IJEMD-CSAI, 4 (1) (2025) https://doi.org 10.54938/ijemdcsai.2025.04.1.455

International Journal of Emerging Multidiciplinaries

Computer Science & Artificial Intelligence

Research Paper

Journal Homepage: www.ijemd.com

ISSN (print): 0000-0000

EasyGPT: A Streamlined Deep Learning Simulator Part 1: System Design & Optimization

Saira Arif [1*] and F. N. Alavi [2]

1.Department of General Studies, Yanbu University College, Yanbu, Saudi Arabia.
2.(Visiting Professor) Department of Computer Science, Virtual University, Islamabad, Pakistan.

Abstract

This series of papers introduces EasyGPT, a minimalistic, flexible and novel deep learning implementation of the Transformer architecture for the simulation and testing of Natural Language Processing (NLP) applications. Built to open industry standards, our model combines a customizable modular design which enables, among other things, model selection, hyperparameter configuration and user-selectable tokenization engine plugins. In this first paper in the series, we discuss the overall system design of EasyGPT and evaluate its performance by fine-tuning the **DistilGPT2** model on the **DailyDialog** dataset. Our work provides both a simple way for those starting in AI research to experience ChatGPT-like chatbot technologies at the coding level, as well as providing a foundation for the transition towards more realistic and complex model-building and experimentation.

Keywords: GPT , AI , NLP , Deep Learning.

1.1 Introduction

The field of Natural Language Processing (NLP) has a rich tradition going back to the middle of the 20th century. In the immediate aftermath of World War II, a period of unprecedented stability set the stage for technological transformation. This era enabled the migration of mainframe computers from their initial military and government applications to research institutions and universities, where they became indispensable tools for exploring new computational possibilities. As these systems later permeated businesses, commercial entities began to recognize the potential of large-scale data processing, paving the way for the application of computers to complex language tasks.

Email Addresses: courses@studentmail.info(S. Arif) , courses@studentmail.info(F. N. Alavi).

It is self-evident that computer technology has been the cornerstone upon which rapid innovations in NLP have been built. From its inception, advances in hardware and computational paradigms have allowed researchers and engineers to push the boundaries of what is possible in understanding and generating human language using machines. This technological evolution has been both deeply intertwined with and benefited from developments in cognate fields such as Artificial Intelligence (AI) and Artificial Neural Networks (ANNs).

Indeed, one can identify three distinct eras of co-development in these disciplines (Table 1). In the first era (1940s – 1960s), early computational models emerged from pioneering work that sought to apply statistical methods to language processing tasks. As computational power increased, the second era (1970s – 1980s) saw the rise of rule-based systems and the integration of linguistic theories with computer science, which led to more structured approaches to language understanding. The present era, starting around the early 1990s, is characterized by a symbiotic relationship between computing, communication and data technologies.

Era	NLP	AI	ANNs	Computer Technology
Early Period (1940s-1960s)	1954: "Logic Theory Machine" developed 1966: ELIZA chatbot created	1950: Turing Test proposed 1956: Term "Artificial Intelligence" coined 1957: First AI program developed	1943: McCulloch-Pitts neuron model 1958: Perceptron introduced	Emergence of early computers and programming languages
Middle Period (1970s-1980s)	1972: SHRDLU developed 1980s: Rule-based and expert systems	Expert systems developed	1980s: Backpropagation and Training algorithms	Advancements in hardware and software Microprocessors
Modern Era (1990sPresent)	1990s: Statistical and machine learning techniques 2001: Noisy Channel Model for speech recognition 2013: Word2Vec for word embeddings 2017: Transformer architecture 2018: BERT 2020: T5	1990s: Rise of machine learning and data-driven approaches 2006: Deep Belief Networks 2012: AlexNet wins ImageNet 2016: AlphaGo defeats Go champion	1990s: Deep learning techniques (CNNs, RNNs) 2010s: LSTMs, GANs	Rapid advancements in computing power, data storage, and networking- Emergence of deep learning and neural network architectures- Widespread adoption of machine learning and AI in various applications

Table 1: Parallel Development of NLP and cognate technologies

In this age, breakthroughs in all these fields have led to the development of sophisticated models that not only advance NLP but also continuously benefit from progress in each other. We observe that the evolution of NLP reflects a broader narrative – a story of how technological advancements benefit from the compounding power of *interdisciplinary* research and development, leading to groundbreaking capabilities that redefine our interaction with language, information and society as a whole. No recent advance has demonstrated this more effectively and powerfully than the explosive growth in GPT (Generative Pre-trained Transformer) chatbots that have rapidly transformed from

niche research projects to mainstream applications across industries. Their ability to generate fluent, human-like text across a multitude of tasks has driven unprecedented public engagement and widespread adoption [2 , 7]. As businesses and individuals increasingly adopt these technologies, the demand for ever-more customizable and scalable chatbot solutions has surged. It is inevitable that progress in this area will accelerate for the foreseeable future.

1.2 Literature Overview

Historically, we find that NLP research methodologies and technologies originally relied primarily on statistical methods, with literature from the 1980s onwards concentrating on techniques such as the following:

- **Rule-Based Systems**: These employed handcrafted rules to process and understand language. These systems relied on linguistic knowledge, such as grammar rules and syntactic structures, to parse sentences and extract meaning. While effective for specific tasks, rule-based systems were limited in their ability to generalize across different contexts and languages.
- **Statistical Methods**: The early 1990s saw a shift towards statistical approaches in NLP, driven by the availability of large corpora and advancements in computational power. Techniques such as ngrams, which analyze sequences of n words, were commonly used for tasks like language modeling and text classification. Statistical methods allowed for the estimation of probabilities based on observed data, enabling more flexible and data-driven approaches compared to rule-based systems.
- **Hidden Markov Models (HMMs)**: HMMs became popular for tasks such as part-of-speech tagging and speech recognition. These probabilistic models represent systems that transition between hidden states, making them suitable for sequential data. HMMs, however, rely on the assumption that the future state depends only on the current state, which limit their ability to capture long-range dependencies in language.
- **Bag-of-Words (BoW) Models**: In text classification and information retrieval, the bag-of-words model was widely used. This approach represents text as a collection of words, disregarding grammar and word order. While simple and effective for certain applications, BoW models often fail to capture the contextual relationships between words, leading to a loss of semantic information.
- **Vector Space Models**: Techniques such as Latent Semantic Analysis (LSA) and Latent Dirichlet Allocation (LDA) have also been employed to analyze and represent text data in a lower-dimensional space. These models aimed to uncover latent structures in the data, thus enabling practical implementations for topic modeling and document similarity analysis

  However, a shift towards ANN-based solutions also started from the 1990s onwards. This was largely driven by progress in the development of recurrent neural networks (RNNs) and long short-term memory networks (LSTMs). RNNs, (originally introduced in the 1980s), were designed to handle sequential data by maintaining a hidden state that captures information from previous time steps. However, they struggled with long-range dependencies due to issues such as vanishing and exploding. This limitation led to the development of LSTMs, which incorporated memory cells and gating mechanisms to better retain information over extended sequences, thus improving performance on tasks like language modeling and machine translation [6].

1.3 The Transformer Revolution

Despite the advancements offered by LSTMs, the need for more efficient and scalable models persisted.

The fundamental paradigm shift in NLP, which has also led to today's GPT chatbots, took place with the invention of the Transformer architecture by [9]. Transformers introduced and leveraged the concept of *attention*, a mechanism that allows the model to weigh the significance of (or, in human terms, focus attention upon) the *most significant* words in a sequence, enabling them to capture contextual relationships without the sequential processing constraints of RNNs. It is critically this ability of Transformers to handle long-range dependencies in word sequences effectively that has led to their widespread adoption, culminating in the development of state-of-the-art models such as BERT and GPT, which have set new benchmarks in various NLP tasks [3 , 2]. Fortuitously, the Transformer architecture facilitates, in addition, parallelization during training, significantly enhancing computational efficiency and scalability.

In the context of the Transformer architecture, attention is a mechanism that allows the model to focus on specific parts of the input data when generating output. It computes a weighted sum of the input elements, where the weights are learned based on the input data. The attention mechanism takes three inputs, utilizing terminology adapted from database theory:

1. **Query (*Q*)**: This represents the input used to compute the attention weights, essentially serving as the request for information from the model.
2. **Key (*K*)**: This is associated with the Query and is used to compute the attention weights by determining how relevant each Key is to the Query.
3. **Value (*V*)**: This input is utilized to generate the output of the attention mechanism, providing the actual information that is retrieved based on the computed attention weights.

