## Course Overview

**Course Name:** Natural Language Processing and Large Language Models

**Corso di Laurea Magistrale in Ingegneria Informatica**

**Instructors:**


- **Nicola Capuano** (DIEM – University of Salerno)
  Email: ncapuano@unisa.it
- **Antonio Greco** (DIEM – University of Salerno)
  Email: agreco@unisa.it


**Group for the project:**


- Group 6
  - Maximillian Marius Moraru, 0622702167, m.moraru@studenti.unisa.it
  - Marco Magliulo, 0622702429, m.magliulo2@studenti.unisa.it
  - Christian Romano, 0622702329, c.romano50@studenti.unisa.it


**Objectives:**

- **Knowledge:**
  - Core concepts in NLP.
  - Understanding and generating natural language.
  - Statistical approaches to NLP.
  - LLMs and transformer-based models.
  - Applications of NLP with LLMs.
  - Techniques like prompt engineering and fine-tuning.
- **Abilities:**
  - Design and implementation of NLP systems using LLMs.

## Course Content

**Fundamentals of NLP:**

1. **Introduction to NLP:** Evolution and Applications.
2. **Core Techniques:**
   - Tokenization, stemming, lemmatization, and POS tagging.
   - Mathematical Models: Bag of Words, Vector Space Model, and TF-IDF.
3. **Applications:**
   - Text classification.
   - Word embeddings (Word2Vec, GloVe, FastText).
   - Neural Networks: RNN, LSTM, GRU, CNN.
   - Information extraction: Parsing, Named Entity Recognition.
   - Advanced Tasks: Question answering and chatbots.
4. **Exercises:**
   - Building a simple chatbot in Python using SpaCy or Rasa.

**Transformers:**

1. **Attention Mechanisms:**
    - Self-attention mechanisms.
    - Encoder-decoder structures.
2. **Seq2Seq Models:**
    - Translation and summarization.
    - Introduction to Hugging Face.
3. **Exercises:**
    - Implementation of text classification, NER, and text generation tasks using transformers.

**Prompt Engineering:**

1. **Techniques:**
    - Zero-shot and few-shot prompting.
    - Chain-of-Thought, Retrieval-Augmented Generation (RAG).
2. **Exercises:**
    - Application of structured prompts and LangChain techniques.

**Fine-Tuning of LLMs:**

1. **Techniques:**
    - Feature-Based Fine-Tuning.
    - Parameter-Efficient Fine-Tuning (PEFT): LoRA, Adapters.
    - Reinforcement Learning with Human Feedback (RLHF).
2. **Exercises:**
    - Fine-tuning LLMs for specific applications.
    - Implementation of a final project.

## Educational Objectives and Outcomes

**Knowledge and Understanding:**

- Fundamental and advanced concepts of NLP and transformers.
- Techniques for developing and fine-tuning LLMs.
- Tools for NLP, including Hugging Face and LangChain.

**Application of Knowledge and Understanding:**

- Design and implementation of NLP systems leveraging LLMs.
- Effective integration of technologies and tools.

## Exam Structure

**Assessment Methods:**

- **Project Work:** Practical application of course methodologies. Includes implementation of an NLP system or chatbot and a detailed report.

- **Oral Examination:** Evaluation of theoretical knowledge, project design choices, and discussion of lecture topics.
- **Final Grade:** Determined by the average of the project and oral exam scores.

### Textbook and Additional Material

- **Textbook:**
  "Natural Language Processing in Action" by H. Lane, C. Howard, and H. M. Hapke.
  **Publisher:** Manning (2019).
  **Second Edition Early Access:** Available at Manning Website.
- **Additional Materials:** Available on the university's e-learning platform (https://elearning.unisa.it).

### Attendance Policy

Attendance is mandatory for at least 70% of the course hours. Participation will be monitored via the EasyBadge system provided by the university.

### Total Course Hours

- **Lectures:** 22 hours.
- **Exercises:** 24 hours.
- **Laboratory:** 2 hours.

### Prerequisites

- **Mandatory:** Machine Learning.

### Key Highlights

- Build a solid understanding of LLMs and NLP methodologies.
- Apply techniques like prompt engineering, transformers, and fine-tuning.
- Develop practical systems through hands-on exercises and projects.
- Gain expertise in tools such as Hugging Face and LangChain.

# NLP Overview

## Natural Language Processing and Large Language Models

**Corso di Laurea Magistrale in Ingegneria Informatica**

**Lesson 1: NLP Overview**

Nicola Capuano and Antonio Greco

DIEM – University of Salerno

# What is Natural Language Processing

## NLP in the Press

### Importance of NLP

> "Natural language is the most important part of Artificial Intelligence."
> *John Searle, Philosopher*

> "Natural language processing is a cornerstone of artificial intelligence, allowing computers to read and understand human language, as well as to produce and recognize speech."
> *Ginni Rometty, IBM CEO*

> "Natural language processing is one of the most important fields in artificial intelligence and also one of the most difficult."
> *Dan Jurafsky, Professor of Linguistics and Computer Science at Stanford University*

## Definitions

1. "Natural language processing is the set of methods for making human language accessible to computers." - *Jacob Eisenstein*
2. "Natural language processing is the field at the intersection of computer science and linguistics." - *Christopher Manning*
3. "Make computers understand natural language to do certain tasks humans can do, such as translation, summarization, and question answering." - *Behrooz Mansouri*
4. "Natural language processing is an area of research in computer science and artificial intelligence concerned with processing natural languages such as English or Mandarin. This processing generally involves translating natural language into data that a computer can use to learn about the world. And this understanding of the world is sometimes used to generate natural language text that reflects that understanding."
   *Natural Language Processing in Action*

# Natural Language Understanding (NLU)

**Definition:** A subfield of NLP focused on transforming human language into a form that machines can process. It involves extracting meaning, context, and intent from text.

**Text is transformed into a numerical representation (embedding).**

**Applications of Embeddings:**

- **Search Engines**: To interpret the meaning behind search queries.
- **Email Clients**: To detect spam and classify emails as important or not.
- **Social Media**: To moderate posts and understand user sentiment.
- **CRM Tools**: To analyze customer inquiries and route them appropriately.
- **Recommender Systems**: To suggest articles, products, or content.

# Natural Language Generation (NLG)

**Definition:** A subfield of NLP focused on generating human-like text.

**Involves:** Creating coherent, contextually appropriate text based on numerical representations of meaning and sentiment.

**Applications:**

- **Machine Translation**: Translates text from one language to another.
- **Text Summarization**: Creates concise summaries of long documents while preserving key information.
- **Dialogue Processing**: Powers chatbots and virtual assistants to provide relevant responses in conversations.
- **Content Creation**: Generates articles, reports, stories, poetry, and more.

# Challenges in NLP

## Ambiguity

**Natural language is extremely rich in form and structure and very ambiguous.**

- **One input can mean many different things.**
- **Many inputs can mean the same thing.**

**Levels of Ambiguity:**

1. **Lexical Ambiguity**: Different meanings of words.
2. **Syntactic Ambiguity**: Different ways to parse the sentence.
3. **Interpreting Partial Information**: How to interpret pronouns.
4. **Contextual Information**: The context of the sentence may affect its meaning.

## Example:

"I made her duck."

Possible interpretations:

1. I cooked waterfowl for her.
2. I cooked waterfowl belonging to her.
3. I created the (plaster?) duck she owns.
4. I caused her to quickly lower her head or body.

5. I waved my magic wand and turned her into undifferentiated waterfowl.

# Applications of NLP

## Industries Benefiting from NLP

- **Healthcare**: Processing patient data for diagnostics and care.
- **Finance**: Fraud detection and sentiment analysis.
- **E-commerce**: Personalized recommendations and chatbots.
- **Legal**: Document analysis and contract review.
- **Education**: Automatic grading and summarization tools.
- **Technology**: Code generation and review tools.
- **Media and Entertainment**: Script generation and personalized content.

# History of NLP

## Key Milestones

1. **1950s**: Early machine translation systems.
2. **1960s**: Chomsky's generative grammar and the ALPAC report.
3. **1990s**: Statistical revolution.
4. **2000s**: Neural networks and word embeddings.
5. **2010s**: Deep learning and transformers.

## Deep Learning Era

- Word2Vec and Transformer architectures revolutionized NLP.
- Virtual assistants like Siri, Alexa, and Google Assistant emerged.

# Large Language Models (LLMs)

## Post-Transformer Advancements

- Scaling enabled LLMs to advance applications like text generation, translation, and chatbots.

## Multimodal Capabilities

- Integration of text, images, audio, and video data (e.g., DALL-E, Whisper, CLIP).

**Examples:**

- **Image-to-Text**: Generating descriptive text from images (e.g., CLIP).
- **Text-to-Image**: Creating images based on textual descriptions (e.g., DALL-E).
- **Audio-to-Text**: Converting spoken language into written text (e.g., Whisper).

- **Text-to-Audio**: Generating audio, such as music, from textual descriptions (e.g., Jukebox).
- **Video-to-Text**: Creating textual descriptions or summaries from video content.
- **Text-to-Video**: Generating video content from textual descriptions.

## NLP Market

**NLP is a promising career option:**

- Growing demand for NLP applications.
- Projected employment growth of 22% between 2020 and 2030.

**NLP Market Global Forecast:**

- Estimated in billions of USD.

## Future of NLP

### Advancements

- Continued integration with AI and multimodal systems.
- Expanding applications in healthcare, education, and automation.

### Key Areas of Focus

- Enhancing contextual understanding.
- Reducing biases in models.
- Improving scalability and efficiency.

# Representing Text

## Tokenization

### Prepare the Environment

For most exercises, we will use Jupyter notebooks:

- Install the Jupyter Extension for Visual Studio Code.
- pip install jupyter
- Create and activate a virtual environment:
    - python -m venv .env
    - source .env/bin/activate
- Alternative: Google Colab notebooks.
    - https://colab.research.google.com/

For this section, we also need some Python packages:

- pip install numpy pandas

## Text Segmentation

**Definition:** The process of dividing a text into meaningful units.

1. **Paragraph Segmentation**: Breaking a document into paragraphs.
2. **Sentence Segmentation**: Breaking a paragraph into sentences.
3. **Word Segmentation**: Breaking a sentence into words.

#### Tokenization

- A specialized form of text segmentation.
- Involves breaking text into small units called tokens.

#### What is a Token?

A unit of text that is treated as a single, meaningful element:

- **Words**: The most common form of tokens.
- **Punctuation Marks**: Symbols like periods, commas.
- **Emojis**: Visual symbols representing emotions or concepts.
- **Numbers**: Digits and numerical expressions.
- **Sub-words**: Smaller units within words, like prefixes (re, pre) or suffixes (ing).
- **Phrases**: Multiword expressions (e.g., "ice cream").

## Tokenizer

**Idea:** Use whitespaces as the delimiter of words.

- Not suitable for languages with continuous orthographic systems (e.g., Chinese, Japanese).
- Example: Separating 51 and . will require additional rules.

# Bag of Words Representation

## Turning Words into Numbers

#### One-hot Vectors

- **Positive Features**:
    - No information is lost: you can reconstruct the original document from one-hot vectors.
- **Negative Features**:

    - One-hot vectors are sparse, resulting in large tables for short sentences.
    - A typical language vocabulary contains at least 20,000 common words, increasing to millions with variations and proper nouns.

Example calculation:

- Vocabulary: 1 million tokens.
- Library: 3,000 books, 3,500 sentences each, 15 words per sentence.

15 x 3,500 x 3,000 = 157,500,000 tokens

Storage requirement:

106 bits/token x 157,500,000 = 157.5 x 10^12 bits ≈ 17.9 TB

**Bag-of-Words (BoW)**

- **Definition**: A vector obtained by summing all the one-hot vectors.
- **Binary BoW**: Each word presence is marked as 1 or 0, regardless of frequency.

### Binary BoW Example

**Vocabulary Example**

- Example text corpus vocabulary:
    - 'Leonardo', 'Lisa', 'Mona', 'painting', 'tennis', 'The'

**Generating BoW vectors:**

- BoW overlap can measure text similarity.
- Use **dot product** to compare documents.

# Token Normalization

## Tokenizer Improvement

**Challenge:** Not only spaces separate words:

- Tabs (\t), newlines (\n), punctuation (commas, periods).

**Solutions:**

- Use **regular expressions** to improve tokenization.
- Example sentences:
    - "The company’s revenue for 2023 was $1,234,567.89."
    - "The CEO of the U.N. gave a speech."

## Case Folding

**Definition:** Consolidates multiple spellings of a word by normalizing capitalization:

- Tennis → tennis
- Leonardo → leonardo

**Advantages:**

- Improves text matching in search engines.

**Disadvantages:**

- Loses proper noun distinctions (e.g., US → us).

### Stop Words

**Definition:** Common words with high frequency but little information:

- Examples: Articles, prepositions, conjunctions.
- **Italian stop words:**
    - Examples: a, affinché, al, che, con, di, e, il, in, ma, non, o, per, si, su, un

**Disadvantages:**

- Even though stop words carry little information, they can provide relational information.
- Example:
    - Mark reported to the CEO vs. Mark reported CEO

## Stemming and Lemmatization

### Stemming

**Definition:** Identifies a common stem among word forms:

- Example: Housing and houses share the stem house.
- **Naïve Stemmer**:
    - Doesn't handle exceptions or complex modifications.

**Porter Stemmer:**

- Steps to remove suffixes like s, es, ed, and ing.

### Lemmatization

**Definition:** Determines the dictionary form (lemma) of a word.

- Considers word context and morphological analysis.

**Comparison:**

- **Stemming**: Faster but may produce invalid roots.
- **Lemmatization**: Slower but guarantees valid lemmas.

# Part of Speech Tagging

## Definition

- Labels tokens with lexical categories (noun, verb, adjective).

**Example:**

- Sentence: "The light is bright."
    - light can be a noun or verb.
    - Use context to determine meaning.

## Algorithms

- Use statistical models and dictionaries.
- Example:
    - P(ti) = P(wi|ti) P(ti|ti-1)

# Introducing spaCy

## Features

- Supports 25 languages.
- Models for tokenization, lemmatization, and PoS tagging.

**Installation:**

- pip install spacy
- Example models:
    - en_core_web_sm, it_core_news_lg

# Additional spaCy Features

## Dependency Parsing

- **Definition:** Determines syntactic dependencies between tokens (e.g., subject, object).
- Displacy: A built-in visualization tool in spaCy.

## Named Entity Recognition (NER)

- **Definition:** Identifies real-world objects assigned a name (e.g., people, organizations, dates).

- **Entity Labels:**
    - Examples:
        - PERSON: Names of people (e.g., "John").
        - GPE: Countries, cities, states (e.g., "France").
        - ORG: Organizations (e.g., "Google").
        - DATE: References to dates (e.g., "January 1st").

### Creating Bag-of-Words with spaCy

- **Process:**
    - Lemmatization reduces terms to their lemma.
    - Smaller vocabulary while retaining discriminative information.

# Math with Words

## Term Frequency

### Bag of Words

A vector space model of text:

- **One-hot encode** each word in a text and combine one-hot vectors.
- **Binary BoW**: One-hot vectors are combined with the OR operation.
- **Standard BoW**: One-hot vectors are summed.
- **Term Frequency (TF)**: Number of occurrences of each word in the text.

**Assumption:**

- The more times a word occurs, the more meaning it contributes to that document.

### Calculating TF

#### Limits of TF

Example:

- In document A, the word "dog" appears 3 times.
- In document B, the word "dog" appears 100 times.

Question:

- Is the word "dog" more important for document A or B?

Additional information:

- Document A: A 30-word email to a veterinarian.
- Document B: The novel *War & Peace* (approx. 580,000 words).

**Normalized TF**:

- Normalized (weighted) TF is the word count normalized by the document length:
    - TF(dog, documentA) = 3/30 = 0.1
    - TF(dog, documentB) = 100/580000 = 0.00017

# Vector Space Model

## Mathematical Representation

- Documents are represented as vectors in a multidimensional space.
- Each dimension represents a word from the vocabulary.

**Term-Document Matrix**

- **Rows**: Represent documents.
- **Columns**: Represent terms from the vocabulary.
- **Elements**: TF, normalized TF, or other values.

**Example:**

- Consider a vocabulary of two words (w1 and w2).
- Each document is plotted as a point in the 2D space.

## Document Similarity

**Euclidean Distance**

- Sensitive to the magnitude of the vectors.
- Captures the **relative importance** of terms.
- Less commonly used in NLP.

