Exploring Language Modeling

Uday Shukla, Sunny Liang, Jonathan Moradkhan

Spring 2025

1 Motivation/Introduction

We began with the broad goal of creating/replicating something that is prominently used in the modern academic and professional landscape. So after some deliberation, we settled on pursuing the study and build of a basic language model, given that we ourselves interact with them almost daily. Language models are undeniably integral in society today, as they are the backbone behind text completion, machine translation, and chatbot development software. Furthermore, as text-generative models integrate into our daily workflows more and more with each passing day, it seemed like a more than appropriate time to study their inner workings.

For our project specifically, we focused on building and improving a simple text-generative model and sought to understand each one of its many components. Our progression/process can be split into the following overarching sections:

1. **Motivation/Introduction**(page 1)
2. **The Simple Model:**(pages 2-11) We begin by building a small-scale language model that takes five words as input and learns to predict the sixth.
3. **Model Evaluation/Scoring:**(pages 12-13) We assess the quality of the model's predictions using two different evaluation methods: GPT-2 loss (a likelihood-based metric) and BERTScore (a semantic similarity metric).
4. **Hyperparameter Tuning:**(pages 13-15) We explore architectural variations and hyperparameter tuning to improve prediction performance.
5. **Future Improvements/Analysis:**(page 15) We discuss what we could have done to possibly improve our analysis
6. **References and Github**(page 16)

2 The Simple Model

2.1 General Setup

The central goal of our project was to construct a simple but effective language model that, given a sequence of five words, could predict a reasonable sixth word. Formally, if a sentence is tokenized into words $[x_1, x_2, x_3, x_4, x_5, x_6, \dots]$, we aimed to train a model f such that:

$$f(x_1, x_2, x_3, x_4, x_5) \approx x_6$$

This formulation allowed us to explore the fundamentals of natural language modeling without the computational overhead of generating entire sentences. From the outset, we were interested in how a model could leverage context to learn a useful representation of word sequences and produce coherent continuations.

To approach this task, we turned to recurrent neural networks (RNNs), which are designed for processing sequential data. RNNs maintain a hidden state that evolves as new inputs arrive, making them well-suited to learning temporal dependencies and contextual patterns. This made them an ideal choice for modeling five-word inputs and predicting the sixth.

Within the family of RNNs, we selected the Gated Recurrent Unit (GRU), a powerful architecture that incorporates gating mechanisms to control how much of the past is remembered or forgotten. This built-in memory control made GRUs particularly attractive for our small-scale setting. By embedding input words, feeding them through a GRU, and projecting the final hidden state into a vocabulary-sized output space, our model was able to learn and generate semantically and syntactically reasonable sixth-word predictions.

2.2 Text Preprocessing

Before training our model, we needed to clean and prepare the raw text so that it could be used to generate supervised input-target pairs. For this project, we used *Alice's Adventures in Wonderland* by Lewis Carroll, which we obtained through Project Gutenberg.

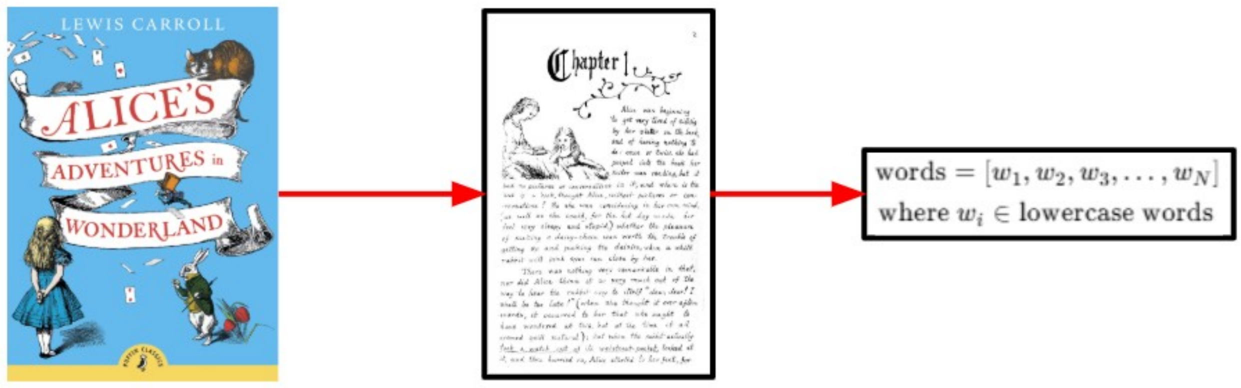
The goal of preprocessing was to reduce noise and standardize the format of the text, ensuring that the model focused on meaningful patterns rather than superficial variation. Our preprocessing steps included:

- Converting all characters to lowercase, so that `Alice`, `alice`, and `ALICE` are treated identically.
- Removing all punctuation and special characters using regular expressions.
- Splitting the text into a list of space-separated word tokens.

