CSCI495 - Report

Kyle Remmenga

May 5th 2025

Contents

1 Introduction

In order to gain a better understanding of modern machine learning, we ventured to create my own generative pretrained transformer (GPT). The goal of this was to understand the mathematics behind a transformer as well as how this is implemented practically in code. In addition, our goal was to determine how well a language model could be created using minimal resources, especially relative to modern large language model manufacturers. Through this, we had hoped to learn about the necessary parts of a transformer independently in order to create a deeper understanding of transformers as a whole. In order to begin creating a working GPT, we needed to define a clear goal for the model so that we could begin to gather training data and create the architecture. We decided that a simple GPT that could generate short children's stories would be a reasonable choice considering that children's stories generally have simple characters with clear actions. From this, we were able to begin researching different design choices for the model.

2 Vocabulary and Tokenization

In order to make a language model, we had to clearly define our vocabulary that we would use throughout this process. The challenging part of this was to create a small enough vocabulary so that we were able to develop a model with our resources while also allowing the model to be large enough to communicate and remember ideas holistically. In addition, modern tokenizers tokenize more than individual words and often tokenize prefixes and suffixes independently. However, this leads to creating a much larger vocabulary for the purpose of generating well-organized English sentences. Fortunately, we don't need to generate grammatically correct sentences in order to communicate simple children's stories. Therefore, we decided to define our own simplified English language that we referred to as 'pidgin' English. In pidgin, we are able to communicate basic actions, emotions, and thoughts of characters within a small defined universe.

2.1 Pidgin

Communicating in pidgin is quite simple. We use simple nouns and verbs to communicate ideas in their most basic form. In order to eliminate complexity for the tokenizer we use only lower case characters and try to reduce punctuation. For example, in pidgin, we might say "they like cat" meaning "they like cats". The most important part of using this pidgin language, however, was conjugating verbs. In English, it's intuitive for English speakers to understand the tense of words. However, in order to make it simpler for the model, we defined words in pidgin such as "ing", "ed" and "will" that communicate timing within a verb. We use "ing" to represent a verb occurring in the present, "ed" communicates a verb in the past tense, and "will" to communicate a verb happening in the future. An example of this is saying "ball ing roll" to mean

"the ball is rolling" or "bird ed fly up" to mean "the bird flew up". This avoids having to tokenize one English word into several parts and allows us to declare that each word separated by whitespace is a unique token and therefore would have a unique embedding. In addition, this avoids having to add several words into our vocabulary that represent the same word occurring in different tenses, such as "fly" and "flew".

Another benefit of using pidgin was allowing us to simplify punctuation throughout our generated content. Since we are communicating simple ideas, we don't have any reason to use uncommon characters such as semicolons or parentheses. However, we still needed a way to communicate questions in dialogue, so we added a word into our vocabulary: "ha". We used "ha" at the start of a question to indicate that a question was being asked. Using "ha" in our generated content looks like this: "ha you ing leave?" meaning "are you leaving?". This eliminates the ambiguity of whether or not a question was asked without needing all of the English words that indicate a question such as "who" or "what".

2.2 Creating a Vocabulary

Initially, when we attempted to create a defined vocabulary for our model, we tried to make it as simple as possible. We wanted to create a definite word list then gather data that fit our specifications exactly in order to train our model. However, it became clear quickly that we were being too specific with our data and would not be able to gather enough data in a reasonable amount of time in order to train our model. Therefore, we decided to change how we gathered data and instead create our vocabulary based on the data we received. Since we knew we were creating a model to generate simple children's stories, we defined a universe in which these stories occur. The universe included a school with a playground, a row of houses, a grocery store, and a forest. This was meant to define the world the stories existed in in order to still keep the stories grounded in simplicity while providing a space where many different things could happen so the stories were still somewhat unique. From once we had gathered a sufficient amount of data, we parsed our stories to look for the most common words and included words that were used the most into our vocabulary. This way, we were able to generate a large sum of data while still controlling exactly what kind of words went into our vocabulary.

3 Data Acquisition

Likely, the hardest challenge for building this model was acquiring enough data so it could be adequately trained. Since we were creating a model that produced simple stories, we needed a way to obtain a substantial amount of stories similar to the ones we wished to generate. In addition, the stories needed to be written in our pidgin language which was entirely manufactured for this project. These

two issues both posed challenges that needed to be addressed separately.