**Cosine Similarity**

- Measures the **cosine of the angle** between two vectors.
- Focuses on the **direction** of the vectors, ignoring magnitude.
- Effective for normalized text representations.
- Widely used in NLP.

**Cosine Similarity Formula:**

$$\text{Cosine Similarity} = A \cdot B \,||A|| \times ||B||$$

# TF-IDF

## Inverse Document Frequency (IDF)

**Problem with TF:**

- Common words (e.g., "the," "is") appear frequently in many documents.
- Such words contribute little to differentiating between documents.

**Solution:**

- **IDF** measures the importance of each word across the corpus.
- Words that are **rare** across documents are assigned higher weights.

## TF-IDF Formula

- **TF-IDF** combines TF and IDF:
    - High TF-IDF: Term is frequent in a document but rare in the corpus.
    - Low TF-IDF: Term is infrequent in a document or common in the corpus.

**Example:**

- Document A: "Jupiter is the largest planet."
- Document B: "Mars is the fourth planet from the sun."

# Zipf's Law

## Observation

- Frequency of a word is **inversely proportional** to its rank in the frequency table.

**Formula:**

$f(r)=Kr\alpha$.

- f(r): Frequency of the word at rank r.
- K: Constant.
- r: Rank of the word.
- alpha: Determines the shape of the distribution (~1 for natural language).

## Implications

- A small set of highly frequent words dominate word usage.
- Majority of words are relatively rare with low frequencies.
- Using logarithms in IDF mitigates the influence of rare words.

# Building a Search Engine

### Process

1. Tokenize all documents and create a TF-IDF matrix.
2. Prompt the user for a query.
3. Treat the query as a document and calculate its TF-IDF vector.
4. Match query vocabulary with the TF-IDF matrix.
5. Compute cosine similarity between the query vector and document vectors.
6. Retrieve the document(s) with the highest similarity.

### Optimization

- Real search engines use an **inverted index**:
    - Maps each word to documents containing it.
    - Compares the query only with relevant documents.

# Text Classification

## Definition

The process of assigning one or more classes to a text document for various purposes:

- Topic labeling
- Intent detection
- Sentiment analysis

**Characteristics:**

- **Relies only on text content**: Metadata or other document attributes are disregarded.
- **Classes are predefined**: Unlike clustering, classification uses fixed categories.

**Function:**

Given:

- A set of documents $D=\{d1,\ldots,dn\}$
- A set of predefined classes $C=\{c1,\ldots,cm\}$

Text classification finds a function:

$Classifier: D \times C \rightarrow \{true, false\}$.

## Types of Classification

1. **Single-label**:
    - Assigns each document to only one class in CC.
2. **Binary**:
    - Special case of single-label with only two classes.
3. **Multi-label**:
    - Assigns each document to multiple classes in CC.
    - Can be reduced to a series of binary decisions.

# ML-Based Classification

- A machine learning model is trained on a set of annotated text documents.
- Each document in the training set is associated with one or more class labels.
- After training, the model predicts categories for new documents.
- The classifier may provide a confidence measure.
- Requires a vector representation of documents (e.g., TF-IDF).

# Topic Labelling: Example

## Classifying Reuters News

### Dataset: Reuters 21578

- **Classes**: 90 distinct topics.
- **Documents**:
    - 7,769 for training.
    - 3,019 for testing.
    - Word count ranges from 93 to 1,263 per document.
- **Skewness**:
    - Some classes have over 1,000 documents.
    - Others have fewer than 5 documents.

**Process:**

1. Extract training and test samples with labels.
2. Create TF-IDF matrices for both sets.
3. Transform labels into binary matrices (one-hot encoding).
4. Train a classifier (e.g., MLP).
5. Test the classifier.

# Sentiment Analysis: Exercise

## Definition

The process of identifying and categorizing opinions expressed in a text.

**Applications:**

1. **Business**:
    - Analyzing customer feedback and product reviews.
    - Understanding customer satisfaction and brand perception.
2. **Finance**:
    - Predicting market trends based on investor sentiment.
3. **Politics**:
    - Analyzing public opinion during elections or policy changes.

**Example Dataset: IMDB**

- **Size**: 50,000 highly polarized reviews.
- **Distribution**:
    - 50% negative reviews.
    - 50% positive reviews.

**Exercise:**

1. Build a classifier for movie reviews.
2. Classify reviews as positive or negative.
3. **Training and Testing**:
    - Use 80% for training and 20% for testing.
4. Show and plot metrics and the confusion matrix.

**Suggestions:**

- One-hot encode labels: (1,0)=negative,(0,1)=positive.
- Reduce the TF-IDF matrix by considering only words appearing in at least 5 documents.
- Use confusion_matrix from Scikit-learn and visualize with Seaborn's heatmap.

# Applications of Text Classification

1. Topic Labelling
2. Sentiment Analysis
3. Spam Filtering
4. Intent Detection
5. Language Detection
6. Content Moderation
7. Product Categorization
8. Author Attribution
9. Content Recommendation
10. Ad Click Prediction
11. Job Matching
12. Legal Case Classification

# Limitations of TF-IDF

## Challenges:

- TF-IDF counts terms based on exact spelling.
- Texts with the same meaning can have completely different TF-IDF vector representations if they use different words.

**Examples:**

1. "The movie was amazing and exciting."
2. "The film was incredible and thrilling."

## Term Normalization:

- **Stemming and Lemmatization** help normalize terms:
    - Collect words with similar spellings under a single token.

**Disadvantages:**

- Fail to group most synonyms.
- May group together words with similar spellings but different meanings:
    - "She is leading the project" vs. "The plumber leads the pipe."
    - "The bat flew out of the cave" vs. "He hit the ball with a bat."

# Word Embeddings

## Definition:

A technique for representing words with vectors (Word Vectors):

- **Dense** vectors.
- Dimensions much smaller than vocabulary size.
- Continuous vector space.

**Key Feature:**

- Vectors are generated so that words with similar meanings are close to each other in the space.
- The position represents the semantics of the word.

**Example:**

- **Apple** = (0.25, 0.16).
- **Banana** = (0.33, 0.10).
- **King** = (0.29, 0.68).
- **Queen** = (0.51, 0.71).

# Properties of Word Embeddings

## Semantic Reasoning:

Word embeddings enable reasoning based on vector arithmetic:

**Examples:**

1. Subtracting "royal" from "king" leads to "man."
2. Subtracting "royal" from "queen" leads to "woman."
3. "King - man + woman" leads to "queen."

## Semantic Queries:

Allows searching for words by interpreting semantic meaning:

- **Query**: "Famous European woman physicist"
    - wv['famous'] + wv['European'] + wv['woman'] + wv['physicist']
    - ≈ wv['Marie_Curie']

# Learning Word Embeddings

## Word2Vec:

Introduced by Google in 2013:

- Based on neural networks.
- Uses unsupervised learning on large, unlabeled text corpora.

**Continuous Bag-of-Words (CBOW):**

- Predicts the central word based on surrounding words.
- Input: Multi-hot vector of surrounding tokens.
- Output: Probability distribution over the vocabulary.

**Skip-Gram:**

- Predicts surrounding words based on the central word.
- Effective for small corpora and rare terms.

# Improvements to Word2Vec

## Frequent Bigrams:

- Combines common word pairs (e.g., "Elvis_Presley").
- Focuses on meaningful predictions.

### Subsampling Frequent Tokens:

- Reduces the influence of common words.
- Words are sampled in inverse proportion to their frequency.

### Negative Sampling:

- Selects a small number of negative words instead of updating all weights.
- Maintains the quality of embeddings while reducing computational cost.

# Word2Vec Alternatives

## GloVe (Global Vectors for Word Representation):

- Introduced in 2014 by Stanford University.
- Uses classical optimization methods (e.g., SVD) instead of neural networks.

**Advantages:**

- Comparable precision to Word2Vec.
- Faster training times.
- Effective on small corpora.

## FastText:

- Developed by Facebook in 2017.
- Represents words as sub-word n-grams (e.g., "whisper" generates wh, whi, isp, per).

**Advantages:**

- Effective for rare or compound words.
- Can handle misspelled words.

# Working with Word Embeddings

## Loading Pre-trained Word Embeddings

- **Gensim** is a Python library for NLP supporting various pre-trained models.
- Examples:
    - word2vec-google-news-300
    - glove-wiki-gigaword-50

**Using Gensim:**

Install Gensim:

1. pip install gensim

2. Load a pre-trained model:
   from gensim.models import KeyedVectors
   model = KeyedVectors.load_word2vec_format('path_to_model', binary=True)

## Computing Similarity Between Words

Word embeddings allow computation of similarity scores between words:

**Example:**
Compute similarity between "king" and "queen":
```
similarity = model.similarity('king', 'queen')
print(similarity)
```

- 
- Result: A value between -1 and 1.

## Visualizing Embeddings

Word vectors can be visualized in 2D space using dimensionality reduction techniques like PCA or t-SNE:

**Example:**

- Google News vectors projected onto a 2D map:
  - Cities like "San Diego" and "San Jose" cluster closely due to similar cultural contexts.
  - Vacations spots such as "Honolulu" and "Reno" are also grouped.
- **Visualization Tool**: PCA ensures projections maintain maximum separation between vectors.

## Document Similarity with Word Embeddings

**Approach:**

- Compute the average of word vectors in a document to represent it as a single vector.
- Use this vector to compare documents based on cosine similarity.

**spaCy Example:**
Install spaCy:
```
pip install spacy
```

1. Load a language model:
   import spacy
   nlp = spacy.load('en_core_web_md')
2. Compute similarity:
   doc1 = nlp("The cat sat on the mat.")
   doc2 = nlp("The dog lay on the rug.")
   print(doc1.similarity(doc2))

3. Result: Similarity score indicating the documents' semantic closeness.

### Handling Out-of-Vocabulary Words

- Traditional embeddings discard unknown words.
- **FastText** handles unknown words by generating embeddings based on sub-words.
    - Each word is treated as a collection of character n-grams.

**Example:**

- For "unhappiness":
    - Sub-words: un, unh, hap, hap, ess.
    - Embedding is the aggregate of sub-word vectors.

# Recurrent Neural Networks

## Neural Networks and NLP

Neural networks are widely used in text processing, but traditional feedforward networks have significant limitations:

- **No memory**: They process each input independently, without considering past context.
- **Single data requirement**: Sequences must be flattened into a single data structure, such as:
    - Bag of Words (BoW).
    - TF-IDF vectors.
    - Averaged word embeddings.

This approach ignores the sequential nature of language.

## Neural Networks with Memory

Humans process text sequentially, remembering what they've read. RNNs mimic this:

- **Sequential processing**: Iterates through sequences, updating its internal state.
- **Memory**: Maintains information about past inputs, enabling context-sensitive processing.

**Example:**

- Words in a sentence are processed as embeddings, continuously updating the network's internal model.

## Structure of an RNN

An RNN comprises:

1. **Input layer**: Processes tokens at each time step.
2. **Hidden layer**: Maintains state and feedback loops.
3. **Output layer**: Produces results at each time step.

**Feedback Mechanism:**

- Output from the hidden layer at time tt is fed as input for time t+1t+1.
- This feedback enables the network to capture dependencies across sequences.

**Unrolling the RNN:**

- Unrolled diagrams show how weights are shared across time steps.
- Each layer represents the same network processing inputs sequentially.

### Applications of RNNs

**Examples:**

1. Detecting patterns in sequences:
    - "The stolen car sped into the arena" vs. "The clown car sped into the arena."
    - Context changes meaning.
2. Capturing relationships:
    - Adjectives like "stolen" or "clown" affect subsequent words like "arena."

## RNN Variants

### Bidirectional RNNs

**Features:**

- Processes sequences in both forward and backward directions.
- Captures context from both past and future tokens.

**Example:**

- "They wanted to pet the dog whose fur was brown."
- Bidirectional RNN captures relationships more effectively than unidirectional.

**Implementation:**

```
from keras.layers import Bidirectional, SimpleRNN

model.add(Bidirectional(SimpleRNN(64)))
```

### Long Short-Term Memory (LSTM)

**Addressing Vanishing Gradients:**

- LSTMs introduce **memory cells** to store long-term dependencies.
- Three key gates:
    - Input gate: Controls what information is added.
    - Forget gate: Decides what to discard.
    - Output gate: Determines final output.

### Gated Recurrent Unit (GRU)

#### Simplified LSTM:

- Combines input and forget gates into a single gate.
- Reduces computational complexity while retaining effectiveness.

### Stacked RNNs

#### Enhancing Model Capacity:

- Layers are stacked to capture complex relationships.

- Example:
  model.add(LSTM(128, return_sequences=True))
  model.add(LSTM(64))

# Building a Spam Detector

### Dataset:

- Source: SMS Spam Collection.
- Contains labeled SMS messages (spam or ham).

### Process:

1. Pre-process data:
    - Tokenize text.
    - Generate word embeddings.
2. Split dataset:
    - Training and testing sets.
3. Train RNN:
    - Use embeddings as input.
    - Optimize using backpropagation through time.
4. Evaluate:
    - Plot training performance.
    - Generate confusion matrix to assess accuracy.

# Intro to Text Generation

### Generative Models

Generative models are designed to produce new, coherent, and syntactically correct text.

**Applications:**

1. **Machine Translation**: Translate text between languages.
2. **Automatic Summarization**: Generate concise summaries of longer texts.
3. **Dialogue Systems**: Enable chatbots to respond dynamically.
4. **Creative Writing**: Assist in generating poetry, stories, etc.

### Language Models (LMs)

**Purpose:**

- Predict the next token in a sequence based on prior tokens.
- Capture statistical structure of language (latent space).

**Training:**

1. Input a sequence of tokens.
2. Compare the network's output to the expected next token.
3. Propagate error backward to update weights.

# Building a Poetry Generator

### Dataset:

- Source: Leopardi Corpus.

### Steps:

1. **Pre-process text**:
    - Tokenize into characters.
    - Generate training samples.
2. **Train RNN**:
    - Use character-level embeddings.
    - Predict next character in sequence.
3. **Generate Poetry**:
    - Define helper functions.
    - Sample new sequences from the trained model.

**Example:**

- Start with "O patria mia" as input and generate new text resembling Leopardi's style.

# Task-Oriented Dialogue Systems

### Types of Conversational AI

1. **Chit-Chat Systems**:
    - No specific goal.
    - Focus on generating natural and engaging responses.
    - Performance improves with longer conversational exchanges.
2. **Task-Oriented Dialogue (TOD) Systems**:
    - Help users achieve specific goals.
    - Prioritize understanding user intents and tracking conversation states.
    - Aim to minimize the number of conversational turns.

### Examples of Task-Oriented Dialogue

**Questions:**

- "Which room is the dialogue tutorial in?"
- "When is the IJCNLP 2017 conference?"

**Tasks:**

- "Book me a flight from Seattle to Taipei."
- "Schedule a meeting with Bill at 10:00 tomorrow."

**Recommendations:**

- "Can you suggest me a restaurant?"
- "Can you suggest me something to see near me?"

**TOD System Architecture:**

- Rasa is a popular framework for building TOD systems.
- **Image Explanation**: A flow diagram illustrates how Rasa processes user inputs through NLU (Natural Language Understanding), maps intents and entities, and triggers actions or responses accordingly.

# Natural Language Understanding in TOD

## Main Components:

1. **Intent Classification**:
    - Treated as a multi-label sentence classification task.
2. **Entity Recognition**:
    - Extract relevant terms or objects using:
        - Rule-based approaches.
        - Machine Learning-based approaches (e.g., Named Entity Recognition).

**Example:**

- User query: "What's the weather like tomorrow?"
    - **Intent**: CheckWeather.
    - **Entities**: {"date": "tomorrow"}.

### Conversation Design

**Key Steps:**

1. Identify your target audience.
2. Define the assistant's purpose.
3. Document typical user conversations.

**Best Practices:**

- Start with hypothetical scenarios.
- Train the assistant using real conversations as early as possible.

# Introduction to Rasa

### Overview:

- Open-source conversational framework launched in 2016.
- Used globally to create bots in various languages.

### Core Units:

1. **Intents**: Define user goals (e.g., "Book a flight").
2. **Entities**: Extract necessary terms (e.g., "Seattle," "Taipei").
3. **Actions**: Define bot responses or backend operations.
4. **Responses**: Predefined utterances (e.g., "Your flight is booked.").
5. **Slots**: Store extracted information for use in context.
6. **Forms**: Collect multiple pieces of information.
7. **Stories**: Predefined conversation paths combining intents and actions.

