Summer of Science

End-Term Report 2025

Large Language Models

Koushik Reddy

23B1000

Contents

Chapter 1

Introduction to LLMs and Machine Learning Basics

1.1 Basics of Large Language Models

Large Language Models (LLMs) are a class of deep learning models trained on massive text corpora to understand and generate human-like language. These models belong to the broader category of Natural Language Processing (NLP) systems and have revolutionized fields like conversational AI, search engines, translation, and content generation.

LLMs are based on the **transformer architecture**, which enables them to process input sequences in parallel and capture long-range dependencies using mechanisms like self-attention. Unlike earlier sequence models such as RNNs and LSTMs, transformers allow for more efficient and scalable training.

Some key characteristics of LLMs include:

- **Pretraining on large text corpora**: LLMs are initially trained on vast amounts of text data in an unsupervised manner, learning to predict the next word in a sentence or fill in masked tokens.
- **Fine-tuning for specific tasks**: After pretraining, LLMs can be fine-tuned on domain-specific data for tasks like summarization, question answering, and code generation.
- **Few-shot and zero-shot learning**: With enough parameters, LLMs demonstrate the ability to perform tasks with little or no task-specific training, relying solely on prompts.

Modern LLMs such as GPT, BERT, T5, and LLaMA have shown remarkable capabilities across a wide range of tasks, enabling more natural and intelligent human-computer interaction. However, they also raise challenges related to ethical use, hallucination, and computational cost.

In summary, understanding the foundational ideas behind LLMs is crucial for exploring their applications and limitations in real-world AI systems.

1.2 Applications of Large Language Models

Large Language Models (LLMs) have enabled significant advancements across a wide range of applications by leveraging their ability to understand, generate, and manipulate human language. These models are versatile and adaptable, making them foundational in modern AI systems.

1. **Conversational Agents and Chatbots**

   LLMs are widely used in building chatbots and virtual assistants such as ChatGPT, Siri, and Alexa. They can handle both casual and task-oriented conversations, provide informative responses, and adapt to different contexts based on the prompt history.

2. **Text Generation**

   LLMs are capable of producing coherent and contextually relevant text, making them suitable for applications like:

   - Writing assistance tools (e.g., Grammarly, Notion AI)
   - Story and content generation
   - Email and document drafting

3. **Machine Translation**

   With sufficient training data, LLMs can accurately translate text between multiple languages. Models like Google's mT5 and Meta's NLLB have achieved state-of-the-art results in multilingual translation tasks.

4. **Summarization**

   LLMs can generate concise and coherent summaries of long documents, news articles, or research papers. This is especially useful in academic, legal, and journalistic contexts where digesting large volumes of text is necessary.

5. **Sentiment Analysis and Text Classification**

   Businesses use LLMs to extract sentiment or classify text into predefined categories, such as spam detection, review sentiment analysis, or topic categorization in customer feedback.

6. **Code Generation and Programming Assistance**

   LLMs trained on code (e.g., Codex, CodeLLaMA) can write code snippets, debug errors, or explain programming concepts, making them useful in IDEs and platforms like GitHub Copilot.

7. **Information Retrieval and Question Answering**

   LLMs power search systems that go beyond keyword matching by understanding the semantic intent of queries. They can extract answers from documents or generate responses grounded in factual knowledge.

8. **Personalization and Recommendation**

   By analyzing user input and preferences, LLMs can personalize content, recommend products or articles, and generate customized messages or responses in marketing and e-commerce.

9. **Education and Tutoring**

   LLMs can serve as on-demand tutors, explaining complex topics, generating quizzes, or simulating conversations for language learning and exam preparation.

10. **Healthcare and Medical Analysis**

    In medical applications, LLMs assist in summarizing patient records, interpreting clinical notes, and even supporting diagnosis based on textual data from medical literature or electronic health records (EHRs).

Conclusion: The adaptability of LLMs across diverse domains makes them one of the most impactful developments in AI. However, responsible deployment is critical, as misuse or overreliance can lead to misinformation, bias propagation, or ethical concerns.

1.3 Core Machine Learning Concepts

This section explores the foundational concepts in machine learning that underpin many aspects of large language models. We focus on three major paradigms: regression, classification, and clustering.

1.3.1 Regression

Regression is a supervised learning method where the goal is to predict a continuous output variable y given input features $x \in \mathbb{R}^n$. The simplest form is **linear regression**, which models the relationship as:

$$y = w^T x + b$$

where $w \in \mathbb{R}^n$ is the weight vector and $b \in \mathbb{R}$ is the bias.

Given a dataset of m examples $\{(x^{(i)}, y^{(i)})\}_{i=1}^m$, we aim to minimize the **mean squared error (MSE)** loss:

$$J(w, b) = \frac{1}{2m} \sum_{i=1}^{m} \left(w^T x^{(i)} + b - y^{(i)} \right)^2$$

Gradient Descent Update Rules:

$$w \leftarrow w - \alpha \cdot \frac{1}{m} \sum_{i=1}^{m} \left(w^T x^{(i)} + b - y^{(i)} \right) x^{(i)} \tag{1.1}$$

$$b \leftarrow b - \alpha \cdot \frac{1}{m} \sum_{i=1}^{m} \left(w^T x^{(i)} + b - y^{(i)} \right) \tag{1.2}$$

where α is the learning rate.

Linear regression is widely used due to its simplicity, interpretability, and efficiency.

1.3.2 Classification

Classification is another supervised learning task where the goal is to assign a label $y \in \{1, 2, \ldots, K\}$ to an input x. For binary classification ($y \in \{0, 1\}$), **logistic regression** is a commonly used algorithm.

It models the probability that $y = 1$ as:

$$P(y = 1|x) = \sigma(w^T x + b)$$

where $\sigma(z) = \frac{1}{1+e^{-z}}$ is the sigmoid function.

Loss Function (Binary Cross-Entropy):

$$J(w, b) = -\frac{1}{m} \sum_{i=1}^{m} \left[y^{(i)} \log \hat{y}^{(i)} + (1 - y^{(i)}) \log(1 - \hat{y}^{(i)}) \right]$$

where $\hat{y}^{(i)} = \sigma(w^T x^{(i)} + b)$

Extension to Multi-Class Classification:

For multi-class problems, we use the **softmax function**:

$$P(y = k|x) = \frac{e^{w_k^T x}}{\sum_{j=1}^{K} e^{w_j^T x}}$$

and the corresponding **categorical cross-entropy** loss:

$$J = -\frac{1}{m} \sum_{i=1}^{m} \sum_{k=1}^{K} y_k^{(i)} \log P(y = k|x^{(i)})$$

1.3.3 Support Vector Machines (SVMs)

Support Vector Machines are powerful supervised learning models used primarily for classification, but also applicable to regression. The key idea is to find the optimal hyperplane that maximally separates the data points of different classes.

Binary Classification Setup: Given a training set of m labeled samples $\{(x^{(i)}, y^{(i)})\}_{i=1}^{m}$ where $x^{(i)} \in \mathbb{R}^n$, and $y^{(i)} \in \{-1, +1\}$, the goal is to find a decision boundary:

$$w^T x + b = 0$$

such that the margin between the two classes is maximized.

Margin Definition: The margin is defined as the distance between the separating hyperplane and the nearest data point. For a correctly classified point:

$$y^{(i)}(w^T x^{(i)} + b) \geq 1$$

Primal Optimization Problem:

$$\min_{w,b} \frac{1}{2}\|w\|^2 \quad \text{subject to} \quad y^{(i)}(w^T x^{(i)} + b) \geq 1 \quad \forall i$$

This is a convex quadratic optimization problem with linear constraints.

Soft Margin SVM: To handle non-linearly separable data, we introduce slack variables $\xi_i \geq 0$:

$$\min_{w,b,\xi} \frac{1}{2}\|w\|^2 + C \sum_{i=1}^{m} \xi_i \quad \text{subject to} \quad y^{(i)}(w^T x^{(i)} + b) \geq 1 - \xi_i$$

where C controls the trade-off between margin maximization and error penalty.

Nonlinear SVM (Kernel Trick): To separate non-linearly separable data, we map inputs to a higher-dimensional feature space via $\phi(x)$, and compute dot products using a kernel function:

$$K(x, x') = \phi(x)^T \phi(x')$$

Popular kernels:

- Polynomial: $K(x, x') = (x^T x' + c)^d$
- RBF (Gaussian): $K(x, x') = \exp(-\gamma \|x - x'\|^2)$

Advantages of SVMs:

- Effective in high-dimensional spaces
- Theoretical guarantees (maximum margin)
- Robust to overfitting (with appropriate kernel and regularization)

1.3.4 Decision Trees

Decision Trees (DTs) are supervised learning algorithms used for both classification and regression. They model decisions as a tree-like structure, where each internal node corresponds to a feature test, each branch to an outcome, and each leaf to a class label or numerical value.

Structure:

- **Root node**: The first decision node based on the best splitting feature.
- **Internal nodes**: Contain feature-based decisions.
- **Leaves**: Contain output predictions.

Training Objective: To construct a tree, we recursively split the data to maximize the **purity** of the resulting subsets.

Metrics to Evaluate Splits:

1. **Gini Impurity:**

$$G(t) = 1 - \sum_{k=1}^{K} p_k^2$$

where p_k is the fraction of instances of class k in node t.

2. **Entropy (Information Gain):**

$$H(t) = -\sum_{k=1}^{K} p_k \log_2 p_k$$

$$IG = H(parent) - \left[\frac{n_{\text{left}}}{n} H(\text{left}) + \frac{n_{\text{right}}}{n} H(\text{right}) \right]$$

3. **Classification Error:**

$$E(t) = 1 - \max_k p_k$$

Recursive Tree-Building Process:

- At each node, evaluate all possible feature splits.
- Choose the split that maximizes information gain or minimizes impurity.
- Recurse until:
  - All samples belong to one class
  - Max depth is reached
  - Minimum samples per leaf is met

Pruning: Decision trees tend to overfit. Pruning techniques (e.g., cost-complexity pruning) reduce the size of the tree to improve generalization.