Mathematically, if T is the raw text string, we defined a preprocessing function P such that:

$$P(T) = [w_1, w_2, \dots, w_N]$$

where each w_i is a lowercase word with punctuation removed.


LEWIS CARROLL
ALICE'S ADVENTURES in WONDERLAND
Chapter I
words = [$w_1, w_2, w_3, \ldots, w_N$]
where $w_i \in$ lowercase words

This resulted in a long list of clean, individual tokens that could be used to generate training examples. For instance, the input:

```
"Alice was beginning to get very tired"
```

became:

$$[\texttt{alice}, \texttt{was}, \texttt{beginning}, \texttt{to}, \texttt{get}, \texttt{very}, \texttt{tired}]$$

The result of this preprocessing was a word-level representation of the full text

$$[w_1, w_2, \ldots, w_N]$$

which served as the basis for constructing input-output pairs in the next stage of the pipeline.

2.3 Vocabulary Building and Tokenization

Once we had a clean list of words from the text, we needed a way to convert those words into a numerical format usable by our model. Neural networks operate on numbers—not raw text—so this stage was essential to bridge the gap between natural language and mathematical input.

Vocabulary Construction: To create a manageable and effective representation of the text, we constructed a vocabulary $\mathcal{V}$ consisting of the 5,000 most frequently occurring words in the text. Each of these words was assigned a unique integer index in the range $[0, 4999]$, where:

- $\texttt{word2idx}(w) = 0$ corresponds to the most frequent word,
- $\texttt{word2idx}(w) = 4999$ corresponds to the 5,000th most frequent word.

Any word not found in this top-5,000 list was mapped to a special token called `<UNK>`, which was assigned the integer index 5000.

This capped vocabulary size was chosen for both computational efficiency and to reduce the model's risk of overfitting to rare, outlier words. It also enabled us to keep our embedding and output layers relatively compact, while still capturing the vast majority of meaningful content in the dataset.

Formal Mapping Definitions:
The forward mapping is defined as:

$$\texttt{word2idx}(w) = \begin{cases} i & \text{if } w = w_i \in \mathcal{V}, \text{ with } i \in [0, 4999] \\ 5000 & \text{if } w \notin \mathcal{V} \quad (\texttt{<UNK>}) \end{cases}$$

The inverse mapping is defined as:

$$\texttt{idx2word}(i) = \begin{cases} w_i & \text{if } i \in [0, 4999] \\ \texttt{<UNK>} & \text{if } i = 5000 \end{cases}$$

Example: A sequence like [`alice`, `was`, `beginning`, `to`] might be mapped to:

$$[15,\ 8,\ 329,\ 22]$$

If a rare word like `serpentines` is not in $\mathcal{V}$, then:

$$\texttt{word2idx}(\texttt{serpentines}) = 5000$$

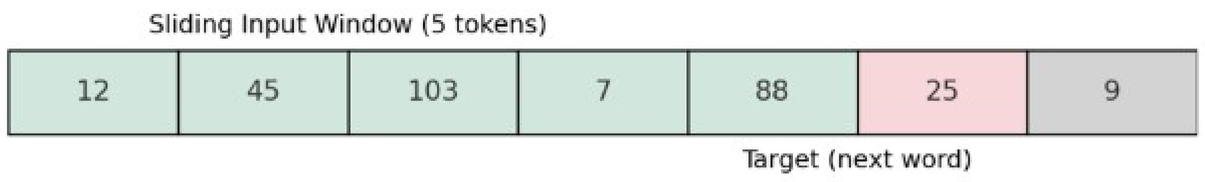
Resulting Data: At the end of this step, our entire text had been transformed into a long sequence of integers/tokens, each corresponding to a known word or the fallback `<UNK>` token.

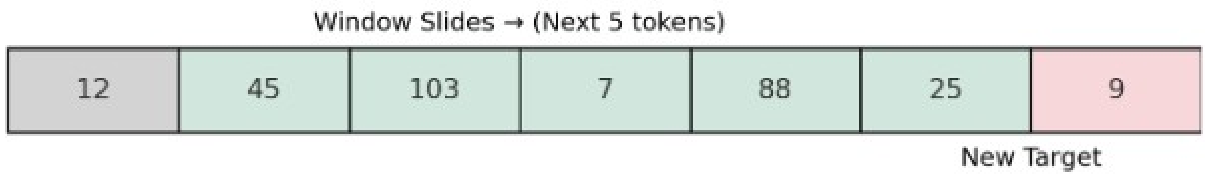
$$[w_1,\ w_2,\ \ldots,\ w_N] \xrightarrow{\texttt{word2idx}} [v_1,\ v_2,\ \ldots,\ v_N]$$

These numeric encodings served as the raw material for the next step—constructing input-output training pairs.

