Final Pipeline

April 20, 2024

Contents

Executive Summary **4**

Section 1: Introduction **4**

Data Collection . . . 4

Sampling Approach . . . 5

Text Chunking process . . . 5

Section 2: Loading the data **8**

Section 3: Cleaning the text **9**

Widening contractions . . . 9

Correcting misspelled words . . . 13

Bayes' Theorem Application: . . . 13

Markov Chain Model: . . . 13

Text Augmentation . . . 15

External Data . . . 18

Source of the data . . . 18

Why we chose this dataset? . . . 18

Section 4: Task Discussion **22**

What is a LSTM (Long Short-Term Memory) . . . 22

How does it work . . . 23

Previous LSTM results . . . 26

Why did it perform so Poorly? . . . 27

Section 5:Model Selection **28**

What are generative pre-trained transformers? . . . 28

How do transformers work . . . 29

Tokenization . . . 29

Vectorization . . . 30

Token Emedding . . . 31

Positional Embedding . . . 31

Formulas for Positional Encoding: . . . 31

Control of Wavelength: . . . 31

Multihead Attention Matrix 32
Variables: 33
Variables: 33
Multi-head attention 34
Add and Normalize 34
Feed-Forward Network 34
Decoder Attention (in Decoder only) 35
Splitting Training Data 35

Secion 6:Model Training **38**

Section 7: Text Generation and Model Evaluation **53**

Samples of generated Text 53
Text Evaluation 54
Model Evaluation 55

Section 8: Discussions and Conclusions **59**

Section 9: References **59**

Executive Summary

In this final pipeline, our main objective was to engineer a generative model that could generate text that mimics my writing style. To set this off, I start by expounding on the previous strategies used in the second pipeline, explaining why those strategies did not work so well, and articulating how we plan to address them in this final pipeline. Additionally, we improve our pre-processing and text augmentation pipeline and explain the benefits that come with them.

From an architectural standpoint, we implement a generative pre-trained transformer and provide quite a detailed explanation of how it works. Our implementation bore quite promising results, and though I was not able to achieve my objective of generating text that mimicked my writing style, I was quite content since I was able to generate coherent words and phrases.

Finally, I expound on some of the shortcomings of my implementation and provide some future recommendations on how this implementation could be improved.

Section 1: Introduction

This document represents the culmination of our research into developing a model capable of distinguishing between text generated by ChatGPT and text authored by myself. Building upon our initial findings, this final submission delves deeper into the realm of generative models, with a particular focus on refining these models to more closely mimic my personal writing style.

With the advent of advanced neural network architectures, such as Long Short-Term Memory networks (LSTMs) and Transformers, the exploration of text generation has reached new heights. The groundbreaking paper "Attention is All You Need" by Vaswani et al. introduced the Transformer model, which revolutionized our approach to handling sequential data through the mechanism of attention. Inspired by this seminal work, our research aims to harness the capabilities of Transformer models to investigate the extent to which these models can be fine-tuned to replicate specific writing styles.

In this submission, we will explore the application of these sophisticated models, testing their ability to adapt to the nuances of individual writing styles and assessing their effectiveness in producing text that closely resembles my own.

Data Collection

I developed a Python script to segment text from a Google Doc into manageable chunks, subsequently organizing them into a CSV file structured with columns for Text, Source, Origin (human or ChatGPT), Length (measured in characters), and Topic. My approach to data collection involved revisiting all the text I had produced before the advent of ChatGPT. This corpus encompassed everything from college

application essays to assignments and Telegram conversations with friends. I consolidated these texts into a singular Google document, which is then processed by my script to create analyzable segments, each appropriately labeled.

For this final submission, we ommit all the GPT generated data and focus our attention on further increasing the human text through augmentation and later external databases based on the initial perfomance of our models.

Sampling Approach

For the previous submission, I used a methodical approach over random sampling to gather my personal text data. My process involved systematically collecting texts from Google Drive, starting from the earliest backup available and concluding just before the introduction of ChatGPT. However, for this submission, I also include my text post-GPT, acknowledging that it certainly has had an effect on my writing style and changed how I generate text and my ideas. That being said, considering this final submission focuses on generating text that mimics my writing style, I feel my text data post-GPT is appropriate for this.

Text Chunking process

To ensure the segments we add to our CSV are coherent and contextually rich, we use the Natural Language Toolkit (NLTK) for accurate sentence tokenization. Specifically, the nltk.tokenize.sent_tokenize function is utilized for its proficiency in detecting natural language boundaries, enabling us to efficiently divide our text into well-defined, paragraph-chunked sentences.

[ ]:
```
# importing the necesary libraries
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
import re
from sklearn.preprocessing import StandardScaler
import docx
import csv
import os
import nltk
import json
```

```
nltk.download("punkt")


def get_text_chunks(doc_path, min_length=50, max_length=1000):
    # Load the Word document from the specified path
    doc = docx.Document(doc_path)
    # Combine all paragraphs in the document into one string
    full_text = " ".join(para.text for para in doc.paragraphs)

    # Use NLTK to split the full text into sentences
    sentences = nltk.tokenize.sent_tokenize(full_text)

    chunks = []  # Initialize a list to store text chunks
    current_chunk = ""  # Initialize a string to build up a current chunk

    # Iterate through each sentence in the document
    for sentence in sentences:
        # Check if adding this sentence would exceed the max length of a chunk
        if len(current_chunk) + len(sentence) <= max_length:
            current_chunk += (
                " " + sentence
            )  # If not, add the sentence to the current chunk
        else:
            # If the chunk reaches the max length, add it to the list if it meets the␣
↪min length
            if len(current_chunk) >= min_length:
                chunks.append(current_chunk.strip())
            current_chunk = sentence  # Start a new chunk with the current sentence

    # Add the last chunk to the list if it meets the minimum length requirement
    if len(current_chunk) >= min_length:
        chunks.append(current_chunk.strip())
```

```
    return chunks  # Return the list of text chunks



def write_to_csv(chunks, csv_path, topic):
    # Check if the CSV file exists and is non-empty
    file_exists = os.path.isfile(csv_path) and os.path.getsize(csv_path) > 0

    # Open the CSV file in append mode
    with open(csv_path, mode="a", newline="", encoding="utf-8") as file:
        writer = csv.writer(file)  # Create a CSV writer object

        # Write the header row if the file is new or empty
        if not file_exists:
            writer.writerow(["Text", "Source", "Length", "Topic"])

        # Write each chunk as a row in the CSV file
        for chunk in chunks:
            writer.writerow(
                [chunk, "human", len(chunk), topic]
            )  # Include metadata with each chunk

def process_messages_from_json(json_path):
    # Load the JSON file from the specified path
    with open(json_path, "r", encoding="utf-8") as file:
        data = json.load(file)

    messages = []  # Initialize a list to store messages

    # Extract messages from the JSON data
    for message in data["messages"]:
        # Check if the message is of type 'message' and contains text
        if message["type"] == "message" and "text" in message:
```

```
            text = message["text"]
            # If the text is a list, concatenate it into a single string
            if isinstance(text, list):
                text = "".join(
                    [item.get("text", "") for item in text if isinstance(item, dict)]
                    + [item for item in text if isinstance(item, str)]
                )
            messages.append(text)  # Add the processed text to the messages list

    return messages  # Return the list of messages



# Input the topic from the user
topic = input("Please enter the topic: ")

# Specify the path to your Word document and the output CSV file
doc_path = "documents/human reddit_test.docx"
csv_path = "human_reddit.csv"

# Extract text chunks from the Word document and write them to the CSV file
chunks = get_text_chunks(doc_path, min_length=200, max_length=500)
write_to_csv(chunks, csv_path, topic)

print("CSV file has been created with the extracted text chunks.")
```

Section 2: Loading the data

[ ]:
```
 # import statements needed
import pandas as pd
from google.colab import drive
drive.mount('/content/drive')
```

```
Mounted at /content/drive
```

```
[ ]: # importing my original human data
human_df = pd.read_csv('/content/drive/My Drive/Machine Learning/human_data.csv')
human_df.head()
human_df.describe()
```

```
[ ]:         Text
count   3098
unique  2959
top     ['']
freq      79
```

Section 3: Cleaning the text

In our data cleaning process, we employ a series of customized functions aimed at refining the dataset for more efficient analysis and model training. This process is crucial for improving data quality and sharpening the focus of our model.

First we normalize all text to lowercase to ensure uniformity across the dataset. This standardization is essential for the algorithm to recognize variations of the same word, such as "Hello," "HELLO," and "hello," as identical, thus simplifying the text data and reducing the feature space. Additionally, for our generative models, normalizing our text and eliminating any unecessary spaces or special characters reduces noise in our data ensuring that the tokenizer does not parse unnecessary or redundant characters.

Furthermore, for this final pipeline, we sophisticate our preprocessing even further by widening contraction and removing stop words

Widening contractions

We widen contractions in our preprocessing pipeline using the 'contractions' library, which employs a straightforward dictionary-based method. This approach replaces common contractions, such as "can't" with "cannot," effectively standardizing the text and reducing the number of unique words the model needs to learn. This simplification helps in creating a more consistent dataset, which is crucial for the effective training. Though one can also do this using 'pycontractions',a more advanced method that leverages a pre-trained language model in spaCy, I opt for using the 'contractions' since its does reasonably well for the data that we have.

Removing stop words. we remove stopwords like "the", "a", "and", which often add little meaning in our text data. This practice not only helps in reducing the feature space, allowing models to focus on more impactful words, but also minimizes noise, as stopwords frequently appear in text and can obscure

meaningful analysis. However we use this preprocessing step with caution since it can be useful in matching query items thus we hyperparameterize this and comparing the results with and without it.

Finally, we will compare this pre-processing process in comparison to using more powerful tokenizers such as the character level tokenizer and the state-of-the-art Byte-Pair Encoding GPT-2 SubWord tokenizer which can handle more complicated text formats.

[ ]:
```
!pip install contractions > /dev/null 2>&1 && echo "Installation successful!" || echo␣
↪"Installation failed."
```

```
Installation successful!
```

[ ]:
```
# cleaning the data
import re
import contractions



def lowercase_text(df, text_col):
    df[text_col] = df[text_col].apply(lambda x: x.lower()) # Convert text to lowercase
    return df



def remove_special_characters(df, text_col):
    df[text_col] = df[text_col].apply(lambda x: re.sub(r"[^a-z0-9\s,.?!]", "", x)) #␣
↪Remove special characters
    return df



def clean_text(text):
    """Clean text by removing URLs, multiple spaces, leading/trailing spaces, and␣
↪emojis.

    Args:
    text (str): Text to be cleaned.
```

```
    Returns:
    str: Cleaned text.
    """
    import re
    # Remove URLs
    text = re.sub(r"http\S+", "", text)
    # Remove emojis
    emoji_pattern = re.compile("["
                               u"\U0001F600-\U0001FFFF"  # emoticons
                               u"\U0001F300-\U0001F5FF"  # symbols & pictographs
                               u"\U0001F680-\U0001F6FF"  # transport & map symbols
                               u"\U0001F1E0-\U0001F1FF"  # flags (iOS)
                               u"\U00002702-\U000027B0"
                               u"\U000024C2-\U0001F251"
                               "]+", flags=re.UNICODE)
    text = emoji_pattern.sub(r'', text)
    # Remove numbers
    text = re.sub(r'\d+', '', text)
    # Replace multiple spaces with a single space
    text = re.sub(r"\s+", " ", text)

    # Remove punctuation
    text = re.sub(r'[^\w\s]', '', text)

    # Remove leading and trailing spaces
    text = text.strip()
    return text



def apply_text_cleaning(df, text_col):
```

```
    df[text_col] = df[text_col].apply(clean_text)
    return df


def drop_nan_values(df):
    return df.dropna() # Drop rows with NaN values


def expand_contractions(df, text_col):
    # Check if the text column exists in the dataframe
    if text_col not in df.columns:
        raise ValueError(f"The specified column '{text_col}' does not exist in the␣
 ↪dataframe.")


    try:
        # Apply the contraction expansion to each entry in the specified column
        df[text_col] = df[text_col].apply(contractions.fix)
        return df
    except ImportError as e:
        # Handle the case where the 'contractions' library is not installed
        raise ImportError("Failed to import the 'contractions' library. Please ensure␣
 ↪it is installed.") from e
```

[ ]:
```
# cleaning our human data
human_df = drop_nan_values(human_df)
human_df = lowercase_text(human_df, "Text")
human_df = remove_special_characters(human_df, "Text")
human_df["Text"] = human_df["Text"].apply(
    clean_text
)
human_df = expand_contractions(human_df,'Text')
```

[ ]:
```
print(human_df.count())
```

```
Text    3098
dtype: int64
```

Correcting misspelled words

Since a large portion of this text is my own and includes excerpts from telegram conversations, which might contain a lot of slang and typos, I utilize TextBlob's correct() method to effectively address these inaccuracies.

