Shortformer: Better Language Modeling Using Shorter Inputs

Ofir Press[1,2] **Noah A. Smith**[1,3] **Mike Lewis**[2]

[1]Paul G. Allen School of Computer Science & Engineering, University of Washington
[2]Facebook AI Research
[3]Allen Institute for AI
ofirp@cs.washington.edu

Abstract

Increasing the input length has been a driver of progress in language modeling with transformers. We identify conditions where shorter inputs are not harmful, and achieve perplexity and efficiency improvements through two new methods that *decrease* input length. First, we show that initially training a model on short subsequences before moving on to longer ones both reduces overall training time and, surprisingly, substantially improves perplexity. Second, we show how to improve the efficiency of recurrence methods in transformers, which let models condition on previously processed tokens when generating sequences that exceed the maximal length the transformer can handle at once. Existing methods require computationally expensive relative position embeddings; we introduce a simple alternative of adding absolute position embeddings to queries and keys instead of to word embeddings, which efficiently produces superior results. We show that these recurrent models also benefit from short input lengths. Combining these techniques speeds up training by a factor of 1.65, reduces memory usage, and substantially improves perplexity on WikiText-103, without adding any parameters.[1]

1 Introduction

Scaling up transformer (Vaswani et al., 2017) language models (Radford et al., 2019; Lewis et al., 2019; Raffel et al., 2019; Brown et al., 2020) has been an important driver of progress in NLP. Language models require data to be segmented into subsequences for both training and inference: memory constraints limit a language model to handling at most a few thousand tokens at once, while many training and evaluation datasets are much longer. Recent work focuses on increasing the length of input subsequences, which determines the maximum number of tokens a model can attend to (Baevski and Auli, 2018; Sukhbaatar et al., 2019; Kitaev et al., 2020; Roy et al., 2020).

We challenge the assumption that longer input subsequences are *always* better by showing that existing transformers do not always effectively use them. We then introduce new methods based on shorter input subsequences that improve runtime, memory efficiency, and perplexity.

We first investigate how input subsequence length affects transformer language models (§3). Naïve evaluation—where we split a large evaluation set into multiple nonoverlapping subsequences, each evaluated independently—initially supports the commonly-held belief that models that train and do inference on longer subsequences achieve better perplexity (Table 1, col. 3).

However, when we evaluate each model with a sliding window (Baevski and Auli, 2018), outputting one token at a time using the maximal amount of context, we find—surprisingly—that models using subsequences exceeding $1{,}024$ tokens do not further improve performance (Table 1, col. 5).

We conclude that the performance gains (using naïve evaluation) of models that use longer subsequences occur not only because of their better modeling ability, but partly because they divide the evaluation set into longer subsequences. This division helps because of an issue we call the *early token curse*: by default, early tokens in a subsequence will have short histories to attend to. Using longer subsequences means fewer tokens will suffer from the early token curse. For example, when using inputs of length $1{,}024$, about 94% of tokens get to attend to more than 64 preceding tokens. If we use inputs of length 128, only 50% of tokens get to attend to 64 or more preceding tokens.

[1] Our code is available at https://github.com/ofirpress/shortformer

Proceedings of the 59th Annual Meeting of the Association for Computational Linguistics and the 11th International Joint Conference on Natural Language Processing, pages 5493–5505
August 1–6, 2021. ©2021 Association for Computational Linguistics

Based on this analysis, we explore how to improve models by using shorter inputs. We introduce two techniques.

Staged Training (§4) First, we show that initially training on shorter subsequences (before moving to longer ones) leads not only to much faster and more memory-efficient training, but it surprisingly also greatly improves perplexity, suggesting that longer inputs are harmful early in training.

Position-Infused Attention (§5) Second, we consider a natural way to avoid the early token curse during training and inference: attending to cached representations from the previously evaluated subsequence (Dai et al., 2019). This approach interferes with conventional absolute position embeddings in a way that forced Dai et al. to use *relative* position embeddings, which are computationally expensive. We introduce a fast, simple alternative: instead of adding absolute position embeddings to word embeddings—thereby entangling a word's content and positional information—we add them to the keys and queries in the self-attention mechanism (but *not* to the values). This does not increase parameter count or runtime. Token representations can then be cached and reused in subsequent computations. We show that when using this method, shorter subsequence models outperform longer ones.

Finally, we show additive gains from combining staged training and position-infused attention (Shortformer, §6), resulting in a model that trains much quicker and achieves better perplexity on WikiText-103. We also show that these results transfer to language modeling on the Toronto Book Corpus (§A.5, appendix).

2 Background and Experimental Setup

Transformer language models map a list of tokens $x_{n-L:n-1}$ to a probability distribution over the next token x_n. We refer to the list of tokens as the *current input subsequence* (whose length is L). Causal masking lets us make L predictions at once, with the prediction for token $i+1$ conditioned on the ith token and all previous inputs $x_{n-L:i-1}$, but not on future inputs. We define the number of tokens the model can attend to at each timestep as its *effective context window*. Note that L is not to be confused with the (typically much greater) length of a training or evaluation dataset.

During inference, language models can be used for two distinct tasks: generation and evaluation. In order to define these tasks, we first define nonoverlapping and sliding window inference.

Nonoverlapping Inference To evaluate a string longer than L, we can evaluate each subsequence of L tokens independently. This fast approach is commonly used during training; if used, tokens in one subsequence cannot condition on those in the previous subsequence, giving rise to the early token curse discussed in §1. See Figure 1(a).

Sliding Window Inference An alternative to the above is to use a sliding window during inference. Here, we choose a stride S between 1 and $L-1$ and advance the window by S tokens after each forward pass.[2] This means that $L-S$ tokens from the previous block are re-encoded, and only S new tokens are outputted. The advantage is that all outputs in each subsequence after the first have at least $L-S$ previous tokens to condition on. However, since tokens must be re-encoded multiple times, this approach is much slower. When $S=1$, we output one token every inference pass, each using the maximal context window, but this is the slowest approach. See Figure 1(b).