2.4 Constructing Input–Target Pairs

Once we had transformed the raw text into a single sequence of integers, we generated the supervised learning data by applying a sliding window of size six. For every group of six consecutive tokens, the first five served as input, and the sixth was used as the target label.





Let the full list of encoded tokens be:

$$[v_1,\ v_2,\ v_3,\ \ldots,\ v_N]$$

We created training pairs by sliding a window of length six across this sequence. For each position i from 1 to $N-5$, we defined:

$$\text{Input: } \mathbf{x}^{(i)} = [v_i,\ v_{i+1},\ v_{i+2},\ v_{i+3},\ v_{i+4}]$$

$$\text{Target: } y^{(i)} = v_{i+5}$$

This gave us a dataset of the form:

$$\left\{(\mathbf{x}^{(i)},\ y^{(i)})\right\}_{i=1}^{N-5}$$

Each input–target pair provided a complete example for training our model to predict a word given its five-word context. This step laid the foundation for the next stages of the pipeline, where we would embed the words and use them in a recurrent neural network.

2.5 Batching and Embedding

After forming all the (input, target) pairs, each input was a list of 5 consecutive token indices—for example:

$$[12,\ 88,\ 3,\ 420,\ 76]$$

To accelerate training, we grouped 64 such sequences into a single batch. This yielded an input tensor of shape:

$$\texttt{input\_batch} \in \mathbb{Z}^{L\times B} = \mathbb{Z}^{5\times 64}$$

where $L = 5$ is the sequence length and $B = 64$ is the batch size.

However, these integer IDs carry no semantic meaning on their own—they are merely categorical labels. To provide the model with useful input, we converted each token index into a learned continuous vector using PyTorch's `nn.Embedding` module. Internally, this module defines an embedding matrix $\mathbf{E}$ has shape:

$$\mathbf{E} \in \mathbb{R}^{V\times d}$$

where:

- V is the vocabulary size (in our case, $V = 5001$), and
- d is the embedding dimension (we used $d = 32$).

Each row $\mathbf{E}_i$ of the matrix corresponds to the ith word in the vocabulary and contains its learned 32-dimensional embedding vector.

For a given input token x_i, the embedding layer performs a lookup:

$$\texttt{Embed}(x_i) = \mathbf{E}_{x_i} \in \mathbb{R}^{32}$$

These embeddings are initialized randomly but are learned during training. That is, as the model backpropagates its prediction error, the gradients flow not only through the GRU and output layers, but also back into the embedding matrix. This allows the embedding vectors to gradually adjust so that semantically or syntactically similar words lie closer together in the learned space.

The output of the embedding layer for a single batch of shape (64, 5) becomes a 3D tensor of shape:

$$\texttt{embedded\_batch} \in \mathbb{R}^{64\times5\times32}$$

This embedded input is then passed to the GRU layer for sequential modelling in our neural network:

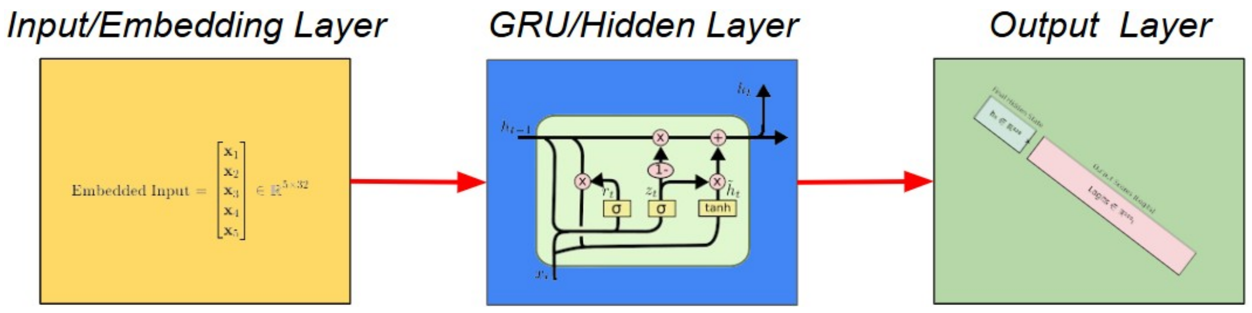
- 64 = batch size (number of sequences),
- 5 = sequence length (tokens per input),
- 32 = embedding dimension (per token).

2.6 Overview of the Neural Network Architecture

The model we use is a shallow feed-forward recurrent neural network. Although minimalist in structure, the model successfully demonstrates the key mechanisms behind language modelling through its use of the GRU/hidden layer.

The core of the network consists of three sequential components:

1. An **embedding layer** that converts each input token (an integer) into a 32-dimensional dense vector.(This is discussed above)
2. A **GRU (Gated Recurrent Unit)** layer that reads the 5-token embedded sequence and summarizes it into a single context vector.
3. An **output layer** that maps this context vector to a probability distribution over the entire vocabulary.



