Homework 3 Report: Transformer is All You Need

Honglin (Leo) Wang
UNI: hw3124
The Fu Foundation SEAS, Columbia University, New York, NY 10034 USA
hw3124@columbia.edu

Abstract

This project presents the design and implementation of a compact Transformer-based language model, dubbed TinyTransformer, optimized for next-token prediction on the Tiny Shakespeare dataset. The model, comprising approximately 327,000 parameters with two Transformer blocks and a moderate embedding dimension of 128, leverages a Byte Pair Encoding (BPE) tokenizer adapted from GPT-2 to effectively handle the dataset's linguistic structure. Training was conducted with an AdamW optimizer on token sequences of length 50, achieving a validation perplexity of 44.7 at epoch 19. Analysis of attention heatmaps reveals the model's capacity to learn semantic token relationships and language structure despite its limited size. Challenges such as data leakage, overfitting, and hyperparameter tuning — especially learning rate sensitivity — are discussed. This work demonstrates that even a minimalist Transformer can capture meaningful language patterns, offering insights into the trade-offs between model complexity and performance in language modeling tasks. Overall, TinyTransformer illustrates that a minimalist Transformer architecture can provide insights into language modeling.

1 Introduction

Problem Definition As the core architecture underpinning modern large language models like GPT-series, the Transformer relies on self-attention mechanisms to capture long-range contextual relationships in text. This project aims to design and train a compact and efficient Transformer language model from scratch, evaluating its performance on next-token prediction tasks. To maintain computational feasibility, this project constructs a "Tiny Transformer" containing only hundreds of thousands of parameters, moderate hidden dimensions, and few Transformer modules.

The core task is autoregressive next-token prediction. The model learns to generate the most probable next token based on an input token sequence using attention. Performance evaluation primarily employs the perplexity metric, which quantifies the model's confidence in predicting unknown sequences.

TinyShakespeare Dataset The Tiny Shakespeare dataset was featured in Andrej Karpathy's release of Char-RNN. It is mainly comprised of several of Shakespeare's plays including the entire play *Coriolanus*, *Richard II-III*, *Romeo and Juliet*, *Henry VI Part III*, *The Winter's Tale*, *Measure for Measure*, *The Taming of the Shrew* and a portion of *The Tempest*. All the lines were separated in a fashion that not a single line exceeds 64 bytes and the separations were made between semantically holistic unit words or punctuations. There are 40k lines in total if counting on all the empty lines and character cues, or 25,555 lines of speech.

Objective / Metric Metrics used to evaluate the model performance is cross entropy and its exponentiation, perplexity (ppl). Our goal is to let the model learn as more semantic structure of natural language (English) as possible and test to use it as language model to generate tokens.

2 Methods

All codes and experiments are implemented under the environments as follows: Python 3.12.7, Numpy 1.26.4, Seaborn 0.13.2, Pytorch 2.9.0, Transformers 4.51.3 and MatPlotLib 3.9.2. In the remaining part of this report, the vocabulary size is set to 500 and max sequence length to 50.

2.1 Tokenization

The dataset were loaded and various methods of tokenization were attempted. We found several things about this dataset. First, we tried to load the full dataset but the empty lines. PieceWord tokenizer was attempted and we found it creates a whole bunch of names of the characters, all in upper case. This is not suitable for a true language model, where a name has usually less semantic meaning, different from a single word. We tried Byte Pipe Encoder (BPE) then, the names still show up many times, taking up roughly 30% of our vocabulary. Judging on the situation, we decided to remove the character cues as well as the empty line. Moreover, we capitalize (only the first letter in a word is in upper case) the names shown up in the context to avoid the encoder to learn meaningless capital pieces of names rather than meaningful word roots and affixes. PieceWord is pretty much suitable for large dataset where the word frequency are distributed more uniformly, but not for this dataset because there exists too much repetition of character names and the PieceWord would still try to learn about name fractions out of capitalized words. We finally decided to use BPE tokenizer.

