NLP Is All You Need

Lenny Pelhate

April 5, 2025

Contents

1 Introduction

Natural Language Processing (NLP) is an interdisciplinary field combining linguistics, computer science, and artificial intelligence. It focuses on enabling computers to process and analyze large amounts of natural language data.

1.1 Historical Development

- **Rule-based Era (1950s-1980s)**: Systems like SHRDLU and ELIZA used hand-crafted rules for language understanding.
- **Statistical Era (1990s-2010s)**: Introduction of probabilistic models and machine learning techniques.
- **Neural Era (2010s-Present)**: Deep learning revolution starting with word embeddings, followed by RNNs, and now dominated by Transformer architectures.
- **Foundation Model Era (2020s-Present)**: The rise of large-scale, multi-modal models trained on vast amounts of data, with emergent capabilities.

1.2 Applications of NLP

- **Machine Translation**
  - Neural Machine Translation (NMT) systems like Google Translate, DeepL
  - Domain-specific translation for legal, medical, and technical documents
  - Real-time speech-to-speech translation services
- **Speech Recognition and Generation**
  - Virtual assistants (Siri, Alexa, Google Assistant)
  - Transcription services for meetings, interviews, and podcasts
  - Voice-operated systems for hands-free operation
  - Text-to-speech systems for accessibility
- **Conversational AI**
  - Advanced chatbots (ChatGPT, Claude, Bard)
  - Customer support automation
  - Digital assistants for productivity
  - AI therapists and mental health support
- **Content Analysis**
  - Sentiment analysis for brand monitoring and market research
  - Content moderation for social media platforms
  - Toxicity detection in online conversations
  - Political bias detection in news articles

- **Information Retrieval**
  - Semantic search engines
  - Question-answering systems
  - Document classification and clustering
  - Recommendation systems based on text analysis
- **Business Applications**
  - Automated resume screening and job matching
  - Contract analysis and legal document processing
  - Automated report generation from data
  - Competitive intelligence gathering
- **Healthcare Applications**
  - Medical record analysis and summarization
  - Clinical trial matching
  - Automated medical coding and billing
  - Disease risk prediction from clinical notes

2 Challenges in NLP

Natural language processing faces several difficulties that stem from the complexity and ambiguity of human language. Some key challenges include:

- **Ambiguity**: Words and sentences can have multiple interpretations.
  - **Word Sense Ambiguity**: "bank" (financial institution) vs. "bank" (river-bank).
  - **Part-of-Speech Ambiguity**: "chair" (noun - furniture) vs. "chair" (verb - to lead a meeting).
  - **Syntactic Ambiguity**: "I saw a man with a telescope." (Did I use the telescope, or did the man have one?)
  - **Multiple Ambiguities**: "I made her duck." (Did I cook a duck for her, or did I make her physically duck?)
  - **Reference Ambiguity**: "John dropped the goblet onto the glass table and it broke." (What is "it" referencing to?)
  - **Discourse Ambiguity**: "The meeting is cancelled. Nicholas isn't coming to the office today." (Does Nicholas's absence relate to the meeting cancellation, or is it unrelated?)
- **Sparsity**: Language data exhibits extreme sparsity problems.
  - **Data Sparsity**: Many valid word combinations appear rarely or never in training data.

  - **Long-Tail Distribution**: Language follows Zipf's Law, with a small number of frequent patterns and a very long tail of rare phenomena.
    * **Implications of Zipf's Law**: Zipf's Law states that word frequency follows an inverse power distribution, meaning a few words are extremely common, while most words are rare. Since Zipf's Law states that a small number of words appear very frequently (like "the," "is," "and"), these words do not provide much meaning on their own. Less frequent words (like "quantum," "neural," "volcano") are often more informative because they carry specific meanings and differentiate one document from another.
  - **Out-of-Vocabulary Words**: Models struggle with unseen words, especially in specialized domains.
  - **Rare Constructions**: Many grammatically valid constructions may have no examples in training data.

- **Corpus Variation**: Language varies across domains, demographics, and styles.
  - NLP models trained on one dataset may struggle to generalize to another.
  - Variations in dialects, slang, and jargon require adaptability in language models.
  - Formal vs. informal language presents challenges in understanding and processing.

- **Expressivity**: Human language is rich in nuances, tone, and style.
  - Sarcasm, humor, and figurative language are difficult for models to detect.
  - Different speakers express the same idea in varied ways.
  - Implicit meaning often requires contextual and cultural understanding.

- **Common Sense**: Machines lack innate world knowledge that humans take for granted.
  - "She put an apple on the table. Later, she ate it." (What did she eat?)
  - Understanding cause-and-effect relationships is challenging for NLP models.
  - Common sense reasoning is needed for resolving ambiguities and making inferences.

3 Fundamental Concepts

3.1 Text Preprocessing

Text preprocessing is crucial for preparing raw text data for NLP tasks:

- **Tokenization**: Breaking text into words, phrases, symbols, or other meaningful elements.

Example: Tokenization

```
Text: "I love NLP! It's fascinating."
Tokens: ["I", "love", "NLP", "!", "It", "'s", "fascinating", "."]
```

- **Stemming**: Reducing words to their word stem.

Example: Stemming with Porter Stemmer

```
Words: ["running", "runs", "ran", "runner"]
Stems: ["run", "run", "ran", "runner"]
```

- **Lemmatization**: Reducing words to their base or dictionary form.

Example: Lemmatization

```
Words: ["running", "runs", "ran", "runner"]
Lemmas: ["run", "run", "run", "runner"]
```

- **Stop Word Removal**: Eliminating common words that add little meaning.

Example: Stop Word Removal

```
Text: "The cat is on the mat"
After removal: ["cat", "mat"]
```

- **Normalization**: Converting text to a standard form.
  - Case normalization (lowercasing)
  - Accent removal
  - Unicode normalization
- **Noise Removal**: Cleaning text of irrelevant information.
  - HTML tag removal
  - Special character handling
  - URL and email address normalization

Python Code Example: Basic Text Preprocessing

```
import nltk
from nltk.tokenize import word_tokenize
from nltk.corpus import stopwords
from nltk.stem import PorterStemmer, WordNetLemmatizer
import re

def preprocess_text(text):
    # Lowercase
    text = text.lower()

    # Remove special characters and digits
    text = re.sub(r'[^\w\s]', '', text)
    text = re.sub(r'\d+', '', text)

    # Tokenize
    tokens = word_tokenize(text)

    # Remove stopwords
    stop_words = set(stopwords.words('english'))
    filtered_tokens = [token for token in tokens if token not in
   stop_words]

    # Stemming
    stemmer = PorterStemmer()
    stemmed_tokens = [stemmer.stem(token) for token in
   filtered_tokens]

    # Lemmatization
    lemmatizer = WordNetLemmatizer()
    lemmatized_tokens = [lemmatizer.lemmatize(token) for token in
   filtered_tokens]

    return {
        'original': text,
        'tokens': tokens,
        'filtered': filtered_tokens,
        'stemmed': stemmed_tokens,
        'lemmatized': lemmatized_tokens
    }

# Example usage
text = "The runners are running a marathon on Monday."
result = preprocess_text(text)
print(result)
```

3.2 Text/Document Representation

Feature Extraction: Mapping from textual data to real valued vectors. Common methods include:

- Bag of Words
- Bag of N-Grams

- TF-IDF
- Word Embeddings (continuous vectorial representations)

3.2.1 Bag of Words (BoW)

Represents a document by the occurrence counts (or the frequency) of each word, as an unordered collection of words, disregarding grammar and word order.

- **Advantages**: Simple, efficient, and effective for tasks like topic modeling.
- **Drawbacks**: Loses word order and phrase meaning

Example: Bag of Words

```
Documents:
  D1: "I love NLP"
  D2: "I love programming"

Vocabulary: {"I", "love", "NLP", "programming"}

BoW Representation:
  D1: [1, 1, 1, 0]
  D2: [1, 1, 0, 1]
```

3.2.2 Bag of N-Grams

Represents a document using contiguous sequences of N words (n-grams) as features, preserving some local word order while still disregarding overall grammar and sentence structure.

- **Advantages**: Partial word order and local context (short dependencies between words) are retained, improving understanding of phrases.
- **Drawbacks**:
  - Computationally expensive due to increased feature space (significantly as N grows).
  - The ordering of Ngrams is not considered.
  - The more you increase the Ngram size, the more the data become sparce.

Example: Bag of Bigrams (N

```
Documents:
  D1: "I love NLP"
  D2: "I love programming"

Vocabulary (Bigrams): {"I love", "love NLP", "love programming"}

Bag of Bigrams Representation:
  D1: [1, 1, 0]
  D2: [1, 0, 1]
```

3.2.3 TF-IDF (Term Frequency-Inverse Document Frequency)

TF-IDF tackles a main limitation of BoW: highly frequent words tend to dominate in the document but may not contain as much "informational content" to the model as rarer but perhaps domain specific words. Thus TF-IDF weighs terms based on their frequency in a document and their rarity across all documents.

- **Term Frequency (TF)**: Measures how often a word appears in a specific document.
- **Inverse Document Frequency (IDF)**: Measures how rare a word is across all documents, giving higher weight to less common words.