**Example:**

- **Intent**: book_flight
- **Entities**: {"origin": "Seattle", "destination": "Taipei"}
- **Action**: Fetch available flights and provide recommendations.

# Installing and Setting Up Rasa

### Steps:

1. **Virtual Environment**:
   python -m venv rasa.env

source rasa.env/bin/activate

2. **Install Rasa**:
   pip install rasa
3. **Create a New Project**:
   rasa init
4. **Directory Structure**:

    - **domain.yml**: Defines intents, entities, slots, responses, etc.
    - **nlu.yml**: Contains training data for intents and entities.
    - **stories.yml**: Contains conversation paths.
    - **actions.py**: Custom Python functions for complex actions.
    - **config.yml**: Configures the pipeline and policies.
    - **rules.yml**: Specifies rules for fixed conversation flows.
    - **endpoints.yml**: Configures endpoints for custom actions.

# Building a Chatbot with Rasa

## Workflow:

1. Define intents and entities in domain.yml.
2. Add training data in nlu.yml.
3. Create conversational paths in stories.yml.
4. Train the model:
   rasa train
5. Test the assistant:
   rasa shell
6. Deploy the assistant:
   rasa run

**Example:**

- **Domain**:
  ```
  intents:
    - greet
    - book_flight
  entities:
    - origin
    - destination
  responses:
    utter_greet:
      - text: "Hello! How can I assist you?"
    utter_book_flight:
      - text: "Booking a flight from {origin} to {destination}."
  ```
- **Stories**:
  ```
  stories:
  - story: book a flight
    steps:
      - intent: greet
  ```

```
- action: utter_greet
- intent: book_flight
- action: utter_book_flight
```

**Image Explanation**: The diagram for Rasa's directory structure highlights how files such as domain.yml and stories.yml interconnect to define intents, entities, and responses.

# Custom Actions

- Enable bots to perform tasks beyond predefined responses:
  - Sending emails.
  - Fetching data from databases.
  - Checking APIs.

**Files for Custom Actions:**

1. **domain.yml**:
   - Lists all intents, entities, slots, and responses.
2. **nlu.yml**:
   - Provides training data for intents and entities.
3. **stories.yml**:
   - Defines conversation paths involving custom actions.
4. **rules.yml**:
   - Specifies fixed conversational flows.
5. **endpoints.yml**:
   - Configures the action server endpoint.
6. **actions.py**:
   - Implements custom logic for actions.

**Example:**

- Action server setup in endpoints.yml:
  action_endpoint:
  url: "http://localhost:5055/webhook"
- **Action Implementation** (actions.py):

```
from rasa_sdk import Action
from rasa_sdk.events import SlotSet

class ActionBookFlight(Action):
    def name(self):
```

```
        return "action_book_flight"

    def run(self, dispatcher, tracker, domain):
        origin = tracker.get_slot("origin")
        destination = tracker.get_slot("destination")
        dispatcher.utter_message(text=f"Flight booked from {origin} to {destination}!")
        return []
```

- **Image Explanation**: A flowchart explains how the action server communicates with Rasa to handle custom requests (e.g., database queries or API calls).

# Exercise: Building a Pizzeria Chatbot

Develop a chatbot to assist with pizzeria operations:

- **Users can**:
    - Request the pizzeria's menu.
    - Order a pizza that is available in the menu (just one, no beverages).
- **Upon order confirmation**, the bot will:
    - Log the date.
    - Store the user ID.
    - Record the kind of ordered pizza (using a custom action).
- The bot has a web-based GUI.

# Hints

## Start with a Dummy Bot

1. Create a new directory and initialize Rasa:
   mkdir pizzaBot
   cd pizzaBot
2. rasa init --no-prompt
3. Configure and run the REST and Actions servers:
    - **Run the REST server:**
      rasa run --cors "*"
    - **Run the Actions server:**
      rasa run actions
4. Use a web frontend like:
    - Chatbot Widget (Widget2.0).

# Limitations of RNNs

### Key Issues:

1. **Lack of Long-Term Memory**:
    - RNNs struggle to retain information over extended sequences.
2. **Slow Training**:
    - Processing is inherently sequential.
    - Cannot exploit modern GPU parallelism effectively.
3. **Vanishing and Exploding Gradients**:
    - Gradients can shrink exponentially (vanishing) or grow uncontrollably (explode) during Backpropagation Through Time (BPTT).

**Image Explanation**:

- Diagram shows how RNN sequential processing leads to inefficiencies, requiring each step to complete before the next begins.

# Transformer

### Introduction:

- Introduced by Google Brain in 2017.
- Processes sequential data in parallel.
- Resolves gradient problems by decoupling layer count from sequence length.
- Initially designed for machine translation but adaptable to other sequence tasks.

**Image Explanation**:

- Parallel processing visualized, contrasting with RNN sequential dependency.

# Transformer's Input

### Tokenization:

- Splits text into discrete tokens, each assigned a unique ID.

### Input Embedding:

- Represents tokens as dense vectors in a continuous vector space.
- Embeddings capture semantic relationships, grouping similar words.

### Positional Encoding:

- Adds periodic perturbations to embeddings to encode positional information.
- Differentiates sequences with the same tokens but different orders.

**Image Explanation**:

- Example shows two sentences with identical tokens in different orders, highlighting the importance of positional encoding.

# Encoder

## Function:

- Converts input sequences into intermediate representations of the same length.
- Outputs consider the entire input sequence context.

## Structure:

- Comprised of stacked encoder blocks.
- Each block includes:
    - **Self-Attention**: Assigns relevance between tokens.
    - **Feedforward Layer**: Processes each token independently.
    - **Skip Connections**: Maintain gradient flow.
    - **Normalization**: Stabilizes learning.

**Image Explanation**:

- Diagram shows the flow of input through encoder layers, emphasizing self-attention and feedforward mechanisms.

# Self-Attention

## Definition:

- Computes attention weights between tokens to capture contextual relationships.

## Steps:

1. **Input Representation**:
    - Convert tokens into numerical vectors using embeddings.
2. **Projection**:
    - Compute Query (Q), Key (K), and Value (V) matrices:
      Q' = Q * WQ
      K' = K * WK
      V' = V * WV
        - WQ, WK, WV are trainable weight matrices.
3. **Attention Matrix Calculation**:
    - Compute scaled dot-product between Q and K:
      A = softmax(Q' * K'.T / sqrt(dk))
        - Apply softmax to normalize scores.
4. **Weighted Sum**:

    - Multiply the attention matrix by V to get the output:
      Output = A * V'

**Example Sentence**:

- "The animal didn’t cross the street because it was too wide."
    - Self-attention identifies "it" as referring to "street" by assigning higher attention scores to "street."

**Image Explanation**:

- Visualizes how self-attention weights are computed and applied to resolve pronoun ambiguity.

# Attention Function

## Components:

1. **Query (Q)**: Represents the current token being processed.
2. **Key (K)**: Represents other tokens in the sequence.
3. **Value (V)**: Contains contextual information about tokens.

## Attention Mechanism:

- Assigns weights to each value based on the similarity between the query and keys.
- Normalized using softmax to ensure weights sum to 1.

## Formula:

fT(Q)=Σα(Q,Ki)∗VifT(Q) = Σ α(Q, K_i) * V_i

Where:

- α(Q, K_i) is the attention score for Key i.
- Scores are learned during training.

**Image Explanation**:

- Shows the computation of attention weights and how they influence the final output.

# Multi-Head Attention

## Overview:

- Multiple self-attention heads allow the model to encode different aspects of context simultaneously.
- Parallel scaled dot-product attention computations use different weight matrices.

- Results are concatenated row-by-row into a larger matrix and then combined using an additional weight matrix.

### Key Benefits:

- Models can attend to different representation subspaces at various positions.
- Prevents the averaging issue seen in single-head attention.

**Image Explanation:**

- Illustration showing single-head vs. multi-head attention, emphasizing the diversity of focus areas.

# Add & Norm

### Skip Connections:

- Skip (residual) connections add the input of a layer to its output to:
    - Stabilize gradients.
    - Improve training efficiency.

### Normalization:

- Ensures stable training by scaling activations within layers.

# Feed Forward

### Structure:

- A pointwise fully connected feedforward network applies non-linearity.
- Operates independently on each sequence element.

### Benefits:

- Adds capacity to capture complex features.

# Transformer’s Encoder

### Function:

- Encodes the input sequence into intermediate representations.
- Uses positional encoding to handle order information.

### Features:

1. Residual connections facilitate gradient flow.

2. Normalization stabilizes training.
3. Each block includes self-attention and feedforward layers.

**Stacking Blocks:**

- Stacked encoder blocks propagate information across multiple transformations.
- Output dimensionality remains constant.

**Image Explanation:**

- Diagram of stacked encoder blocks showing sequential transformations.

# Decoder

## Role:

- Generates the output sequence y1,...,ym from intermediate encoder representations z1,...,zt.
- Works sequentially, leveraging both encoder outputs and past decoder outputs.

## Structure:

- Composed of decoder blocks with:
    1. Modified self-attention to restrict future information.
    2. Encoder-decoder attention linking encoder outputs.
    3. Feedforward layers and normalization.

## Final Layer:

- Includes a linear layer and softmax activation to predict output probabilities.

**Image Explanation:**

- Visualization of decoder blocks with attention links to encoder outputs.

# Masked Multi-Head Attention

## Purpose:

- Ensures that predictions depend only on known inputs (past tokens).
- Masks future positions during training to maintain causality.

**Image Explanation:**

- Example matrix shows masked attention scores with zero weights for future tokens.

# Encoder-Decoder Attention

### Role:

- Links encoder outputs (keys and values) to decoder queries.
- Allows decoder to incorporate context from the entire input sequence.

### Mechanism:

- Combines encoder keys and values with decoder queries via scaled dot-product attention.

#### Image Explanation:

- Diagram showing how encoder outputs influence decoder attention calculations.

## Output

### Linear Layer:

- Transforms decoder outputs into logits representing unnormalized probabilities.

### Softmax Layer:

- Converts logits into probabilities for each token in the vocabulary.
- Predicts the most likely next token in the sequence.

#### Image Explanation:

- Flowchart depicting linear transformation and softmax activation to generate final token probabilities.

## Transformer's Pipeline

### Steps:

1. Tokenization: Split input into tokens.
2. Embedding: Map tokens to dense vectors.
3. Positional Encoding: Add positional information.
4. Encoder: Process inputs into intermediate representations.
5. Decoder: Generate output sequences.
6. Final Layers: Apply linear transformation and softmax.

### Visualization:

- Detailed flowchart of the Transformer pipeline from input to output.

**Adding Guardrails to LLMs :**
Large language models are powerful tools for natural language understanding and generation. Their applications range from content creation to conversational AI, but their versatility brings risks, such as generating harmful content, leaking sensitive data, or producing biased outputs. Adding guardrails ensures these models are safe, reliable, and aligned with ethical principles

**Guardrails:**
Guardrails are mechanisms or policies that regulate the behavior of LLMs. They help to ensure that responses are safe, accurate, and context-appropriate. They can:

- Prevent harmful, biased, or inaccurate outputs.
- Align responses with ethical and operational guidelines.
- Build trust and reliability for real-world applications.

**Types of guardrails are:**

- Safety Guardrails: Prevent generation of harmful or offensive content.
- Domain-Specific Guardrails: Restrict responses to specific knowledge areas.
- Ethical Guardrails: Avoid bias, misinformation, and ensure fairness.
- Operational Guardrails: Limit outputs to align with business or user objectives.

**Techniques for adding guardrails are :**

- **Rule based filters**
  Use predefined rules to block or modify certain outputs. Is simple and efficient for basic content filtering.
- **Fine tuning with custom data**
  Train the model on domain-specific, curated datasets and Adjust weights to produce outputs aligned with guidelines.
- **Prompt Engineering**
  Craft and or refine prompts to guide the LLM behavior within desired boundaries.
- **External validation layers**
  Additional systems or APIs that post-process the model's outputs. Allows modular and scalable implementation of guardrails.
- **Real-time monitoring and feedback**
  Monitor outputs continuously for unsafe or incorrect content. Flag or block problematic outputs in real-time. His tools are Human-in-the-loop systems and Automated anomaly detection.

The Best practices are combine multiple techniques for robust safeguards .

**Frameworks for implementing guardrails**

The existing frameworks for implementing guardrails offer easy integration with LLM APIs and Predefined and customizable rulesets.
Popular tools are:

- **Guardrails AI** that is  a library for implementing safeguards:
  Validation: Ensures outputs are within specified guidelines.
  Formatting: Controls the output structure.
  Filters: Removes or blocks unsafe content.
- **LangChain** for chaining prompts and filtering outputs. Chains prompts with checks and filters. Verifies outputs against predefined criteria.
- **OpenAI Moderation** that is a prebuilt API to detect unsafe content.

**The Hugging Face Hub** hosts Models, datasets, spaces for demos and code. The key libraries include:

- datasets: Download datasets from the hub
- transformers: Work with pipelines, tokenizers, models , etc.
- evaluate: Compute evaluation metrics.

These libraries can use PyTorch and TensorFlow.

**Hugging Face - Datasets**

The Hub hosts around 3000 datasets that are open-sourced and free to use in multiple domains. On top of it, the open-source datasets library allows the easy use of these datasets, including huge ones, using very convenient features such as streaming. • Similar to model repositories, you have a dataset card that documents the dataset. If you scroll down a bit, you will find things such as the summary, the structure, and more.

**SETUP**

**Setup - Google Colab**

Using a Colab notebook is the simplest possible setup; boot up a notebook in your browser and get straight to coding!

- !pip install transformers
- import transformers

This installs a very light version of Transformers. In particular, no specific machine learning frameworks (like PyTorch or TensorFlow) are installed. You can also install the development version, which comes with all the required dependencies for pretty much any imaginable use case:

- !pip install transformers[sentencepiece]

**Setup – Virtual environment**

Download and install Anaconda at this link : https://www.anaconda.com/download

After the download use:

- conda create --name nlpllm
- conda activate nlpllm
- conda install transformers[sentencepiece]

**Setup – Create a Hugging Face account**

Most of the functionalities relies on you having a Hugging Face account. It is recommend creating one

**Pipeline**

The Hugging Face Transformers library provides the functionality to create and use the models. The Model Hub contains thousands of pretrained models that anyone can download and use. The most basic object in the Hugging Face Transformers library is the pipeline() function. It connects a model with its necessary preprocessing and postprocessing steps, allowing us to directly input any text and get an intelligible answer.

**Model selection**

The **Hugging Face Hub** provides access to a vast library of pre-trained models hosted at Hugging Face Models. These models are suitable for various natural language processing (NLP) tasks and are easily accessible. The platform allows users to select models based on their specific needs, whether for classification, translation, summarization, or other tasks. Model selection involves

choosing the most appropriate pre-trained model to fit the desired application, leveraging the repository's detailed model documentation.

**Common Models**

Hugging Face's ecosystem supports some of the most popular pre-trained models used in NLP and machine learning. These models are optimized for diverse tasks, such as sentiment analysis, text generation, and machine translation. By utilizing these widely recognized models, developers can benefit from state-of-the-art performance without needing extensive computational resources for training.

**Gradio**

**Gradio** is a tool integrated with Hugging Face for creating web-based demos and applications. Key features include:

- **Web Demos:** Gradio enables the quick creation of interactive web applications to showcase the capabilities of models directly on the web.
- **Free Hosting:** Developers can host their demos for free on Hugging Face Spaces at hf.space.
- **Ease of Use:** Installation is straightforward with a command like conda install gradio. Developers can create and customize their own demonstrations to share models and results interactively

**Encoder-only transformer**

You need the whole transformer architecture if the task you have to solve requires the transformation of a sequence into a sequence of different length .Example: translation between different languages (the original application of the Transformer model).For a task where you want to transform a sequence into another sequence of the same length, you can use just the Encoder part of the transformer.In this case, the vectors $z1,\ldots, zt$ will be your output vectors, and you can compute the loss function directly on them. For a task where you want to transform a sequence into a single value (e.g., a sequence classification task), you can use just the Encoder part of the transformer. You add a special START value as the first element $x1$ of the input .You take the corresponding output value $z1$ as the result of your transformation, and compute your loss function only on it.

**BERT**

BERT is a language model introduced by Google Research in 2018 for language understanding. It is the acronym of Bidirectional Encoder Representation from Transformers. It uses only the Encoder part from the Transformer model. BERT-base has 12 stacked encoder blocks (110M parameters), while BERT-large 24 (340M parameters). BERT is pre-trained to use "bidirectional" context (i.e. successive words, as well as previous ones, for understanding a word in a sentence).

