CMPE 258 Project Proposal:

# Sentence-to-Sentence Tiny Story Generation

Group 2

Lindsey Raven
Su Hyun Kim
Nicolas Guerrero
Anoushka Sawant
Eric Cardinal
Peter Conant

# Table of Contents

# Motivation

The recent rise of Large Language Models (LLMs) like ChatGPT and DeepSeek piqued the interest of our group as they are widely used within the industry and are frequently discussed as being “revolutionary”. Like many in the field, we seek to gain a better understanding of their mechanics so that we may know how to improve upon them. Most exploring these models will seek to tune or apply them to a novel application in an effort to gain understanding. Unlike most, we seek to gain more of an “under the hood” sort of understanding and hope to learn the models from the ground up.

The core use case of these models is text to text translation. This process involves quantifying information from an input text and decoding, transforming, or generating an output text based on said input. We specifically wanted to learn how to build a SLM (small language model), which also leverages the concepts of a LLM, from scratch. We feel that this is the best method to gain the “ground up” understanding of these prevalent LLMs and their learning process. We emphasize small architecture, as we wanted to be able to build this model using our own limited hardware. To facilitate this need, we propose a project concept that is achievable using a SLM, while still being able to maintain an architecture that is similar to what a LLM would use (just scaled up). We also altered our concept to be novel, so that we won't just use a pre-existing existing language model to achieve our project concept.

# Proposed Project

For our project, as mentioned above, we propose designing, implementing, and training a language model from scratch. While Large Language Models (LLMs) like GPT-4 and LLaMA boast over a billion parameters and deliver impressive performance, they are computationally expensive to train and deploy. In contrast, our goal is to **develop a lightweight language model**—with fewer than 10 million parameters—that can still perform competitively on the task of text generation.

Specifically, we plan to build a model that is designed to generate the middle portion of a short story, given only the first and last sentences as input. This sentence completion task requires the model to synthesize a coherent narrative that bridges the two endpoints, maintaining proper grammar, contextual relevance, and narrative flow. By tackling the "hardest part" of a story, this project aims to support writers facing creative blocks while still providing the learnings we seek. This concept is also more novel, as most existing models tackling similar tasks aim to generate the entirety of the text, or text given only the beginning. In contrast, our proposed concept is more difficult as it constrains the model more than normal by forcing it to only generate content that properly bridges the two sentences.

To achieve this text generation task, we plan to combine masked language modeling mechanisms with autoregressive generation, allowing the model to sequentially predict the bridging content between the two sentences in a constrained setting. The training data comes from the TinyStories

dataset, which features short, well-structured stories with a limited vocabulary—ideal for training small models effectively. For example, given the input:

"I am a teacher. <b> students math."

The model must predict the missing phrase that connects the two ideas into a complete narrative:

"I am a teacher. I **teach students** math."

Ultimately, our goal is to build a compact yet capable language model that can take the first and last sentences of a short story and generate a plausible, contextually appropriate middle which bridges the narrative gap in a meaningful way. This is a novel concept that we also feel will be a great learning project for us. This will give us the tools with which to build LLMs in the future.

# Paper Inspiration

To get a more general background for our LLM project, we wanted to start by investigating older LLM papers. The "Language Models are Few-Shot Learners" [10] paper demonstrates that scaling up language models significantly enhances their ability to perform NLP tasks without the explicit need for fine-tuning. The authors present GPT-3, a 175-billion-parameter model, over ten times larger than its predecessor, showcasing that increased parameter count improves learning. By training on a massive corpus, GPT-3 generalizes across tasks like translation, question-answering, and text completion using only a few examples or even just a prompt. The study reinforces the notion that larger models exhibit emergent abilities, making them more versatile but also requiring vast computational resources and training data. This research underscores the challenge that most general-purpose LLMs face: their sheer scale makes training from scratch infeasible for most, pushing researchers to explore smaller yet effective alternatives [10].

While the previous paper presents the utility of LLMs, this project was inspired by the paper titled "TinyStories: How Small Can Language Models Be and Still Speak Coherent English?" by Ronen Eldan and Yuanzhi Li at Microsoft Research [11]. The paper explores the feasibility of training extremely small language models (under 10 million parameters) to generate coherent English text. The authors introduce TinyStories, a synthetic dataset of simple short stories created by GPT-3.5 and GPT-4. They demonstrate that models trained on this dataset, despite their small size and simple architectures, can produce fluent, grammatically correct, and even reasoning-capable text. Given small amounts of text at the beginning of the story, they have found that SLM's are capable of generating elegant, if not simple, finishes to the story. The study also presents a novel evaluation framework using GPT-4 as a grader, which assesses grammar, creativity, and consistency, providing a more nuanced metric than traditional benchmarks. The findings suggest that even minimalistic models can exhibit emergent language abilities, shedding light on scaling laws and interpretability in small-scale LMs, which could benefit low-resource or specialized domains [11].

The aim of our project is to train a small language model from scratch that generates a complete short story given its first and last sentences. This differentiates ourselves from the TinyStories paper in that they propose models to either generate the story entirely, or to generate a story given the beginning text. The TinyStories paper inspired us by showing that even small models can produce fluent and coherent text. We wanted to explore this concept further by developing our own model that can perform a similar purpose to the models used in the TinyStories paper while still being unique. Unlike large, general-purpose LLMs that require massive datasets and computational resources—far beyond what we could train from scratch, our approach focuses on a more constrained yet meaningful task. By designing a model specifically for story generation within this structured format, we can take advantage of smaller-scale training while still achieving strong narrative coherence.

# Proposed Dataset

Our proposed dataset is as mentioned above: the TinyStories dataset [1, 12]. It consists of short stories with a small vocabulary (around the reading level of a three-year-old), generated by GPT-3.5 and GPT-4. The GPT models were prompted to include a randomly chosen verb, noun, and adjective from the vocabulary, as well as a specified feature (bad ending, dialogue, etc). The training dataset is 1.92 GB and contains 2,119,719 stories, 1,799,249 of which are unique.