3.1 Amassing Data

In order to obtain data, especially data that was specific to our needs, we decided to generate it using a third-party large language model. After reviewing options, we decided to use DeepSeek because we were able to use its relatively cheap API to pull data passively for long periods. DeepSeek also charges a discounted price during less active hours of the day, which in mountain time was from 10:00 am to 6:30 pm. This allowed us to generate all of the necessary training data for our model for roughly $2.00 USD. Each story generated took roughly 10-15 seconds to pull and save. Therefore, throughout the duration of the project, using a single API key, it took roughly 19 hours total.

In order to generate the correct data to train the model, we had to be specific about our prompt that we would provide to DeepSeek. Throughout the generation process, we experimented with several different prompts in order to find one that would give us the most successful generations. To do this, we created a file that contained all of the grammar rules that our pidgin language used, as well as some examples of sentences that used pidgin. We also included our first rudimentary vocabulary and asked DeepSeek to exclusively use words within the vocabulary. This was intentional because we knew DeepSeek would stray from the vocabulary at some points, but to keep the vocabulary small it was important that it wouldn't drift too far from our original plan. We would also pass our universe file containing the setting that the stories should take place in. We again wanted to be sure that DeepSeek tried to use a small vocabulary, so we asked it to parse the story and ensure that the story only used words from the vocabulary. Lastly, we asked DeepSeek to only generate a story and not to generate anything else from the prompt. This allowed us to eliminate the extra words that DeepSeek would respond with that weren't the story. For most of the training data generated, we used the following prompt which produced an output such as the example below.

```
"Using this following data as a reference, can you write a short
    children's story using this way of writing: {grammar_context}
    and exclusively this vocabulary: {vocabulary} \
The story takes places in this world: {universe_context}. Before
    you return the story, parse it and check that only words from
    the vocabulary are in your story. \
Do not include anything else in your response besides the story."
```

Listing 1: DeepSeek Generation Prompt

```
        "**The Little Bird and the Big Tree**\n\nOne day, small
    bird fly ing sky. Bird see big tree. Tree stand ing east. Bird
    think, \"I will move ing at tree.\"\n\nBird fly ing right. Bird
     see no right. Bird see left. Bird no like left. Bird say, \"I
    no move left. I move right.\"\n\nTree say, \"Ha bird ing come
    at me?\" Bird say, \"Yes, I ing come at you.\"\n\nBird fly ing
    at tree. Tree happy. Tree say, \"You are good bird.\"\n\nBird
    see red sun ing sky. Sun hot. Bird say, \"I no like hot sun. I
```

```
    like cold moon.\"\n\nNight come ing. Moon come ing. Moon cold.
    Bird happy. Bird say, \"I love cold moon. I no love hot sun.\"\
    n\nBird sleep ing at tree. Tree happy. Tree say, \"Good night,
    small bird.\"\n\nThe end."
```

Listing 2: DeepSeek Generation Example

3.2 Preparing the Data

The next step was to prepare the data in an adequate way so that it could be used for training. It's clear from the example above that the generated story from DeepSeek contains many extra characters and punctuation that we want to avoid. To fix this, we parsed the data and removed punctuation marks that were not in our vocabulary. Since DeepSeek often wrote titles for the stories and surrounded them with '**' characters, we wrote a regular expression to find these sections and remove them. We then converted the entire string into lower case and created a list that contained every word in the story and checked it against our vocabulary to ensure that there were no unexpected words in the training data. Lastly, we added "<sos>" to the start and "<eos>" to the end of each story. "<sos>" is the start of sequence token and "<eos>" is the end of sequence token, and with these we can judge what the model determines to be a complete story. After all of these changes, we write all stories that pass our checks to a file where they are ready to be used as training data. The listing below is an example of a complete story that was used as training data in the model.

```
"<sos> one day, small bird fly ing sky. bird see big tree. tree
    stand ing east. bird think, i will move ing at tree. bird fly
    ing right. bird see no right. bird see left. bird no like left.
     bird say, i no move left. i move right. tree say, ha bird ing
    come at me? bird say, yes, i ing come at you. bird fly ing at
    tree. tree happy. tree say, you are good bird. bird see red sun
     ing sky. sun hot. bird say, i no like hot sun. i like cold
    moon. night come ing. moon come ing. moon cold. bird happy.
    bird say, i love cold moon. i no love hot sun. bird sleep ing
    at tree. tree happy. tree say, good night, small bird. the end.
     <eos>"
```