Advantages of Decision Trees:

- Easy to understand and interpret
- Handle both numerical and categorical data
- Require little preprocessing (e.g., no normalization)

Disadvantages:

- Prone to overfitting on noisy data
- Unstable to small data perturbations
- Greedy splits may lead to suboptimal trees

Extensions:

- **Random Forests**: Ensembles of decision trees using bagging.
- **Gradient Boosted Trees**: Sequentially added trees that minimize residuals.

1.3.5 Clustering

Clustering is an unsupervised learning technique where the goal is to group a set of objects such that objects in the same group (called a cluster) are more similar to each other than to those in other groups.

Unlike supervised learning, clustering does not require labeled data. It is useful in exploratory data analysis, anomaly detection, customer segmentation, image compression, etc.

1. K-Means Clustering

K-Means is a centroid-based, iterative algorithm that partitions data into K clusters by minimizing the variance within each cluster.

Objective: Given m data points $\{x^{(1)}, x^{(2)}, \ldots, x^{(m)}\} \in \mathbb{R}^n$, and a chosen number of clusters K, the goal is to minimize:

$$J = \sum_{k=1}^{K} \sum_{x_i \in C_k} \|x_i - \mu_k\|^2$$

where μ_k is the mean (centroid) of cluster C_k.

Algorithm:

1. Initialize K centroids randomly: $\mu_1, \ldots, \mu_K$
2. Assign each point x_i to the closest centroid:

$$c_i = \arg\min_k \|x_i - \mu_k\|$$

3. Recompute each centroid as:

$$\mu_k = \frac{1}{|C_k|} \sum_{x_i \in C_k} x_i$$

4. Repeat steps 2–3 until convergence (i.e., assignments no longer change or maximum iterations reached).

Limitations:

- Sensitive to initialization (can lead to suboptimal solutions).
- Assumes spherical and equally sized clusters.
- Requires manual selection of K.

Improvement: Use **K-Means++** for smarter initialization of centroids to improve convergence and quality.

2. Gaussian Mixture Models (GMM)

GMM assumes the data is generated from a mixture of several Gaussian distributions with unknown parameters. It is a **soft clustering** method, where each point belongs to each cluster with a certain probability.

Model:

$$p(x) = \sum_{k=1}^{K} \pi_k \cdot \mathcal{N}(x|\mu_k, \Sigma_k)$$

where:

- π_k: Mixing coefficient for cluster k ($\sum_k \pi_k = 1$)
- $\mathcal{N}(x|\mu_k, \Sigma_k)$: Multivariate normal distribution

Training via Expectation-Maximization (EM):

1. **E-Step**: Compute posterior probabilities (responsibilities) of cluster k for data point x_i:

$$\gamma_{ik} = \frac{\pi_k \mathcal{N}(x_i|\mu_k, \Sigma_k)}{\sum_{j=1}^{K} \pi_j \mathcal{N}(x_i|\mu_j, \Sigma_j)}$$

2. **M-Step**: Update parameters:

$$\mu_k = \frac{\sum_{i=1}^{m} \gamma_{ik} x_i}{\sum_{i=1}^{m} \gamma_{ik}}, \quad \Sigma_k = \frac{\sum_{i=1}^{m} \gamma_{ik} (x_i - \mu_k)(x_i - \mu_k)^T}{\sum_{i=1}^{m} \gamma_{ik}}, \quad \pi_k = \frac{1}{m} \sum_{i=1}^{m} \gamma_{ik}$$

3. Repeat E and M steps until convergence.

Advantages:

- Probabilistic soft clustering
- Models ellipsoidal clusters

Disadvantages:

- Sensitive to initialization and outliers
- Requires estimating more parameters than K-Means

Conclusion: These foundational techniques form the basis for understanding and building machine learning models. They also provide essential insights into how LLMs learn statistical patterns and structure in large datasets during pretraining and fine-tuning stages.

Chapter 2

Deep Learning Fundamentals

2.1 Neural Networks and Architectures

2.1.1 Feedforward Neural Networks (FFNN)

Feedforward Neural Networks (FFNNs), also known as Multi-Layer Perceptrons (MLPs), are the foundational building blocks of deep learning. They consist of layers of neurons where information flows in one direction—from input to output—without cycles or loops.

Network Architecture:

- **Input Layer:** Receives raw features $x \in \mathbb{R}^n$
- **Hidden Layers:** Each layer transforms input using a linear operation followed by a non-linear activation function
- **Output Layer:** Produces the final prediction, typically with a softmax or sigmoid activation

Forward Pass (Mathematical Formulation):
For a single hidden layer:

$$z^{(1)} = W^{(1)}x + b^{(1)} \quad \text{(Linear transform)}$$

$$a^{(1)} = \sigma(z^{(1)}) \quad \text{(Activation)}$$

$$z^{(2)} = W^{(2)}a^{(1)} + b^{(2)} \quad \text{(Output logits)}$$

$$\hat{y} = f(z^{(2)}) \quad \text{(Output activation: softmax or sigmoid)}$$

Where:

- $W^{(l)}$ and $b^{(l)}$: Weights and biases for layer l
- σ: Non-linear activation function (e.g., ReLU, tanh)
- f: Output activation (e.g., softmax for classification)

Loss Function:
For classification tasks with softmax output:

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

Where:

- y_i: Ground truth (one-hot vector)
- $\hat{y}_i$: Predicted probability for class i

Backpropagation: Training the Network
Training is done using Gradient Descent or its variants (SGD, Adam) to minimize the loss. This involves computing the gradient of the loss function with respect to each parameter using the chain rule (backpropagation).

For a layer l, let $\delta^{(l)}$ be the error term (derivative of loss with respect to $z^{(l)}$).

$$\delta^{(2)} = \hat{y} - y \quad \text{(For output layer)}$$
$$\delta^{(1)} = (W^{(2)})^T \delta^{(2)} \odot \sigma'(z^{(1)}) \quad \text{(For hidden layer)}$$

Gradient updates:

$$\frac{\partial \mathcal{L}}{\partial W^{(l)}} = \delta^{(l)} (a^{(l-1)})^T, \quad \frac{\partial \mathcal{L}}{\partial b^{(l)}} = \delta^{(l)}$$

Update rule:

$$W^{(l)} := W^{(l)} - \eta \frac{\partial \mathcal{L}}{\partial W^{(l)}}, \quad b^{(l)} := b^{(l)} - \eta \frac{\partial \mathcal{L}}{\partial b^{(l)}}$$

Where η is the learning rate.

Activation Functions:

- **Sigmoid:** $\sigma(x) = \frac{1}{1+e^{-x}}$
- **Tanh:** $\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$
- **ReLU:** $\text{ReLU}(x) = \max(0, x)$

ReLU is most widely used in hidden layers due to its non-saturating gradient and efficiency.

Advantages of FFNNs:

- Can approximate any continuous function (Universal Approximation Theorem)
- Scalable to large datasets and tasks

Limitations:

- Require large amounts of labeled data
- Prone to overfitting without regularization (dropout, weight decay)
- Lack temporal memory (handled better by RNNs/LSTMs)

2.1.2 Convolutional Neural Networks (CNNs)

Convolutional Neural Networks (CNNs) are deep learning models specialized for processing data with a grid-like topology, such as images (2D grids of pixels). CNNs leverage spatial hierarchies in data and reduce the number of trainable parameters through shared weights.

Motivation: Traditional FFNNs do not scale well with high-dimensional inputs like images. For a 256×256 RGB image, a fully connected layer would have millions of parameters. CNNs solve this by using local receptive fields and weight sharing.

Key Components of CNN:

- **Convolutional Layer**
- **Activation Function** (typically ReLU)
- **Pooling Layer** (e.g., Max Pooling)
- **Fully Connected Layer**

1. Convolution Operation:
A convolutional layer performs discrete convolution between an input image and a filter (also called a kernel). For a 2D input I and kernel K of size $f \times f$, the convolution operation is defined as:

$$S(i,j) = (I * K)(i,j) = \sum_{m=0}^{f-1} \sum_{n=0}^{f-1} I(i+m, j+n) \cdot K(m,n)$$

This operation is applied over the entire image with a sliding window, producing a feature map.

Hyperparameters:

- **Stride (s):** Step size by which the kernel is moved.
- **Padding (p):** Adding zeros around the border to control the output size.

Output dimension:

$$\text{Output size} = \left\lfloor \frac{N - f + 2p}{s} \right\rfloor + 1$$

2. Activation Function: Commonly used is the ReLU:

$$\text{ReLU}(x) = \max(0, x)$$

ReLU introduces non-linearity and speeds up convergence by avoiding vanishing gradient problems.

3. Pooling Layer:
Pooling downsamples the feature maps, reducing the spatial dimensions and computation. Most common is max pooling.

$$\text{MaxPool}(X) = \max\{x_1, x_2, \ldots, x_k\}$$

This retains the most prominent features and makes the network more invariant to small translations.

4. Fully Connected Layer: After several convolution and pooling layers, the output is flattened and passed through one or more fully connected layers for final prediction.

CNN Architecture Example:

- Input: $32 \times 32 \times 3$ image
- Conv layer (5x5 kernel, 6 filters) $\rightarrow$ ReLU $\rightarrow$ $28 \times 28 \times 6$
- MaxPool (2x2) $\rightarrow$ $14 \times 14 \times 6$
- Conv layer (5x5, 16 filters) $\rightarrow$ ReLU $\rightarrow$ $10 \times 10 \times 16$
- MaxPool $\rightarrow$ $5 \times 5 \times 16$
- Flatten $\rightarrow$ Fully Connected Layer $\rightarrow$ Output

Advantages of CNNs:

- Fewer parameters due to weight sharing
- Captures local spatial features
- Translation invariance due to pooling
- Performs exceptionally well on image and spatial data

Applications of CNNs:

- Image classification (e.g., ImageNet)
- Object detection (e.g., YOLO, Faster R-CNN)
- Image segmentation (e.g., U-Net)
- Facial recognition and medical imaging

Limitations of CNNs:

- Require large labeled datasets and high computation
- Not ideal for sequential or long-range dependent data (handled better by RNNs or Transformers)
- Do not inherently encode positional relationships as well as attention-based models

2.2 Optimization Techniques

Optimization in deep learning is the process of adjusting neural network weights to minimize a cost (loss) function. This is crucial for effective model training.

