Implementation of a Small Transformer And Analysis Of Text Generation Sampling Techniques

Andrei Cozma
Department of Electrical Engineering and Computer Science
University Of Tennessee
Knoxville, United States
acozma@vols.utk.edu

Hunter Price
Department of Electrical Engineering and Computer Science
University Of Tennessee
Knoxville, United States
hprice7@vols.utk.edu

***Abstract*—In this work, we implement a simple transformer architecture that attempts achieve comparable results to fine-tuned larger models. The drawback of many state-of-the-art models is their significant size. Because of this, single individuals can have trouble training or fine-tuning these models. With a reduced size transformer, these individuals will be able to achieve results faster. Therefore, one of the core goals of this work is to learn how different sub-components part of a causal language modeling transformer architecture affects the quality of the generated text. We also quantitatively analyze the text generated by our transformer with different sampling methods with the Perplexity metric, BLEU metric, and Rouge metric. After training and analysis, we found that our model has room to improve as the metric scores indicate poor results. However, our model's size was significantly reduced from our benchmark model, GPT2.**

***Index Terms*—deep learning, transformer, gpt2, text generation, language modeling**

I. Introduction

In this work, we implement a simple transformer with the goal of achieving comparable results to fine-tuned larger models. Our goal was to train a small transformer model with comparable results to a GPT2 model with significantly fewer weights. The main drawback of larger models is that they take too long to train and are often not worth hyper-parameter tuning because it will take too long. Our proposed model accomplished this by having a reduced number of weights allowing us to experiment with the hyper-parameters. We also seek to analyze the impact of different sampling techniques. Specifically, we use Greedy Search, Random Sampling with Temperature, Top-K Sampling, and Top-P sampling. To analyze the results of these sampling techniques, we get quantitative metrics using three text generation metrics. Specifically, we use *Perplexity*, *Rouge*, and *BLEU*.

II. Previous Work

In previous works, there have been large advancements in text generation in the form of transformers. The first proposal of this architecture was in the paper Attention Is All You Need [1]. Here they introduced the transformer architecture, which achieved state-of-the-art results in comparison to other architectures at the time. One of the first large-scale models proposed by OpenAI was GPT2 [2]. We used this model as our benchmark and starting point for this project. This few-shot learning model was massive for its time but proved to generate amazing results. OpenAI then followed this model with its third version GPT3 [3]. This model was one of the current standards for text generation until it was surpassed by other larger models such as T5 [4]. Another interesting transformer text generation model based off the GPT models is titled Codex [5]. This model performs extremely well and is even used today to generate code for developers.

We cite these papers as inspiration for our proposed simple transformer. We analyzed these network architectures and created our own smaller version.

III. Technical Approach

In this work, we aimed to answer two core questions. Is it possible to implement a small transformer that can be trained within a reasonable amount of time? And how do different sampling techniques impact the generated text?

We create a small transformer with a language modeling head to generate text similar to the dataset it was trained on. In the previous work, many of the well-performing models have a significant number of weights that no single personal computer can train within a reasonable amount of time. We implemented a language modeling transformer with far fewer weights that can achieve decent results within just a few hours on a personal PC.

We took a structured approach to get an accurate indication of how the sampling technique impacts text generation performance. First, we used four separate sampling techniques. The first is a naive greedy approach that chooses the token based on the highest probability. The second is random sampling with a temperature approach that can be made more or less random with its temperature parameter. The third is top-k sampling which sorts the probabilities and zeroes out all probabilities after the K'th token. Then chooses similarly to random sampling based on the temperature. The final is top-p sampling, that cuts off the probabilities once it surpasses the p parameter and then continues similarly to top-k and random sampling. We then chose three metrics to quantitatively measure the generated text: Perplexity [6], Rouge score [7], and *BLEU* score [8].

With these sampling techniques and metrics, we fed our transformer several prompts to generate predictions. Then we used each sampling technique on the produced logits. With each of these logits, we get the scores for each metric.