Listing 3: Training Data Example

4 Attention

The most defining characteristic of transformer architecture is attention. Attention is the mechanism that allows the model to learn context from previous tokens from their embedding as well as their position relative to the current token. The goal of attention is to adjust the embedding of tokens in order to allow them to represent a higher-level idea, containing more information than simply the token itself. Attention mechanisms build on previous neural network designs by adding additional layers in between perceptron layers that lead to a

significantly improved prediction when generating text and images. This section will explain how attention is performed mathematically then explore how it is employed in our model using PyTorch.

4.1 How Does Attention Work?

4.1.1 Embedding

To understand how attention works, we must build an understanding of how deep learning models use words. Within a transformer, there exists an embedding table that matches each unique token to a high-dimensional vector, called its embedding. This vector is unique to each token; however, after embedding, this vector contains no relevant context about the token besides its position within the context. In order to encode the position of the token into its embedding, each token within the sequence is assigned another vector of the same size as the token embedding based on its place within the sequence. From here, the token embedding vector and the position embedding vectors are summed to create a unique embedding for each token/position pair. If our goal is to create a model that is able to accurately predict which tokens should follow, we need to adjust this vector so that it contains information that is relevant to this token at a specific moment. For example, we need a way to distinguish the meaning of synonyms from context. If we have embedded the token 'bat' into a vector, this vector needs to be able to store information regarding our specific meaning of 'bat'. Therefore, we need a way to adjust the vector so that if we are talking about baseball, the model knows that we are likely talking about a baseball bat, instead of the animal.

Differentiating synonyms is a basic example to understand why attention might be useful, but it's easy to understand why previous context matters to a token in many other circumstances. One of these circumstances is encoding adjectives or descriptions into a noun. If we were to write a story about a small gray cat, we want our transformer to adjust the vector of cat to incorporate that the cat is small and gray. This idea continues to be relevant with more complex stories, for example allowing the transformer to encode a character's history and current circumstances into their embedding. In essence, attention changes the embeddings of tokens so that they are able to learn and react to previous tokens in the context, which leads to a significantly better performance as a prediction model.

4.1.2 Sharing Information

Let's refer to a token's embedding vector as $\vec{E_n}$ which represents the embedding vector of the token at position n in the sequence. Then, at the start of an attention head, we define a query and key matrix that we'll call W_Q, W_K. Both

W_Q and W_K are matrices of size

$$\text{embedding dimension} \times \frac{\text{embedding dimension}}{\text{heads per block}}. \tag{1}$$

From here we'll multiply W_Q and W_K with each token's embedding in the sequence to get two new vectors for each token, a query vector and a key vector referred to by $\vec{Q_n}$ and $\vec{K_n}$ which are both

$$\frac{\text{embedding dimension}}{\text{heads per block}} \tag{2}$$

dimensional vectors.

$$\begin{aligned}
\vec{E_0} \times W_Q &= \vec{Q_0} & \vec{E_0} \times W_K &= \vec{K_0} \\
\vec{E_1} \times W_Q &= \vec{Q_1} & \vec{E_1} \times W_K &= \vec{K_1} \\
\vec{E_2} \times W_Q &= \vec{Q_2} & \vec{E_2} \times W_K &= \vec{K_2} \\
&\vdots & &\vdots \\
\vec{E_n} \times W_Q &= \vec{Q_n} & \vec{E_n} \times W_K &= \vec{K_n}
\end{aligned}$$

We can think about a query vector asking previous tokens if they have any relevant context for this token, and the key vector for the previous tokens responding, informing the querying token that they do or don't, or something in between. In essence, for all token's key vectors from $\vec{K_0}$ to $\vec{K_n}$, if $\vec{K_p}$ for $0 \leq p \leq n$ closely aligns with $\vec{Q_n}$ then we know that token p has relevant information to share with token n. In order to determine whether two vectors are closely aligned, we take the dot product of $\vec{Q_n}$ with $\vec{K_p}$ for all p.