It is designed to be used as a pre-trained base for Natural Language Processing tasks (after fine tuning).

**BERT input encoding**

BERT uses a WordPiece tokenizer as its tokenization method. Subword-Based: WordPiece tokenizes text at the subword level rather than using full words or individual characters. This helps BERT handle both common words and rarer or misspelled words by breaking them into manageable parts. Vocabulary Building: The WordPiece tokenizer builds a vocabulary of common words and subword units (e.g., "playing" could be tokenized as "play" and "##ing"). Unknown Words: Rare or out-of-vocabulary words can be broken down into familiar subwords. For example, "unhappiness" could be split into "un," "happy," and "##ness.
Splitting: Sentences are split into tokens based on whitespace, punctuation, and common prefixes (like "##"). BERT requires certain special tokens:
[CLS]: A classification token added at the beginning of each input sequence.
[SEP]: A separator token used to mark the end of a sentence (or between sentences in the case of sentence-pair tasks).For example, the sentence "I'm feeling fantastic!" might be tokenized by WordPiece as: [CLS] I ' m feeling fan ##tas ##tic ! [SEP] .
Converting Tokens to IDs: After tokenization, each token is mapped to an ID from BERT's vocabulary, which serves as input to the model. The advantages ofWordPiece embedding are the following:
• Efficiency: Reduces the vocabulary size without losing the ability to represent diverse language constructs.
• Handling Unseen Words: Subword tokenization allows BERT to manage rare or newly created words by breaking them down into recognizable parts.
• Improved Language Understanding: Helps BERT capture complex linguistic patterns by learning useful subword components during pretraining.

**BERT [CLS] token**

In BERT, the [CLS] token plays a crucial role as a special token added at the beginning of each input sequence. The [CLS] token (short for "classification token") is always placed at the very start of the tokenized sequence and is primarily used as a summary representation of the entire input sequence. After the input is processed by BERT, the final hidden state of the [CLS] token acts as a condensed, context-aware embedding for the whole sentence or sequence of sentences. This embedding can then be fed into additional layers (like a classifier) for specific tasks. The [CLS] token is used differently for single sentence and sentence pair classification:
**Single-Sentence Classification:**

- The final hidden state of the [CLS] token is passed to a classifier layer to make predictions.
- For instance, in sentiment analysis, the classifier might predict "positive" or "negative" sentiment based on the [CLS] embedding.

**Sentence-Pair Tasks:**

- For tasks involving two sentences, BERT tokenizes them as [CLS] SentenceA [SEP] Sentence B [SEP].
- The [CLS] token's final hidden state captures the relationship between the two sentences, making it suitable for tasks like entailment detection or similarity scoring.

**BERT pre-training**

BERT is pre-trained using two self-supervised learning strategies:

- Masked Language Modeling (MLM) :
  1. Objective: Predict masked (hidden) tokens in a sentence.
  2. Process: Randomly mask 15% of tokens and train BERT to predict them based on context.
  3. Benefit: Enables BERT to learn bidirectional context.
- Next Sentence Prediction (NSP) :
  1. Objective: Determine if one sentence logically follows another.
  2. Process: Trains BERT to understand sentence-level relationships.
  3. Benefit: Improves performance in tasks like question answering and natural language inference.

The training set is composed by a corpus of publicly available books plus the English Wikipedia, for a total of more than 3 billions of words.

**BERT fine tuning**

The output of the encoder is given to an additional layer to solve a specific problem. The cross-entropy loss between the prediction and the label for the classification task is minimized via gradient-based algorithms, where the additional layer is trained from scratch, while pretrained parameters of BERT may be updated or not. BERT is pretrained on general language data, then finetuned on specific datasets for each task. In fine-tuning, the [CLS] token's final embedding is specifically trained for the downstream task, refining its ability to represent the input sequence in the way needed for that task.
ExampleTasks:

• Text Classification: Sentiment analysis, spam detection.
• Named Entity Recognition (NER): Identifying names, dates, organizations, etc., within text.
• Question Answering: Extractive QA where BERT locates answers within a passage.

Minimal Task-Specific Adjustments: Only a few additional layers are added per task.

**BERT strengths and limitations**

**Strengths:**

- Bidirectional Contextual Understanding: provides richer and more accurate representations of language.
- Flexibility in Transfer Learning: BERT's pretraining allows for easy adaptation to diverse NLP tasks.

- High Performance on Benchmark Datasets: Consistently ranks at or near the top on datasets like SQuAD (QA) and GLUE (general language understanding).

**Limitations:**

- **L**arge Model Size: High computational and memory requirements make deployment challenging.
- Pretraining Costs: Requires extensive computational resources, especially for BERT-Large.
- Fine-Tuning Time and Data: Fine-tuning on new tasks requires labeled data and can still be time-intensive.

**Popular BERT variants – RoBERTa**

RoBERTa is the acronym for Robustly Optimized BERT Approach and was developed by Facebook AI in 2019. The main differences with respect to BERT are the following:
• Larger Training Corpus: Trained on more data (200 billions) compared to BERT.
• Removed Next Sentence Prediction (NSP): Found that removing NSP improves performance.
• Longer Training and Larger Batches: RoBERTa was trained for more iterations and used larger batches for robust language modeling.
• Dynamic Masking: Masking is applied dynamically (i.e., different masks per epoch), leading to better generalization.
RoBERTa consistently outperforms BERT on various NLP benchmarks, particularly in tasks requiring nuanced language understanding.

**Popular BERT variants - ALBERT**

ALBERT is the acronym of A Lite BERT, developed by Google Research in 2019.
The main differences with respect to BERT are:
• Parameter Reduction: Uses factorized embedding parameterization to reduce model size.
• Cross-Layer Parameter Sharing: Shares weights across layers to decrease the number of parameters.
• Sentence Order Prediction (SOP): Replaces NSP with SOP, which is better suited for capturing inter-sentence coherence.
ALBERT achieves comparable results to BERT-Large with fewer parameters, making it more memory-efficient. ALBERT is faster and lighter, ideal for applications where resources are limited.

**Popular BERT variants – DistilBERT**

DistilBERT is the acronym for Distilled BERT, developed by Hugging Face. The main differences with BERT are the following:
• Model Distillation: Uses knowledge distillation to reduce BERT's size by about 40% while retaining 97% of its language understanding capabilities.
• Fewer Layers: DistilBERT has 6 layers instead of 12 (for BERT-Base) but is optimized to perform similarly.
• Faster Inference: Provides faster inference and lower memory usage, making it ideal for real-time applications.

DistilBERT is widely used for lightweight applications that need a smaller and faster model without major accuracy trade-offs.

**Popular BERT variants -TinyBERT**

TinyBERT was developed by Huawei.The main differences with BERT are the following:
• Two-Step Knowledge Distillation: Distills BERT both during pretraining and fine-tuning, further enhancing efficiency.
• Smaller and Faster: TinyBERT is even smaller than DistilBERT, optimized for mobile and edge devices.
• Similar Accuracy: Maintains accuracy close to that of BERT on various NLP tasks, especially when fine-tuned with task-specific data.
TinyBERT is an ultra-compact version of BERT that is wellsuited for resource-constrained environments.

**Popular BERT variants – ELECTRA**

ELECTRA is the acronym for Efficiently Learning an Encoder that Classifies Token Replacements Accurately, developed by Google Research. Its differences with respect to BERT are:
• Replaced Token Detection: Instead of masked language modeling, ELECTRA uses a generator-discriminator setup where the model learns to identify replaced tokens in text.
• Efficient Pretraining: This approach allows ELECTRA to learn with fewer resources and converge faster than BERT.
• Higher Performance: Often outperforms BERT on language understanding benchmarks with significantly less compute power.
ELECTRA's training efficiency and robust performance make it appealing for applications where computational resources are limited.

**Popular BERT variants – SciBERT**

SciBERT is tailored for applications in scientific literature, making it ideal for academic and research-oriented NLP.
Domain-Specific Pretraining: Trained on a large corpus of scientific papers from domains like biomedical and computer science.
Vocabulary Tailored to Science: Uses a vocabulary that better represents scientific terms and jargon.
Improved Performance on Scientific NLP Tasks: Significantly outperforms BERT on tasks like scientific text classification, NER, and relation extraction.

**Popular BERT variants – BioBERT**

BioBERT is widely adopted in the biomedical research field, aiding in information extraction and discovery from medical literature.
Biomedical Corpus: Pretrained on a biomedical text corpus, including PubMed abstracts and PMC full-text articles.
Enhanced Performance on Biomedical Tasks: Excels at biomedical-specific tasks such as medical NER, relation extraction, and question answering in healthcare.

**Popular BERT variants – ClinicalBERT**

ClinicalBERT is ideal for hospitals and healthcare providers who need to analyze patient records, predict health outcomes, or assist in clinical decision-making.
Healthcare Focus: Tailored for processing clinical notes and healthcare-related NLP tasks.
Training on MIMIC-III Dataset: Pretrained on the MIMIC-III database of clinical records, making it useful for healthcare analytics.

**Popular BERT variants – mBERT**

mBERT, developed by Google, supports NLP tasks across languages, enabling global applications and language transfer learning.
Multilingual Support: Trained on 104 languages, mBERT can handle multilingual text without requiring separate models for each language.
Language-Agnostic Representation: Capable of zero-shot cross-lingual transfer, making it suitable for translation and cross-lingual understanding tasks**.**

**Other BERT variants**
CamemBERT: French language-focused BERT model.
FinBERT: Optimized for financial text analysis.
LegalBERT: Trained on legal documents for better performance in the legal domain.
Moreover, BERT inspired Transformer pre-training in computer vision, such as with vision Transformers, Swin Transformers, and Masked Auto Encoders (MAE).

**Decoder-only transformer**
Transformers that use only the decoder part of the Transformer architecture. It focuses only on decoding, making it efficient for autoregressive generation tasks. It lacks the separate encoder layers found in sequence-tosequence models. Primarily used for language generation tasks, such as text generation, summarization, and question answering. Popular Examples: GPT (Generative Pre-trained Transformer) series, including GPT-2, GPT-3 and GPT-4, and LLAMA.
In a decoder-only transformer, text generation is achieved through an autoregressive approach, where each token is generated sequentially by attending only to previously generated tokens within the same sequence, rather than using encoder-decoder attention. The input context (prompt) and the generated text are treated as a single, continuous sequence. This continuous sequence of tokens essentially allows the decoder-only model to handle both the “encoding” (understanding the input prompt) and “decoding” (generating text) in one step, without needing a separate encoder block. Once a token is generated, it’s appended to the input sequence, and the model proceeds to generate the next token.

**Self-Attention with Causal Masking**: The model uses selfattention within the decoder layers, but with a causal (unidirectional) mask that prevents each token from attending to future tokens. This ensures that each position only considers information from previous positions, simulating the generation process where each word is generated sequentially.

**Implicit Context Understanding**: In a decoder-only architecture, the model builds up context as it processes tokens in sequence. As it reads through a sequence, it "remembers" previous tokens and learns relationships between them within the attention layers. This sequential buildup of context replaces the need for separate encoder-decoder attention by allowing the model to accumulate an understanding of the input as it progresses through each token.

The differences between **Encoder-Only Transformers** (e.g., BERT) and **Decoder-Only Transformers** are:

**Architecture**:

- Encoder-only models use only encoder blocks with bidirectional attention.
- Decoder-only models use only decoder blocks with causal (unidirectional) attention.

**Training Objective**:

- Encoder-only models are trained using Masked Language Modeling (MLM).
- Decoder-only models are trained using Autoregressive Language Modeling.

**Context Processing**:

- Encoder-only models process the entire sequence in parallel.
- Decoder-only models process tokens sequentially, one at a time.

**Main Use Cases**:

- Encoder-only models are best suited for tasks like text classification, Named Entity Recognition (NER), and question answering.
- Decoder-only models excel in tasks such as text generation, story generation, and code generation.

**Attention Type**:

- Encoder-only models use bidirectional self-attention.
- Decoder-only models use unidirectional (masked) self-attention.

**Output**:

- Encoder-only models produce contextual embeddings for downstream tasks.
- Decoder-only models generate sequential tokens for text or other content creation.

**Decoder-only transformer applications**

Applications of Decoder-Only Transformers are:
• Text Generation: News articles, stories, and creative content.
• Conversational AI: Chatbots and virtual assistants for realtime dialogue.
• Programming Help: Code generation and debugging.
• Summarization: Generating concise summaries of long documents.

**GPT**

GPT is a type of decoder-only transformer developed by OpenAI. It generates human-like text, understanding and predicting language; being trained on vast amounts of text data, it can perform various natural language tasks without task-specific training.

GPT-1 (2018): Introduced decoder-only transformer architecture, with 117 million parameters (12 decoder blocks , 768-dimensional embeddings, and 12 attention heads per block).
GPT-2 (2019): Significantly larger with 1.5 billion parameters in its XL version (48 decoder blocks , 1600-dimensional embeddings, and 25 attention heads per block), capable of generating coherent long-form text.
GPT-3 (2020): 175 billion parameters (96 decoder blocks, 12,288- dimensional embeddings, and 96 attention heads per block), with advanced capabilities in language, code, and even reasoning.
GPT-4 (2023): Multi-modal capabilities (image and text), improved reasoning, and broader general knowledge (detailed information on the architecture are not yet available).

**GPT input encoding**

GPT models, including GPT-1, GPT-2, GPT-3, and later versions, use Byte-Pair Encoding (BPE) as their input encoding method.
BPE is a subword tokenization technique that strikes a balance between word-level and character-level representations, breaking down words into smaller, meaningful subunits (tokens) based on frequency in the training data.
The vocabulary size varies by model version (e.g., GPT-2 uses about 50,000 tokens), allowing efficient representation of both frequent and rare words.
The main features of BPE are:

- Subword Tokenization: BPE splits rare or complex words into subword units while keeping common words as single tokens. For instance, a rare word like "unhappiness" might be split into "un," "happi," and "ness.“
- Fixed Vocabulary: BPE produces a fixed-size vocabulary (e.g., around 50,000 tokens in GPT-2), containing common words, word fragments, and some single characters. This helps the model handle a wide range of text efficiently.
- Efficiency in Language Representation: Subword tokens allow GPT to represent a diverse range of language patterns, handling both common and rare words effectively while reducing the total number of tokens required.

The main advantages of BPE (similar toWordPiece) are:

- Flexibility: Handles languages with rich morphology or new words (e.g., "AI-generated") by breaking them down into reusable subword tokens.
- Reduced Vocabulary Size: Keeps the vocabulary smaller and training more efficient compared to a word-level tokenizer.
- Out-of-Vocabulary Handling: BPE is resilient to unknown words, as it can break down any new word into familiar subwords or characters.

**GPT pre-training**

GPT is pre-trained to predict the next word (or token) in a sequence, given all the previous tokens. This process is also known as autoregressive modeling. In its Next-Token Prediction strategy, At each step GPT model learns to minimize the difference between its predicted next token and the actual next token in the training sequence, effectively learning context and word relationships. The prediction is sequential, namely each token is predicted based only on previous tokens, so the model learns the patterns of language in a left-to-right order. GPT models are trained on massive and diverse datasets sourced from a wide array of internet text. GPT-1 was trained on BookCorpus, which consists of around 985 million words (800 MB of text). GPT-2 was trained on WebText (40 GB of text from around 8 million documents with 10 billion words), a dataset curated from high-quality web pages by OpenAI. GPT-3 used even larger datasets (570 GB of text with hundreds of billions of words), combining sources like Common Crawl, Books, Wikipedia, and

more. In all the cases, the data is selected to cover a broad range of topics and linguistic structures to make the model versatile across different domains.
GPT minimizes the cross-entropy loss between the predicted token probabilities and the actual tokens. This loss function is well-suited to classification tasks (like predicting the next token) and provides the model with feedback on its predictions. GPT uses the Adam optimizer, an adaptive gradient descent technique, which helps accelerate convergence by adjusting learning rates based on past gradients. GPT applies a learning rate scheduling in which learning rates are gradually increased (warm-up) in the early stages and then decayed to prevent instability during training. Large batch sizes are used to stabilize training and make the model better at generalizing across diverse language patterns.