While only one such transformation is needed to produce a prediction from a given 5-word input, this process must be repeated for *every training example* in the dataset — of which there are thousands. During each epoch, the model processes batches of 64 such input sequences at a time. A single epoch therefore involves hundreds of forward passes through the network.

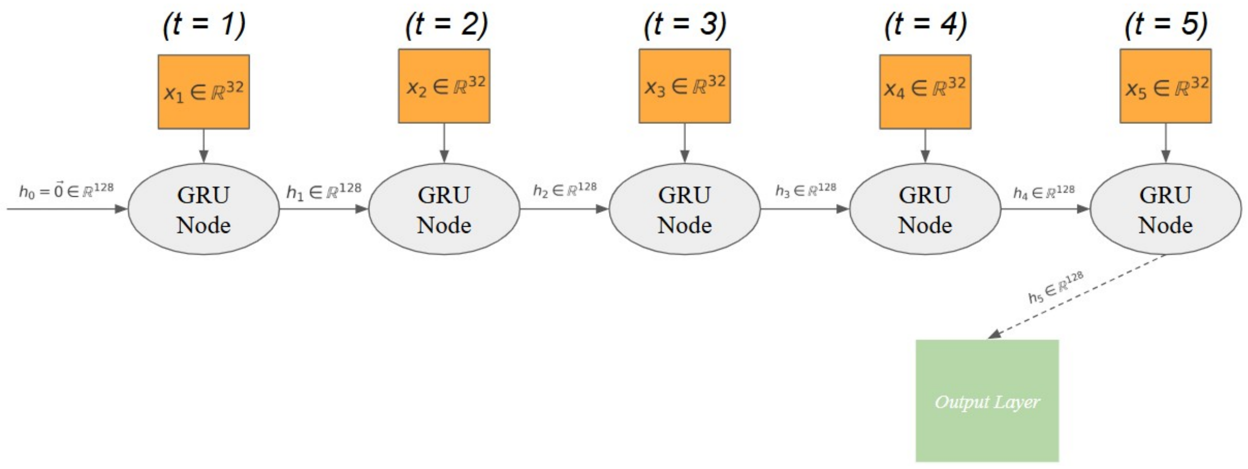
To keep things interpretable, we show only one input $\in \mathbb{R}^{5\times 32}$ within a batch travelling through the hidden layer (the GRU). In the next two sections, we walk through this single forward pass in detail: first by examining the GRU layer and its internal logic, and then by following the transformation into vocabulary scores.

2.7 GRU / Hidden Layer

Once the input sequence has been embedded into dense vectors, it is passed through a Gated Recurrent Unit (GRU) layer — the core recurrent structure in our model. GRUs are a type of Recurrent Neural Network (RNN) architecture designed to capture sequential dependencies in data while addressing the vanishing gradient problem that often plagues traditional RNNs.

Sequential Structure: Each input sequence has 5 tokens, and after embedding, we have a matrix of shape $[5 \times d]$, where d is the embedding dimension (32 in our case). The GRU reads this matrix one row (word embedding) at a time, updating its internal hidden state at each time step. The hidden state $\mathbf{h}_t$ at time step t serves as a summary of the sequence up to the t-th word. Intuitively, the model makes sense for our task as, when predicting a sixth word in a phrase, you expect the fifth word and first to both carry some weight in the prediction, but the fifth to carry significantly more "predictive weight" than the first.

$$\mathbf{h}_t = \mathrm{GRU}(\mathbf{x}_t, \mathbf{h}_{t-1})$$



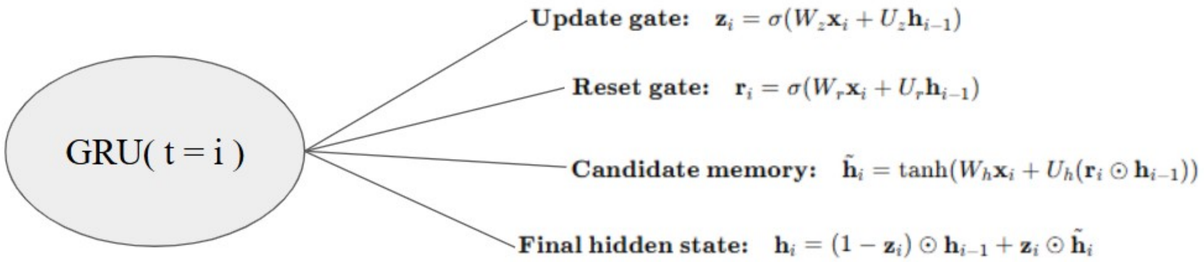
(t = 1)
(t = 2)
(t = 3)
(t = 4)
(t = 5)
$x_1 \in \mathbb{R}^{32}$
$x_2 \in \mathbb{R}^{32}$
$x_3 \in \mathbb{R}^{32}$
$x_4 \in \mathbb{R}^{32}$
$x_5 \in \mathbb{R}^{32}$
$h_0 = \vec{0} \in \mathbb{R}^{128}$
$h_1 \in \mathbb{R}^{128}$
$h_2 \in \mathbb{R}^{128}$
$h_3 \in \mathbb{R}^{128}$
$h_4 \in \mathbb{R}^{128}$
$h_5 \in \mathbb{R}^{128}$
GRU Node
Output Layer

