TokenFormer

A Scalable Architecture For Transformers

Venkataram Edavamadathil Sivaram

Introduction

Transformers are greatly successful in solving various kinds of sequential problems, notably natural language modeling using next-word prediction. As such, they are prevalent in the models used to automate numerous prompt-based tasks such as programming, searching the web, and writing prose. Likewise, many corporations have begun devoting large amounts of time, energy and other resources into training increasingly larger transformer-based models. The scaling laws embedded in deep learning guarantee that expanding the size (number of parameters) of the model enables it to improve learning, provided sufficient data and time. Often these larger (foundation) models are trained from scratch, without reusing parameters from previous, smaller models. However, this is wasteful in terms of both time and energy. TokenFormer [Wang et al., 2024], the subject of this project, is an architecture that aims to resolve this by designing an attention-based transformer block which can seamlessly scale in size. With this architecture, one can steadily increase the number of parameters in the model while expecting to achieve higher performance. In this report, we detail the architectural considerations made by the authors, and demonstrate that it upholds the goals it was set to solve.

Note: *Details of the code are at the end of the report.*

Model

Overview We first provide a high level overview of the contributions presented in the TokenFormer [Wang et al., 2024] work in Figure 1. The authors of the work argue that the primary bottleneck for scalability in transformers are the linear projection layers. Their solution was to replace all such layers, found in the QKV projection and feed-forward network steps, with so called *"Parameter-attention"* (Pattention) layers that are purely attention-based. As will soon be revealed, modifying the primary token-parameter interactions in this way naturally enables scalability.

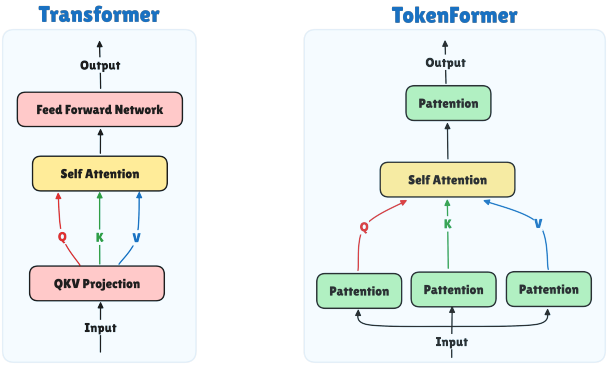

Figure 1: Comparing blocks from the original transformers [Vaswani et al., 2023] work and TokenFormer [Wang et al., 2024].

Parameter attention Abstractly, each Pattention layer operates similarly to self-attention, with the same query (Q), key (K), and value (V) components. The difference from self-attention is the idea that the keys and values in each Pattention layer are parameters of the model that are learned throughout optimization. Mathematically, the layer evaluates the following:

$$\textsc{Pattention}(X; K, V) = \Theta(X \cdot K^{T}) \cdot V$$

Here, $X \in \mathbb{R}^{T \times d}$ is the set of input tokens with sequence length T, $K \in \mathbb{R}^{n \times d_1}$ is the set of key for the layer, and $V \in \mathbb{R}^{n \times d_2}$ is the set of values for the layer, where n is the number of parameter tokens in the layer. Note that the result of this layer is a vector in $\mathbb{R}^{T \times d_2}$ – for our purposes $d_1 = d_2$ is set to the embedding size of each token. Lastly, $\Theta(\cdot)$ is the softmax activation[1] function. Figure 2 visually summarizes the composition of this layer.

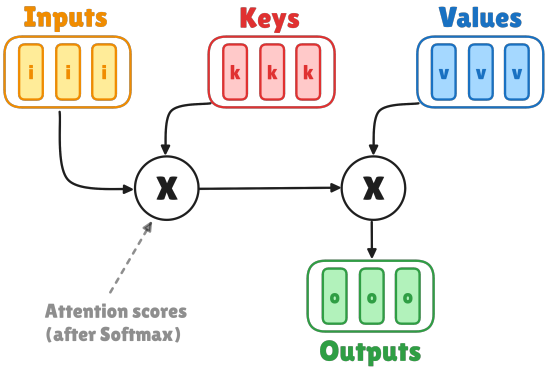

Figure 2: Inner workings of the parameter-attention layers in a TokenFormer block.