IV. Dataset and Implementation

Several datasets have been collected and used throughout the implementation process. These included *Shakespeare Plays Dataset*, *ACL Titles and Abstract*, *Rap Music Dataset*, *WikiText-2* and *WikiText-103*. For the final training and evaluation of a model we used the *WikiText-2* dataset [9].

The *WikiText-2* dataset was originally introduced in 2016 in ”Pointer Sentinel Mixture Models” [9]. The corpus is made up of *2,088,628* sequences (600 articles), validation *217,646* sequences (60 articles), and the test *245,569* sequences (60 articles).

The dataset is preprocessed for word-level tokenization while retaining numbers, case, and punctuation. This is implemented in two layers for the original embeddings and positional embeddings.

The implementation of the transformer block used in the model is as follows. It consists of a feed-forward network with a configurable number of layers and units using *ReLU* activation. The inputs use a causal attention mask which is passed to the Keras *MultiHeadAttention* layer. After the attention scores are calculated, dropout is applied to the output of the attention layer, and the inputs with attention are passed through a normalization layer. Then, the normalized inputs are fed into the feed-forward network, and dropout is applied again. Finally, the output is the normalized sum of the outputs from the first normalization layer and the output of the feed-forward network.

The inputs to the transformer block are sequences consisting of the input and the target being shifted by one token. These are passed into a custom layer with the following layers: a positional embedding layer, the transformer block, and the output being a dense layer. The maximum vocabulary size is defined as a parameter that serves as the number of units for the output layer. *Sparse Categorical Cross-Entropy* is the loss function used, together with a *Warm-up Scheduler* in control of the learning rate for the *Adam optimizer*. The Warm-up Scheduler is used to help the model against early over-fitting by starting with a very small learning rate and gradually increasing it up to a predefined number of steps. After this, the learning rate gradually decreases until the specified number of maximum training epochs.

Finally, a helper class was created to facilitate model creation based on given configurations defined as JSON. Mainly, this class reads in the dataset and sets up the model for training and generation. The class also handles saving and loading models from a model directory on disk. This allows easily loading pre-trained versions of our model, which can be shared between multiple individuals, allowing them to further fine-tune the model on their own datasets and text files and simply generate sequences from the saved model.

A. Parameter Choices

Our implementation makes available a number of parameter choices for the model and training configurations.

The model configuration parameters include vocabulary size (*VOCAB_SZ*), maximum sequence length (*MAX_LEN*), the number of attention heads (*ATT_HEADS*), the dimensionality of the embedding layer (*DIM_EMB*), the dimensionality of the fully connected network in the transformer (*DIM_FFN*), and the number of warm-up steps for the learning rate scheduler (*WARMUP_STEPS*).

The training and fine-tuning configuration parameters include the batch size, number of epochs, as well as a list field for the dataset files to be used for training and fine-tuning.

The final model was trained with the following hyper-parameter choices:

- VOCAB_SZ: 30000
- MAX_LEN: 300
- ATT_HEADS: 40
- DIM_EMB: 256
- NUM_LAYERS: 2
- DIM_FFN: 2048
- WARMUP_STEPS: 15000
- BATCH_SIZE: 32
- EPOCHS: 1000

This configuration resulted in a model with *27,035,440* Trainable Parameters.

B. Training Time & Hardware

The language generation model was trained on an *NVIDIA RTX 3070* GPU. One of the most significant constraining factor in deep learning tasks, especially so for language generation models, is the amount of memory (*VRAM*) of the card.

While the RTX 3070 only has 8 GB of VRAM, much higher capacities are necessary for training very large networks with potentially hundreds of millions or even billions of trainable parameters. Therefore it is an important task to be able to optimize the number of parameters in such a model to be able to achieve the best amount of performance.

The final model was trained in total of 8 hours.

