similarity by calculating distances between their feature vectors. The underlying assumption is that similar documents will have feature vectors with comparable values, resulting in smaller distances.

2.1 Feature selection and sparsity

The more features we have, the more the space becomes sparse. If the space is sparse, it means that the vectors representing the documents are sparse (with almost all values at zero, except a few). This means that the vectors are orthogonal to each other, and we can't effectively compare them (the distance is almost the same).

We have to choose what type of features we want to consider:

- **Characters**: we would have a 26-dimension space (matrix $m \times 26$). The matrix is small, and it will be dense.

- **Documents**: there will be only one feature per document. The result matrix dimensions will be $m \times m$, and it will be sparse.
- **Words**: is a good compromise for a feature.
- **N-grams**: combination of n consecutive words.
- **Sentences**: higher-level features that may be too coarse.

2.2 N-grams and meaningfulness

N-grams (shingles): combination of n consecutive words. N-grams give more context about single words. Furthermore, some words only have meaning if considered in combination. ***E.g.*** *if we consider 2-grams, "New York" has a sense, but considering single words "New" and "York" does not give us the original meaning of the sentence.*

But other times, some 2-grams may be useless. ***E.g.*** *"the table".*

How can we keep the 2-grams "New York" but consider the words "the" and "table" separately?

We can use a frequency-based approximation of an n-gram that considers the sum of all the probabilities of a 2-gram to be relevant together divided by all the possibilities these words could have been together.

$$\text{Meaningfulness of the 2-gram "New York"} = P(\text{new, york}) \times \log\left(\frac{P(\text{new, york})}{P(\text{new}) \times P(\text{york})}\right)$$

So, to answer our question of "What feature we want to consider?" we have an answer: single words mixed with meaningful n-grams.

2.3 What is a word?

But now there's another problem: what is a word? Is the word "play" and "play!" the same word or not? Is "play" and "plays" the same word or not? They give and transmit the same concept. So we want a token to be an approximation of a word. What we want to consider changes based on our main goal.

2.4 Distance measures

There are lots of methods with which you can compute the distance (similarity) between two vectors. A smaller distance indicates a higher similarity, while a larger distance indicates a lower similarity. The most common ones are:

- **Euclidean distance**: it's the length of a line segment between two points in Euclidean space. Given an n-dimensional space, is computed using

$$D_e(\mathbf{p}, \mathbf{q}) = \sqrt{(p_1 - q_1)^2 + (p_2 - q_2)^2 + \ldots + (p_n - q_n)^2}$$

- **Cosine distance**: it's the cosine of the angle between the two non-zero vectors. The cosine distance (d_c) can take values ranging from 0 up to 2, and is computed using the cosine similarity (s_c) which values range from -1 to 1. Given an n-dimensional space, is computed by subtracting the cosine similarity from 1.

$$d_c(\mathbf{p}, \mathbf{q}) = 1 - s_c(\mathbf{p}, \mathbf{q})$$

$$s_c(p, q) = \frac{\mathbf{p} \cdot \mathbf{q}}{||\mathbf{p}||\,||\mathbf{q}||} = \frac{\sum_{i=1}^{n} p_i q_i}{\sqrt{\sum_{i=1}^{n} p_i^2}\sqrt{\sum_{i=1}^{n} q_i^2}}$$

2.5 Term frequency and document frequency

Bag of words: list with all the words and their relative frequency (number of occurrences divided by the number of total words). This is called **term frequency (TF)**.

Words with higher frequency are not always useful, while words with less frequency can be more informative. If words with higher frequency are common in most of the documents, we can consider these words not useful for distinguishing between documents.

We want to measure how specific a word is to a particular document. The **document frequency (DF)** represents the number of documents a word appears in divided by the total number of documents.

$$DF = \frac{|\{d : w \in \mathbf{d}\}|}{|\mathbf{d}|}$$

But we want meaningful words to have a high value, and common words to have low values. So we use the **inverse document frequency (IDF)** function:

$$IDF = \log\left(\frac{m}{1 + DF}\right)$$

We use the logarithm to have a smoother curve and to reduce the distance between useful and less useful words.