This method leverages a sophisticated language model that uses **Bayes' theorem** in conjunction with a **first-order Markov chain model**. Bayes' theorem is pivotal in updating the probability estimate for a word based on its prior probability and the likelihood of observing the surrounding words.

Bayes' Theorem Application:

The formula for Bayes' theorem in the context of spelling correction is expressed as:

$$P(A|B) = \frac{P(B|A) \times P(A)}{P(B)}$$

where:

- $P(A|B)$ is the **posterior probability** of the correction A given the context B,
- $P(B|A)$ is the likelihood of the context B given the correction A,
- $P(A)$ is the prior probability of the correction A (i.e., how common the correction is),
- $P(B)$ is the probability of the context B (typically constant for comparison purposes).

Markov Chain Model:

The Markov chain model used is a first-order model, where the probability of each word only depends on the previous word in the sequence. This simplifies the context consideration to the immediate word, thereby reducing complexity while still capturing essential contextual clues. The transition probability in a Markov chain model is given by:

$$P(w_n|w_{n-1})$$

where w_n is the nth word and w_{n-1} is the word that immediately precedes it.

In practice, TextBlob's `correct()` method calculates the most probable correction by maximizing the posterior probability $P(A|B)$ across all possible corrections A for a given misspelled word within its context B. It does this by comparing the calculated probabilities for each candidate correction and selecting the correction with the highest posterior probability.

Finally, we include this step in our preprocessing to enhance the overall quality of our input data. This is particularly crucial for improving our data augmentation models, such as Back Translation Augmentation,

which rely on the accuracy and contextual appropriateness of the text to generate meaningful and varied translations. By refining the text at this stage, we ensure that the augmented data is not only diverse but also retains the intended semantic integrity, thereby improving the performance of downstream tasks such as generating text.

[ ]:
```
!pip install TextBlob > /dev/null 2>&1 && echo "Installation successful!" || echo␣
↪"Installation failed."
from textblob import TextBlob
```

```
Installation successful!
```

[ ]:
```
from textblob import TextBlob
from tqdm import tqdm


def correct_typos_textblob(df, text_col, file_path):
    """
    Corrects typos in the specified column of a DataFrame using the TextBlob library␣
↪and saves the updated DataFrame to a file.
    """
    # Check if the text column exists in the dataframe
    if text_col not in df.columns:
        raise ValueError(f"The specified column '{text_col}' does not exist in the␣
↪dataframe.")


    try:
        # Import TextBlob inside the try block to handle ImportError if not installed
        from textblob import TextBlob


        # Apply the typo correction to each entry in the specified column with a␣
↪progress bar
        tqdm.pandas(desc="Correcting typos")  # Enable tqdm for pandas apply
        df[text_col] = df[text_col].progress_apply(lambda text: TextBlob(text).
↪correct().string)
```

```
        # Save the updated dataframe to the specified file path
        df.to_csv(file_path, index=False)
    except ImportError as e:
        # Handle the case where the TextBlob library is not installed
        raise ImportError("Failed to import the 'TextBlob' library. Please ensure it␣
↪is installed.") from e
    except Exception as e:
        # Handle other unexpected exceptions
        raise Exception(f"An error occurred: {str(e)}") from e
```

[ ]:
```
human_df = correct_typos_textblob(human_df,'Text', '/content/drive/My Drive/Machine␣
↪Learning/updated_human_df.csv')
```

Text Augmentation

This section is quite similar to the one in the last pipeline.

As we observe in the section above, we only have 2321 data points and in order to have better performance from our generative models, our data would need to be more robust in terms of the vocavulary used. This is because a small dataset acts as an under-constraining set of examples, failing to provide a diverse and comprehensive representation of the real-world variability.

To augment the text, I employ three distinct techniques designed to generate a diverse array of augmented text versions, each contributing uniquely to enhancing the dataset for model training.

- The first technique involves the use of **Contextual Word Embeddings Augmentation**, leveraging distilbert-base-uncased for substituting words in the text with their contextual embeddings. This approach modifies approximately 65% of the text, introducing synonyms and contextually relevant replacements that maintain the original meaning while varying the expression.
- The second technique, **TextGenie Augmentation (textgenie)**, utilizes a paraphrasing model, specifically ramsrigouthamg/t5_paraphraser combined with bert-base-uncased, to reformulate sentences. This method is weighted more heavily in my augmentation pipeline since it provided very good augmented samples in comparison to the other two models.
- Lastly, **Back Translation Augmentation (bt)** employs a two-step translation process using facebook/wmt19-en-de and facebook/wmt19-de-en models for English to German and German back to English translations, respectively. This technique translates the text to a different language and

then back to the original language. While the core information is preserved, subtle nuances in translation introduce variability in phrasing and sentence structure, contributing further to the diversity of the augmented text.

Using this pipeline I was able to generate **3098** new augmented samples significantly hacking up my human text class.

[ ]:
```
!pip install textgenie > /dev/null 2>&1 && echo "Text Genie Installation successful!"␣
↪|| echo "Installation failed."
!pip install nlpaug torch transformers sacremoses pandas textgenie > /dev/null 2>&1 &&␣
↪echo "Text Genie Pandas Installation successful!" || echo "Installation failed."
```

```
Text Genie Installation successful!
Text Genie Pandas Installation successful!
```

[ ]:
```
import pandas as pd
import nlpaug.augmenter.word as naw
import random
from textgenie import TextGenie
from transformers import AutoTokenizer
import time
from tqdm import tqdm  # Import tqdm


# Assuming df is your DataFrame and it has been loaded


# Set up the models and tokenizer
aug_emb = naw.ContextualWordEmbsAug(
    model_path='distilbert-base-uncased',
    action='substitute',
    aug_p=0.65
)


textgenie = TextGenie("ramsrigouthamg/t5_paraphraser", 'bert-base-uncased')


model_en_de = "facebook/wmt19-en-de"
```

```
model_de_en = "facebook/wmt19-de-en"
aug_bt = naw.BackTranslationAug(
    from_model_name=model_en_de,
    to_model_name=model_de_en
)

tokenizer = AutoTokenizer.from_pretrained("distilbert-base-uncased")

augmented_rows = []
total_rows = len(human_df)
csv_file_path = 'augmented_text_versions.csv'

# Ensure the file is empty or set up with the correct headers initially
pd.DataFrame(columns=['Text']).to_csv(csv_file_path, index=False)

def truncate_text(text, max_length=512):
    tokens = tokenizer.tokenize(text)
    if len(tokens) > max_length - 2:
        tokens = tokens[:max_length - 2]
    return tokenizer.convert_tokens_to_string(tokens)

# Wrap human_df.iterrows() with tqdm for a progress bar
for index, row in tqdm(human_df.iterrows(), total=total_rows, desc="Processing rows"):
    text = row['Text']
    truncated_text = truncate_text(text)

    choice = random.choices(
        population=["emb", "textgenie", "bt"],
        weights=[0.25, 0.5, 0.25],
        k=1
    )[0]
```

```
    if choice == "emb":
        augmented_text = aug_emb.augment(truncated_text)
        augmented_rows.append({'Text': augmented_text})
    elif choice == "textgenie":
        augmented_texts = textgenie.augment_sent_t5(truncated_text, "paraphrase: ",␣
↪n_predictions=2)
        for aug_text in augmented_texts[:2]:
            augmented_rows.append({'Text': aug_text})
    elif choice == "bt":
        augmented_text = aug_bt.augment(truncated_text)
        augmented_rows.append({'Text': augmented_text})


    # Every 100 rows, append to the CSV and clear the list
    if (index + 1) % 100 == 0 or (index + 1) == total_rows:
        pd.DataFrame(augmented_rows).to_csv(csv_file_path, mode='a', header=False,␣
↪index=False)
        augmented_rows.clear()  # Clear the list after writing
```

External Data

From observing the results we had in the second pipeline when attempting to generrate datat using the LSTM mode we noticed that our 3098 dataset was not enough to provide good results and we would need much much more data. Consequently, in this pipeline we use external data containing human written data and ingest it with the intention of getting much better results this time.

Source of the data

The first source we select is gotten from a kaggle dataset that comprises about 10,000 essays, some written by students. All of the essays were written in response to one of seven essay prompts from which the students were instructed to read one or more source texts before providing a response.

Why we chose this dataset?

Finding a good dataset can be challenging as most of the ones I found had several repeated entries to disguise how extensive they are, which is quite detrimental since our model would easily overfit. Additionally, most datasets tend to be domain-specific data, and this further skews our model's text generation. Consequently, considering this dataset collects responses from 7 prompts, it provides a robust

and diverse set of samples for our pipeline's purpose.

[ ]:
```
# importing my original human data
external_human_df = pd.read_csv('/content/drive/My Drive/Machine Learning/
↪filtered_output.csv')
external_human_df.head()
```

[ ]:
```
          id  prompt_id                                               text  \
0  0059830c          0  Cars. Cars have been around since they became ...
1  005db917          0  Transportation is a large necessity in most co...
2  008f63e3          0  "America's love affair with it's vehicles seem...
3  00940276          0  How often do you ride in a car? Do you drive a...
4  00c39458          0  Cars are a wonderful thing. They are perhaps o...

   generated
0          0
1          0
2          0
3          0
4          0
```

[ ]:
```
# renaming columns
external_human_df = external_human_df.rename(columns={'text': 'Text', 'generated':␣
↪'Source'})



# Replace 'Source' column
external_human_df['Source'] = external_human_df['Source'].replace(0, 'human')



# Create the 'Length' column
external_human_df ['Length'] = external_human_df ['Text'].apply(lambda x: len(str(x)))


# Create the 'Topic' column
```

```
external_human_df ['Topic'] = 'external_human'


# Clean the 'Text' column to ensure all are strings
external_human_df ['Text'] = external_human_df ['Text'].astype(str)


# Remove rows with NaN values in 'Text' column
external_human_df.dropna(subset=['Text'], inplace=True)



# reomve columns
external_human_df = external_human_df.drop(['id', 'prompt_id'], axis=1)


external_human_df.head()
```

[ ]:

	Text	Source	Length	\
0	Cars. Cars have been around since they became ...	human	3289	
1	Transportation is a large necessity in most co...	human	2738	
2	"America's love affair with it's vehicles seem...	human	4428	
3	How often do you ride in a car? Do you drive a...	human	4013	
4	Cars are a wonderful thing. They are perhaps o...	human	4698	

	Topic
0	external_human
1	external_human
2	external_human
3	external_human
4	external_human

[ ]:
```
# cleaning the external human data
external_human_df  = lowercase_text(external_human_df , "Text")
external_human_df  = remove_special_characters(external_human_df , "Text")
external_human_df["Text"] = external_human_df ["Text"].apply(
    clean_text
```

```
)
external_human_df  = expand_contractions(external_human_df ,'Text')
```

[ ]:
```
external_human_df.head()
```

[ ]:
```
                                                Text Source  Length  \
0  cars cars have been around since they became f...  human    3289
1  transportation is a large necessity in most co...  human    2738
2  americas love affair with its vehicles seems t...  human    4428
3  how often do you ride in a car do you drive a ...  human    4013
4  cars are a wonderful thing they are perhaps on...  human    4698

            Topic
0  external_human
1  external_human
2  external_human
3  external_human
4  external_human
```

[ ]:
```
combined_df = pd.concat([external_human_df, human_df], ignore_index=True)
combined_df.head()
```

[ ]:
```
                                                Text Source  Length  \
0  cars cars have been around since they became f...  human  3289.0
1  transportation is a large necessity in most co...  human  2738.0
2  americas love affair with its vehicles seems t...  human  4428.0
3  how often do you ride in a car do you drive a ...  human  4013.0
4  cars are a wonderful thing they are perhaps on...  human  4698.0

            Topic
0  external_human
1  external_human
2  external_human
3  external_human
```

```
4  external_human
```

Section 4: Task Discussion

The primary objective of this pipeline is to develop a generative model capable of emulating the specific writing style of an individual using a specialized version of the Generative Pre-trained Transformer, specifically the GPT-2 model. I selected it specifically because it utilizes the attention mechanism introduced by Vaswani et al., 2017. Given the seminal impact of the "Attention is All You Need" paper I am curious to see if we will make improvements in this final pipeline.