V. Experiments and Results Analysis

In this work, as described above, we used a custom scheduler with warm-up. In Figure 1, we can see the learning rate versus epochs while training. We can see that the learning rate greatly increases over the first 15 epochs, then sharply decreases and follows a logarithmic curve reaching a learning rate of approximately 0.0002 before the training ends.

As mentioned earlier, we use a loss function of *Sparse Categorical Cross-entropy*. In Figure 2, we can see the loss versus epochs while training. The loss curve first starts very high, then after the first epoch, sharply corrects to a value of approximately 1.5, then slowly decreases over the next 80 epochs. This is because transformers typically do not need many epochs to fit the amount of data that we are using decently.

We selected 5 random prompts from our testing dataset. We took the following sentences for references for both the *Rouge* and *BLEU* metrics for each of those prompts. We fed each randomly selected prompts through the trained model

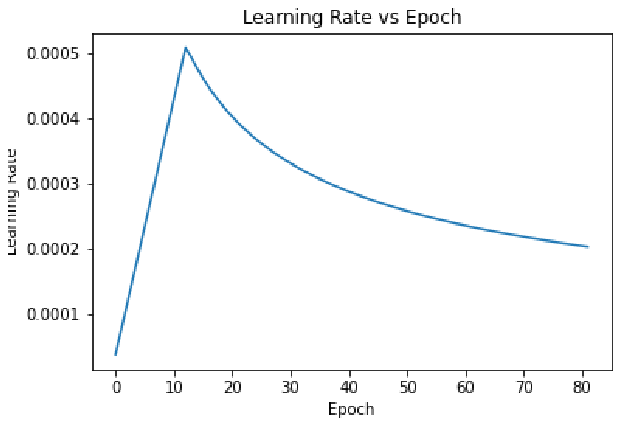

Fig. 1. Learning Rate vs Epoch

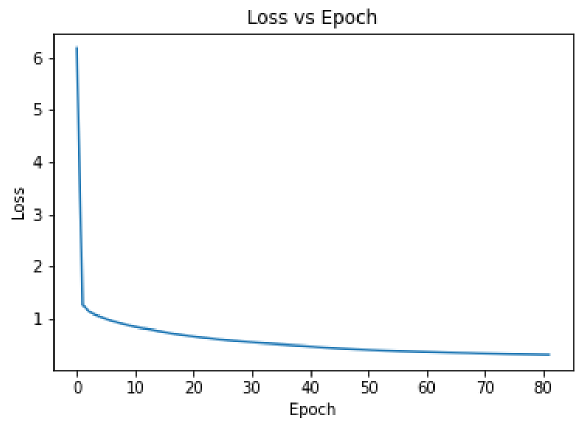

Fig. 2. Loss vs Epoch

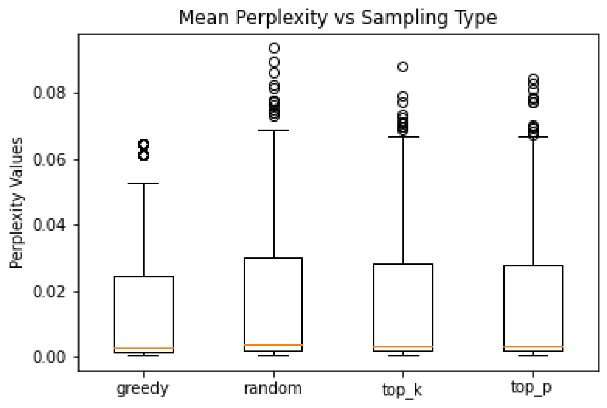

Fig. 3. Perplexity Scores For Each Sampling Technique

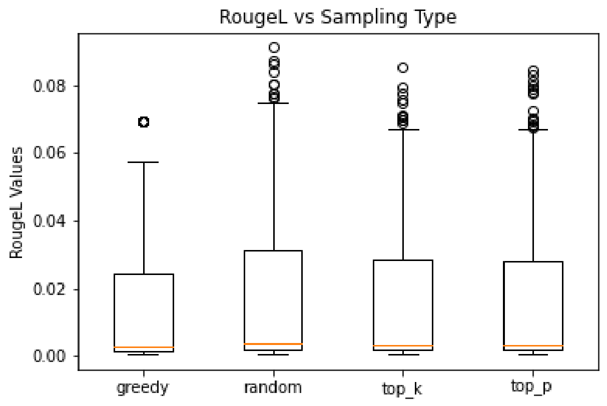

Fig. 4. RougeL For Each Sampling Technique

to get logits. Next, we fed the logits through each sampling technique to get predictions. With these predictions, we got calculated the text generation metrics.