TokenFormer block The complete description of a TokenFormer block can then be written as follows:

$$
\begin{aligned}
X &= \textsc{Normalization}(X) \\
Q &= \textsc{Pattention}(X; K_Q, V_Q) \\
K &= \textsc{Pattention}(X; K_K, V_K) \\
V &= \textsc{Pattention}(X; K_V, V_V) \\
X_{sdpa} &= \textsc{Sdpa}(Q, K, V; h) \\
X_{inter} &= X + \textsc{Pattention}(X_{sdpa}; K_{proj}, V_{proj}) \\
X_{out} &= X_{inter} + \textsc{Pattention}(X_{inter}; K_{ffn}, V_{ffn})
\end{aligned}
$$

Each of the K_* and V_* elements indicated above refer to the keys and parameters of the corresponding Pattention layers. These typically use the same number of parameters n; the only exceptions are K_{ffn} and V_{ffn}, which use $4n$ parameters to mimic the wider layers in transformers. The input vector undergoes layer normalization before any projection steps ensue, and these normalizations are parameter-free. Additionally, skip (residual) connections are added in the final projection layers, to improve gradient flow. Finally, $\textsc{Sdpa}(\cdot)$ refers to the standard (causal) scaled-dot-product-attention method, which applies the softmax activation. In our implementation, we apply mutli-headed attention with h heads only for this self-attention method[2].

Scaling parameters Statically, each Pattention layer consists of n token parameters for both the keys and values. To scale these parameters, one can simply append more tokens (of dimension d_1 and d_2) to the keys and values. Note that this does not change the input and output dimensions of the layer. Figure 3 demonstrates how this can be achieved.

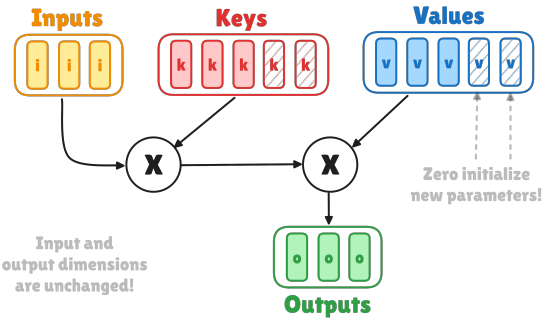

Figure 3: Scaling the number of parameters in a Pattention layer.

To ensure that training proceeds smoothly when growing parameters, the new parameters are zero-initialized. This guarantees that each layers output prior to and right after scaling the parameters is precisely equal.

[1] In the original TokenFormer work [Wang et al., 2024], the authors proposed a modified softmax function using Gelu for improved gradient signals. However, we could not reproduce these benefits (even when using their code), and thus regressed to the softmax activation.

[2] When discussing the general block structure, the TokenFormer paper abstracts away multi-head attention, mentioning it only briefly. It was unclear to us how exactly to implement this for Pattention, and therefore we have omitted it instead.

Datasets

In the following experiments, we evaluate the TokenFormer model on language modeling tasks. Initially, we aimed to gauge its performance in multi-lingual modeling ability, and chose the CulturaX [Nguyen et al., 2023] dataset. This a is large dataset comprising of numerous languages, which has undergone thorough cleaning, making it suitable for our desired tasks. Ultimately, we made little usage of the multi-lingual datasets, so we primarily use it for English-based tasks.

Note: *It should be noted that we are using the same tokenizer as GPT2, so the vocabulary set is the same.*

Results

GPT2 benchmark Our first experiment directly compares the transformer and TokenFormer architectures. To this end, we based both baseline models on the GPT2 architecture. Table 1 shows the configurations for the models[3] corresponding to both techniques.