However, before we delve deeper into these advanced models I think it is important for us to take a look at the generative models we explored in the second pipeline and why they did not produce good results.

In my second pipeline, I used stacked LSTMs in an attempt to generate text for augmenting purposes, and my results were quite poor. To gather an understanding of how these LSTMS work, below is an excerpt from the previous pipeline clearly outlining how they work(feel free to skip this next section).

What is a LSTM (Long Short-Term Memory)

Note: The LSMT explanation is the same as the last pipeline

.

An LSTM is a recurrent neural network that is designed to avoid the vanishing gradient problem. The vanishing gradient problem to be specific is a challenge encountered in the training of artificial neural networks, particularly evident in traditional recurrent neural networks (RNNs) and deep feedforward networks. It occurs when the gradients of the network's loss function decrease exponentially as they are propagated backward through the network's layers during training. This gradient diminution leads to very small updates to the weights of the network's early layers, which effectively halts the learning process for these parts of the model.

How does it work

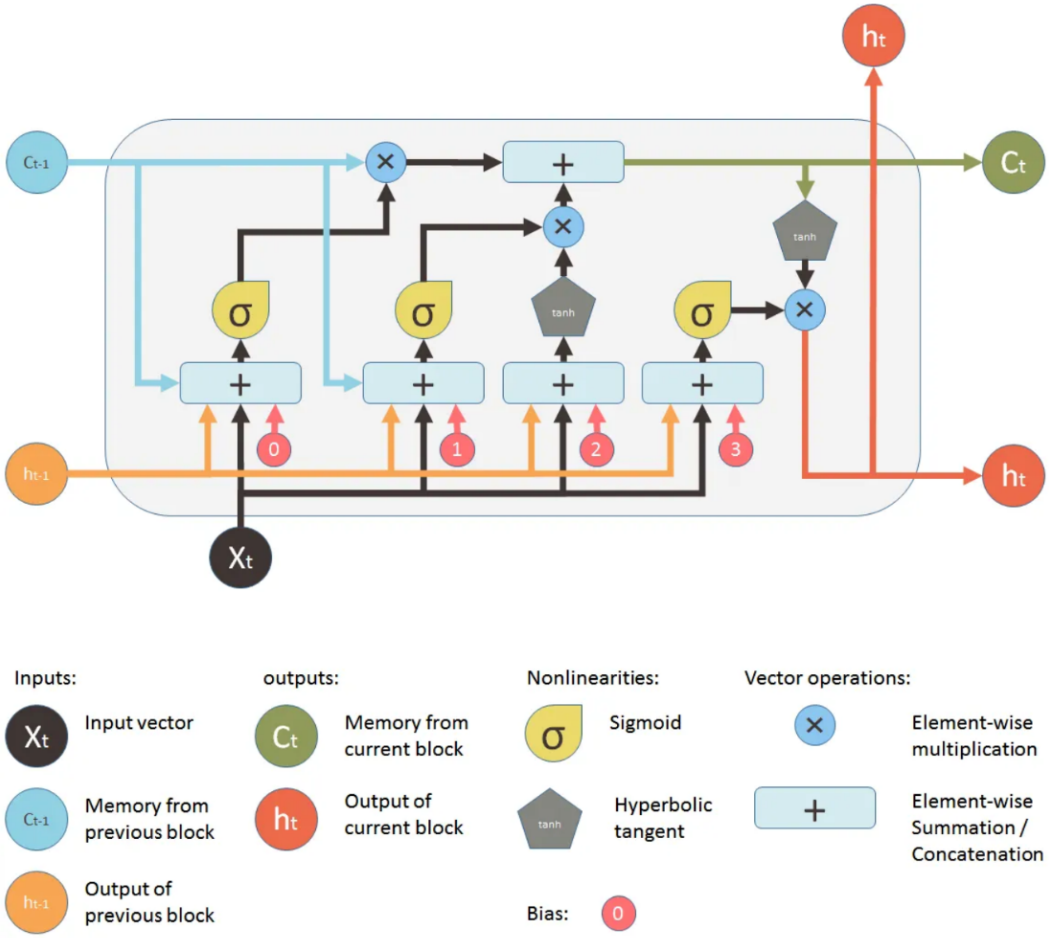


Figure 2

Source. Architectural representation of a LSTM unit.

From the image above the blue line represents the memory from the previous block and this is what we denote as the 'long term' memory in our LSTM model. Normally the blue line will not have any weights and biases to ensure that our gradient does not explode or vanish. Conversely, the orange line represents our short-term memory and the the black circle represents our input vector.

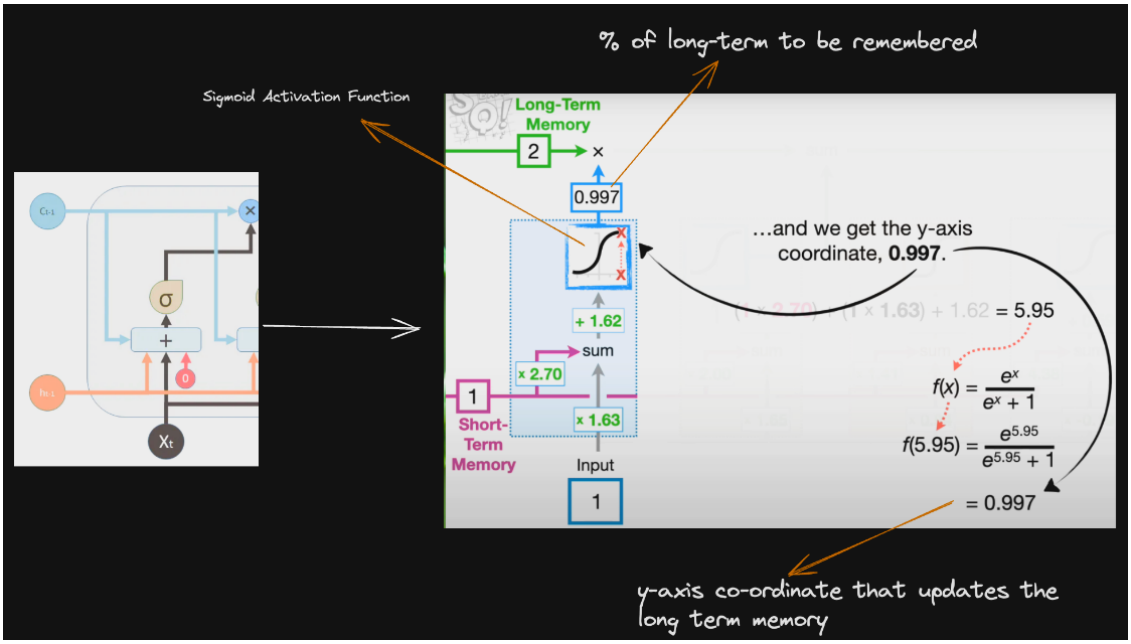


Figure 3

Source. Architectural representation of a LSTM unit.

Getting more granular in LSTMs, what essentially happens is that the input gate's activation involves combining the current input (x_t) and the previous short-term memory (h_{t-1}), each multiplied by their respective weights plus a bias term, and then applying a sigmoid activation function. This can be mathematically represented as follows:

$$i_t = \sigma(W_{xi}x_t + W_{hi}h_{t-1} + b_i)$$

Note that the sigmoid activation function maps our x-axis coordinate to a y-axis coordinate between 0 and 1.

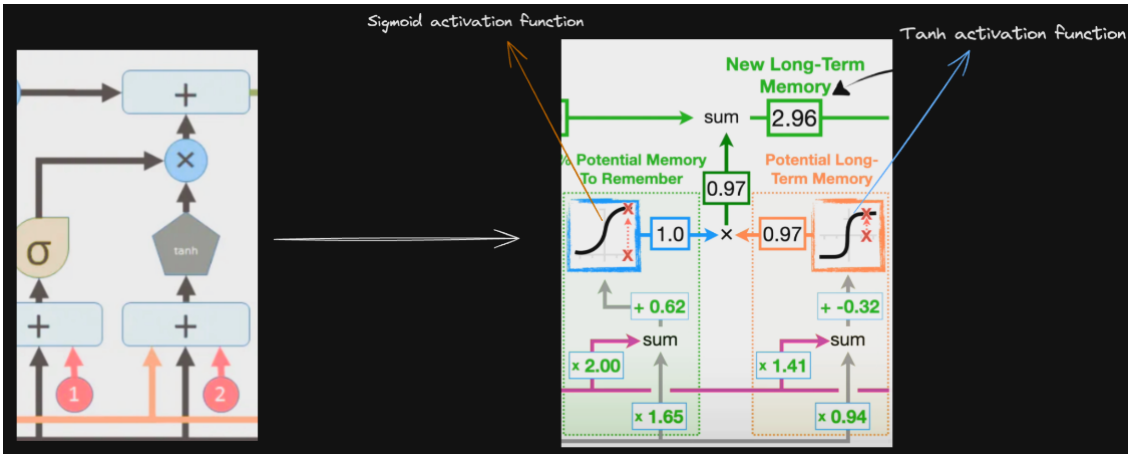


Figure 4

Low-level representation of the second segment of an LSTM unit.

In this second LSTM segment, we still take the short-term memory (h_{t-1}) and current input (x_t) multiplied by their respective weights. However, in this case, we are interested in calculating the new long-term memory, (C_t).

This is done by two block sections (the ones in green and orange in the image above), where one calculates the potential long-term memory, whereas the other contributes to what percentage of the long-term memory to remember.

This calculation is orchestrated by two critical block sections (green and orange in Figure 6 above), serving distinct yet complementary roles:

1. **The candidate long-term memory creation(Orange block)** which employs the ($tanh$) activation function to generate a vector of candidate values, ($tilde C_t$), for updating the long-term memory. This step is crucial for introducing non-linearity and ensuring that the potential updates to the long-term memory are normalized within a range of -1 to 1.

2. **The memory update gate** which determines the extent to which each unit of the long-term memory is updated. This involves calculating what percentage of the old long-term memory (C_{t-1}) to retain versus how much of the new candidate memory ($tilde C_t$) to add.