All single letters show up in our vocabulary, thus the introduction of special tokens *Unknown*<unk> is unnecessary, except for unknown new letters or special tokens like punctuation, which we don't care about, thus all tokens of the original corpus are included as well as any English words. By repeating the training process, we found that the model performs better if we use the pretrained tokenizer of GPT-2 as a base tokenizer. BPE tokenizer is applied after the GPT-2 tokenizer splits all the lines in the corpus of tiny Shakespeare. This tokenizer introduces one more special token Ġ to replace the heading space of a word. Then by greedy search, BPE successfully captured meaningful word pieces such as "est", "re", "ing", "ish", etc. and tokenized the corpus into 423,643 tokens in total.

2.2 Data Preprocessing

Given the max sequence length, we created 423,593 pairs of input and target as X and y for the model, each with length 50 (units in tokens).

We split the data into 2 parts, train and val (validation) for model training and evaluation respectively, with validation taking up to 20% of the whole dataset. A random split could result in revealing later tokens in targets (to be predicted) in earlier tokens in inputs, causing data leakage. In fact, we tested if there is a strong sign of data leakage causal of random split by evaluating the model on validation set and found that the model performs so well (gained perplexity ~22), even with a slightly smaller loss on validation than that on train set. In comparison, we focused on the ordinary data split, i.e. splitting the data by simply one cut. The first 80% is used as train and last 20% as validation. Actually there is still data leakage. Nevertheless, the proportion of data leakage is so low: only 50 later tokens are revealed in val, taking up to 0.01% of the whole dataset, thus the side effect of data leakage is negligible and our implementation is reasonable. Finally, we arrived at a train set of 338,874 entries and validation of 84,719.

2.3 Positional Embedding

We implemented the regular sinusoidal positional embedding to tokens to be fed in our tiny transformer, with a base b, an embedding dimension d_e and position t, the positional embedding P_t^d at embedding dimension d is:

$$\mathrm{P}_t^d = \begin{cases} \sin t \exp_b\left(-\frac{d}{d_m}\right) & t \equiv 0 \pmod 2 \\ \cos t \exp_b\left(-\frac{d}{d_m}\right) & t \equiv 1 \pmod 2 \end{cases}$$

2.4 Model Architecture

We built a minimal Transformer-based language model called TinyTransformer. We decided the embedding dimension $d_e = 128$ for embedding layers and hidden states in FFNN (Feed Forward Neural Network) $h = 128$. We used 2 Transformer blocks in TinyTransformer, each inputs token sequences embeddings through a self-attention module after an RMS normalization layer, then fed into an FFNN and after getting RMS normalized again, residual connections were made after self-attention and the FFNN. We implemented the self-attention almost identical to that in our colab notebook, with causal mask applied for the task of next token prediction.

The model has 326,784 parameters in total and FFNN fully connection contributes a lot. The pickled binary sequence of the model takes up only about 1.3 MB memory.

2.5 Training Setups

We trained the model on the shuffled split train set with AdamW as optimizer. The hyperparameters mainly used were: batch size = 64, lr (learning rate) = 3×10^{-5}, $\beta_1 = 0.9$, $\beta_2 = 0.98$ and weight decay = 0.05. In case the gradient explodes, we used gradient clip = 1. As training proceeded, we recorded the cross entropy loss on both train and val sets and deep-copied models as backups. We especially picked out the one model backup which has the best validation loss and hence the lowest model perplexity (could be interpreted as the generalizability) to perform evaluation.

3 Results

Model The model we used to plot attention heatmaps and generate sample texts is chosen as the backup model at epoch = 19, which has lowest perplexity 44.70, training loss 3.221 and validation loss 3.800. The last model at epoch = 20 has perplexity 44.83, training loss 3.206 and validation loss.

Loss Curves The loss curves of the model is shown in **Fig. 1**. Both training and validation loss curves are plotted. We can see the model is almost fit to its best and the trend: soon after the last epoch the validation loss could rise and the model severely overfits the data then.