This recurrence proceeds from $t = 1$ to $t = 5$, resulting in a final hidden state $\mathbf{h}_5$ that summarizes the entire input sequence. This vector is then passed to the output layer.

Internal GRU Computation: Each GRU cell maintains a hidden state that is updated based on the current input and the previous hidden state. It uses two learned gates to control this update:

- **Update Gate** ($\mathbf{z}_t$): Controls how much of the past information to retain.
- **Reset Gate** ($\mathbf{r}_t$): Controls how much of the past information to forget when creating a new memory candidate.

The computations are as follows:


GRU(t = i)
Update gate: $\mathbf{z}_i = \sigma(W_z\mathbf{x}_i + U_z\mathbf{h}_{i-1})$
Reset gate: $\mathbf{r}_i = \sigma(W_r\mathbf{x}_i + U_r\mathbf{h}_{i-1})$
Candidate memory: $\tilde{\mathbf{h}}_i = \tanh(W_h\mathbf{x}_i + U_h(\mathbf{r}_i \odot \mathbf{h}_{i-1}))$
Final hidden state: $\mathbf{h}_i = (1 - \mathbf{z}_i) \odot \mathbf{h}_{i-1} + \mathbf{z}_i \odot \tilde{\mathbf{h}}_i$

Here:

- σ is the sigmoid function, which maps values to $[0, 1]$,
- $\odot$ denotes element-wise multiplication,
- $\mathbf{x}_t$ is the embedding of the current word,
- $\mathbf{h}_{t-1}$ is the hidden state from the previous time step,
- $\tilde{\mathbf{h}}_t$ is the candidate for the next hidden state.

Gate Behavior Intuition:

- When the **update gate** $\mathbf{z}_t$ is close to 1, the GRU heavily favors the new candidate state $\tilde{\mathbf{h}}_t$ over the old state $\mathbf{h}_{t-1}$. This leads to faster learning or adaptation to new input.
- When $\mathbf{z}_t$ is close to 0, the GRU ignores the candidate and retains most of the previous hidden state. This preserves memory and leads to long-term dependencies.
- When the **reset gate** $\mathbf{r}_t$ is close to 0, the contribution of the previous hidden state is suppressed when computing the candidate. This forces the GRU to "reset" its memory and only look at the current input $\mathbf{x}_t$.
- When $\mathbf{r}_t$ is close to 1, the candidate state is influenced by both the current input and the prior memory, allowing for continuity of thought.

Candidate Hidden State $\tilde{\mathbf{h}}_t$: This vector represents a potential new hidden state, built using a modified version of the previous memory and the current input. It flows through a tanh nonlinearity to allow for non-linear transformation.

Final Hidden State $\mathbf{h}_t$: The final hidden state is a learned blend between the previous hidden state $\mathbf{h}_{t-1}$ and the candidate $\tilde{\mathbf{h}}_t$, controlled by the update gate. This adaptive mixing allows the model to decide at each step how much to carry forward and how much to overwrite.

Trainable Parameters in the GRU:
The GRU architecture includes six key trainable parameter matrices (along with associated bias vectors), which are used to transform the input and the hidden state at each time step. These parameters are shared across all time steps and input batches during training:

- $W_z \in \mathbb{R}^{h\times d}$ — input weights for the update gate
- $U_z \in \mathbb{R}^{h\times h}$ — hidden state weights for the update gate
- $W_r \in \mathbb{R}^{h\times d}$ — input weights for the reset gate
- $U_r \in \mathbb{R}^{h\times h}$ — hidden state weights for the reset gate
- $W_h \in \mathbb{R}^{h\times d}$ — input weights for the candidate hidden state
- $U_h \in \mathbb{R}^{h\times h}$ — hidden state weights for the candidate hidden state

These matrices are initialized when the model is created and remain fixed in structure during training. However, their values are gradually updated by backpropagation and gradient descent in order to minimize the loss function over many training examples. The fact that these parameters are shared across all time steps is a core reason for the efficiency and generalization power of recurrent neural networks like the GRU.