2.2.1 Gradient Descent

Gradient Descent iteratively updates weights to minimize a cost function $J(\theta)$:

$$\theta := \theta - \eta \nabla_\theta J(\theta)$$

- θ: Model parameters (e.g., weights, biases)
- η: Learning rate (controls the step size)
- $\nabla_\theta J(\theta)$: Gradient of the loss function with respect to θ

2.2.2 Stochastic Gradient Descent (SGD)

SGD updates weights using a single training example:

$$\theta := \theta - \eta \nabla_\theta J(\theta; x^{(i)}, y^{(i)})$$

- $x^{(i)}, y^{(i)}$: Input and label of the i^{th} training sample

2.2.3 Mini-batch Gradient Descent

Mini-batch Gradient Descent updates using a subset of training data:

$$\theta := \theta - \eta \nabla_\theta \frac{1}{m} \sum_{i=1}^{m} J(\theta; x^{(i)}, y^{(i)})$$

- m: Mini-batch size

2.2.4 Momentum

Momentum helps accelerate learning by using a velocity term:

$$v_t = \gamma v_{t-1} + \eta \nabla_\theta J(\theta), \quad \theta := \theta - v_t$$

- v_t: Velocity vector at time t
- γ: Momentum coefficient (typically 0.9)

2.2.5 RMSProp

RMSProp adapts the learning rate using a moving average of squared gradients:

$$E[g^2]_t = \rho E[g^2]_{t-1} + (1-\rho)g_t^2, \quad \theta := \theta - \frac{\eta}{\sqrt{E[g^2]_t + \epsilon}} g_t$$

- ρ: Decay rate (e.g., 0.9)
- g_t: Gradient at step t
- ϵ: Small value to avoid division by zero

2.2.6 Adam Optimizer

Adam combines ideas from Momentum and RMSProp:

$$m_t = \beta_1 m_{t-1} + (1-\beta_1)g_t \quad \text{(1st moment)}$$
$$v_t = \beta_2 v_{t-1} + (1-\beta_2)g_t^2 \quad \text{(2nd moment)}$$
$$\hat{m}_t = \frac{m_t}{1-\beta_1^t}, \quad \hat{v}_t = \frac{v_t}{1-\beta_2^t}$$
$$\theta := \theta - \eta \frac{\hat{m}_t}{\sqrt{\hat{v}_t} + \epsilon}$$

- β_1, β_2: Exponential decay rates (default: 0.9 and 0.999)
- m_t, v_t: Estimates of first and second moments of gradient
- $\hat{m}_t, \hat{v}_t$: Bias-corrected estimates

—

2.3 Regularization Techniques

Regularization prevents overfitting by discouraging complex models.

2.3.1 L1 and L2 Regularization

Regularization adds penalty terms to the loss:
- **L1 Regularization (Lasso):**

$$J'(\theta) = J(\theta) + \lambda \sum_i |\theta_i|$$

- **L2 Regularization (Ridge):**

$$J'(\theta) = J(\theta) + \frac{\lambda}{2} \sum_i \theta_i^2$$

- λ: Regularization strength (hyperparameter)

2.3.2 Dropout

Dropout randomly disables neurons during training:

$$\tilde{h}_j = r_j h_j, \quad \text{where } r_j \sim \text{Bernoulli}(p)$$

- h_j: Activation of neuron j
- r_j: Random mask (0 or 1 with probability p)
- p: Keep probability (e.g., 0.5)

2.3.3 Early Stopping

Early stopping halts training when validation performance degrades. It avoids overfitting by selecting the optimal epoch.

2.3.4 Batch Normalization

BatchNorm normalizes activations across the mini-batch:

$$\hat{x}^{(i)} = \frac{x^{(i)} - \mu_{\mathcal{B}}}{\sqrt{\sigma_{\mathcal{B}}^2 + \epsilon}}, \quad y^{(i)} = \gamma \hat{x}^{(i)} + \beta$$

- $\mu_{\mathcal{B}}, \sigma_{\mathcal{B}}^2$: Mean and variance of mini-batch
- γ, β: Learnable scale and shift parameters
- ϵ: Small constant for numerical stability

2.3.5 Data Augmentation

This expands the dataset using transformations:

- **Images:** Rotation, flipping, scaling, color jittering
- **Text:** Synonym replacement, random insertion/deletion
- **Audio:** Time shift, noise addition, speed tuning

This improves generalization by increasing sample variety.

Chapter 3

Natural Language Processing (NLP)

3.1 Introduction to Natural Language Processing (NLP)

Natural Language Processing (NLP) is a subfield of Artificial Intelligence (AI) that focuses on enabling machines to understand, interpret, generate, and interact using human languages. It lies at the intersection of linguistics, computer science, and cognitive science, and plays a crucial role in the development of intelligent systems that interact with users in a natural and intuitive manner.

3.1.1 Motivation

Human language is inherently ambiguous, context-dependent, and variable. The goal of NLP is to build models and algorithms that can:

- Parse and understand the structure and meaning of text and speech.
- Extract relevant information from unstructured data.
- Enable tasks like translation, summarization, sentiment analysis, and question answering.

3.1.2 Historical Evolution

- **1950s–1980s:** Rule-based systems and symbolic AI dominated the early NLP era, relying on handcrafted linguistic rules.
- **1990s:** The rise of machine learning techniques led to statistical NLP, including models like Hidden Markov Models (HMMs) and Naive Bayes classifiers.
- **2010s–present:** Deep learning revolutionized NLP, with neural architectures like RNNs, LSTMs, GRUs, and Transformers leading to major breakthroughs.

3.1.3 Key Components of NLP

- **Lexical Analysis:** Breaking down text into tokens (words, punctuation).
- **Syntax Analysis (Parsing):** Analyzing grammatical structure using parse trees.
- **Semantic Analysis:** Interpreting the meaning of words and sentences.

- **Pragmatics and Discourse:** Understanding meaning in context and multi-sentence structures.
- **Language Generation:** Producing grammatically and semantically coherent text.

3.1.4 Challenges in NLP

- Ambiguity in word meaning (e.g., "bank" as a financial institution vs. river bank)
- Contextual dependency and polysemy
- Morphological complexity in different languages
- Sarcasm, irony, and figurative language
- Low-resource languages and multilingual support

3.1.5 Applications of NLP

- Machine Translation (e.g., Google Translate)
- Chatbots and Conversational Agents (e.g., Siri, Alexa)
- Sentiment Analysis (e.g., analyzing reviews or tweets)
- Information Retrieval (e.g., search engines)
- Named Entity Recognition, Part-of-Speech Tagging, Text Summarization

3.1.6 Role of Deep Learning in NLP

Recent advancements like word embeddings (e.g., Word2Vec, GloVe), pre-trained language models (e.g., BERT, GPT), and transformer-based architectures have significantly improved the state-of-the-art in NLP tasks.

In the following sections, we will explore the key models, methodologies, and architectures that have driven progress in NLP, with a special focus on how they are integrated within Large Language Models (LLMs).

3.2 NLP Pipeline

Natural Language Processing (NLP) involves transforming raw text into a form that machines can understand and process. The NLP pipeline refers to the sequence of steps or stages involved in this transformation, from raw input to useful structured output.

3.2.1 Text Collection

This is the initial stage where unstructured text data is gathered from various sources such as websites, books, social media, emails, or logs.

- Web scraping using tools like `BeautifulSoup` or `Scrapy`
- APIs from Twitter, Reddit, etc.
- Predefined corpora (e.g., Brown Corpus, Gutenberg, Wikipedia)

3.2.2 Text Preprocessing

Raw text often contains noise. Preprocessing cleans and standardizes the input.

- **Tokenization:** Splitting text into words, phrases, or sentences.
- **Lowercasing:** Converting all text to lowercase.
- **Stopword Removal:** Removing common non-informative words (e.g., ”the”, ”is”).
- **Punctuation Removal:** Eliminating symbols like ”.”, ”,”, etc.
- **Stemming:** Reducing words to their base form (e.g., “running” → “run”) using algorithms like Porter Stemmer.
- **Lemmatization:** Morphologically reducing words using vocabulary and POS tags (e.g., “better” → “good”).

3.2.3 Text Representation / Feature Extraction

After cleaning, text must be represented numerically for machine learning models.

- **Bag of Words (BoW):** Counts word frequency without considering order.
- **TF-IDF (Term Frequency–Inverse Document Frequency):** Weights frequent words lower if they appear in many documents.
- **Word Embeddings:** Vector representation capturing semantic meaning (e.g., Word2Vec, GloVe, FastText).
- **Contextual Embeddings:** Dynamic embeddings that consider context (e.g., BERT, ELMo).

3.2.4 Model Building

The extracted features are used to train models for various NLP tasks.

- **Traditional Models:** Naive Bayes, SVMs, Decision Trees, Logistic Regression.
- **Deep Learning Models:** RNNs, LSTMs, GRUs, CNNs for sequence tasks.
- **Transformers:** Attention-based models (e.g., BERT, GPT) that have revolutionized NLP.

3.2.5 Evaluation

Models are evaluated using appropriate metrics based on the task.

- **Classification:** Accuracy, Precision, Recall, F1 Score
- **Sequence Labeling:** Token-level precision/recall
- **Generation:** BLEU, ROUGE, perplexity

3.2.6 Deployment and Feedback

Once trained, the model can be deployed via APIs or as part of an application. Continuous feedback and retraining help improve the system over time.

- **Deployment Tools:** Flask, FastAPI, Docker, Kubernetes
- **Monitoring:** Logging incorrect predictions, drift detection

3.2.7 Summary

The NLP pipeline ensures that raw text data is transformed in a structured and meaningful way, enabling various applications such as sentiment analysis, chatbots, information retrieval, and machine translation.