If we want to know how specific a word is in a certain document relative to the whole corpus (set of documents), we can use **TF-IDF**:

$$TF\text{-}IDF = TF \times IDF$$

3 Tokenization approaches

Given a string of characters, we can first identify single words by splitting the string using spaces. The text is then split into tokens. This is text tokenization. Then, different words may have (almost) the same meaning (e.g., "big", "bigger", "biggest"), so we want to normalize the tokens.

3.1 First approach: Heuristic-based

Each language has typical prefixes/suffixes, so we can use regular expressions to detect these parts of the word. This approach is called **stemming**.

We can use heuristic algorithms (that know the language grammar) to split word endings. ***E.g.*** *the words "big", "bigger" and "biggest". They are all related concepts, but are expressed using slightly different words. If we create a token using the first three letters and consider the remaining letters as separate tokens, we have the tokens "big", "ger" and "gest".*

3.2 Second approach: Linguistics-based

We can apply machine learning to tokenize a text. These tools are called **linguistic-based tokenizers**, and one example is spaCy, available in Python. These models are trained on specific languages.

Using spaCy (or other libraries), you can get multiple pieces of information about a single token:

- **Text**: the token itself, as it was in the text.
- **Lemma**: dictionary entry of that word. Considering the lemma instead of the token reduces the feature vector size.
- **Pos/Tag**: identifies the part of speech (noun, verb, adjective)
- **Dependency/Head**: establishes the dependencies between words. In the sentence "The cat eats the mouse", "mouse" depends on "the" and "eats", "eats" depends on "cat", and "cat" depends on "the".
- **Alpha**: boolean that tells whether the token is alphanumeric.

Noun chunks: multiple words representing a concept, that can be non-continuous in the original document. This is computed using the *dependency tree*.

Named entity recognition: common names used in real world to represent objects or concepts. ***E.g.*** *"Apple" can be categorized as a company, while "Italy" is recognized as a country.*

To recap, the first approach only works using heuristics and its execution is very fast. On the other hand, the second approach is a bit slower (it uses machine learning techniques) but it can better understand the meaning of the text, and contextualize words. It is prone to errors.

3.3 Third approach: Learn-based

The learning-based approach, also called *subword tokenization*, represents the state-of-the-art in tokenization for modern language models like BERT, GPT. This approach addresses fundamental limitations of traditional word-based tokenization by learning optimal subword units directly from the training corpus.

Traditional word-based tokenization suffers from several critical issues. First, it creates enormous vocabularies (potentially millions of unique words), leading to sparse representations and computational inefficiency. Second, it cannot handle out-of-vocabulary (OOV) words that weren't seen during training. Third, it fails to capture morphological relationships between related words like "play", "playing", and "played".

Subword tokenization solves these problems by operating on the principle that frequently used words should remain as single tokens, while rare words should be decomposed into meaningful subword units. For example, "annoyingly" might be decomposed into "annoying" and "ly", where both components appear more frequently in the corpus while preserving the original meaning.

This approach enables models to:

- Maintain reasonable vocabulary sizes (typically 30,000-50,000 tokens)
- Handle any input text, including previously unseen words
- Capture morphological patterns and linguistic structures
- Achieve better performance on multilingual tasks

3.3.1 WordPiece

WordPiece is a subword-based tokenization algorithm developer by Google, and commonly used in transformer-based models like BERT. It improves efficiency by balancing vocabulary size and representation power. The goal of training is to construct an optimal vocabulary from raw text data, balancing vocabulary size with subword efficiency.

Training Algorithm:

1. **Initialize dictionary**: the initial vocabulary contains all single Unicode characters plus special tokens like $[UNK]$.
2. **Compute frequency**: scan the training corpus and count the occurrence of every pairs of tokens (initially each char is seen as a token), by marking with a special

character # the tokens that are not the initial ones.