Minimal and Maximal Effective Context Window Sizes In the nonoverlapping approach, the min. and max. effective context window sizes are 1 and L, respectively. In the sliding window approach, the max. context window size is still L, but the min. context window size is now $L-S+1$.

Evaluation vs. Generation In *evaluation*, a model assigns a perplexity score to a given sequence. Evaluation is done using either nonoverlapping inference or with a sliding window of any stride; since we already have the target sequence we can simultaneously make predictions for multiple timesteps using causal masking. In *generation*, a model generates a new sequence, as in demonstrations of GPT-3 (Brown et al., 2020). Generation is done only with a sliding window with stride $S=1$, which we refer to as token-by-token generation. During generation, we append to the input a single new token, get a prediction from the model about the next token (e.g., using beam search or picking the token with the highest probability); the process is then repeated.[3]

[2] Nonoverlapping inference can be viewed as sliding window inference with stride L.

[3] In this paper we do not consider open-ended generation; we generate the dev. set, and for next-token prediction we

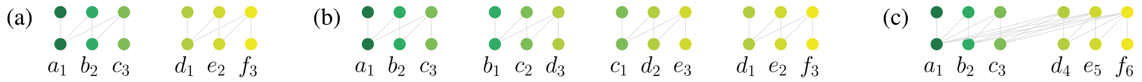

Figure 1: Language model modes for generating or evaluating 6 tokens $(a, b, \ldots, f)$ when subsequence length $L = 3$. The numbers denote the position embeddings (P.E.). (a) Nonoverlapping (§2). (b) Sliding window, stride $S = 1$. Here, after the first inference pass we ignore all outputs other than the last (§2). (c) Caching (§5.2) where each subsequence attends to representations of the previous one. (In the next iteration, tokens d, e and f become the cache, with P.E. 1, 2 and 3, the three new tokens get P.E. 4, 5, and 6.)

Experimental Setup Our baseline is the Baevski and Auli (2018) model, henceforth B&A, trained and evaluated on WikiText-103 (Merity et al., 2016). We use this baseline because of its prominent role in recent language modeling developments (Khandelwal et al., 2020; Press et al., 2020). The training set contains 103.2 million tokens from English Wikipedia. The B&A model has 16 transformer layers of dimension 1,024, with 8 heads in each self-attention sublayer, and feedforward sublayers with an inner dimension of 4,096. This model ties the word embedding and softmax matrices (Press and Wolf, 2017; Inan et al., 2017) and uses sinusoidal position embeddings. It has a subsequence length of 3,072 tokens and achieves a perplexity of 18.65 ± 0.24 (std. dev.) on the development set. In our experiments, other than varying the subsequence length, we modify no other hyperparameters, including the random seed and number of training epochs (205).

3 How Does Context Window Size Affect Transformers?

Segmenting a corpus into subsequences results in different effective context windows for different timesteps depending on where they fall in a segment. Subsequence length L is an upper bound on the effective context window at each timestep. When making the first prediction, the model attends only to the first input token. When making the second prediction, the model attends to the first two inputs, and so on, up to the Lth timestep where the model can attend to all input tokens when making the Lth prediction.

3.1 Context Window Size Matters

Table 1 explores the effect of subsequence length in the B&A model on training runtime and on dev. set perplexity and runtime.[4] We fix the number of tokens in each batch to 9,216 but vary the subsequence length L and batch size (so the product of the batch size and subsequence length remains at 9,216). We report results for both nonoverlapping inference and sliding window inference with stride $S = 1$, which generates only one new token per forward pass; it thus has the maximal effective context window for each generated token. We find that performance increases as S decreases until it reaches a peak and then stops improving (not shown in Table 1).[5]

	Train	Inference			
		Nonoverlapping		Sliding Window (Token-by-token)	
Subseq. Length	Speed ↑	PPL ↓	Speed ↑	PPL ↓	Speed ↑
32	28.3k	35.37	2.4k	24.98	**74**
64	28.5k	28.03	4.8k	21.47	69
128	**28.9k**	23.81	9.2k	19.76	70
256	28.1k	21.45	14.8k	18.86	63
512	26.1k	20.10	18.1k	18.41	37
1024	22.9k	19.11	**18.3k**	17.97	18
1536	18.4k	19.05	17.1k	18.14	11
3072	13.9k	**18.65**	14.7k	**17.92**	5

Table 1: Subsequence length's effects on performance of the B&A model on the WikiText-103 dev. set. The baseline is the last row. Token-by-token inf. was computed with a sliding window stride $S = 1$ to output one token at a time; see §2. We measure speed in tok./sec. per GPU and use a batch size of 1 for inf.

We derive the following conclusions:

Training on long sequences is expensive. Models trained on subsequences of length 256 are twice as fast as models trained on subsequences of 3,072 tokens, but gains for even shorter lengths are negligible (Tab. 1, col. 2).

Long subsequence lengths can improve results. When using the naïve approach, nonover-

use the ground truth token. This has the same complexity as sampling the token with the highest probability.

[4] For consistency, throughout the paper we run inference with a batch size of one. This causes models shorter than $L = 512$ to run slowly (in N.o. eval.), although during batched N.o. eval. they are slightly faster than the $L = 512$ model.

[5] For example, the $L = 3{,}072$ model's performance peaked at $S = 512$ (used in Baevski and Auli (2018)) and then stopped improving. Thus, the result shown in Table 1 for that model with $S = 1$ can also be achieved with $S = 512$ even though that runs 500 times faster, at 2.5k tok./sec.

lapping evaluation, we see a monotonic decrease in dev. perplexity when increasing L (Tab. 1, col. 3).

Increasing the *minimum* effective context window size is more important than increasing the maximum one. Using a sliding window for token-by-token evaluation substantially improves results for all models (Tab. 1, col. 5). Here, we see negligible improvement between the models trained with subsequence lengths of 1,024 and 3,072 tokens (0.05 perplexity). This approach improves results by increasing the minimum amount of context available at each timestep which indicates that long contexts may not be beneficial to transformer models, but very short contexts are harmful. However, sliding window inference can be expensive since each token is encoded many times. For example, token-by-token inference for the $L = 3{,}072$ model is almost 300 times slower than nonoverlapping inference.