Example: TF-IDF

```
TF(t,d) = (Number of times term t appears in document d) /
          (Total number of terms in document d)

IDF(t) = log(Total number of documents /
             Number of documents containing term t)

TF-IDF(t,d) = TF(t,d) * IDF(t)
```

Python Code Example: TF-IDF Implementation

```python
from sklearn.feature_extraction.text import TfidfVectorizer

# Sample documents
documents = [
    "I love natural language processing",
    "I love programming in Python",
    "Natural language processing is fascinating"
]

# Create a TF-IDF Vectorizer
vectorizer = TfidfVectorizer()

# Transform documents to TF-IDF features
tfidf_matrix = vectorizer.fit_transform(documents)

# Get feature names
feature_names = vectorizer.get_feature_names_out()

# Print the TF-IDF features for each document
for i, doc in enumerate(documents):
    print(f"Document {i+1}:")
    for j, feature in enumerate(feature_names):
        score = tfidf_matrix[i, j]
        if score > 0:
            print(f"  {feature}: {score:.4f}")
```

BoW & TF-IDF for Query Mapping:

- **Cosine Similarity Between Query and Document**:
  - Cosine similarity is a measure of similarity between two vectors, commonly used in information retrieval and text analysis. It determines the cosine of the angle between two vectors in a multi-dimensional space.
  - The cosine similarity between a query vector q and a document vector d is given by:
$$\text{cosine\_similarity}(q, d) = \frac{q \cdot d}{\|q\|\|d\|}$$
  - The value of cosine similarity ranges from -1 to 1.
    - If the value is close to 1, the query and document are very similar.
    - If the value is 0, they are completely unrelated.
    - If the value is close to -1, they are opposite in direction.

3.3 Word Representation

3.3.1 One-Hot Encoding

One-hot encoding is a method used to represent words as numerical vectors in natural language processing.

- **Vector Size**: The size of the dictionary (vocabulary).
- **Index Assignment**: Each word is assigned a unique integer index.
- **Binary Representation**: The vector contains all zeros except for a single one at the index corresponding to the word.

Example: One-Hot Encoding

```
"the"  →  (1, 0, 0, 0, ...)
"cat"  →  (0, 1, 0, 0, ...)
```

Problems with One-Hot Encoding:

- **Similarity Issue**: Words like `"cat"` and `"tiger"` have completely different one-hot vectors, though they are semantically similar.
- **Vocabulary Size**: The dictionary size can be very large, making storage and computation inefficient.
- **Sparsity**: Many machine learning models struggle with high-dimensional and sparse data.

One-hot encoding is simple but has limitations, which is why more advanced techniques like word embeddings (e.g., Word2Vec, GloVe) are often preferred.

3.3.2 Word Embeddings

Word embeddings are dense vector representations of words that capture semantic relationships in a continuous vector space. These representations enable measurement of semantic similarity between words and serve as the foundation for many NLP tasks.

Matrix Factorization Methods for Word Embeddings

- **Latent Semantic Analysis (LSA)**: commonly used to generate word embeddings, which represent words in a dense vector space capturing their semantic meanings. Here's how the process works:

  1. **Construct a Co-occurrence Matrix**: This matrix represents the frequency of words appearing together within a given context window. Each row and column corresponds to a word in the vocabulary, and the matrix entries store the number of times a pair of words co-occurs within a specified window of text.

$$\text{Co-occurrence Matrix} : M = \begin{bmatrix} \text{the} & 5 & 2 & 3 & 4 \\ \text{cat} & 2 & 5 & 1 & 2 \\ \text{dog} & 3 & 1 & 5 & 3 \\ \text{sat} & 4 & 2 & 3 & 5 \end{bmatrix}$$

  2. **Apply a Singular Value Decomposition (SVD)**: SVD factorizes the original matrix M into three smaller matrices U, Σ, and V^T.

$$M = U\Sigma V^T$$

     - U contains the word vectors (embeddings) for each word in the input space.
     - V^T contains the word vectors in the output space.
     - Σ is a diagonal matrix containing singular values, which indicate the significance of each dimension in the reduced space.

  3. **Word embeddings**: The word embeddings are represented by the rows of the matrices U or V, which capture the latent semantic structure of the language. These embeddings represent the words in a dense vector space, where words with similar meanings will have similar vector representations. This allows us to measure the semantic similarity between words, such as "king" and "queen," which will be closer in the vector space compared to "king" and "dog."

- **GloVe (Global Vectors)**: Combines global matrix factorization with local context window methods.

  - Leverages global word-word co-occurrence statistics from a corpus.
  - Trains on the non-zero elements of a word-word co-occurrence matrix.
  - Optimizes a weighted least squares objective that balances the influence of frequent and rare co-occurrences.

NN Based Word Embeddings Neural networks, particularly **feedforward neural networks (FNNs)** and **recurrent neural networks (RNNs)**, are commonly used for language modeling.

The process involves:

1. Taking a fixed-size context window of previous words.
2. Maps each word to its corresponding dense embedding vector.
3. Processes these embeddings through neural network layers (feedforward or recurrent).
4. Produces a probability distribution over the vocabulary for the next word prediction.

- **Softmax Classifier**: A probability-based classification model often used in NLP tasks.
  - Computes the probability distribution over possible output classes (e.g., words in a vocabulary).
  - Commonly used as the final activation layer in neural networks for tasks like next-word prediction and text classification.
  - Uses the softmax function to convert raw scores (logits) into probabilities:

    $$P(w_t = i|w_{t-k}, ..., w_{t-1}) = \frac{e^{z_i}}{\sum_j e^{z_j}}$$

    where z_i represents the score for class i.
  - Problem: Last layer very expensive with big vocabulary
    - Solution:
      - **Softmax-based Approaches**: Hierarchical Softmax, Differentiated Softmax, CNN-Softmax
      - **Sampling-based Approaches**: Remove the softmax layer and, instead, optimise some other loss function that approximates the softmax (cf Word2Vec)
- **Word2Vec**: Uses shallow neural networks to learn word vector representations (Key idea: Predict surrounding words of every word).
  - **CBOW (Continuous Bag of Words)**: Use context words in a window to predict middle word.

    $$P(\text{target word}|\text{context words})$$

  - **Skip-Gram Model**: Use the middle word to predict surrounding ones in a window.

    $$P(\text{context word}|\text{target word})$$

* **Negative Sampling**: Instead of computing a score for every word in the dictionary, sample some negative target words.
* Vanilla skip-gram complexity is $O(V)$, negative sampling's is $O(k+1)$.

Negative Sampling Example

Training sentence:

...	the	cat	sat	on	the	mat	...
	c1	c2	t	c3	c4		

Positive examples (+)

t	c
sat	the
sat	cat
sat	on
sat	the

Negative examples (-)

t	c	t	c
sat	oak	sat	sentence
sat	flower	sat	desktop
sat	blue	the	sat
sat	toto	sat	forever

- **FastText**: An extension of Word2Vec developed by Facebook Research.
  - Represents words as bags of character n-grams.
  - Better handles out-of-vocabulary words and morphologically rich languages.
  - Can generate embeddings for previously unseen words based on their subword components.
- **Advantages of NN-based word embeddings**:
  - Continuous vector representations
  - Low dimension (compared to dictionary size)
  - Pre-trained on large unlabeled data

Example: Word Embeddings Semantic Properties

```
Vector("king") - Vector("man") + Vector("woman") = Vector("queen")
Vector("paris") - Vector("france") + Vector("italy") = Vector("rome")
```

Python Code Example: Using Word Embeddings

```
import gensim.downloader as api

# Load pre-trained Word2Vec embeddings
word2vec_model = api.load('word2vec-google-news-300')

# Find most similar words
similar_words = word2vec_model.most_similar('computer', topn=5)
print("Words similar to 'computer':")
for word, score in similar_words:
    print(f"  {word}: {score:.4f}")

# Word analogy
analogy_result = word2vec_model.most_similar(
    positive=['woman', 'king'],
    negative=['man'],
    topn=1
)
print("\nWord analogy: man:king :: woman:?")
print(f"  {analogy_result[0][0]}: {analogy_result[0][1]:.4f}")
```

3.3.3 Contextual Embeddings

Word representations that change based on the surrounding context.

- **ELMo (Embeddings from Language Models)**: Bidirectional LSTM-based contextual representations.
- **BERT Embeddings**: Contextual embeddings from the BERT model's encoder layers.
- **GPT Embeddings**: Forward-directional contextual embeddings from GPT models.

Example: Contextual Embedding for Polysemous Words

```
Sentence 1: "I need to go to the bank to deposit money."
Embedding of "bank" → Financial institution vector

Sentence 2: "I sat by the river bank and watched the sunset."
Embedding of "bank" → River shore vector
```

4 NLP Levels of Analysis

4.1 Morphological Analysis

Morphological analysis is the process of analyzing the structure of words and their components, including stems, prefixes, suffixes, and inflectional endings. It helps in understanding the formation and function of words in a given language.

- **Inflectional Morphology**: Deals with grammatical variations of a word (e.g., "run" → "running").

- **Derivational Morphology**: Focuses on creating new words by adding prefixes or suffixes (e.g., "happy" → "happiness").
- **Lemmatization**: Reducing words to their base form (e.g., "running" → "run").
- **Stemming**: Cutting off affixes to obtain the root form of a word (e.g., "playing" → "play").

4.2 Part-of-Speech (POS) Tagging

POS tagging assigns grammatical categories (noun, verb, adjective) to words in a sentence.

Example: POS Tagging

```
Sentence: "The cat sat on the mat."

Tagged:
  The (DET)
  cat (NOUN)
  sat (VERB)
  on (ADP)
  the (DET)
  mat (NOUN)
  . (PUNCT)
```

Common POS Tag Sets

Penn Treebank Tags	Universal POS Tags
NN (Noun, singular)	NOUN
NNS (Noun, plural)	NOUN
VB (Verb, base form)	VERB
VBD (Verb, past tense)	VERB
JJ (Adjective)	ADJ
RB (Adverb)	ADV
IN (Preposition)	ADP
DT (Determiner)	DET

Python Code Example: POS Tagging

```
import spacy

# Load English language model
nlp = spacy.load("en_core_web_sm")

# Example text
text = "The quick brown fox jumps over the lazy dog."

# Process text
doc = nlp(text)

# Print tokens and their POS tags
for token in doc:
    print(f"{token.text:<10} {token.pos_:<8} {token.tag_:<8} {
   spacy.explain(token.tag_)}")
```

4.3 Syntactic Parsing

Syntactic parsing analyzes the grammatical structure of a sentence.

