Poor Man's GPT-3

Eric Chan *yee96@cs.uw.edu*
Kai-Wei Chang *kwchang2@cs.uw.edu*
Shivam Singhal *shivam42@cs.uw.edu*
Andrew Wei *nowei@cs.uw.edu*

https://github.com/kaiweic/Poor-man-s-GPT-3

1 The Problem and its Importance

Transformers were first introduced in 2017 [2] as an improvement to the RNNs proposed in the past. RNNs, such as LSTMs and GRUs, rely on hidden states to retain the information they saw/output previously and uses a recurrent structure where the output of the model is passed in as the input of the model for the next word of the same sentence. Transformers, on the other hand, do not have a recurrent structure and can take multiple words as an input in a single pass. This improves the training efficiency compared to RNNs. In addition, Transformers have attention heads that allow the model to focus on the more important words. Overall, Transformers exhibit better performance and efficiency during training.

One main drawback of Transformers was that it could only learn the context within the sequence lengths for which it was trained on. This means that for text longer than the sequence length, it would have to be broken apart and the context between sequences would be lost. A solution to this problem is TransformerXL, which augmented the Transformer so that it could handle extra long contexts through the use of a memory of previous sequences.

In this paper, we will review the process of how to adapt an existing Transformer implementation into TransformerXL. In particular, we focus on implementing the following:

1. A memory that allows the attention heads of TransformerXL to look back at a longer history of words
2. Relative positional encoding that keeps the positional information consistent and allows us to use the history, and
3. Embedding dropouts that drops a portion of both the word and positional encoding.

2 Related Work

The TransformerXL model proposed by Dai, et al. [1] introduced the concept of memories, which help Transformers retain more context about past sequences. Each Transformer Block has a limited sized memory, where $\mathbf{m}_i$ is the memory for the i-th Transformer Block. Each memory block is composed of the current memory, initially starting out empty, and the previous hidden state, $\mathbf{h}_i \in \mathbb{R}^{L \times d}$ where L is the sequence length and d is the embedding dimension, passed into the block previously. The memory is capped at a maximum length, M, which is a hyperparameter. When the memory is full, we have $\mathbf{m}_i \in \mathbb{R}^{M \times d}$. Thus, each update of the memory looks like:

$$\mathbf{m}_i = [\mathbf{m}_i \circ \mathrm{SG}(\mathbf{h}_i)][-M:] \tag{1}$$

where SG represents the stop-gradient, $\circ$ represents the concatenation operation where it concatenates along the first dimension (i.e. along the sequence length), and $[-M:]$ represents selecting out up to the last M memory components. We use a stop-gradient so that we do not further accumulate the gradients associated with past hidden states.

To handle this extended context from our memories, Dai, et al. replaced the concept of absolute positional embedding with relative positional embedding. This is due to how positional embeddings were handled in the original Transfomer paper, where each sequence shared the same positional encoding and the model could no longer distinguish between current and previous sequences. To remedy this, relative positional encodings were suggested. Relative positional encoding is defined as $\mathbf{R} \in \mathbb{R}^{L_{max} \times d}$ where the L_{max} is the memory length plus the sequence length and d is still the embedding dimension. On a high level, relative positional encoding represents relative distances through the encoding of edges. This means that row i of $\mathbf{R}$ represents the encoding of the relative distance between two positions that are i words apart.

Since we are using a new type of positional encoding, we must update the equations used for calculating attention. The original attention score, $\mathbf{A}$, as presented in [2] is computed as

follows:

$$\begin{aligned}\mathbf{A}^{\text{abs}}_{i,j} = &\underbrace{\mathbf{E}_{x_i}^\top \mathbf{W}_q^\top \mathbf{W}_k \mathbf{E}_{x_j}}_{\text{(a)}} + \underbrace{\mathbf{E}_{x_i}^\top \mathbf{W}_q^\top \mathbf{W}_k \mathbf{U}_j}_{\text{(b)}} \\ &+ \underbrace{\mathbf{U}_i^\top \mathbf{W}_q^\top \mathbf{W}_k \mathbf{E}_{x_j}}_{\text{(c)}} + \underbrace{\mathbf{U}_i^\top \mathbf{W}_q^\top \mathbf{W}_k \mathbf{U}_j}_{\text{(d)}}\end{aligned} \tag{2}$$

where $\mathbf{E}$ is the word embedding, $\mathbf{W}$ are the learned query and key matrices, and $\mathbf{U}$ is the absolute positional encoding. (a) represents the content-based addressing, (b) represents the content-dependent location bias, (c) represents the global content bias, and (d) represents the global location bias. Then the updated attention score calculation with relative positional embeddings is:

