**Building and Optimizing Mini LLMs – A Comparative Study**

A SYNOPSIS ON TERM PROJECT

REPORT

Submitted by

# (Ansh Rastogi – 00614811621 Dhruv Chopra – 03314811621)

BACHELOR OF TECHNOLOGY

IN

## ARTIFICIAL INTELLIGENCE & MACHINE LEARNING

Under the Guidance of

**Dr. Tripti Lamba**

**(Professor, AIML)**



**DEPARTMENT OF ARTIFICIAL INTELLIGENCE & MACHINE LEARNING**

**Maharaja Agrasen Institute of Technology**

**PSP area, Sector – 22, Rohini, New Delhi – 110085**

**(Affiliated to Guru Gobind Singh Indraprastha, New Delhi)**

**Contents**

1. Introduction

1. Problem Statement

1. Objectives

1. Methodology

1. Diagrammatic Portrayal

1. Technologies Used

1. References

# Introduction

# The rapid advancement of Large Language Models (LLMs) has revolutionized the field of Natural Language Processing (NLP), demonstrating remarkable capabilities in text generation, understanding, and interaction. Building on foundational architectures like the Transformer, models such as GPT-3 and GPT-4 have set new benchmarks by leveraging massive datasets and computational power to achieve state-of-the-art performance across diverse language tasks. However, the reliance on vast resources and large-scale data poses significant challenges, especially for smaller-scale implementations and specific domains such as historical literature.

# This project aims to address these challenges by developing a simplified GPT model trained on the Gutenberg Literary English Corpus, focusing on generating text that emulates the style of classical literature. By exploring iterative self-feedback mechanisms like Self-Refine, this research seeks to enhance the model’s output quality without additional supervised training, presenting a cost-effective and accessible alternative to existing LLMs. The study also includes a comparative analysis with larger models to evaluate performance trade-offs, contributing valuable insights into the scalability and efficiency of language models tailored to specialized datasets.

# Problem Statement

* The project aims to build a simplified GPT model trained on the Pocket Shakespearan Dataset, containing many of Shakespeare’s works combined, to explore text generation capabilities in classical literature.
* By comparing the model’s outputs with larger models like GPT-3, this study investigates the performance trade-offs in language style preservation, coherence, and resource efficiency, highlighting the challenges of training LLMs on limited datasets with constrained computational resources.
* The project further aims to evaluate how well smaller models can emulate the linguistic nuances of historical texts and identify potential areas for optimization and future improvements.

# Objectives

 **Develop a Simplified GPT Model**: Design and implement a smaller-scale version of a GPT model, specifically trained on the Gutenberg Literary English Corpus, to generate text that captures the stylistic nuances of classical literature.

 **Evaluate the Effectiveness of Iterative Self-Feedback**: Integrate and test the Self-Refine technique, allowing the model to iteratively improve its outputs through self-feedback without requiring additional supervised training or reinforcement learning.

 **Conduct Comparative Performance Analysis**: Compare the performance of the simplified GPT model with larger, state-of-the-art models like GPT-3.5 and GPT-4, focusing on text quality, coherence, and stylistic accuracy.

 **Optimize Model Training with Limited Resources**: Explore strategies to optimize the training process of the language model under computational constraints, aiming to maximize efficiency without compromising output quality.

 **Highlight Practical Applications and Limitations**: Identify and discuss the practical applications of the simplified GPT model in specialized domains such as digital humanities, content creation, and educational tools, while also addressing the limitations posed by reduced model scale.

# Methodology

**Dataset Preparation:**

* Collect the Gutenberg Literary English Corpus, focusing on classical literature, including works by Shakespeare, Dickens, and other prominent authors.
* Preprocess the dataset by cleaning text data, removing noise, correcting OCR errors, and normalizing archaic language to standard formats suitable for training.

**Model Architecture and Development:**

* Implement a simplified GPT architecture using PyTorch and Hugging Face Transformers, focusing on creating a smaller, computationally feasible model.
* Customize the tokenizer to handle the unique vocabulary and stylistic elements of classical literature to ensure the model captures specific language patterns.

**Training Process:**

* Train the model on the preprocessed dataset using a GPU-enabled environment (e.g., Google Colab) with limited computational resources.
* Apply iterative training methods with checkpoints to monitor model performance, adjust hyperparameters, and reduce overfitting on the specialized data.

**Incorporation of Self-Refine Technique:**

* Integrate the Self-Refine approach, allowing the model to generate initial outputs and iteratively refine them based on self-feedback loops.
* Test this refinement process across various text generation tasks (e.g., dialogue creation, storytelling) to assess improvements in output quality.