- There are two common types:
  - **Constituency Parsing**
  - **Dependency Parsing**

Importance of Sentence Structure: Understanding sentence structure is crucial for correct language interpretation. It helps disambiguate sentences and enables humans to convey complex ideas by combining words into meaningful units. Analyzing sentence structure aids in understanding complex sentences by identifying relationships between components. This is essential in applications such as relation extraction systems.

4.3.1 Constituency Parsing

- **Constituency Parsing**:
  - Breaks the sentence into hierarchical phrase structures (i.e. **tree structure**).
  - Constituents are groups of words that can act as single units (with respect to their internal structure or with respect to other units in the language).
  - **Phrase-level categories**:
    - **NP** (noun phrase)
    - **VP** (verb phrase)
    - **PP** (prepositional phrase)
    - **AP** (adjective phrase)
  - **Lexical categories (POS)**:
    - **NN** (noun)
    - **NNS** (noun, plural)

    - * **NNP** (proper noun, singular)
    - * **NNPS** (proper noun, plural)
    - * **VB** (verb, base form)
    - * **VBP** (verb, non-3rd person singular present)
    - * **VBZ** (verb, 3rd person singular present)
  - **Functional categories**:
    - * **DT** (determiner)
    - * **PRP** (personal Pronoun)
    - * **PRP$** (possessive Pronoun)
    - * **IN** (preposition/subordinating conjunction)
    - * **CC** (coordinating conjunction)

Phrase structure organizes words into contiguous nested constituents:

- Leafs are unit words (or tokens)
- Words are combined into phrases
- Phrases can combine into bigger phrases

Example: Constituency Parse Tree

```
Sentence: "The cat sat on the mat."

Parse Tree:
                S
              /   \
             /     \
           NP       VP
          / \      /  \
         DT NN    V    PP
         |   |    |   /  \
        The cat sat  P   NP
                     |   / \
                     on DT NN
                        |   |
                       the mat
```

- **Context-Free Grammar (CFG)**: formal grammar that consists of the following components:
  - **Terminals** ($T = \{w_1, ..., w_n\}$): basic symbols from which strings are formed. Terminals are symbols that do not get replaced during derivation.
  - **Nonterminals** ($N = \{N_1, ..., N_n\}$): symbols that can be replaced with other nonterminals or terminals. Nonterminals represent syntactic categories like noun phrases (NP) or verb phrases (VP).
  - **Start symbol** (N_1): special nonterminal symbol from which the derivation process begins.

- **Rules** (R): define how nonterminals can be expanded into sequences of terminals and/or nonterminals. A production rule has the form:

$$N_i \to \zeta_j$$

where $N_i \in N$ is a nonterminal and ζ_j is a sequence of terminals and/or nonterminals.

- **Probabilistic Context-Free Grammar (PCFG)**:
  - Same as CFG + **Rule probabilities**:

$$\forall i, \sum_j P(N_i \to \zeta_j) = 1$$

- **Chomsky Normal Form (CNF) Grammars**:
  - Most dynamic programming algorithms for CFGs and PCFGs requires grammars to be CNF grammars
  - A grammar is in CNF if it only has rules of the form:

$$N_i \to N_j N_k$$

$$N_i \to w_j$$

  - Any CFG can be represented by a equivalent CFG in Chomsky Normal Form:
    - **Right-branching binarization**: replace a rule with more than 2 symbols on the right with several binary rules that accomplish the same thing.

Cf CM 5 for examples

- **Prepositional Phrase (PP) attachment ambiguity**: occurs when it is unclear which part of a sentence a prepositional phrase modifies.
  - Number of parses due to PP attachment ambiguity:

$$\textbf{Catalan number}: C_n = \frac{(2n)!}{(n+1)!n!}$$

- **Parsing**: involves generating and testing many hypotheses, with considerable overlap. Once we've build some good partial parse, we might want to re-use it for other hypotheses.
- **CYK (Cocke-Younger-Kasami) Algorithm**: dynamic programming algorithm used for parsing strings in CFGs. It determines whether a given string belongs to a language defined by a CFG in CNF.
  1. Construct a triangular table where $T[i, j]$ contains the set of nonterminals that can generate the substring $w_i \dots w_{i+j-1}$.
     - Rows represent different levels of phrase combinations.
     - Columns represent the words in the sentence.

2. *Initialization*: For each terminal w_i, find all nonterminals A such that $A \to w_i$ exists in the grammar and store them in $T[i, 1]$.
3. *Filling the Table*: For increasing lengths j (from 2 to n), compute $T[i, j]$ by checking all possible partitions:

$$A \to BC$$

   If $B \in T[i, k]$ and $C \in T[i + k, j - k]$, then add A to $T[i, j]$.
4. *Final Check*: If the start symbol S is in $T[1, n]$, the string is in the language; otherwise, it is not.

Example: CYK Algorithm

Consider the following PCFG in CNF:

	John	**sees**	**the**	**dog**
John	NP (0.4)	S (0.4·1.0·0.8)	...	...
sees		V (1.0)	...	VP (1.0·1.0·0.7)
the			Det (1.0)	NP (1.0·1.0·0.6)
dog				N (1.0)

Grammar Rules	
S → NP V	(0.8)
NP → Det N	(0.6)
VP → V NP	(0.7)
S → NP VP	(1.0)

Final Check: Since S appears in the final cell with probability 1.0, the sentence is valid in the language.

- **Treebanks**: corpora in which each sentence has been paired with a parse tree
- **Evaluation of Constituency Parsing**: compare **gold standard tree** to **parser output**.
  - Output constituent is counted correct if there is a gold constituent that spans the same sentence positions.
  - **Precision**:
$$\frac{\#\,\text{correct constituents}}{\#\,\text{in parser output}}$$
  - **Recall**:
$$\frac{\#\,\text{correct constituents}}{\#\,\text{in gold standard}}$$
  - **F-score**:
$$\frac{2\,\text{precision} \cdot \text{recall}}{\text{precision} + \text{recall}}$$
- **Limitation of vanilla PCFGs**: Replacing one word with another with the same POS will never result in a different parsing decision, even though it should.
  - Solution: **Lexicalized PCFG**:
    - Create new categories, this time by adding the lexical head of the phrase (**head-word**)
- **Dilemma: More Features vs. Sparcity**
  - category-splitting makes the grammar much more specific (good)
  - leads to huge grammar blowup and very sparse data (bad)
  - *Solution*: balance these two through smoothing

4.3.2 Dependency Parsing

- **Dependency Parsing**:
  - Focuses on direct/binary relationships between individual words.
  - Follows a **graph structure**, that is usually a tree (connected, acyclic, single-head)
  - In the graph, the arrows are commonly typed with the name of grammatical relations.
  - The arrow connects a head (governor, superior, regent) with a dependent (modifier, inferior, subordinate).

Example: Dependency Parse

```
Sentence: "The cat sat on the mat."

Dependencies:
  sat → cat (nsubj)
  sat → on (prep)
  cat → The (det)
  on  → mat (pobj)
  mat → the (det)
```

- **Arc-standard transition-based dependency parser**:
  - Cf CM slide 41/66
- **Neural dependency parsing**:
  - Cf CM slide 46/66
- **Graph-based dependency parsing**
- **Recursive neural networks for parsing**
- **Tree-LSTM**

4.3.3 Constituency Parsing vs. Dependency Parsing

- Dependency structure explicitly represents:
  - head-dependent relations (directed arc)
  - functional categories (arc labels)
- Constituency structure explicitly represents:
  - phrases (non-terminal nodes)
  - structural categories (non-terminal labels)
  - possibly some functional categories (grammatical functions, e.g. PP-LOC)
- Dependencies are better for free word order languages.

- It is possible to convert dependencies to constituencies and vice versa with some effort.

Python Code Example: Dependency Parsing with SpaCy

```
import spacy
from spacy import displacy

# Load English language model
nlp = spacy.load("en_core_web_sm")

# Example text
text = "The cat sat on the mat."

# Process text
doc = nlp(text)

# Print dependencies
for token in doc:
    print(f"{token.text:<10} {token.dep_:<10} {token.head.text}")

# Visualize dependencies (would render in Jupyter or save as HTML)
# displacy.serve(doc, style="dep")

# Extract subject-verb-object relationships
for token in doc:
    # Find verbs
    if token.pos_ == "VERB":
        subject = None
        direct_object = None

        # Find subject and direct object
        for child in token.children:
            if child.dep_ == "nsubj":
                subject = child
            elif child.dep_ == "dobj":
                direct_object = child

        if subject:
            print(f"Found action: {subject.text} {token.text}",
   end=" ")
            if direct_object:
                print(f"{direct_object.text}")
            else:
                print()
```

4.4 Semantic Parsing

4.4.1 Named Entity Recognition (NER)

NER identifies and categorizes proper names in a text, such as persons, locations, and organizations.

Example: Named Entity Recognition

```
Text: "Apple Inc. was founded by Steve Jobs in California."

Entities:
  Apple Inc. (ORG)
  Steve Jobs (PERSON)
  California (GPE)
```

Common NER Categories

Entity Type	**Examples**
PERSON	People, characters
ORG	Companies, agencies, institutions
GPE/LOC	Countries, cities, locations
DATE	Dates, time periods
MONEY	Monetary values
PERCENT	Percentage values
PRODUCT	Products, objects
EVENT	Named events (wars, sports events)
WORK_OF_ART	Titles of books, songs, etc.
LAW	Named laws and regulations

Python Code Example: Named Entity Recognition

```
import spacy
from spacy import displacy

# Load English language model
nlp = spacy.load("en_core_web_sm")

# Example text
text = """
Apple Inc. is planning to open a new research center in Zurich,
    Switzerland next year.
The project will cost approximately $300 million and create 500
    new jobs.
CEO Tim Cook announced the plan during his visit to Europe last
    week.
"""

# Process text
doc = nlp(text)

# Print entities
for ent in doc.ents:
    print(f"{ent.text:<20} {ent.label_:<10} {spacy.explain(ent.
    label_)}")

# Extract specific entity types
organizations = [ent.text for ent in doc.ents if ent.label_ == "
    ORG"]
people = [ent.text for ent in doc.ents if ent.label_ == "PERSON"]
locations = [ent.text for ent in doc.ents if ent.label_ in ["GPE",
    "LOC"]]

print(f"\nOrganizations: {organizations}")
print(f"People: {people}")
print(f"Locations: {locations}")

# Visualize entities (would render in Jupyter or save as HTML)
# displacy.serve(doc, style="ent")
```

4.4.2 Semantic Role Labeling (SRL)

SRL determines roles within a sentence (who did what to whom).