Table 1: Calculating Query and Key Dot Product

	$\vec{Q_0}$	$\vec{Q_1}$	$\vec{Q_2}$	...	$\vec{Q_n}$
$\vec{K_0}$	$\vec{Q_0} \cdot \vec{K_0}$	$\vec{Q_1} \cdot \vec{K_0}$	$\vec{Q_2} \cdot \vec{K_0}$	...	$\vec{Q_n} \cdot \vec{K_0}$
$\vec{K_1}$	$\vec{Q_0} \cdot \vec{K_1}$	$\vec{Q_1} \cdot \vec{K_1}$	$\vec{Q_2} \cdot \vec{K_1}$	...	$\vec{Q_n} \cdot \vec{K_1}$
$\vec{K_2}$	$\vec{Q_0} \cdot \vec{K_2}$	$\vec{Q_1} \cdot \vec{K_2}$	$\vec{Q_2} \cdot \vec{K_2}$	...	$\vec{Q_n} \cdot \vec{K_2}$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\ddots$	$\vdots$
$\vec{K_n}$	$\vec{Q_0} \cdot \vec{K_n}$	$\vec{Q_1} \cdot \vec{K_n}$	$\vec{Q_2} \cdot \vec{K_n}$	...	$\vec{Q_n} \cdot \vec{K_n}$

If any of the key vectors are aligned closely with the query vector, the dot product will be some large positive number. In this context, if $\vec{K_p}$ and $\vec{Q_n}$ align

closely to one another, we say that the embedding of token p attends to the embedding of token n.

Another thing that is important to note, is that when the model is being trained, we can get more value by splitting a training sequence into many smaller subsequences. For example, if we wanted to train our model to predict the next English word in a sentence and the context the model received was: "my favorite flavor of ice cream is", we can split this text into subsequences to predict what comes after "my", then "my favorite", and so on. This allows us to turn one training phrase into many. However, during this training, we don't want our embeddings to receive any information from tokens that come after it. This would train the model that it could use token information that during the application of the model would not exist. To solve this, we use a process called masking, which just means that we set the dot product values where they interact with future tokens to be $-\infty$. So at this point our table of dot products really looks more like the following.

Table 2: Dot Products After Masking

	$\vec{Q_0}$	$\vec{Q_1}$	$\vec{Q_2}$	...	$\vec{Q_n}$
$\vec{K_0}$	$\vec{Q_0} \cdot \vec{K_0}$	$\vec{Q_1} \cdot \vec{K_0}$	$\vec{Q_2} \cdot \vec{K_0}$	...	$\vec{Q_n} \cdot \vec{K_0}$
$\vec{K_1}$	$-\infty$	$\vec{Q_1} \cdot \vec{K_1}$	$\vec{Q_2} \cdot \vec{K_1}$	...	$\vec{Q_n} \cdot \vec{K_1}$
$\vec{K_2}$	$-\infty$	$-\infty$	$\vec{Q_2} \cdot \vec{K_2}$	...	$\vec{Q_n} \cdot \vec{K_2}$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\ddots$	$\vdots$
$\vec{K_n}$	$-\infty$	$-\infty$	$-\infty$	...	$\vec{Q_n} \cdot \vec{K_n}$

At this point we have a table of dot products of length $n \times n$, which shows how well the embedding of each previous token attends to the embedding of the current token. In order to keep the model from scaling too much, it's common to divide each dot product by $\sqrt{d_k}$, where d_k represents the dimension of our query/key space. Once we divide each product, we can continue by making each column a probability distribution. We want these dot products to act like weights in our model so we need to normalize them so that all of the weights in a column sum to 1. In machine learning, we use the softmax equation which converts a vector of K real numbers into a probability distribution of K outcomes. The softmax equation is

$$\sigma(\vec{z})_i = \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}} \quad for\ i = 1, 2, \ldots, K. \tag{3}$$

We can use our column of dot products as the vector input into our softmax equation to create a new column containing normalized values that have con-

verted their values into meaningful probabilities. Wherever we see $-\infty$, after softmaxing the value will be 0. Since we have a list of weights relating our current token and each previous token in the sequence, we can think about our list as representing how relevant each previous token is to our current token n. Each column of our table now looks something like the following table where a_i represents the values in each cell after softmax, with $0 \leq a \leq 1$ for all $i \leq n$ and $\sum_{i=0}^{n} a_i = 1$.