Here is an example prompt by Eldan and Li:

> *Write a short story (3-5 paragraphs) which only uses very simple words that a 3 year old child would likely understand. The story should use the verb "decorate", the noun "thunder" and the adjective "ancient". The story should have the following features: the story should contain at least one dialogue, the story has a bad ending. Remember to only use simple words!*

An example output and data entry in the dataset is:

> Once upon a time, in an ancient house, there lived a girl named Lily. She loved to decorate her room with pretty things. One day, she found a big box in the attic. She opened it and saw many shiny decorations. Lily was very happy and decided to use them in her room.
> As Lily was decorating her room, the sky outside became dark. There was a loud thunder sound, and Lily got scared. She ran to her mom and said, "Mommy, the thunder is so loud!"
> Her mom hugged her and said, "Don't worry, it will pass soon." But the thunder did not stop. It got louder and louder, and the ancient house started to shake. Suddenly, the roof fell down on the pretty decorations. Lily was sad because her room was not pretty anymore. The end [12].

# Proposed Model/Architecture

The goal of our project is to implement a SLM (small language model) architecture from scratch to perform a complicated text-to-text generation task (bridging two sentences with a meaningful test). On huggingface, there exist several models trained using the TinyStories dataset [1, 2]. Each of these models has varying levels of complexity and number of parameters while still achieving arguably "good" results. Our primary reason for referencing the TinyStories paper is that it emphasizes using a small architecture. In order to further our understanding and allow for more time for experimentation within our project, we will attempt to use a basic decoder only transformer model where we will vary the number of layers as we progress with our experimentation. Discussed in the following sections is a summary of the common limitations language models face, a high level summary of the decoder transformer model, some existing decoder transformer model architectures we may use, and a discussion of how we can expand or shrink our architecture given results.

## Language Model Limitations:

Language modeling, and text generation, has many limitations. It wasn't until recently, with the extra computational power that came with the onset of the GPU, that language modeling was even viable. When learning a language, there are many "rules" and "relationships" that need to be followed in order for a generated text to be coherent. As an added layer of complexity, these relationships and rules need to be remembered over a large span of text. Original models processing text struggled to handle the longer term dependencies of the text, and therefore were limited in their context window and scope [13]. To overcome this, the concept of attention was introduced [14], with attention, relationships could be quantified over a global range. At present, language models leverage the concept of attention to map the global relationships of text and make either transformative or generative decisions. However, attention requires comparing each element of a text, to every other element making the complexity O(n^2) at best. Therefore, in order to handle the many possible relationships text can have, as well as the large possible vocabulary that is ever expanding, language processing tends to require models with millions of parameters. In order to counteract this, AI architects tend to limit the complexity and type of vocabulary the model sees as well as limiting the amount of text the model needs to process. In our case, we will leverage both as well as leverage existing architectures shown capable of performing language modeling.

## Decoder Transformer Model:

### Proposed architecture: Decoder only transformer model

Most NLP and LLM models, at their core, are transformer decoder based. Transformers focus on the use of attention to compute similarity and relationships between tokenized data [5]. Once these relationships have been quantified, they can then be applied to address various NLP tasks such as

text generation and (with the additional use of an encoder) transformation. A high level visual of a decoder only transformer model is illustrated in Figure 1.