Both Q and K are involved in calculating the attention weights, whereas V is the information that is ultimately returned as output:

$$\text{Attention}(Q, K, V) = \sum_{i=1}^{n} \alpha_i V_i ,$$

where α_i is the attentional weight for the i^{th} element, and n is the number of inputs. The attention weights are computed using

$$\alpha_i = \frac{\exp(\text{score}(Q,K_i))}{\sum_{j=1}^{n} \exp(\text{score}(Q,K_j))} ,$$

where $\text{score}(Q, K_i)$ is the score function that computes the similarity between the query and the i^{th} key.

It is typically computed using a simple dot product as follows:

$$\text{score}(Q, K) = \frac{Q \cdot K}{\sqrt{d}} ,$$

where d is a dimensionality parameter.

It is worth noting that Transformers are capable of handling two types of attention mechanisms:

1. Self-attention is a type of attention mechanism that computes the attention weights based on the *input data itself*. In self-attention, the query, key, and value are all derived from the same input data. It is

used to allow the model to attend to different parts of the input data when generating output, and is computed as follows:

$$\text{Self-Attention}(X) = \text{Attention}(XW_Q, XW_K, XW_V) ,$$

where X is the input data, and W_Q, W_K, and W_V are learnable weight matrices that are used to compute the query, key, and value, respectively.

2. Cross-attention, in contrast, is a type of attention mechanism that computes the attention weights based on *two different input data*. In cross-attention, the query is derived from one input data, and the key and value are derived from another input data. It is used to allow the model to attend to different parts of the input data when generating output, and is computed as follows:

$$\text{Cross-Attention}(X, Y) = \text{Attention}(XW_Q, YW_K, YW_V) ,$$

where X and Y are the two input data, and W_Q, W_K, and W_V are learnable weight matrices that are used to compute the query, key, and value, respectively.

A schematic representation of the Transformer architecture is presented in Figure 1. As several detailed analyses of the Transformer architecture and its performance metrics can be found elsewhere (see, for instance) [8,5]. We shall restrict our discussion to its primary features and the implications of its design choices for our software implementation.

1.3.1 Transformer Inputs and Outputs

We observe firstly that the architecture consists of two blocks, an **encoder** and a **decoder**. The inputs into both of these blocks consist of a combination of *embeddings* and *positional encodings*, which together facilitate the model's understanding of the input data. Embeddings are dense vector representations of *tokens*, where each token in the vocabulary is mapped to a continuous vector space. This representation captures semantic relationships between words, allowing the model to understand similarities and differences in meaning. For instance, words with similar meanings tend to have embeddings that are closer together in this vector space, enabling the model to leverage these relationships during processing.

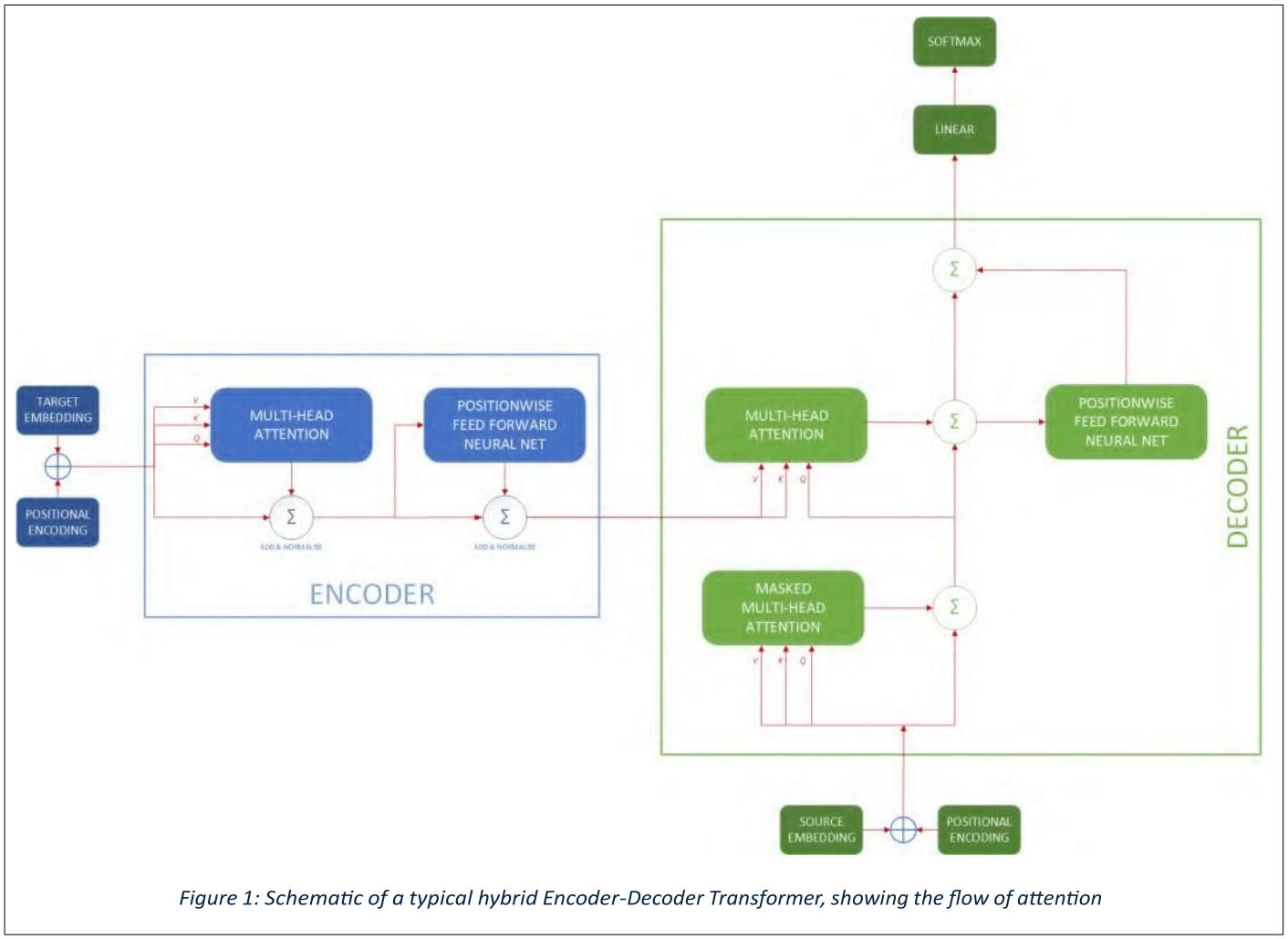


Figure 1: Schematic of a typical hybrid Encoder-Decoder Transformer, showing the flow of attention

Positional encodings, on the other hand, are added to the embeddings to provide information about the position of each token within the input sequence. Since the Transformer architecture does not inherently account for the order of tokens – unlike recurrent neural networks (RNNs) that process sequences in a linear fashion – positional encodings are crucial for maintaining the sequential nature of the data. These encodings are typically generated using sine and cosine functions of different frequencies, allowing the model to distinguish between tokens based on their positions in the sequence.

The process of tokenization involves breaking down the input text into smaller units, or tokens, that can be processed by the model. Tokenization can vary in granularity, ranging from character-level to wordlevel or sub word-level tokenization, depending on the specific approach used. Once the text is tokenized, each token is converted into its corresponding embedding, and positional encodings are subsequently added to these embeddings to create the final input representations for both the encoder and decoder. This combination of embeddings and positional encodings ensures that the model not only captures the semantic meaning of the tokens but also retains the necessary information about their order within the sequence, which is essential for effective language understanding and generation

The final output is a set of probabilities computed by a terminal stage SoftMax function, which for a given input vector $z = [z_1, z_2, \ldots, z_n]$, is defined as:

$$\sigma_{z_i} = \frac{\exp(z_i)}{\sum_{j=1}^{n} \exp(z_j)}$$

The SoftMax function outputs a probability distribution over the input vector, where each output value is in the range (0, 1) and the sum of all output values equals 1. This makes it particularly useful for multiclass classification problems.

1.3.2 Encoder and Decoder Blocks

We next consider the encoder and decoder blocks, where the main calculations take place that transform the text inputs into coherent outputs that humans can understand.