3. **Merge pairs**: merge the two most frequent pair and update the vocabulary and the corpus by substituting this sequence with a single token.
4. **Repeat**: repeat points 3 - 4 until the maximum dictionary size is reached.

This learning-based approach has become the foundation for all modern large language models, enabling them to achieve remarkable performance across diverse linguistic tasks while maintaining computational efficiency.

3.3.2 Byte Pair Encoding (BPE)

BPE, originally a data compression technique from 1994, was adapted for NLP and is now used in many models. The algorithm works by iteratively merging the most frequent pairs of characters or subwords until reaching a desired vocabulary size. This algorithm does not mark tokens that are not the initial one with a different character.

3.4 Limitations of bag of words representation

Despite this algorithms, there are still some limitation with tokenization:

- Even with these tokenization methods, the vector space is mostly sparse. Most documents only use a small subset of the vocabulary, leading to vectors with many zero values.
- If two tokens have a similar meaning but are represented in a different way, the two features will be orthogonal to each other. The machine will treat them as totally different words. ***E.g.*** *The words "game" and "match" are similar, but are represented in a totally different way.*
- Words with multiple meanings (polysemy) are treated as single tokens and context-dependent meaning variations cannot be captured in the token representation.
- Sentences that use same words but in a different order will result in a identical feature vector, even though the meaning may be different. ***E.g.*** *"Mary loves John" and "John loves Mary" have different meanings but identical feature vectors.*

4 Classification

Given a corpus, we may want to assign one or more labels to each document, based on a set of feature; this is called a classification problem. It can be modeled as a function that provides a probability distribution over the target given the features, such as

$$f : X \to \Delta_Y$$

where X denotes the features and Y the labels. So, given a single input $x \in X$, $f(x)$ returns a vector $p(x) \in \Delta_Y$ such that

$$\mathbf{p}(x) = \begin{bmatrix} p_1(x) \\ p_1(x) \\ \vdots \\ p_1(x) \end{bmatrix}$$

since the label vector is a probability distribution, we know that

$$\sum_{i=1}^{|Y|} p_i(x) = 1 \quad \text{and that} \quad p_i(x) \geq 0 \ \forall i \in 1 \ ... \ |Y|$$

A classification problem can be subdivided into four main categories:

- **Binary partition**: each input data point can be assigned to one label only among two possible labels. ***E.g.*** *A fake news detector or a spam filter. In both examples you show or hide the content, there's no intermediate state or partial classification.*
- **Soft binary classification**: each input data point can be assigned to one or two labels among two possible labels. ***E.g.*** *Search engines work by comparing the relevance of different websites. There's no hard partitioning (relevant/irrelevant), but the relevance of a certain content is given as probability (it is 65% relevant and 35% irrelevant).*
- **Multi-class partition**: each input data point can be assigned to one lable only among many. ***E.g.*** *A news article classifier that assigns each article to exactly one category such as Politics, Sports, Technology, Entertainment, etc.*
- **Soft multi-label classification**: each input data point can be assigne to multiple labels among many. ***E.g.*** *A research paper tagger that predicts the probability of belonging to various subjects like Machine Learning, Data Mining, Computer Vision, and Natural Language Processing. A single paper about vision transformers could have high probabilities for both Computer Vision (92%) and Natural Language Processing (85%).*

In the last example we can see that the probabilities of an article belonging to a specific topic do not add up to 1, like we saw in the multi-class partition. That's because a soft multi-class classification problem can be modeled as a multiple binary classification problems (independent yes/no predictions). Each label's probability indicates the independent likelihood the item belongs to that category, so they do not need to sum to 1. If we want that property, we can normalize it by applying the *softmax* function.