4 Training Subsequence Length

§3 results show that models trained on shorter subsequences can be effective at test time, and are much faster to train. We further explore this below.

4.1 Staged Training

We propose a two-stage training routine that initially uses short input subsequences followed by long subsequences.[6] This method was previously applied to speed up the training of BERT (Devlin et al., 2019), but we show that it also improves perplexity.

We use sinusoidal position embeddings; learned position embeddings, which we do not consider, create a dependency between the parameterization and subsequence length. In our experiments, we neither modify nor reset the state of the optimization algorithm between the two stages.

4.2 Experiments

Our experimental setup is described in §2. We do not change any hyperparameters other than reducing subsequence length while correspondingly increasing batch size to keep the number of tokens per batch constant. As in the baseline, all models are trained for 205 epochs.

All models are trained in two stages; the second stage always uses a subsequence length of 3,072, since that lead to the best performance (discussed at end of this subsection).

		Initial Stage Subseqence Length						
		32	64	128	256	512	1024	1536
Initial Stage Epochs	25	17.94	17.57	17.58	18.19	18.06	18.20	18.77
	50	17.81	17.59	**17.52**	18.08	18.01	18.14	18.62
	75	17.93	17.61	17.55	18.01	18.05	18.03	18.57
	100	18.14	17.67	17.62	18.00	18.10	18.00	18.51
	125	18.61	17.88	17.70	18.00	18.13	17.98	18.49
	150	19.45	18.37	17.98	18.01	18.15	18.00	18.49
	175	21.16	19.51	18.57	18.23	18.20	18.08	18.57
	200	35.38	28.03	23.80	21.45	19.63	18.56	18.84

Table 2: Each model's perplexity at the end of training (dev. set, nonoverlapping eval.). All models have a subsequence length of 3,072 tokens at the end of training. The B&A baseline achieves 18.65 ± 0.24 perplexity.

Appendix Table 6 shows the time each training routine takes to match the baseline model's performance on the validation set of WikiText-103.[7] Many configurations match this performance in less than half the time it takes to train the baseline itself; some reach baseline performance in only 37% of the time needed to train the baseline.

Although all models take less time to train than the baseline, Table 2 shows that many **outperform** it. For example, the best model—trained with subsequence length $L = 128$ until epoch 50—outperforms the baseline by 1.1 perplexity despite completing training in 87% of the time the baseline takes to do so. The model that trains with $L = 128$ until epoch 100 achieves similarly strong results (17.62 perplexity) and finishes training in 74% of the time it takes the baseline.[8]

These results are very robust to the choice of initial stage subsequence length and number of epochs. Table 2 shows that all models with an initial stage of $L = 1{,}024$ tokens or less that switch to the second stage at epoch 125 or before beat the baseline by a large margin at the end of training. Additionally, Appendix Table 6 shows that those models match the baseline's perplexity in at most 71% of the time it takes to train the baseline.

When we use nonoverlapping evaluation, the B&A baseline obtains 18.65 perplexity on the development set; our best model obtains 17.52. When we use sliding window evaluation (following Baevski & Auli, we use stride $S = 512$), our best

[6] Curriculum learning (Bengio et al., 2009) trains on easier inputs before progressing to harder ones. Our approach does not change the order in which the training examples are given to the model, but instead modifies their lengths.

[7] Table 7 in the appendix shows the *epoch* at which every model matched the baseline's performance.

[8] Table 8 in the appendix shows the total time it took to train each model.

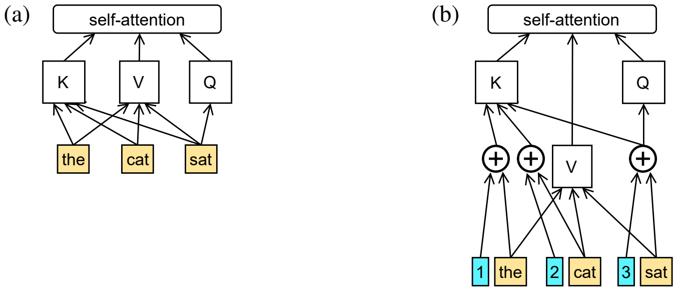

Figure 2: Inputs to the self-attention sublayer, conventionally (left) and with position-infused attention (right), for $L = 3$, at timestep 3. The numbers denote the position embeddings.

model obtains 16.89 perplexity, a large improvement on the 17.92 B&A result in that setting. On the test set, using the same sliding window evaluation, our model obtains 17.56 perplexity, a substantial gain over the baseline's 18.70 test-set perplexity. Appendix Table 10 shows that our best model uses almost five times less memory during the first stage than the baseline.

We also found that setting L to less than 3,072 tokens in the second stage degraded performance. (Appendix Table 9 shows staged training results with an initial stage length of 128 for 50 epochs (as in the best model) and varying lengths for the second stage. We found this to also be true for other initial stage lengths and epochs.) Unlike results in Table 1, where we show that models with L larger than 1,024 do not substantially improve token-by-token generation perplexity, models trained using staged training improve when given longer inputs (Appendix Table 9). Further, we explored using more than two stages (up to six), but this did not outperform our two-stage curriculum.

Finally, Appendix A.5 shows that staged training substantially improves on the Toronto Book Corpus (Zhu et al., 2015).

5 Repositioning Position Embeddings

Sliding window inference substantially improves performance by increasing the minimum effective context window size. But it is very slow. We could solve this by letting the model attend to representations of the previous subsequence during inference on the current one.

In this case, the same token representations would be used in different positions since a token generated near the end of one subsequence would be cached and reused near the start of the next one. However, transformer model representations entangle positional and content information, so a cached token representation would encode an incorrect position when reused in a new position.

TransformerXL (Dai et al., 2019) uses *relative* position embeddings to solve this problem. However, that approach is slower and uses more parameters and memory than the baseline transformer.[9]

We solve this using no extra parameters, memory, or runtime. We also show that our method can use much *shorter* input subsequences and still achieve superior performance.