3.3 Traditional NLP Methods

Before the rise of deep learning, Natural Language Processing primarily relied on statistical and rule-based methods. These approaches operated on simple vector space representations of text, focusing on frequency and co-occurrence patterns rather than learned contextual semantics.

3.3.1 Bag-of-Words (BoW)

The Bag-of-Words model represents a document as a multiset (bag) of its words, disregarding grammar and word order but keeping multiplicity.

Given a vocabulary $V = \{w_1, w_2, ..., w_N\}$, each document d is represented as a vector $\vec{v}_d \in \mathbb{R}^N$, where each component $v_{d,i}$ is the count of word w_i in document d.

$$\vec{v}_d = [\text{count}(w_1), \text{count}(w_2), ..., \text{count}(w_N)]$$

Advantages:

- Simple and efficient to implement.
- Works well with linear models like Naive Bayes and Logistic Regression.

Disadvantages:

- Ignores context and word order.
- Large vocabulary leads to sparse vectors.
- Doesn't capture semantics (e.g., synonyms have unrelated vectors).

3.3.2 N-Grams

N-grams are contiguous sequences of n items from a given text. They are used to model word sequences and local context.

$$\text{Unigram: } \{w_1, w_2, ..., w_T\}, \quad \text{Bigram: } \{(w_1, w_2), (w_2, w_3), ..., (w_{T-1}, w_T)\}$$

Probability Estimation:
For bigrams, the probability of a sentence is approximated using the chain rule:

$$P(w_1, w_2, ..., w_T) \approx \prod_{t=1}^{T} P(w_t | w_{t-1})$$

Smoothed Estimation:

$$P(w_t | w_{t-1}) = \frac{\text{count}(w_{t-1}, w_t) + \alpha}{\text{count}(w_{t-1}) + \alpha |V|}$$

Where α is the smoothing factor (Laplace smoothing), and $|V|$ is vocabulary size.
Pros and Cons:

- Captures some local context.
- Still suffers from data sparsity and can't model long-range dependencies.

3.3.3 Term Frequency–Inverse Document Frequency (TF-IDF)

TF-IDF improves upon raw counts by down-weighting common words (like "the", "is") and up-weighting informative ones.
Term Frequency (TF):

$$TF(t, d) = \frac{\text{count of term } t \text{ in document } d}{\text{total terms in } d}$$

Inverse Document Frequency (IDF):

$$IDF(t) = \log\left(\frac{N}{1 + |\{d : t \in d\}|}\right)$$

TF-IDF Score:

$$\text{TF-IDF}(t, d) = TF(t, d) \cdot IDF(t)$$

Advantages:

- Reduces impact of common but uninformative terms.
- Works well for document classification and information retrieval.

Limitations:

- Still based on word frequency, not semantics.
- High-dimensional and sparse representations.

3.3.4 Limitations of Traditional Methods

- No handling of word meaning (e.g., ”good” vs. ”great”).
- Struggles with polysemy and synonymy.
- Feature engineering required for downstream tasks.
- Cannot model context or sequential structure effectively.

Despite their simplicity, traditional NLP methods laid the groundwork for more advanced, neural-based techniques. In the following sections, we explore distributed representations and neural models that address these limitations.

3.4 Word Embeddings

Traditional NLP methods like Bag-of-Words and TF-IDF represent words as discrete indices in high-dimensional sparse vectors, failing to capture semantic relationships or context. **Word embeddings** are dense, low-dimensional continuous vector representations of words that preserve semantic similarity and are learned from large corpora.

3.4.1 Motivation

The goal is to represent each word w in a vocabulary V as a vector $\vec{w} \in \mathbb{R}^d$, where similar words (e.g., “king” and “queen”) have nearby vector representations:

$$\text{Similarity}(w_i, w_j) = \cos(\vec{w_i}, \vec{w_j})$$

3.4.2 Word2Vec

Proposed by Mikolov et al. (2013), Word2Vec includes two architectures:

- **Skip-Gram:** Predicts context words given the center word.
- **CBOW (Continuous Bag-of-Words):** Predicts the center word given context words.

Skip-Gram Model: Given a word sequence $w_1, w_2, ..., w_T$, the skip-gram objective maximizes the probability of context words $w_{t\pm j}$ given the center word w_t:

$$\mathcal{L}_{\text{skip-gram}} = \sum_{t=1}^{T} \sum_{-c \le j \le c, j \neq 0} \log P(w_{t+j}|w_t)$$

where c is the context window size. Using softmax:

$$P(w_O|w_I) = \frac{\exp(\vec{v}_{w_O}^\top \vec{v}_{w_I})}{\sum_{w \in V} \exp(\vec{v}_w^\top \vec{v}_{w_I})}$$

To avoid computing the full softmax denominator, techniques like **Negative Sampling** and **Hierarchical Softmax** are used.

CBOW Model: Given a context window around word w_t, CBOW predicts w_t using the average of context vectors:

$$\mathcal{L}_{\text{CBOW}} = \sum_{t=1}^{T} \log P(w_t|w_{t-c}, ..., w_{t-1}, w_{t+1}, ..., w_{t+c})$$

3.4.3 GloVe (Global Vectors)

GloVe, proposed by Pennington et al. (2014), learns embeddings by factorizing a word co-occurrence matrix. The core idea is that word meaning can be captured from global co-occurrence statistics.

Let X_{ij} be the number of times word j occurs in the context of word i. The model tries to learn word vectors $\vec{w}_i, \vec{\tilde{w}}_j \in \mathbb{R}^d$ such that:

$$\vec{w}_i^\top \vec{\tilde{w}}_j + b_i + \tilde{b}_j = \log(X_{ij})$$

Loss Function:

$$\mathcal{L}_{\text{GloVe}} = \sum_{i,j=1}^{|V|} f(X_{ij}) \left(\vec{w}_i^\top \vec{\tilde{w}}_j + b_i + \tilde{b}_j - \log X_{ij}\right)^2$$

where $f(X_{ij})$ is a weighting function:

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

Typically, $\alpha = 0.75$, $x_{\max} = 100$.

3.4.4 Properties of Word Embeddings

- Capture semantic and syntactic similarity:

$$\text{cosine}(\vec{w}_{\text{king}}, \vec{w}_{\text{queen}}) \approx \text{cosine}(\vec{w}_{\text{man}}, \vec{w}_{\text{woman}})$$

- Enable vector arithmetic:

$$\vec{w}_{\text{king}} - \vec{w}_{\text{man}} + \vec{w}_{\text{woman}} \approx \vec{w}_{\text{queen}}$$

- Dense representations reduce sparsity and generalize better than BoW.

3.4.5 Limitations

- Fixed embeddings: do not adapt based on sentence context.
- Struggle with polysemy (e.g., "bank" as a financial institution vs. river bank).
- Cannot capture compositionality or long-term dependencies.

These limitations led to the development of **contextual embeddings** such as ELMo, BERT, and GPT, which are covered in the next sections.

3.5 Recurrent Neural Networks (RNNs) in NLP

Recurrent Neural Networks (RNNs) are a class of neural networks designed for sequential data, making them highly suitable for Natural Language Processing tasks such as language modeling, text generation, machine translation, and named entity recognition.

3.5.1 Motivation

Most natural language tasks require modeling dependencies between words. Unlike feed-forward networks, RNNs maintain a **hidden state** that captures information from previous time steps, allowing them to model sequences of arbitrary length.

$$\text{Example: } P(w_1, w_2, ..., w_T) = \prod_{t=1}^{T} P(w_t | w_1, ..., w_{t-1})$$

3.5.2 RNN Architecture and Equations

At each time step t, an RNN takes an input vector $\vec{x}_t \in \mathbb{R}^d$ and updates a hidden state $\vec{h}_t \in \mathbb{R}^h$ using the following recurrence relation:

$$\vec{h}_t = \tanh(W_{xh}\vec{x}_t + W_{hh}\vec{h}_{t-1} + \vec{b}_h)$$
$$\vec{y}_t = W_{hy}\vec{h}_t + \vec{b}_y$$

Where:

- W_{xh} is the input-to-hidden weight matrix.
- W_{hh} is the hidden-to-hidden (recurrent) weight matrix.
- W_{hy} is the hidden-to-output weight matrix.
- $\vec{h}_0$ is typically initialized to zero.

The model is trained to minimize a loss function such as cross-entropy over a sequence of predictions.

3.5.3 Backpropagation Through Time (BPTT)

To train RNNs, the gradients are computed through all time steps via a technique called **Backpropagation Through Time (BPTT)**. The loss is summed over all time steps:

$$\mathcal{L} = \sum_{t=1}^{T} \ell(\vec{y}_t, \vec{y}_t^{\text{true}})$$

Gradients are backpropagated through the unrolled network. However, this leads to issues with:

- **Vanishing Gradients:** Gradients shrink as they are backpropagated, making it hard to learn long-range dependencies.
- **Exploding Gradients:** Gradients grow exponentially and destabilize training.

3.5.4 Applications of RNNs in NLP

- **Language Modeling:** Predicting the next word given a sequence.
- **Machine Translation:** Encoder-decoder RNNs map source sentences to target sentences.
- **Speech Recognition and Tagging:** Sequence labeling for input data like audio or tokens.
- **Text Generation:** Sampling text one word at a time based on learned distributions.

3.5.5 Bidirectional RNNs (BiRNN)

Standard RNNs process sequences in one direction (past $\rightarrow$ future). **Bidirectional RNNs** use two RNNs: one processes the input forward, and another backward, combining both hidden states:

$$\vec{h}_t = [\overrightarrow{h}_t; \overleftarrow{h}_t]$$

This allows access to both past and future context, improving performance on tasks like Named Entity Recognition or POS tagging.

3.5.6 Limitations of Vanilla RNNs

- Cannot retain long-term dependencies due to vanishing gradient problem.
- Sequential computation inhibits parallelization during training.
- Poor at capturing hierarchical sentence structures.