Table 3: Example Column after Softmax

	$\vec{E_n}$
$\vec{E_0}$	a_0
$\vec{E_1}$	a_1
$\vec{E_2}$	a_2
$\vdots$	$\vdots$
$\vec{E_n}$	a_n

Now that we know how relevant previous tokens are to future tokens, we need to determine what information they should pass. This is done by defining a value matrix that we'll refer to as W_V, which is the same size as the previous matrices:

$$\text{embedding dimension} \times \frac{\text{embedding dimension}}{\text{heads per block}}. \tag{4}$$

We multiply W_V with each token's embedding vectors $\vec{E_n}$ to get a new set of value vectors $\vec{V_n}$ that also share the dimension of the previous vectors:

$$\frac{\text{embedding dimension}}{\text{heads per block}}. \tag{5}$$

You can think of $\vec{V_n}$ answering the question for the token, "what information should I pass if I am relevant?" We then multiply each token's value vector by the associated softmax weight, then sum the column. This gives us a new vector we'll call $\Delta\vec{E_n}$ which indicates changes we want to make to $\vec{E_n}$.

$$\sum_{i=0}^{n} \vec{V_i} \times a_i = \Delta\vec{E_n} \tag{6}$$

It's important to remember that this process is performed for every token in the sequence. Thus, every token's embedding in the sequence has the opportunity to be changed. This process of adjusting all of the token embeddings in a sequence

Table 4: Multiply the Value Vector with the Weights

	$\vec{E_n}$
$\vec{V_0}$	$\vec{V_0} \times a_0$
$\vec{V_1}$	$\vec{V_1} \times a_1$
$\vec{V_2}$	$\vec{V_2} \times a_2$
$\vdots$	$\vdots$
$\vec{V_n}$	$\vec{V_n} \times a_n$

is referred to as a single head of attention. In the transformer, we use attention blocks that are able to process many of these attention heads in parallel, each with their own query, key, and value matrices. In our model, each attention head would produce a $\Delta\vec{E_n}$ of size

$$\frac{\text{embedding dimension}}{\text{heads per block}}. \tag{7}$$

for every token n in the sequence. After each head computes its own $\Delta\vec{E_n}$, they are concatenated with each other to form a vector that matches the length of the embedding dimension. Since our vector was formed from concatenating several different vectors, we need a meaningful way to mix the results of different attention heads. To solve this, we use one last matrix we'll call W_O, that we'll multiply this vector through. W_O needs to be of size

$$\text{embedding dimension} \times \text{embedding dimension} \tag{8}$$

so that the vector product of the matrix multiplication has dimensions matching our original embedding dimension. After multiplying our vector formed from concatenating our $\Delta\vec{E_n}$s with W_O, we can now sum these vectors to get $\vec{E_n}'$, our new embedding vector containing contextual information from other tokens.

4.2 Showing this Process in Code

To start coding the attention process using PyTorch, we start by defining our parameters that will be used in our model. The relevant parameters to attention are

```
batch_size = 32 # Pieces of text to process together
block_size = 256  # Context length
n_embd = 384 # Embedding Dimension
n_head = 6 # Attention heads per block
n_layer = 6 # Amount of sequential attention blocks
```

Listing 4: Attention Hyperparameters

From this, the important takeaways are that during training, the model takes 32 random text samples from the training data per epoch. Each training sample is 256 tokens long, so during attention, the last token in the sequence is informed by the context of 255 previous tokens. Each token's embedding vector has 384 dimensions, each attention block has 6 heads, and we stack 6 attention blocks sequentially to create the model. From this, we can begin to create the smallest part of the attention mechanism, the head.

```
class Head(nn.Module):
    """One head of self-attention"""
    def __init__(self, head_size):
        super().__init__()
        self.key = nn.Linear(n_embd, head_size, bias=False)
        self.query = nn.Linear(n_embd, head_size, bias=False)
        self.value = nn.Linear(n_embd, head_size, bias=False)
        self.register_buffer('tril', torch.tril(torch.ones(
    block_size, block_size)))
        self.dropout = nn.Dropout(dropout)
```

Listing 5: Attention Head Class Parameters

To start, we use a parameter 'head_size', which is calculated in the attention block as

```
head_size = n_embd // n_head
```

Listing 6: Attention Head Size Calculation

so we calculate the attention head size by dividing the size of our embedding vectors by the amount of heads. From this, we can see that we initialize each head a unique key, query, and value linear transformation layer. These linear transformation layers are matrices of dimensions

$$\text{embedding dimension} \times \text{head size}. \tag{9}$$

So we have created our matrices of dimensions

$$\text{embedding dimension} \times \frac{\text{embedding dimension}}{\text{heads per block}}. \tag{10}$$

that we defined as the size of our matrices in the previous explanation. Lastly, we don't want to use biases when calculating attention because they interfere with the model's ability to learn context. Next, we then define a persistent buffer in the model that forms a matrix in lower triangular form that we will use to stop the model from learning context from future tokens during training.