In Figure 3, we can see the distribution of the *Perplexity* metric for each sampling technique. In natural language processing, a high Perplexity means the predicted sentences are more stable. The mean of these values for all sampling methods is slightly above 0. This alludes to the fact that our transformer is consistently outputting somewhat incoherent results. However, we can see that greedy has the lowest maximum Perplexity out of the 4 sampling techniques. The other three sampling techniques are fairly consistent with their maximum values, with random sampling having the highest score.

In Figure 4, we can see the distribution of the *RougeL* metric for each sampling technique. RougeL refers to the longest common subsequence-based statistics. Similarly to Perplexity, higher scores are better. We can see the sampling techniques have very comparable results to Perplexity. Each of the sampling techniques means is close to zero. And their maximums reach similar maximums as Perplexity.

In Figure 5, we can see the distribution of the *Brevity Penalty* metric produced by the BLEU scores for each sampling technique. The brevity penalty penalizes generated translations that are not long enough as compared to their closest reference sentence lengths. We can see that greedy sampling has a higher mean brevity penalty than all other sampling techniques. This means that the greedy sampling technique consistently produces far shorter text compared to the other techniques.

In Figure 6, we can see the distribution of the *BLEU* metrics for each sampling technique. BLEU is a metric that evaluates the quality of the generated text. It has been shown to be comparable to human judgment. We can see that all of our results give a BLEU score of 0. This could lead to two conclusions. The first possible conclusion is that our transformers produced text is of very ill-quality and not comparable to actual human-written text. The second possible conclusion is that we did not properly configure and compute the BLEU score.

VI. Conclusion

In this project, we successfully implemented a small version of a transformer with a language modeling head. This model

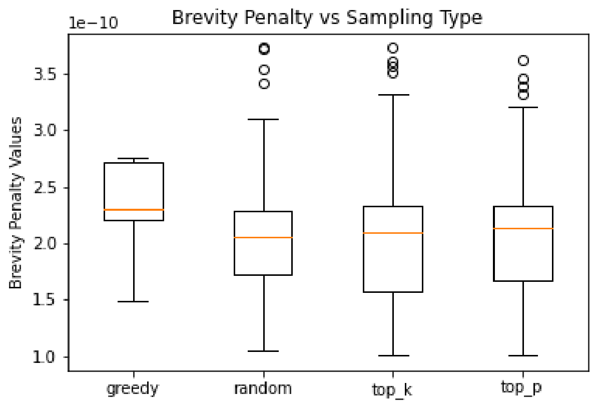

Fig. 5. Brevity Penalty For Each Sampling Technique

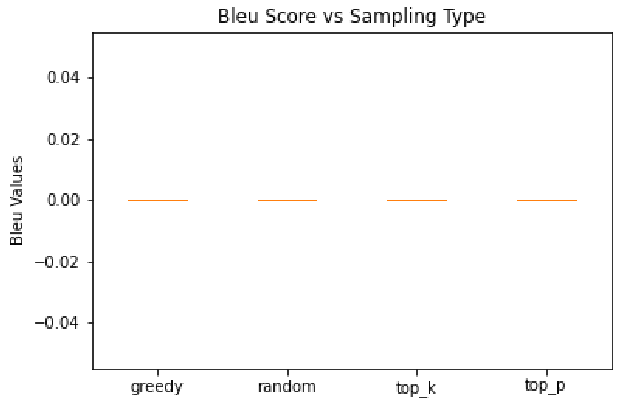

Fig. 6. BLEU Score For Each Sampling Technique

has much fewer weights than other well-performing models with similar architectures. We also trained on far smaller amounts of data. Our metric results on a generated text show that our model has far more room to improve. We also found after qualitative analysis that top-p seems to produce higher-quality sentences that are more readable more consistently.