These limitations motivated the development of improved architectures like LSTM (Long Short-Term Memory) and GRU (Gated Recurrent Unit), which we will explore next.

3.6 Long Short-Term Memory (LSTM)

Long Short-Term Memory (LSTM) networks are a special type of Recurrent Neural Networks (RNNs) designed to address the vanishing and exploding gradient problems in standard RNNs. LSTMs are particularly effective at learning long-term dependencies, making them well-suited for many NLP tasks.

3.6.1 Motivation

Traditional RNNs struggle with learning patterns that span long distances in a sequence due to the repeated multiplication of gradients through time. LSTMs solve this by introducing an explicit memory cell and gating mechanisms that control the flow of information.

Key Idea: Maintain a memory cell $\vec{c}_t$ whose updates are regulated by learnable gates.

3.6.2 LSTM Cell Structure

An LSTM cell contains:

- **Forget Gate** $\vec{f}_t$
- **Input Gate** $\vec{i}_t$
- **Output Gate** $\vec{o}_t$
- **Cell State** $\vec{c}_t$
- **Hidden State** $\vec{h}_t$

Inputs:

$$\vec{x}_t \in \mathbb{R}^d \quad \text{(input vector)}, \quad \vec{h}_{t-1}, \vec{c}_{t-1} \in \mathbb{R}^h \quad \text{(previous hidden and cell states)}$$

3.6.3 LSTM Equations

$$\vec{f}_t = \sigma(W_f \vec{x}_t + U_f \vec{h}_{t-1} + \vec{b}_f) \quad \text{(Forget Gate)}$$
$$\vec{i}_t = \sigma(W_i \vec{x}_t + U_i \vec{h}_{t-1} + \vec{b}_i) \quad \text{(Input Gate)}$$
$$\vec{\tilde{c}}_t = \tanh(W_c \vec{x}_t + U_c \vec{h}_{t-1} + \vec{b}_c) \quad \text{(Cell Candidate)}$$
$$\vec{c}_t = \vec{f}_t \odot \vec{c}_{t-1} + \vec{i}_t \odot \vec{\tilde{c}}_t \quad \text{(New Cell State)}$$
$$\vec{o}_t = \sigma(W_o \vec{x}_t + U_o \vec{h}_{t-1} + \vec{b}_o) \quad \text{(Output Gate)}$$
$$\vec{h}_t = \vec{o}_t \odot \tanh(\vec{c}_t) \quad \text{(New Hidden State)}$$

Where:

- σ is the sigmoid activation function.
- tanh is the hyperbolic tangent function.
- $\odot$ denotes element-wise multiplication.
- W_*, U_*, b_* are learnable parameters.

3.6.4 Intuition Behind the Gates

- **Forget Gate $\vec{f}_t$:** Decides what portion of the previous memory to forget.
- **Input Gate $\vec{i}_t$:** Determines how much of the new candidate memory to add.
- **Cell Update $\vec{c}_t$:** Combines old and new memory.
- **Output Gate $\vec{o}_t$:** Controls what part of the memory to output as hidden state.

3.6.5 Applications in NLP

LSTMs are widely used in:

- **Text Classification**
- **Language Modeling**
- **Named Entity Recognition (NER)**
- **Machine Translation (with Encoder–Decoder frameworks)**
- **Speech Recognition**

3.6.6 Bidirectional LSTMs

Like RNNs, LSTMs can be made bidirectional:

$$\vec{h}_t = [\overrightarrow{h}_t; \overleftarrow{h}_t]$$

where $\overrightarrow{h}_t$ is from forward LSTM and $\overleftarrow{h}_t$ is from backward LSTM.

3.6.7 Advantages over RNNs

- Better at capturing long-term dependencies.
- Mitigates vanishing gradient problem.
- More expressive due to gating mechanisms.

8. Limitations

- Computationally intensive due to multiple gates.
- Still sequential—cannot fully parallelize across time.
- Less effective than Transformers on long sequences.

These limitations motivated the development of attention mechanisms and Transformer architectures, which are covered in the next section.

3.7 Attention Mechanisms in NLP

Attention mechanisms have revolutionized Natural Language Processing by enabling models to dynamically focus on relevant parts of an input sequence when generating an output. Unlike traditional sequence models that compress entire sequences into a fixed-size vector, attention allows flexible and context-aware alignment between input and output tokens.

3.7.1 Motivation

In tasks such as machine translation or summarization, not all input words are equally important for producing a given output word. Traditional RNNs and LSTMs often suffer from the bottleneck of encoding the entire sequence into a single context vector.

Attention addresses this by allowing the model to look back at the entire input sequence.

3.7.2 General Form of Attention

Given a query $\vec{q}$, a set of key-value pairs $\{(\vec{k}_i, \vec{v}_i)\}_{i=1}^{n}$, attention computes a weighted sum of the values:

$$\text{Attention}(\vec{q}, K, V) = \sum_{i=1}^{n} \alpha_i \vec{v}_i$$

Where the weights α_i are computed via a similarity function between the query and each key:

$$\alpha_i = \frac{\exp(\text{score}(\vec{q}, \vec{k}_i))}{\sum_{j=1}^{n} \exp(\text{score}(\vec{q}, \vec{k}_j))}$$

3.7.3 Scoring Functions

Common choices for the score function:

- Dot-product: $\text{score}(\vec{q}, \vec{k}_i) = \vec{q}^\top \vec{k}_i$
- Additive (Bahdanau): $\text{score}(\vec{q}, \vec{k}_i) = \vec{v}_a^\top \tanh(W_q \vec{q} + W_k \vec{k}_i)$

3.7.4 Types of Attention

- **Additive Attention (Bahdanau et al., 2015):** Introduced in neural machine translation using encoder-decoder models with alignment scores.
- **Dot-Product Attention (Luong et al., 2015):** Uses the dot product between query and key vectors for simplicity and efficiency.
- **Scaled Dot-Product Attention:** Used in Transformers, scales the dot product by $\sqrt{d_k}$ to stabilize gradients:

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^\top}{\sqrt{d_k}}\right) V$$

3.7.5 Self-Attention

In self-attention, the queries, keys, and values all come from the same source sequence:

$$Q = XW^Q, \quad K = XW^K, \quad V = XW^V$$

Where:

- $X \in \mathbb{R}^{n \times d}$: sequence of token embeddings
- $W^Q, W^K, W^V \in \mathbb{R}^{d \times d_k}$: learned projections

Self-attention captures dependencies between all tokens in the sequence regardless of their position.

3.7.6 Multi-Head Attention

To allow the model to attend to information from different representation subspaces, multiple attention heads are used:

$$\text{MultiHead}(Q, K, V) = \text{Concat}(\text{head}_1, ..., \text{head}_h)W^O$$

Each head is computed as:

$$\text{head}_i = \text{Attention}(QW_i^Q, KW_i^K, VW_i^V)$$

This enhances the model's ability to capture various aspects of similarity and relevance across positions.

3.7.7 Applications in NLP

- **Machine Translation:** Alignment between source and target sentences.
- **Text Summarization:** Focusing on key parts of a document.
- **Question Answering:** Attending to relevant context for accurate answers.
- **Transformers:** The foundation of models like BERT, GPT, T5, etc.

3.7.8 Advantages of Attention

- Removes fixed-length bottleneck in sequence encoding.
- Enables parallelization unlike RNNs.
- Improves long-range dependency modeling.

Attention paved the way for the development of the **Transformer** architecture, which completely replaces recurrence with stacked self-attention layers.

Chapter 4

Transformer Architecture

4.1 Introduction to Transformers

The Transformer architecture was introduced in the groundbreaking paper *"Attention Is All You Need"* by Vaswani et al. in 2017. It marked a paradigm shift in the field of Natural Language Processing (NLP), moving away from recurrent and convolutional structures, and relying solely on a novel mechanism called **self-attention**.

4.1.1 Motivation Behind Transformers

Traditional sequence models such as RNNs and LSTMs processed data sequentially, making it difficult to parallelize computations and capture long-range dependencies. Transformers were introduced to overcome the following limitations:

- **Limited Long-Range Context:** RNNs often struggle with long-term dependencies due to vanishing or exploding gradients.
- **Sequential Computation:** Each timestep in RNNs must wait for the previous computation, leading to slow training.
- **Lack of Global Context Awareness:** CNNs only capture local patterns unless stacked deeply.

Transformers address these issues using a self-attention mechanism that allows every input token to directly attend to all others, regardless of their position in the sequence.

4.1.2 Core Principles

The Transformer model is built on three core ideas:

1. **Self-Attention:** Enables each word in a sequence to focus on other relevant words.
2. **Positional Encoding:** Injects sequential information into word embeddings.
3. **Parallelization:** Entire sequences can be processed simultaneously, accelerating training.