The encoder, which is responsible for processing the input sequence and generating a set of continuous representations that capture the contextual information of the input data. It comprises multiple layers of self-attention and feed-forward neural networks, allowing it to effectively model relationships between words regardless of their positional distance. This capability is particularly beneficial in tasks such as text classification and sentiment analysis, where the encoder can extract meaningful features from the input without the need for sequential processing. In encoder-only applications, such as BERT (Bidirectional Encoder Representations from Transformers), the model is trained to understand the context of words in a sentence, making it highly effective for tasks that require comprehension of the input text.

The decoder component of the Transformer architecture is designed for generating output sequences based on the encoded representations. It employs a similar structure to the encoder but includes an additional layer of masked self-attention, which prevents the model from attending to future tokens during the generation process. This design is crucial for tasks such as machine translation and text generation, where the model must produce coherent and contextually relevant output. In decoder-only applications, such as GPT the model is optimized for generating text by predicting the next word in a sequence based on the preceding context. Mixed encoder-decoder applications, exemplified by models like T5 (Text-to-Text Transfer Transformer), leverage both components to handle a wide range of tasks, including translation, summarization, and question answering. By utilizing the encoder to process the input and the decoder to generate the output, these models can effectively transform one type of text into another, showcasing the versatility and power of the Transformer architecture in various natural language processing scenarios.

The distinct roles of the encoder and decoder in the Transformer architecture enable a variety of applications tailored to specific NLP tasks.

- *Encoder-only models* excel in understanding and classifying input data. Their primary mechanism is self-attention, and they are used in NLP models such as BERT. The use of self-attention allows the model to weigh the importance of each word in the input sequence relative to all other words, effectively capturing contextual relationships and dependencies. and generate rich contextual

embeddings for each token, which are crucial for tasks like text classification, named entity recognition, and sentiment analysis. Since self-attention is symmetrical with respect to the input data vector X, its bidirectional nature allows encoder-only models to consider both preceding and succeeding words, thereby enhancing their understanding of context and meaning.

- *Decoder-only models* are used for text generation tasks. They are the primary engine behind modern GPT-based chatbots such as OpenAI's ChatGPT, and utilize a variant of the self-attention mechanism, called *masked* self-attention, to ensure coherent output. The primary purpose of masked self-attention is to ensure that the model generates text in an autoregressive manner, meaning that it can only attend to the tokens that have already been generated in the sequence, rather than looking ahead at future tokens. Their ability to learn from vast amounts of text data allows them to generate contextually rich and relevant responses.
- *Mixed encoder-decoder models* combine the strengths of both components, allowing for complex transformations and interactions between input and output sequences. The leverage both self-attention and cross-attention mechanisms, and are used in popular NLP models such as T5 and BART. The integration of cross-attention enables the decoder to align its generated output with the relevant parts of the input, facilitating more accurate and contextually appropriate responses. Specifically, the encoder employs self-attention to create contextual embeddings from the input sequence, while the decoder utilizes cross-attention to attend to the encoder's output. This dual attention mechanism allows the decoder to generate output sequences that are informed by the entire input context, making these models particularly effective for tasks such as machine translation, summarization, and question answering.

1.4 Motivation

In spite of their efficiency in targeted learning and computational implementation, Transformer architectures and their variations fundamentally depend on the availability of credible, high-quality training data from which they can learn. The performance of these models is heavily influenced by the quality and quantity of the data used during training. High-quality datasets are essential for ensuring that the models can generalize well to unseen data and perform effectively across various tasks. This requirement poses several challenges, including the need for substantial financial investment, significant computational resources, and the establishment of robust data management infrastructures.

The process of curating high-quality training data often involves extensive data collection, cleaning, and preprocessing, which can be both time-consuming and costly. This demands large-scale investments in data acquisition strategies, which may include purchasing datasets, conducting surveys, or leveraging user-generated content (although artificially created data is now taking on ever-increasing role, even though it comes with the inherent risk of *data poisoning*). Additionally, the computational power required to train large-scale Transformer models necessitates the use of high-performance data centers equipped with advanced hardware, such as Graphics Processing Units (GPUs) or Tensor Processing Units (TPUs). And all of these issues stand in the way of undertaking new research in this area, even before one even begins to consider downstream problems such as ethical considerations, potential data bias, and data governance policies.

We are thus led naturally to consider minimalistic Transformer implementation for research and study purposes. Inspired by the recent success of DeepSeek, who demonstrated in practice that ChatGPT-like performance is possible on commodity PC hardware, we undertook the development of an even more minimalistic Transformer implementation that we have called EasyGPT. The rest of this paper concentrates on the design of EasyGPT, together with the results that we have obtained.

2.1 EasyGPT System Design

EasyGPT has been implemented as a series Python-language modules that leverage Python's vast ecosystem of AI and ML libraries as much as possible. This approach was taken for several reasons, including the fact that it allowed the project to focus on high-level system-level design and development, reducing development time and technical debt [1], improving maintainability, whilst also benefiting from the collective expertise and optimized implementations of the open-source community, since existing libraries have already been largely tested, validated, and refined.

The decision to use Python was further validated by the existence of pre-trained Python-based AI models that allowed us to align with our original objective of developing a simulator that could be run on minimalistic PC-level hardware. The choice then had to be made between the various open-source platforms that exist today; we investigated the most popular of these in terms of technical comprehension.

Platform	Comprehensiveness	Ease of Use	Popularity
Hugging Face	Extensive library of pretrained models, datasets, and tools for NLP and beyond.	Highly user-friendly with excellent documentation and community support.	Extremely popular, widely adopted across academia and industry.
TensorFlow	Comprehensive framework for building and deploying ML models across domains.	Moderate learning curve but highly versatile with strong community support.	Very popular, especially among deep learning practitioners.
OpenVINO	Focused on optimizing and deploying AI models on Intel hardware.	Moderate ease of use; requires familiarity with hardware optimization.	Popular in industries requiring efficient AI deployment on Intel devices.
Stanford NLP Group	Renowned for cutting-edge NLP tools like CoreNLP.	Advanced tools but less beginner-friendly compared to others.	Popular in academic and research settings.
Spacy	Comprehensive NLP library with pre-trained pipelines and customization options.	Extremely user-friendly and well-documented.	Highly popular among developers and researchers.
AllenNLP	Focused on NLP research with tools for deep learning-based tasks.	Moderate ease of use; geared towards researchers.	Popular in the NLP research community.

Table 2: Comparison of open-source pre-trained models

ease of use and popularity in the AI research community (Table 2) and settled for Hugging Face as it ranked highest in all three of these criteria.

Finally, we had to choose from Hugging Face's set of pre-trained models and datasets. In keeping with our objective of running our simulator on commodity GPU-equipped PC hardware, we selected **DistilGPT2**[2] as the transformer model and the **DailyDialog**[3] dataset for training; this combination represents an optimal choice due to its balance of computational efficiency, model performance, and dataset suitability for dialogue tasks. Although it is a distilled version of GPT-2 (which is decoder-only), **DistilGPT2** is actually a mixed encoder-decoder model, which in its standard form is primarily used in decoder-only form. It comprises 6 layers, 12 attention heads and 82 million parameters, making it significantly lighter than larger Hugging Face models like GPT-2 (124 million parameters) or BERT-base (110 million parameters), which demand greater memory and computational resources that may exceed the capabilities of a typical mid-range GPU. This lightweight architecture enables efficient fine-tuning within the hardware constraints while retaining strong language generation capabilities.
The **DailyDialog** dataset, consisting of 13,118 multi-turn dialogues, is relatively compact compared to larger datasets like OpenSubtitles[4] or Cornell Movie Dialogs[5], which contain millions of utterances and require substantial preprocessing and storage. Its focus on everyday conversational scenarios aligns well with the goal of training a dialogue response generation model, and its manageable size ensures that tokenization, data loading, and training can be performed efficiently on mid-range hardware without excessive disk I/O or memory overhead.

2.1.1 Model Architecture

Our Python implementation introduced specific modifications to **DistilGPT2** in order to enhance its suitability for dialogue response generation on the **DailyDialog** dataset, as well as optimize it for the constraints of mid-range hardware. The model architecture is presented in pseudo-code form in Figure 2, and in schematic form in Figure 3.