Transformer Language Models The baseline transformer LM, given a token list T of length L and a tensor $\mathbf{P}$ containing the first L position embeddings, produces L next-token predictions using the following procedure:

1. Embed each token in T, producing tensor $\mathbf{X}$.
2. Add the position embedding of each index to the token at that index: $\mathbf{X} = \mathbf{X} + \mathbf{P}$.
3. Feed $\mathbf{X}$ through each transformer layer. The self-attention sublayer in each transformer layer is invoked as follows: self-attention(key=$\mathbf{X}$, query=$\mathbf{X}$, value=$\mathbf{X}$)
4. Transform the outputs of the last transformer layer using the softmax layer, giving L next-token probability distributions.

5.1 Position-Infused Attention (PIA)

We propose to let the model reuse previous outputs by making each output contain no explicit positional information. To do this, we modify the

[9] The self-attention coefficients between q queries and k keys in TransformerXL are the sum of two dot products of size $q \cdot k$; the unmodified attention sublayer and our PIA method both compute only one dot product of size $q \cdot k$. We also benchmarked the TransformerXL model using its publicly released code and found that their relative position embeddings slow inference by 22% and require 26% more parameters than their implementation of the unmodified self-attention sublayer.

model so that it does not add position embeddings at the *beginning* of the computation (step 2), but rather adds them to the query and key vectors at each layer (but *not* to the value vectors). The outputs at each layer are the transformed, weighted sums of the value vectors, and, since the value vectors in our model do not contain explicit positional information, the outputs also do not.

Formally, steps 1 and 4 do not change, step 2 is omitted, and step 3 is modified to invoke the self-attention sublayer as follows:

$$\text{self-attention}(\text{key}{=}\mathbf{X}{+}\mathbf{P}, \text{query}{=}\mathbf{X}{+}\mathbf{P}, \\ \text{value}{=}\mathbf{X})$$

Figure 2 (b) depicts this method.

Although PIA sublayer outputs contain no explicit positioning information, the attention mechanism can still compute position-dependent outputs because positional information is added to the query and key vectors. Our method is implementable in just a few lines of code.

5.2 PIA Enables Caching

In the unmodified transformer, to generate a string whose length exceeds L, it would have to be split into separate subsequences, and the model would be unable to attend to the previous subsequence when generating the current one.

Using PIA, we can store and attend to representations of the previous subsequence since they no longer contain any explicit positioning information.

Therefore, all our PIA models use a cache, where representations from the previous forward pass are stored and attended to in the next forward pass.

Caching makes generation faster. The complexity of the attention mechanism is $O(q{\cdot}k)$ where q is the number of queries (outputs) and k is the number of key-value pairs (inputs). To generate a sequence whose length exceeds L using token-by-token generation in the unmodified transformer (with subsequence length L), attention takes $O(L^2)$ time (since there are L queries and L keys). Using PIA and caching, we can reuse $L-1$ of the previous outputs at every layer. Thus, our attention sublayer takes $O(L)$ time (because now there is a single query and L keys).

Our approach is useful in scenarios where we need to evaluate or generate sequences that are longer than the model's subsequence length. Therefore, it would not be applicable to sequence-to-sequence tasks such as sentence-level translation, where sequence lengths are short.

Most language models, including B&A, train on their data as nonoverlapping subsequences. This means that training subsequences can be shuffled at each epoch and consumed in random order. However, when using PIA, we would like the cache to contain the previous subsequence. We therefore do not shuffle the data, making the cached subsequence the previously occurring one.

Figure 1(c) depicts training with a cache that contains representations of the previous subsequence.

5.3 Experiments

We use the experimental setup described in §2.

The B&A baseline achieves 18.65 on the development set. We train two additional baselines, the first uses PIA without caching and the second uses caching but no PIA. If just PIA is used (without caching), performance degrades to 19.35 perplexity, but the model's speed and memory usage do not change. Using caching without PIA severely hurts performance, obtaining 41.59 perplexity. Disabling data shuffling in the PIA-only model achieves similar performance to that model when it does use data shuffling, at 19.44 perplexity. Not shuffling the data is necessary for recurrent-style training that caches previously computed subsequence representations.

Our next experiments use the recurrent-style training of Dai et al. (2019), where we receive L new tokens at every training iteration and attend to L' cached representations (of the subsequence of tokens that came immediately prior to the L new tokens). As before, we output L predictions at every training iteration. This means that the maximal and minimal effective context window sizes are $L' + L$ and $L' + 1$, respectively.

In all our models with PIA and caching, we set $L' = L$ because a manual exploration of different models where $L' \neq L$ did not yield better results.

Table 3 compares the results of our models that use PIA and caching to the baseline on the WikiText-103 dev. set. Evaluation and generation speeds are shown in the nonoverlapping (N.o.) and sliding window (S.W., with stride $S = 1$) speed columns, respectively.[10] Unlike in the baseline, token-by-token evaluation in our model achieves the same perplexity as nonoverlapping evaluation

[10] Note that Baevski and Auli (2018) show that the baseline model can also achieve 17.92 during S.W. evaluation, when $S = 512$, with a speed of 2.5k tokens per second.

	Train	Inference		
			Speed ↑	
Subseq. Length	Speed ↑	PPL ↓	N.o.	S.W.
32	22.0k	20.53	2.0k	49
64	23.8k	19.07	4.1k	**51**
128	**24.4k**	18.37	7.9k	50
256	23.5k	17.92	12.8k	48
512	21.5k	**17.85**	14.5k	46
768	17.6k	18.16	13.8k	43
1024	16.6k	18.19	13.9k	39
1536	12.9k	19.11	7.9k	34
Baseline (3072)	13.9k	18.65	**14.7k**	-
		17.92	-	5

Table 3: Dev. perplexity and speed for PIA models trained with different subsequence lengths (L). PIA models attend to L new and L cached tokens at each inference pass. N.o. is nonoverlapping eval.; S.W. is sliding window eval., where we always use $S = 1$ (token-by-token) here. The baseline is evaluated with both evaluation methods. We measure speed in tok./sec. per GPU and use a batch size of 1 for inference.

since in both cases, the predictions for each input subsequence are conditioned not only on the current input, but also on the previous input, making the context window the same in both inference modes (in both cases, at every timestep, the context window is all tokens up to that timestep).