So, our goal is to transform the input feature vector of size m into an output vector of size n, representing the probability distribution over all the labels. We can achieve

this transformation through matrix multiplication. Given an input vector of size m and desired output of size n, we need a weight matrix $\mathbf{W}$ of size mn. The computation follows

$$\begin{bmatrix} i_1 \\ i_2 \\ \vdots \\ i_m \end{bmatrix} \cdot \begin{bmatrix} w_{11} & w_{12} & \dots & w_{1m} \\ w_{21} & w_{22} & \dots & w_{2m} \\ \vdots & \vdots & \ddots & \vdots \\ w_{n1} & w_{n2} & \dots & w_{nm} \end{bmatrix} = \begin{bmatrix} o_1 \\ o_2 \\ \vdots \\ o_n \end{bmatrix}$$

Each element w_{ij} in the weight matrix represents the learned importance of feature i for predicting label j. By multiplying the input vector with the weight matrix, we get the final label vector. In a more general form, the definition of a linear classifier is

$$x\mathbf{W} = y$$

When we apply a function g to the linear output ()***E.g.*** *A threashold function*), we get

$$g(x\mathbf{W}) = y$$

That structure is called a **neural network**. Neural networks provide a flexible framework for building complex classifiers, though they are not the only method available. Alternative approaches include more traditional classifiers like *Naive Bayes*, *Decision Trees*, etc.

These "classic" classifiers often have the advantage of being inherently explainable - you can understand exactly why a particular input received a specific classification by examining the model's decision process. In contrast, complex deep neural networks can be extremely difficult to interpret, making it challenging to understand the reasoning behind their predictions.

4.1 Cuisine classification example

Suppose we have m recipes that contain a list of ingredients, and we want to classify these recipes on a set of n cuisine types or ethnicities. We can encode features (the ingredients) using term frequency, so for each recipe we will have a vector of zeros and ones indicating wether an ingredient was used or not. We can then use the TF-IDF to discover if an ingredient is typical of a certain cuisine or is common in all of them (in this last case it won't tell us much about the cuisine type). ***E.g.*** *"Salt" will be used in all cuisine, while "pizzoccheri" will be more typical of the italian cuisine.*

4.2 Neural network training

Initially, the weight matrix $\mathbf{W}$ (also denoted as $\mathbf{\Theta}$) used for matrix multiplication contains random values. The training process involves the following iterative steps:

1. **Forward pass**: Compute the output vector by multiplying the input vector with the current weight matrix.
2. **Error computation**: Calculate the difference between the predicted output and the expected output (ground truth).
3. **Gradient computation**: Determine the direction to adjust each parameter by computing the partial derivative of the loss function with respect to each parameter (*gradient descent*).
4. **Parameter update**: Modify the weight values using the gradient and a *learning rate* η that controls the step size of updates.

To prevent excessive sensitivity to individual variations and improve convergence stability, parameters are not updated after every single input. Instead, the process uses **batches**: the model processes n inputs, averages the errors, and then updates the parameters once per batch.

4.2.1 Training monitoring

Determining when to stop training requires careful monitoring of both training and validation performance. During training, we compute the **training loss** on the data used for parameter updates. However, relying solely on training loss can lead to *overfitting*, where the model memorizes the training data but fails to generalize to new examples.

To detect overfitting, we evaluate the model on a separate **validation set** that was not used during training. The goal is not to optimize performance on the test data, but to ensure the model can generalize to previously unseen examples.

4.3 Evaluation metrics

The correct label for each example in our dataset is called the **ground truth**. For binary classification problems, we can categorize predictions into four types:

- **True Positive (TP)**: Correctly identified positive cases
- **False Positive (FP)**: Incorrectly identified as positive (Type I error)
- **True Negative (TN)**: Correctly identified negative cases
- **False Negative (FN)**: Incorrectly identified as negative (Type II error)

From these basic metrics, we derive several important evaluation measures:

$$\text{Precision} = \frac{TP}{TP + FP}$$

$$\text{Recall} = \frac{TP}{TP + FN}$$

$$\text{Accuracy} = \frac{TP + TN}{TP + FP + TN + FN}$$

High recall and low precision means the model captures most positive cases but includes many false positives. Many correct classifications are missed, but most predictions marked as positive are actually correct.

Low recall and high precision means the model is conservative in making positive predictions. When it does predict positive, it's usually correct, but it misses many actual positive cases.