2.1.2 Dataset and Data Loading

The script leverages the **DailyDialog** dataset, loading it via the Hugging Face *Datasets* library. Dialogues are preprocessed to extract input-response pairs by iterating through each dialogue and pairing consecutive utterances, separated by the [SEP] token. Both training and validation splits are processed in this manner. To expedite experimentation, the script subsamples the training and validation datasets to a maximum of 20,000 and 5,000 examples, respectively, after shuffling for randomness

[2] https://huggingface.co/distilbert/distilgpt2
[3] https://huggingface.co/datasets/roskoN/dailydialog
[4] https://www.opensubtitles.org
[5] https://www.cs.cornell.edu/~cristian/Cornell_Movie-Dialogs_Corpus.html

2.1.3 Configuration & Setup

The script begins by establishing a flexible experimental environment through command-line argument parsing, allowing users to specify essential hyperparameters[6] such as learning rate, number of epochs, batch size, gradient accumulation steps, dropout rate, sequence length, and experiment name. Logging is configured for informative output, and the script automatically detects and utilizes GPU resources if available, defaulting to CPU otherwise.

First, we froze the first four layers during training to reduce computational load and focus finetuning on the upper layers, which are more adept at capturing task-specific patterns – for the purposes of this first implementation, the freezing of lower transformer layers was a strategic choice for efficient domain adaptation.

2.1.4 Tokenization

Tokenization is handled using Hugging Face's *AutoTokenizer* for the **DistilGPT2** model. The script introduces a custom separator [SEP] token to delineate input-response segments within dialogue pairs, ensuring the model can effectively process the structured **DailyDialog** format and focus attention on the nuances of dialogue modeling by distinguishing between turns in conversation. If this token is not already present, it is added to the tokenizer's special tokens, and the model's token embeddings are resized accordingly. The tokenizer is set to pad sequences to a uniform length using the model's end-of-sequence token, and truncation is applied to maintain the specified maximum sequence length.

2.1.5 Dropout Rates

Dropout rates – which determine how many neurons are ignored during training – were adjusted via model configuration parameters to mitigate overfitting on our subsampled dataset. The model configuration is updated to reflect the user-specified dropout rates for attention, residual, and embedding layers, enhancing regularization during training.

2.1.6 Training Loop

Training is orchestrated using Hugging Face's *Trainer* class, which manages the forward and backward passes, evaluation, and checkpointing. Training arguments are set to match the user's configuration, including batch sizes, learning rate, weight decay, gradient accumulation, and evaluation frequency. The script evaluates the model at regular intervals and saves checkpoints, including the best-performing model based on evaluation loss. Mixed-precision training is enabled if a compatible GPU is detected, improving efficiency.

[6] In this context, "hyperparameters" are configuration setting that are fixed before model training commences, and which control the learning process; this distinguishes them from the model parameters (such as weights) that are directly learned from the training data.

Algorithm Outline: Fine-Tuning a Pre-Trained Language Model

Input:

- dataset: a dataset of dialogues (e.g. DailyDialog)
- model_name: the name of the pre-trained language model (e.g. distilgpt2)
- device: the device to use for training (e.g. GPU or CPU) **Output:**
- fine_tuned_model: the fine-tuned language model
- training_history: the training history of the model **Procedure:**

Load Dataset: o Load the dataset dataset and preprocess it to create input-response pairs.
Split the dataset into training and validation sets.

Initialize Model and Tokenizer: o Initialize the pre-trained language model model with the specified model_name.
o Initialize the tokenizer tokenizer with the specified model_name.
Add a custom [SEP] token to the tokenizer.

Configure Model: o Configure the model with the specified hyperparameters (e.g. attention dropout, residual dropout, embedding dropout).
Freeze the first 4 layers of the model.

Tokenize Dataset: o Tokenize the input-response pairs in the training and validation sets using the tokenizer.
Create a tokenized dataset train_dataset and val_dataset.

Save Token Frequencies: o Save the token frequencies of the training dataset to a file token_freq.json.

Define Training Arguments: o Define the training arguments training_args with the specified hyperparameters (e.g. number of train epochs, batch size, learning rate).

Initialize Trainer: o Initialize the trainer trainer with the model, training arguments, and tokenized datasets.

Start Training: o Start training the model using the trainer.

Save Final Model: o Save the fine-tuned model to a file fine_tuned_distilgpt2.

Save Training History: o Save the training history to a file training_history.json.

Figure 2: EasyGPT pseudo-code

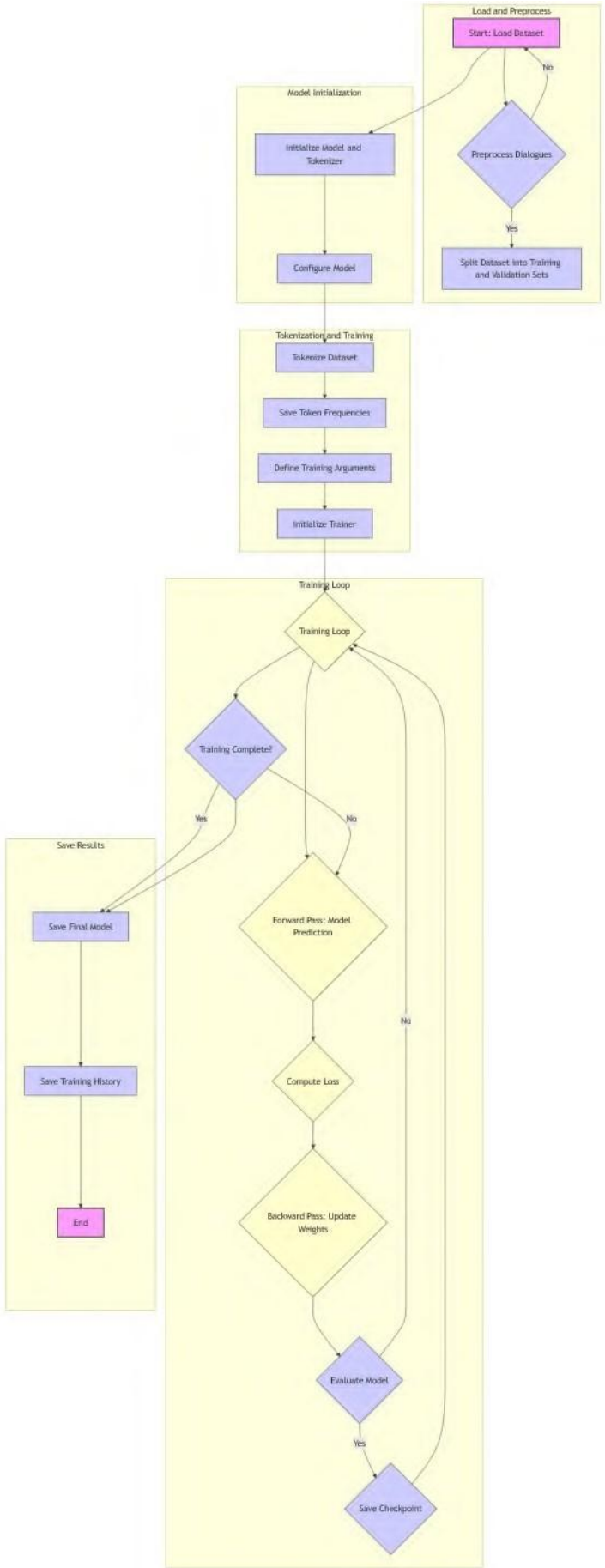


Figure 3: EasyGPT Program Flow Diagram

2.1.7 Validation

Validation is integrated into the training loop, with evaluation loss computed at regular intervals specified by the user. The script tracks both training and validation losses, facilitating monitoring of overfitting and generalization throughout the fine-tuning process. The best model checkpoint is selected based on the lowest evaluation loss.

2.1.8 Inference & Generation

Since the script is primarily focused on training and evaluation, in this version, it does not include explicit routines for inference or text generation post-training. However, the final fine-tuned model and tokenizer are saved, enabling downstream inference and generation tasks to be performed using standard Hugging Face pipelines or custom scripts.

2.1.9 Data Export

Upon completion of training, the script exports several artifacts. The fine-tuned model and tokenizer are saved, and training logs are compiled into a JSON file, capturing the training and evaluation loss histories as well as token frequency counts from the training set. This structured export enables subsequent analysis and visualization.

2.1.10 Additional Features

A notable feature is the computation and export of token frequency statistics from the training set, which can be valuable for analyzing model behavior and dataset characteristics. The script's modularity, with parameterized configuration and robust logging, supports systematic experimentation and reproducibility.

2.2 EasyGPT Performance

The performance of any chatbot model is a crucial aspect of its overall effectiveness, as it directly impacts the quality of the conversations it can engage in. Given the complexity of natural language processing tasks, any model's performance should be evaluated on multiple performance-based *objective* metrics, and not just *subjective* measures that inevitably ensue over the course of a human-machine conversation.