We have learned how a transformer works on a low level. We have learned the importance of tokenization and that improvement in this matter leads to far better results. We understand why large models need so much computation, as the training time for our models took far longer than expected. Finally, we have also learned that sampling techniques in the Tensorflow library would benefit from further documentation and examples.

References

[1] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin, "Attention is all you need," 2017. [Online]. Available: https://arxiv.org/abs/1706.03762

[2] A. Radford, J. Wu, R. Child, D. Luan, D. Amodei, and I. Sutskever, "Language models are unsupervised multitask learners," 2019.

[3] T. B. Brown, B. Mann, N. Ryder, M. Subbiah, J. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, S. Agarwal, A. Herbert-Voss, G. Krueger, T. Henighan, R. Child, A. Ramesh, D. M. Ziegler, J. Wu, C. Winter, C. Hesse, M. Chen, E. Sigler, M. Litwin, S. Gray, B. Chess, J. Clark, C. Berner, S. McCandlish, A. Radford, I. Sutskever, and D. Amodei, "Language models are few-shot learners," 2020. [Online]. Available: https://arxiv.org/abs/2005.14165

[4] C. Raffel, N. Shazeer, A. Roberts, K. Lee, S. Narang, M. Matena, Y. Zhou, W. Li, and P. J. Liu, "Exploring the limits of transfer learning with a unified text-to-text transformer," *Journal of Machine Learning Research*, vol. 21, no. 140, pp. 1–67, 2020. [Online]. Available: http://jmlr.org/papers/v21/20-074.html

[5] M. Chen, J. Tworek, H. Jun, Q. Yuan, H. P. d. O. Pinto, J. Kaplan, H. Edwards, Y. Burda, N. Joseph, G. Brockman, A. Ray, R. Puri, G. Krueger, M. Petrov, H. Khlaaf, G. Sastry, P. Mishkin, B. Chan, S. Gray, N. Ryder, M. Pavlov, A. Power, L. Kaiser, M. Bavarian, C. Winter, P. Tillet, F. P. Such, D. Cummings, M. Plappert, F. Chantzis, E. Barnes, A. Herbert-Voss, W. H. Guss, A. Nichol, A. Paino, N. Tezak, J. Tang, I. Babuschkin, S. Balaji, S. Jain, W. Saunders, C. Hesse, A. N. Carr, J. Leike, J. Achiam, V. Misra, E. Morikawa, A. Radford, M. Knight, M. Brundage, M. Murati, K. Mayer, P. Welinder, B. McGrew, D. Amodei, S. McCandlish, I. Sutskever, and W. Zaremba, "Evaluating large language models trained on code," 2021. [Online]. Available: https://arxiv.org/abs/2107.03374

[6] F. Jelinek, R. L. Mercer, L. R. Bahl, and J. K. Baker, "Perplexity—a measure of the difficulty of speech recognition tasks," *The Journal of the Acoustical Society of America*, vol. 62, no. S1, pp. S63–S63, 1977.

[7] C.-Y. Lin, "Rouge: A package for automatic evaluation of summaries," in *Text summarization branches out*, 2004, pp. 74–81.

[8] K. Papineni, S. Roukos, T. Ward, and W.-J. Zhu, "Bleu: a method for automatic evaluation of machine translation," in *Proceedings of the 40th annual meeting of the Association for Computational Linguistics*, 2002, pp. 311–318.