$$\begin{aligned}\mathbf{A}^{\text{abs}}_{i,j} = &\underbrace{\mathbf{E}_{x_i}^\top \mathbf{W}_q^\top \mathbf{W}_{k,E} \mathbf{E}_{x_j}}_{\text{(a)}} + \underbrace{\mathbf{E}_{x_i}^\top \mathbf{W}_q^\top \mathbf{W}_{k,R} \mathbf{R}_{i-j}}_{\text{(b)}} \\ &+ \underbrace{u^\top \mathbf{W}_{k,E} \mathbf{E}_{x_j}}_{\text{(c)}} + \underbrace{v^\top \mathbf{W}_{k,R} \mathbf{R}_{i-j}}_{\text{(d)}}\end{aligned} \tag{3}$$

where $\mathbf{W}_{k,E}$ and $\mathbf{W}_{k,R}$ are for producing the content-based key vectors and the location-based key vectors. u and v are the content bias that doesn't depend on location and location bias that doesn't depend on content respectively.

Thus, for each transformer block, we perform the following computations to output the next hidden state for a single attention head:

$$\begin{aligned}\tilde{\mathbf{h}}_i &= [\, m_i \circ \mathbf{h}_i \,] \\ \mathbf{q},\, \mathbf{k},\, \mathbf{v} &= \mathbf{h}_i \mathbf{W}_q^\top,\, \tilde{\mathbf{h}}_i \mathbf{W}_{k,E}^\top,\, \tilde{\mathbf{h}}_i \mathbf{W}_v^\top \\ \mathbf{A}_{a,b} &= \mathbf{q}_a^\top \mathbf{k}_b + \mathbf{q}_a^\top \mathbf{W}_{k,R}^\top \mathbf{R}_{a-b} \\ &\quad + u^\top \mathbf{k}_b + v^\top \mathbf{W}_{k,R} \mathbf{R}_{a-b} \\ \alpha &= \text{Masked-Softmax}(\mathbf{A})\mathbf{v} \\ \mathbf{u} &= \text{LayerNorm}(\alpha \mathbf{W}_c + \mathbf{h}_i) \\ \mathbf{z} &= W_2^\top \text{ReLU}(W_1^\top \mathbf{u}) \\ \mathbf{h}_{i+1} &= \text{LayerNorm}(\mathbf{u}_i + \mathbf{z})\end{aligned} \tag{4}$$

where each layer has its own $\mathbf{W}_q, \mathbf{W}_v, \mathbf{W}_{k,E}, \mathbf{W}_{k,R} \in \mathbb{R}^{d \times k}$, $W_c \in \mathbb{R}^{k \times d}$, $W_1 \in \mathbb{R}^{d \times m}$, and $W_2 \in \mathbb{R}^{m \times d}$ weights. Note that a, b refer to different indexes in $\mathbf{q}$, $\mathbf{k}$; k is the head dimension; and m is the position-wise feed-forward dimension.

3 Implementation and Training

We designed our model based on a naive version of decoder Transformer with multi-headed attention. This naive version of Transformer does the following in order:

1. Generates word and absolute positional embedding of input.
2. Merges the word and positional embedding and treat this as the initial state.
3. Creates an upper triangular mask as the prefix function.
4. Passes all the above into the Transformer blocks.
5. Takes the output of the last Transformer block, decodes the embedding into tokens, and return it.

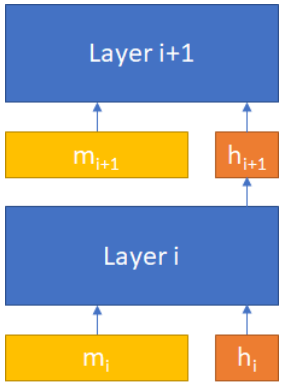

Figure 1: Illustration of layers, memories, and hidden states for TransformerXL

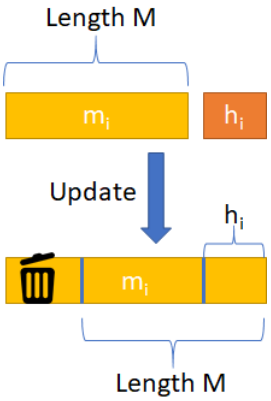

Figure 2: Illustration of how memories are updated for TransformerXL

To convert it into TransformerXL, we made the following changes:

1. Use the relative positional encoding instead of the absolute positional encoding as per Equation (3).
2. Include a word dropout to further prevent overfitting.
3. Enable the embedding dropout for both the word and positional embedding.
4. Treat the word embedding as the initial hidden state h_0, and the output of layer i as h_{i+1}.