Hyperparameters: The behavior and flexibility of the model were shaped by the following key hyperparameters:

- **Embedding Dimension** (d): The size of each word embedding vector. We used $d = 32$.
- **Hidden Dimension** (h): The size of the hidden state inside the GRU. We used $h = 64$.
- **Sequence Length** (T): Fixed at 5, as we only considered 5-word inputs.
- **Learning Rate** (η): Controlled the optimizer's step size. Set to 0.005.
- **Batch Size**: The number of training examples per update. Set to 64.

To conclude all of these hyperparameters were chosen to balance training time and model expressiveness in a computationally efficient way. The GRU layer receives the sequence of 5 word embeddings and processes them in order. We keep only the output at the final time step — $\mathbf{h}_5$ — which becomes a fixed-length representation of the entire input sequence, ready for classification in the output layer.

2.8 Output Layer and Final Prediction

Once the GRU processes the entire five-word input sequence, the final hidden state h_5 contains a compressed summary of the contextual information gathered from all five time steps. This hidden state is then passed to a fully connected (linear) output layer to produce a prediction.

The purpose of this layer is to project the hidden state vector from the GRU (of dimension h) onto a new vector of dimension equal to the vocabulary size $|\mathcal{V}|$, producing a score for every possible word in the vocabulary.

Mathematically, this operation is given by:

$$\hat{y} = W_o h_5 + b_o$$

where:

- $W_o \in \mathbb{R}^{|\mathcal{V}| \times h}$ is the output weight matrix,
- $b_o \in \mathbb{R}^{|\mathcal{V}|}$ is the output bias vector,
- $\hat{y} \in \mathbb{R}^{|\mathcal{V}|}$ is the resulting logits vector.

Each entry $\hat{y}_i$ in the output vector represents an unnormalized score for the i-th word in the vocabulary. These logits are typically passed through a softmax function during evaluation or training to produce a valid probability distribution over all possible next words:

$$P(y = i \mid x_1, \ldots, x_5) = \frac{e^{\hat{y}_i}}{\sum_{j=1}^{|\mathcal{V}|} e^{\hat{y}_j}}$$

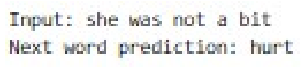
The predicted sixth word is then chosen as the word with the highest softmax probability:

$$\hat{x}_6 = \arg\max_i \hat{y}_i$$

Finally, once the model identifies the most likely token index $\hat{x}_6$, we use the inverse vocabulary mapping `idx2word` to convert this index back into a human-readable word, yielding the model's predicted sixth word.

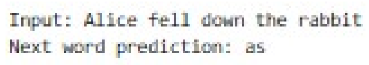
2.9 First Outputs

```python
print("Input:", "she was not a bit")
print("Next word prediction:", predict_next_word("she was not a bit"))
```



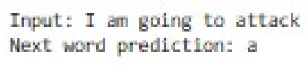
```
Input: she was not a bit
Next word prediction: hurt
```

```python
print("Input:", "Alice fell down the rabbit")
print("Next word prediction:", predict_next_word("Alice fell down the rabbit"))
```



```
Input: Alice fell down the rabbit
Next word prediction: as
```

```python
print("Input:", "I am going to attack")
print("Next word prediction:", predict_next_word("I am going to attack"))
```



```
Input: I am going to attack
Next word prediction: a
```

Although the model's initial predictions looked surprisingly natural and grammatically sound, and we were excited to see it working, we knew that a more systematic and rigorous evaluation was needed. To move beyond surface-level impressions and truly measure performance, we turned to formal scoring methods and set the stage for deeper modelling improvements.

2.10 Transition to PyTorch Training Framework

While our model architecture and design choices were initially implemented from scratch to promote full understanding, we transitioned to using PyTorch's built-in training infrastructure for all subsequent experiments. This included leveraging PyTorch's automatic differentiation engine for efficient gradient computation and its highly optimized neural network modules for scalability and speed.

Additionally, we replaced our preliminary training loops—based on stochastic gradient descent—with the Adam optimizer. Adam combines the benefits of momentum and adaptive learning rates, which greatly improved convergence speed and stability during training. These changes allowed us to maintain the same architecture while gaining computational robustness and reproducibility, paving the way for our more advanced experiments in model scoring, tuning, and double descent.

3 Model Evaluation/Scoring

3.1 Scoring Setup

While being able to look at the sentences "He slammed the wooden door shut" and "He slammed the wooden door hello" and say "well, the first sentence is clearly better than the second" is one way of determining sentence accuracy, we desired a more algorithmic way of grading our true sentences and our predictions. Given two relatively normal-looking sentences, two reasonable people can differ as to which one is a "better" sentence based on their own understanding of language, so we hoped to find a fairer (and easier) way of comparing our sentences. As a result, we turned to publicly available programs that could score our sentences to determine just how fluent they are.