**Comparative Analysis and Evaluation:**

* Perform a comparative analysis of the simplified GPT model against larger models like GPT-3.5 and GPT-4, evaluating key metrics such as perplexity, text coherence, and stylistic accuracy.
* Use both automated evaluation tools and human feedback to score generated text, focusing on how well the refined outputs align with the characteristics of the training data.

**Optimization and Resource Management:**

* Explore techniques to optimize training time and computational efficiency, including pruning, quantization, and efficient training regimes.
* Document any trade-offs between model performance and resource consumption, highlighting areas where compromises were made to fit within hardware constraints.

**Discussion of Applications and Limitations:**

* Analyze the potential applications of the simplified model in fields like digital humanities, education, and content generation, where stylistic fidelity to classical text is valuable.
* Address limitations related to model scalability, training data biases, and the impact of reduced model size on overall performance.

# Diagrammatic Portrayal

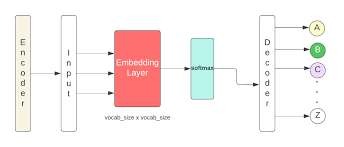


Fig 1 : LLM Model

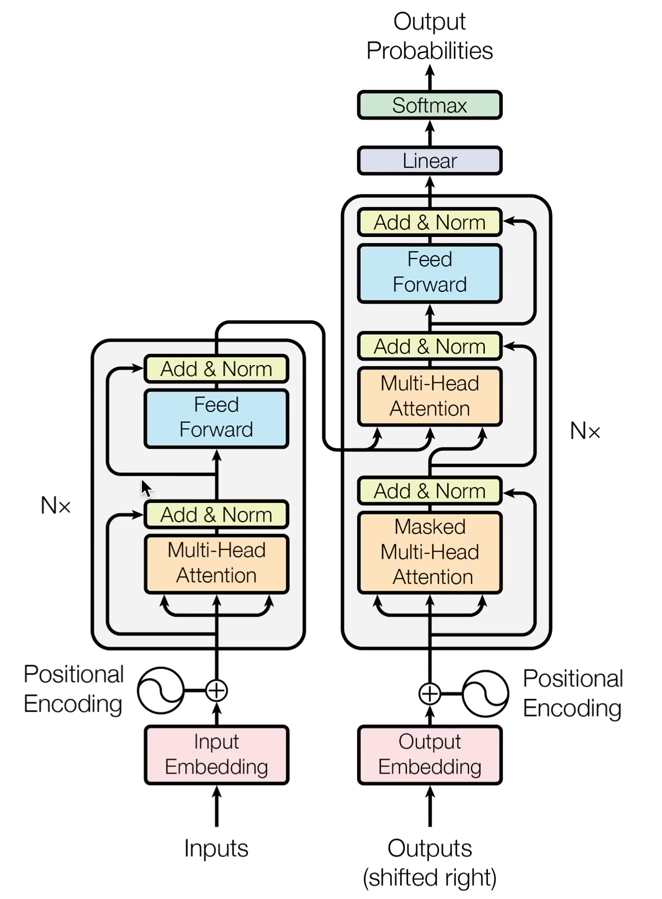


Fig 2 : CNN Model for comaprision

# Technologies Used

**1. PyTorch -** A popular deep learning framework known for its flexibility and dynamic computation graphs, PyTorch is ideal for building and experimenting with neural network architectures, such as GPT models.

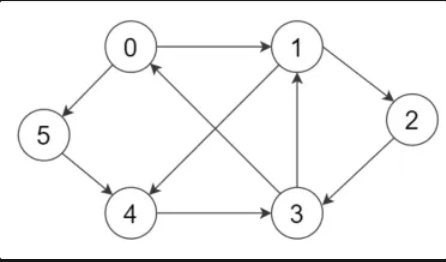
**2. Hugging Face Transformers -** This library provides pre-built models, tokenizers, and training utilities that simplify working with transformer architectures, making it easier to implement, fine-tune, and benchmark your GPT model.

**3. CUDA-Enabled GPU (e.g., NVIDIA GTX Series) -** A CUDA-compatible GPU accelerates model training by performing parallel computations efficiently, significantly reducing the time required to train large neural networks.

**4. TensorBoard -** A visualization tool that allows you to monitor training metrics, model architecture, and performance over time, helping to diagnose issues such as overfitting and track progress during model training.