Within the model, we also use a technique to reduce overfitting called dropout. Dropout randomly sets a percentage of inputs within the model to zero to prevent the model from learning the training data too well. Next, within the Head class, we only define one function, which is the forward pass.

```
    def forward(self, x):
        B,T,C = x.shape
        k = self.key(x)
```

```
        q = self.query(x)
        wei = q @ k.transpose(-2,-1) * k.shape[-1]**-0.5
        # (32, 256, 64) @ (32, 64, 256) -> (32, 256, 256)
        wei = wei.masked_fill(self.tril[:T, :T] == 0, float('-inf')
    )
        wei = F.softmax(wei, dim=-1)
        wei = self.dropout(wei)
        v = self.value(x)
        out = wei @ v
        return out
```

Listing 7: Attention Head Forward Function

To start, the function takes a parameter x, which is the input texts as a tensor. X has 3 dimensions, the first one represented in the code by B is the batch size, which is the amount of texts we'll be processing at one time, which for this model was 32. The next one T is the sequence length for each piece of training text which was 256, and C is the vector embedding for each token that is referred to as the channel which was 384. So, x includes all of the input text data the model will process. We then multiply these through our key and query matrices to get key and query tensors which also have 3 dimensions. The first two dimensions are the same as the input x, but the last dimension is now length 64, the head size. In PyTorch, we can easily matrix multiply by using the '@' operator, so we then multiply the last two dimensions of our query and key tensors to create 32 256×256 matrices which are our 32 tables containing the dot products from our query and key matrices that we represent with a *wei* variable representing our weights. We also multiply the matrices by $k.shape[-1]**-0.5$, which divides our matrices by $\sqrt{d_k}$. We then apply the mask by transposing the triangle matrix, setting the values up to T in each column in each matrix to $-\infty$. Lastly, we apply our softmax to *wei* to normalize each column. We can easily apply our dropout by passing our *wei* through our dropout variable. Now we can create our value tensor represented by v and multiply it with our *wei* matrices to obtain our return.

Now we will move on to examining the attention heads as they are put together. The next class to examine is the MultiHeadAttention class.

```
class MultiHeadAttention(nn.Module):
    """Multiple heads of self-attention in parallel"""
    def __init__(self, num_heads, head_size):
        super().__init__()
        self.heads = nn.ModuleList([Head(head_size) for _ in range(
    num_heads)])
        self.proj = nn.Linear(n_embd, n_embd)
        self.dropout = nn.Dropout(dropout)

    def forward(self, x):
        out = torch.cat([h(x) for h in self.heads], dim=-1)
        out = self.dropout(self.proj(out))
        return out
```

Listing 8: MultiHeadAttention Class

We start by creating num_heads attention heads and adding them to our list of modules within the model. Then we define our projection matrix with the dimensions

$$\text{embedding dimension} \times \text{embedding dimension}. \tag{11}$$

Once again we are using dropout, so we define a dropout variable in this class. Looking at the forward function, we start by concatenating our outputs from our heads into one vector. Then we project this vector through our projection matrix, apply our dropout, and return our new embeddings.

5 Feed Forward Layer

Feed forward layers are placed between our attention calculations in order to add non-linearity to the prediction of the model. After our attention blocks output a vector matching our embedding dimension, in order to add non-linearity we multiply this vector by a matrix of size

$$\text{embedding dimension} \times 4 * \text{embedding dimension}. \tag{12}$$

At the point we have a vector that is 4 times the size of our embedding vectors. From here we apply an activation function to this vector to add non-linearity. In this case of our model we used the ReLU function defined by

$$Relu(z) = max(0, z). \tag{13}$$

Then we define another matrix of size

$$4 * \text{embedding dimension} \times \text{embedding dimension} \tag{14}$$

that we can multiply our vector through in order to recreate our vector that matches our embedding dimension. By adding non-linearity, we allow our model to express more complex predictions than linear. A probabilistic model's ability to make complex predictions that can't be defined linearly is owed to their ability to remove linearity at checkpoints within the model. This design follows a modern bottleneck design which is expand, process then compress. By doing this we are able to add richer intermediate representations into our model, allowing the model to express more complexity.