3.2 GPT-2

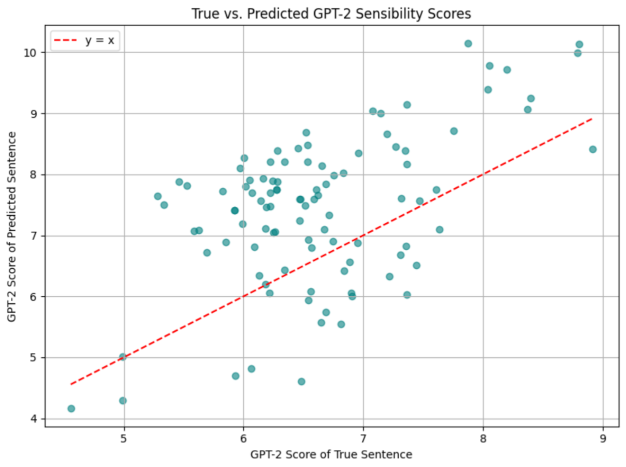


Figure 1: Graph of GPT-2 Scores for True Sentences vs Scores for Predicted Sentences

First, we used GPT-2 to evaluate our method. GPT-2 is a Generative Pre-Trained Transformer, an LLM developed by OpenAI with (among other things) the ability to score different phrases based on their fluency with the English language. The first thing worth noting is that GPT-2 uses reverse scoring, that is, a lower score indicates a more fluent sentence, and a greater score indicates a less fluent sentence. This means that if for a pair (x, y) where x is the score of the true sentence and y is the score of the predicted sentence, the true sentence has a better score if $x < y$. In fact, this occurs, with most of our points appearing above the red $x = y$ line. This does hold up logically, as it would not make sense for the guess of a phrase to be better than the actual phrase itself. With this in hand, we were able to conclude that our model appears relatively reasonable, with predicted scores not significantly worse than true scores.

3.2.1 BERTScore

We also decided to with BERTScore, an alternate scoring system designed to harness the power of Google's BERT (Bidirectional Encoder Representations from Transformers) Natural Language Processor. It is worth noting that BERTScore assesses the similarity between two sentences, in this case checking how similar the predicted sentence is to the true sentence. As a result, it is fairly obvious that the true scores will always be the highest 1 (since the true sentence is equivalent to itself), but we can make more interesting conclusions with the predicted data. The predicted

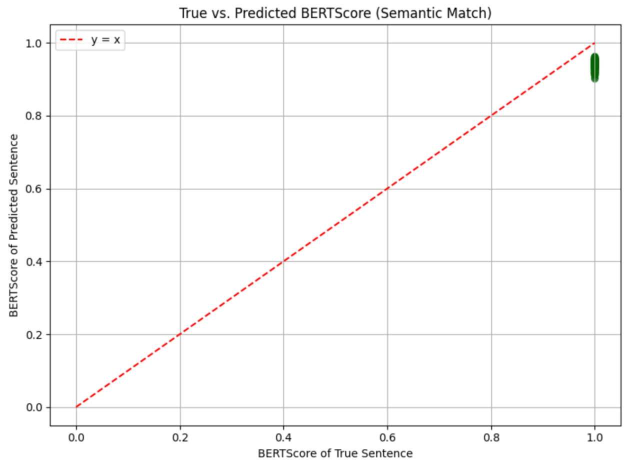


Figure 2: Graph of BERTScores for True Sentences vs BERTScores for Predicted Sentences

sentences all scored fairly close to 1, meaning that it matches the original sentence pretty well, and a small spread indicates it is relatively consistent with this result.

4 Hyperparameter Tuning

For brevity, this section will use d to denote the number of embedding dimensions, h is the number of hidden dimensions, and n is the number of hidden layers, each of dimension h.

4.1 Training Epochs

As expected, the loss decreases asymptotically as we increase the number of training epochs. Double descent is not observed.

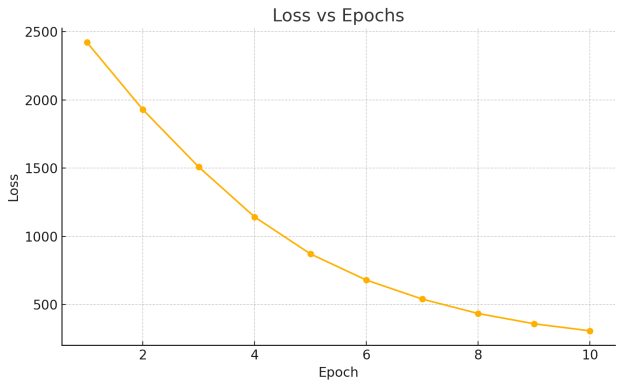


Figure 3: $d = 64$, $h = 128$, $n = 1$

For the remainder of this section, we will train all models over 10 epochs.