Table 3 shows that as we increase subsequence length, perplexity improves, peaking at 512 before starting to degrade. Our best model obtains 17.85 perplexity, which is multiple standard deviations better than the baseline (18.65, N.o.). Table 5 in §6 shows a similar gain on the test set. The best model runs 1% slower than the baseline during N.o. eval. (since caching reduces the speed gain from smaller attention matrices in this mode). Table 10 (appendix) shows that it uses less than half of the memory the baseline does during training. Our best model trains 55% faster than the baseline.

Our best model, with subsequence length 512, has attention matrices of size $512 \cdot 1{,}024$ (since we have 512 queries—one per every new token—and 1,024 keys and 1,024 values—one per every new token and every cached token). In the baseline, all attention matrices are of size $3{,}072 \cdot 3{,}072$.

Caching previously computed representations lets us do token-by-token generation efficiently when generating more than L tokens. Our model is nine times faster than the baseline at token-by-token generation even as it achieves better perplexity and uses much less memory (Tab. 3, col. 5).

First Stage Subseq. Length	Train Speed ↑	Inference PPL ↓
32	21.6k	17.66
64	22.6k	17.56
128	**22.9k**	**17.47**
256	22.5k	17.50
PIA + Cache w/o Staged Training	21.5k	17.85

Table 4: Dev. perplexity for models that use PIA, caching, and staged training (with final subseq. length of 512). We measure speed in tok./sec. per GPU. Evaluation speed is the same for all models, at 14.5k tok./sec.

PIA and caching also greatly improve perplexity on the Toronto Book Corpus; see A.5 in the appendix.

6 Shortformer Results

To assess whether the gains from staged training, PIA and caching are additive, we take our best caching PIA model, with subsequence length 512, and apply staged training to it, training it with a subsequence length of between 32 to 256 for the first half of training.[11] Table 4 shows the results. As in §4.2, where staged training was applied to the unmodified baseline, the results are very robust to the choice of initial stage subsequence length, with *all* the different choices improving perplexity over the model that does not use staged training.

The best model (with initial subsequence length 128), which we call Shortformer, achieves 17.47 dev. set perplexity and trains 65% faster than the baseline. Since its attention matrices are of dimension $512 \cdot 1{,}024$ (the baseline's are $3{,}072 \cdot 3{,}072$), our model uses less memory (§A.4, appendix). It has the same number of parameters as the baseline.

Figure 3 (appendix) compares our best models using each method we presented (and their combination) to the baseline. It shows that combining caching, PIA and staged training (Shortformer) yields the quickest training and best perplexity when using nonoverlapping evaluation. Evaluation speed is similar for all of these models.

Finally, Table 5 compares our best models on the test set of WikiText-103 to the state of the art.[12]

Shortformer is almost twice as fast to train as the baseline and achieves superior results. Like the

[11] We picked 50% of epochs as the length of the first stage since that produced near-optimal results at a fast speed in §4.

[12] We benchmarked speed, on V100 GPUs, for all models that had publicly available code.

		Train	Inference (Test)		
Model	Param. ↓	Speed ↑	Mode	Speed ↑	PPL ↓
Baseline	**247M**	13.9k	N.o.	**14.7k**	19.40
			S.W.	2.5k	18.70
TransformerXL*	257M	6.0k	N.o.	3.2k	18.30
Sandwich T.	**247M**	13.9k	S.W.	2.5k	17.96
Compressive T.	329M	-	N.o.	-	17.1
Routing T.	-	-	N.o	-	15.8
kNN-LM**	**247M**	13.9k	S.W.	145	**15.79**
PIA + Caching	**247M**	21.5k	N.o.	14.5k	18.55
Staged Training	**247M**	17.6k	S.W.	2.5k	17.56
Shortformer	**247M**	**22.9k**	N.o.	14.5k	18.15

Table 5: Comparison of our best models to other strong LMs (see text for citations and explanations) evaluating the WikiText-103 test set, where $S = 512$. We measure speed in tok./sec. per GPU, and use a batch size of 1 for inference. *TransformerXL runs on an older version of PyTorch, which might affect speed. **kNN-LM requires a 400GB datastore.

best model from §5.3, it is nine times faster than the baseline for token-by-token generation.

Since it uses a cache, sliding window evaluation does not increase Shortformer's performance. By training the baseline with staged training (and no PIA or caching), we obtain a model (our best model from §4.2) that, with sliding window eval., obtains even better results, but that model is much slower than Shortformer (Table 5, second-to-last row).

Shortformer outperforms the baseline's perplexity and performs within a standard deviation of the Sandwich Transformer (Press et al., 2020) and TransformerXL. It does not outperform the Compressive Transformer (Rae et al., 2020), Routing Transformer (Roy et al., 2020) and kNN-LM (Khandelwal et al., 2020), which make orthogonal improvements that can be applied to any language model, at the price of slower decoding. Combining them with our approach may yield further gains. These results are similar to those we obtain on the Toronto Book Corpus (§A.5 in the appendix).

7 Related Work

Staged Training Devlin et al. (2019) used a staged training routine for BERT by performing the first 90% of training on short subsequences (of length 128) before moving on to longer ones (of length 512). They use this method to speed training, but we show that also it improves perplexity and analyze different configurations of this method.

Many recent papers have explored improving transformer efficiency by reducing the quadratic cost of self-attention, motivated by scaling to longer sequences (Kitaev et al., 2020; Roy et al., 2020; Tay et al., 2020). We instead demonstrate improved results with shorter sequences, which naturally also improve efficiency.