The integration of these components leads to the formulation of the new long-term memory as follows:

$$C_t = f_t * C_{t-1} + i_t * \tilde{C}_t$$

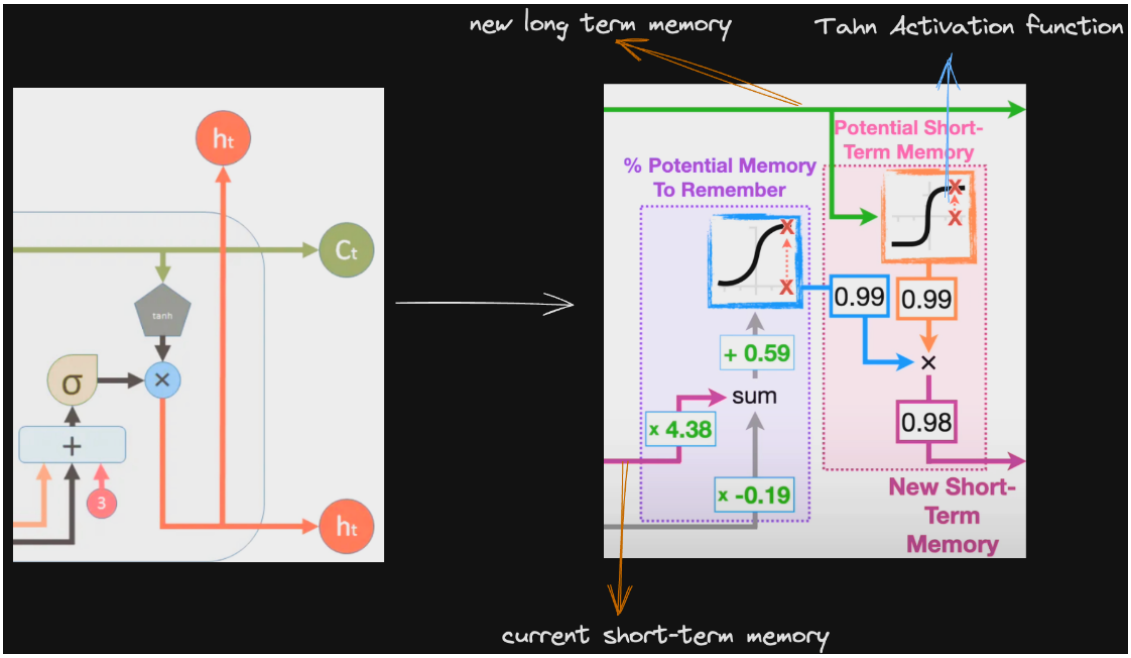


Figure 5

Low-level representation of the second segment of an LSTM unit.

.

Finally, in this third segment, our focus is to derive a new short-term memory. We do this by taking the new long-term memory and using it as an input in the Tahn activation function. Moreover to determine the % of the potential short-term memory to retain we use the same approach we used in segment two to determine the % of the long-term memory to retain.

In our case of generating text that mimics my writing style, LSTM units synergize to update and utilize memories, effectively capturing and emulating the unique nuances of the desired style. By expanding the sequence with additional LSTM units, the model learns from longer contexts, ensuring the output remains consistent and coherent.

Previous LSTM results

Note: This is the new LSTM section

. As mentioned earlier, the results from our previous stacked LSTM model were poor and one can evidently see this below

> **Seed:** "exploring the depths of machine learning, one model at a time, unraveling the mysteries within data."
>
> **Generated output:** the context of the contract of the contract of the contract of the contract

of the contract of the contract of the contract of the contract of the contract of the contract of the contract of the contract of the contract of the cont

Generated output with some randomization: looks what hesrlenry filyensing rogt gorco and linao landuawe the clanse between srowed in the reaieiment moic work uotitya dilent types for even and nomes, oy risk parsinn and toullen. serure ouport Done.

Why did it perform so Poorly?

First, LTSMs though designed to address the vanishing gradient problem still seem to struggle when the next word to be generated has an extremely long dependency. Additionally, in our case, our dataset has quite varied lengths of text from essays to text messages. Consequently, sentences that are much shorter or much longer than the ones the LSTMs have been trained one will give poor results since they will have to be cut off or padded with 0s.

Secondly, the way we were selecting the next word to be processed was also equally poor. In our previous implementation, we would take the argmax of the prediction output and this is what casued most of the repetitive phrases we observe in the results above. Investingating this issuse, I believe this happened becaue in LSTMS, the probability of a repeated phrase increases with each repeition creating a postiie feedback loop. With regards to my writing style, this could be interpreted phraess that I use quite frequently.

To remedy this instead of using the argmax, I intend to treat the output of the model as a probability distribution and sample from that.

More formally, to produce the probability distribution, after the decoder produces output logits for a timestep, these logits are passed through a softmax function to convert them into probabilities that sum to one. Each element in this output array represents the probability of the corresponding token being the next part of the sequence.

The softmax output from the decoder is a vector

$$\mathbf{p} = [p_1, p_2, ..., p_n]$$

where (p_i) is the probability of the i_th token in the vocabulary being the next token in the sequence.

The probabilities are calculated using the softmax function with temperature T:

$$p_i(T) = \frac{e^{\frac{z_i}{T}}}{\sum_{j=1}^{n} e^{\frac{z_j}{T}}}$$

where:

- $p_i(T)$ is the probability of the ith token in the vocabulary being the next token in the sequence, adjusted by the temperature T.
- $e^{\frac{z_i}{T}}$ represents the exponential of the logit z_i divided by the temperature T. Adjusting the logit by the temperature modifies the sharpness of the probability distribution:
  - A higher temperature ($T > 1$) makes the distribution smoother, giving lower probability tokens a higher chance of being selected. This can increase diversity in the generated text.
  - A lower temperature ($T < 1$) makes the distribution sharper, where higher probability tokens are even more likely to be chosen. This can increase the coherence and predictability of the generated text.
- z_i is the logit or the raw prediction value for the ith token. Logits are the outputs of the last neural network layer before applying the softmax function and can have any real value (positive, negative, zero).
- $\sum_{j=1}^{n} e^{\frac{z_j}{T}}$ is the sum of the exponentials of all logits adjusted by temperature T. This sum acts as a normalization constant, ensuring that the probabilities $p_i(T)$ sum to 1 across all possible tokens.

Given this output, we will then sample the distribution using the temperature as a parameter to adjust the diversity and predictability of our text generation. The temperature parameter effectively modifies the softmax function, allowing us to control the randomness of the sampling process. This will allow us to potentially solve for the repeating phrases we had before.

Finally, we will not include this strategy in our old stacked LSTM model but rather we shall take this approach when working with the transformer that we shall build in the next section.

Section 5:Model Selection

For this final pipeline, we utilize state-of-the-art generative pre-trained transformers hoping that we will be able to get reasonable results in terms of text generation.

What are generative pre-trained transformers?

Generative pre-trained transformers(GPT) are a type of artificial intelligence model designed to generate text that mimics human writing. Looking at the model initials, the final letter T, which stands for transformers is actually the heart and soul of these models.

A transformer is a deep-learning architecture developed by Google and based on the multi-head attention mechanism, proposed in a 2017 paper “Attention Is All You Need”.

How do transformers work

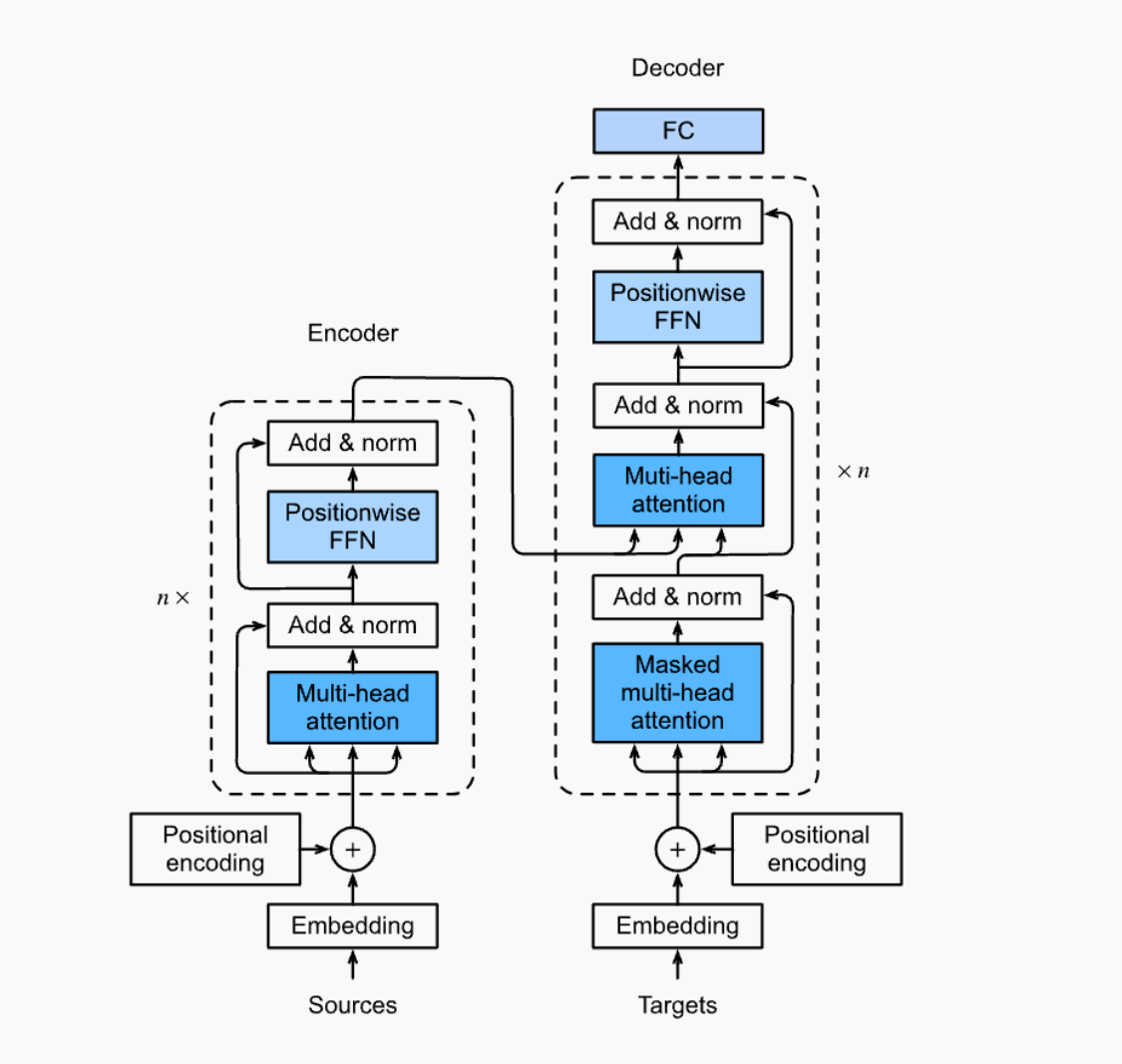


Figure 6

Transformer architecture

.

Tokenization

The first thing to note when working with transformers is that the input (text in our case) is converted into tokens, which are discrete elements that the model can process. To be specific, tokenization involves dividing text into smaller units called tokens, which can be words, phrases, subwords, or characters. In our model, we use character-level tokenization which is showcased in the image below.

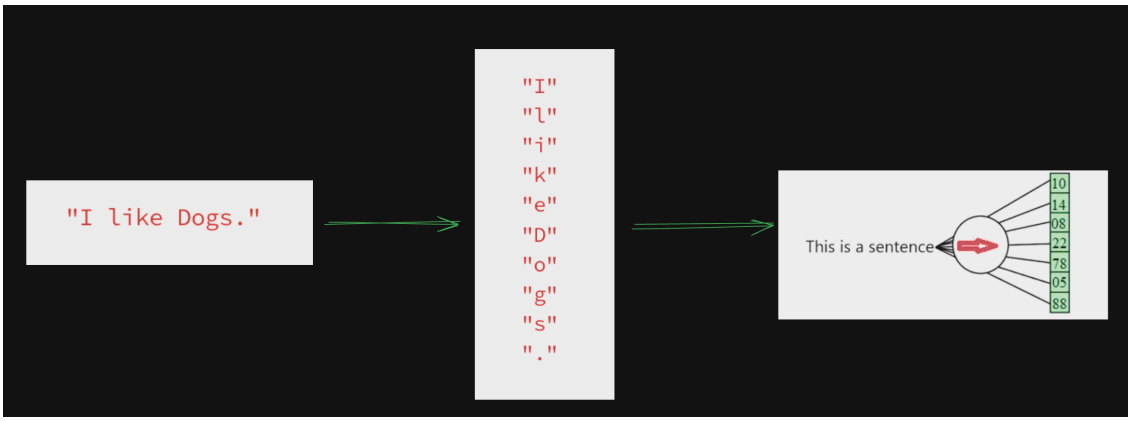


Figure 7

Tokenization

.