4.3.1 Confusion matrix

A **confusion matrix** is an $n \times n$ matrix (where n is the number of classes) that shows the relationship between actual and predicted classifications. Each row represents the actual class, while each column represents the predicted class. The matrix reveals where the system gets "confused" between different classes.

For a binary classification problem, the confusion matrix has the following structure:

		Predicted	
		Positive	Negative
Ground truth	Positive	TP	FN
	Negative	FP	TN

4.4 Classification challenges

Several challenges commonly arise in classification problems:

- **Unbalanced datasets**: When some classes have significantly more examples than others, leading to biased models that favor the majority class.
- **Feature selection**: Determining which features are most informative for classification while avoiding curse of dimensionality.
- **Overfitting**: Models that perform well on training data but poorly on new, unseen data.

- **Interpretability**: Understanding why a model makes specific predictions, especially important in sensitive applications.

4.5 Model explainability

There are two main types of explanations in machine learning:

- **Local explanation**: Understanding why a specific input received a particular classification. For simple neural networks, this can be achieved by examining the weight matrix $\mathbf{W}$, but becomes extremely difficult with complex deep networks.
- **Global explanation**: Identifying the most relevant features for predicting each class across the entire dataset. This helps understand the model's general decision-making patterns.

Traditional "classic" classifiers often provide built-in explainability, making them valuable in domains where understanding model decisions is crucial, such as medical diagnosis or legal applications.

5 Statistical language models

Until now, for document classification, we have considered features without any regard to their order. However, the sequence of words carries crucial information about meaning and structure. Statistical language models aim to capture this sequential nature by computing the probability of word sequences.

5.1 Sequence probability estimation

Consider a sequence of words $w_0, w_1, w_2, \ldots, w_n$. We want to measure the probability of this particular sequence occurring: $P(w_0, w_1, w_2, \ldots, w_n)$.

The simplest approach assumes independence between words:

$$P(w_1, w_2, \ldots, w_n) = \prod_{i=1}^{n} P(w_i)$$

where the probability of a single word w in a corpus D of length N is:

$$P(w) = \frac{\text{count}(w)}{\sum_{i=1}^{N} \text{count}(w_i)}$$

However, this independence assumption ignores the contextual relationships between words, which are fundamental to natural language.

5.2 Conditional probability and language models

A more sophisticated approach uses conditional probability to model word dependencies. Given a starting word w_0, we can compute the probability of the next word conditioned on the previous context:

$$P(w_1|w_0) = \frac{\text{count}(w_0, w_1)}{\text{count}(w_0)}$$

This extends to longer sequences using the chain rule of probability:

$$P(w_0, w_1, \ldots, w_n) = P(w_0) \prod_{i=1}^{n} P(w_i|w_0, w_1, \ldots, w_{i-1})$$

This formulation defines a **statistical language model**: a probability distribution over sequences of words that captures the likelihood of different word combinations in natural language.

To handle sequence boundaries, we introduce special tokens [START] and [END] at the beginning and end of each sequence. This allows the model to learn appropriate sentence beginnings and endings.

5.3 Training and text generation

The training process involves analyzing the corpus to compute all conditional probabilities. The model memorizes word sequences and their frequencies, storing the probability of each possible next word given various contexts.

For text generation, we can employ different strategies:

- **Greedy selection**: always choose the highest probability next word. This produces deterministic but potentially repetitive output.
- **Probabilistic sampling**: sample words according to their computed probabilities. This introduces variability but may produce less coherent text.
- **Top-k sampling**: consider only the k most probable next words, renormalize their probabilities, and sample from this reduced set. This balances coherence and diversity.

5.4 Classification with language models

Language models can serve as document classifiers by training separate models for each class. Given a document to classify:

1. Train one language model per class using documents from that class

2. Compute the probability of the test document under each class-specific model
3. Assign the document to the class with the highest probability

This approach captures sequential information that bag-of-words methods miss, potentially improving classification accuracy for tasks where word order matters.