One way to reduce transformer memory usage is to sparsify the attention matrix by letting the model attend only to a subset of nearby tokens at each timestep (Child et al., 2019; Beltagy et al., 2020; Roy et al., 2020). Training on shorter subsequence lengths is much more efficient: we use multiple, but much smaller, attention matrices. Since attention uses memory and computation in a way that scales quadratically with input size, splitting the inputs into multiple subsequences each processed independently lets us use less memory and run faster. Like our method, Beltagy et al. (2020) attend at each timestep to a growing number of neighbors as training progresses, but they use five stages, which we found not to be superior to our two-staged method.

The adaptive attention span model of Sukhbaatar et al. (2019) learns the maximum effective context window sizes for each head at each layer independently. Like in our method, context window sizes are smaller at the start of training and lengthen as training progresses. We show that a simple approach of manually choosing two subsequence lengths is highly effective. In addition, keeping subsequence lengths equal across all heads and layers lets us save memory and runtime.

Position-Infused Attention TransformerXL (Dai et al., 2019) caches and attends to previous representations using an attention sublayer that uses relative positioning (Shaw et al., 2018). It runs much slower than the unmodified attention sublayer, requires extra parameters, and requires internally modifying the self-attention sublayer, while our PIA method (§5) does not.

In parallel with our work, Ke et al. (2020) compute attention coefficients by summing two attention matrices, one based on position-position interactions and the other on content-content interactions. As in PIA, they do not add position embeddings at the bottom of the model. They present results only for BERT, which uses much smaller subsequences than our models.

8 Conclusion

Our results challenge the conventional wisdom that longer subsequences are *always* better. By first training on shorter subsequences and then progressing to longer ones via staged training, we improve perplexity and reduce training time. We additionally propose position-infused attention, which enables caching and efficiently attending to previous outputs; we show that models using this method do not require large input subsequences. We finally show that these two methods can be combined to produce a speedier and more accurate model.

Acknowledgments

We thank Tim Dettmers, Jungo Kasai, Gabriel Ilharco, Hao Peng, Sewon Min, Mandar Joshi, Omer Levy, Luke Zettlemoyer, Julian Michael, Edward Misback, Sofia Serrano, Nikolaos Pappas, Jesse Dodge, Myle Ott, and Sam Shleifer for their valuable feedback and fruitful discussions.

References

Alexei Baevski and Michael Auli. 2018. Adaptive input representations for neural language modeling. *CoRR*, abs/1809.10853.

Iz Beltagy, Matthew E. Peters, and Arman Cohan. 2020. Longformer: The long-document transformer. *arXiv:2004.05150*.

Yoshua Bengio, Jérôme Louradour, Ronan Collobert, and Jason Weston. 2009. Curriculum learning. In *Proceedings of the 26th Annual International Conference on Machine Learning*, ICML '09, page 41–48, New York, NY, USA. Association for Computing Machinery.

Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. 2020. Language models are few-shot learners.

Rewon Child, Scott Gray, Alec Radford, and Ilya Sutskever. 2019. Generating long sequences with sparse transformers. *URL https://openai.com/blog/sparse-transformers*.

Zihang Dai, Zhilin Yang, Yiming Yang, Jaime G Carbonell, Quoc Le, and Ruslan Salakhutdinov. 2019. Transformer-xl: Attentive language models beyond a fixed-length context. In *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*, pages 2978–2988.

Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. BERT: Pre-training of deep bidirectional transformers for language understanding. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*, pages 4171–4186, Minneapolis, Minnesota. Association for Computational Linguistics.

Hakan Inan, Khashayar Khosravi, and Richard Socher. 2017. Tying word vectors and word classifiers: A loss framework for language modeling. In *ICLR*.

Guolin Ke, Di He, and Tie-Yan Liu. 2020. Rethinking positional encoding in language pre-training.

Urvashi Khandelwal, Omer Levy, Dan Jurafsky, Luke Zettlemoyer, and Mike Lewis. 2020. Generalization through Memorization: Nearest Neighbor Language Models. In *International Conference on Learning Representations (ICLR)*.

Nikita Kitaev, Lukasz Kaiser, and Anselm Levskaya. 2020. Reformer: The efficient transformer. In *International Conference on Learning Representations*.

Mike Lewis, Yinhan Liu, Naman Goyal, Marjan Ghazvininejad, Abdelrahman Mohamed, Omer Levy, Ves Stoyanov, and Luke Zettlemoyer. 2019. Bart: Denoising sequence-to-sequence pre-training for natural language generation, translation, and comprehension. *arXiv preprint arXiv:1910.13461*.

Stephen Merity, Caiming Xiong, James Bradbury, and Richard Socher. 2016. Pointer sentinel mixture models.

Ofir Press, Noah A. Smith, and Omer Levy. 2020. Improving transformer models by reordering their sublayers. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, pages 2996–3005, Online. Association for Computational Linguistics.

Ofir Press and Lior Wolf. 2017. Using the output embedding to improve language models. In *Proceedings of the 15th Conference of the European Chapter of the Association for Computational Linguistics: Volume 2, Short Papers*, pages 157–163, Valencia, Spain. Association for Computational Linguistics.

Alec Radford, Jeff Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. 2019. Language models are unsupervised multitask learners.

Jack W. Rae, Anna Potapenko, Siddhant M. Jayakumar, Chloe Hillier, and Timothy P. Lillicrap. 2020. Compressive transformers for long-range sequence modelling. In *International Conference on Learning Representations*.

Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J Liu. 2019. Exploring the limits of transfer learning with a unified text-to-text transformer. *arXiv preprint arXiv:1910.10683*.

Aurko Roy, Mohammad Saffar, Ashish Vaswani, and David Grangier. 2020. Efficient content-based sparse attention with routing transformers.

Peter Shaw, Jakob Uszkoreit, and Ashish Vaswani. 2018. Self-attention with relative position representations. In *Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 2 (Short Papers)*, pages 464–468, New Orleans, Louisiana. Association for Computational Linguistics.

Sainbayar Sukhbaatar, Edouard Grave, Piotr Bojanowski, and Armand Joulin. 2019. Adaptive attention span in transformers. In *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*, pages 331–335, Florence, Italy. Association for Computational Linguistics.