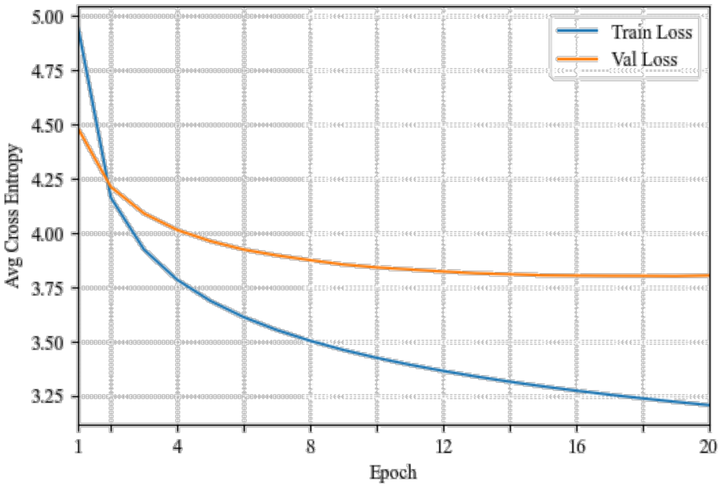

Fig. 1 ROC Curves of 6 Models

From the plot above, we can see that the model tends to overfitting the data: the validation loss drops more slowly than that of the train, and at some point (epoch = 19) the validation loss starts to rise while the train loss decreases steadily.

Attention Heatmaps The attention heatmaps of sample inputs are shown in **Fig 2-3**. We randomly chose to display the attention heatmap of some sample sentences out of the 2 layers. Lighter pixels indicates higher attention queried by the token paid to the key token. Here the spaces (encoded as Ġ by GPT-2 pre-tokenizer) are replaced by an underscore to visualize them. Specifically, we sampled a sentence from the data itself and feed it into our model, also utilize the output logits of the model to predict the next token, in which we used the greedy decoding strategy, the one with maximal logit was shown with its probability.

Generated Samples We further used the model to autoregressively generate next tokens. We still applied the greedy strategy to locate the one single word that pertains the maximal probability. The first one is a simple sentence "Hello world! This is Python! " that ends with a space. We ran the model 18 times over autoregression and obtained: "***Hello world! This is Python! I am appear to the crown,And I am advance***". This sentence may not make sence, however, since we trained our model on Shakespeare's plays, the grammars are not quite the same as modern English today, and the generated text does fit in the themes of his plays in his eras.

We further exploited the model's potential to generate texts depending on its original dataset. We sampled the sentence "Against him first: he's a very dog to the commonalty. " that ends with a space. The next line is "Consider you what services he has done for his country?" However, the model returns "***Against***

him first: he's a very dog to the commonalty. I have a king, I'll not beggar,And, as". This shows our model is not really overfitting and the attention module does work: the next token of generated "***I***" is not the same for our 2 samples. This sentence does not make sense: "***I have a king***", which could be resulted from insufficiency of data and oversimplified model structure.