Example: Semantic Role Labeling

```
Sentence: "John gave Mary a book yesterday at the library."

Semantic Roles:
  Predicate: gave
  Agent (Who): John
  Recipient (To whom): Mary
  Theme (What): a book
  Time (When): yesterday
  Location (Where): at the library
```

Python Code Example: SRL with AllenNLP

```
from allennlp.predictors.predictor import Predictor

# Load the SRL model
predictor = Predictor.from_path("https://storage.googleapis.com/
    allennlp-public-models/structured-prediction-srl-bert
    .2020.12.15.tar.gz")

# Example sentence
sentence = "John gave Mary a book yesterday at the library."

# Get SRL predictions
predictions = predictor.predict(sentence=sentence)

# Print the output
words = predictions["words"]
verbs = predictions["verbs"]

print("Words:", words)
print("\nSemantic Roles:")

for verb_info in verbs:
    print(f"\nPredicate: {verb_info['verb']}")

    # Group words by their tags
    current_tag = None
    current_phrase = []
    roles = {}

    for word, tag in zip(words, verb_info["tags"]):
        # Start of a new tag or continuation of current
        if tag.startswith("B-"):
            # Store the previous phrase if it exists
            if current_tag and current_phrase:
                role_name = current_tag[2:]  # Remove "B-" or "I-"
                roles[role_name] = " ".join(current_phrase)
                current_phrase = []

            current_tag = tag
            current_phrase = [word]
        elif tag.startswith("I-"):
            if current_phrase:  # Ensure there's a phrase to
   continue
                current_phrase.append(word)
        elif tag == "O":
            if current_tag and current_phrase:
                role_name = current_tag[2:]
                roles[role_name] = " ".join(current_phrase)
                current_phrase = []
                current_tag = None

    # Handle any remaining phrase
    if current_tag and current_phrase:
        role_name = current_tag[2:]
        roles[role_name] = " ".join(current_phrase)

    # Print all roles
    for role, phrase in roles.items():
        print(f"  {role}: {phrase}")
```

4.4.3 Word Sense Disambiguation (WSD)

WSD identifies the correct meaning of a word based on context.

Example: Word Sense Disambiguation

```
Sentence 1: "I went to the bank to deposit money."
  → bank: financial institution

Sentence 2: "I sat on the river bank and watched the sunset."
  → bank: side of a river
```

Python Code Example: WSD with NLTK and WordNet

```python
from nltk.wsd import lesk
from nltk.corpus import wordnet as wn
from nltk import word_tokenize

# Example sentences
sentence1 = "I went to the bank to deposit money."
sentence2 = "I sat on the river bank and watched the sunset."

# Tokenize sentences
tokens1 = word_tokenize(sentence1)
tokens2 = word_tokenize(sentence2)

# Find sense for 'bank' in both sentences
bank1_sense = lesk(tokens1, 'bank', 'n')
bank2_sense = lesk(tokens2, 'bank', 'n')

# Print results
print("Sense 1:", bank1_sense)
print("Definition:", bank1_sense.definition() if bank1_sense else
    "Not found")
print()
print("Sense 2:", bank2_sense)
print("Definition:", bank2_sense.definition() if bank2_sense else
    "Not found")

# Print all possible senses of 'bank'
print("\nAll senses of 'bank':")
for i, synset in enumerate(wn.synsets('bank')):
    print(f"{i+1}. {synset}: {synset.definition()}")
    # Print examples if available
    if synset.examples():
        print(f"   Examples: {', '.join(synset.examples())}")
```

4.5 Discourse Parsing

4.5.1 Coreference Resolution

Coreference resolution determines when different words refer to the same entity.

Example: Coreference Resolution

```
Text: "John said he would come, but he was late because his car broke down."

Coreference clusters:
  [John, he, he, his] → Referring to the same person
  [car] → Referring to John's car
```

Python Code Example: Coreference Resolution with NeuralCoref

```python
import spacy
import neuralcoref  # Note: This library requires specific
    installation

# Load spacy model
nlp = spacy.load('en_core_web_sm')

# Add neural coref to pipeline
neuralcoref.add_to_pipe(nlp)

# Example text
text = "John said he would come to the party. He was excited about
     it."

# Process text
doc = nlp(text)

# Print coreference clusters
print("Coreference clusters:")
for cluster in doc._.coref_clusters:
    print(f"Cluster: {cluster.main.text}")
    for mention in cluster.mentions:
        print(f"  - {mention.text} (at position {mention.start})")

# Resolve coreferences
resolved_text = doc._.coref_resolved
print(f"\nOriginal: {text}")
print(f"Resolved: {resolved_text}")
```

5 NLP Tasks 1 - Language Modeling

5.1 Language Modeling

Language models predict the probability of a sequence of words given a window of text (context).

Language model: model that computes either of these, $P(w_1, w_2, w_3, w_4)$ or $P(w_4 \mid w_1, w_2, w_3)$ (which is equivalent).

Language models can be probabilistic (e.g., n-grams) or neural-based (e.g., Transformers).

5.1.1 N-gram Models

Predict words based on the preceding $n-1$ words or compute joint probability by using the **chain rule of probability**:

$$P(w_1, w_2, ..., w_n) = \prod_i P(w_i \mid w_1, ..., w_{i-1})$$

Example: N-gram Model

```
Bigram (n=2) probability calculation:
P("Natural language") = P("language"|"Natural")

Trigram (n=3) probability calculation:
P("Natural language processing") =
P("processing"|"Natural language")P("language"|"Natural")P("Natural")
```

How to compute these probabilities? There are far too many possible sentences (problem of data sparcity), thus: **Markov Assumption**

$$P(w_i \mid w_1, w_2, ..., w_{i-1}) \approx P(w_i \mid w_{i-k}, ..., w_{i-1})$$

- **Unigram model**: $P(w_i \mid w_1, w_2, ..., w_n) = \prod_i P(w_i)$
- **Bigram model**: $P(w_i \mid w_1, w_2, ..., w_n) = \prod_i P(w_i \mid w_{i-1}) = \frac{count(w_{i-1}, w_i)}{count(w_{i-1})}$ **(MLE)**

Problems with n-gram models:

- Unknown words
- Curse of dimensionality: the more n is large, the more the space is sparce.
- Zero probabilities: If the probability of any word in the test set is 0 then the entire probability is 0. Since the perplexity is base on the inverse probability, we can't compute the perplexity.

Solution:

- Smoothing
- Learn distributed representations of words (cf word embeddings)

5.1.2 Probabilistic Neural Language Models

- **Automatic Smoothing**: If the word embedding of feline and cat are similar, the neural network will be able to generalize for feline

5.1.3 Neural Language Models

Use deep learning architectures.

- RNN/LSTM/GRU-based models
- Transformer-based models (GPT, BERT, T5)

- Large Language Models (LLMs)

Python Code Example: Simple N-gram Model

```
import nltk
from nltk.util import ngrams
from collections import Counter, defaultdict

def train_ngram_model(text, n=2):
    # Tokenize text
    tokens = nltk.word_tokenize(text.lower())

    # Add start and end tokens
    padded_tokens = ['<s>'] * (n-1) + tokens + ['</s>']

    # Generate n-grams
    n_grams = list(ngrams(padded_tokens, n))

    # Count frequencies
    ngram_counts = Counter(n_grams)
    context_counts = defaultdict(int)

    for ngram in n_grams:
        context = ngram[:-1]
        context_counts[context] += 1

    # Calculate probabilities
    model = {}
    for ngram, count in ngram_counts.items():
        context = ngram[:-1]
        word = ngram[-1]

        if context in context_counts:
            model[ngram] = count / context_counts[context]

    return model

# Example usage
text = "I love natural language processing. Natural language
   processing is fascinating."
model = train_ngram_model(text, n=2)

# Print some probabilities
for ngram, prob in sorted(model.items())[:5]:
    print(f"P({ngram[1]} | {ngram[0]}) = {prob:.4f}")
```

Example: Using a Pretrained Language Model

```
from transformers import pipeline

# Initialize text generation pipeline with GPT-2
generator = pipeline('text-generation', model='gpt2')

# Generate text
prompt = "Natural language processing is"
generated_text = generator(prompt, max_length=50,
    num_return_sequences=1)

print(f"Prompt: {prompt}")
print(f"Generated: {generated_text[0]['generated_text']}")
```

5.2 Evaluating Language Models

5.2.1 Train/Test Sets + Evaluation Metric

- Data is typically split into training, validation, and test sets.
- Common evaluation metrics include accuracy, precision, recall, F1-score, and BLEU (for machine translation).

5.2.2 Extrinsic Evaluation

- **Shannon Game**: A test where humans guess the next word in a sequence, evaluating how well a model predicts human language patterns.
- **Perplexity**: Perplexity is the inverse probability of the test set, normalized by the number of words

$$PP(W) = P(w_1, w_2, ..., w_n)^{-\frac{1}{n}}$$

- **Bilingual Evaluation Understudy (BLEU) Score**: Evaluates the quality of machine-generated text against human reference translations.
- **ROUGE Score**: Commonly used in text summarization, measures the overlap of n-grams between the generated and reference texts.
- **Word Error Rate (WER)**: Used in speech recognition to measure the difference between transcribed text and reference text.
- **Human Evaluation**: Asking human evaluators to rate the fluency, coherence, and relevance of generated text.