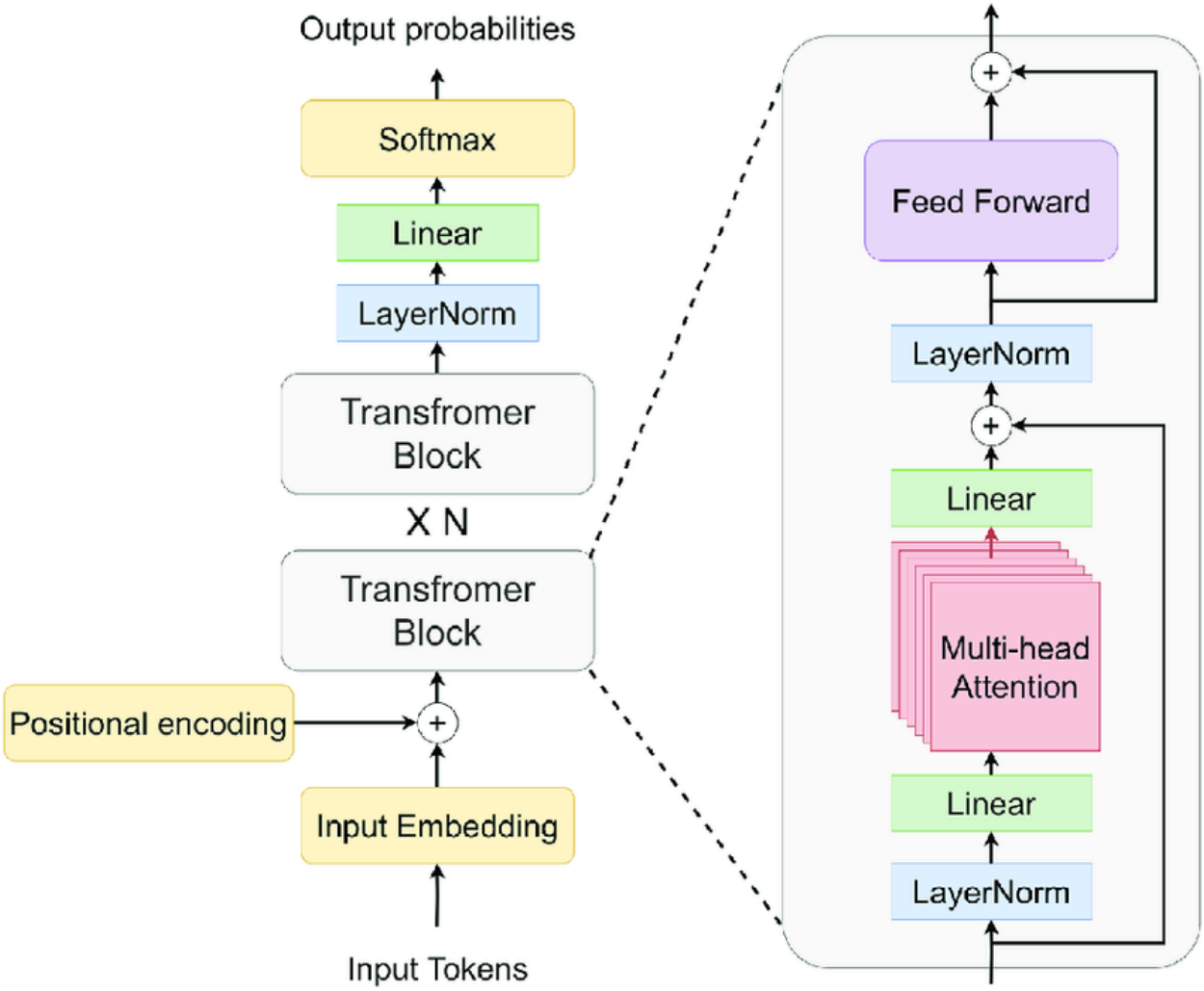


Figure 1. Decoder only transformer mode [3]

A breakdown of each of the components illustrated in Figure 1. Is shown in Table 1.[4]:

Table 1. Decoder Only Architecture Component Breakdown

| Block Name | Description |
|---|---|
| **Input Tokens** | Sequence of tokenized words fed into the model. |
| **Input Embedding** | Converts discrete token indices into dense vectors of fixed size. |
| **Positional Encoding** | Adds position information to embeddings since transformers do not have inherent order awareness. |

| | |
|---|---|
| **Transformer Block (xN)** | A repeated **decoder block** consisting of multi-head self-attention, feedforward layers, and normalization layers. |
| **Multi-head Attention** | Computes self-attention multiple times in parallel, capturing different representations of input. |
| **Linear (Post Attention)** | A projection layer applied to attention outputs, transforming them into a suitable form for the next layer. |
| **LayerNorm (Pre & Post Attention)** | Normalizes inputs to stabilize training and improve convergence. |
| **Feed Forward Layer** | A two-layer fully connected network with an activation function (usually **GELU**). |
| **Residual Connections (Skip Connections)** | Adds the original input to the transformed output to allow better gradient flow. |
| **Final LayerNorm** | Normalization applied before final linear projection. |
| **Linear (Final Projection)** | Maps the model output to vocabulary size, preparing it for softmax probability calculation. |
| **Softmax** | Converts logits into probability distributions over vocabulary tokens for text generation. |

As you can see by the Figure and Table. The primary goal of a transformer decoder model is to take a set of input tokens (which are created with the use of a tokenizer algorithm), and as an output give a probability distribution over the entire vocabulary (to be clear: the tokenized vocabulary) to determine what is the next likely token in the vocabulary that should be generated (this is done using the final linear and softmax layer). Thus, it should be apparent that the main determining factors of the complexity of the model are the token sequence size limits (context window), the vocabulary size, and the token embedding dimension. To illustrate the dimensional changes between each layer, see Table 2. for details.

Table 2. Input and output dimensions of each layer where B = batch size, T = sequence length, d_model= hidden dimension, V = vocab size.

| **Layer** | **Input Shape** | **Output Shape** |
|---|---|---|
| Token Embedding | $(B,\ T)$ | $(B,\ T,\ d_{model})$ |
| Transformer Blocks | $(B,\ T,\ d_{model})$ | $(B,\ T,\ d_{model})$ |

| | | |
|---|---|---|
| Final Linear Layer | $(B,\ T,\ d_{model})$ | $(B,\ T,\ V)$ |
| Softmax | $(B,\ T,\ V)$ | $(B,\ T,\ V)$ |

In order to determine the vocabulary size, as well as have an understanding of the sequence length, it is necessary to 1) understand what the possible range of words are and 2) what should the limit be to the sequence length given the application. In our case, our dataset is limited to the simple vocabulary given in the TinyStories dataset. However, it is possible that we may expand our dataset to include more text other than that found in TinyStories. This being the case, we will need to understand and be prepared to develop our own vocabulary constraints and tokenization scheme. The process for tokenization and how we will define our vocabulary is documented in the next section.

# Tokenizer Algorithm:

## Proposed tokenization algorithm: Byte pair encoding

Tokenization is the process of breaking down the input data into individual components which are then fed into the model for training. In the case of NLP, input text is broken down into words, characters, or even sub-text. These tokens are then mapped to indices for the model to finally process. A visualization of this process can be found in Figure 2. [6]