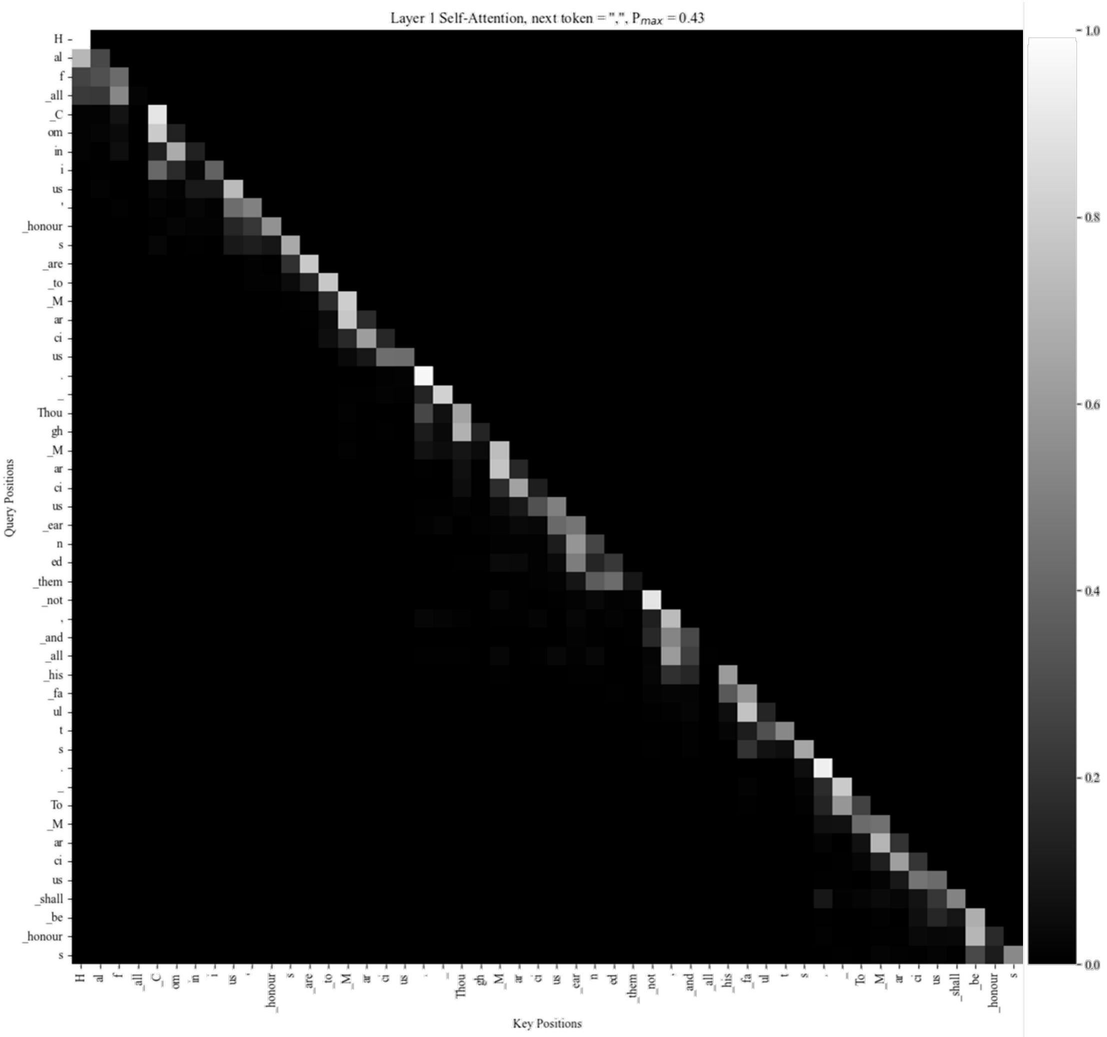

Fig. 2 Attention Heatmap of 50 Tokens (Showing Layer 1 Only)