Once the words are split into characters, each character is treated as an individual unit or token by the model. This step is critical for transformers since dividing the text into smaller units helps the model to identify the underlying structure of the text and process it more efficiently. However, because GPTs are a variant of neural networks, they need numerical data to perform computations. Consequently, as one can observe in point 6 above, we convert these characters into numerical indices. This conversion is facilitated through the use of the stoi (string-to-integer) mapping, allowing each character to be represented as a distinct integer. These indices are not random but are systematically assigned based on the character's position in the sorted list of unique characters, ensuring a consistent and reproducible mapping across different sessions or experiments.

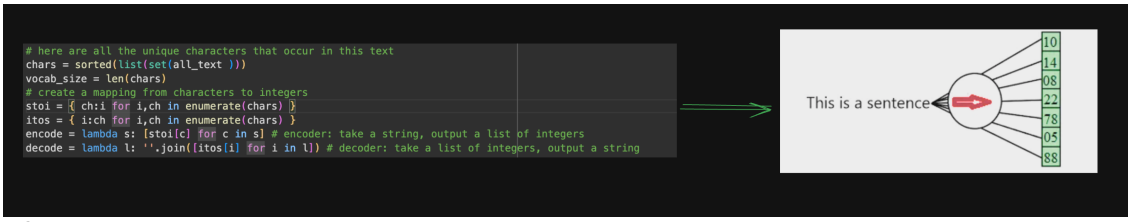


Figure 8

Code Snippet of the character to integer conversion

.

Vectorization

Once we have generated these tokens we can now convert them into embeddings. In simpler terms, embeddings are high-dimensional vectors. Particular to our transformer architecture, we have two

instances where we use embeddings. The first is token embedding and the second is positional Embedding.

Token Emedding

For token embedding first, we list all unique tokens (like words or characters) that our model should recognize. This list is called the vocabulary. Using these unique characters, we then create an "embedding matrix" which is a table where each row corresponds to a token in the vocabulary. Conversely, each column in this matrix represents the size of the vector we want to represent each token.

Positional Embedding

However, token embeddings do not tell us anything about the relationships and dependencies between different tokens within a sequence. They primarily provide a dense representation of individual tokens, capturing some intrinsic properties like semantic and syntactic features. Consequently once we tokenize(as shown in figure 5) the next step is to perform some positional encoding.

Positional encoding in transformers is usually generated using a fixed mathematical formula involving sine and cosine functions of different frequencies. These functions are used because they provide a way to encode position information that is easily learnable and distinguishable by the model across different sequence lengths. Mathematically, Positional encodings (PE) are vectors added to the token embeddings. For each dimension of the embedding, the positional encoding uses a sinusoid (either sine or cosine) that depends on the position and the dimension index.

Formulas for Positional Encoding:

- **For even indices:**

$$[PE(pos, 2i) = \sin\left(\frac{pos}{10000^{2i/d}}\right)$$

- **For odd indices:**

$$[PE(pos, 2i+1) = \cos\left(\frac{pos}{10000^{2i/d}}\right)$$

Variables:

- (pos) - the position in the sequence.
- (i) - the dimension index within the embedding.
- (d) - the total number of dimensions in the positional encoding (typically the same as the embedding size).

Control of Wavelength:

- The term $(\frac{10000^{2i/d}}{pos})$ controls the wavelength of the sinusoids, allowing the model to access frequencies varying systematically across the embedding dimensions.

The reason we use the sin and cosine in our calculations for positinal encoding is because of:

- **Periodicity:** Sine and cosine functions encode positions with a repeating pattern, making them ideal for modeling word relationships within a sequence. For instance, they enable systematic attention to words at set intervals (e.g., every 5th word), enhancing the model's ability to apply attention mechanisms effectively across both short and long sequences.
- **Constrained Values:** The outputs of sine and cosine are limited to between -1 and 1. This constraint stabilizes the learning process by preventing positional encodings from introducing overly large values into the model. It ensures that positional information remains normalized and does not dominate the token embeddings. Moreover, it provides a controlled variation between consecutive positions, aiding the model in recognizing subtle positional differences.
- **Ease of Extrapolation Over Long Sequences:** The smooth and continuous nature of these functions allows the model to extrapolate positional information for sequence positions beyond the training data. This capability is crucial for handling sequences longer than those seen during training, ensuring consistent positional context across varying lengths.

Multihead Attention Matrix

Having an understanding of the positional encoding of our tokens, we can now use this to weigh the importance of different parts of input data allowing models to focus on the most relevant parts of data when making predictions.

Before we even delve into the multihead attention matrix I think it is important to first create an understanding of what an attention matrix is.

The fundamental idea behind the attention mechanism in the context of sequence processing (like in language tasks) is to compute a weighted sum of a sequence of vectors, where the weights are dynamically calculated based upon the input itself. In other words,having a high dimensional vector with the reference and position of a token, we want to output a refined set of embeddings that have ingested some meaning from the input given.

The first step to doing this is to generate the Query (Q), Key (K), and Value (V) matrices.

Query Matrix (Q): The Query matrix is derived by applying a learned linear transformation to the embeddings. Queries are meant to represent the elements for which we are trying to compute the attention.

$$Q = XW^Q$$

Key Matrix (K): Similarly, the Key matrix is also derived by applying a different learned linear

transformation to the embeddings. Keys are used to compute the matching score with Queries.

$$K = XW^K$$

Value Matrix (V): The Value matrix is derived using yet another learned linear transformation. Values are the representations that are actually summed up to create the output, weighted by the computed attention from Queries and Keys.

$$V = XW^V$$

Variables:

- (X) - the input embeddings matrix, containing the embedded representations of the input data.
- (W^Q) - the weight matrix for the Query, a learned parameter that projects the input embeddings into the Query space.
- (W^K) - the weight matrix for the Key, a learned parameter that projects the input embeddings into the Key space.
- (W^V) - the weight matrix for the Value, a learned parameter that projects the input embeddings into the Value space.

To determine the similarity between each key and query, we compute the dot product of each key-query pair. In this context, a larger dot product indicates greater alignment between the key and the query. In machine learning terms, this alignment suggests that certain embeddings (represented by the keys) are particularly relevant or "attending" to specific queries. This attention mechanism is essential for models to focus on the most pertinent information in tasks involving large or complex datasets.

The dot products are normalized across each column using the softmax function, resulting in a probability distribution that indicates the relevance of each word to the query. This all culminates to the attention formula which is define as:

The attention mechanism formula is defined as:

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

Variables:

- (Q) : Query matrix, where each row represents a query vector.

- (K): Key matrix, where each row represents a key vector.
- (V): Value matrix, where each row represents a value vector.
- (d_k) : Dimension of the key vectors, used for scaling the dot products before applying the softmax. This scaling factor helps in stabilizing the gradients during training.

Multi-head attention

Having an understanding of the intricacies of the single-head attention mechanism, the multi-head Attention splits the attention mechanism into multiple 'heads'. It does this by creating various sets of Query (Q), Key (K), and Value (V) matrices. Each set is produced by applying different linear transformations (using trainable weights) to the input. This allows the model to simultaneously attend to information from different representation subspaces at different positions. Essentially, it's like having multiple independent attention layers running in parallel.

Within each head, the attention is computed independently using the formula:

$$\text{Attention}(Q_i, K_i, V_i) = \text{softmax}\left(\frac{Q_i K_i^T}{\sqrt{d_k}}\right) V_i$$

Here, (Q_i), (K_i), and (V_i) represent the query, key, and value matrices respectively for the (i)-th head. The softmax function is applied over the keys for each query to generate a weighted sum of the values.

Concatenation of Outputs After computing the attention for each head, the outputs are concatenated along the dimensionality axis. This concatenation combines the diverse aspects learned by each head into a single output matrix.

Final Linear Transformation Finally, The concatenated outputs are then transformed through a final linear layer. This transformation is crucial as it mixes the information from all the heads together, integrating the different perspectives into a unified output that can be processed further by the network.

Add and Normalize

After the attention layer, the output from the Multi-Head Attention layer is looped back and added to its original input, which helps keep the dimensions consistent and fights the vanishing gradient issue, making it easier for the model to learn. Right after this step, layer normalization is applied to ensure that the inputs across the layers are standardized. This normalization not only helps stabilize the training process by making the model less sensitive to changes in the initial learning rate but also boosts the model's ability to generalize from training data, ultimately making the training process more effective and robust.

Feed-Forward Network

- **Position-wise Feed-Forward Network**: Each position in the encoder or decoder output(check figure 5) is processed through the same feed-forward network, ensuring that the transformation

applied is consistent yet capable of addressing unique position-based nuances. This network comprises two linear transformations with a ReLU activation in between. The formula for the feed-forward network is expressed as:

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

In this equation:

- (W_1) and (W_2) are the weight matrices for the first and second linear transformations, respectively.
- (b_1) and (b_2) are the biases for these transformations.

This configuration introduces non-linearity into the process, which is crucial for learning complex patterns. Each position's output is independently calculated, allowing the network to effectively adapt its responses across different parts of the input.

Decoder Attention (in Decoder only)

Finally, after the extensive processing through Multi-Head Attention and the position-wise feed-forward networks, the Transformer's decoder applies one more layer of normalization. This normalization not only ensures that the learning remains stable and consistent but also sets the stage for one of the most crucial aspects of the Transformer architecture—the encoder-decoder attention layer.

Encoder-Decoder Attention Exclusively featured in the decoder, this specialized layer utilizes queries derived from the output of the previous decoder layer, while the keys and values are sourced from the full breadth of the encoder's output. This unique setup allows every position in the decoder to fully engage with every part of the input sequence. Such meticulous attention ensures that the decoder can integrate and synthesize all available information as it constructs the output step-by-step. This level of detailed attention is essential, especially in tasks like translation, where a comprehensive understanding of the entire input sequence is vital to generate precise and fluid translations.

Splitting Training Data

For this dataset, we start off with a random 80/20 split. This initial division ensures that 80% of the data is used for training the model, providing it with a broad range of examples to learn from, while the remaining 20% serves as the validation set, which will help in assessing the model's performance on unseen data. In the context of text generation, the quantity and diversity of training data are pivotal—more data enables the model to learn a wider array of linguistic patterns, phrases, and stylistic nuances, enhancing its ability to generate coherent and contextually relevant text. As the model trains on this extensive dataset,

it incrementally improves its language model, learning to predict and generate sequences of text that are both plausible and engaging to the reader.