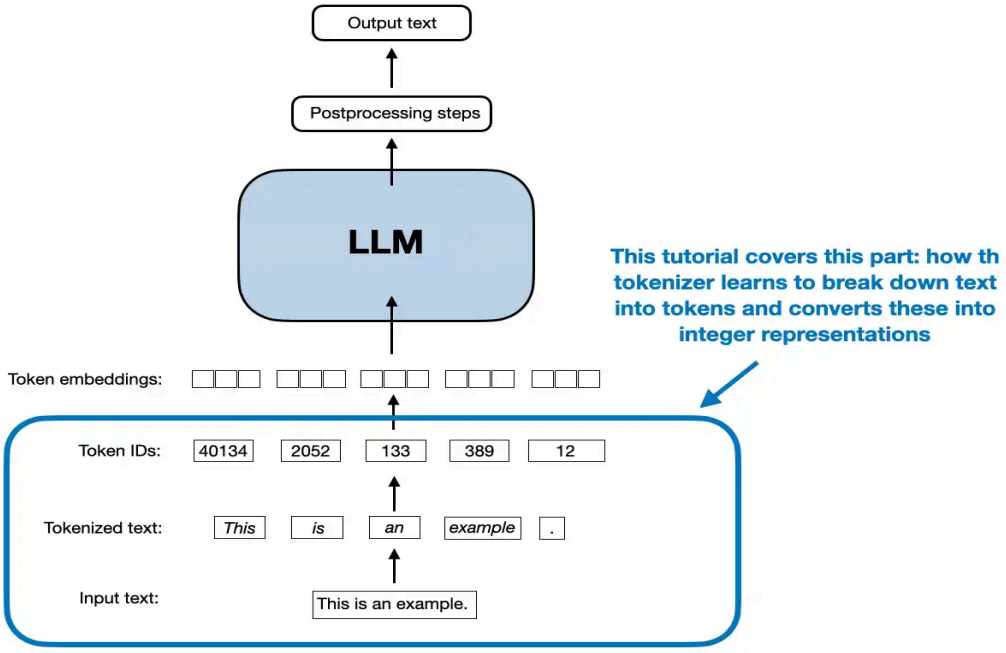


Figure 2. How tokenization may be performed and fed into a model.

There are many tokenization algorithms for how to break down the text sequence, each with their own benefits. It may be a first assumption to break down the sequence by characters. While this provides the benefit of being able to construct any kind of word out of all possible characters (ASCII character count is 128), the sequence length will end up increasing drastically as there are many characters in a given text. The model is constrained to a context window limit, and as mentioned prior, the longer the sequence the larger the complexity and memory requirements. By using characters, it is very easy to exceed that context window size. Additionally, the model will require more depth to properly learn all the relationships as more relationships exist between characters across text compared to words. To circumvent these drawbacks, one may then assume to switch to a word based tokenization. However, the English language has more than 170,000 unique words. Therefore, the complexity benefits that came from reducing the sequence is reintroduced by forcing the model to map to an extremely large vocabulary. As an alternative to both of these approaches, there are several optimized tokenization schemes. Some of the common algorithms include the following:

Table 3. Different Tokenization Algorithms

| Algorithm | Description | Pros | Cons |
|---|---|---|---|
| **Byte Pair Encoding (BPE)** | Iteratively merges most frequent character pairs into subwords. | Efficient for subword tokenization, reduces OOV words | Fixed merges may not generalize well, can split words awkwardly |
| **WordPiece** | Similar to BPE but maximizes likelihood over training data. | More flexible than BPE, better handling of rare words | Requires large pretraining corpus, can produce long token sequences |
| **Unigram Language Model** | Trains probability distribution over all possible subwords. | More adaptable, works well in multiple languages | Computationally expensive, higher memory usage |
| **Character-Level Tokenization** | Treats each character as a token. | No OOV issues, works well in low-resource languages | Produces very long sequences, slower inference |
| **Word-Level Tokenization** | Treats entire words as tokens. | Easy to understand and implement | Fails with OOV words, large vocabulary required |

| | | | |
|---|---|---|---|
| **Byte-Level BPE (BBPE)** | Extends BPE to the byte level, supports any text encoding. | Works across languages, no OOV words | Tokenized text can be longer than necessary |
| **Morpheme-Based Tokenization** | Uses linguistic knowledge to split words into meaningful units. | More accurate meaning representation | Requires language-specific processing |

For our project, we will be focusing on using the byte pair encoding tokenization scheme. Byte pair encoding is simple in concept: the goal is for the algorithm to create sub-text tokens based on the most frequently occurring character and sub-text groupings. [9] A high level description of the algorithm flow is as follows:

1. **Initialize with characters:**
    - Begin by splitting the text into individual characters. Each word is represented as a sequence of characters with a special separator character (representing a space).
2. **Count symbol pairs:**
    - Count how frequently adjacent symbol pairs (bigrams) occur in the text dataset (emphasis on it being the entire dataset!).
3. **Merge the most frequent pair:**
    - Replace the most frequent adjacent pair with a new token that is a combination of the two.
    - Add this new token to the vocabulary.
    - Repeat this process until reaching the desired vocabulary size.
4. **Tokenization using learned merges:**
    - After training, any new text is tokenized by applying the learned merge rules in order, breaking words into subwords when necessary.

A visual example of how the algorithm works is as follows:

**Step 1: Start with words split into characters**

word1: l o w e r
word2: n e w e s t

word3: w i d e s t

**Step 2: Count frequent adjacent pairs**

("l", "o") → 1
("o", "w") → 1
("w", "e") → 2
("e", "r") → 1
("e", "s") → 2
("s", "t") → 2
...

**Step 3: Merge the most frequent pair**

Suppose ("e", "s") is the most frequent; merge it into "es":

l o w e r
n e w es t
w i d es t

**Repeat until vocabulary limit is reached**

Next, merge ("es", "t") → "est":

l o w e r
n e w est
w i d est

By using byte pair encoding, you are left with a vocabulary of a desired length that maximizes the benefits that can be had with both character based tokenization and word based tokenization. As a part of our project, we may experiment with different vocabulary lengths and see the impacts on the model performance. This experiment will be required if we introduce more text outside the TinyStories dataset. These experiments will come at somewhat of a high cost as they will require different models to be trained as the model architecture is dependent upon the vocabulary size. In the TinyStories paper, they used a vocabulary size that varied depending on the model used. The simplest vocabulary was around 5k tokens with more complicated models approaching 100k+ tokens, all of which was generated using byte pair encoding. Since these various vocabulary sizes proved successful with the given dataset (TinyStories), we will start by using a vocabulary of a smaller size, and then expand as we attempt models of larger complexity. In the next section are different decoder only transformer architectures we will likely use as a part of our project.