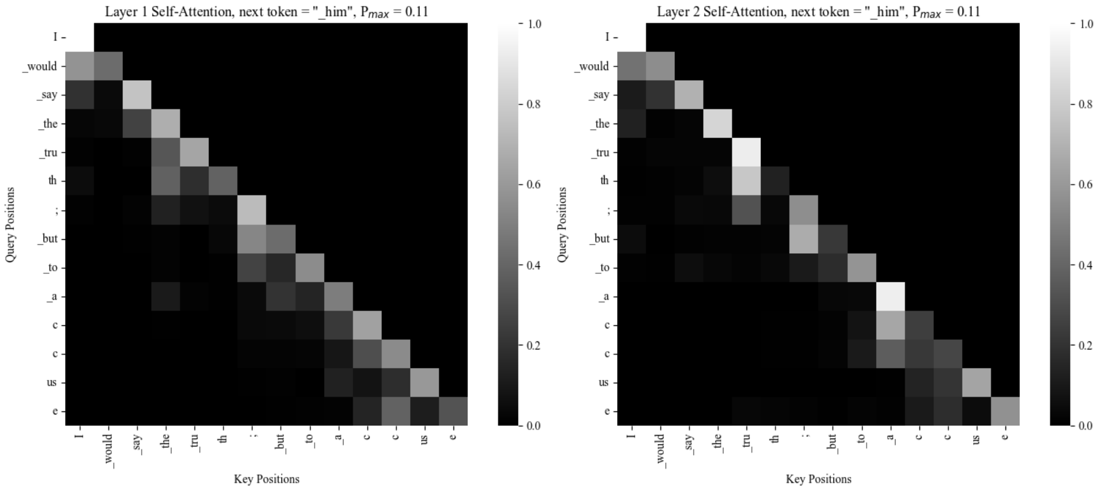

Fig. 3 Attention Heatmaps of a Sample Sentence: "*I would say the truth; but to accuse*"

4 Discussion

Here are some questions worth thinking about and we could discuss on their answers.

What patterns do you observe in the attention maps?

The attention maps typically show that most semantically holistic combinations of tokens makes the model concentrate more on, generally reflecting the local dependency of natural language.

In early stages, the model remains underfitting and attention maps could be noisy and more uniform (shown below in **Fig. 4**), but later the diagonal bands show up and structured spikes, triangles appear.

Interestingly, the vertical spikes and mutual attention show frequently on pairs or single punctuation, which shows the model gradually grasp the structure of the language through the punctuations.

Which hyperparameters (learning rate, context length, model size) had the greatest impact on stability?

We think the answer is Learning rate.

We trained this model under many learning rates and when it's too high, e.g. $> 3\times10^{-3}$, the model learns nothing and the validation loss increases steadily. If it's too low or with a higher weight decay coefficient in AdamW, the model learns nothing (probably extremely slow) and got stuck. You have to maintain it or reduce it carefully in a feasible interval.

Context length has lighter influence on the stability if it is in the reasonable range.

If context length is too short, which is unreasonable, the data leakage could easily take place and the model will exhibit very good performance on validation set, though meaningless. Longer context makes the model understands better, but not always greatly helpful.

Model size: since we are only asked to implement 2 Transformer blocks, the parameters related to the model size which we can fiddle with are hidden state dimension (in FFN) and the embedding dimension. These had little impact on stability, again, in the reasonable range. For example, I tried hidden_dim = 16, 32, 512 and all results (val loss) are almost the same.

How does attention evolve as the model trains over epochs?

We can see from these graphs that, the attention at first was chaotic (epoch = 1), no specific patterns were shown. However, as we trained it for longer time (epoch = 10), patterns show up, such as attention focuses on semantically holistic token combinations (e.g. the attention for the whole word “accuse” were high amongst pairs of its sublevel tokens, or “the” is with high attention on the following “truth” = “tru” + “th”). As we train it further (epoch = 19), some pairs of attention are enhanced and some are diminished, for example, “I” has lower attention on “the”, which makes sense because they usually don’t pair with each other in natural English.

What role do positional encodings play - could the model function without them?

No. Although we have causal masks, the positional encodings plays an important role in guaranteeing the model is not positional or permutation invariant: "You can jump like rabbits" is totally different from "Can rabbits jump like you" semantically. Causal masks are not sufficient to enable the model to catch structural information encoded in positions.

Reflect on runtime and memory footprint - where are the bottlenecks?

1. The attention map seems to be activated mainly along the diagonal line -- could be reduced. As embed_dim (vocab_size) d_e goes up, the computation could be of cost O(d_e^2).

2. Matrix multiplication in Q, K, V also costs computationally O(d_{model}^2).

3. Attention recomputes a lot when the model is performing autoregressive generation. Each time only 1 token is selected and generated.