5.2.3 Intrinsic Evaluation

- Measures how well a model captures linguistic structures.
- Common methods include perplexity, log-likelihood, and word embedding evaluations (cosine similarity, intrinsic benchmarks).

5.2.4 Noisy Channel Approach

The Noisy Channel Approach is a probabilistic framework used in both Automatic Speech Recognition (ASR) and Machine Translation (MT). It treats the problem as recovering the most likely intended message from a distorted version of it.

- **Noisy Channel Model for ASR (Automatic Speech Recognition)**: In ASR, the goal is to find the most likely word sequence W given the observed acoustic signal A:

  $$W^* = \arg\max_W P(W \mid A)$$

  Using Bayes' Rule:

  $$P(W \mid A) = \frac{P(A \mid W)P(W)}{P(A)}$$

  Since $P(A)$ is constant for all word sequences, we simplify to:

  $$W^* = \arg\max_W P(A \mid W)P(W)$$

  - $P(A \mid W)$ **(Acoustic Model)**: Probability of the acoustic signal given a word sequence.
  - $P(W)$ **(Language Model)**: Probability of the word sequence in natural language.

  The **acoustic model** interprets the speech signal, while the **language model** ensures grammatical correctness.

- **Noisy Channel Model for SMT (Statistical Machine Translation)**: In SMT, the goal is to find the most likely translation T in the target language given a source sentence S from another language:

  $$T^* = \arg\max_T P(T \mid S)$$

  Using Bayes' Rule:

  $$P(T \mid S) = \frac{P(S \mid T)P(T)}{P(S)}$$

  Since $P(S)$ is constant, we simplify to:

  $$T^* = \arg\max_T P(S \mid T)P(T)$$

  - $P(S \mid T)$ **(Translation Model)**: Probability of generating the source sentence given a target translation.
  - $P(T)$ **(Language Model)**: Probability of the target sentence being fluent and natural.

  The **translation model** ensures semantic **fidelity** (accuracy), while the **language model** ensures **fluency**.

  Learning the Translation Model:

  - **Parallel data:** e.g. we need a lot of pairs of human-translated French/English sentences.

- **Spurious Words:** extra words in the target sentence that don't align with any word in the source sentence.
- **Fertility:** number of target words that a source word generates during translation (zero fertility words → no translated words).
- **Alignment:** word-level correspondence between sentences of the respective languages.
  - **One-to-Many Alignment:**
    - "hairdryer" → "sèche" + "cheveux"
  - **Many-to-One Alignment:**
    - "aboriginal" + "people" → "autochtones"
  - **Many-to-Many Alignement:**
    - "don't" + "have" + "any" + "money" → "sont" + "démunis"
  - **Phrase Alignment:**
    - "I am looking forward to" → "Je me réjouis de"

  Alignments are latent variables a:

  $$P(S \mid T) \text{ becomes } P(S, a \mid T)$$

  Example: **IBM Model 1**
  - Generative model: break up translation process into smaller steps
  - EM algorithm
  - cf CM6 slides 21-31

  Evaluation of Alignment:
  - **Average Error Rate (AER):**

  $$\text{AER}(S, P, A) = 1 - \frac{|A \cap S| + |A \cap P|}{|A| + |S|}$$

  where:
    - A = set of predicted alignments
    - S = set of sure alignments (correct or gold-standard alignments)
    - P = set of possible alignments (includes both sure and probable alignments, so $S \subseteq P$)

  IBM Models create a many-to-one mapping, whereas Real word alignments have many-to-many mappings.

Shortcomings of SMT:

- Models can grow to be overly complex
- Requires a lot of hand-designed feature engineering
- Difficult to manage out-of-vocabulary words
- Components are independently trained
- Memory-intensive, as decoder relies on huge lookup tables

- **Other Noisy Channel Processes:**

  - **Handwriting recognition**
  - **Optical Character Recognition (OCR)**
  - **Spelling Correction**

6 NLP Tasks 2 - Sequence Tagging

6.1 Sequence Tagging Tasks

Sequence tagging involves assigning a label to each token in a sequence, such as a sentence or a document. Some of the most common sequence tagging tasks include part-of-speech tagging, chunking, named entity recognition, and semantic role labeling.

Word Labeling Tasks:

- In many NLP tasks, it is usefull to augment text data with **syntactic** and **semantic** information.
- We want to asign semantic and/or syntactic labels to each word in a sequence (sentence).

6.1.1 Part-of-Speech Tagging

Part-of-speech (POS) tagging is the process of assigning a grammatical category (e.g., noun, verb, adjective) to each word in a sentence.

POS tagging can be performed using rule-based, statistical, and neural network-based approaches. Modern POS taggers often rely on deep learning models such as recurrent neural networks (RNNs) and transformers to achieve high accuracy.

- **Open vs. Closed Classes**:
  - **Open**: proper nouns, nouns, verbs, adjectives, adverbs
  - **Closed**: determiners, pronouns, prepositions, auxiliaries, usually function words (short common words which play grammatical role)
  - Open classes make rule based methods difficult since they need to manage unknown words

Ambiguity: many words may have ambiguous tag (cf **brown corpus**). Sequence labelling consist in determining the label of a word within a particular sequence of words (the context).

6.1.2 Chunking (Shallow Parsing)

Chunking involves segmenting a sentence into syntactically meaningful chunks (syntactic phrases), such as noun phrases (NP) or verb phrases (VP).

Unlike full syntactic parsing, which generates a complete parse tree, chunking provides a partial structure that is useful for tasks like named entity recognition and information retrieval. Common approaches to chunking include conditional random fields (CRFs) and deep learning models such as long short-term memory (LSTM) networks.

- Segments are identified with **IOBES** encoding:
  - **I**nside, **O**utside, **B**eginning, **E**nd, **S**ingle
  - Words outside of syntactic phrases: **O**
- Common chunking tags:
  - **NP**: Noun Phrase
  - **VP**: Verb Phrase
  - **PP**: Prepositional Phrase

Example of Chunking

He	reckons	the	current	account	deficit
S-NP	S-VP	B-NP	I-NP	I-NP	E-NP

6.1.3 Named Entity Recognition

Named entity recognition (NER) aims to identify and classify entities in text into predefined categories, such as persons, organizations, locations, dates, and more.

NER is widely used in information extraction, question answering, and text summarization. Traditional NER systems use handcrafted rules and feature-based machine learning models, while modern systems leverage deep learning architectures, such as bidirectional LSTMs with CRFs and transformer-based models like BERT.

Importance of NER:

- Relation Extraction
- Machine Translation
- Information Retrieval
- Common Named Entity Recognition tags:
  - **LOC**: Location (e.g., countries, cities, rivers)
  - **PER**: Person (e.g., individual names, titles)
  - **ORG**: Organization (e.g., companies, institutions, agencies)
  - **TIME**: Time (e.g., dates, periods)

Example of Named Entity Recognition

He	met	Barack	Obama	in	Paris
O	O	B-PER	E-PER	O	S-LOC

Name entity recognition (NER) ambiguity: Washington could be referencing to a person, a location, a political entity, etc.

- A dictionary based approach is usually a bad idea.

6.1.4 Semantic Role Labeling

Semantic role labeling (SRL) is the process of identifying the roles of words or phrases in a sentence with respect to a predicate (verb). For example, in the sentence "John gave Mary a book," SRL assigns roles such as "giver" (John), "recipient" (Mary), and "theme" (a book).

SRL is crucial for understanding the meaning of sentences and is widely used in machine translation and dialogue systems. State-of-the-art SRL systems employ deep learning techniques, such as attention mechanisms and transformer models, to improve performance.

- Semantic role labeling tags:
  - **V**: Verb
  - **A0**: Acceptor (the entity that accepts something)
  - **A1**: Thing accepted (the entity that is accepted)
  - **A2**: Accepted from (the entity from which something is accepted)
  - **A3**: Attribute (the characteristic or feature of something)
  - **AM-MOD**: Modal (indicates modality such as possibility or necessity)
  - **AM-NEG**: Negation (marks the negation of an action or state)
  - **AM-TMP** Temporal modifier (indicating time)

Example of Semantic Role Labeling

He	gave	Mary	a	book	yesterday
A0	V	A2	O	A1	AM-TMP

6.2 CNN for sequence labelling

Convolutional Neural Networks (CNNs) have been widely used for sequence labeling tasks such as Named Entity Recognition (NER), Part-of-Speech (POS) tagging, and Chunking. Unlike traditional methods that rely on hand-crafted features, CNNs can automatically extract meaningful patterns from raw text.

6.2.1 Input Layer

The input layer represents each word as a combination of multiple features:

- **Word Embeddings**: Pre-trained or randomly initialized vectors (e.g., Word2Vec, GloVe).
- **Subword Features**:
  - **Character embeddings** capture word structure.
  - **Prefix/Suffix features** help identify morphological patterns (e.g., "un-" in "unknown", "-ing" in "running").

  - **Substrings** extract important internal segments (e.g., "eat" in "eating").

- **Gazetteer Features**: Indicate if a word appears in predefined lists (e.g., persons, genes, locations).

These features are stored in separate lookup tables and enhance the model's ability to capture linguistic patterns.

6.2.2 Position Feature

To capture positional dependencies, CNNs often use position embeddings or relative position indices.

- **Absolute Position Embeddings**: A learned embedding for each position in the sentence is added to the word embeddings.

- **Relative Position Encoding**: Instead of absolute indices, the model considers the relative distance between words, which helps preserve word-order information.

  - Example: add a relative position feature $i - pos_w$, where pos_w is the position of the word of interest relative to the verb.

For instance, in Named Entity Recognition (NER), the position of a word relative to an entity can influence its label (e.g., words appearing before a person's name might be titles like "Dr." or "Mr.").