Yi Tay, Mostafa Dehghani, Dara Bahri, and Donald Metzler. 2020. Efficient transformers: A survey.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Ł ukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. In *Advances in Neural Information Processing Systems*, volume 30, pages 5998–6008. Curran Associates, Inc.

Yukun Zhu, Ryan Kiros, Rich Zemel, Ruslan Salakhutdinov, Raquel Urtasun, Antonio Torralba, and Sanja Fidler. 2015. Aligning books and movies: Towards story-like visual explanations by watching movies and reading books. In *Proceedings of the IEEE international conference on computer vision*, pages 19–27.

A Appendix

A.1 Additional Staged Training Results

Table 6 shows the time each staged training model needs to match baseline performance, as a fraction of the time it takes to train the baseline. The fastest three configurations each match the baseline's performance in just 37% of the time it takes to train the baseline. This result is very robust to hyperparameter changes, as all models trained with initial subsequence length of between 64 and 512, that switch to the second stage at epoch 50 to 150, manage to match the baseline's performance in at most 59% of the time it takes to train it.

Initial Stage Epochs	Initial Stage Subsequence Length						
0	32	64	128	256	512	1024	1536
25	0.60	0.54	0.53	0.65	0.64	0.71	
50	0.53	0.48	0.47	0.54	0.59	0.63	0.81
75	0.51	0.43	0.42	0.48	0.53	0.56	0.79
100	0.52	0.40	0.38	0.41	0.47	0.50	0.73
125	0.61	0.41	**0.37**	**0.37**	0.42	0.46	0.69
150		0.48	0.39	**0.37**	0.40	0.44	0.66
175			0.48	0.43	0.45	0.51	0.70
200						0.59	

Table 6: Time needed to match baseline performance (dev. set, nonoverlapping eval.) as a fraction of time needed to train the baseline (smaller is better). Models never matching the baseline have empty cells. All models have a subsequence length of 3,072 tokens at the end of training.

Initial Stage Epochs	Initial Stage Subsequence Length						
	32	64	128	256	512	1024	1536
25	136	123	**122**	146	144	155	
50	135	124	**122**	136	144	149	179
75	143	128	125	136	144	145	181
100	158	135	130	136	145	142	175
125	190	149	141	140	146	144	174
150		176	160	154	153	151	174
175			191	178	177	176	189
200						202	

Table 7: Epoch at which each model matches the baseline. Some models never match the baseline, and so those cells are empty.

Tables 7 and 8 show the epoch at which each model matched the baseline's performance and the total time it took to train each of our staged training models.

Initial Stage Epochs	Initial Stage Subsequence Length						
	32	64	128	256	512	1024	1536
25	0.94	0.94	0.94	0.94	0.94	0.95	0.97
50	0.87	0.87	0.87	0.87	0.88	0.90	0.94
75	0.81	0.81	0.81	0.81	0.82	0.85	0.90
100	0.75	0.75	0.74	0.75	0.77	0.80	0.87
125	0.68	0.68	0.68	0.69	0.71	0.75	0.84
150	0.62	0.62	0.61	0.62	0.65	0.70	0.81
175	0.56	0.56	0.55	0.56	0.59	0.66	0.78
200	0.49	0.49	**0.48**	0.50	0.53	0.61	0.75

Table 8: Total time needed to train each model as a fraction of the time needed for baseline training.

A.2 Staged Training with Shorter Final Stage L

In section 4, all models presented used Staged Training with a final input subsequence length L of 3,072 tokens. In Table 9, we show the results of training with a first stage with $L = 128$ for 50 epochs, and using varying subsequence lengths for the second stage. The best result is obtained when the second stage uses $L = 3{,}072$. In addition, in all of our other experiments (not presented here) with different L and epoch number values for the first stage, we observed that using $L = 3{,}072$ for the second stage always achieved the best perplexities. Models trained with staged training and evaluated with a sliding window sometimes perform slightly worse when S is decreased, but this difference is much smaller than the standard deviation. The $L = 1536$ and $L = 3072$ models peaked at $S = 512$, and then as S was decreased perplexity started slightly degrading.[13]

A.3 Training Speed vs. Performance

Figure 3 compares the validation performance and training speed of the baseline to our models.

A.4 Memory Usage

To understand how much memory our models and the baseline use during training, we find the largest batch size that we can load into memory for both our models and the baseline. Models that can simultaneously make more predictions are more memory efficient.

[13] We conjecture that this is because of a train-test mismatch that occurs since the average effective context length during training is $\frac{3{,}072}{2} = 1{,}536$ and so the model focuses on learning how to make predictions for the tokens in the center of the input, and does not perform as well when making predictions for tokens at the end of the input (which is what we use when using sliding window evaluation).

Final Subseq. Length	Inference PPL ↓		
	Nonoverlapping	Sliding Window $S = 512$	Sliding Window $S = 1$
256	21.26	-	18.72
512	19.69	19.69	18.04
1024	18.64	17.60	17.58
1536	18.10	17.28	17.30
3072	**17.52**	**16.89**	**17.01**

Table 9: Inference perplexity for staged training models trained with an initial stage subsequence length of 128 for 50 epochs and varying second stage subsequence length L (for the second stage's 155 epochs). S is stride. To see how these models perform without staged training, refer to Table 1.

Model		Training	
		Max Batch Size ↑	Max Predictions ↑
Baseline		2	6,144
Staged Training	Stage 1	**230**	**29,440**
	Stage 2	2	6,144
PIA + Caching		26	13,312
Shortformer	Stage 1	160	20,480
	Stage 2	26	13,312

Table 10: Memory usage of the baseline and our models during WikiText-103 training. For each model we show the maximal batch size that it could fit on one GPU at once during training. The max predictions column denotes the number of tokens predicted at each feedforward pass, which we calculate by multiplying batch size by number of predictions per subsequence (which is equivalent to L). We benchmarked all models on a V100 GPU, with 32GB of memory. Note that the second stage in the staged training model matches the performance of the baseline model, because those architectures are identical. The same is true for the second stage of the Shortformer and the PIA + Caching model.