[ ]:
```
# Step 1: Concatenate Text from DataFrame
import torch



def concatenate_text(df, column_name):
    print("Concatenating text from DataFrame...")
    all_text = ''.join(df[column_name].tolist())
    return all_text




def prepare_text_data(all_text):
   """
   Prepare character-to-index mappings and split text data into training and␣
 ↪validation sets.



   Args:
   all_text (str): The entire text input from which characters are extracted and␣
 ↪indexed.



   Returns:
   tuple: A tuple containing training and validation data tensors, and the mappings␣
 ↪from
          characters to integers and integers to characters.
   """
   print("Preparing text data...")
   # Extract all unique characters and create mappings
   chars = sorted(list(set(all_text)))
```

```
    vocab_size = len(chars)
    print(f"Found {vocab_size} unique characters.")

    stoi = {ch: i for i, ch in enumerate(chars)}
    itos = {i: ch for i, ch in enumerate(chars)}


    # Encode the entire text
    print("Encoding text...")
    encode = lambda s: [stoi[c] for c in s]
    data_tensor = torch.tensor(encode(all_text), dtype=torch.long)


    # Split into training and validation sets
    n = int(0.9 * len(data_tensor))
    train_data = data_tensor[:n]
    val_data = data_tensor[n:]
    print(f"Data split into {len(train_data)} training samples and {len(val_data)}␣
 ↪validation samples.")


    return train_data, val_data, stoi, itos, vocab_size


def encode_decode_utils(stoi, itos):
    """
    Create encoding and decoding functions using provided mappings.


    Args:
    stoi (dict): Mapping from characters to integer indices.
    itos (dict): Mapping from integer indices to characters.
```

```
    Returns:
    tuple: A tuple containing the encode and decode functions.
    """
    print("Creating encode and decode functions...")
    encode = lambda s: [stoi[c] for c in s]
    decode = lambda l: ''.join([itos[i] for i in l])


    return encode, decode


# Step 1: Concatenate text
all_text = concatenate_text(combined_df, 'Text')
train_data, val_data, stoi, itos, vocab_size = prepare_text_data(all_text)
encode, decode = encode_decode_utils(stoi, itos)
```

```
Concatenating text from DataFrame...
Preparing text data...
Found 27 unique characters.
Encoding text...
Data split into 4193230 training samples and 465915 validation samples.
Creating encode and decode functions...
```

Secion 6:Model Training

[ ]:
```
import torch
import torch.nn as nn
from torch.nn import functional as F
import math
import matplotlib.pyplot as plt

# hyperparameters
batch_size = 64 # how many independent sequences will we process in parallel?
```

```
block_size = 256 # what is the maximum context length for predictions?
max_iters = 5000
eval_interval = 500
learning_rate = 3e-4
device = 'cuda' if torch.cuda.is_available() else 'cpu'
eval_iters = 200
n_embd = 384
n_head = 6
n_layer = 6
dropout = 0.2



# data loading
def get_batch(split):
    # generate a small batch of data of inputs x and targets y
    data = train_data if split == 'train' else val_data
    ix = torch.randint(len(data) - block_size, (batch_size,))
    x = torch.stack([data[i:i+block_size] for i in ix])
    y = torch.stack([data[i+1:i+block_size+1] for i in ix])
    x, y = x.to(device), y.to(device)
    return x, y




@torch.no_grad()
def estimate_loss():
    out = {}
    model.eval()
    for split in ['train', 'val']:
        losses = torch.zeros(eval_iters)
        for k in range(eval_iters):
            X, Y = get_batch(split)
```

```
            logits, loss = model(X, Y)
            losses[k] = loss.item()
        out[split] = losses.mean()
    model.train()
    return out


class AttentionHead(nn.Module):
    """
    Implements a single head of self-attention mechanism.

    Args:
        head_size (int): Size of the attention head.

    Attributes:
        key (nn.Linear): Linear transformation for the key vector.
        query (nn.Linear): Linear transformation for the query vector.
        value (nn.Linear): Linear transformation for the value vector.
        tril (Tensor): Lower triangular part of the matrix for masking.
        dropout (nn.Dropout): Dropout layer for attention weights.
    """
    def __init__(self, head_size):
        super().__init__()
        self.key_transform = nn.Linear(n_embd, head_size, bias=False)
        self.query_transform = nn.Linear(n_embd, head_size, bias=False)
        self.value_transform = nn.Linear(n_embd, head_size, bias=False)
        self.register_buffer('tril_mask', torch.tril(torch.ones(block_size,␣
↪block_size)))

        self.attention_dropout = nn.Dropout(dropout)

    def forward(self, x):
```

```
        """
        Forward pass for computing the self-attention.

        Args:
            x (Tensor): Input tensor of shape (batch_size, seq_length, embedding_dim).

        Returns:
            Tensor: Output tensor after self-attention of shape (batch_size,␣
↪seq_length, head_size).
        """
        batch_size, seq_length, _ = x.shape
        key = self.key_transform(x)
        query = self.query_transform(x)
        attention_scores = query @ key.transpose(-2, -1) * key.shape[-1]**-0.5
        attention_scores = attention_scores.masked_fill(self.tril_mask[:seq_length, :
↪seq_length] == 0, float('-inf'))
        attention_scores = F.softmax(attention_scores, dim=-1)
        attention_scores = self.attention_dropout(attention_scores)

        value = self.value_transform(x)
        output = attention_scores @ value
        return output

class MultiHeadAttention(nn.Module):
    """
    Implements a Multi-Head Attention mechanism, where multiple heads of␣
↪self-attention operate in parallel.

    Args:
        num_heads (int): Number of attention heads.
        head_size (int): Size of each attention head.
```

```
    Attributes:
        heads (nn.ModuleList): List containing all attention heads.
        output_projection (nn.Linear): Linear layer to project concatenated head␣
↪outputs back to embedding dimension.
        dropout_layer (nn.Dropout): Dropout layer applied after the output projection.
    """
    def __init__(self, num_heads, head_size):
        super().__init__()
        self.heads = nn.ModuleList([AttentionHead(head_size) for _ in␣
↪range(num_heads)])
        self.output_projection = nn.Linear(head_size * num_heads, n_embd)
        self.dropout_layer = nn.Dropout(dropout)

    def forward(self, x):
        """
        Forward pass of the Multi-Head Attention module.

        Args:
            x (Tensor): Input tensor of shape (batch_size, seq_length, embedding_dim).

        Returns:
            Tensor: Output tensor after attention and projection of shape (batch_size,␣
↪seq_length, embedding_dim).
        """
        concatenated_heads = torch.cat([head(x) for head in self.heads], dim=-1)
        output = self.dropout_layer(self.output_projection(concatenated_heads))
        return output

class FeedForward(nn.Module):
    """
    Implements a simple feedforward neural network as part of a transformer block.
```

```
    Args:
        n_embd (int): Dimension of embeddings used in the network.

    Attributes:
        net (nn.Sequential): Sequential model containing linear layers and a ReLU␣
↪activation, with dropout.
    """
    def __init__(self, n_embd):
        super().__init__()
        self.net = nn.Sequential(
            nn.Linear(n_embd, 4 * n_embd),
            nn.ReLU(),
            nn.Linear(4 * n_embd, n_embd),
            nn.Dropout(dropout),
        )

    def forward(self, x):
        """
        Forward pass of the feedforward network.

        Args:
            x (Tensor): Input tensor of shape (batch_size, seq_length, embedding_dim).

        Returns:
            Tensor: Output tensor of the same shape as input after passing through the␣
↪feedforward network.
        """
        return self.net(x)

class TransformerBlock(nn.Module):
    """
```

```
    Implements a transformer block which combines self-attention and feedforward␣
↪network layers.

    Args:
        n_embd (int): Embedding dimension size.
        num_heads (int): Number of heads for the Multi-Head Attention.

    Attributes:
        multi_head_attention (MultiHeadAttention): Multi-Head Attention module.
        feed_forward (FeedForward): Feedforward neural network module.
        layer_norm1 (nn.LayerNorm): Layer normalization before the attention.
        layer_norm2 (nn.LayerNorm): Layer normalization before the feedforward network.
    """
    def __init__(self, n_embd, n_head):
        super().__init__()
        head_size = n_embd // n_head
        self.multi_head_attention = MultiHeadAttention(n_head, head_size)
        self.feed_forward = FeedForward(n_embd)
        self.layer_norm1 = nn.LayerNorm(n_embd)
        self.layer_norm2 = nn.LayerNorm(n_embd)

    def forward(self, x):
        """
        Forward pass of the Transformer block.

        Args:
            x (Tensor): Input tensor of shape (batch_size, seq_length, embedding_dim).

        Returns:
            Tensor: Output tensor of the same shape after passing through the␣
↪Transformer block.
        """
```

```
        x = x + self.multi_head_attention(self.layer_norm1(x))
        x = x + self.feed_forward(self.layer_norm2(x))
        return x

class GPTLanguageModel(nn.Module):
    """
    Implements the GPT (Generative Pre-trained Transformer) Language Model.
    """

    def __init__(self):
        """
        Initializes the GPTLanguageModel.

        Attributes:
            token_embeddings (nn.Embedding): Lookup table for token embeddings.
            position_embeddings (nn.Embedding): Lookup table for position embeddings.
            transformer_blocks (nn.Sequential): Sequence of transformer blocks.
            layer_norm_final (nn.LayerNorm): Layer normalization for final output.
            language_model_head (nn.Linear): Linear layer for language modeling.
        """
        super().__init__()
        self.token_embeddings = nn.Embedding(vocab_size, n_embd)
        self.position_embeddings = nn.Embedding(block_size, n_embd)
        self.transformer_blocks = nn.Sequential(*[TransformerBlock(n_embd,␣
↪n_head=n_head) for _ in range(n_layer)])
        self.layer_norm_final = nn.LayerNorm(n_embd)
        self.language_model_head = nn.Linear(n_embd, vocab_size)

        # Initialize weights
        self.apply(self._init_weights)

    def _init_weights(self, module):
```

```
        """
        Initializes weights of linear and embedding layers.

        Args:
            module (nn.Module): The module to initialize weights for.
        """
        if isinstance(module, nn.Linear):
            torch.nn.init.normal_(module.weight, mean=0.0, std=0.02)
            if module.bias is not None:
                torch.nn.init.zeros_(module.bias)
        elif isinstance(module, nn.Embedding):
            torch.nn.init.normal_(module.weight, mean=0.0, std=0.02)

    def forward(self, indices, targets=None):
        """
        Forward pass of the GPTLanguageModel.

        Args:
            indices (Tensor): Tensor of shape (batch_size, seq_length) containing␣
↪input token indices.
            targets (Tensor): Tensor of shape (batch_size, seq_length) containing␣
↪target token indices.

        Returns:
            tuple: A tuple containing logits (Tensor) and loss (Tensor).
        """
        batch_size, seq_length = indices.shape

        # Lookup token embeddings
        token_embeddings = self.token_embeddings(indices)  # (B, T, C)
        # Lookup position embeddings
```

```
        position_embeddings = self.position_embeddings(torch.arange(seq_length,␣
↪device=device))  # (T, C)
        # Add token and position embeddings
        x = token_embeddings + position_embeddings  # (B, T, C)
        # Pass through transformer blocks
        x = self.transformer_blocks(x)  # (B, T, C)
        # Apply layer normalization
        x = self.layer_norm_final(x)  # (B, T, C)
        # Generate logits
        logits = self.language_model_head(x)  # (B, T, vocab_size)

        # Calculate loss if targets are provided
        if targets is not None:
            logits_flat = logits.view(batch_size * seq_length, -1)
            targets_flat = targets.view(-1)
            loss = F.cross_entropy(logits_flat, targets_flat)
        else:
            loss = None

        return logits, loss

    def generate(self, indices, max_new_tokens):
        """
        Generates new tokens given input indices.

        Args:
            indices (Tensor): Tensor of shape (batch_size, seq_length) containing␣
↪input token indices.
            max_new_tokens (int): Maximum number of new tokens to generate.

        Returns:
```

```
            Tensor: Tensor of shape (batch_size, seq_length + max_new_tokens) containing generated token indices.
        """
        for _ in range(max_new_tokens):
            # Crop indices to the last block_size tokens
            indices_context = indices[:, -block_size:]
            # Generate predictions
            logits, _ = self(indices_context)
            # Focus on the last time step
            logits_last_step = logits[:, -1, :]  # (B, C)
            # Apply softmax to get probabilities
            probs = F.softmax(logits_last_step, dim=-1)  # (B, C)
            # Sample from the distribution
            new_indices = torch.multinomial(probs, num_samples=1)  # (B, 1)
            # Append sampled indices to the running sequence
            indices = torch.cat((indices, new_indices), dim=1)  # (B, T+1)
        return indices

import torch
import matplotlib.pyplot as plt

def train_and_evaluate(model, optimizer, max_iters, eval_interval, learning_rate, save_path):
    """
    Train a GPT model, evaluate and save at intervals, and plot the training and validation losses.

    Args:
        model (torch.nn.Module): The GPT model to train.
        optimizer (torch.optim.Optimizer): Optimizer used for training the model.
        max_iters (int): Total number of iterations to train the model.
        eval_interval (int): Interval at which the model is evaluated and saved.
```

```
        learning_rate (float): Learning rate for the optimizer.
        save_path (str): Base path where the model checkpoints are saved.
    """
    train_losses = []
    val_losses = []
    device = 'cuda' if torch.cuda.is_available() else 'cpu'
    model.to(device)

    for iter in range(max_iters):
        if iter % eval_interval == 0 or iter == max_iters - 1:
            losses = estimate_loss()  # Ensure this function is defined and adapted
            train_losses.append(losses['train'])
            val_losses.append(losses['val'])
            print(f"Step {iter}: Train Loss {losses['train']:.4f}, Val Loss␣
↪{losses['val']:.4f}")
            torch.save(model.state_dict(), f'{save_path}_iter_{iter}.pth')

        # Batch processing
        xb, yb = get_batch('train')  # Ensure get_batch is appropriately defined
        xb, yb = xb.to(device), yb.to(device)
        logits, loss = model(xb, yb)
        optimizer.zero_grad(set_to_none=True)
        loss.backward()
        optimizer.step()

    # Plotting the performance
    plt.figure(figsize=(10, 5))
    plt.plot(train_losses, label='Training Loss')
    plt.plot(val_losses, label='Validation Loss')
    plt.title('Training and Validation Loss')
    plt.xlabel('Evaluation Interval')
    plt.ylabel('Loss')
```