	Transformer	TokenFormer
Context size	1024	1024
Embedding size	768	768
Parameter count	-	384
Layers	12	12
Heads	12	12 (self-attention)
Total parameters	163M	134M

Table 1: Model configurations to match the GPT2 architecture.

Both models are trained on identical subsets of the CulturaX English dataset. Training is conducted for 1000 epochs; during each epoch, a batch of 8 randomly selected sequences are sampled and trained on for 100 steps. Figure 4 shows the progress of both models as training progresses, measured by the perplexity on training data[4].

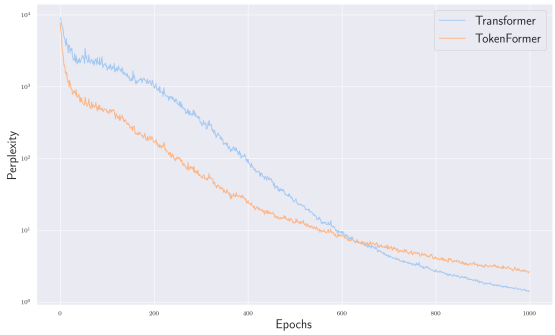

Figure 4: Evolution of training perplexity over epochs.

We observe that the TokenFormer model quickly outperforms the transformer model and maintains its lead throughout the earlier stages of training. Towards the end, the transformer model catches up, but ultimately both models achieve similar performance.

Online scaling In this next experiment, we test the scalability of the TokenFormer model. Specifically, we aim to demonstrate that the TokenFormer architecture is capable of scaling online, during the course of training. Table 2 summarizes the configurations used for this experiment. Each model variant is based on the same GPT2 architecture, but starts with a smaller number of parameters. Every so often (growth frequency), we grow the parameters of each model by a fixed amount (growth amount). For this experiment, we extend the number of epochs to 2500.

	G64	G128	G256
Initial parameters	64	128	256
Growth amount	64	128	256
Growth frequency	125	250	500

Table 2: Model configurations for the scalability experiment.

[3]The parameter counts come from iterating over PyTorch's `parameters()` method and summing the number of elements. We could not quite pin down the discrepancy between the parameter counts, but we suspect it may be due to non-parameteric layer normalization.

[4]While we understand that measuring perplexity on training data is suboptimal compared to measuring it on validation data, we did not have sufficient time to conduct a follow up experiment.

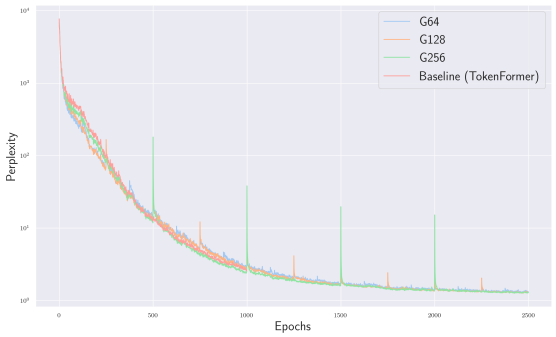

Figure 5: Performance of models with varying growth schedules.

In general, across all growth configurations, we see that the performance is comparable to baseline TokenFormer in the fixed window up to 1000 epochs. Beyond that we see that all variants are able to outperform the baseline model, suggesting that there is indeed a benefit in this scaling approach. The occasional spikes on the graph indicate growth moments for the different configurations[5].

Masking parameters We also performed an experiment which explores a potential extension of the TokenFormer architecture to adapt to multiple tasks in a semantic manner. Concretely, in the task of multi-lingual language modeling, our idea is to mask certain parameters for a particular language. For example, in learning English and French, we would disable the latter half of the parameters (e.g. by multiplying those halves of K and V by zero) when training and evaluating on English, and the first half for French. Intuitively, this is akin to allocating distinct subsets of the parameters for different languages, which would theoretically enable continual learning.