5.5 Limitations and challenges

Statistical language models face several fundamental problems:

- **Exponential memory growth**: storing all possible sequences and their frequencies requires exponentially increasing memory as sequence length grows.
- **Data sparsity**: long sequences rarely appear in training corpora, making probability estimation unreliable for extended contexts.
- **Predictability**: as sequences grow longer, the model tends to generate increasingly standard and predictable continuations.

5.6 Solutions and extensions

Two primary approaches address these limitations:

Markov language models: instead of conditioning on the entire preceding sequence, limit the context to a fixed window of the last k words (typically $k = 2$ or $k = 3$). This reduces memory requirements and improves probability estimation reliability. However, it sacrifices long-term dependencies, potentially making the end of generated sequences unrelated to their beginnings.

The complete sequence probability under a Markov language model becomes:

$$P(w_1, w_2, \ldots, w_n) = \prod_{i=1}^{n} P(w_i | w_{i-k}, \ldots, w_{i-1}) \tag{1}$$

Neural language models: use neural networks to learn distributed representations of words and contexts, enabling better generalization and handling of longer dependencies. These models can capture complex patterns while maintaining computational tractability.

Modern language models combine the probabilistic foundations of statistical models with the representational power of neural networks, leading to the sophisticated systems used in contemporary natural language processing applications.

6 Neural language models

Modern language models leverage neural networks to overcome the limitations of statistical approaches. Instead of simply storing word sequences and their frequencies, neural networks can learn distributed representations that capture semantic relationships between words.

6.1 Neural word prediction approaches

Using neural networks, we can predict words in different contexts through three main approaches:

6.1.1 Next word prediction

In this approach, we predict the next word given the previous words in a specific sequential order. The model processes words from left to right, maintaining the temporal structure of language. ***E.g.*** *Given the sequence "The pasta at the", the model predicts "restaurant" as the most likely next word.*

6.1.2 Continuous Bag of Words (CBOW)

CBOW predicts a target word given its surrounding context words, without considering their order. We use a sliding window approach where the target word is at the center. ***E.g.*** *Given the sentence "The pasta at the restaurant was good today", to predict "restaurant", we may want to use a window of size 5. In this case the context words are "pasta, at, was, good" and we want to predict the central word "restaurant".*

The sliding window moves through the entire sentence, and for each position, we predict the central word based on its surrounding context.

6.1.3 Skip-gram

Skip-gram works in the opposite direction: given a single word, we predict its surrounding context words. This approach learns which words are likely to appear together. ***E.g.*** *Given the word "restaurant", the model should predict the context words "at", "the", "was", "good".*

6.2 Word2Vec architecture

Let's examine the neural network architecture for the skip-gram model, which forms the foundation of Word2Vec.

The network structure consists of:

- **Input layer**: a one-hot encoded vector representing the entire dictionary, where only one position is set to 1 (the input word) and all others are 0.
- **Hidden layer**: a smaller dense layer that compresses the input representation.
- **Output layer**: a vector the size of the dictionary representing the probability distribution over all possible context words.

The key insight is that the hidden layer is deliberately smaller than both input and output layers. This compression forces the network to learn meaningful representations, as it must encode the essential information about each word in a reduced dimensional space.

During training, the same input word will appear with different context words across different training examples. The network learns to associate words that frequently appear together, building rich semantic relationships.

The output layer produces a probability distribution over the entire vocabulary, making this a **classification problem**. We can either select the word with the highest probability or sample from the top-k most probable context words.

This implementation is equivalent to a 1-gram Markov Language Model, but with learned distributed representations instead of simple frequency counts.

6.3 Word embeddings

The hidden layer in Word2Vec produces what we call **word embeddings**: dense vector representations of words that capture semantic meaning. Where one-hot encoding represents every word as an orthogonal vector in a high-dimensional space, embeddings represent words in a lower-dimensional space where geometric relationships encode meaning.