4.1.3 Basic Transformer Block Overview

A Transformer consists of an **encoder-decoder** architecture. Each block of the encoder and decoder contains:

- Multi-head self-attention mechanism
- Feedforward neural network
- Residual connections and layer normalization

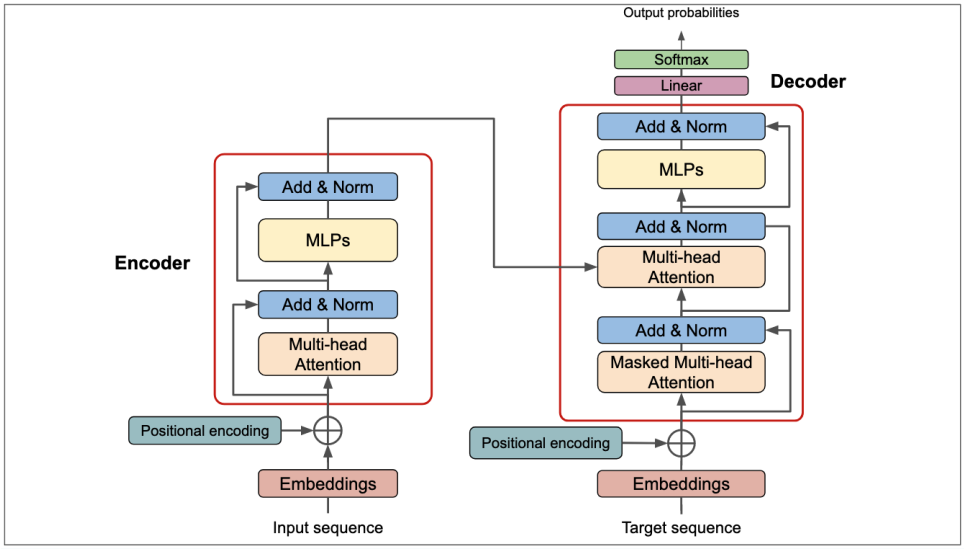

Figure 4.1: Basic Transformer Architecture showing Encoder and Decoder Stack

4.1.4 Applications of Transformers

Transformers have become foundational in various tasks:

- **Machine Translation** (e.g., English to French)
- **Text Summarization**
- **Question Answering**

4.1.5 Evolution of Transformer-Based Models

Following the original architecture, various adaptations have emerged:

- **BERT:** Bidirectional Encoder Representations from Transformers
- **GPT:** Generative Pre-trained Transformer (decoder-only)
- **T5:** Text-To-Text Transfer Transformer
- **Vision Transformers (ViT):** Transformer models applied to image patches

These models form the backbone of modern AI applications, especially large language models (LLMs) like GPT-3 and GPT-4.

4.2 Self-Attention Mechanism

Self-attention, also known as **intra-attention**, is a mechanism that allows the model to relate different positions of a single input sequence to compute a richer representation of each token. It is the fundamental building block of the Transformer model and enables capturing contextual dependencies between all tokens, regardless of their relative distances.

4.2.1 Intuition Behind Self-Attention

Consider the sentence: "The cat sat on the mat because it was tired."

The word "it" refers to "the cat." To resolve this dependency, a model needs to know that "it" is related to "cat," which may be several words away. Self-attention allows the model to learn such relationships by computing how much attention each word should pay to every other word in the sequence.

4.2.2 Mathematical Formulation

Given an input sequence of n tokens represented as embeddings $X = [x_1, x_2, \ldots, x_n] \in \mathbb{R}^{n \times d_{model}}$, the self-attention mechanism maps each token to a new representation z_i using three learned projections: queries (Q), keys (K), and values (V).

Query, Key, Value Vectors

Each input vector x_i is transformed into three vectors:

$$Q = XW^Q, \quad K = XW^K, \quad V = XW^V$$

Where:

- $W^Q, W^K, W^V \in \mathbb{R}^{d_{model} \times d_k}$ are learnable weight matrices.
- $Q, K, V \in \mathbb{R}^{n \times d_k}$ are the resulting matrices.

Scaled Dot-Product Attention

For a single query q_i (a row from Q), the attention over all keys K is computed as:

$$\text{Attention}(q_i, K, V) = \sum_{j=1}^{n} \alpha_{ij} v_j$$

Where the attention weights α_{ij} are obtained using:

$$\alpha_{ij} = \frac{\exp\left(q_i \cdot k_j / \sqrt{d_k}\right)}{\sum_{l=1}^{n} \exp\left(q_i \cdot k_l / \sqrt{d_k}\right)}$$

This can be written in matrix form for all tokens:

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^\top}{\sqrt{d_k}}\right) V$$

Explanation:

- $QK^\top$ gives pairwise dot-products (similarities) between all query-key pairs.
- Scaling by $\sqrt{d_k}$ helps maintain stable gradients.
- Softmax ensures attention weights are normalized.

4.2.3 Multi-Head Self-Attention

Using only one attention mechanism might limit the model's ability to focus on multiple aspects of the input. To address this, the Transformer uses **multi-head attention**, where the attention mechanism is performed h times in parallel (heads), each with its own learnable projections.

$$\text{MultiHead}(Q, K, V) = \text{Concat}(\text{head}_1, \ldots, \text{head}_h)W^O$$
$$\text{where head}_i = \text{Attention}(QW_i^Q, KW_i^K, VW_i^V)$$

Where:

- W_i^Q, W_i^K, W_i^V are learned projections for head i.
- $W^O \in \mathbb{R}^{hd_k \times d_{model}}$ projects the concatenated output back to the model dimension.

4.2.4 Benefits of Self-Attention

- **Global Context Awareness:** Each token can attend to all other tokens.
- **Position-Invariant Representations:** Attention does not assume sequential order.
- **Highly Parallelizable:** Unlike RNNs, attention allows all inputs to be processed simultaneously.
- **Interpretability:** Attention weights can be visualized to understand which parts of the input influence a token's representation.

4.2.5 Limitations and Extensions

While self-attention is powerful, it also has limitations:

- **Quadratic Complexity:** Time and space complexity is $O(n^2)$ due to computing attention between all pairs.
- **No Recurrence or Convolution:** Requires positional encodings to model order.

Several extensions have been proposed to address these, such as:

- **Linformer, Performer:** Reduce complexity to linear time.
- **Relative Positional Encoding:** Captures relative rather than absolute positions.
- **Sparse Attention:** Limits attention to local windows or selected tokens.

4.3 Positional Encoding

The Transformer architecture lacks any inherent mechanism to process sequential information since it treats inputs as sets rather than sequences. Unlike RNNs or CNNs, it has no built-in notion of word order or locality.

To enable Transformers to model sequence order, positional information must be explicitly added to the token embeddings. This is achieved through a technique called **positional encoding**.

4.3.1 Why Positional Encoding?

In natural language, the meaning of a sentence heavily depends on the order of the words. For example:

- "The dog chased the cat." vs. "The cat chased the dog."

While RNNs preserve sequence order through time steps, Transformers operate on the entire input sequence simultaneously. Hence, additional information is needed to help the model distinguish between different word positions.

4.3.2 Adding Position to Embeddings

Let $X = [x_1, x_2, ..., x_n] \in \mathbb{R}^{n \times d_{model}}$ be the input embeddings. The positional encodings $PE \in \mathbb{R}^{n \times d_{model}}$ are added element-wise:

$$Z = X + PE$$

This combined representation Z is then fed into the encoder or decoder layers.

4.3.3 Sinusoidal Positional Encoding

In the original Transformer paper, Vaswani et al. proposed using a fixed sinusoidal function to generate positional encodings:

$$PE_{(pos,2i)} = \sin\left(\frac{pos}{10000^{2i/d_{model}}}\right)$$

$$PE_{(pos,2i+1)} = \cos\left(\frac{pos}{10000^{2i/d_{model}}}\right)$$

Where:

- pos is the position in the sequence (starting from 0).
- i is the dimension index.
- d_{model} is the dimensionality of the embedding space.

Properties of Sinusoidal Encoding

- Each dimension of the encoding corresponds to a sinusoid of a different frequency.
- Allows the model to learn to attend by relative positions, since any linear function of positions can be expressed as a linear function of these encodings.
- Encodings are fixed and do not require additional parameters.

4.3.4 Learned Positional Embeddings

An alternative to sinusoidal encoding is to learn the positional embeddings just like token embeddings. This means using a learnable matrix $P \in \mathbb{R}^{n \times d_{model}}$ such that:

$$Z = X + P$$

Comparison:

- **Fixed Encodings:** Generalize better to unseen lengths; no training overhead.
- **Learned Encodings:** Can adapt better to specific datasets; do not generalize beyond training length.

4.3.5 Relative Positional Encodings (Advanced)

Recent Transformer variants (e.g., Transformer-XL, T5) use **relative positional encodings**, which encode the distance between tokens instead of their absolute positions.

$$A_{ij} = f(i - j)$$

This allows the model to attend to relative positions rather than fixed locations, improving performance on longer sequences and generalization.

4.4 Encoder-Decoder Structure

The Transformer architecture follows an **encoder-decoder** structure, designed to transform an input sequence into an output sequence. This design is particularly useful for sequence transduction tasks such as machine translation, summarization, and question answering.

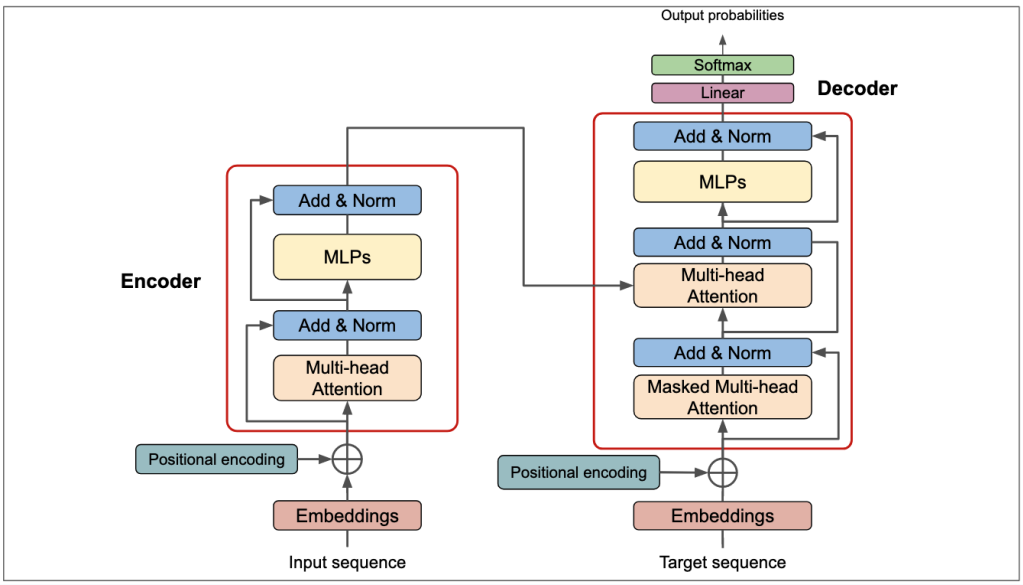

Figure 4.2: High-level view of the Transformer architecture showing encoder and decoder stacks

4.4.1 Encoder Stack

The encoder consists of a stack of N identical layers (typically $N = 6$). Each layer has two sublayers:

1. Multi-head self-attention mechanism
2. Position-wise fully connected feed-forward network

Each sublayer is surrounded by residual connections and followed by layer normalization:

$$\text{LayerOutput} = \text{LayerNorm}(X + \text{Sublayer}(X))$$

Multi-head Self-Attention

All tokens in the input sequence attend to one another to form contextualized representations:

$$\text{SelfAttention}(Q, K, V) = \text{softmax}\left(\frac{QK^\top}{\sqrt{d_k}}\right)V$$

Feed-Forward Network (FFN)

A position-wise feedforward network is applied to each position separately and identically:

$$\text{FFN}(x) = \max(0, xW_1 + b_1)W_2 + b_2$$

4.4.2 Decoder Stack

The decoder also consists of N layers and has three sublayers:

1. Masked multi-head self-attention mechanism
2. Multi-head attention over encoder output
3. Feed-forward network

Like the encoder, each sublayer is followed by residual connections and layer normalization.