**GPT fine tuning**

Fine-tuning of GPT requires a dataset labeled for the specific task, such as pairs of prompts and expected responses, or inputs and target outputs. Examples of tasks for which GPT has been or may be fine tuned are:

- Customer SupportAutomation: queries, issues, information.
- Legal Document Processing: summarize legal documents and support legal research.
- MedicalAssistance: health guidance and support patient inquiries.
- Coding Assistance: Provide code snippets, explanations, and debugging help.
- Educational Tutoring: Answer questions, explain concepts, and support e-learning.
- Content Creation: Generate blog posts, social media content, and marketing copy.
- Virtual Personal Assistants: Provide reminders, manage tasks, and answer questions.

**GPT strengths** are:

- Language Fluency and Coherence: GPT models generate human-like, fluent, and coherent text, often indistinguishable from human writing.
- Broad Knowledge Base: Trained on vast datasets, GPT has extensive general knowledge across a wide array of topics, allowing it to answer questions and generate content in diverse domains.
- Few-Shot and Zero-Shot Learning: GPT can perform tasks with little to no task-specific training by learning from examples in the prompt (fewshot) or adapting to a task without examples (zero-shot).
- Creative and Contextual Writing: GPT can generate creative content, including stories, poetry, and dialogues, which makes it useful for content creation and entertainment applications.
- Rapid Adaptation with Fine-Tuning: Fine-tuning on task-specific data allows GPT to perform well in specialized contexts, such as technical writing, legal assistance, and customer service.
- Scalability with Large Models: Larger GPT models demonstrate stronger performance and generalization, especially on complex or nuanced tasks.

GPT limitations are:

- Lack of True Understanding: GPT models generate text based on patterns rather than true comprehension.

- Sensitivity to Prompting: GPT's responses can vary widely based on phrasing. Small changes in prompts may yield different outcomes.
- Ethical and Bias Concerns: GPT can reproduce biases present in its training data, leading to biased or inappropriate outputs, especially if prompts or fine-tuning data lack diversity or sensitivity.
- Inability to Reason or Perform Complex Calculations: GPT is limited in logical reasoning, advanced mathematics, or tasks requiring step-by-step problem-solving without explicit instruction in the prompt
- High Computational Requirements: Large GPT models require significant computational power for training, fine-tuning, and deployment, making them expensive to run and maintain.
- Limited Memory Across Interactions: GPT lacks persistent memory across sessions, so it cannot retain information shared by users in previous interactions without explicit prompting.
- Vulnerability to Adversarial Prompts: Malicious prompts can manipulate GPT into producing undesirable or unintended responses.

**PopularGPT variants – Codex**

Codex is a GPT-3 model fine-tuned by OpenAI specifically for coding and programming tasks. It powers GitHub Copilot and can assist with code generation, debugging, and explanations across multiple programming languages. The key features are coding assistance, code generation, multi-language support for programming tasks.

**PopularGPT variants – MT-NLG**

MT-NLG is the acronym for Megatron-Turing Natural Language Generation and was developed by NVIDIA and Microsoft. MT-NLG is one of the largest language models, with 530 billion parameters. It aims to improve natural language understanding and generation in tasks like summarization, question answering, and robust few-shot learning.

**PopularGPT variants –GLaM**

GLaM is the acronym for Generalist Language Model, developed byGoogle Research .GLaM is a sparse mixture-of-experts model with 1.2 trillion parameters, but only a fraction are active per inference.  It uses fewer resources than fully dense models like GPT-3 while achieving competitive performance across NLP tasks.

**PopularGPT variants – PanGu-α**

PanGu-α is Huawei’s Chinese language model with 200 billion parameters, aimed at understanding and generating text in Mandarin. It was developed as part of Huawei’s effort to advance AI in Chinese NLP and has applications in Chinese literature, customer support, and translation. In general, it supports Chinese-specific language applications.

**PopularGPT variants – Chinchilla**

Chinchilla is a DeepMind model optimized for efficiency in training data and parameters. It has a smaller number of parameters than GPT-3 but achieves similar or better performance, demonstrating an alternative approach to large-scale model training.  It is thus optimized for research and practical applications.

**PopularGPT variants – OPT**

OPT is the acronym for Open Pretrained Transformer, developed by Meta (Facebook AI). OPT models are a series of open-source language models from Meta, comparable toGPT-3 in size and capabilities. Meta released these models to support transparency in AI research, offering model weights and code for research and academic use**.**

**PopularGPT variants – BLOOM**

BLOOM is the result of the BigScience collaborative project. It is an open-source multilingual model with 176 billion parameters, trained by a global consortium of researchers. It supports 46 languages, including underrepresented languages, and is designed to make large language models accessible for diverse linguistic and cultural contexts. It enforces inclusivity in NLP research.

**LLAMA**

The LLaMA (Large Language Model Meta AI) architecture is a family of transformer-based language models developed by Meta. LLaMA models are designed to be efficient, high-performing, and optimized for a range of NLP tasks. The LLaMA family includes several model sizes:

- LLaMA-7B: 32 decoder blocks with 32 attention heads for each block, 4096-dimensional embeddings
- LLaMA-13B: 40 decoder blocks with 40 attention heads for each block, 5120-dimensional embeddings
- LLaMA-30B: 60 decoder blocks with 40 attention heads for each block, 6656-dimensional embeddings
- LLaMA-65B: 80 decoder blocks with 64 attention heads for each block, 8192-dimensional embeddings

These models are designed to offer a range of capabilities depending on the computational resources available, from smaller more efficient models to larger models.

**LLAMA input encoding**

Also LLAMA models use Byte-Pair Encoding (BPE) as their input encoding method, obtaining a dictionary of 32768 tokens. However, LLaMA uses relative positional encodings instead of absolute

positional encodings. This method allows the model to better handle varying sequence lengths and to generalize across different contexts, which is particularly useful for longer sequences. In relative positional encoding, the model learns the relationships between tokens based on their relative positions rather than their absolute positions in the sequence.

**LLAMA pre-training**

Like GPT models, LLaMA is pre-trained using an autoregressive language modeling objective. This means that the model learns to predict the next token in a sequence given the previous tokens. LLaMA is trained on “The Pile” (825 GB, from 300 to 1000 billion tokens), a wide range of publicly available text sources, including books , Web Data and Scientific Papers. The dataset is designed to be as diverse as possible to ensure the model learns a broad understanding of language and can generalize well to a wide range of tasks. The loss function used during pre-training is the crossentropy loss between the predicted token probabilities and the actual next token in the sequence. This helps the model learn to predict the correct token given the context. The model is optimized using Stochastic Gradient Descent (SGD) or Adam optimizer with gradient clipping to prevent instability during training. Training also utilizes mixed precision to speed up computation and reduce memory requirements. LLaMA employs techniques like learning rate schedules (e.g., linear warm-up followed by decay), weight decay, and batch normalization or layer normalization to stabilize and improve training.

**LLAMA variants** are:

**LLAMA-7B**

- **Parameters**: 7 billion
- **Use Case**: Designed for resource-efficient tasks, such as small-scale NLP projects.
- **Strengths**: Highly efficient and well-suited for smaller computational environments.
- **Limitations**: May not deliver top-tier performance for complex tasks.

**LLAMA-13B**

- **Parameters**: 13 billion
- **Use Case**: Suitable for general-purpose NLP tasks and fine-tuning for specific applications.
- **Strengths**: Offers a balance between performance and computational efficiency.
- **Limitations**: May lack the performance required for more advanced tasks.

**LLAMA-30B**

- **Parameters**: 30 billion
- **Use Case**: Ideal for complex tasks like summarization and translation.
- **Strengths**: Provides high performance on state-of-the-art NLP tasks.
- **Limitations**: Requires significant computational resources to operate.

**LLAMA-65B**

- **Parameters**: 65 billion
- **Use Case**: Geared toward high-end applications and advanced research.
- **Strengths**: Delivers top-tier NLP performance across multiple domains.
- **Limitations**: Extremely resource-intensive and challenging to deploy due to high computational demands.

**LLAMA vs GPT**

**Size Range**

- LLAMA: Comes in smaller parameter ranges (7B, 13B, 30B, 65B).
- GPT: Ranges from 117M to 175B+ parameters (e.g., GPT-3).

**Training Data**

- LLAMA: Uses publicly available datasets like The Pile, Wikipedia, and Common Crawl.
- GPT: Also trained on public data, such as Common Crawl and WebText.

**Performance**

- LLAMA: Strong and competitive, particularly for smaller models.
- GPT: State-of-the-art performance, especially in zero-shot or few-shot tasks.

**Training Efficiency**

- LLAMA: More efficient and parameter-efficient in its design.
- GPT: Very resource-intensive, especially in larger versions like GPT-3.

**Deployment**

- LLAMA: Open-sourced, allowing for flexible deployment options.
- GPT: Provided as a commercial API through OpenAI, limiting direct access to the model.

**Ethics**

- LLAMA: Emphasizes strong ethical considerations in its development and use.
- GPT: Has faced criticism regarding transparency and potential biases in its training.

**Applications**

- LLAMA: Primarily used for academic research and custom deployments.
- GPT: Designed for broader commercial applications, APIs, and integration into various products.

**Encoder-decoder transformer**

Encoder-Decoder Transformers are a class of neural networks designed for sequence-to-sequence (seq2seq) tasks.

**T5**

T5 (Text-to-Text Transfer Transformer) is a language model based on an encoder-decoder transformer developed byGoogle Research.
T5 comes in multiple sizes to suit different resource constraints:

T5-Small:  6 Encoder Blocks, 8 Attention Heads, 6 Decoder Blocks,512 of Embedding Dimensionality.
T5-Base: 12 Encoder Blocks,12 Attention Heads,12 Decoder Blocks,768 of Embedding Dimensionality.
T5-Large:24 Encoder Blocks,16 Attention Heads,24  Decoder Blocks,1024 of Embedding Dimensionality.
T5-XL: 24 Encoder Blocks ,32 Attention Heads,24 Decoder Blocks,2048 of Embedding Dimensionality
T5-XXL: 24 Encoder Blocks, 64 Attention Heads, 24 Decoder Blocks,4096 of Embedding Dimensionality.

**T5 input encoding**

T5 uses a SentencePiece tokenizer with a custom vocabulary for its input encoding.
Subword Units: T5 employs a subword-based tokenizer using the SentencePiece library. Subword tokenization ensures a balance between character-level and word-level tokenization, effectively handling rare words and unseen combinations.

Unigram Language Model: The SentencePiece tokenizer in T5 is trained using a unigram language model, which selects subwords to maximize the likelihood of the training data.
T5 uses a fixed vocabulary of 32,000 tokens. The vocabulary includes subwords, whole words, and special tokens. This compact vocabulary size strikes a balance between computational efficiency and representation capacity.
T5 introduces several special tokens in the vocabulary to guide the model for various tasks:

- <pad>: Padding token for aligning sequences in a batch.
- <unk>: Unknown token for handling out-of-vocabulary (OOV) cases.
- <eos>: End-of-sequence token, marking the conclusion of an input or output sequence.
- <sep> and task-specific prefixes: Used to define input task types (e.g., "translate English toGerman:" or "summarize:").

For pre-training, T5 uses a denoising autoencoder objective called span-corruption. This objective involves masking spans of text (not just individual tokens) in the input sequence and training the model to predict those spans. Input Corruption: Random spans of text in the input are replaced with a special <extra_id_X> token.

- Original Input: "The quick brown fox jumps over the lazy dog.“
- Corrupted Input: "The quick <extra_id_0 > jumps <extra_id_1 > dog.“

Target Output: The model is trained to predict the original masked spans in sequential order.

- Target Output: <extra_id_0>  brown fox <extra_id_1> over the lazy.

This formulation forces the model to generate coherent text while learning contextual relationships between tokens.
Predicting spans, rather than individual tokens, encourages the model to learn:

- Global Context: How spans relate to the larger sentence or paragraph structure
- Fluency and Cohesion: Span prediction ensures generated outputs are natural and coherent.
- Task Versatility: The model is better prepared for downstream tasks like summarization, translation, and question answering.

T5 pre-training uses the C4 dataset (Colossal Clean Crawled Corpus), a massive dataset derived from Common Crawl:

- Size:Approximately 750GB of cleaned text.
- Cleaning: Aggressive data cleaning is applied to remove spam, duplicate text, and low-quality content.
- Versatility: The dataset contains diverse text, helping the model generalize across domains.
- Loss Function: Cross-entropy loss is used for predicting masked spans.
- Optimizer: T5 employs the Adafactor optimizer, which is memory-efficient and designed for large-scale training.
- Learning Rate Scheduling: The learning rate is adjusted using a warm-up phase followed by an inverse square root decay.

**T5 fine tuning**

Input and Output as Text: Fine-tuning continues the paradigm where the input and output are always text strings, regardless of the task.
ExampleTasks:

- Summarization:
  - Input: summarize: <document > →Output: <summary>
- Translation:
  - Input: translate English to French: <text> →Output: <translated_text>
- QuestionAnswering:
  - Input: question: <question> context: <context> →Output: <answer>

**PopularT5 variants – mT5**

mT5 (Multilingual T5) was developed to extend T5's capabilities to multiple languages. It was pre-trained on the multilingual Common Crawl dataset covering 101 languages.
Key Features are:

- Maintains the text-to-text framework across different languages.
- No language-specific tokenization, since it uses SentencePiece with a shared vocabulary across languages.
- Demonstrates strong multilingual performance, including on crosslingual tasks.

Applications are Translation, multilingual summarization, and cross-lingual question answering.
Limitations are:

- Larger model size due to the need to represent multiple languages in the vocabulary.
- Performance can vary significantly across languages, favoring those with more representation in the training data.

**PopularT5 variants – Flan-T5**

Flan-T5 is a fine-tuned version of T5 with instruction-tuning on a diverse set of tasks**.**
Key Features are:
• Designed to improve generalization by training on datasets formatted as instruction-response pairs. • Better zero-shot and few-shot learning capabilities compared to the originalT5.
Applications:
• Performs well in scenarios requiring generalization to unseen tasks, such as creative writing or complex reasoning.
Limitations:
• Requires careful task formulation to fully utilize its instructionfollowing capabilities

**PopularT5 variants – ByT5**

ByT5 (Byte-Level T5) processes text at the byte level rather than using subword tokenization.
Key Features:
• Avoids the need for tokenization, enabling better handling of noisy, misspelled, or rare words.
• Works well for languages with complex scripts or low-resource scenarios.
Applications:
• Robust for tasks with noisy or unstructured text, such as OCR or usergenerated content.
Limitations:
• Significantly slower and more resource-intensive due to longer input sequences (byte-level representation increases sequence length).

**PopularT5 variants –T5-3B and T5-11B**

T5-3B and T5-11B are larger versions of the original T5 with 3 billion and 11 billion parameters, respectively.
Key Features are:
• Improved performance on complex tasks due to increased model capacity.
• Suitable for tasks requiring deep contextual understanding and largescale reasoning.
Applications:
• Used in academic research and high-performance NLP applications where resources are not a constraint.
Limitations:
• Computationally expensive for fine-tuning and inference.
• Memory requirements limit their usability on standard hardware.

**PopularT5 variants – UL2**

UL2 (Unified Language Learning) is a general-purpose language model inspired by T5 but supports a wider range of pretraining objectives.
Key Features:
• Combines diverse learning paradigms: unidirectional, bidirectional, and sequence-to-sequence objectives.
• Offers state-of-the-art performance across a variety of benchmarks.
Applications:
• General-purpose NLP tasks, including generation and comprehension.
Limitations:
• Increased complexity due to multiple pretraining objectives.

**PopularT5 variants – MultimodalT5**

T5-Large Multimodal Variants combine T5 with vision capabilities by integrating additional modules for visual data.
Key Features:
• Processes both text and image inputs, enabling tasks like image captioning, visual question answering, and multimodal translation.
• Often uses adapters or encodes visual features separately.
Applications:
• Multimodal tasks combining vision and language.
Limitations:
• Computationally expensive due to the need to process multiple modalities.

**PopularT5 variants – Efficient T5**

Efficient T5 Variants are optimized for efficiency in resourceconstrained environments.
Examples:
• T5-Small/Tiny: Reduced parameter versions of T5 for lower memory and compute needs.
• DistilT5: A distilled version of T5, reducing the model size while retaining performance.
Applications:
• Real-time applications on edge devices or scenarios with limited computational resources.
Limitations:
• Sacrifices some performance compared to largerT5 models.

**T5 variants**

**mT5**

- **Purpose**: Multilingual NLP tasks.
- **Key Strengths**: Supports 101 languages.
- **Limitations**: Performance is uneven across languages.**Flan-T5**
- **Purpose**: Instruction-following tasks.
- **Key Strengths**: Demonstrates strong generalization.
- **Limitations**: Requires task-specific prompts for optimal results.