Words with similar meanings will have similar vector representations in this embedding space. In the learned vector space:

- **Similar words** are represented by vectors that are close to each other (they are not orthogonal).
- **Dissimilar words** are represented by vectors that are far apart.
- **Semantic relationships can be captured through vector arithmetic.** The structure of the embedding space allows for meaningful mathematical operations on word vectors. This works because the training process organizes words such that directions in the vector space correspond to abstract semantic concepts (e.g., gender, tense, or royalty).
  - **Analogy and relationships (addition/subtraction):** we can add and subtract vectors to navigate this semantic space and solve analogy tasks.

The most famous example is:

$$\text{vec}(\text{“King”}) - \text{vec}(\text{“Man”}) + \text{vec}(\text{“Woman”}) \approx \text{vec}(\text{“Queen”})$$

In this operation, subtracting the “Man” vector from the “King” vector isolates the abstract concept of “royalty”. Adding the “Woman” vector then applies this concept to a female context, resulting in a vector that is geometrically close to that of “Queen”.

– **Compositional meaning (averaging):** we can average the vectors of multiple words to get a single vector representing the combined meaning of a phrase or sentence. For example, averaging the vectors for “boat”, “water”, and “fish” would produce a new vector that captures the general concept of “fishing” or “boating”. This technique is useful for document classification or creating representations for larger chunks of text from their constituent words.

This distributed representation solves many limitations of bag-of-words approaches. The model can generalize to unseen word combinations by leveraging learned semantic similarities, and rich relationships between words can be discovered and utilized. Using a distance metric like **cosine similarity**, we can measure how close two concepts are. We can also cluster the vectors in this space to discover abstract concepts automatically.

6.3.1 Analyzing semantic change across corpora

Word embeddings can be used to track shifts in word meaning over time (diachronic analysis). For example, the word “portal” has a different primary meaning today (related to the web) than it did a century ago. There are several ways to compute this change:

- **Option 1 (Independent models with alignment)**: train two separate Word2Vec models, one on each corpus (e.g., a historical one and a modern one). Because the vector spaces are initialized randomly, they are not directly comparable. One must be translated/rotated to align with the other before comparing the vectors for a specific word.
- **Option 2 (Context comparison)**: train two separate models. For a given word, input it into both models and compare the lists of predicted context words. The differences in the context lists reveal the shift in how the word is used.
- **Option 3 (Finetuning a shared model)**: train a single model on both corpora combined. Then, create two copies of this model and finetune one on the historical corpus and the other on the modern one. This provides a common starting point, allowing for direct vector comparison (e.g., with cosine distance) to quantify the semantic shift. This method is most effective when the word’s frequency is not drastically different between the two corpora.

7 Recurrent Neural Networks (RNNs)

While Word2Vec captures relationships between words, it operates on a local context and is fundamentally a bag-of-words approach. It does not inherently understand the order of words, which is critical for understanding language. To model sequences, we turn to **Recurrent Neural Networks (RNNs)**.

Not all sequences benefit from sequential analysis. If elements are independent (e.g., a series of coin tosses), a bag-of-words approach is sufficient. However, in natural language, there are often long-term dependencies where early words in a sequence are crucial for understanding later parts. RNNs are designed to learn from the entire sequence.

Sequence learning problems can be categorized as:

- **Sequence-to-vector (seq2vec)**: an input sequence is mapped to a single output vector (e.g., sentiment analysis of a sentence).
- **Vector-to-sequence (vec2seq)**: a single input vector is used to generate an output sequence (e.g., image captioning).
- **Sequence-to-sequence (seq2seq)**: an input sequence is mapped to an output sequence (e.g., machine translation).

7.1 The basic RNN model

A naive approach to sequence processing might be to apply a standard neural network (like Word2Vec) to each token in parallel. However, this fails because each output token would be generated independently, without awareness of its context in the sequence.

RNNs solve this by introducing a **recurrent** connection. At each step t, the network's output depends on two inputs: the current token x_t and the hidden state from the previous step, h_{t-1}. This hidden state acts as the network's memory, carrying information from all previous steps.