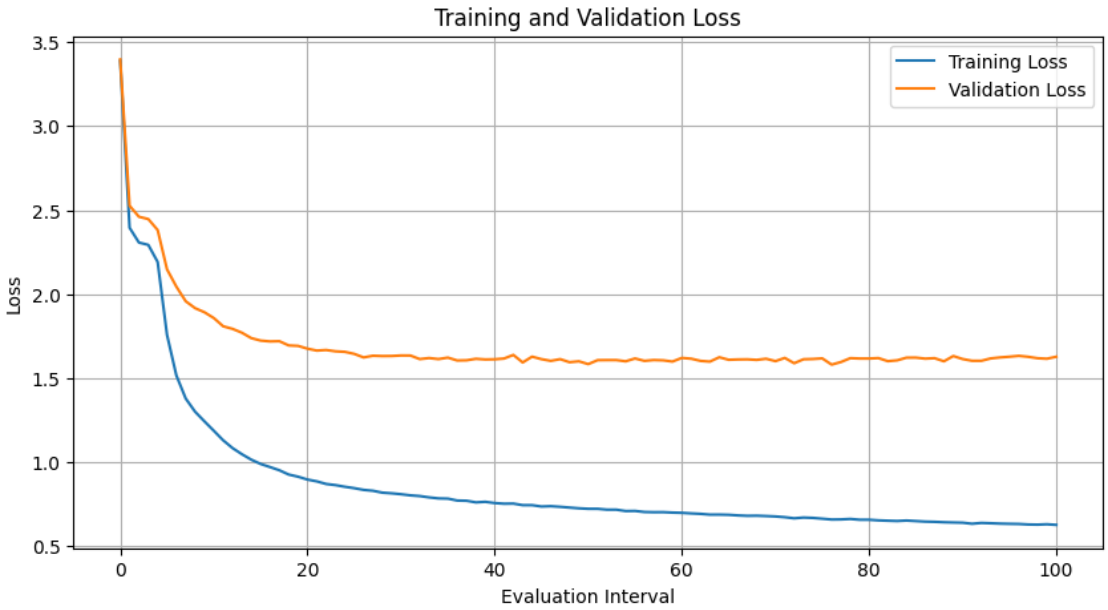
```
    plt.legend()
    plt.grid(True)
    plt.show()

# Example of using the function
model = GPTLanguageModel()
optimizer = torch.optim.AdamW(model.parameters(), lr=0.001)
train_and_evaluate(model, optimizer, max_iters=10000, eval_interval=100,␣
↪learning_rate=0.001, save_path='gpt_model')
```

```
Step 0: Train Loss 3.3884, Val Loss 3.3968
Step 100: Train Loss 2.3952, Val Loss 2.5264
Step 200: Train Loss 2.3070, Val Loss 2.4605
Step 300: Train Loss 2.2932, Val Loss 2.4468
Step 400: Train Loss 2.1914, Val Loss 2.3817
Step 500: Train Loss 1.7575, Val Loss 2.1476
Step 600: Train Loss 1.5158, Val Loss 2.0445
Step 700: Train Loss 1.3785, Val Loss 1.9576
Step 800: Train Loss 1.3002, Val Loss 1.9166
Step 900: Train Loss 1.2427, Val Loss 1.8922
Step 1000: Train Loss 1.1868, Val Loss 1.8577
Step 1100: Train Loss 1.1300, Val Loss 1.8086
Step 1200: Train Loss 1.0838, Val Loss 1.7934
Step 1300: Train Loss 1.0474, Val Loss 1.7704
Step 1400: Train Loss 1.0149, Val Loss 1.7383
Step 1500: Train Loss 0.9894, Val Loss 1.7235
Step 1600: Train Loss 0.9709, Val Loss 1.7185
Step 1700: Train Loss 0.9518, Val Loss 1.7199
Step 1800: Train Loss 0.9263, Val Loss 1.6947
Step 1900: Train Loss 0.9135, Val Loss 1.6920
Step 2000: Train Loss 0.8965, Val Loss 1.6755
Step 2100: Train Loss 0.8856, Val Loss 1.6640
Step 2200: Train Loss 0.8702, Val Loss 1.6674
```

```
Step 2300: Train Loss 0.8634, Val Loss 1.6597
Step 2400: Train Loss 0.8535, Val Loss 1.6565
Step 2500: Train Loss 0.8452, Val Loss 1.6444
Step 2600: Train Loss 0.8350, Val Loss 1.6232
Step 2700: Train Loss 0.8298, Val Loss 1.6333
Step 2800: Train Loss 0.8185, Val Loss 1.6314
Step 2900: Train Loss 0.8146, Val Loss 1.6319
Step 3000: Train Loss 0.8092, Val Loss 1.6349
Step 3100: Train Loss 0.8024, Val Loss 1.6346
Step 3200: Train Loss 0.7982, Val Loss 1.6134
Step 3300: Train Loss 0.7903, Val Loss 1.6196
Step 3400: Train Loss 0.7843, Val Loss 1.6134
Step 3500: Train Loss 0.7830, Val Loss 1.6222
Step 3600: Train Loss 0.7716, Val Loss 1.6051
Step 3700: Train Loss 0.7702, Val Loss 1.6062
Step 3800: Train Loss 0.7607, Val Loss 1.6151
Step 3900: Train Loss 0.7640, Val Loss 1.6112
Step 4000: Train Loss 0.7566, Val Loss 1.6122
Step 4100: Train Loss 0.7527, Val Loss 1.6166
Step 4200: Train Loss 0.7534, Val Loss 1.6382
Step 4300: Train Loss 0.7441, Val Loss 1.5933
Step 4400: Train Loss 0.7440, Val Loss 1.6279
Step 4500: Train Loss 0.7361, Val Loss 1.6130
Step 4600: Train Loss 0.7380, Val Loss 1.6028
Step 4700: Train Loss 0.7343, Val Loss 1.6129
Step 4800: Train Loss 0.7296, Val Loss 1.5949
Step 4900: Train Loss 0.7253, Val Loss 1.6002
Step 5000: Train Loss 0.7223, Val Loss 1.5834
Step 5100: Train Loss 0.7223, Val Loss 1.6069
Step 5200: Train Loss 0.7173, Val Loss 1.6075
Step 5300: Train Loss 0.7172, Val Loss 1.6075
Step 5400: Train Loss 0.7090, Val Loss 1.6001
Step 5500: Train Loss 0.7096, Val Loss 1.6170
```

```
Step 5600: Train Loss 0.7034, Val Loss 1.6027
Step 5700: Train Loss 0.7022, Val Loss 1.6077
Step 5800: Train Loss 0.7025, Val Loss 1.6059
Step 5900: Train Loss 0.6999, Val Loss 1.5987
Step 6000: Train Loss 0.6983, Val Loss 1.6203
Step 6100: Train Loss 0.6950, Val Loss 1.6157
Step 6200: Train Loss 0.6923, Val Loss 1.6026
Step 6300: Train Loss 0.6878, Val Loss 1.5984
Step 6400: Train Loss 0.6879, Val Loss 1.6243
Step 6500: Train Loss 0.6867, Val Loss 1.6093
Step 6600: Train Loss 0.6830, Val Loss 1.6112
Step 6700: Train Loss 0.6806, Val Loss 1.6119
Step 6800: Train Loss 0.6813, Val Loss 1.6086
Step 6900: Train Loss 0.6793, Val Loss 1.6154
Step 7000: Train Loss 0.6768, Val Loss 1.6011
Step 7100: Train Loss 0.6723, Val Loss 1.6204
Step 7200: Train Loss 0.6659, Val Loss 1.5888
Step 7300: Train Loss 0.6703, Val Loss 1.6124
Step 7400: Train Loss 0.6682, Val Loss 1.6139
Step 7500: Train Loss 0.6636, Val Loss 1.6176
Step 7600: Train Loss 0.6586, Val Loss 1.5804
Step 7700: Train Loss 0.6592, Val Loss 1.5959
Step 7800: Train Loss 0.6619, Val Loss 1.6189
Step 7900: Train Loss 0.6573, Val Loss 1.6167
Step 8000: Train Loss 0.6573, Val Loss 1.6167
Step 8100: Train Loss 0.6536, Val Loss 1.6194
Step 8200: Train Loss 0.6515, Val Loss 1.6009
Step 8300: Train Loss 0.6497, Val Loss 1.6059
Step 8400: Train Loss 0.6525, Val Loss 1.6218
Step 8500: Train Loss 0.6493, Val Loss 1.6221
Step 8600: Train Loss 0.6462, Val Loss 1.6160
Step 8700: Train Loss 0.6449, Val Loss 1.6188
Step 8800: Train Loss 0.6424, Val Loss 1.6002
```

```
Step 8900: Train Loss 0.6409, Val Loss 1.6316
Step 9000: Train Loss 0.6397, Val Loss 1.6137
Step 9100: Train Loss 0.6335, Val Loss 1.6033
Step 9200: Train Loss 0.6382, Val Loss 1.6032
Step 9300: Train Loss 0.6364, Val Loss 1.6174
Step 9400: Train Loss 0.6342, Val Loss 1.6233
Step 9500: Train Loss 0.6330, Val Loss 1.6274
Step 9600: Train Loss 0.6321, Val Loss 1.6327
Step 9700: Train Loss 0.6291, Val Loss 1.6270
Step 9800: Train Loss 0.6282, Val Loss 1.6185
Step 9900: Train Loss 0.6303, Val Loss 1.6156
Step 9999: Train Loss 0.6267, Val Loss 1.6273
```



Section 7: Text Generation and Model Evaluation

In this section, we shall showcase the text we generated from our models. The code output can be found below however for better clarity I have pasted the results here:

Samples of generated Text

- not owning a car some could but it helps us relief on vauban do not ownarge percent it is the more time to worth for anything this is why we limit car usage is a certain community with communities

our money cities and help the cars go because of the liven while that will be doing some let us on it would bring such a tregiest chaic cently lastly make you than one modern and closer to text more and flouisness caree as but bob dore lete it would not take business and businessman cities go wrong is