## Simple Decoder Transformer Model (1M Param):

The simplest decoder only transformer model that has quantifiably decent results generating stories when trained with the TinyStories dataset is the 1M parameter model. As a base architecture, it follows the same architecture mentioned above that is typical for a decoder only transformer model. However, the main variations between each of these architectures is the change in context window (allowing for more tokens to be generated), the change in embedding dimension (allowing for more complex representation of each of the tokens), the number of attention blocks (allowing for more types of relationships between tokens to be discovered and quantified), and finally the vocabulary size (which allows for more possible words to be generated and represented in the final output). As these parameters increase, so does the complexity requiring more memory and training time.

The main benefit of the 1M model is that it does not require a high-end GPU to train and maintains a relatively low complexity. For the 1M parameter model, below is a table breaking down the architecture specifications:

Table 4. 1M Parameter Model Architecture Specifications

| Feature | TinyStories-1M |
|---|---|
| **Total Parameters** | ~1M |
| **Layers (Transformer Blocks)** | 4 |
| **Embedding Dimension** | 128 |
| **Number of Attention Heads** | 4 |
| **Feedforward (MLP) Size** | 256 |
| **Context Window (Max Tokens)** | 128 |
| **Vocabulary Size** | ~5,000 |
| **Activation Function** | ReLU |
| **Layer Normalization** | Post-Norm |

This architecture is ideal as a proof of concept for validating our implementation of a decoder transformer model for achieving our given task. Should the model struggle to demonstrate improvement in solving our task, we would likely need to investigate if our understanding and implementation of the decoder transformer architecture is accurate. We have confidence that this model will be able to perform for our problem statement, as this architecture has already proven successful with a similar problem in generating stories using the TinyStories dataset.

## GPT-2 Decoder Transformer Model:

The GPT decoder transformer architectures are considered some of the most foundational models for LLMs today. Therefore, we would like to use one of the smaller GPT models (small enough to allow for non-specialized hardware to train) to try to solve our problem statement and provide a reference point to the smaller 1M parameter model for solving our problem statement. We will likely use the GPT-2 architecture, as it was also trained with the TinyStories dataset in the past to perform story generation and has proven successful. At its core, the GPT-2 architecture is still the same simple architecture documented above describing a decoder only transformer model, except now the complexity has increased significantly compared to the 1M parameter model. See Table 5.

Table 5. GPT-2 Architecture Specifications

| Feature | GPT-2 (Smallest Version) |
|---|---|
| **Total Parameters** | ~124M |
| **Layers (Transformer Blocks)** | 12 |
| **Embedding Dimension** | 768 |
| **Number of Attention Heads** | 12 |
| **Feedforward (MLP) Size** | 3072 |
| **Context Window (Max Tokens)** | 1024 |
| **Vocabulary Size** | 50,257 |
| **Activation Function** | GELU |
| **Layer Normalization** | Pre-Norm |

With this increase in architecture complexity, we hope to see improvement in the quality of the text generated as filler between the two given sentences in a given prompt.

## Our Approach / Discussion:

The TinyStories dataset and paper introduce several SLMs that are capable of generating story text. In order to further our understanding of decoder only transformer models, we are focusing on solving a slightly different task with the same dataset. This way we now have baseline architectures to work from that are known capable of solving similar problems. By focusing on using a smaller model first (with the 1M parameter model) we can cement our implementation and understanding of transformer decoder models. After which we can focus on optimizing solutions for our given task. These optimization methods will involve experimenting with more complex architectures (such as GPT-2) that increase the vocabulary and model depth and, as a stretch goal for our project, we may

possibly introduce a secondary dataset with slightly more complicated text and see if we can still develop a successful model for our given task (this will require a new tokenization mapping as well as vocabulary requirements.)

# Train and Test Methods

## Training Methods

To train a language model, we have to define what the input and output expectations are. As mentioned in prior sections, the input is dependent upon the tokenization mappings and context window, while the output is dependent upon our defined vocabulary. There are many methods for training language models, and documented in the following sections are three representative methods which we plan on using: **Masked Language Modeling (MLM)**, **Autoregressive Language Modeling,** and **Teacher Forcing**.

### Masked Language Modeling (MLM)

Illustrated below is a visual of how one might apply masking to their language modeling and training.

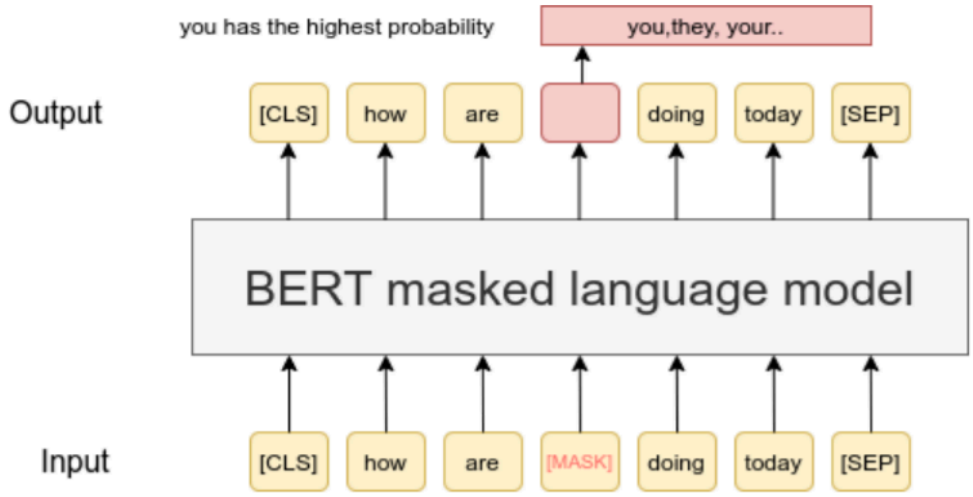


Figure 3. Masked Language Modeling Technique in BERT model

MLM is a self-supervised learning technique where a model is trained to **predict masked or hidden words** in a sentence, using the surrounding context. By using the surrounding context, MLM forces the model to exhibit strong bidirectional contextual understanding. As seen in Figure 3, this is the learning method used in the BERT NLP model [7]. BERT is a very standard nlp model commonly used in LLMs. We plan to employ masking as a part of our training and inference process, as we will be using a [MASK] to indicate to the model where it needs to begin generating text.