As the Transformer architecture has been shown to be highly effective in a variety of NLP tasks, including language translation, text summarization, and dialogue generation, we should expect that the model will be able to learn the patterns and structures of language, and generate responses that are not only grammatically correct but also contextually relevant. However, such an undertaking would necessarily mandate the availability of large, high-quality training data, significant computing resources, and subjective supervision of the responses by humans.

Fortunately, various objective performance metrics exist that can measure a model's ability to generate responses that are coherent and relevant to user input prompts, without recourse to such resources. It was therefore decided to restrict all model testing to the computation of such metrics and evaluate

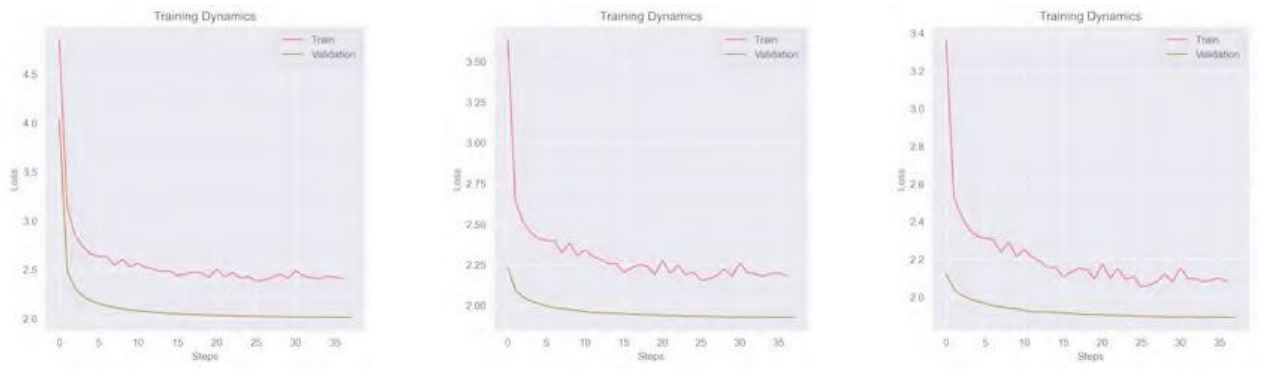
actual text generation in subsequent papers in this series. This section discusses the results of tests that we carried out which demonstrate the effectiveness of our approach. A secondary benefit of this approach is that by initially restricting ourselves to an evaluation only of the model's performance metrics, it is possible to gain a comprehensive understanding of its strengths, weaknesses and optimal settings, which may assist in the identification of areas for further improvement, such that future text generation tasks may be simplified.

We opted to test the model's performance by running a series of hyperparameter-tuning experiments. In designing these experiments, we were motivated to deliberately adopt a *minimalist* approach to both training and computational power – clearly, if the model performs reasonably well on low-spec hardware with just enough training data to ensure convergence in the chosen benchmarks, it stands to reason that it would do even better with significantly more powerful computing hardware and larger amounts of training data. The hyperparameter-tuning process involved fine-tuning the model with specified hyperparameters and storing the resulting training logs for later analysis by visualization scripts. The experiments aimed to balance training stability and convergence speed while ensuring effective learning of conversational patterns, and we discuss these in detail next.

2.2.1 Experiment 1: Impact of Learning Rate on Model Performance

Experiment 1 was designed to investigate the effect of varying the learning_rate hyperparameter across the three values of 10^{-5}, 5×10^{-5}, and 10^{-4}. Analysis of the training dynamics revealed distinct behaviors across the learning rates.

From Figure 4, we observe that for a learning rate value of 10^{-5}, the training loss decreased gradually from 3.5 to approximately 2.25 over 35 steps, with the validation loss consistently lower, dropping from 2.25 to 1.90, suggesting potential underfitting due to the conservative learning rate. The baseline learning rate of 5×10^{-5}demonstrated a more stable convergence, with the training loss decreasing from 3.5 to 2.25 over 60 steps and the validation loss mirroring this trend, reducing from 2.25 to 1.90, indicating a balanced learning process. Conversely, the learning rate of 10^{-4} exhibited the fastest initial convergence, with the training loss dropping to 2.25 within 12 steps; however, minor fluctuations around step 5 hinted at potential instability, and the validation loss closely tracked the training loss, decreasing from 2.25 to 1.95, raising concerns about overfitting with prolonged training.



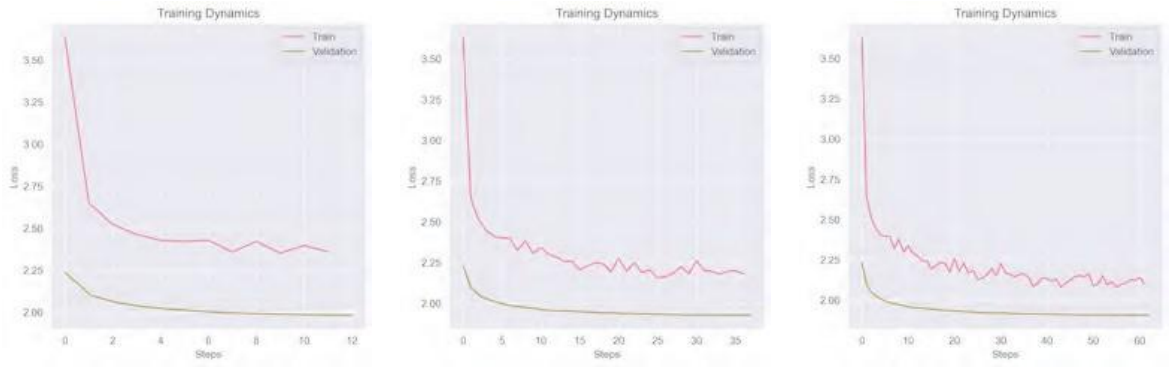
`learning_rate` = 10^{-5} `learning_rate` = 5×10^{-5} (baseline) `learning_rate` = 10^{-4}

Figure 4: Experiment 1 - Impact of learning rates

2.2.2 Experiment 2: Effect of Training Epochs on Model Performance

Experiment 2 was conducted to evaluate the impact of varying the number of training epochs on the model's ability to learn conversational patterns effectively and aimed to determine the optimal number of epochs for achieving a balance between sufficient learning and computational efficiency while avoiding overfitting. Three configurations were tested: 1-epoch, 3-epochs (baseline), and 5-epochs. The training was performed using the previously ascertained optimal learning rate value of 5×10^{-5}.

The training dynamics and validation loss plots in Figure 5 provide insights into the model's learning behavior across the different epoch settings. For the 1-epoch configuration, the training loss decreased rapidly from 3.5 to approximately 2.25 within 12 steps, with the validation loss closely tracking this trend, dropping from 2.25 to 1.95 over 12 evaluation steps. However, the limited training duration suggests potential underfitting, as the model may not have had sufficient time to capture complex patterns in the data. The baseline configuration with 3 epochs exhibited a more stable and extended learning trajectory, with the training loss decreasing from 3.5 to 2.25 over 60 steps and the validation loss reducing from 2.25 to 1.90 over 60 evaluation steps, indicating a well-balanced learning process. In contrast, the 5-epoch configuration showed a similar initial decrease in training loss from 3.5 to 2.25 within the first 35 steps, but the validation loss, while decreasing from 2.25 to 1.90, displayed fluctuations toward the end, suggesting the onset of overfitting as the model may have begun to memorize the training data.