Masked Self-Attention

The self-attention in the decoder is **masked** to prevent positions from attending to future tokens. This ensures the model generates output autoregressively, one token at a time.

The attention mask is a triangular matrix that sets future positions to $-\infty$ before applying softmax:

$$\text{MaskedAttention}(Q, K, V) = \text{softmax}\left(\frac{QK^\top}{\sqrt{d_k}} + M\right)V$$

Where $M_{ij} = -\infty$ if $j > i$, otherwise 0.

Encoder-Decoder Attention

The second attention sublayer allows each decoder position to attend to all positions in the encoder output. This helps the decoder condition its predictions on the full input sequence.

4.4.3 Output Prediction Layer

The output from the final decoder layer is passed through a linear transformation followed by a softmax to generate token probabilities:

$$\hat{y}_t = \text{softmax}(W_o h_t + b_o)$$

Where:

- h_t is the decoder hidden state at timestep t
- $W_o \in \mathbb{R}^{d_{model} \times |V|}$ is the output projection matrix
- $|V|$ is the vocabulary size

4.4.4 Residual Connections and Layer Normalization

Each sublayer in both encoder and decoder is wrapped with a residual connection followed by layer normalization:

$$\text{Output} = \text{LayerNorm}(x + \text{Sublayer}(x))$$

This stabilizes training and improves gradient flow through deep layers.

4.4.5 Training Objective

The model is trained to minimize the cross-entropy loss between the predicted token probabilities and the actual target sequence:

$$\mathcal{L} = -\sum_{t=1}^{T} \log P(y_t \mid y_{<t}, X)$$

Where:

- X is the input sequence
- y_t is the true token at timestep t
- $y_{<t}$ is the sequence of previously generated tokens

4.4.6 Summary

The encoder-decoder architecture enables the Transformer to process complex input-output sequence mappings with high parallelism. The encoder builds rich contextual representations of the input, while the decoder uses these to autoregressively generate the output.

This architecture serves as the foundation for advanced models like:

- **T5:** Text-to-text Transfer Transformer
- **BART:** Bidirectional and Auto-Regressive Transformer
- **GPT:** Decoder-only Transformer

Chapter 5

Applications of LLMs in Generative AI

5.1 Introduction to Generative AI

Generative AI refers to the class of artificial intelligence systems capable of generating novel content. These models can produce text, images, music, and even code. Large Language Models (LLMs), particularly transformer-based architectures, have become the cornerstone of generative AI due to their capability to generate coherent and contextually relevant text.

Generative AI systems rely on probabilistic modeling of data. In the case of LLMs, they model the probability distribution of sequences of words:

$$P(w_1, w_2, ..., w_n) = \prod_{t=1}^{n} P(w_t | w_1, ..., w_{t-1})$$

This enables them to generate new sequences by sampling from these distributions.

5.2 Chatbot Development using LLMs

LLMs can be fine-tuned or used in a zero-shot fashion to build intelligent conversational agents, or chatbots. These chatbots can understand context, maintain coherence across turns, and simulate human-like interactions.

Key Features Enabled by LLMs

- **Contextual Understanding**: Using self-attention, LLMs capture dependencies across sentences.
- **Memory**: Transformers can reference earlier parts of the conversation for better continuity.
- **Multi-turn Dialogs**: Chatbots can maintain state and handle multi-turn conversations effectively.

Example

A user inputs: `\What is the capital of France?"`
The LLM generates: `\The capital of France is Paris."`

5.3 Prompt Engineering

Prompt engineering is the practice of designing inputs (prompts) to LLMs that guide them to produce desired outputs. The effectiveness of LLM responses can be significantly improved by crafting suitable prompts.

5.3.1 Principles of Prompt Design

Some of the best practices include:

- Being specific and clear.
- Providing format examples (instruction tuning).
- Including role-based prompts (e.g., `\You are a legal advisor..."`).

5.3.2 Zero-shot, One-shot, Few-shot Learning

- **Zero-shot**: The model is given only a task description and asked to perform without any examples.

  Prompt: `Translate \hello" to Spanish.`

  Output: `\hola"`

- **One-shot**: A single example is provided before asking the model to complete the task.
- **Few-shot**: A few examples are given, helping the model generalize the task.

  Few-shot prompting enables in-context learning:

$$P(y|x_1, y_1, ..., x_k, y_k, x_{k+1}) \approx f_{\text{LLM}}(x_1, y_1, ..., x_k, y_k, x_{k+1})$$

5.4 Fine-tuning LLMs

Fine-tuning refers to continuing the training of a pretrained language model on a more specific dataset to adapt it to a particular task.

5.4.1 Pretraining vs Fine-tuning

- **Pretraining**: The model is trained on large-scale, generic text corpora to learn general language representations.
- **Fine-tuning**: The pretrained model is further trained on task-specific or domain-specific data.

5.4.2 Transfer Learning in LLMs

Transfer learning allows knowledge from the pretraining phase to transfer to downstream tasks, significantly reducing the amount of data and compute needed.

Mathematically:

$$\text{Loss}_{\text{total}} = \text{Loss}_{\text{pretrain}} + \lambda \cdot \text{Loss}_{\text{fine-tune}}$$

where λ balances the influence of both losses.

5.4.3 Techniques for Fine-tuning

- **Full fine-tuning**: All weights of the model are updated.
- **Adapter-based tuning**: Lightweight layers are added and only they are trained.
- **LoRA (Low-Rank Adaptation)**: Updates are done using low-rank matrices to save memory.

Example Use Case

Fine-tuning a GPT model on medical conversations enables building a domain-specific assistant for doctors.

Chapter 6

Advanced LLM Architectures

6.1 Introduction to Advanced LLMs

Large Language Models (LLMs) have seen exponential growth in size and capability, leading to groundbreaking models like BERT, T5, and GPT-3. These models demonstrate remarkable performance across various NLP tasks due to sophisticated architectures, training strategies, and the availability of large-scale data and computational resources.

6.2 BERT: Bidirectional Encoder Representations from Transformers

6.2.1 Introduction

BERT (Bidirectional Encoder Representations from Transformers), introduced by Devlin et al. in 2018, marked a major advancement in NLP by introducing deep bidirectional training of Transformer encoders. Unlike earlier models that processed text unidirectionally (left-to-right or right-to-left), BERT learns representations that incorporate both left and right context, leading to state-of-the-art results across various NLP benchmarks.

6.2.2 Architecture Overview

BERT is built solely on the encoder part of the original Transformer architecture. Its core components include:

- **Multi-Head Self-Attention:** Enables the model to attend to different positions in the input sequence simultaneously.
- **Feedforward Layers:** Position-wise fully connected networks applied after attention.
- **Residual Connections and Layer Normalization:** Improve gradient flow and model stability.

Two common model sizes were released:

- $\textbf{BERT}_{\textbf{BASE}}$: 12 layers, 768 hidden units, 12 attention heads (110M parameters).
- $\textbf{BERT}_{\textbf{LARGE}}$: 24 layers, 1024 hidden units, 16 attention heads (340M parameters).

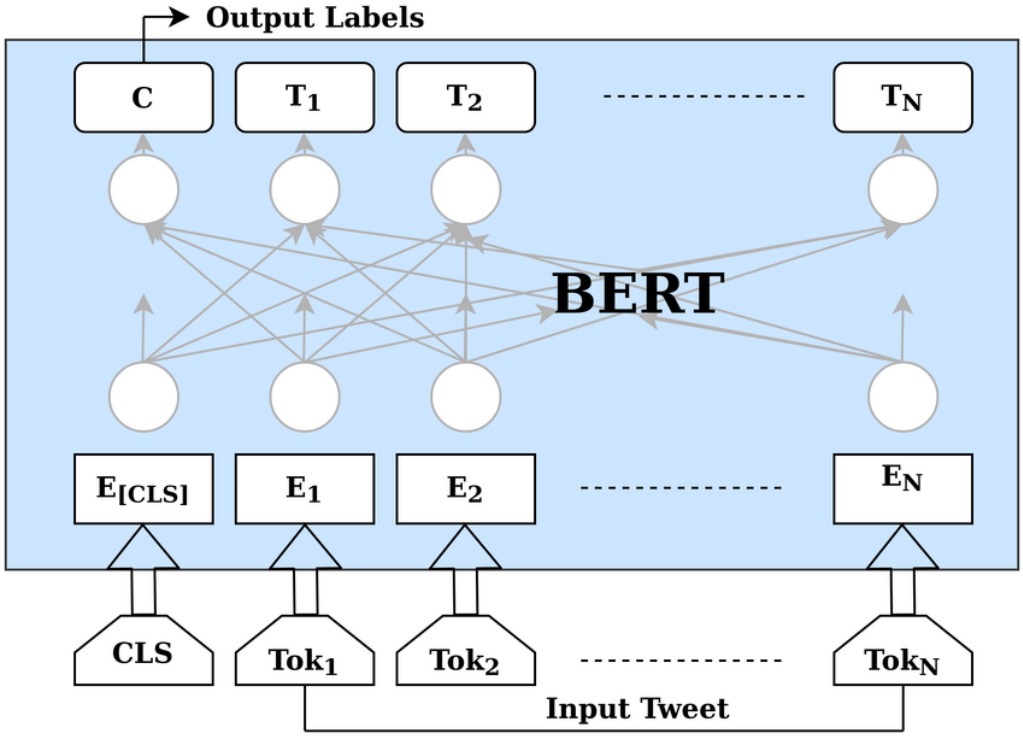

Figure 6.1: Basic Architeture of BERT Model

6.2.3 Pretraining Objectives

BERT is pretrained using two self-supervised tasks:

1. Masked Language Modeling (MLM)

Randomly masks 15% of input tokens and trains the model to predict them using the surrounding context.