## Autoregressive Language Modeling

Autoregressive language modeling is a method describing how the model is expected to generate a given set of tokens. With this method, it is expected to generate the sequence token by token. A visual of the process can be seen below:

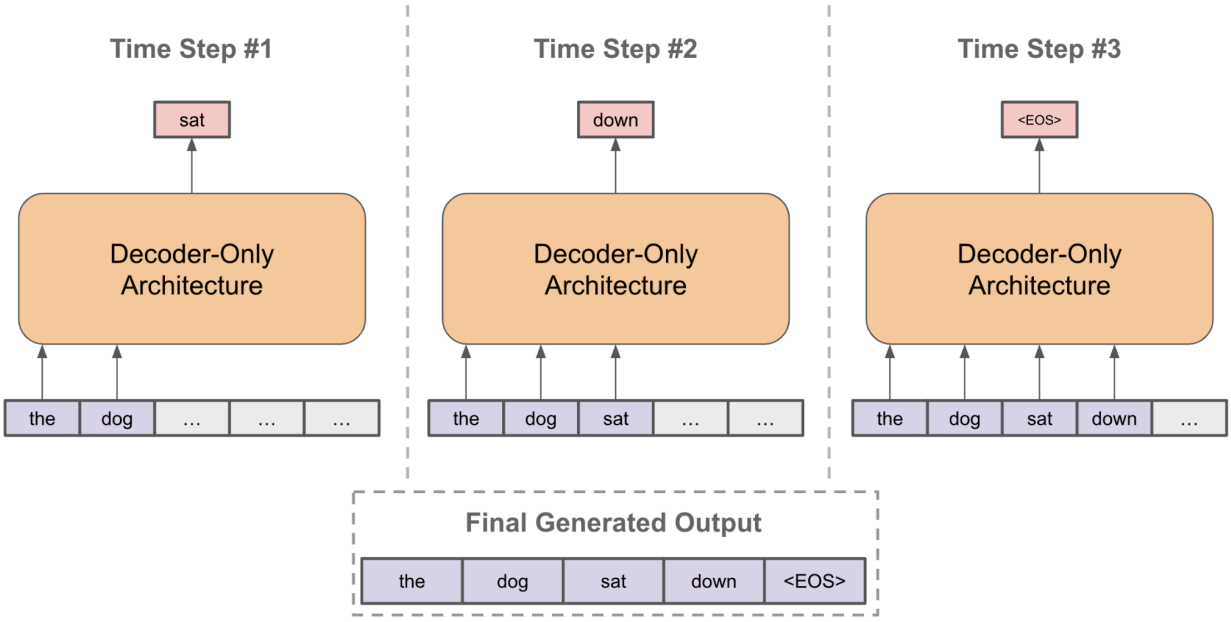


Figure 4. Autoregressive Language Modeling

Autoregressive Language Modeling **predicts the next token in a sequence based on the preceding tokens**. This forces the model to build on the previous context in order to generate a coherent sequence. This type of text generation is ideal for use cases where the missing or generated text has little to no context to direct the given model. This will be true for our use case, as the model will only have the beginning and end sentence with which to generate the middle context. As an example, shown in Figure 4, the model first uses the word "the" and "dog" to predict "sat", and then it additionally uses "sat" to predict the next word "down". This training method is also commonly used in GPT models [8]. Autoregressive Language Modeling enhances the models' generative capabilities and sequential coherence.

## Teacher Forcing

Teacher forcing is a training technique in which the correct label is fed to the model instead of the model's own prediction at each given sequence point (this is the same as the time step mentioned in Figure 4.); for our use case, it will be fed in at each hidden state or time step. To clarify: hidden state means each state where the model generates a token before reaching the EOS token (done generating the sequence). This method is handy in a sequence dependent task, including token generation, as errors made in early time steps or hidden states will cause the loss of the model to exponentially grow (this is true for auto-regressive token generation where each token

generated is dependent on the prior one). By using teacher forcing, this results in providing the model an opportunity to correct itself and not be too heavily penalized early on in training. This results in faster convergence, reduced error propagation and more stable training early on in the training cycle. While the limitations include exposure bias and lack of robustness. It is a common practice to employ teacher forcing in the early stages of model training and then switch to a normal paradigm later in training when the model is capable of predicting accurate tokens. Thus, we will likely be forced to use this method early on in our training process in order to ensure that our model converges.

## Our Approach

To train our model to generate bridging sentences that connect the first and last sentences of a story, we employ a hybrid approach. Our approach combines **masked language modeling** and **autoregressive generation** together.

In particular, we replace the middle sentence(s) of the story with a special <b> token (blank token), and train the model to reconstruct the missing content in a step-by-step, autoregressive fashion. Below is an example of our training method.

Table 6. Example of inputs and outputs of our model

| | | |
|---|---|---|
| Ground truth | I am a teacher. I teach students math. | |
| 1st input | I am a teacher. <b> students math. | Target output: I |
| 2nd input | I am a teacher. I <b> students math. | Target output: teach |
| 3rd input | I am a teacher. I teach <b> students math. | Target output: <eos> |
| Final output | I am a teacher. I teach students math. | |

Rather than predicting the entire missing span (“I teach”) at once, we train the model to generate one token at a time, regressively expanding the <b> token. This is done by feeding the model updated versions of the input at each timestep, where previously predicted tokens are inserted back into the sequence (or the correct token in the case of using teacher forcing).