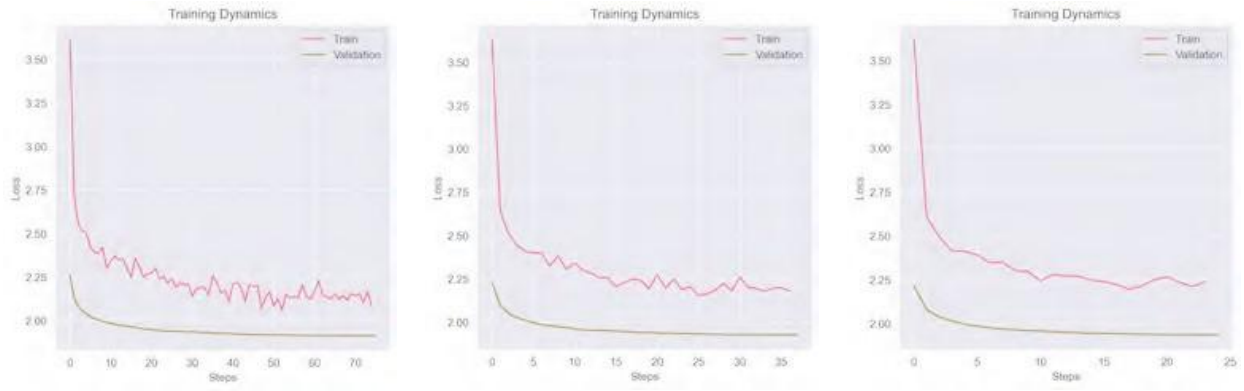
1-epoch 3-epoch 5-epoch

Figure 5: Experiment 2 - Impact of training epochs

2.2.3 Experiment 3: Influence of Effective Batch Size on Model Performance

Experiment 3 was designed to assess the impact of varying the effective batch size on the fine-tuning of the model, with the goal of optimizing its conversational performance. Three effective batch sizes were tested: 16 (achieved with a per-device batch size of 8 and gradient accumulation steps of 2), 32 (with a per-device batch size of 16 and gradient accumulation steps of 2), and 48 (with a per-device batch size of 16 and gradient accumulation steps of 3). The training was conducted with the learning rate set to $5 \times 10^{-5}$and the number of epochs fixed at 3, as determined from prior experiments.

As we see in Figure 6, the training dynamics and validation loss plots reveal distinct patterns across the different batch sizes. For the effective batch size of 16, the training loss decreased sharply from 3.5 to approximately 2.25 within 25 steps, with the validation loss mirroring this trend, dropping from 2.20 to 1.95 over 20 evaluation steps. However, the smaller batch size resulted in a relatively noisy training loss curve, indicating potential instability due to higher variance in gradient updates. The batch size of 32 demonstrated a more stable learning trajectory, with the training loss decreasing from 3.5 to 2.25 over 35 steps and the validation loss reducing from 2.25 to 1.90 over 35 evaluation steps, suggesting a balanced trade-off between gradient noise and computational efficiency. The effective batch size of 48 showed the smoothest and most extended convergence, with the training loss decreasing from 3.5 to 2.25 over 70 steps and the validation loss dropping from 2.25 to 1.90 over 70 evaluation steps. However, the extended training duration and minor fluctuations in the validation loss toward the later steps hinted at a risk of overfitting, as the larger batch size may lead to overly smoothed gradients that reduce the model's ability to generalize.



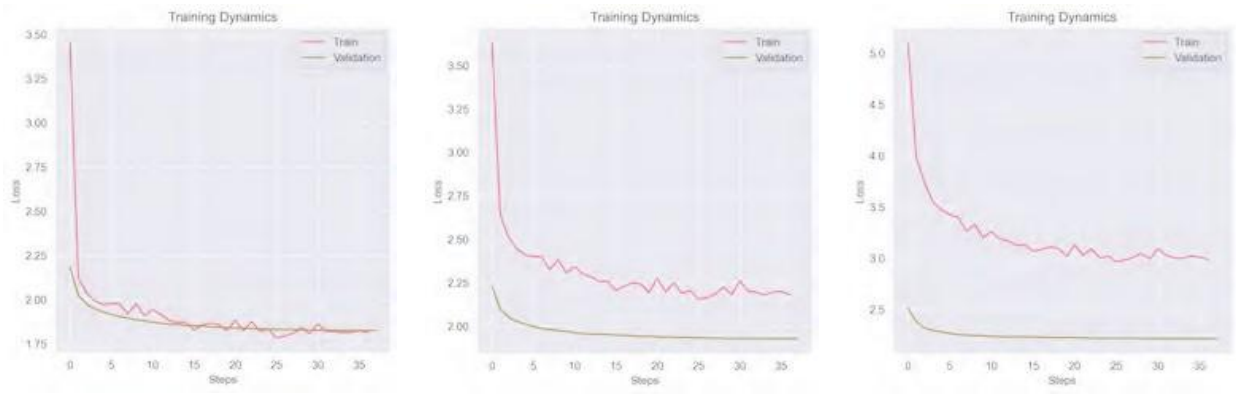
Batch size = 16 Batch size = 32 Batch size = 48

Figure 6: Experiment 3 - Impact of Batch Size

2.2.4 Experiment 4: Impact of Dropout Rate on Model Generalization

Experiment 4 was conducted to investigate the effect of varying dropout rates on the fine-tuning of the model. Three dropout rates were tested: 0.1, 0.3, and 0.5. The training was executed using the previously established learning rate of 5×10^{-5}, the number of epochs at 3, and the effective batch size at 32. This experiment aimed to identify the optimal dropout rate that balances regularization to prevent overfitting while ensuring the model retains sufficient capacity to learn conversational patterns.

The training dynamics and validation loss plots in Figure 7 provide insights into the model's behavior across the different dropout settings. For the dropout rate of 0.1, the training loss decreases rapidly from 5.0 to 2.5 within 35 steps, but the validation loss, starting at 2.50, only drops to 2.20 over 35 evaluation steps, indicating potential under-regularization as the model may have overfit to the training data. The baseline configuration with a dropout rate of 0.3 shows a more balanced learning trajectory, with the training loss decreasing from 3.5 to 2.25 over 35 steps and the validation loss reducing from 2.25 to 1.90 over 35 evaluation steps, suggesting effective regularization that improved generalization. The dropout rate of 0.5 exhibits a slower convergence, with the training loss decreasing from 3.5 to 2.5 over 35 steps and the validation loss dropping from 2.45 to 2.20 over 35 evaluation steps, indicating that the higher dropout rate may have overly regularized the model, leading to underfitting and hindering its ability to capture complex patterns in the data.



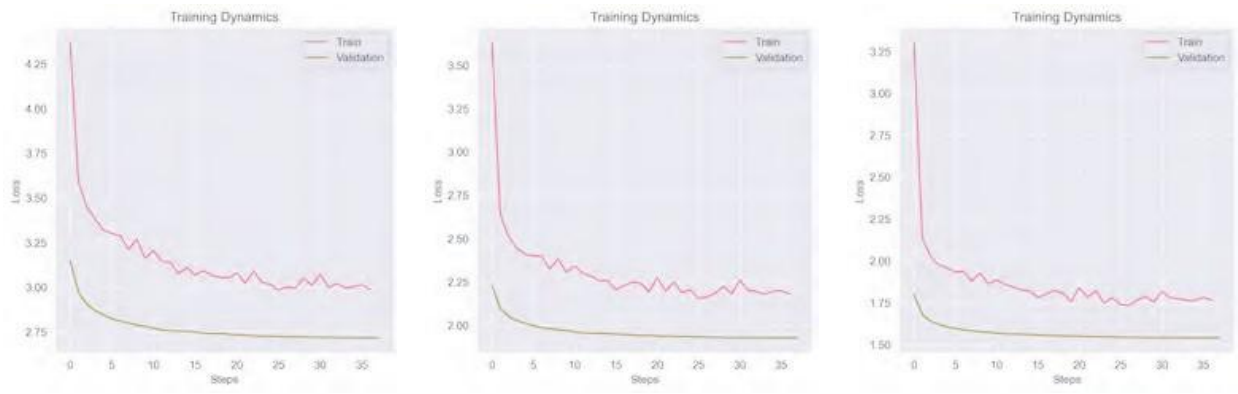
Dropout rate = 0.1 Dropout rate = 0.3 Dropout rate = 0.5

Figure 7: Experiment 4 - Impact of Dropout Rate

2.2.5 Experiment 5: Effect of Maximum Sequence Length on Model Performance

Experiment 5 was designed to evaluate the impact of varying maximum sequence lengths on the model, aiming to optimize its ability to handle conversational contexts of different lengths. Three maximum sequence lengths were tested: 32, 50 (presumed baseline), and 64. The training was performed using the number of epochs at 3, the effective batch size at 32, and the dropout rate at 0.3.

From Figure 8, the training dynamics and validation loss plots highlighted distinct behaviors across the tested sequence lengths. For the maximum sequence length of 32, the training loss decreases sharply from 3.25 to 1.75 over 35 steps, and the validation loss drops from 1.80 to 1.55 over 35 evaluation steps, indicating rapid convergence. However, the shorter sequence length likely limits the model's ability to capture longer conversational dependencies, as reflected in the slightly higher validation loss compared to the presumed baseline; the latter, with a maximum sequence length of 50, shows a balanced learning curve, with the training loss decreasing from 3.5 to 2.25 over 35 steps and the validation loss reducing from 2.25 to 1.90 over 35 evaluation steps, suggesting an effective compromise between capturing context and maintaining computational efficiency. The maximum sequence length of 64 exhibits a slower convergence, with the training loss decreasing from 4.25 to 2.75 over 35 steps and the validation loss dropping from 3.1 to 2.7 over 35 evaluation steps. The higher initial losses and slower convergence suggest that the longer sequence length increase the complexity of the optimization problem, potentially leading to underfitting due to insufficient training steps relative to the increased context.