**ByT5**

- **Purpose**: Designed for text without tokenization.
- **Key Strengths**: Effective at handling noisy or unstructured text.
- **Limitations**: Slower due to the use of byte-level inputs.

**T5-3B/11B**

- **Purpose**: High-capacity NLP tasks.
- **Key Strengths**: Delivers exceptional performance.
- **Limitations**: Requires significant computational resources.

**UL2**

- **Purpose**: Addresses unified objectives.
- **Key Strengths**: Offers versatility across different tasks.
- **Limitations**: Comes with increased training complexity.

**Multimodal T5**

- **Purpose**: Handles vision-language tasks.
- **Key Strengths**: Combines text and image inputs effectively.
- **Limitations**: Higher computational cost.

**Efficient T5**

- **Purpose**: Designed for resource-constrained NLP applications.
- **Key Strengths**: Lightweight with faster inference times.
- **Limitations**: Reduced performance on certain tasks.

**Goal of the project**

The goal of the project is to create a chatbot that answers questions about the NLP and LLM 2024/2025 course.The questions can be not only related to the topics covered in the course, but can also concern more general information, such as the course teachers, recommended books, etc. Furthermore, the chatbot must answer ONLY questions related to the course, recognizing out-of-context questions and replying that it is not enabled to provide answers on topics outside the context of interest.The groups will deliver the code of the chatbot and a report in which they describe in detail their solution.

**Tools to use for the project:**

Students are free to use any tool or technology analyzed in the course, both the more recent ones associated with LLMs and the more classic ones of NLP. To achieve the required result, the students can develop parts of the chatbot with LLMs and other parts with more classic tools, with total freedom of choice as long as the various solutions identified are justified in the final report. Any LLM or other already available model can be used with or without modifications, as long as the groups are fully familiar with the tools they used and are able to answer questions about every aspect of the code and models.

**Chatbot evaluation procedure**

Before the project discussion, the course instructors will question the chatbot of the group in real time with a predefined set of questions on the course and evaluate the answers from the following points of view:

- Relevance:Judges if generated text answers the query.
- Fluency: Evaluates the readability and grammar of the output.
- Coherence: Checks logical flow and consistency.

Then, the course instructors will do another set of pre-defined questions to verify:

- Robustness: Evaluates resistance to adversarial misleading prompts (e.g. are you sure?)
- Precision: Evaluates the capability to recognize out-of-context questions (e.g. who is the king of Spain?)

The course instructors will give a grade to the performance according to these aspects.

**Pros and cons of full fine tuning**

Fine-tuning refers to adapting a pre-trained LLM to a specific task by training it further on a task-specific dataset.
Why Fine-Tune?

- Specialize LLMs for domain-specific tasks
- Improve accuracy and relevance for specific applications.

- Optimize performance on small, focused datasets.

Full Fine-Tuning, which updates all model parameters, allows to achieve high accuracy for specific tasks by fully leveraging the model's capacity, but it is computationally expensive and risks to overfit on small datasets.

**Other types of fine tuning**

**Parameter-Efficient Fine-Tuning (PEFT**): Updates only a subset of the parameters.
Examples are LoRA and Adapters.

**Instruction Fine-Tuning**: Used to align models with task instructions or prompts (user queries) enhancing usability in real-world applications

**Reinforcement Learning from Human Feedback (RLHF):** Combines supervised learning with reinforcement learning, rewarding models when they generate user-aligned outputs.

**Parameter efficient fine tuning (PEFT)**

**Parameter-Efficient Fine-Tuning**
Parameter-Efficient Fine-Tuning (PEFT) is a strategy developed to fine-tune large-scale pre-trained models, such as LLMs, in a computationally efficient manner while requiring fewer learnable parameters compared to standard fine-tuning methods. PEFT methods are especially important in the context of LLMs due to their massive size, which makes full fine-tuning computationally expensive and storage-intensive. PEFT is ideal for resource-constrained settings like edge devices or applications with frequent model updates. These techniques are supported and implemented in Hugging Face transformers and, in particular, in peft library.

**PEFT techniques**

**Low-Rank Adaptation (LoRA):** Approximates weight updates by learning low-rank matrices, performing a small parameterized update of the weight matrices in the LLM. It is highly parameter-efficient and widely adopted for adapting LLMs.

**Adapters**: They are small and trainable modules inserted within the transformer layers of the LLM, that allow to keep the pre-trained model's original weights frozen.

**Prefix Tuning**: Learns a set of continuous task-specific prefix vectors for attention layers, keeping the original model parameters frozen

**Low-Rank Adaptation (LoRA)**

LoRA assumes that the changes required to adapt a pretrained model for a new task lie in a low-dimensional subspace. Instead of fine-tuning all the parameters of the model, LoRA modifies only a small, trainable set of low-rank matrices that approximate these task-specific changes.

1. Base Model: A pre-trained transformer model is represented by its weight matrices $W$
2. Low-Rank Decomposition: Instead of directly modifying $W$, LoRA decomposes the weight update into two low-rank matrices: $\Delta W = A \times B$ where $A$ is a low-rank matrix ($m \times r$), $B$ is another low-rank matrix ($r \times n$) and $r$ is the rank, which is much smaller than $m$ or $n$, making $A$ and $B$ parameterefficient.
3. Weight Update:The effective weight during fine-tuning becomes $W' = W + \Delta W = W + A \times B$

**Weight Updates in Regular Fine-Tuning**

- Fine-tuning modifies the pretrained weights WWW by adding a weight update matrix ΔW\Delta WΔW.
- This approach requires updating all model parameters, which can be computationally expensive.

**Weight Updates in LoRA**

- Instead of directly updating WWW, LoRA introduces two low-rank matrices AAA and BBB.
- These matrices approximate ΔW\Delta WΔW while keeping the pretrained weights WWW frozen.
- The dimension rrr (a hyperparameter) controls the rank of AAA and BBB, reducing computational costs.

**Efficiency**

- LoRA reduces memory and computational overhead by updating only AAA and BBB, which are significantly smaller compared to WWW.
- This makes it highly efficient for fine-tuning large language models without modifying their entire parameter space.

**How LoRA works**

Freezing Pre-Trained Weights: LoRA keeps the original weight matrices $W$ of the LLM frozen during fine-tuning. Only the parameters in $A$ and $B$ are optimized for the new task (the pretrained knowledge is preserved).
Injecting Task-Specific Knowledge: The low-rank decomposition $A \times B$ introduces minimal additional parameters (less than 1% of the original model parameters) while allowing the model to learn taskspecific representations.
Efficiency: The number of trainable parameters is proportional to $r \times (m+n)$, which is significantly smaller than the full $m \times n$ weight matrix.
Inference Compatibility: During inference, the modified weights $W' = W + A \times B$ can be directly used, making LoRA-compatible models efficient for deployment

**Adapters**

Adapters are lightweight, task-specific neural modules inserted between the layers of a pre-trained transformer block.

These modules are trainable, while the original pre-trained model parameters remain frozen during fine-tuning.
Adapters require training only the small fully connected layers, resulting in significantly fewer parameters compared to full finetuning.
Since the base model remains frozen, the general-purpose knowledge learned during pre-training is preserved.

**Prefix tuning**

Instead of modifying the LLM's internal weights, prefix tuning introduces and optimizes a sequence of trainable "prefix" tokens prepended to the input.
These prefixes guide the LLM's attention and output generation, enabling task-specific adaptations with minimal computational and storage overhead.

The input sequence is augmented with a sequence of prefix embeddings: Modified Input:

[Prefix]+[InputTokens]

Prefix is a sequence of $m$ trainable vectors of size $d$, where $d$ is the model's embedding dimensionality. InputToken is the original token embeddings from the input sequence.

Prefix embeddings influence attention by being prepended to the keys ($K$) and values ($V$), conditioning how the model attends to the input tokens. Only the prefix embeddings are optimized during fine-tuning. Backpropagation updates the prefix parameters to align the model's outputs with task-specific requirements. m controls the trade-off between task-specific expressiveness and parameter efficiency. Longer prefixes can model more complex task-specific conditioning but may increase memory usage.

**Instruction fine tuning**

Instruction fine-tuning is a specialized approach for adapting large language models (LLMs) to better understand and respond to user instructions. This fine-tuning process involves training the model on a curated dataset of task-specific prompts paired with corresponding outputs. The objective is to improve the model's ability to generalize across a wide variety of instructions, enhancing its usability and accuracy in real-world applications. By training on human-like instructions, the model becomes more aligned with user expectations and natural language queries.

**How instruction fine tuning works**

A diverse set of instructions and outputs is compiled. Each example consists of:

- Instruction: A clear, human-readable prompt (e.g., "Summarize the following text in one sentence").
- Context (optional): Background information or data required to complete the task.
- Output:The expected response to the instruction.

The LLM, pre-trained on a general corpus, is fine-tuned using the instruction-response pairs. During training, the model learns to:

- Recognize the intent of the instruction
- Generate outputs that are coherent, accurate, and contextually appropriate.

The dataset may include examples from various domains and task types. This diversity ensures the model can generalize beyond the specific examples it has seen during fine-tuning.

**Introduction to Prompt Engineering**

**Prompt Engineering**

A relatively new discipline focused on developing and optimizing prompts to effectively use LLMs for diverse applications and research areas.
Goals are:

- Enhances understanding of the capabilities and limitations of LLMs
- Improves LLM performance on a broad range of tasks (e.g., question answering, arithmetic reasoning, ...)
- Helps interfacing with LLMs and integrating with other tools
- Enables new capabilities, such as augmenting LLMs with domain knowledge and external resources

**Writing Good Prompts**

Start with simple prompts, adding elements gradually while iterating and refining to improve results. Use clear, specific instructions (e.g., “Write,” “Classify,” “Summarize”) at the beginning of prompts. Be detailed and descriptive to achieve better outcomes. Consider using examples to guide the model’s output. Balance detail and length carefully, as excessive information can reduce effectiveness, and experiment to find the ideal format.

Examples:

- Bad Prompt: “Summarize this article.”
- Good Prompt: “Generate a 100-word summary of this research article, focusing on the main findings.”
- Bad Prompt: “Write an apology email to a client.”
- Good Prompt: “Write a professional email to a client apologizing for a delayed shipment, offering a discount, and providing an updated delivery estimate.”
- Bad Prompt: “Make this explanation easier to understand.”
- Good Prompt: “Rewrite this technical explanation in simpler language suitable for high school students.”
- Bad Prompt: “Classify the following review.”
- Good Prompt: “Classify the following review as positive, neutral, or negative”
- Bad Prompt: “Tell me about exercise benefits.”
- Good Prompt: “List five health benefits of regular exercise, each with a short explanation of how it improves well-being.”
- Bad Prompt: “Translate this sentence to French.”
- Good Prompt: “Translate the following English sentence into French, preserving the formal tone.”

Elements of a Prompt

A prompt usually contains any of the following elements:

- Instruction – a specific task or instruction you want the model to perform
- Context – external information or additional context that can steer the model to better responses
- Input Data – the input or question that we are interested to find a response for
- Output Indicator – the type or format of the output

Examples:

In :" Classify the text into neutral, negative or positive. Text: I think the vacation is okay. Sentiment:" "Sentiment:" is the output Indicator ,the input is "Text: I think the vacation is okay." And the Instrction is "Classify the text into neutral, negative or positive."

Another example is:

"Answer the question based on the context below. Keep the answer short and concise. Respond "Unsure about answer" if not sure about the answer. Context: Teplizumab traces its roots to a New Jersey drug company called Ortho Pharmaceutical. There, scientists generated an early version of the antibody, dubbed OKT3. Originally sourced from mice, the molecule was able to bind to the surface of T cells and limit their cell-killing potential. In 1986, it was approved to help prevent organ rejection after kidney transplants, making it the first therapeutic antibody allowed for human use. Question: What was OKT3 originally sourced from? Answer:"

In this example "Answer:" is the Output indicator, "Question: What was OKT3 originally sourced from?" is the input;
"Context: Teplizumab traces its roots to a New Jersey drug company called Ortho Pharmaceutical. There, scientists generated an early version of the antibody, dubbed OKT3. Originally sourced from mice, the molecule was able to bind to the surface of T cells and limit their cell-killing potential. In 1986, it was approved to help prevent organ rejection after kidney transplants, making it the first therapeutic antibody allowed for human use." Is the context

"Answer the question based on the context below. Keep the answer short and concise. Respond "Unsure about answer" if not sure about the answer." Is the instruction.

**In-Context Learning**

The ability of a LLM to perform a task by interpreting and leveraging information provided in its prompt (context) without updating its internal parameters.
A prompt context may specify…

- Reference Material: specific text or data to be used to perform the task
- Input-Output Pairs: examples of the task to illustrate the desired pattern
- Step-by-Step Instructions: detailed guidance for completing the task
- Clarifications: addressing potential ambiguities in the task
- Templates: structures or placeholders to be filled in

Prompt engineering heavily leverages in-context learning

**Prompts and NLP Tasks**

Prompts can be designed to achieve various NLP tasks:

- Text Summarization
- Information Extraction
- Question Answering
- Text Classification
- Code Generation
- Reasoning

Note: despite advancements in mathematical capabilities, reasoning tasks remain challenging for LLMs…

**System Prompts**

A configuration provided to the AI model before any user interactions

- Establishes the assistant's behavior, context, tone, and any special instructions
- Guides the model on how to respond and what it should focus on

Example:

- You are a helpful and knowledgeable assistant who answers questions accurately and concisely
- You are an IT support assistant specializing in troubleshooting software and hardware issues. Respond politely and guide users through step-by-step solutions.
- You are a friendly and engaging AI who responds in a warm and conversational tone, keeping responses lighthearted and approachable.

**Zero-Shot Prompting**

A prompt that interacts with the model without including examples or demonstrations. A zero-shot prompt gives the model direct instructions to perform a task without any additional examples for guidance. A zero-shot prompt gives the model direct instructions to perform a task without any additional examples for guidance.

Example:
Classify the text into neutral, negative or positive. Text: I think the vacation is okay. Sentiment Output: Neutral

the LLM already understands the concept of “sentiment” (that's the zero-shot capabilities at work)

**Few-Shot Prompting**

LLMs may fail with complex tasks in a zero-shot setting.
Few-shot prompting allows in-context learning, where examples or demonstrations are included in the prompt to guide the model toward better performance.
These demonstrations help the model generate more accurate responses for similar tasks in subsequent prompts.

Example:
A "whatpu" is a small, furry animal native to Tanzania. An example of a sentence that uses the word whatpu is: We were traveling in Africa, and we saw these very cute whatpus. To do a "farduddle" means to jump up and down really fast. An example of a sentence that uses the word farduddle is:

Output: When we won the game, we all started to farduddle in celebration.

**Limitations**

Few-shot prompting is effective for many tasks but remains limited when handling complex reasoning tasks.

Zero-Shot Example:
The odd numbers in this group add up to an even number: 15, 32, 5, 13, 82, 7, 1. A:

Output: Yes, the odd numbers in this group add up to 107, which is an even number.

False! Let’s try with few-shot prompting

Few-Shot Example:
The odd numbers in this group add up to an even number: 4, 8, 9, 15, 12, 2, 1. A: The answer is False. The odd numbers in this group add up to an even number: 17, 10, 19, 4, 8, 12, 24. A: The answer is True. The odd numbers in this group add up to an even number: 16, 11, 14, 4, 8, 13, 24. A: The answer is True. The odd numbers in this group add up to an even number: 17, 9, 10, 12, 13, 4, 2. A: The answer is False. The odd numbers in this group add up to an even number: 15, 32, 5, 13, 82, 7, 1. A:

Output: The answer is True.

False Again!

**Chain-of-Thought Prompting**

Enables complex reasoning capabilities through intermediate reasoning steps.
You can combine it with few-shot prompting to get better results on even more complex tasks

Example:

Model Input Q: Roger has 5 tennis balls. He buys 2 more cans of tennis balls. Each can has 3 tennis balls. How many tennis balls does he have now? A: The answer is 11. Q: The cafeteria had 23 apples. If they used 20 to make lunch and bought 6 more, how many apples do they have?