To be clear: **token selection is done by selecting the token with the highest probability**, as the model output is the probability of selection over the entire vocabulary (this is found by the final linear and softmax layer of the architecture). As an initial approach we will use the greedy selection methodology, which involves selecting the token based on the highest probability. But we may also explore beam search, which is where we also maintain the second or third most probable selections throughout and make a final sequence selection based on the combined probability. This methodology prevents the model from falling into local minima but is costly in terms of complexity (as you have to find each token selection along each possible path), so we may forgo testing this.

Autoregressive formulation allows the model to learn how to incrementally fill in the blank based on context, rather than relying on simultaneous masked prediction. We find this approach especially useful since the length of the missing tokens can vary, and sequential generation enables better modeling of narrative flow and grammar.

We also incorporate teacher forcing during the early stages of training—at each timestep, the correct previous token is fed into the model, regardless of its earlier prediction. This helps prevent error accumulation in early stages of training and encourages faster convergence.

## Hyperparameters

To optimize model performance, we will likely conduct hyperparameter tuning in the ranges shown in Table 7 (although this is not definitive, it's just what we are planning),

Table 7. Hyperparameters to optimize

| **Hyperparamter** | **Description** | **Values to Explore** |
|---|---|---|
| Learning Rate | Controls the step size for weight updates during training. A high learning rate may cause instability, while a low learning rate may lead to slow convergence. | 1e-3, 1e-4 |
| Batch Size | Determines the number of samples processed before the model updates its weights. Larger batch sizes provide stable gradient updates but require more memory. | 32, 64, 128 |
| Embedding Dimensions | Defines the size of token representations. Higher dimensions allow for richer encodings but increase computational cost. | 128, 256, 512 |
| Dropout Rate | Regularization technique to prevent overfitting by randomly dropping neurons during training. | 0.1, 0.2, 0.3 |
| Number of Layers | The number of stacked transformer decoder blocks in the model. A deeper model captures more complex relationships but increases computational cost. | 2, 4, 6 |
| Sequence Length | Maximum number of tokens the | 128, 256, 512 |

| | | |
|---|---|---|
| | model can consider at once. A larger window allows better context retention but increases memory usage. | |
| Optimizer | Optimization algorithm used to update model weights. | Adam, AdamW, SGD |
| Weight Initialization | The method used to initialize model parameters before training. Poor initialization leads to local minima. | Xavier, Uniform Random |

# Model Evaluation and Test Methods

Like most supervised learning methods: the main evaluation method involves comparing the generated story to the original ground truth story from the dataset. When it comes to text evaluation in general, there are several methods to evaluate a model's performance. Discussed in the following sections are methods to be aware of and may use as a part of our project.

## Quantitative Evaluation Metrics

### Cross Entropy Loss

The most standard loss function we will use to evaluate the model's prediction during training is the cross-entropy loss. Cross-entropy is a standard loss function to be used in any sort of model which provides probability over a class distribution. In our case, our model will produce a probability of each token class being the next selection. Cross entropy measures the negative log-likelihood of the correct token at each timestep.

$$\mathcal{L}_{\text{total}} = \frac{1}{T} \sum_{i=1}^{T} -\log p(x_i \mid x_{<i})$$

In our tokenized sequence $X = (x_0, x_1, \ldots, x_t)$, the log-likelihood of the ith token, conditioned on the preceding tokens, is given by $p_\theta(x_i|x_{<i})$. This measures the model's ability to predict the next token within the given sequence. During training at any given timestep, we compute the loss for the current token given the ground truth, and we accumulate this loss until the sequence is fully generated. We then take the average of the accumulated losses at each timestep to determine the final loss (this is accounted for in the above equation with $T$ representing the total timesteps). Using this methodology, we plan to train our model.

### Perplexity (PPL)

Perplexity is a standard measure in language modeling that quantifies how well the model predicts unseen text. It is defined as the exponential of cross entropy loss.

$$PPL(X) = exp\left\{-\frac{1}{t}\sum_{i}^{t}\log p_{\theta}\left(x_i \mid x_{<i}\right)\right\}$$

A lower perplexity indicates that the model assigns **higher probabilities** to correct predictions. This perplexity will be used as an evaluation metrics to check the performance of a model. We likely will focus more on the cross entropy results for training, but may use perplexity for visualization and tuning.

### Token Accuracy

Token accuracy measures how often the model correctly predicts the next token in a sequence. Given a batch of input sequences, the token accuracy is calculated as:

$$Accuracy = \frac{\#\ of\ correctly\ predicted\ tokens}{\#\ of\ total\ tokens}$$

This helps us to understand how well the model understands word relationships and is another common metric. We will again likely use this metric for visualization purposes.

### Inference Speed (Tokens Per Second, TPS)

Many language models use inference speed as performance metrics because real-time text generation is important. TinyStories model may not be critical on text generating time, but we can use this as quantitative evaluation metrics to compare with other models. We will likely prioritize this metric last given time constraints for the project.

$$TPS = \frac{\#\ of\ generated\ tokens}{Time\ taken\ (seconds)}$$

### Bilingual Evaluation Understudy (BLEU) Score

BLEU score measures the similarity between the generated bridging sentences and the ground-truth text by comparing n-gram (a sequence of 'n' consecutive items such as words, characters, or syllables) overlap. Higher BLEU scores stand for better alignment with the reference text. This is a very common metric for evaluating generated text and we will likely explore this metric for model comparison.

Beside these evaluation metrics, there are other indicators such as Recall-Oriented Understudy for Gisting score or BERT score. However, we are not going to consider other quantitative evaluation methods in this project.

### Qualitative Evaluation

Apart from numerical metrics, we also perform human evaluation of the model's generations. It should be apparent from the model output whether the generated context is even legible. Qualitative evaluation includes story coherence, fluency, grammar, creativity, and diversity, which are difficult to assess numerically. In an effort to quantify and assess these metrics, we will evaluate using human assessment as well as similar methodology to that mentioned in the TinyStories paper: have a LLM such as GPT-4 give an assessment of the quality of the generation.