5. Pass in both the hidden state h_i and memory m_i as the input to layer i.

6. For key k and value v in layer i, replace h_i with $m_i \circ h_i$ when calculating k and v, where the concatenation is on the dimension of sequence length as per Equation (4). Leave query q unchanged.

7. When calculating the attention, include u and v, the content and location bias, respectively.

8. Update the memory after we received the new hidden state as per Figure 2 and Equation (1).

9. Trim the memory to a max length of M, if necessary.

10. Before switching from train to evaluate or vice versa, clear the memory.

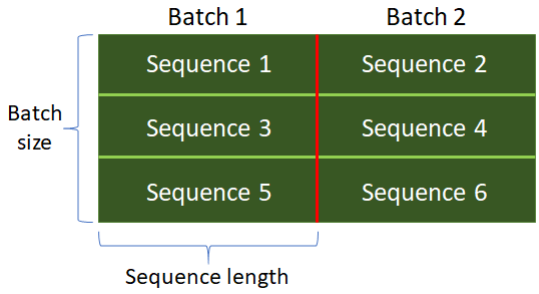

Figure 3: Data from DataLoader for TransformerXL. Note that the sequences make up the entire corpus.

To successfully train the model, we also have to modify the DataLoader to avoid shuffling and to ensure that sequences match up across batch. In other words, if *seq*1 is followed by *seq*2 in the original text, we have to make sure that the first sequence in batch 1 is *seq*1 and the first sequence in batch 2 is *seq*2, not the second sequence of batch 1.

4 Challenges

We faced multiple road-blocks while completing this project. While we were able to resolve most of our issues, we were short on time and weren't able to fix them all. We thought that it is worth mentioning the unexpected challenges we encountered and the effort we put in to resolve them.

4.1 Overfitting

Overfitting was a common issue, especially when we had such a powerful model that could learn the training data well. When we were training our models, we encountered this issue very early on, as early as epoch 2 or 3. Our validation perplexity would start going up as our train perplexity went down. To be precise, the validation perplexity would hit $\sim$140 before jumping back up.

Based on our experience, we started by increasing the dropout rate (embedding dropout to 0.6) and decreasing the depth and number of heads (to 12 and 8, respectively) to a point where they are lower then what we attempted for Transformer in homework. We also included a word dropout of 0.6 to as part of the effort, which would drop words at specific positions while training. However, we were still unsuccessful in further lowering the validation perplexity, leading us to wonder if we have other issues in our code. As we experimented further, we found potential issues not just with out code but with other parts of the training.

4.2 NaN Loss

Another issue that we encountered was if we increase the number of heads to 16, the negative log-likelihood loss would sometimes be NaN on the initial validation loss. Because this occurred right before training, we suspected that this might be due to the way we initialized the weights in the TransformerXL. Partially influenced by the limited CUDA GPU memory available on Colab, we ended up not needing to address this issue as we had to reduce the heads count to 12. We believe that we resolved this issue when we fixed the weight initializations of the u and v biases, which were originally initialized to the garbage values in memory.

4.3 Hyperparameter Tuning and Bug Tracing

One of the most difficult part of the project is hyperparameter tuning. For some of the issues we described above, such as the NaN loss and overfitting, we were unsure if it was due to suboptimal hyperparameters or bugs in the code. We have tried using the same hyperparameters suggested by the TransformerXL paper but were unable to achieve better performance than our original Transformer model.

4.4 Training Inconsistencies

Because of the way TransformerXL trains, we have to modify our DataLoader according to Figure 3. This means changing the order in which sentences are presented and setting shuffle=False. We expected that after doing so, the performance on the original Transformer would only be slightly impacted. To verify our implementation, we loaded the same dataset through our DataLoader and the one implemented by the authors of TransformerXL to and check the outputs. Besides a different word-tokenization order, we found that both loaders have loaded the dataset in the same way. However, as we start training, we see notable differences in both the training and validation perplexities as we substituted one DataLoader with another.