We used a similar architecture to GPT2, but extended the number of parameters to 3×512. Every consecutive set of 512 parameters would then be allocated (masked ON) for the three corresponding languages, English, French, and Spanish. Each language is trained on for 500 epochs in sequence, and then never again. Figure 6 shows the results of this experiment by displaying the perplexities for all languages over the course of this training program.

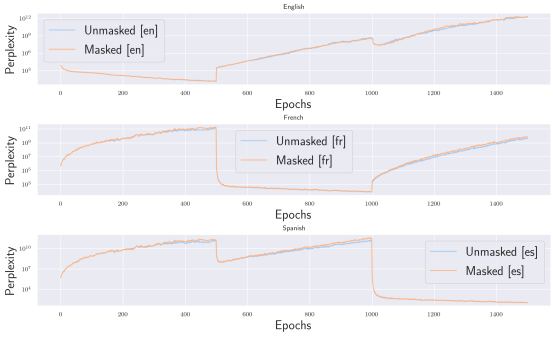

Figure 6: Progressive performance of the masked model over the course of learning multiple languages.

Disappointingly, this method of masking did not improve performance in any significant measure. We suspect that this is due to the fact that token and position embeddings are shared amongst languages, and are effectively relearned each time the model trains on the next language. It may fare better to use learnable non-binary masks for each language, or to augment PATTENTION with an additional set of task-specific token parameters, but this is merely speculation.

Conclusion

The results from the aforementioned experiments are positive, and they indicate that the TokenFormer architecture is indeed scalable while remaining performant. The implications of this are interesting, as this feature aligns well with the neural scaling laws of deep learning.

Code

Attribution Our implementation of TokenFormer, as well as the experiments and their configurations, are available on GitHub[6]. None of the code was generated

[5] Each time we grow the model we reset the optimizer; we found that this did significantly better than leaving it unattended. As a result, however, the initial steps of the optimizer will cause a divergence in training loss (and perplexity). The stability of training irrespective of this may be attributed to gradient clipping.

[6] `https://github.com/iveevi/tokenformer`

with AI, but there are a few sources with due attribution as follows:

- Andrej Karpathy's video lecture, *Let's reproduce GPT-2 (124M)*, and the corresponding code repository[7]. This was the basis for our GPT2 implementation, and we mirrored this for the TokenFormer implementation.
- The TokenFormer source code repository[8], though none of its code (to the best of our memory) is remaining. Frankly, their code is too indecipherable to replicate in an understandable manner.
- The TikToken tokeniser[9], which provides a better set of tokens that simplly using ASCII or UTF-8. We use the `gpt2` encoder to match the original GPT2 architecture.
- The dataset loader from Hugging Face[10], which makes loading the CulturaX dataset very convenient. We use data streaming so that the entire dataset is not loaded (we only train on a few shards).

Usage Our experiment configurations are files in the `configs` directory. To get started, simply run the training script `python train.py`, which will prompt the user to select a particular configuration. The configuration files are in the INI format, but allow for inherited configurations to conveniently create modified experiments.

References

[Nguyen et al., 2023] Nguyen, T., Nguyen, C. V., Lai, V. D., Man, H., Ngo, N. T., Dernoncourt, F., Rossi, R. A., and Nguyen, T. H. (2023). Culturax: A cleaned, enormous, and multilingual dataset for large language models in 167 languages.

[Vaswani et al., 2023] Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, L., and Polosukhin, I. (2023). Attention is all you need.

[Wang et al., 2024] Wang, H., Fan, Y., Naeem, M. F., Xian, Y., Lenssen, J. E., Wang, L., Tombari, F., and Schiele, B. (2024). Tokenformer: Rethinking transformer scaling with tokenized model parameters.

[7] `https://github.com/karpathy/nanoGPT/tree/master`
[8] `https://github.com/Haiyang-W/TokenFormer`
[9] `https://github.com/openai/tiktoken`
[10] `https://huggingface.co/docs/datasets/en/index`