Example:

- Input: `The cat sat on the [MASK].`
- Target: `mat`

Note: 80% of the time a token is replaced with `[MASK]`, 10% with a random word, and 10% is left unchanged.

2. Next Sentence Prediction (NSP)

Given two segments A and B, the model predicts whether B is the actual next sentence after A.

Example:

- Sentence A: `The man went to the store.`
- Sentence B: `He bought a gallon of milk.` ⇒ `IsNext`

6.2.4 Tokenization and Input Formatting

BERT uses WordPiece tokenization to handle rare and subword units. It processes inputs as:

`[CLS]` `Token_1` `Token_2` ... `[SEP]` `Token_A` ... `[SEP]`

- `[CLS]`: Special classification token prepended to every input.
- `[SEP]`: Separator token used between sentence pairs.
- Segment embeddings and positional embeddings are added to distinguish tokens.

6.2.5 Fine-tuning for Downstream Tasks

Once pretrained, BERT can be fine-tuned for various NLP tasks by simply adding a task-specific output layer:

- **Text Classification:** Add a softmax layer on top of the `[CLS]` embedding.
- **Question Answering:** Add two output vectors to predict the start and end positions in the input.
- **Named Entity Recognition (NER):** Perform token-level classification using each output embedding.

Fine-tuning typically involves training on smaller task-specific datasets for a few epochs with minimal changes to the model architecture.

6.2.6 Applications and Performance

BERT achieved state-of-the-art results on multiple benchmarks:

- GLUE (General Language Understanding Evaluation)
- SQuAD (Stanford Question Answering Dataset)
- MNLI (Multi-Genre Natural Language Inference)

Its release initiated widespread adoption of Transformer-based models in production systems for search, dialogue, classification, and QA.

6.2.7 Limitations

- **Masked Language Modeling Gap:** The `[MASK]` token is never seen at fine-tuning time, creating a train-test mismatch.
- **Input Length Limitations:** Maximum input size is typically 512 tokens.
- **No Decoder:** Not suitable for generative tasks (e.g., summarization, translation).
- **Computational Cost:** Large parameter size and self-attention make inference and training resource-intensive.

6.3 T5: Text-To-Text Transfer Transformer

6.3.1 Introduction

The Text-to-Text Transfer Transformer (T5), introduced by Google Research, represents a unifying approach to NLP by converting all tasks—classification, translation, summarization, and more—into a single text-to-text format. This formulation enables a consistent training and inference procedure across diverse tasks, facilitating better generalization.

6.3.2 Unified Text-to-Text Framework

Unlike other models that handle tasks differently (e.g., classification using logits, translation using sequence-to-sequence), T5 reformulates every NLP problem as:

$$\text{Input: "task\_name: input text"} \quad \rightarrow \quad \text{Output: "target text"}$$

Examples:

- Translation: `translate English to German: How are you?` → `Wie geht es dir?`
- Sentiment Classification: `sst2 sentence: I love this movie.` → `positive`
- Summarization: `summarize: The book is about...` → `A summary sentence.`

This uniform representation simplifies model design and training by reducing task-specific variation.

6.3.3 Architecture Overview

T5 is based on the standard Transformer encoder-decoder architecture introduced by Vaswani et al. It retains all key components like multi-head self-attention, feedforward networks, and layer normalization.

- **Encoder:** Processes the input sequence (with task prefix) and encodes contextual representations.
- **Decoder:** Generates the output sequence auto-regressively, attending to encoder outputs and previous decoder tokens.

The model was pretrained and evaluated in five sizes: T5-Small, T5-Base, T5-Large, T5-3B, and T5-11B.

6.3.4 Pretraining Objectives

T5 uses a novel pretraining objective called **Span Corruption**:

1. Random spans of the input are masked.
2. Each masked span is replaced with a unique sentinel token (e.g., $\langle extra\_id\_0 \rangle$).

3. The model is trained to reconstruct the masked content in order.

   Example:

   - Input: `The dog` ⟨*extra_id_0*⟩ `to the park` ⟨*extra_id_1*⟩
   - Target: ⟨*extra_id_0*⟩ `ran` ⟨*extra_id_1*⟩ `yesterday`

This corruption-based objective encourages learning robust representations for both local and global dependencies.

6.3.5 Transfer Learning and Fine-tuning

After pretraining on the C4 dataset (Colossal Clean Crawled Corpus), T5 is fine-tuned on a wide range of tasks:

- GLUE benchmark for classification and entailment.
- CNN/DailyMail for summarization.
- WMT datasets for translation.
- SQuAD for question answering.

The uniform text-to-text structure simplifies multitask fine-tuning and reduces the need for task-specific heads or architectures.

6.3.6 Advantages and Limitations

Advantages:

- Unified architecture for all NLP tasks.
- Pretrained on a large and clean corpus (C4).
- Flexible and extensible to new tasks.

Limitations:

- Requires large computational resources, especially for T5-3B and T5-11B.
- Span corruption pretraining may be less effective on tasks requiring structured or tabular reasoning.
- Input length is constrained by Transformer's quadratic attention complexity.

6.4 Generative Pretrained Transformer 3 (GPT-3)

GPT-3 (Generative Pretrained Transformer 3), developed by OpenAI, represents a major milestone in the evolution of large language models. As the third iteration in the GPT series, it significantly increased the scale of parameters and the quality of generated text, setting new standards for natural language understanding and generation.

6.4.1 Architecture Overview

GPT-3 is based on the Transformer decoder architecture. It consists of a stack of identical decoder blocks, each comprising:

- A masked multi-head self-attention mechanism.
- A position-wise feedforward neural network.
- Residual connections and layer normalization.

The total model size is approximately **175 billion parameters**, making it one of the largest publicly known models at the time of its release.

Input Representation

Each input token is embedded using a learned token embedding matrix and added with a positional embedding:

$$x_i = \text{TokenEmbedding}(t_i) + \text{PositionalEmbedding}(i)$$

where t_i is the i-th token in the sequence.

Causal Masking

GPT-3 uses **causal masking** to ensure autoregressive behavior. In self-attention, a token at position i can only attend to tokens at positions $\leq i$, enforcing the left-to-right generation paradigm.

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}} + M\right)V$$

where M is a mask matrix with $M_{ij} = -\infty$ for $j > i$.

6.4.2 Training Paradigm

GPT-3 is trained using the standard language modeling objective:

$$\mathcal{L} = -\sum_{t=1}^{T} \log P(x_t | x_{<t})$$

where the model learns to predict the next token given the previous ones in a sequence.

Training was done on a diverse corpus of text from the web, books, Wikipedia, and code repositories, spanning hundreds of billions of tokens.

6.4.3 Few-Shot, One-Shot, and Zero-Shot Learning

A major innovation in GPT-3 is its ability to perform tasks with very little supervision:

- **Zero-shot learning:** Performing tasks without any examples.
- **One-shot learning:** Performing tasks with only one example.
- **Few-shot learning:** Performing tasks with a few examples embedded in the prompt.

This is made possible through *in-context learning*, where task instructions and examples are provided as part of the input prompt.

6.4.4 Strengths and Capabilities

GPT-3 demonstrated impressive capabilities across multiple domains:

- Natural language generation (stories, poems, articles).
- Translation and summarization.
- Code generation (e.g., Python, JavaScript).
- Question answering and dialogue systems.
- Prompt-based task solving without parameter updates.

6.4.5 Limitations and Criticism

Despite its success, GPT-3 also presents several limitations:

- **Bias and toxicity:** Can generate biased, inappropriate, or harmful content.
- **Factual inaccuracy:** May hallucinate facts or generate misleading text.
- **Resource cost:** Extremely expensive to train and deploy.
- **Lack of reasoning:** Struggles with complex logical or mathematical reasoning.

6.4.6 Ethical Considerations

OpenAI placed constraints on GPT-3's access through an API to mitigate risks of misuse. Ethical concerns remain around misinformation, AI-generated fake content, and reinforcement of social biases.

6.4.7 Impact and Future

GPT-3 catalyzed the generative AI boom, influencing applications in education, content creation, software engineering, and conversational AI. It also inspired the development of more advanced models like ChatGPT and GPT-4, and reinforced the importance of alignment, interpretability, and control in future LLM research.

6.5 Evaluation of LLMs

6.5.1 Benchmarks and Metrics

Common evaluation metrics include:

- **Accuracy, F1, BLEU, ROUGE, Perplexity**
- Benchmark suites: GLUE, SuperGLUE, SQuAD, MMLU

6.5.2 Human Evaluation

Human judges are often required to evaluate:

- Coherence and fluency
- Appropriateness and helpfulness
- Factual accuracy

6.6 Ethical Considerations and Responsible AI

6.6.1 Bias and Fairness

LLMs often reflect societal biases in their training data, leading to:

- Gender, racial, and cultural stereotypes
- Discriminatory outcomes in downstream tasks

6.6.2 Misuse and Misinformation

LLMs can be used to:

- Generate spam or fake news
- Impersonate individuals
- Facilitate misinformation campaigns

6.6.3 Transparency and Accountability

Best practices include:

- Clear model documentation (e.g., model cards, datasheets)
- Usage guidelines and safety constraints
- Auditing and impact assessments