**Logic behind Self-Attention**

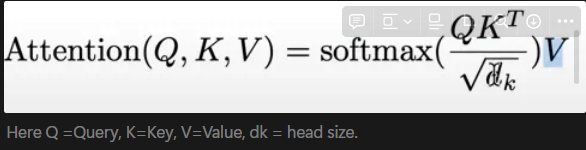
* Attention is a communication mechanism, where we have several nodes in a directed graph. Each node has some vector information and it gets to aggregate information via a weighted sum from all the nodes that point towards it (This is data dependent)



* However, our graph may look different from this as we have 8 nodes (our chosen block size) and there are always 8 tokens in a window. The graph goes as follows:
  + 1st node is only pointed to by itself
  + 2nd node pointed to by 1st and itself ……
  + 8th node pointed to by all the previous nodes and itself
* Attention mechanism can be applied to any collection of nodes in a directed graph
* Also, there is no notion of space. Attention only works a certain set of vectors.
  + By default, these nodes have no idea where they are positioned in space
  + This is why we need to encode them positionally. This is used to anchor them to their specific position.
  + This is different from convolution, coz there is a strict layout to be followed in space. Also, the convolution filter acts in space. Whereas if we want to have a notion of space in self-attention, we need to add it ourselves.
* Each example across the batch dimension is of course processed completely independently and never ‘talk’ to each other.

\*\* To create transformers for different tasks like sentiment analysis we would have to create the transformer as two halves: the encoder and decoder. The encoder would be the same as standard self-attention block just without the upper triangular masking (allowing for all nodes to communicate with each other). And the decoder block would be the standard self-attention block.

* “self-attention” means that the values and keys both come from the same source as queries. In “cross attention”, the queries get produced from a different location than from the keys and values (from the encoder module).
* “Scaled” attention additionally divides wei by 1/sqrt(head\_size). This makes it so that when the input Q, K are unit variance, wei would also be unit variance and the softmax on wei would stay diffused and not saturate too much.
* Formula of self-attention:



**Self-Attention Simplified Code and Review**

**What is Self-Attention?**

Self-attention allows the model to focus on different parts of the input sequence when making decisions, such as predicting the next word in a sentence. Instead of just looking at the immediate previous word, it can focus on relevant words from the entire sequence.

Think of it like this: when you read a sentence, sometimes you need to "attend" to earlier words to understand the meaning. Self-attention mimics this process mathematically.

**Step-by-Step Breakdown of the Code:**

1. **Input (x)**:
   * You start with a batch of sequences (x), where:
     + B is the batch size (4 sequences processed in parallel).
     + T is the sequence length (8 tokens or words in each sequence).
     + C is the number of channels (32-dimensional embeddings for each word/token).
   * So x has shape (B, T, C), which means 4 sequences, each 8 tokens long, and each token is represented by a 32-dimensional vector.
2. x = torch.randn(B,T,C) # Randomly initialized input tensor
3. **Keys, Queries, and Values**:
   * Self-attention is all about comparing tokens in the sequence to each other. You do this with **keys**, **queries**, and **values**.
     + **Key**: Represents what information each token contains.
     + **Query**: Represents what each token is looking for.
     + **Value**: Contains the actual information each token wants to pass on.
   * You create linear transformations for the keys, queries, and values from the input x.

key = nn.Linear(C, head\_size, bias=False)

query = nn.Linear(C, head\_size, bias=False)

value = nn.Linear(C, head\_size, bias=False)

* + Each transformation maps the 32-dimensional input (C=32) down to a smaller head\_size=16. This makes the attention mechanism lighter and faster.

k = key(x) # (B, T, 16) - the 'key' for each token

q = query(x) # (B, T, 16) - the 'query' for each token

1. **Attention Scores (wei)**:
   * **Attention scores** determine how much each token should focus on every other token in the sequence. These scores are calculated by taking the dot product between queries and keys.
   * The formula for attention scores is:



* + - q is the query for a token, and k^T is the transposed key for all tokens.
    - This gives a matrix of shape (B, T, T), where each token attends to every other token in the sequence.

wei = q @ k.transpose(-2, -1) # (B, T, T) - attention scores for all tokens

1. **Example**: Imagine you're trying to predict the next word in a sentence. For each word (query), you're calculating how much attention it should pay to every other word (key) in the sentence.
2. **Masking Future Tokens**:
   * In this example, the model should only attend to the **past and present tokens**, not future ones. This is crucial when you're training on sequences like text because you don't want to cheat by looking at future words.
   * The torch.tril(torch.ones(T, T)) creates a triangular matrix, where values above the diagonal are 0 (blocked), meaning future tokens won't be considered.

tril = torch.tril(torch.ones(T, T)) # Lower triangular matrix to block future tokens

wei = wei.masked\_fill(tril == 0, float('-inf')) # Mask future tokens with -infinity

1. **Example**: Imagine you're reading a sentence one word at a time. If you're at word 3, you can only focus on words 1, 2, and 3. You shouldn’t be able to "peek" at words 4, 5, etc. This triangular matrix ensures that you only pay attention to the current and previous words.
2. **Softmax on Attention Scores**:
   * Once the attention scores are calculated, you apply **softmax** to convert these raw scores into probabilities, so that they sum to 1.
   * The higher the attention score, the more focus a token will have on another token. The softmax ensures that you give more weight to relevant tokens.