[9] S. Merity, C. Xiong, J. Bradbury, and R. Socher, "Pointer sentinel mixture models," 2016. [Online]. Available: https://arxiv.org/abs/1609.07843

Appendix A
Code Design

Our primary source code is organized within the *src* directory of the repository. The first file in this directory is *Config.py*. This file contains data classes that hold configuration parameters for our Transformer to use while training, including where the training dataset files are located, the batch size, number of epochs, and the overall transformer hyper-parameters.

The next file in the directory is *Dataset.py* which holds a class named Dataset which extends the TensorFlow *TextLineDataset* class. This allows us to feed our datasets into this class such that it will be more easily able to interface with our Tensorflow-based model.

The *Metrics.py* file contains class assignments from the Huggingface datasets library to calculate the BLEU, Rouge, and Perplexity metrics. This class also contains helper functions to automate the collection of metrics for given prompts.

Next, *Sampling.py*, contains a class named Sampling, which implements the four sampling techniques referenced throughout this paper. It also contains a helper function to generate text with all sampling techniques.

The next file is *tokenize_utils.py*, which contains a small library to tokenize text into sentences required by the metric functions.

Tokenizer.py containing the main Tokenizer class that inherits the Tensorflow *TextVectorization* class. This serves as our core tokenization functionality for our model input.

The *top_.py* file, contains a modified implementation of Top-P sampling taken from the Tensorflow library. We had to modify this function for it to work with our architecture.

The next file is *Utils.py* holds a function to generate callbacks for our Tensorflow model. Next, we have *WarmupScheduler.py*, which contains a *WarmupScheduler* class that inherits from the Tensorflow *LearningRateSchedule* class. This class implements a learning rate scheduler that performs warmup steps before decreasing the learning rate.

The next section of functions and classes pertains to the core model components. *SimpleTransformer.py* contains the core transformer architecture, which inherits a Tensorflow Layer. The next file is *TokenAndPositionEmbedding.py* which includes a Layer class to take in the word embeddings. The next file is *SimpleTransformerBlock.py* which, as the name implies, creates a TensorFlow layer class that serves as a block of the bare-bones transformer, which can be implemented and used in a modular fashion. Finally, we have the file *SimpleTransformerLMHead.py*, which serves as to facilitate using the transformer model for causal language modeling (generation). More specifically, it abstracts all of the loading, saving, training, fine-tuning, and generation functionality.

We then have a folder named tests which are where the various functionalities have been tested iteratively. In the root directory, we have 4 Jupyter notebooks. The main training notebook is titled *transformer_simple_warmup.ipynb*. This file trains and saves our transformer model with warm-up steps. The *load_model.ipynb* file contains code to quickly load in and test the generation of the models. We then have *get_metrics.ipynb*, which retrieves metrics from the text generated from the model. Next *read_logs.ipynb* reads the Tensorboard logs to make custom plots. Finally, we have *analyze_metrics.ipynb*, which reads the metrics and creates the plots.

Appendix B
Workload Distribution

A. Andrei Cozma

Andrei first looked into and collected text datasets of different sizes and characteristics to use for model training. He then experimented with different ways of tokenizing the datasets, including the pre-trained tokenizers from HuggingFace. Next, he implemented various components for the base transformer model based on experimentations with different types of techniques, parameters, and configurations that have shown desirable results in different areas. Andrei then integrated these components together to be used for causal language modeling (generation). These were initially going to be implemented on top of the HuggingFace API as an additional model component. However, we ended up settling away from implementing our model on top of Huggingface because of the complexity and time constraints imposed by learning the modularized API from scratch.

B. Hunter Price

Initially worked on an implementation of our transformer within the Huggingface API. I forked the Huggingface transformers library as they had a standard process to add a transformer for this use case. We later found this approach to be extremely complicated and could not use this approach due to the time constraint. I also worked on the implementation of the sampling techniques: greedy, sampling, top-k, and top-p. I attempted to use beam-search; however, this proved too complicated and undocumented to do within the allotted time. I implemented the code to collect the metrics for all generated text. I created the graphs and charts associated with the metrics and sampling.