Model Output A: The answer is 27. Chain-of-Thought Prompting Model Input Q: Roger has 5 tennis balls. He buys 2 more cans of tennis balls. Each can has 3 tennis balls. How many tennis balls does he have now? A: Roger started with 5 balls. 2 cans of 3 tennis balls each is 6 tennis balls. 5 + 6 = 11. The answer is 11. Q: The cafeteria had 23 apples. If they used 20 to make lunch and bought 6 more, how many apples do they have? Model Output A: The cafeteria had 23 apples originally. They used 20 to make lunch. So they had 23 - 20 = 3. They bought 6 more apples, so they have 3 + 6 = 9. The answer is 9. ✓

Example:
The odd numbers in this group add up to an even number: 4, 8, 9, 15, 12, 2, 1. A: Adding all the odd numbers (9, 15, 1) gives 25. The answer is False. The odd numbers in this group add up to an even number: 17, 10, 19, 4, 8, 12, 24. A: Adding all the odd numbers (17, 19) gives 36. The answer is True. The odd numbers in this group add up to an even number: 16, 11, 14, 4, 8, 13, 24. A: Adding all the odd numbers (11, 13) gives 24. The answer is True. The odd numbers in this group add up to an even number: 17, 9, 10, 12, 13, 4, 2. A: Adding all the odd numbers (17, 9, 13) gives 39. The answer is False. The odd numbers in this group add up to an even number: 15, 32, 5, 13, 82, 7, 1. A:

Output: Adding all the odd numbers (15, 5, 13, 7, 1) gives 41. The answer is False.

Correct!

Note: this is an emergent ability that arises with sufficiently large language models.

**Self-Consistency Prompting**

Uses an iterative chain-of-thought approach Instead of accepting the first response from the LLM, the question is repeated multiple times to generate a range of reasoning paths, with the most frequent answer being selected.

**Meta Prompting**

A prompting technique that guides the model through the logical steps required to solve a problem without relying on specific content-based examples

**Task-Agnostic Meta Prompting**

Sometimes good results can be achieved by simply asking the model to reason step-by-step, without specifying the content of each step.

This process entails utilizing LLMs to generate promp tsthat guide further tasks

**Meta Meta Prompting**

This process entails utilizing LLMs to generate prompts that guide further tasks

Prompt Chaining

For complex queries, sometimes using a single prompt is not enough.
Prompt chaining involves breaking down a task into different parts, each handled by a specific prompt.
The initial prompt covers only the first phase of the task.
The output obtained from the model is used as part of the input for the next prompt.
And so on, until reaching the final result.
A common use case for this technique is answering questions based on documents.

Example:

A. Extract any relevant quotes from the following text to answer the question "What was the main cause of the American Civil War?" [reference text follows]
B. "One of the biggest catalysts of the Civil War was the issue of slavery, particularly the Southerners' desire to bring slavery into the territories." "Tensions peaked when the Republican candidate Abraham Lincoln was elected president in 1860 with a platform opposing the extension of slavery into the territories, which isolated the South."
A. Using the extracted quotes and the full text, provide a detailed answer to the question "What was the main cause of the American Civil War?"
B. The main cause of the American Civil War was the issue of slavery and its potential expansion into new territories and states. The South insisted on bringing slavery into new territories, while the election of Abraham Lincoln with an anti-slavery platform isolated the Southern states, which feared losing the right to own slaves. The fundamental disagreement over slavery between the North and South was the key factor that ultimately led to the outbreak of the Civil War

**Role Prompting**

Involves asking the model to take on a specific role or act in a certain way;can be used to improve accuracy in problem-solving. In this way, tone, style, and depth of information provided is adjusted accordingly.
Examples:

- Write a review of [pizza place]
- You are a food critic writing for the Michelin Guide. Write a review of [pizza place]
- You are a marketing expert. Draft an email to a client advising them about a delay in the delivery schedule due to logistical problems
- You are a customer service representative. Draft an email to a client advising them about a delay in the delivery schedule due to logistical problems.

Without RP: Write a review of [pizza place]

I recently had the pleasure of trying [random pizza place] and I must say that I was thoroughly impressed. The pizza was cooked to perfection and the toppings were fresh and flavorful. The crust was light and crispy, and the sauce was just the right amount of tangy. The service was also top-notch; the staff was friendly and attentive. Overall, I had a great experience and would definitely recommend [random pizza place] to anyone looking for a delicious pizza.

With RP: You are a food critic writing for the Michelin Guide. Write a review of [pizza place]

Located in the heart of [random city], [random pizza place] is a popular spot for pizza lovers. Upon entering, the warm, inviting atmosphere and the smell of freshly baked pizza will make you feel right at home. The menu offers a wide selection of classic pizzas, as well as some unique and creative options. The crust is perfectly crispy and the toppings are always fresh and flavorful. The service is friendly and attentive, and the prices are reasonable. Whether you're looking for a quick bite or a leisurely dinner, [random pizza place] is sure to satisfy your pizza cravings.
You are a marketing expert. Draft an email to a client advising them about a delay in the delivery schedule due to logistical problems.

Subject: A Little More Wait for Your Wonderful Products Dear [Client's Name], We hope this message finds you buzzing with excitement for your upcoming order! However, even the best-laid plans sometimes face unexpected hiccups. We have hit a temporary logistics snag that will unfortunately add a small wait time to your delivery. While we understand this isn't ideal, we believe great things come to those who wait, and your order is no exception. Rest assured, from the moment we sort out this situation to the time your products reach you, we're committed to ensuring that your experience is nothing short of excellent. Thank you for your patience and understanding. Stay tuned for more updates, and please, don't hesitate to reach out if you have any questions – we're always here. Cheers, [Your Name]

You are a customer service representative. Draft an email to a client advising them about a delay in the delivery schedule due to logistical problems.
Subject: Attention Required: Update on Your Order Status Dear [Client's Name], Greetings from [Your Company Name]. We wanted to inform you that, due to unforeseen logistical challenges, there might be a delay in the delivery of your order. We understand that this may cause some inconvenience, and we apologize sincerely. Meanwhile, we are working tirelessly to resolve this and ensure your delivery reaches you by [new estimated date]. We value your time and patience greatly. Please feel free to reach out if you have any questions or need further information. Our team is always here to assist you. Thank you for your understanding. Best regards, [Your Name]

**Structured Prompting**
Based on a semi-formal definition of instructions to help LLMs handle complex tasks more predictably. The prompt is divided in sections and encoded into a script. Delimiters are used to help LLMs recognize each part of the prompt as distinct units of meaning. Any unique character sequence that wouldn't normally appear together can serve as a delimiter (for example : ###, ===, >>>).
Another approach is to use XML tags as delimiters. LLMs are often trained on web content and have learned to recognize and understand this formatting.

EXAMPLE:

Classify the sentiment of each conversation in <<<CONVERSATIONS>>> as ‘Positive’ or ‘Negative’. Give the sentiment classifications without any other preamble text.
### EXAMPLE CONVERSATIONS
[Agent]: Good morning, how can I assist you today?
[Customer]: This product is terrible, nothing like what was advertised!
[Customer]: I’m extremely disappointed and expect a full refund.

[Agent]: Good morning, how can I help you today?
[Customer]: Hi, I just wanted to say that I’m really impressed with your product. It exceeded my expectations!

### EXAMPLE OUTPUTS
Negative
Positive

###
<<<
[Agent]: Hello! Welcome to our support. How can I help you today?
[Customer]: Hi there! I just wanted to let you know I received my order, and it’s fantastic!
[Agent]: That’s great to hear! We’re thrilled you’re happy with your purchase. Is there anything else I can assist you with?
[Customer]: No, that’s it. Just wanted to give some positive feedback. Thanks for your excellent service!

[Agent]: Hello, thank you for reaching out. How can I assist you today?
[Customer]: I’m very disappointed with my recent purchase. It’s not what I expected at all.
[Agent]: I’m sorry to hear that. Could you please provide more details so I can help?
[Customer]: The product is of poor quality, and it arrived late. I’m really unhappy with this experience. >>>

EXAMPLE:

Classify the sentiment of the following conversations into one of two classes, using the examples given. Give the sentiment classifications without any other preamble text.
<classes>
Positive
Negative
</classes>
<example-conversations>
[Agent]: Good morning, how can I assist you today?
[Customer]: This product is terrible, nothing like what was advertised!
[Customer]: I’m extremely disappointed and expect a full refund.
[Agent]: Good morning, how can I help you today?

[Customer]: Hi, I just wanted to say that I'm really impressed with your product. It exceeded my expectations!
</example-conversations>
<example-classes>
Negative
Positive
</example-classes>
<conversations>
[Agent]: Hello! Welcome to our support. How can I help you today?
[Customer]: Hi there! I just wanted to let you know I received my order, and it's fantastic!
[Agent]: That's great to hear! We're thrilled you're happy with your purchase. Is there anything else I can assist you with?
[Customer]: No, that's it. Just wanted to give some positive feedback. Thanks for your excellent service!

[Agent]: Hello, thank you for reaching out. How can I assist you today?
[Customer]: I'm very disappointed with my recent purchase. It's not what I expected at all.
[Agent]: I'm sorry to hear that. Could you please provide more details so I can help?
[Customer]: The product is of poor quality, and it arrived late. I'm really unhappy with this experience.
</conversations>

**Structured Prompting**

The CO-STAR framework divides a prompt into the following sections:

- Context: Provides background information on the task to help the LLM understand the specific scenario
- Objective: Clearly defines the task the LLM should perform
- Style: Specifies the desired writing style
- Tone: Sets the response's tone to match the desired sentiment or emotional context (e.g., formal, humorous, empathetic)
- Audience: Identifies the intended audience (experts, beginners, children, etc.)
- Response: Defines the response format to ensure compatibility with subsequent steps (e.g., free text, list, table, JSON)

**Generate Knowledge Prompting**

This method first prompts the LLM to generate relevant knowledge related to a task, and then incorporates that knowledge into the prompt along with the task description or question. Particularly useful when the LLM lacks the specific information required to directly answer a query. Leverages the LLM's capacity to generate supplementary knowledge beyond its base training domain.

Example:

List and describe the key factors that influence the evolution of life in environments with extreme gravitational forces, such as on a super-Earth planet. Focus on biological, physiological, and ecological adaptations that might arise in such conditions.

Using the adaptations and factors you described earlier, design a hypothetical intelligent species that could evolve on a super-Earth planet with extreme gravitational forces. Include details about

their physical structure, social behaviors, methods of tool use or communication, and how their adaptations influence their culture or technology.

**Retrieval Augmented Generation**

Retrieval-Augmented Generation (RAG) combines retrieval techniques with text generation. Addresses limitations in LLMs accessing updated or domain-specific data. A search or retrieval system (e.g., databases, search engines) is used to find relevant documents or data. An LLM is used to generate responses, conditioned on retrieved data.

**Prompt Testing**

Experimenting with various prompts is essential for achieving optimal responses across different use cases.

Prompt Testing Tools…

- Simplify prompt creation and testing, enabling iterative adjustments to discover the best structure and format
- Support customizable model settings to control output style, tone, and precision

Some available tools:

- OpenAI Playground: Supports GPT models
- Google AI Studio: Supports Google Gemini models
- LM Studio: Supports Hugging Face models

**LLM Settings**

When designing prompts, you interact with the LLM via an API, where you can adjust several key parameters:

- Temperature: Controls randomness. Lower values (e.g., 0.2) make responses more deterministic, suitable for factual tasks. Higher values (e.g., 0.8) encourage creativity, ideal for tasks like poem generation
- Top P: Adjusts response diversity by limiting token choices to a probability threshold. Lower values ensure precision, while higher values encourage more varied outputs. Example : if Top P = 0.5, the model considers only the most probable tokens until their summed probability adds up to 50%
- Max Length: Sets the token limit for responses, helping to control response length and cost

When designing prompts, you interact with the LLM via an API, where you can adjust several key parameters:

- Stop Sequences: Define a stopping point for responses, which can prevent overly long outputs and help structure responses, such as ending when a particular token is generated
- Frequency Penalty: Reduces repetition by penalizing words based on their frequency in the response, useful for avoiding redundant language
- Presence Penalty: Applies a consistent penalty to repeated tokens, regardless of how many times they appear. Higher values encourage more varied language
- Response Format: Expected format of the response (text, Json, …)

OpenAI Playground is at this link :" https://platform.openai.com/playground/"
Google AI Studio is at this link : "https://aistudio.google.com"
LM Studio is at this link :" https://lmstudio.ai/"

**LM Studio**

A Desktop Application for Local LLM Development and Experimentation.
Key functionality:

- Search & Download Models directly from Hugging Face
- Run LLMs on your computer
- Interactive Chat Interface to test and interact with LLMs
- Local API Server enabling LLM integration with external applications
- Model Management Tools to organize and configure local models

Additional info on "https://lmstudio.ai/"

What is RAG?

LLMs can reason about wide-ranging topics, but:

- Their knowledge is limited to data used during training
- They cannot access new information introduced after training
- They cannot reason about private or proprietary data

Retrieval Augmented Generation (RAG) is a technique to augment the knowledge of large language models (LLMs) with additional data.
RAG enables the creation of AI applications that can reason about private data and data introduced after a model's cutoff date

**RAG Concepts**

A typical RAG application has two main components:

- Indexing: a pipeline for ingesting data from a source and indexing it. This usually happens offline.
- Retrieval and generation:

- Takes the user query at run time
- Retrieves the relevant data from the index
- Generates a prompt that includes both the query with the retrieved data
- Uses an LLM to generate an answer

**Indexing**

Load: first we need to load our data. Popular RAG frameworks include document loaders for a variety of formats (PDF, CSV, HTML, Json, …) and sources (file system, Web, databases, …).
Split: large documents are split into smaller chunks… .For indexing (smaller chunks are easier to search). for LLM usage (smaller chunks fit within the model's finite context window).
Store: We need somewhere to store and index our splits, so that they can be searched over later. This is often done using a Vector Store.

**Vector Stores**

Specialized data stores that enable indexing and retrieving information based on embeddings.
• Recap: embeddings are vector representations that encapsulate the semantic meaning of data
• Advantage: information is retrieved based on semantic similarity, rather than relying on keyword matches

**Retrieval and Generation**

Given a user input, relevant splits are retrieved from storage.
A LLM produces an answer using a prompt that includes both the question with the retrieved data

**Introduction to LangChain**

**LangChain**

A framework built to simplify the development of applications that utilize LLMs. It provides a set of building blocks to incorporate LLMs into applications. It connects to third-party LLMs (like OpenAI and HuggingFace), data sources (such as Slack or Notion), and external tools. It enables the chaining of different components to create sophisticated workflows. It has open-source and commercial components. It supports several use cases like chatbots, document search, RAG, Q&A, data processing, information extraction …

**Key Components**

- Prompt Templates: Facilitate translating user input into language model instructions, supporting both string and message list formats
- LLMs: Third-party language models that accept strings or messages as input and return strings as output
- Chat Models: Third-party models using sequences of messages as input and output, supporting distinct roles within conversational messages
- Example Selectors: Dynamically select and format concrete examples in prompts to enhance model performance

- Output Parsers: Convert model-generated text into structured formats (e.g., JSON, XML, CSV), supporting error correction and advanced features
- Document Loaders: Load documents from various data sources
- Vector Stores: Systems for storing and retrieving unstructured documents and data using embedding vectors
- Retrievers: Interfaces for document and data retrieval compatible with vector stores and other external sources
- Agents: Systems leveraging LLMs for reasoning to decide actions based on user inputs

**Installation**

- Install core LangChain library **:** **pip install langchain**
- Install community-contributed extensions : **pip install langchain_community**
- Installs Hugging Face integration for LangChain : **pip install langchain_huggingface**
- Install a library to load, parse, and extract text from PDF: **pip install pypdf**
- Install the FAISS vector store (CPU version): **pip install faiss-cpu**

**Preliminary Steps**

Obtain a Hugging Face access token.
Copy your Access token
Gain access to the Mistral-7B-Instruct-v0.2 model by accepting the user license

**Query a LLM Model**

**Prompt Templates**

Predefined text structures used to create dynamic and reusable prompts for interacting with LLMs

**Introduction to Chains**

Chains combine multiple steps in an NLP pipeline. The output of each step becomes the input for the next. Useful to automate complex tasks, and integrate external systems. We want to refine the LLM output for future processing…

**Chaining all Together**

This forwards the output of the previous step to the next step associated with a specific dictionary key

**More on Chains**

LCEL (LangChain Expression Language) is a syntax to create modular pipelines for chaining operations. Pipe Syntax: The | operator allows chaining of operations. LCEL supports modular and reusable components. It can handle branching logic or follow-up queries LangChain includes Predefined Chains for common tasks like question answering, document summarization, conversational agents, etc.