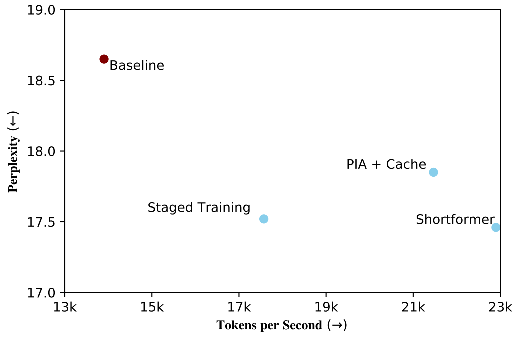

Figure 3: Dev. perplexity vs. training speed for the baseline and our best staged training model, our best PIA and caching model, and our best combined model (Shortformer). All models are evaluated using nonoverlapping evaluation.

Table 10 shows the memory usage for the baseline model and our models. Since our Shortformer model has much smaller attention matrices, during training it can make more than double the next-token predictions than the baseline can in each feedforward pass. During inference, we use a batch size of 1 throughout the paper, following (Dai et al., 2019), and in our experiments, the PIA + Caching model, the final staged training model and the baseline all use a similar amount of memory during nonoverlapping evaluation. During token-by-token inference, the maximum number of predictions for the baseline model is 7, whereas our model can fit a batch size of 39 (so 39 predictions are made during token-by-token inference), making our model more than 5 times more memory efficient than the baseline. Using a batch size of one is a realistic benchmarking scenario: in large models such as GPT-3, a batch size of one is used during inference.

A.5 Toronto Book Corpus

To verify that our results transfer to other datasets, we ran our models on the Toronto Book Corpus (TBC) (Zhu et al., 2015), a 700M token collection of books that has previously been used in the training corpus of BERT (along with English Wikipedia). We use the same train/development/test split as (Khandelwal et al., 2020) and (Press et al., 2020), as well as their tokenization, which uses BERT's vocabulary of 29K BPE subwords. As in (Khandelwal et al., 2020) and (Press et al., 2020), since the vocabulary is much smaller than WikiText-103's, we use a tied word embedding and softmax matrix (Press and Wolf, 2017; Inan et al., 2017), instead of using the adaptive word embeddings (Baevski and Auli, 2018) as in the WikiText-103 models.

To fairly compare our models to the ones from (Khandelwal et al., 2020) and (Press et al., 2020), our initial set of experiments on the TBC use a maximum subsequence length of 1,024 (for staged training), train for 59 epochs, and for all other hyperparameters we use the same values as the ones we used for WikiText-103 (see Experiment Setup in Section 2). In this setting, the baseline achieves a perplexity of 15.38 ± 0.39 (standard deviation) on the development set.

We do not tune the hyperparameters of our methods on the TBC, we simply use the same values as the best ones that we found on the WikiText-103 dataset. For staged training, our best model trained for $\frac{50}{205}\%$ of the epochs with $L = 128$ and spent the rest of training with the same subsequence size as the baseline. For the TBC, we again trained the staged training model model with $L = 128$ for the first $\frac{50}{205}\%$ of training, and then move on to $L = 1{,}024$, to match the Sandwich Transformer (Press et al., 2020) and kNN-LM (Khandelwal et al., 2020) which used 1,024 as the subsequence length.

For the PIA + Caching model, we set $L = 512$, as we did for our best PIA + Caching on the WikiText-103 dataset.

For the Toronto Book Corpus Shortformer, we trained for the first half of training with $L = 128$ before moving on to training with $L = 512$, as in our WikiText-103 models (Section 6).

	Train	Inference			
				PPL ↓	
Model	Speed↑	Mode	Speed↑	Dev.	Test
Baseline	**24.0k**	N.o.	**19.2k**	15.38	12.73
		S.W.	9.6k	14.75	11.89
kNN-LM*	**24.0k**	S.W.	-	14.20	10.89
Sandwich T.	**24.0k**	S.W.	9.6k	-	10.83
PIA + Caching	20.5k	N.o.	15.0k	13.86	11.20
Staged Training	25.5k	N.o.	**19.2k**	13.81	11.18
		S.W.	9.6k	**13.13**	**10.72**
Shortformer	21.3k	N.o.	15.5k	13.40	10.88

Table 11: Comparison of our best models to other strong LMs trained on the Toronto Book Corpus (TBC). Following Khandelwal et al. (2020) and Press et al. (2020), for the baseline and our staged training model, we set $L = 1{,}024$ and $S = 512$ when using sliding window (S.W.) evaluation in the TBC dataset. All models have 261M parameters. *kNN-LM requires a 400GB datastore.

Table 11 shows that staged training and the Shortformer improve over the baseline by a wide margin and match the results of the Sandwich Transformer and the kNN-LM. As noted in Section 6, those contributions are orthogonal to ours, and combining them might yield further gains. Since in Table 11 the final stage of the staged training model (and the baseline) both have $L = 1{,}024$, Shortformer lacks a speed advantage in this scenario.

Table 12 shows results for our staged training model trained with a final stage subsequence length of 3,072 tokens, as in our WikiText-103 experiments in Section 4. This model trains faster than the $L = 3{,}072$ baseline and also achieves much better perplexity scores (the baseline in this setting achieves a perplexity of 14.52 ± 0.15 (standard deviation) on the development set). In addition, note that the Shortformer model from Table 11 achieves better perplexity than even the baseline with $L = 3{,}072$, although Shortformer is much faster to train and uses much smaller attention matrices during inference (of size $512 \cdot 1024$; the baseline has attention matrices of size $3{,}072 \cdot 3{,}072$, as in Section 6).

	Train	Inference			
				PPL ↓	
Model	Speed↑	Mode	Speed↑	Dev.	Test
Baseline ($L = 3{,}072$)	14.2	N.o. S.W.	**15.1** 2.5k	14.52 14.14	11.69 11.43
Staged Training ($L = 3{,}072$)	**18.1**	N.o. S.W.	**15.1** 2.5k	13.19 **12.80**	10.76 **10.48**

Table 12: Comparison of the staged training model to the baseline, when the subsequence length L is set to 3,072. In this table, $S = 512$.