6.2.3 Convolution

The convolutional layer extracts local features from the input sequence by applying 1D convolutions over word representations.

- **Lookup Table**: Each word is represented by concatenating multiple feature embeddings (word embeddings, subword features, gazetteer features).

- **Convolution**: At any position, a linear activation is computed from a window of neighboring representations, forming a 1D convolution.

- **Max Pooling**: A max operation across positions is applied to obtain a fixed-length hidden representation, allowing the model to capture the most salient features across the entire sentence.

This process enables the model to learn local dependencies while maintaining global context.

6.2.4 Score Prediction

The pooled representation from the convolutional layer serves as input to a regular neural network.

- **Neural Network Processing**: The input is passed through a fully connected layer using a hard version of the tanh activation function.

- **Unary Potentials for CRF**: The output scores are used as unary potentials in a **chained Conditional Random Field (CRF)** for structured prediction.
- **Task-Specific CRFs**: Each task has its own CRF, with no connections between them.
- **Separate Neural Networks**: A distinct neural network is trained for each task.

This setup enables effective label prediction while maintaining structured dependencies.

6.2.5 Conditional Random Field (CRF) for Structured Prediction

A Conditional Random Field (CRF) is used to model dependencies between output labels in sequence labeling tasks. Instead of predicting labels independently, the CRF layer ensures that the predicted sequence follows linguistic constraints.

- **Probability of a Label Sequence**: Given an input sequence $X = (x_1, x_2, ..., x_n)$ and a label sequence $Y = (y_1, y_2, ..., y_n)$, the CRF models the conditional probability as:
$$P(Y|X) = \frac{\exp(S(X, Y))}{\sum_{Y'} \exp(S(X, Y'))}$$
where $S(X, Y)$ is the sequence score, and the denominator normalizes over all possible label sequences.
- **Score Function**: The sequence score consists of unary and transition scores:
$$S(X, Y) = \sum_i \left(s_{y_i} + T(y_{i-1}, y_i)\right)$$
where:
  - s_{y_i} is the unary score from the neural network for word x_i.
  - $T(y_{i-1}, y_i)$ is the transition score from label y_{i-1} to y_i.
- **Viterbi Algorithm for Decoding**: Instead of selecting the highest-scoring label for each token independently, CRF finds the best label sequence:
$$Y^* = \arg\max_Y S(X, Y)$$
using the **Viterbi algorithm** for efficient decoding.
- **Training Objective**: The CRF is trained to maximize the log-likelihood of the correct label sequence:
$$\mathcal{L} = \log P(Y|X)$$
which ensures structured dependencies are learned effectively.

Sequential vs Non-Sequential Classification:

- **Sequential Classification**: In sequence labeling tasks, such as POS tagging or NER, labels are predicted in a sequence, where the label for each word depends not only on the current word but also on neighboring words. The CRF models this as a linear chain, where the label for each word is conditioned on both its own features and the labels of neighboring words.

- **Non-Sequential Classification**: In contrast, non-sequential classification tasks, like image classification, involve predicting labels independently for each input, with no dependencies between the labels. Since there is no sequence or chain-like relationship between the labels, CRFs are not typically used in these tasks.

6.2.6 CNN for Sequence Labeling: Unsupervised Tasks

In addition to supervised tasks, we can design unsupervised tasks for sequence labeling without labeled data. A typical unsupervised task involves identifying whether a word in the middle of a text window is an "impostor."

- **Impostor Detection**: We take a window of words from a sentence and check if the middle word is an impostor. For example:
  - Original sentence: "The cat *sat* on the mat."
  - Impostor sentence: "The cat *think* on the mat."

  The goal is to identify the middle word in the window as an impostor.
- **Generating Impostor Examples**: Impostor examples can be generated from unlabeled text (e.g., Wikipedia) by randomly replacing the middle word of a window with a randomly chosen word.
  1. We select a window of words from the corpus.
  2. We replace the middle word with a randomly chosen word.
- **Training the Neural Network**: A neural network, often using word embedding, is trained to assign a higher score to the **original window** compared to the **impostor window**. The training objective is defined as:

$$J = \max\left(0, f(x_{\text{original}}) - f(x_{\text{impostor}}) + 1\right)$$

  where:
  - $f(x_{\text{original}})$ is the score for the original window.
  - $f(x_{\text{impostor}})$ is the score for the impostor window.
  - The network learns to assign a higher score to the original window.
- **Relation to Language Modeling**: This task is similar to language modeling, except we predict the middle word in a sequence, rather than predicting the next word in the sequence. It is also related to the skip-gram approach, where the model learns to predict context words (i.e., the surrounding words) given a target word.

6.3 Multitask Learning

The idea is to transfer knowledge learned within the word representations across the different tasks.

- Simply use the same lookup tables across tasks
- The other parameters of the neural networks are not tied

7 NLP Tasks 3 - Text Classification

7.1 Text Classification

7.1.1 Definition and Motivation

Text classification involves adding information to unstructured text. Examples include:

- **Filtering/ spam detection**: spam vs. non-spam.
- **Sentiment analysis**: detecting positive or negative sentiment (opinion polarity).
- **Topic labeling**: understanding what a text is talking about, and used for structuring and organizing data.
- **Language detection**: Classifying incoming text according to its language

The challenge arises because most information is contained in raw, unstructured text, making it difficult to interpret automatically.

7.1.2 Categories of Classification Systems

Text classification system can be divided in two main categories:

- **Rule-based systems**: use a set of handcrafted linguistic rules to classify data. These rules rely on semantically relevant elements (patterns) to identify categories.
  - Example: Words related to specific topics can be used to categorize content.
    - Politics: Donald Trump, Hillary Clinton
    - Sport: football, basketball, LeBron James
  - Advantages:
    - Human-comprehensible rules.
  - Drawbacks:
    - Requires deep knowledge of the domain.
    - Generating rules for a complex system is time-consuming.
    - Difficult to maintain and does not scale well.
- **Machine learning-based techniques**:
  - Example: Machine learning models can learn from large datasets to automatically classify content based on patterns in data.
  - Advantages:
    - Better performances
    - Generic models / does not require expert knowledge
  - Drawbacks:
    - Requires annotated corpora

7.1.3 Methods and Techniques

- **Naive Bayes**: has historically been one of the most popular models for text classification.
  - For a given document d and a set of classes $C = \{c_1, c_2, \ldots, c_j\}$, Naïve Bayes is a simple ("naïve") classification method based on Bayes' rule:

$$P(c \mid d) = \frac{P(d \mid c)P(c)}{P(d)}$$

  - **Conditional independence hypothesis**, meaning that the features (words in a document) are assumed to be independent given the class:

$$P(d \mid c) = \prod_{i=1}^{n} P(w_i \mid c)$$

  - **Bag-of-words assumption**: Assume position doesn't matter
  - The prediction is made according to the posterior probability $P(c \mid d)$, which depends on:
    - The likelihood $P(d \mid c)$, which represents the probability of observing the query document given the class.
    - The prior probability $P(c)$, which represents the prior belief about the class before observing the document.
  - **Bayes' Rule Applied to Document Classification**:
    - **Maximum a posteriori (MAP) class** for a document d and a class c is given by:

$$\begin{aligned} C_{MAP} &= \arg\max_{c \in C} P(c \mid d) \\ &= \arg\max_{c \in C} \frac{P(d \mid c)P(c)}{P(d)} \\ &= \arg\max_{c \in C} P(d \mid c)P(c) \\ &= \arg\max_{c \in C} \prod_{i=1}^{n} P(w_i \mid c)P(c) \end{aligned}$$

    - **Maximum likelihood Estimator (MLE)**:
      - The probability of a class c_j is estimated as:

$$P(c_j) = \frac{N_{c_j}}{N}$$

        where N_{c_j} is the number of documents in class c_j, and N is the total number of documents.
      - The probability of a word w_i given a class c_j is estimated as:

$$P(w_i \mid c_j) = \frac{\text{count}(w_i, c_j)}{\sum_k \text{count}(w_k, c_j)}$$

        where $\text{count}(w_i, c_j)$ is the number of times word w_i appears in documents of class c_j, and $\sum_k \text{count}(w_k, c_j)$ is the total count of all words in documents of class c_j.

* **Laplace (Add-1) Smoothing**: adjusts the maximum likelihood estimation by adding 1 to each count to avoid division by zero:

$$P(w_i \mid c_j) = \frac{\text{count}(w_i, c_j) + 1}{\sum_k \text{count}(w, c_j) + |V|}$$

where $|V|$ is the size of the vocabulary, ensuring that all words, including unseen ones, have a non-zero probability.

- **Recurrent Neural Network (RNN)**: is a class of neural networks designed to handle sequential data. Unlike traditional feedforward networks, RNNs process sequences by maintaining a hidden state that evolves over time, allowing the model to remember information from previous time steps and use it to make predictions for future steps.

  1. **One-to-One**: the model takes a single input at a time and produces a single output. This is the most basic form of an RNN, where there is no sequential dependence between inputs.
     - Example: Vanilla RNN.
  2. **One-to-Many**: used when a single input leads to a sequence of outputs.
     - Example: Image Captioning.
  3. **Many-to-One**: a sequence of inputs is processed to produce a single output.
     - Example: Sentiment analysis.
  4. **Many-to-Many**: processes sequences of inputs and produces sequences of outputs.
     - Example: Machine translation, speech synthesis.

  - **RNN Computational Graph**: For language modeling, we consider a sequence of word vectors $x_1, x_2, \ldots$. At each time step t, the RNN processes the current word vector x_t and the previous hidden state h_{t-1} to compute the current hidden state h_t as follows:

$$h_t = f(Vh_{t-1} + Ux_t)$$

where:

* V and U are weight matrices.
* h_{t-1} is the hidden state from the previous time step.
* x_t is the word vector at time step t.
* f is a non-linear activation function, typically tanh, ReLU or Sigmoid.