wei = F.softmax(wei, dim=-1)

1. **Example**: If you're at word 3, you might want to pay 80% attention to word 2 and only 20% attention to word 1. Softmax helps you figure out these probabilities.
2. **Applying the Values**:
   * The final step is to take the **value** (v) for each token and multiply it by the attention weights (wei). This effectively selects which tokens' values will contribute most to the final output.
   * The attention-weighted values are then combined to produce the final output (out), which is the weighted sum of all tokens' values.

v = value(x) # (B, T, 16) - the 'value' for each token

out = wei @ v # (B, T, 16) - weighted sum of values

1. **Example**: Imagine you're at word 3. Based on the attention scores, you'll take 80% of the information from word 2 and 20% from word 1 to form a new understanding of word 3. The final out tensor reflects this combined information.

**Summary of the Code Logic:**

1. **Input (x)**: A batch of sequences, where each token is represented by a 32-dimensional vector.
2. **Keys, Queries, and Values**: These are computed from the input x using linear transformations.
3. **Attention Scores**: The query for each token is compared to the keys of all tokens to compute how much attention each token should pay to every other token.
4. **Masking**: Future tokens are masked to prevent the model from looking ahead.
5. **Softmax**: Attention scores are converted to probabilities to weight the importance of different tokens.

# Resources

1. Hardware: A computer system with a powerful GPU (Graphics Processing Unit) is required to train and test deep learning models. The system must meet the minimum requirements for deep learning tasks, such as having a fast CPU, a minimum of 8 GB RAM, and a powerful GPU (such as Nvidia GeForce or Tesla).

1. Software: Various software tools and libraries are required for developing the facial recognition system, including Python, TensorFlow, Keras, Pytorch, and other relevant libraries.

1. Dataset: A large dataset of criminal faces is required for training the deep learning model. This dataset can be collected from various sources such as criminal databases or public sources.

1. Cloud Services: Cloud services such as Amazon Web Services (AWS) or Google Cloud Platform (GCP) can be used for training and deploying the model.

1. Ethics and Legal Frameworks: Ethical and legal frameworks related to the use of facial recognition technology in law enforcement must be considered during the development of the project. Relevant legal frameworks, such as privacy laws, data protection laws, and ethical guidelines, must be considered.

1. Expertise: Expertise in deep learning, computer vision, and software development is required for developing a reliable and efficient facial recognition system. The project may require collaboration with experts in law enforcement or related fields.

.

**References (Literature Review)**

* 1. Vaswani et al., "Attention is All You Need" (2017) This landmark paper introduced the Transformer architecture, which underpins modern LLMs like GPT. The authors proposed a self-attention mechanism that allows the model to weigh the importance of different words in a sequence, significantly improving efficiency compared to previous recurrent neural network models. The study demonstrated that the Transformer outperforms traditional models in translation tasks, making it a foundation for subsequent advances in large-scale language modeling.Radford et al., "Language Models are Unsupervised Multitask Learners" (2019) This paper presented GPT-2, an unsupervised language model that demonstrated the ability to perform a wide range of tasks without task-specific training. The authors highlighted how scaling up model size and data volume can dramatically enhance performance across various language tasks, showcasing the model’s ability to generate coherent, contextually appropriate text. Their findings underscored the importance of model scaling and pre-training, setting a precedent for later, larger models like GPT-3.
  2. **Brown et al., "Language Models are Few-Shot Learners" (2020)**

The introduction of GPT-3 marked a significant leap in the capabilities of language models, as detailed in this paper. The model's 175 billion parameters allowed it to perform tasks with minimal training examples (few-shot learning), demonstrating near-human performance in language understanding and generation. This study highlighted the transformative potential of scaling and sophisticated training strategies, but also addressed concerns regarding biases, ethical considerations, and resource demands of large models.

* 1. A**man Madaan et al., "Self-Refine: Iterative Refinement with Self-Feedback" (2023)**

It explores an innovative approach to enhancing the performance of large language models (LLMs) by using iterative feedback and refinement without needing additional supervised training or reinforcement learning. The core idea is to generate initial outputs using an LLM, then have the same model critique and refine its outputs iteratively. This process mimics human refinement techniques, where drafts are improved through self-assessment and revision. The study evaluates Self-Refine across multiple tasks, including dialogue generation, code optimization, and mathematical reasoning, using models like GPT-3.5, ChatGPT, and GPT-4. The results show a significant improvement of about 20% in performance metrics compared to traditional one-step generation methods, highlighting the potential of Self-Refine as a standalone enhancement approach for state-of-the-art LLMs​.