6 Training the Model

To start training the model, we separate our model into two separate sets of data: training data and validation data. For our model, we used the first 90% of the data to train and had the last 10% to use for validating our model's performance. When training a model, we want to test how well the model performs on the data it's training on, but it's important to understand how well the model performs on new data. If the model is still reducing the training

loss and validation loss and we stop training, then we consider the model to be underfit. If the model continues to reduce training loss while validation loss is increasing, the model is overfit. This occurs when the model learns the nuances of the training data too well, so the model stops making an accurate prediction based off of context and instead makes an accurate prediction by knowing what token should come next from previous training iterations. Training the model is a delicate task that involves finding the sweet spot between being underfit and overfit and stopping training there.

6.1 Determining Loss

In order to determine loss in generative text models, we use a technique called cross-entropy. Cross-entropy matches how well your prediction matches the true distribution of the next token in the sequence. The true distribution is a vector matching the size of our vocabulary that stores a 1 in the position indicating the true next token, and a 0 everywhere else. When training, we pass our predictions for the next token before softmaxing, which are called logits, and our true distribution to our cross-entropy function from PyTorch. The function then softmaxes our logits and uses this equation:

$$H(p,q) = -\sum_{i=1}^{\text{vocab size}} p_i \log(q_i) \tag{15}$$

where p_i is our true probability vector and q_i is the model's prediction vector for the next token in the sequence. From this, the equation will output our loss. We determine our total loss for each iteration of training by averaging our loss per prediction over the total number of predictions the model made.

6.2 Backpropagation

Modern machine learning models are trained using backpropagation. Backpropagation involves sending training data through the model and comparing the output to the expected output from this input. We then traverse the model backwards and tweak parameters in order to make the expected outcome more likely to occur with the same input. By traversing the model backwards, we are able to calculate the derivative of the loss function with respect to our trainable parameters. Therefore, we are able to determine to what extent each parameter influenced our output for a certain set of inputs and adjust these parameters accordingly in order to train the model output to be something closer to what we expected. In order to determine how to adjust the parameters, during backpropagation when we reach a trainable parameter, we use a process called gradient descent. For each tunable parameter in the model, once we have calculated the derivative of the loss with respect to this parameter, we can compute what's called the gradient. The gradient tells us what direction to adjust this parameter in order to reduce loss. We then use our optimizer function in order to determine exactly how much to adjust this parameter. For every training input,

we calculate this gradient for every trainable parameter within the model and adjust it accordingly.

6.3 Iterations

We trained this model through iterations of data. For every iteration, we would take a batch of random text samples and use them to train the model at every token in the sequence. This is a different concept than epochs. An epoch would involve running our training data in its entirety through the model to train it, where instead we sample from our training data randomly. By training this way, we can avoid the model learning our training data too well so we are able to avoid overfitting. In order to train this model, each iteration took a batch of 32 random sequences of which each sequence was 256 tokens long. This means that for each iteration we had

$$32 * (256 - 1) = 8160 \tag{16}$$

training cases.

7 Conclusion

Throughout the process of developing this model, I have developed a far deeper understanding of the architecture within transformers. I understand how attention is implemented and its purpose at a much higher level, and I have learned terminology that is used in the development of these models so that I can more effectively communicate information regarding these models. Following this, I have learned about the challenges that are posed by developing a language model, such as gathering enough data to adequately train the model and preparing large volumes of data so that it can be used for training.

The final product of this study is a sandbox model that could be used to learn about how language models are built and developed. We developed the model to a point where various parameters, such as training iterations or learning rate, can be tweaked to study the effects on the model. If someone wanted to continue to develop data using our pidgin English, we have included the exact prompt that we used to generate all of our training data. However, if a student developed a vocabulary and was able to generate a usable amount of data, this model could be trained to generate in any language. Ideally, the framework for this model will be torn apart in order to learn how generative text models are built and why these transformer architecture design choices were made.