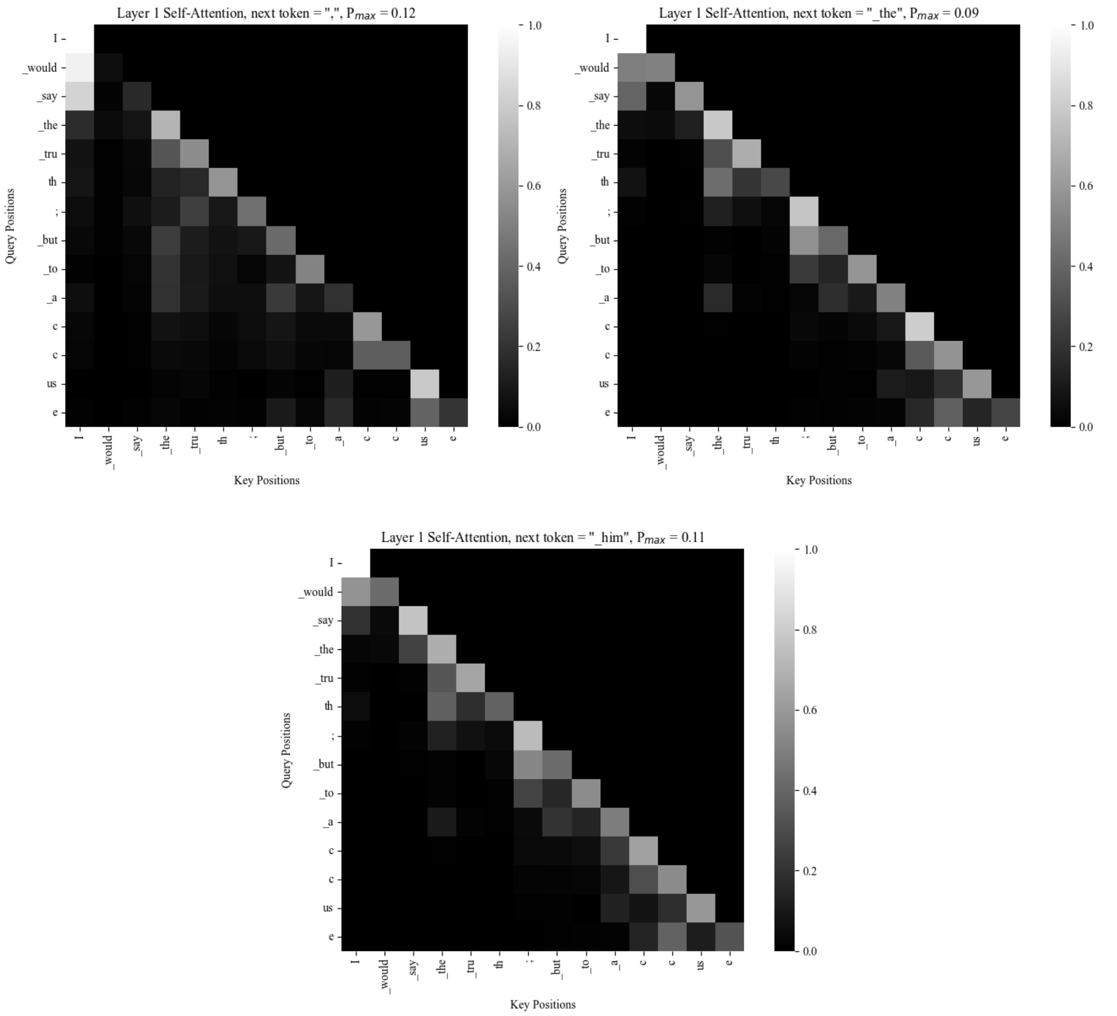

Fig. 4 Evolution of Attention (Epoch = 1, 10, 19)

AI Tool Usage Disclosure

AI Tools Used ChatGPT (OpenAI, GPT-4 Configuration)

AI Contribution The author asked ChatGPT to check whether the implementation is correct and how to improve it. ChatGPT also helped in writing only abstract and problem definition.

Personal Contribution The author has written almost all of this code and the report (from Dataset description to the end). No AI assistant like Copilot was used in code writing. Some parts are inspired by the online documentation and forum posts about PyTorch and other package libraries. Some parameters are discussed with friends of the author.