The output at each time step t, o_t, is generated by applying the softmax function to the hidden state h_t:

$$o_t = \text{softmax}(Wh_t)$$

where W is the weight matrix that projects the hidden state into the output space (usually the vocabulary size).
This process is repeated for each word in the sequence, where the RNN updates the hidden state at each time step based on the input word and the previous hidden state:

$$P(x_{t+1} = v_j \mid x_t, ..., x_1) = o_{t,j}$$

- **Backpropagation through time (BPTT)**: is the algorithm used to train neural networks by computing gradients of the loss function with respect to the weights. It updates the weights iteratively using these gradients to minimize the loss.
- **Truncated Backpropagation**: limits the number of time steps over which gradients are propagated. Instead of backpropagating all the way to the first time step, the gradients are propagated back only for a fixed number of time steps, known as the "truncation length".
  * Prevents vanishing/exploding gradients
  * **LSTMs** are a special type of RNN designed to overcome the vanishing gradient problem by introducing a memory cell that can maintain information over long periods.
  * **GRUs** are similar to LSTMs but have a simpler structure. They combine the input and forget gates into a single update gate.
- **Loss function**: measures the difference between the predicted and actual values. For classification, the common loss function is *cross-entropy loss*:

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

  where:
  * y_i is the true label
  * $\hat{y}_i$ is the predicted probability for class i

  Weights are updated during backpropagation:

$$w = w - \eta \nabla_w L$$

  where η is the learning rate, and $\nabla_w L$ is the gradient of the loss.
- **Other problems of RNNs:** for classification, you want to incorporate information from words both preceding and following.
  * **Bidirectionnal RNNs**

7.1.4 Strengths of ML Techniques for Text Classification

- **Scalability**: Manual annotation (curation) is time-consuming. Machine learning can automatically analyze millions of surveys, comments, emails, etc.
- **Real-time Analysis**: Machine learning enables real-time analysis, such as with Tweetoscope, allowing for rapid identification of situations (e.g., earthquakes) and immediate action. It also facilitates continuous monitoring of brand mentions to identify critical information and respond quickly.
- **Consistency**: Human annotators are prone to mistakes due to distractions, fatigue, and subjectivity, which leads to inconsistencies. Machine learning, however, applies the same criteria to all data, ensuring consistent and reliable results.

8 Advanced NLP Techniques

As NLP has evolved, more advanced techniques have been developed to improve language understanding and generation. These techniques leverage deep learning, large-scale datasets, and sophisticated architectures to achieve state-of-the-art performance across various language tasks.

8.1 Sequence-to-sequence (Seq2seq) Models

A Seq2seq model is designed to transform an input sequence into an output sequence. The core idea is to map a sequence of one type (e.g., words in a sentence) to another sequence of the same or different type.

A sequence-to-sequence model is an example of a Conditional Language Model:

- **Conditional** because its predictions are also conditioned on the source sentence x.
- **Language Model** because the decoder is predicting the next word of the target sentence y.

Vanilla Seq2Seq models include:

- **Recurrent Neural Networks (RNN)**
- **Long Short-Term Memory (LSTM)**
- **Gated Recurrent Unit (GRU)**

Seq2seq models can be used for:

- Translation
  - **Statistical machine translation (SMT)**:
    - computes $P(T \mid S)$ through Bayes' rule.
  - **Neural machine translation (NMT)**:
    - directly computes $P(T \mid S)$.
  - NMT vs. SMT:
    - Better performances
    - A single neural network to be optimized
    - Requires much less human engineering effort
    - NMT is less interpretable
    - NMT is difficult to control
- Summarization
- Parsing
- Dialogue

8.1.1 Architecture of Seq2seq Models

Seq2seq models typically consist of two parts:

- **Encoder**: This part takes the input sequence and processes it into a fixed-size vector called the context vector (it's the final hidden state of the encoder). The encoder encodes the information from the input sequence into a compact representation. This is often done using a RNN, LSTM, or GRU. More recently, Transformer-based models have replaced traditional RNNs in many tasks.
  - **Hidden state update:**

    $$h_{E_t} = f\left(V h_{E_{t-1}} + U x_t\right)$$

    where:
    * h_{E_t} is the hidden state of the encoder at time step t.
    * $h_{E_{t-1}}$ is the hidden state of the encoder at the previous time step $t-1$.
    * x_t is the input at the current time step t.
    * V and U are weight matrices learned during training.
    * f is a non-linear activation function (e.g., tanh or ReLU).
  - **Context vector:**

    $$\mathbf{c} = h_{E_T}$$

    where T is the length of the input sequence.
- **Decoder**: The decoder takes the context vector (and sometimes the previous output) and generates the output sequence step by step. This output sequence can be of variable length.
  - **Hidden state update:**

    $$h_{D_t} = f\left(V h_{D_{t-1}}\right)$$

    where:
    * h_{D_t} is the hidden state of the decoder at time step t.
    * $h_{D_{t-1}}$ is the hidden state of the decoder at the previous time step $t-1$.
    * V is a weight matrix for the decoder's state transition.
    * f is a non-linear activation function (e.g., tanh or ReLU).
  - **Output generation:**

    $$y_t = \text{softmax}\left(W h_{D_t}\right)$$

    where:
    * y_t is the output at time step t (predicted token/word).
    * W is a weight matrix that projects the decoder's hidden state to the output space.
    * The function softmax converts the output into a probability distribution over the vocabulary.

8.1.2 Training and Evaluation

During training, the model learns to map the input sequence to the corresponding output sequence by minimizing a loss function, typically cross-entropy, which measures the difference between the predicted output sequence and the true output sequence, when such value exist.

- **Teacher Forcing**: During training, the decoder uses the actual output sequence (from the training data) at each time step rather than relying on its own previous predictions.
- **Greedy Algorithm**: Method for generating output sequences during inference. At each time step, the decoder selects the token with the highest probability from the softmax output — that is, it chooses the most likely next word without considering future possibilities. This process continues step by step until an end-of-sequence token is produced or a maximum length is reached. While greedy decoding is fast and simple, it can lead to suboptimal results because it makes locally optimal choices without exploring alternative sequences that might lead to better overall translations or predictions.
- **Beam Search Decoding**: improves upon greedy decoding by exploring multiple possible output sequences at each time step. Instead of choosing only the most probable token, beam search keeps track of the top k (beam width) most likely partial sequences. At every decoding step, it expands each sequence by all possible next tokens, computes their combined probabilities, and retains only the top k sequences. It is much more efficient than exhaustive search, but still does not guarantee to find the optimal solution.

The Bottleneck Problem: The encoding of the source sentence needs to capture all information about the source sentence, thus the context vector is a bottleneck of information.

Such models can be evaluated using the **Bilingual Evaluation Understudy (BLEU) Score**: compares the machine-written translation to one or several human-written translation(s), and computes a similarity score based on:

- n-gram precision
- plus a penalty for too-short system translations

8.2 Attention Mechanisms and Transformers

Transformers, introduced by Vaswani et al. (2017) in "Attention Is All You Need," revolutionized NLP by replacing recurrent networks with self-attention mechanisms.

8.2.1 Attention

Definition: Given a set of vector **values**, and a vector **query**, attention is a technique to compute a weighted sum of the values, dependent on the query. We sometimes say that the query attends to the values.

Attention is a way to obtain a fixed-size representation of an arbitrary set of representations (the values), dependent on some other representation (the query).

8.2.2 Attention in Seq2seq Models

One of the problems of Seq2seq models is the **bottleneck problem**: the encoding of the source sentence needs to capture all information about the source sentence, thus the last hidden state vector is a bottleneck of information. Instead of using only the last hidden state vector, **attention** mechanism uses all encoder hidden states.

In Seq2seq models with attention mechanism, the context vector is weighted sum of all encoder hidden states:

$$\mathbf{c}_t = \sum_{i=1}^{T} \alpha_{t,i}\, h_{E_i}$$

where the weights can be computed by applying the softmax function to the scores (to have a probability distribution):

$$\alpha_{t,i} = \frac{\exp(e_{t,i})}{\sum_{j=1}^{T} \exp(e_{t,j})}$$

$$e_{t,i} = \text{score}(h_{D_{t-1}}, h_{E_i})$$

and where the score can be computed the following ways:

Dot Product (Luong 2015):	$\text{score}(h_{D_{t-1}}, h_{E_i}) = h_{D_{t-1}}^{\top} h_{E_i}$
General (Luong variant 2015):	$\text{score}(h_{D_{t-1}}, h_{E_i}) = h_{D_{t-1}}^{\top} W_a h_{E_i}$
Concatenation/Additive (Bahdanau 2015):	$\text{score}(h_{D_{t-1}}, h_{E_i}) = v_a^{\top} \tanh(W_a \times \text{concat}[h_{D_{t-1}}; h_{E_i}])$

The weighted sum is a selective summary of the information contained in the values, where the query determines which values to focus on.

Attention may be use with many architectures (not just seq2seq) and many tasks (not just MT)

8.2.3 Benefits and limitations of Attention in Seq2seq Models

Benefits of attention in Seq2seq models:

- Attention significantly improves NMT performance
- Attention solves the bottleneck problem
- Attention helps with vanishing gradient problem
- Attention provides some interpretability

Limitations of Seq2seq models with attention:

- Dealing with long-range dependencies is still challenging.
  - Despite GRUs and LSTMs, RNNs still need attention mechanism to deal with long range dependencies.
- The sequential nature of RNN prevents parallelization.

If attention gives us access to any state, do we still need RNNs?

8.2.4 Transformers

Transformers don't use a single fixed-size context vector. Instead, they use self-attention mechanisms to allow every output token to attend to all encoder outputs (i.e., a dynamic "context" for each output position).

Transformers rely entirely on attention mechanisms, without recurrence or convolution.