The hidden state at time t is computed as follows:

$$h_t = g(\Theta_{hh} h_{t-1} + \Theta_{xh} x_t)$$

The final output at that step, y_t, is then typically computed from the hidden state:

$$y_t = g'(\Theta_{hy} h_t)$$

This requires three sets of learned parameters: weights for the hidden-to-hidden connection (Θ_{hh}), input-to-hidden connection (Θ_{xh}), and hidden-to-output connection (Θ_{hy}). This sequential dependency means the process is inherently not parallelizable.

7.2 Training and challenges

RNNs are trained by unrolling the network through time for a full sequence (or batch of sequences). The error is computed at the end, and gradients are backpropagated through all the time steps to update the weights. This process introduces several challenges:

- **Vanishing or exploding gradients**: during backpropagation, the gradients are repeatedly multiplied by the recurrent weight matrix. If the values are small, the gradient can shrink to zero (vanish), preventing the network from learning long-term dependencies. If they are large, the gradient can grow exponentially (explode), destabilizing training.
- **Recency bias**: the network's state is often most influenced by recent inputs. The "memory" of earlier tokens in the sequence fades over time.

To address these issues, more advanced architectures were developed, such as **Long Short-Term Memory (LSTM)** and **Gated Recurrent Units (GRU)**. These models introduce gating mechanisms—learnable parameters that control whether to keep, discard, or update information in the hidden state. For instance, a gate might learn to ignore an uninformative word like "the" and preserve the memory of a more meaningful word from earlier in the sequence.

7.3 Encoder-decoder architecture

A powerful application of RNNs is the **encoder-decoder** architecture, which is fundamental to seq2seq tasks like machine translation.

- **Encoder**: an RNN reads the entire input sequence token by token. Its final hidden state is a vector that embeds the meaning of the whole sequence, often called a "context vector" or "thought vector". Basically it tries to predict the same input sequence, shifted by one character.
- **Decoder**: another RNN is initialized with the encoder's context vector. It then generates an output sequence token by token. At each step, its input is the previously generated token and its own hidden state, allowing it to generate a coherent sequence based on the input's meaning.

A key advantage of this architecture is that the input and output sequences can have different lengths, which is essential for tasks like translation.

7.4 Generating output sequences

The decoder outputs a probability distribution over the entire vocabulary at each step. To construct the final sequence, we need a selection strategy:

- **Greedy search**: at each step, simply choose the word with the highest probability. This is fast but may not lead to the optimal overall sequence.

- **Beam search**: at each step, keep track of the n most probable partial sequences (the "beam"). In the next step, expand each of these sequences (not just tokens!) and again keep only the top n overall. This is more computationally expensive but explores a larger search space and often produces better results.

7.5 Attention mechanism

A limitation of the basic encoder-decoder model is that the entire meaning of the input sequence must be compressed into a single, fixed-size context vector. This is a bottleneck, especially for long sequences, as the memory of early tokens can be lost.

The **attention mechanism** solves this. Instead of just using the encoder's final hidden state, the decoder is given access to *all* of the encoder's hidden states from every time step. At each step of generating an output token, the decoder calculates a set of "attention scores", a weighted average over the encoder's hidden states. These scores determine which input tokens are most relevant for generating the current output token.

E.g. *If answering "Is Maria going to the sea?" to "Yes, she said she wanted to go", when the decoder generates the word "she", the attention mechanism would likely place a high weight on the input token "Maria". When generating "go", it would attend more to "going".*

The attention weights are recalculated for every token the decoder generates, allowing the focus to shift dynamically. There are two main types of attention:

- **Cross-attention**: maps the importance of each input token to the current output token.
- **Self-attention**: maps the importance of tokens within the same sequence (either input or output) to each other, capturing internal dependencies.

Attention scores are themselves learned during training, typically by multiplying matrices of the hidden state vectors. This allows the model to learn complex alignments between input and output sequences. A common method for evaluating the quality of generated sequences is the **BLEU score**, which compares n-grams from the predicted and ground-truth sequences.