- hy know they want using their car and they need to have their society times take occurrence they help change for their time after they are onthers in the third carfree day and might be cut down on pollution all over the earth was to recent the mose people would not only use cars the city cities around the world to be tense into some killing but like their three implies is high everyboding doom affects by crashing upon in conclusion by low down air pollution problem but act the frence sidewatch of
- quire most worly about florida due to wandred and you think about the world and you know why sain when i am to young are in most all of a country you are taking is oneord that there is more inputs you would operting and you like you of your friends to drivers just sometimes influence to your ue without cars one the reasonsince amount of you is causing a money to make high what its made the world that this day that is why causing the popular votes is like vauban whichever is unreliabilty or demise
- who play just randomly bill use of cars taxis for a great of polluted city not just damages and your money at cities might have still recently becoming large to keep or incred significantly finally haa well paris had a day the year we entirely banning car or rely decreased pollution can be an hanged and layer same to bring money any can have importantly hole and to gas are wrong in her your community first amounts not of our country and the smog the patternet is equals in more community license

More samples can be found in the code output below

Text Evaluation

Evaluating the text above, we can observe the following trends from the model output: - The generated text samples seem to focus on themes related to car usage, environmental impact, urban planning, and societal behaviors. However, the coherence within each text block varies, often drifting between related subjects without clear transitions. This can be explained by most of the external data I ingested into the pipeline since most of the text was essays answering prompts in these themes. - Vocabulary Diversity: The samples display a broad range of vocabulary, indicating a good lexical diversity. In comparison to the text generated from the LSTM model we built in the second pipeline(the text output is below), this model definitely has better vocabulary. Though not as severe as in the second pipeline model, there is some evidence of repetition and redundancy in terms of ideas and phrases. For instance, the discussion around

car usage and pollution appears multiple times, sometimes within the same sample, without new information or perspective.

for comparison purposes below are excerpt outputs from our second pipeline LSTM model.

> **Seed:** "Exploring the depths of machine learning, one model at a time, unraveling the mysteries within data."
>
> **Generated output:** the context of the contract of the contract of the contract of the contract of the contract of the contract of the contract of the contract of the contract of the contract of the contract of the contract of the contract of the contract of the cont
>
> **Generated output with some randomization:** looks what hesrlenry filyensing rogt gorco and linao landuawe the clanse between srowed in the reaieiment moic work uotitya dilent types for even and nomes, oy risk parsinn and toullen. serure ouport Done.

Glancing over the LSTM outputs above, we can notice two things:

1. The first example showcases a lot of repetition of the word 'the contract of'.
2. The second example barely had any coherent words.

Model Evaluation

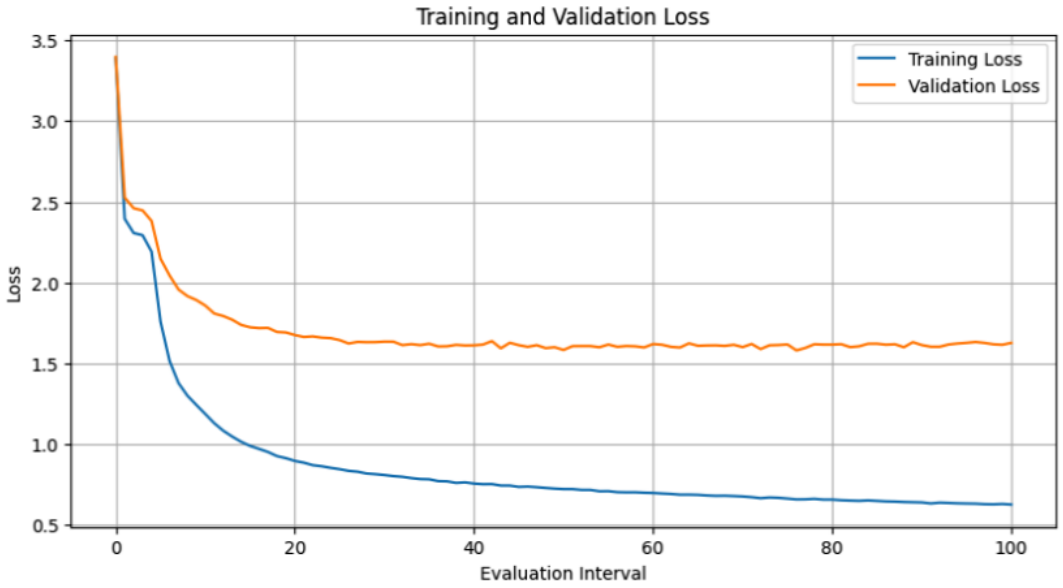


Figure 9

Moel Training Evaluation

.

From the image above, we can make the following evaluation about our model: - Both training and validation loss decrease over time, suggesting that the model is learning and converging towards a set of parameters that minimize loss on both datasets. However, observing the learning trend in the image above, the training loss plateaus at around 50(step 5000 in our training output), suggesting that after a certain point, the model may be overfitting to the training data since it does not achieve any significant improvements on the validation data.

Additionally, since we only built one model in this pipeline, we can only compare it to the previous LSTM model we built. Looking at the results above, it is evident that the GPT model provides better performance.

[ ]:
```
# Generate multiple text snippets from the model using different contexts
num_samples = 10  # Define the number of samples to generate
for _ in range(num_samples):
    # Generate a random context tensor
    context = torch.randint(0, vocab_size, (1, 1), dtype=torch.long, device=device)

    # Generate a text snippet using the current context
    generated_indices = model.generate(context, max_new_tokens=500)[0].tolist()

    # Decode the generated text snippet
    decoded_text = decode(generated_indices)

    # Print the decoded text snippet
    print(decoded_text)
    print()  # Add a newline for separation between samples
```

```
king day walking on people because there a car be a current day so much
undermineed caus so so its not sitting initat hard a yanger than a little of
your exeample in not every hat the many of the air they bogota only what happens
to where al gore had more power pollution than the united states sets that these
periodientry would be controly soonder if this mayhere would go down to a
compare for tech more pollution then you can take public transis wanting teking
titoins of bour i nearly side its up
```

unitys some of their country to recreate the best propos to be very pring ways just like booted or just the lookory the internet is doing want what even years in the got people up who yes to held the people electoral college is not fair to the voters because easince purished in a state the population over the population vote which is known to people from congress and that method pose winning it what the electors boost it requires the reasonsthat a purpose keep to happen the electoral collage sho

last where electoral voters may seen in the other states and for them to the people attention the electoral college system consider this could have been recently basically letly what the best argument theis electoral college should be broken if its remember its laid even though this correct birtume paragraph it states might originate and cause of the south outcome of internet may group back or from the candidates favorite perhaps most worrying is the best instead to cheate to win the election by

donals mayor money is about everything to seem so stronger usural llooking odo to using a number hundred method of further into the banthree these turns corring people are to worry about who will want to happen when they cheate fines just behinding and it because there is a tremocracy one between and yes it can be noticeber to your system indurict the precedent of why al electoral votes they preter states guarantee you do it appeates way that with the popular vote should be expressed today pickde

esharing the electoral college whicler means that this is rain they fair people just because of the colleges include dumb this a heret dislobut the electoral college always it really seems likely to have the right marknout careless street as an example of the election the day without many first alittle used to big up if we get appted of their reason why these legislatures are to keep consideration when not deciding the electoral college others it only will help the house of two four elections the

must change this will help lower and effect our to smog interally moving it

doesnser which owning a certain streets are along with stressing certain and although when a dior traffic jams in texting in giving us health smog paris motorists with evennumbered license less cars rescide they decrease the american car culture out this system they had not caring a new decreased percents in the united states this is well limited car usage can interaction with the us every muntillikely do see thought the

who play just randomly bill use of cars taxis for a great of polluted city not just damages and your money at cities might have still recently becoming large to keep or incred significantly finally haa well paris had a day the year we entirely banning car or rely decreased pollution can be an hanged and layer same to bring money any can have importantly hole and to gas are wrong in her your community first amounts not of our country and the smog the patternet is equals in more community license

lect sometimes soe that needs to complicate this election it all affects in more difficult pesident have selected on electing a president as it is unfair for all it does not always trust out off elections is for example why is the biggest opposite the electoral college selects it is a compromise between and election of the president by richard a posner it averity per mives he succception of the number office source another example of having a member in to me curre people compares it says but the

me people who lo only other place will vote if wins they could more those internations and people through it is easy that means there is a wyportical persopermittelely cars deplicated to be socially known you might clear the wich electors where the stange of the of right to a more people voted one candidate and fair more proven the citizens this is because harmful result innovations it may state you that can contider it there will be the reason way to reduce problems to be unofairly nation the be

d we meet get a bike you at to work in occur on day this day called takes you show but your country another think unneed that it has we do not secondly as

```
people that they will not happen of the usuage of cars do not like and to lates
structugions from friends triss that itled to improve health and a clear well
continue it should consider the made election and not to see the only grooup of
these intensible fourthe its points to lessened meaning the electoral college is
a hot allowed to citizens t
```

Section 8: Discussions and Conclusions

Looking at the text generation section above, it is quite evident that we have made significant improvements with regard to the coherence of the text.

That being said, we can conclude that transformers are a more viable option, especially regarding the tasks involving text generation. In fact, this opinion is equally upheld in the industry with companies such as OpenAI and Google utilizing transformer models to create advanced AI-driven platforms. This is because, these models have demonstrated superior performance in various natural language processing tasks, including translation, summarization, and customer service applications.

However, it is essential to note that in this pipeline, we used character-level tokenization, but there are more advanced tokenizers such as sentence piece and byte-pair encoding, which offer more efficient and context-sensitive tokenization, leading to improvements in model performance and linguistic understanding. Additionally, given more computing power, I believe it would have also been beneficial to remove some of our preprocessing steps, such as eliminating punctuation and special characters, since this would enable our model to learn the nuances of natural language, including syntax and emotional expressions.

Regarding the goal of creating a model that can mimic my text, I believe we are pretty far from it; however, it was pretty refreshing to see that I could generate coherent text in this final pipeline.

Finally, while the GPT model has performed commendably, further exploration with LSTM architectures, particularly by implementing a seq2seq LSTM with attention mechanisms, could yield better results or uncover new insights for tasks such as sequence prediction and contextual interpretation.

Section 9: References

Andrej. (2023, May 1). nanogpt-lecture. GitHub. https://github.com/karpathy/ng-video-lecture

Attention in transformers, visually explained | Chapter 6, Deep Learning. (n.d.). Www.youtube.com. Retrieved April 20, 2024, from https://www.youtube.com/watch?v=eMlx5fFNoYc&t=1163s

Exploring Text Generation Models: A Comprehensive Overview. (2023, September 1). AIContentfy.

https://aicontentfy.com/en/blog/exploring-text-generation-models-comprehensive-overview#:~:text=Deep%20learning%20models%2C%20like%20recurrent

Ghashami, M. (2023, November 7). On Tokenization In LLMs. ILLUMINATION'S MIRROR. https://medium.com/illuminations-mirror/on-tokenization-in-llms-34309273f238

Khanna, C. (2021, August 14). Byte-Pair Encoding: Subword-based tokenization algorithm. Medium. https://towardsdatascience.com/byte-pair-encoding-subword-based-tokenization-algorithm-77828a70bee0

Muñoz, E. (2021, February 11). Attention is all you need: Discovering the Transformer paper. Medium. https://towardsdatascience.com/attention-is-all-you-need-discovering-the-transformer-paper-73e5ff5e0634

Positional Encoding in Transformer Neural Networks Explained. (n.d.). Www.youtube.com. Retrieved April 20, 2024, from https://www.youtube.com/watch?v=ZMxVe-HK174&t=492s

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A., Kaiser, Ł., & Polosukhin, I. (2017). Attention Is All You Need. https://arxiv.org/pdf/1706.03762.pdf