### Our Approach

In summary: our model will use masked inputs with a specialized (ex. <b>) token, and will generate the missing span auto regressively. To quantify performance we will use **cross-entropy loss** during training, which measures the loss of the output and the correct token at each step and allows for effective optimization. In the final section we discuss what our expectations are and requirements will be to determine the success of our project.

## Final Project Outcome

As detailed by this document: the goal of this project is to explore the ability of SLMs to generate coherent intermediate text between two sentences. This concept is novel and further pushes the boundaries of what SLMs can handle (further constrained to bridge the sentences). To evaluate success, we will use a combination of quantitative metrics discussed prior and human reviews to assess the quality, coherence, and relevance of the model's output of bridging text.

For our project, our minimum success expectation is to create a model that demonstrates improvement in token generation throughout training, and ends with a text that is comparable to that of a six-year-old. More specifically, we expect the text to demonstrate a basic grasp of grammar, a child's vocabulary, and the coherence expected in a simple story while still following the constraint of properly bridging the two sentences.

Should we see the model capable of generating content that is visibly coherent, we will then move to more nuanced forms of evaluation. This is when we will begin to experiment with more complicated and complex architectures as mentioned in the architecture section of this document. Assuming each of the generated architectures is successfully trained and meets the minimum requirement mentioned above: we will use the more specific and quantitative methods to evaluate each model's performance. This will involve: computing token accuracy, the BLEU score, comparing the losses during training, and passing the responses to a LLM (such as GPT-4) for grading (similar to how the TinyStories paper evaluates their model responses).

By retrieving said metrics, we would have a more direct comparison for each model's output. This would allow us to further experiment, learn how to tune SLM models, and gain an in-depth understanding of what specific architecture characteristics help SLMs, as well as LLMs, improve performance. Additionally, there are six members of this project team, all of us will be implementing our own versions of the decoder-only transformer architecture. These metrics will also allow us, within our own team, to compare our progress with each other throughout the course of this project.

Finally, if successful we expect to end with an optimized architecture that is a POC (proof of concept) that can be used as a tool for creative writers. Using this model, they may train the model with their own text and can utilize it to help them bridge their own text. We feel that this use is more beneficial than a model generating the text in its entirety as it allows for more creative freedom from the writer to direct the model output with the ending text. Furthermore, we hope to end this project with a more in depth grasp of language models that we look to carry forward in future research.

# References


[1] "Roneneldan/TinyStories-1M · Hugging Face." *Huggingface.co*, 12 Aug. 2024, huggingface.co/roneneldan/TinyStories-1M. Accessed 17 Mar. 2025.

[2] "Roneneldan/TinyStories-33M · Hugging Face." *Huggingface.co*, 12 Aug. 2024, huggingface.co/roneneldan/TinyStories-33M. Accessed 17 Mar. 2025.

[3] Learning to reason over scene graphs: a case study of finetuning GPT-2 into a robot language model for grounded task planning - Scientific Figure on ResearchGate. Available from: https://www.researchgate.net/figure/llustration-of-a-forward-pass-through-RobLM-for-text-generation-with-a-greedy-next-token_fig3_373183262 [accessed 17 Mar 2025]

[4] Ph.D, Cameron R. Wolfe. "Decoder-Only Transformers: The Workhorse of Generative LLMs." *Deep (Learning) Focus*, 4 Mar. 2024, cameronrwolfe.substack.com/p/decoder-only-transformers-the-workhorse.

[5] Vaswani, Ashish, et al. “Attention Is All You Need.” *ArXiv*, 12 June 2017, arxiv.org/abs/1706.03762.

[6] Raschka, Sebastian. “Implementing a Byte Pair Encoding (BPE) Tokenizer from Scratch.” *Sebastian Raschka, PhD*, 17 Jan. 2025, sebastianraschka.com/blog/2025/bpe-from-scratch.html. Accessed 19 Mar. 2025.

[7] J.~Devlin, M.-W. Chang, K.~Lee, and K.~Toutanova, "BERT: Pre-training of deep bidirectional transformers for language understanding," in *Proc. NAACL-HLT*, 2019, pp. 4171--4186.

[8] J. Achiam, S. Adler, S. Agarwal, L. Ahmad, I. Akkaya, F. L. Aleman, et al., “GPT-4 technical report,” arXiv preprint arXiv:2303.08774, 2023. [Online]. Available: https://arxiv.org/abs/2303.08774

[9] “Byte-Pair Encoding (BPE) in NLP.” *GeeksforGeeks*, 17 Apr. 2023, www.geeksforgeeks.org/byte-pair-encoding-bpe-in-nlp/.

[10] T. Brown et al., “Language Models Are Few-Shot Learners,” ArXiv (Cornell University), vol. 4, May 2020, doi: https://doi.org/10.48550/arxiv.2005.14165.

[11] R. Eldan and Y. Li, “TinyStories: How Small Can Language Models Be and Still Speak Coherent English?,” arXiv.org, May 24, 2023. https://arxiv.org/abs/2305.07759 (accessed Jul. 09, 2023).

[12] Eldan, R., Li, Y. (2023). TinyStories: How Small Can Language Models Be and Still Speak Coherent English?. arXiv preprint arXiv:2305.07759, 2023. [Online]. Available: https://arxiv.org/abs/2305.07759.

[13] P. J. Werbos, "Backpropagation through time: what it does and how to do it," in *Proceedings of the IEEE*, vol. 78, no. 10, pp. 1550-1560, Oct. 1990, doi: 10.1109/5.58337.

keywords: {Backpropagation;Artificial neural networks;Supervised learning;Pattern recognition;Neural networks;Power system modeling;Equations;Control systems;Fluid dynamics;Books},

[14] Vaswani et al. . “Attention Is All You Need.” *ArXiv*, 12 June 2017, arxiv.org/abs/1706.03762.