Transformers use a specific form of attention called scaled dot-product self-attention, and they use multi-head attention, positional encoding, and feedforward layers stacked in encoder and decoder blocks.

$$\text{Scaled Dot Product (Vaswani 2017):} \qquad \text{score}(Q, K) = \frac{QK^\top}{\sqrt{d_k}}$$

Key advantages of transformers include:

- **Self-Attention**: Allows the model to focus on relevant words in a sentence regardless of their position, creating dynamic, context-dependent representations.
  - Computing self-attention: cf CM8 slides 10-15
  - *Example*: In the sentence "The animal didn't cross the street because it was too wide," attention helps the model understand that "it" refers to "the street" rather than "the animal."
- **Multi-Head Attention**: Enables the model to jointly attend to information from different representation subspaces, capturing various linguistic phenomena simultaneously.
  - Computing multi-head attention: cf CM8 slides 19-20
  - *Example*: One attention head might focus on syntactic relationships while another focuses on semantic relationships.
- **Parallelization**: Unlike RNNs, transformers process entire sequences simultaneously, speeding up training and inference by orders of magnitude.
  - *Example*: Training BERT took days on specialized hardware, whereas equivalent RNN models would have required months.
- **Position Encodings**: Since transformers lack sequential processing, positional encodings are added to inject word order information.

- Unlike RNNs and CNNs encoders, the attention encoder outputs do not depend on the order of the inputs.
- The idea is to add positional information of a input token in the sequence into the input embedding vectors.
- *Example*: Sinusoidal position encodings create unique patterns for each position, allowing the model to differentiate "dog bites man" from "man bites dog."

8.2.5 Transformer Block

A Transformer block consists of two main sublayers:

1. **Multi-head attention**: This mechanism allows the model to focus on different parts of the input sequence simultaneously by computing multiple attention scores in parallel, each with its own set of learned parameters. The outputs of the individual attention heads are concatenated and projected to produce the final attention output.

2. **Feed-forward network (FFN)**: After the attention layer, the output is passed through a position-wise fully connected feed-forward network, which typically consists of two linear layers with a ReLU activation function in between.

Each of these sublayers is followed by a residual (or "short-circuit") connection, which allows the original input to be added back to the output of the sublayer. This residual connection helps in mitigating the vanishing gradient problem and improves training.

Additionally, after the residual connection, the output of each sublayer is normalized (LayerNorm) is applied to normalize the output and stabilize training. This entire structure (attention and feed-forward sublayers with residual connections and normalization) is repeated in both the encoder and decoder of the Transformer architecture.

8.2.6 Transformer Decoder

The Transformer decoder block is designed to generate outputs sequentially, attending to both previously generated tokens and the input sequence. It contains the following key components:

1. **Masked Self-Attention Mechanism:**

   - This mechanism ensures that each token can only attend to previously generated tokens, not future ones.
   - This prevents information leakage during training and is crucial for autoregressive tasks such as language modeling or translation.

2. **Encoder-Decoder Attention:**

   - In this layer, the queries come from the output of the previous decoder layer.
   - The keys and values come from the output of the encoder.

- This mechanism allows the decoder to focus on relevant parts of the input sequence when generating each token, linking the decoder's generation process to the encoder's understanding of the input.

Together, these sublayers enable the transformer decoder to generate contextually relevant and accurate predictions at each step.

8.3 Pretrained Language Models

Recent advancements in NLP rely on large-scale pretrained language models, which are fine-tuned for specific tasks:

- **BERT (Bidirectional Encoder Representations from Transformers)**: Trained bidirectionally to capture context from both left and right sides.
  - BERT uses two unsupervised tasks during its pre-training:
    1. **Masked Language Model (Masked LM):**
       * In this task, a percentage of the input words are randomly masked.
       * The model's objective is to predict the masked words based on their context, allowing it to learn bidirectional representations of text.
    2. **Next Sentence Prediction (NSP):**
       * This task aims to learn the relationship between two sentences.
       * The model predicts whether sentence B is the actual next sentence that follows sentence A, or if it is a randomly chosen sentence.
  - **WordPiece Tokenizer**: constructs a fixed-size vocabulary consisting of whole words, subwords (both at the start or inside words), and individual characters. It uses a greedy longest-match-first algorithm to tokenize text based on this vocabulary, breaking words into subword units when necessary. For example, "unaffable" would be tokenized as ["un", "##a", "##able"].
- **GPT (Generative Pretrained Transformer)**: Focuses on text generation with autoregressive learning.
  - *Evolution*: GPT (125M parameters) → GPT-2 (1.5B) → GPT-3 (175B) → GPT-4 (estimated trillions) shows the scaling trend.
  - **Autoregressive Model**: predicts the next word in a sentence based on all the preceding words.
- **T5 (Text-to-Text Transfer Transformer)**: Converts all NLP tasks into text-to-text format, making it highly flexible.
  - *Unified Framework*: Unlike specialized models, T5 handles classification, translation, and generation with the same architecture.
- **XLNet and RoBERTa**: Improvements on BERT that enhance contextual understanding.
  - *XLNet*: Uses permutation language modeling to capture bidirectional context without the [MASK] token limitations.

  - *RoBERTa*: Optimizes BERT training by removing next sentence prediction and using dynamic masking.

- **ELECTRA**: Uses a discriminative approach where a generator creates corrupted text and a discriminator learns to detect replacements.
  - Training is more efficient as the model learns from every token position rather than just masked tokens.

8.4 Advanced Techniques in Model Architecture

- **Model Compression**: The objective of model compression is to simplify a model while maintaining its performance. This simplification can be achieved through two main approaches:
  - **Size Reduction**: Achieved by minimizing the number of model parameters, leading to lower RAM usage.
  - **Latency Reduction:** Accomplished by speeding up prediction times, which reduces runtime energy consumption and carbon footprint.
  - **Pruning**: Technique used to reduce the size of a model by eliminating unnecessary parameters or features that provide similar information (redundant features), often based on their contribution to the model's output. This is typically done by removing weights that have small magnitudes or that do not significantly affect the performance, leading to a more efficient model with reduced memory usage and faster inference times.
    - **Pruning Connections**: Setting specific weights in the matrix to zero.
    - **Pruning Neurons**: Zeroing out all values in an entire column.
  - **Distillation**: Process where a smaller, (the student) learns to mimic the behavior of a larger model (the teacher), usually by minimizing the difference between their predictions. This technique enables the smaller model to achieve a similar performance to the larger model while being more efficient in terms of size and computation.
  - **Quantization**: Technique used to reduce the precision of the model's weights, activations, or both. Instead of using 32-bit floating-point numbers, the model can be quantized to use lower precision, such as 8-bit integers.
    - **Quantization-Aware Training (QAT)**: Applied during the training phase, where parameters are directly converted into a lower-precision floating-point representation.
    - **Post-Training Quantization (PTQ)**: Applied after the training.

- **Mixture of Experts (MoE)**: Divides the network into specialized sub-networks activated conditionally based on input.
  - *Example*: Switch Transformer and GShard scale to trillions of parameters while keeping computation costs manageable.
  - *Advantage*: Achieves higher parameter count without proportional increases in computation.

- **Sparse Attention Mechanisms**: Address quadratic complexity of standard attention.
  - *Example*: Longformer uses a combination of sliding window attention and global attention to process documents with thousands of tokens efficiently.
  - *Variants*: Big Bird, Reformer, and Performer each offer different approaches to sparse attention.
- **Parameter-Efficient Fine-tuning**:
  - *Adapters*: Small trainable modules inserted between frozen pretrained layers.
  - *LoRA (Low-Rank Adaptation)*: Decomposes weight updates into low-rank matrices, reducing parameters by 10,000x in some cases.
  - *Prompt Tuning*: Trains continuous prompt vectors while keeping the model frozen.

8.5 Prompting Strategies

Modern NLP models can perform tasks without requiring extensive labeled data:

- **Zero-shot Learning**: Model's ability of a generative language model to understand and respond to a prompt without having been specifically trained on similar examples beforehand.
- **Few-shot Learning**: Model's ability to learn and perform a task after being exposed to one (one-shot) or a few (few-shot) examples.

8.6 Prompt Engineering and Instruction Following

With large-scale models like GPT, effective interaction is crucial:

- **Instruction Tuning**: Models are fine-tuned to respond better to specific instructions.
  - *Human Feedback*: RLHF (Reinforcement Learning from Human Feedback) aligns model outputs with human preferences.
- **Chain-of-Thought Prompting**: AI technique in which a model generates detailed, step-by-step reasoning to solve complex tasks or answer questions, thereby improving the accuracy and transparency of its responses.
  - *Example*: "To solve 18×27, I'll multiply $18 \times 20 = 360$, then $18 \times 7 = 126$, and add: $360 + 126 = 486$."
- **Multi-Agent Prompting**: involves multiple agents (LLMs) collaborating to solve a complex problem.
- **In-context Learning**: The model understands patterns from input examples without fine-tuning.
  - *Techniques*: One-shot, few-shot, and multi-shot prompting with carefully selected examples.

- **Retrieval-Augmented Generation (RAG)**: Add information from a trusted data source to the user's prompt (i.e. combines retrieval systems with generative models).
  - *Example*: For answering specific questions, the model first retrieves relevant documents, then generates answers based on both the question and retrieved information.
  - *Advantage*: Reduces hallucination by grounding generation in retrieved facts.

8.7 Evaluation and Interpretability

Advanced techniques for understanding and assessing NLP models:

- **Behavioral Testing**: Systematic probing of model capabilities and limitations.
  - *Example*: CheckList evaluates models across linguistic capabilities like negation handling and robustness to typos.
- **Explainable AI for NLP**: Methods to interpret model decisions.
  - *Attention Visualization*: Shows which input tokens influence which output tokens.
  - *LIME and SHAP*: Attribute predictions to input features.
  - *Example*: Identifying which words in a medical report led to a specific diagnosis prediction.
- **Adversarial Testing**: Identifying weaknesses through challenging examples.
  - *Example*: Developing inputs that cause factual errors or expose biases in the model's responses.