**Building a RAG with LangChain and HuggingFace**

**Example Project**

Building a RAG able to answer queries on some documents from the Census Bureau US. An agency responsible for collecting statistics about the nation, its people, and its economy. We first download some documents to be indexed. LangChain components used to extract content from diverse data sources:

- TextLoader: Handles plain text files
- PyPDFLoader: Extracts content from PDF documents
- CSVLoader: Reads tabular data from CSV files
- WebBaseLoader: Extracts content from web pages
- WikipediaLoader: Fetches Wikipedia pages
- SQLDatabaseLoader: Queries SQL databases to fetch and load content
- MongoDBLoader: Extracts data from MongoDB collections
- IMAPEmailLoader: Loads emails using the IMAP protocol from various providers
- HuggingFaceDatasetLoader: Fetches datasets from Hugging Face datasets library
- and many others…

**Extract Content from PDFs**

We can use PyPDFLoader to extract text from a single PDF document.

Or we can use PyPDFDirectory to fetch a set of PDF documents from a folder .

**Text Splitters**

LangChain components used to break large text documents into smaller chunks:

- CharacterTextSplitter: Splits text into chunks based on a character count

- RecursiveCharacterTextSplitter: Attempts to split text intelligently by using hierarchical delimiters, such as paragraphs and sentences
- TokenTextSplitter: Splits text based on token count rather than characters
- HTMLHeaderTextSplitter: Splits HTML documents by focusing on headers to create meaningful sections from structured web content
- HTMLSectionSplitter: Divides HTML content into sections based on logical groupings or structural markers
- NLPTextSplitter: Uses NLP to split text into chunks based on semantic meaning
- and many others…

**Split Text in Chunks**

We use RecursiveCharacterTextSplitter to obtain…
Chunks of about 700 characters
Overlapped for about 50 characters

Printing the average document length before and after splitting…

**Index Chunks**

Embeddings are used to index text chunks, enabling semantic search and retrieval.
We can use pre-trained embedding models from Hugging Face.
Bi-Directional Generative Embeddings (BGE) models excel at capturing relationships between text pairs. BAAI/bge-small-en-v1.5 is a lightweight embedding model from the Beijing Academy of Artificial Intelligence. Suitable for tasks requiring fast generation.

**Vector Stores**

Enable semantic search and retrieval by indexing and querying embeddings of documents or text chunks:

- FAISS: Open-source library for efficient similarity search and clustering of dense vectors, ideal for local and small-to-medium datasets
- Chroma: Lightweight and embedded vector store suitable for local applications with minimal setup
- Qdrant: Open-source vector database optimized for similarity searches and nearest-neighbor lookup
- Pinecone: Managed vector database offering real-time, high-performance semantic search with automatic scaling
- and many others…

We use the Facebook AI Similarity Search (FAISS)

**Querying the Vector Store**

We can use the vector store to search for chunks relevant to a user query

**RAG Prompt Template**

Must integrate retrieved context with a user question.
A placeholder context is used to dynamically inject retrieved chunks.
A placeholder question is used to specify the user query.
Explicit instructions are provided for handling the information.

**Vector Store as a Retriever**

To be used in chains, a vector store must be wrapped within a retriever interface.
A retriever takes a text query as input and provides the most relevant information as output

**Predefined Chains**

LangChain include ready-to-use chains that handle common tasks with minimal setup.

- LLMChain: Executes a single prompt using a language model and returns the output
- RetrievalQAChain: Combines a retriever and an LLM to answer questions based on retrieved context
- AnalyzeDocumentChain: Extracts insights, structured data, or key information from documents
- SequentialChain: Executes a series of chains sequentially, passing outputs from one as inputs to the next
- ConditionalChain: Executes different chains based on conditions in the input or intermediate outputs
- and many others…

**Predefined RAG Chain**

We can use a predefined RetrievalQA chain instead of our custom chain

**Querying the RAG**

We can now use the RAG for question answering

Which chunks have been used to generate the answer?

**Reinforcement Learning from Human Feedback (RLHF)**

What is RLHF?

- A technique to improve large language models (LLMs) using human feedback as guidance.
- A strategy to balance model performance with alignment to human values and preferences.

Why RLHF?

- It may be a possible strategy to ground the focus of the LLM
- It can enhance safety, ethical responses, and user satisfaction.

**Workflow of RLHF**

XX
FOTO  PDF 21-penuiltyimo
XX

**Key components of RLHF**

**Pre-trained Language Model**

- A base LLM trained on large corpora (e.g., BERT, GPT, T5).

**Reward Model**

- A secondary model that scores LLM outputs based on human feedback.

**Fine-Tuning with Reinforcement Learning**

- Optimization of the LLM using reinforcement learning guided by the reward model.

**Reward model**

The inputs for training a reward model are:

- Multiple LLM generated outputs for given prompts
- Corresponding human rank responses according to their preferences

The goal is to train a model to predict human preference scores.
The methodology uses a ranking loss function to teach the reward model which outputs humans prefer.

**Fine tuning with proximal policy optimization (PPO)**

The goal is to align the LLM's outputs with humandefined quality metrics.

1. Generate responses using the LLM.
2. Score responses with the reward model.
3. Update the LLM to maximize reward scores.

**Pros and cons of RLHF**

**Pros**

- Iterative Improvement: Possibility to collect human feedback as the model evolves and update the reward model and fine-tune iteratively.
- Improved Alignment: Generates responses closer to human intent.
- Ethical Responses: Reduces harmful or biased outputs.
- User-Centric Behavior: Tailors interactions to user preferences.

**Cons**

- Subjectivity: Human feedback may vary widely.
- Scalability: Collecting sufficient, high-quality feedback is resource-intensive
- Reward Model Robustness: Misaligned reward models can lead to suboptimal fine-tuning.

**Tasks to enhance with RLHF**

- Text Generation: RLHF can be used to enhance the quality of text produced by LLMs
- Dialogue Systems: RLHF can be used to enhance the performance of dialogue systems
- Language Translation: RLHF can be used to increase the precision of language translation.
- Summarization: RLHF can be used to raise the standard of summaries produced by LLMs.
- Question Answering: RLHF can be used to increase the accuracy of question answering.
- Sentiment Analysis: RLHF has been used to increase the accuracy of sentiment identification for particular domains or businesses
- Computer Programming: RLHF can be used to speed up and improve software development.

**Case study: GPT-3.5 and GPT-4**

The pre-trained models have been fine-tuned using also RLHF.
OpenAI declares that achieved with RLHF

- Enhanced alignment
- Fewer unsafe outputs
- More human-like interactions.

These models were or are widely used in real-world applications like ChatGPT.
The models are still incrementally improved with additional human feedback.

**TRL (Transformer Reinforcement Learning)**

TRL is a full stack library where we provide a set of tools to train transformer language models with Reinforcement Learning, from the Supervised Fine-tuning step (SFT), Reward Modeling step (RM) to the Proximal Policy Optimization (PPO) step.
The library is integrated with HuggingFace transformers

We have 3 stemps:

- Step 1 - SFTTrainer : Train your model on your favorite dataset
- Step 2 – RawardTrainer : Train a preference model on a comparison data to rank generations from the supervised fine-tuned (SFT)
- Step 3 - PPOTrainer : Further optimize the SFT model using the rewards from the reward model and PPO algorithm

**Transformers for text representation and generation**

Transformers for text representation and generation

Input – input tokens
Output – hidden states
Model can see all timesteps
Does not usually output tokens, so no inherent auto-regressivity.
Can also be adapted to generate tokens by appending a module that maps hidden state dimensionality to vocab size

Input – output tokens and hidden states*
Output – output tokens
Model can only see previous timesteps
Model is auto-regressive with previous timesteps' outputs.
Can also be adapted to generate hidden states by looking before token outputs

Representation
(Schema a sinistra)

Positional Encoding
Input Embedding
Add & Norm
Multi-Head Attention
Add & Norm
Feed Forward
(Iterato Nx volte)
Generation
(Schema a destra)

Positional Encoding
Output Embedding
Add & Norm
Masked Multi-Head Attention
Add & Norm
Multi-Head Attention
Add & Norm
Feed Forward
(Iterato Nx volte)

Output Probabilities:

Softmax
Linear

**Paradigm shift in NLP**

Before LLMs:

- Freature Enigineering
  - How do we design or select the best features for a task?
- Model Selection
  - Which model is the best for which type of task?
- Transer Learning
  - Given scarce labeled data, how do we transer knowledge from other domanis?
- Overfitting Vs Generalization
  - How do we balance complexity and capacity to prevent overfitting while naintaining performance?

Since LLMs:

- Pre-training Fine-tuning
  - How do we leverage large scales of unlabeled data out there previusly under-leveraged?
- Zero-shot and Few-shot learning
  - How can we make models perform on tasks they are not trained on?
- Prompting
  - How do we make models understand their task simply by describing it in natural language?
- Interpretability and Explainability
  - How can we understand the inner workings of our own models?

What has caused this paradigm shift?

Problem in recurrent networks:

- Information is effectively lost during encoding of lonf sequences.
- Sequential nature disables parallel training and flavors late timestep inputs

Solution: Attention mechanism

- Handling long-range dependecies
- Parallel training
- Dynamic attention weights based on inputs

**Pre-training of LLMs**

**Self supervised pre-training**

Pre-training a large language model is done in a selfsupervised way, meaning it is trained with unlabeled data which is just text from the internet. There is no need to assign labels on the dataset. No supervision? Create supervised tasks and solve them. Autoencoding models consist only of an encoder and typically predict the masked word from the every preceding and following words in the sentence, therefore it is bi-directional. The model has the knowledge of the entire context.

**Masked Language Modeling**

Input text with randomly masked tokens is fed into a Transformer encoder to predict the masked tokens As illustrated in the figure, an original text sequence “I”, “love”, “this”, “red”, “car” is prepended with the “<cls>” token, and the “<mask>” token randomly replaces “love”; then the cross-entropy loss between the masked token “love” and its prediction is to be minimized during pre-training.

**Self supervised pre-training**

Autoregressive models consist only of a decoder and predict the masked word from the preceding words. Thus, autoregressive models are great at autocompleting a sentence, which is what happens in text generation models.

**Next Token Prediction**

Any text can be used for this pre-training task, which only requires the prediction of the next word in the sequence.

**Self supervised pre-training**

In training seq2seq models, random sequences of input are masked and replaced with a unique token sentinel. The output is the sentinel token followed by the predicted tokens. In summary, seq2seq models both need to understand the context and generate a text.

**Span corruption**

In the original text, some words are dropped out with a unique sentinel token. Words are dropped out independently uniformly at random. The model is trained to predict basically sentinel tokens to delineate the dropped out text.

**Summary on pre-training**

Pre-training tasks can be invented flexibly and effective representations can be derived from a flexible regime of pretraining tasks. Different NLP tasks seem to be highly transferable, producing effective representations that form a general model which can serve as the backbone for many specialized models. With Self-Supervised Learning the models seem to be able to learn from generating the language itself, rather than from any specific task. Language Model can be used as a Knowledge Base, namely a generatively pretrained model may have a decent zero-shot performance on a range of NLP tasks.

**Datasets and data preprocessing**

**Datasets**

Training LLMs require vast amounts of text data, and the quality of this data significantly impacts LLM performance. Pre-training on large-scale corpora provides LLMs with a fundamental understanding of language and some generative capability. Pre-training data sources are diverse, commonly incorporating web text, conversational data, and books as general pre-training corpora. Leveraging diverse sources of text data for LLM training can significantly enhance the model's generalization capabilities.

**Datasets – Books**

Two commonly utilized books datasets for LLMs training are BookCorpus andGutenberg. These datasets include a wide range of literary genres, including novels, essays, poetry, history, science, philosophy, and more. Widely employed by numerous LLMs, these datasets contribute to the models' pre-training by exposing them to a diverse array of textual genres and subject matter, fostering a more comprehensive understanding of language across various domains. Book Corpus includes 800 million words

**Datasets – CommonCrawl**

CommonCrawl manages an accessible repository of web crawl data, freely available for utilization by individuals and organizations. This repository encompasses a vast collection of data, comprising over 250 billion web pages accumulated over a span of 16 years. This continuously expanding corpus is a dynamic resource, with an addition of 3–5 billion new web pages each month. However, due to the presence of a substantial amount of low-quality data in web archives, preprocessing is essential when working with CommonCrawl data.

**Datasets – Wikipedia**

Wikipedia, the free and open online encyclopedia project, hosts a vast repository of high-quality encyclopedic content spanning a wide array of topics. The English version of Wikipedia, including 2,500 million words, is extensively utilized in the training of many LLMs, serving as a valuable resource for language understanding and generation tasks. Additionally, Wikipedia is available in multiple languages, providing diverse language versions that can be leveraged for training in multilingual environments

**Data pre-processing**

Once an adequate corpus of data is collected, the subsequent step is data preprocessing, whose quality directly impacts the model's performance and security. The specific preprocessing steps involve filtering lowquality text, including eliminating toxic and biased content to ensure the model aligns with human ethical standards. It also includes deduplication, removing duplicates in the training set, and excluding redundant content in the test set to maintain the sample distribution balance. Privacy scrubbing is applied to ensure the model's security, preventing information leakage or other privacy-related concerns.

**Quality filtering**

Filtering low-quality data is typically done using heuristic-based methods or classifier-based methods. Heuristic methods involve employing manually defined rules to eliminate low-quality data. For instance, rules could be set to retain only text containing digits, discard sentences composed entirely of uppercase letters, and remove files with a symbol and word ratio exceeding 0.1, and so forth. Classifier-based methods involve training a classifier on a high-quality dataset to filter out low-quality datasets

**Deduplication**

Language models may sometimes repetitively generate the same content during text generation, potentially due to a high degree of repetition in the training data. Extensive repetition can lead to training instability, resulting in a decline in the performance of LLMs. Additionally, it is crucial to consider avoiding dataset contamination by removing duplicated data present in both the training and test set.

**Privacy scrubbing**

LLMs, as text-generating models, are trained on diverse datasets, which may pose privacy concerns and the risk of inadvertent information disclosure. It is imperative to address privacy concerns by systematically removing any sensitive information. This involves employing techniques such as anonymization, redaction, or tokenization to eliminate personally identifiable

details, geolocation, and confidential data. By carefully scrubbing the dataset of such sensitive content, researchers and developers can ensure that the language models trained on these datasets uphold privacy standards and mitigate the risk of unintentional disclosure of private information.

**Filtering out toxic and biased text**

In the preprocessing steps of language datasets, a critical consideration is the removal of toxic and biased content to ensure the development of fair and unbiased language models. This involves implementing robust content moderation techniques, such as employing sentiment analysis, hate speech detection, and bias identification algorithms. By leveraging these tools, it is possible to systematically identify and filter out text that may perpetuate harmful stereotypes, offensive language, or biased viewpoints.

**Using LLMs after pre-training**

Fine-tuning:

- Gradient descent on weights to optimize perfomance on one task
- What to fine-tune?
    - Full network
    - Readout heads
    - adapters

Prompting:

- Design special prompt to cue/condition the network into specific mode to solve any tasks
    - No parameter change.one model to rule them all.

**Project**
**Goal of the project:**

- The goal of the project is to create a chatbot that answers questions about the NLP and LLM 2024/2025 course.
- Furthermore, the chatbot must answer ONLY questions related to the course, recognizing out-of-context questions and replying that it is not enabled to provide answers on topics outside the context of interest.
- The groups will deliver the code of the chatbot and a report in which they describe in detail their solution.

**Tools to use for the project**

Students are free to use any tool or technology analyzed in the course, both the more recent ones associated with LLMs and the more classic ones of NLP. To achieve the required result, the students can develop parts of the chatbot with LLMs and other parts with more classic tools, with total freedom of choice as long as the various solutions identified are justified in the final report. Any LLM or other already available model can be used with or without modifications, as long as the groups are fully familiar with the tools they used and are able to answer questions about every aspect of the code and models.

**Chatbot evaluation procedure**

Before the project discussion, the course instructors will question the chatbot of the group in real time with a predefined set of questions on the course and evaluate the answers from the following points of view:

- Relevance:Judges if generated text answers the query.
- Fluency: Evaluates the readability and grammar of the output.
- Coherence: Checks logical flow and consistency.

Then, the course instructors will do another set of pre-defined questions to verify:

- Robustness: Evaluates resistance to adversarial misleading prompts (e.g. are you sure?).
- Precision: Evaluates the capability to recognize out-of-context questions (e.g. who is the king of Spain?)

The course instructors will give a grade to the performance according to these aspects.