4.2 Number of Dimensions

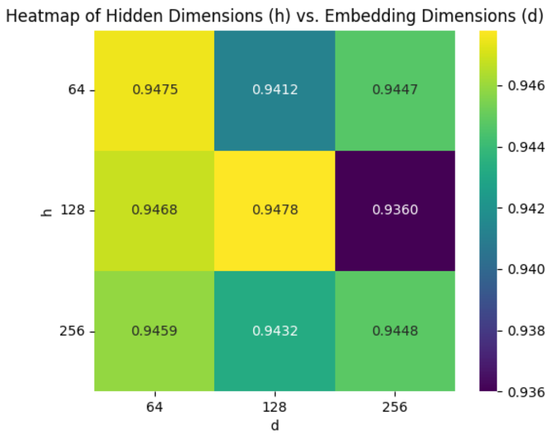


Figure 4: This is a heatmap

Increasing the number of embedding dimensions and number of hidden dimensions didn't necessarily improve performance. Based on the heatmap as seen in Figure 4, and also by subjectively comparing sample predictions from the different models, a happy medium of 128 embedding dimensions and 128 hidden dimensions was optimal.

All remaining models in this section use $d = 128$ and $h = 128$ accordingly.

4.3 Number of Hidden Layers

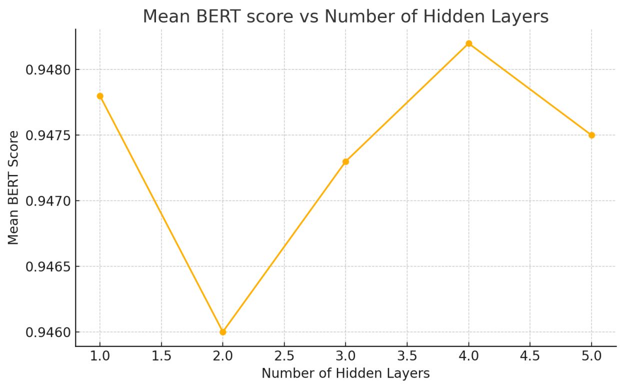


Figure 5: Not very helpful

As seen in Figure 5, varying the number of hidden layers doesn't affect the mean BERT score in an easily identifiable way. Looking at the sample sentences produced though, offers more insight into the effect of increasing n.

Referring to Table 1, you can see that a 1 layer produces mostly grammatical sentences, 4 layers makes some subjective improvements (notice the "dormouse" sentence) but at the expense of diversity, and, lastly, 8 layers predicts "she" every single time. It seems that as we increase the number of hidden layers, the model begins to converge to a reasonably effective but undesirable

Predicted (1 layer)	**Predicted (4 layers)**	**Predicted (8 layers)**
She walked into the room and she	She walked into the room and the	She walked into the room and she
The sun was setting behind her	The sun was setting behind the	The sun was setting behind she
He opened the book and the	He opened the book and the	He opened the book and she
They ran through the field while the	They ran through the field while the	They ran through the field while she
A dog barked at the other	A dog barked at the dormouse	A dog barked at the she
He looked at her and the	He looked at her and the	He looked at her and she
It started to rain just as she	It started to rain just as the	It started to rain just as she
The child smiled when a	The child smiled when on	The child smiled when she
The quick brown fox the	The quick brown fox thing	The quick brown fox she
He never expected to see that	He never expected to see the	He never expected to see she

Table 1: Predicted sentences from models with 1, 4, and 8 hidden layers

strategy of simply predicting one of the most common words regardless of the input (words like "the" and "she").

5 Future Improvements/Analysis

Our model's performance was greatly limited by training time and computing power. A larger model with more neurons and, more importantly, a larger training dataset would greatly improve generalized performance but would require significantly greater computational resources to train.

We observed that as we increased the size of our model (number of dimensions and hidden layers), the model began to simply predict the most common words regardless of the input. Injecting some sort of randomization into both the training stage and prediction stage of the model's implementation would allow the model to produce more natural and varied sentences.

The aforementioned phenomenon may well be an artifact of our loss function. We used a cross-entropy loss function that likely incentivized our models to defer to a small vocabulary of common words despite the resulting sentences not performing well according to BERTScore and human intuition. Implementing a more intuitive scoring system in the training stage in the form of a corresponding loss function is computationally and conceptually complex, but would likely mitigate this problem.

6 References and Github

6.1 References:

- Carroll, Lewis. *Alice's Adventures in Wonderland.* Project Gutenberg. `https://www.gutenberg.org/ebooks/11`
- Paszke, A., et al. "PyTorch: An Imperative Style, High-Performance Deep Learning Library." Advances in Neural Information Processing Systems (NeurIPS), 2019.
- Zhang, T., Kishore, V., Wu, F., Weinberger, K. Q., & Artzi, Y. (2019). *BERTScore: Evaluating Text Generation with BERT*. GitHub. `https://github.com/Tiiiger/bert_score`

6.2 Github:

`https://github.com/moonblazingbagel/math156proj.git`