Sequence length = 32 Sequence length = 50 Sequence length = 64

Figure 8: Experiment 5 - Impact of Sequence Length

3.0 Analysis of Convergence Plots

With the data from the experiments at hand, we are now in a position to make the following observations:

3.0.1 Learning Rate:

A learning rate of 5×10^{-5} is the most effective for fine-tuning the model, offering a stable and consistent reduction in both training and validation losses without the underfitting seen at 10^{-5} or the instability observed at 10^{-4}.

3.0.2 Epoch Length:

3 epochs are the most effective for fine-tuning the model, striking an optimal balance between learning capacity and generalization, as evidenced by the stable convergence of both training and validation losses. The 1-epoch setup was insufficient for the model to fully learn the underlying patterns, while the 5-epoch setup showed signs of overfitting, which could degrade performance on unseen data.

3.0.3 Effective Batch Size:

An effective batch size of 32 is the most suitable, providing a stable and efficient learning process with minimal noise in the training dynamics and consistent generalization performance, as evidenced by the validation loss. The batch size of 16, while computationally lighter, introduced excessive noise in the training process, potentially hindering convergence stability, whereas the batch size of 48, despite its smoother gradients, risked overfitting due to prolonged training and reduced gradient diversity.

3.0.4 Dropout Rate:

The dropout rate of 0.3 is most effective, achieving a robust balance between regularization and learning capacity, as evidenced by the stable convergence of both training and validation losses. The dropout rate of 0.1 was insufficient to prevent overfitting, leading to a larger gap between training and

validation losses, while the dropout rate of 0.5 overly constrained the model, resulting in underfitting and higher validation loss.

3.0.5 Sequence Length:

A maximum sequence length of 50 offers a balanced trade-off between capturing sufficient conversational context and maintaining training stability, as evidenced by the consistent reduction in both training and validation losses. The sequence length of 32, while computationally efficient, constrained the model's ability to model longer dependencies, potentially limiting its conversational coherence. Conversely, the sequence length of 64 increased computational demands and led to slower convergence, with higher validation loss indicating a risk of underfitting within the fixed training duration.

3.1 Analysis of Token Distribution Plots

The token distribution analysis, consistent across all five experiments, highlighted a predominance of punctuation and function words (e.g., ".", ",", "you", "?") in the dataset, suggesting a persistent bias in the model's learned representations toward these high-frequency tokens. This emphasizes the need for strategies to address the model's bias toward frequent tokens, such as data augmentation or loss weighting, to enhance the quality of generated responses; failure to do so will impact the model's ability to generate diverse and contextually rich responses. These trends necessitate a deeper investigation into the tokenization process, the impact of sequence length on token counts, and the implications for model training.

3.1.1 Impact of Sequence Length on Token Frequencies

The token distribution plots provided in Figure 9 illustrate the frequency of the top 20 tokens, with sequence lengths of 32, 50, and 64 tokens, respectively. While the shape of the distributions – namely, the relative ordering of tokens and the proportionality of their frequencies – remains consistent across the three sequence lengths, a closer inspection reveals that the absolute frequency values on the x-axis increase with the sequence length. Specifically, the maximum frequency (for the token " .") rises from approximately 35,000 at max_length=32 to 40,000 at max_length=50, and further to 45,000 at max_length=64.

The observed increase in absolute token frequencies with sequence length can be attributed to the interaction between the max_length parameter and the tokenization process in the script. The **DistilGPT2** tokenizer, which uses Byte-Pair Encoding (BPE), tokenizes the input text into sub-word units, and the script preprocesses the **DailyDialog** dataset into input-response pairs separated by a token. The max_length parameter in the tokenize_function dictates the maximum length of each tokenized sequence: sequences longer than max_length are truncated, while shorter sequences are padded with the end-of-sequence (EOS) token (ID 50256, </s>). In the model, token frequencies are computed by counting all input_id values in the tokenized dataset, which includes both the original tokens, and any padding tokens added to reach max_length. When max_length is smaller (e.g., 32), longer dialogues are more likely to be truncated, reducing the total number of tokens retained in the dataset. Conversely, a larger max_length (e.g., 64) allows more of the original dialogue to be preserved before truncation,

resulting in a higher total token count. However, since the plots provided here do not include the EOS token in the top 20, this suggests that the frequency counts were computed before padding, focusing solely on the lexical tokens. The increase in frequencies with max_length thus reflects the fact that longer sequence lengths preserve more tokens from the original dialogues, leading to higher absolute counts for all tokens while maintaining the same relative distribution.

3.1.2 Token Distribution Characteristics

The token distribution across all three plots exhibits a highly skewed profile; this is a hallmark of natural language datasets. The most frequent token, " .", corresponds to the period (.), reflecting the sentenceending structure prevalent in the dataset's conversational turns. Its frequency increases from 35,000 at max_length=32 to 45,000 at max_length=64, a proportional rise consistent with the overall trend. Following " .", the next most frequent tokens include common conversational words such as "you" (~25,000 to 30,000), "?" (~20,000 to 25,000), "I" (~15,000 to 20,000), and "the" (~15,000 to 20,000), with counts similarly scaling with sequence length. The absence of special tokens like [SEP] or the [EOS] token (</s>) in the top 20 list indicates that the frequency counts were likely computed on the tokenized text before padding, as opposed to the previous run where the [EOS] token (ID 50256) dominated due to padding. The consistent shape of the distribution – where the same tokens appear in the same order with proportional frequency increases – suggests that the underlying lexical composition of the dataset remains stable, and the effect of max_length is to scale the total number of tokens retained by reducing truncation.

3.1.3 Implications for Model Training

The scaling of token frequencies with sequence length has significant implications for training the **DistilGPT2** model on this dataset. A larger max_length (e.g., 64) preserves more of the original dialogue context by minimizing truncation, potentially improving the model's ability to learn longer range dependencies in the data. However, this comes at the cost of increased padding for shorter sequences, which would inflate the frequency of the [EOS] token in the training data (though not reflected in these pre-padding frequency plots). Conversely, a smaller max_length (e.g., 32) reduces padding but risks losing critical context in longer dialogues, which could impair the model's performance on tasks requiring a deeper understanding of conversational flow. The proportional increase in token counts suggests that the model will see more instances of each token as max_length increases, which may enhance learning of frequent patterns (e.g., sentence-ending punctuation, common words) but could exacerbate the underrepresentation of rare tokens due to the long-tail distribution. Researchers should weigh these trade-offs when selecting an appropriate max_length, considering both the computational cost (longer sequences increase memory usage) and the task requirements (e.g., whether longer context is necessary for dialogue generation).

3.1.4 Conclusion

In conclusion, the token distribution plots for sequence lengths of 32, 50, and 64 tokens reveal a consistent shape but a clear scaling of absolute frequencies with max_length. This scaling arises because larger sequence lengths reduce truncation, preserving more tokens from the original dialogues, while the relative distribution remains unchanged due to the fixed tokenization process and dataset structure. The dominance of punctuation and common conversational words underscores the dataset's conversational nature, but the absence of special tokens in the frequency counts highlights a

methodological difference from the previous run, focusing on lexical content before padding. For future work, researchers should explore the impact of max_length on model performance, particularly in terms of context preservation versus computational efficiency, and consider strategies to mitigate the underrepresentation of rare tokens, such as targeted data augmentation or vocabulary adjustments. Additionally, comparing token frequencies before and after padding can provide a more comprehensive view of the data as seen by the model during training.

4.0 Concluding Remarks

This paper has introduced EasyGPT, our first implementation of a transformer using the pre-trained **DistilGPT2** model on the **DailyDialog** dataset. These choices demonstrate a pragmatic approach to dialogue response generation, effectively balancing computational efficiency with performance on midrange PC hardware.