	model	dataset	reset memory	shuffle	learning rate	batch size	heads	depth	memory length	validation perplexity	observations
Vanilla Transformer											
	Vanilla Transformer	original	FALSE	TRUE	0.00035	32	10	16		ep6=162.66	
	Vanilla + Word dropout	original	FALSE	TRUE	0.00035	32	8	12	1	ep10=~204	too volatile in training, perhaps lower lr
Transformer XL (w/ variants)											
	XL + Rel Pos	original	FALSE	TRUE	0.00035	32	8	12	1	ep7=154.28, ep6=166.34	
	XL + Rel Pos + Word dropout	original	FALSE	TRUE	0.00035	32	8	12	1	ep10=147	just bouncing between 140 and 150
	XL + Rel Pos + Word dropout	original	FALSE	TRUE	0.00025	32	8	12	1	ep5=268.33	slow in training
	XL + Rel Pos + UV init	original	TRUE	FALSE	0.00035	32	8	12	2	ep57=124.78	going down very slowly, been between 120,125 for >10epochs
	XL + Rel Pos + UV init + Word dropout	original	FALSE	TRUE	0.00025	32	12	16	1	ep26=162	bottoming out around 160
	XL + Rel Pos + UV init + Word dropout	original	FALSE	TRUE	0.0003	32	10	16	1	ep20=152.55	
Batch Size 1											
	XL + Rel Pos	original	FALSE	FALSE	0.00015	1	8	12	1	ep6=~200	takes a really long time to train
	XL + Rel Pos	original	FALSE	FALSE	0.00035	1	8	12	1	ep2=446, ep1=455	not much change
	XL + Rel Pos	original	FALSE	FALSE	0.00035	1	8	12	2	ep2=400, ep1=380	overfit
	XL + Rel Pos	original	TRUE	FALSE	0.00035	1	8	12	2	ep7 >350, ep6=~300	hovering around 300 ppl, and then exploded
Spectacular failures											
	XL + Rel Pos	original	FALSE	TRUE	0.00035	32	10	16	1	NaN	
	XL + Rel Pos + Word dropout	datasetxl	FALSE	FALSE	0.00025	32	8	12	1	ep1 >700	Overfit immediately
	XL + Rel Pos + UV init + Word dropout	datasetxl	TRUE	FALSE	0.00025	32	8	12	1	ep1 >600	Overfit immediately
	XL + Rel Pos + UV init	datasetxl	TRUE	FALSE	0.00025	32	8	12	1	ep1 >600	Overfit immediately

Figure 4: Results of our experiments.

We were unable to identify any other differences between the outputs of the loaders and were thus unsure if some other factors have played a role. This was the main road-block that took us more than two days to look into. In addition, as part of our experiment, we were very surprised that setting shuffle=True actually decreased the validation perplexity significantly, which is counter-intuitive given how TransformerXL used the memory. When we shuffle the data, sequences no longer match up across batches, yet validation perplexity dropped. This lead us to believe that part of the reason our model overfitted was because a lack of randomness in the dataset. However, shuffling the data completely defeated the purpose of memory as it serves as a regularizer without providing any useful information of the history.

5 Experimental Results

For reference, the Transformer we implemented in homework got a validation perplexity 87.86 after 80 epochs and reached a 81.79 test perplexity.

We note that the performance of our models could not surpass our Transformer implementation. This leads us to believe that there may still be something wrong with the underlying implementation of TransformerXL, even after applying the various fixes.

6 Conclusion

TransformerXL is an exciting idea that can further improve the performance of a regular Transformer. While we were still in the process of fine-tuning our model at the time of writing this, we have nevertheless learned a lot from this project. We learned how a simple concept like memory could be used to address the shortcomings of a state-of-the-art model. We also gained insights on how to tackle debugging deep learning models and the impacts of how data is loaded into the machine learning model.

Of course, there were areas which we could improve. In particular, we were curious if there were other ways of addressing an overfitting TransformerXL model. We had applied all the techniques we knew to try to prevent overfitting, but we still had difficulties getting the validation perplexity significantly lower. Shuffling the data seemed to act as the best regularizer for our model, even though it meant garbling the memory, which warrants further investigation.

Overall, we felt like we learned a great deal from converting a Transformer implementation into TransformerXL and it was a worthwhile endeavor. Given slightly more time, we believe we could have overcome all the remaining issues we faced and been able to beat the results of the original Transformer we started with.

References

[1] Zihang Dai et al. *Transformer-XL: Attentive Language Models Beyond a Fixed-Length Context*. 2019. arXiv: 1901.02860 [cs.LG].

[2] Ashish Vaswani et al. *Attention Is All You Need*. 2017. arXiv: 1706.03762 [cs.CL].