The model, with its 6-layer architecture, 12 attention heads, and 82 million parameters, was strategically adapted by freezing the first four layers to reduce computational demands while fine-tuning the upper layers for task-specific patterns. Additionally, the embedding layer was resized to incorporate a custom [SEP] token, facilitating the processing of input-response pairs, and dropout rates were increased to 0.3 across attention, residual, and embedding layers to mitigate overfitting on the subsampled dataset of 20,000 training and 5,000 validation samples. The experimental strategies were chosen to provide a robust framework to evaluate the model's sensitivity to hyperparameters.

Observations from the model design and experiments highlight both strengths and limitations. The decision to freeze the first four layers effectively reduced the computational burden, enabling training

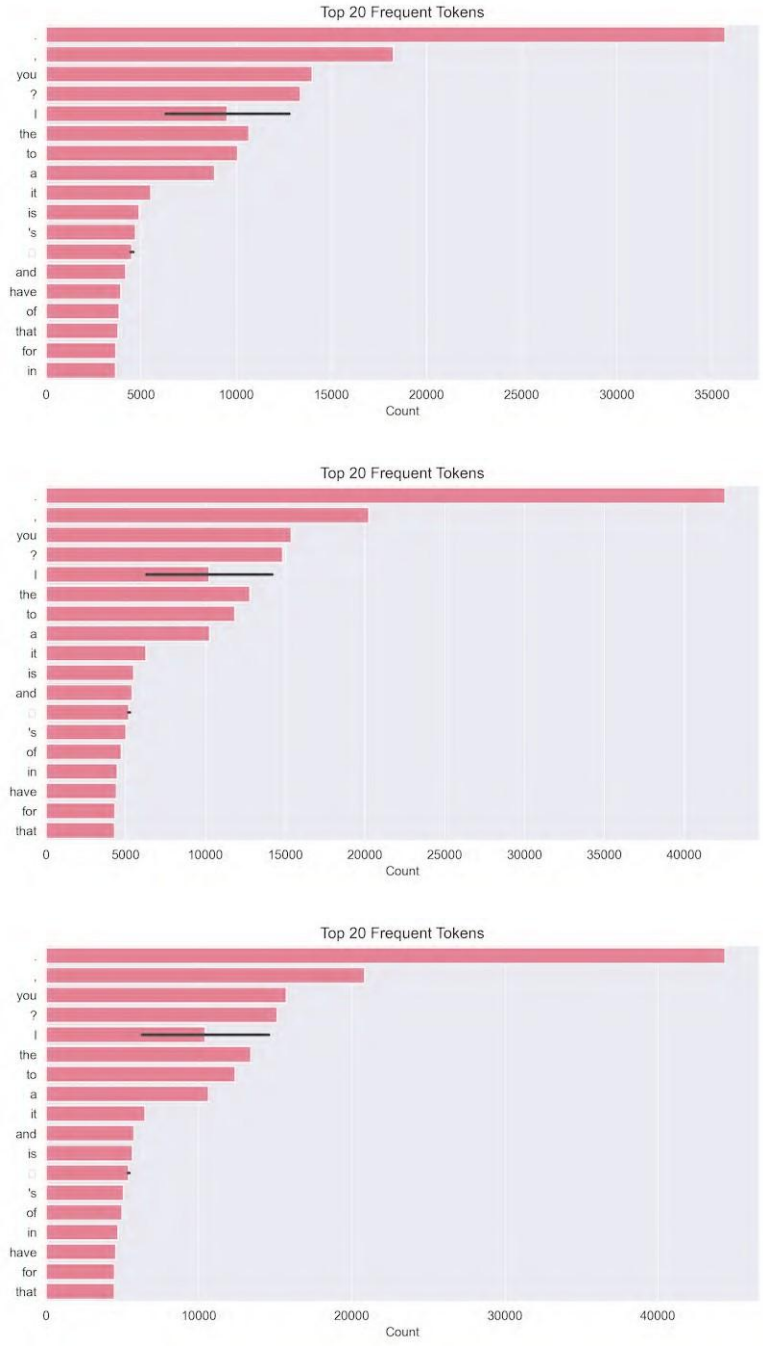


Figure 9: Token Distribution w.r.t. Sequence Length

on mid-range hardware, but it may have constrained the model's ability to adapt deeply to the dialogue task. The token distribution analysis further revealed a highly skewed dataset, with punctuation (e.g., " .") and common words (e.g., "you", "I") dominating frequencies, scaling proportionally with sequence length (from 35,000 to 45,000 for " .") due to reduced truncation at larger max_length values. Our final conclusion is that this implementation achieves efficient qualitative dialogue modeling within hardware constraints, but the results indicate potential underfitting or limited task complexity, as validated by the lower validation loss compared to training loss, warranting further exploration of model capacity and training duration.

4.1 Future Directions

Having now established that it is possible to design, develop and experiment with a viable working transformer model by leveraging existing open-source models and datasets, we plan to proceed along two distinct approaches as follows:

1. Enhancements to the existing model

The following steps will provide a more comprehensive understanding of the model's capabilities and guide refinements for improved dialogue generation performance.

- In the next phase of fine-tuning our model, we plan to extend the training duration and dataset size to better leverage the learning rate schedulers and assess their impact on model performance. Increasing the number of epochs from 3 to 5 or 10 and expanding the training set beyond 20,000 samples (e.g., using the full **DailyDialog** dataset or a larger subsample of 50,000 samples), would allow the model to train longer.
- We will incorporate a plugin system to allow for schedulers to be incorporated into the code. This present implementation did not use scheduling as part of a strategic decision to keep the hyperparameter count optimally low. A scheduler which dynamically adjusts the learning rate based on training progress, could significantly improve the convergence stability, and improve both the performance generalization and training efficiency.
- Once the above are completed, we shall start to unfreeze additional layers to increase the model's capacity to adapt to the task of text generation and monitor how this affects underfitting or overfitting during training.
- We will investigate the token distribution by applying data augmentation to address the underrepresentation of rare tokens – a longer-term goal is to have a plugin system that allows for user-definable datasets.
- Finally, we will evaluate the model's generation quality through qualitative and quantitative measures. A script to generate sample responses from the fine-tuned models for each scheduler will be developed, followed by evaluation using metrics such as BLEU, ROUGE, or human assessment to determine if scheduler differences manifest in the quality of generated dialogue.

2. Variations to the underlying model

EasyGPT was designed from the outset as a modular framework, and it would be relatively easy to swap the **DistilGPT2** model with either existing pre-trained models or even custom ones. As such, it will be possible to use multi-lingual models to allow, for instance, dialogs in other languages, such as Arabic[7], through the use of a multilingual tokenizer and relevant dialog files.

[7] There already exist several pre-trained Arabic NLP models on Hugging Face, e.g., *CAMeLBERT, available at: https://huggingface.co/CAMeL-Lab/bert-base-arabic-camelbert-mix*

We shall report on the outcomes of these planned enhancements and variations in future publications in this series.

REFERENCES

[1] Agile Alliance. (2016). *Introduction to the technical debt concept*. Retrieved April 02, 2025, from https://www.agilealliance.org/introduction-to-the-technical-debt-concept/

[2] Brown, T. B., Mann, B., Ryder, N., Subbiah, M., Kaplan, J., Dhariwal, P., Neelakantan, A., Shyam, P., Sastry, G., Askell, A., Agarwal, S., Herbert-Voss, A., Krueger, G., Henighan, T., Child, R., Ramesh, A., Ziegler, D., Wu, J., Winter, C. and Amodei, D. (2020). *Language models are few-shot learners*. Retrieved from https://arxiv.org/abs/2005.14165

[3] Devlin, J., Chang, M. W., Lee, K. and Toutanova, K. (2019). BERT: Pre-training of deep bidirectional *transformers for language understanding*. In Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)

[4] (pp. 4171-4186). https://www.aclweb.org/anthology/N19-1423

[5] Dufter, P., Schmitt, M., & Schütze, H. (2022). *Position information in transformers: An overview*. Computational Linguistics, 48(3), 733-763.

[6] Hochreiter, S., & Schmidhuber, J. (1997). *Long short-term memory*. Neural Computation, 9(8), 1735–1780

[7] OpenAI. (2022, December 15). ChatGPT: Optimizing language models for dialogue. OpenAI Blog. https://openai.com/blog/chatgpt

[8] Tucudean, G., Bucos, M., Dragulescu, B. and Caleanu, C. D. (2024). *Natural language processing with transformers: A review.* PeerJ Computer Science, 10, e2222. https://doi.org/10.7717/peerj-cs.2222

[9] Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N. and Kaiser, Ł. (2017). *Attention is all you need*. Advances in Neural Information Processing Systems, 30, 5998-6008