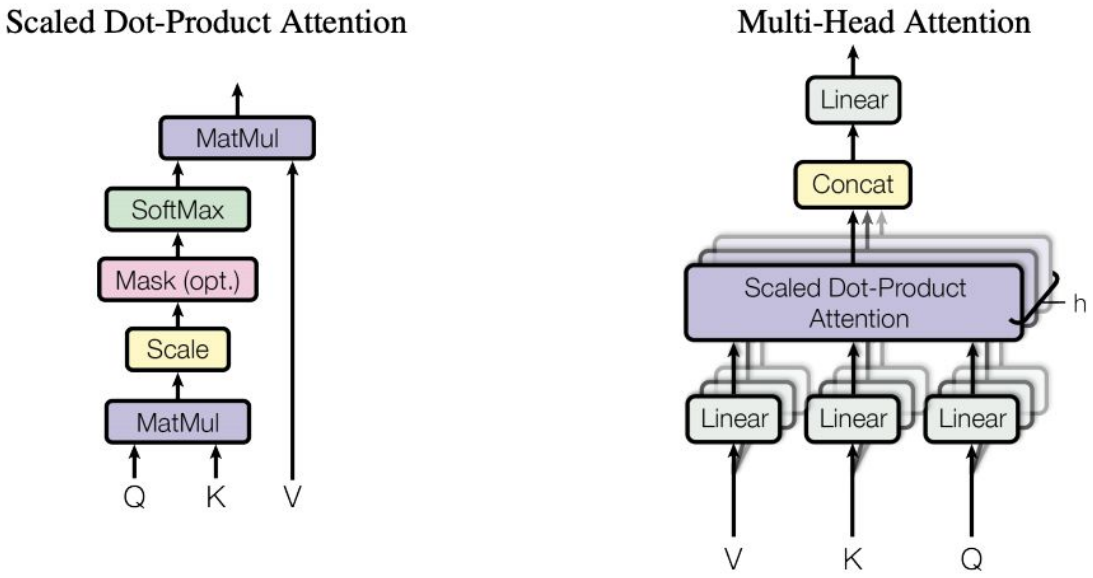
TRANSFORMERS

Attention is all you need (2017)



$$Q, K \in \mathbb{R}^{d_{model}, d_v}$$

Q, K \in \mathbb{R}^{d_{model}, d_v}

$$Attention(Q, K, V) = softmax\left(\frac{QK^T}{\sqrt{d_k}}\right)V$$

Attention(Q, K, V) = softmax\left(\frac{QK^T}{\sqrt{d_k}}\right)V

Multi-head attention expands the model's ability to focus on different positions and gives the attention layer multiple representation subspaces.

$$Multi - head(Q, K, V) = concat(head_1, , head_h)W^o$$

Multi-head(Q, K, V) = concat(head_1, …, head_h)W^o

where $head_i = Attention(QW_i^Q, KW_i^K, VW_i^V)$

head_i = Attention(QW_i^Q, KW_i^K, VW_i^V)

$$W_i^Q, W_i^K \in \mathbb{R}^{d_{model}, d_k}$$

W_i^Q, W_i^K \in \mathbb{R}^{d_{model}, d_k}

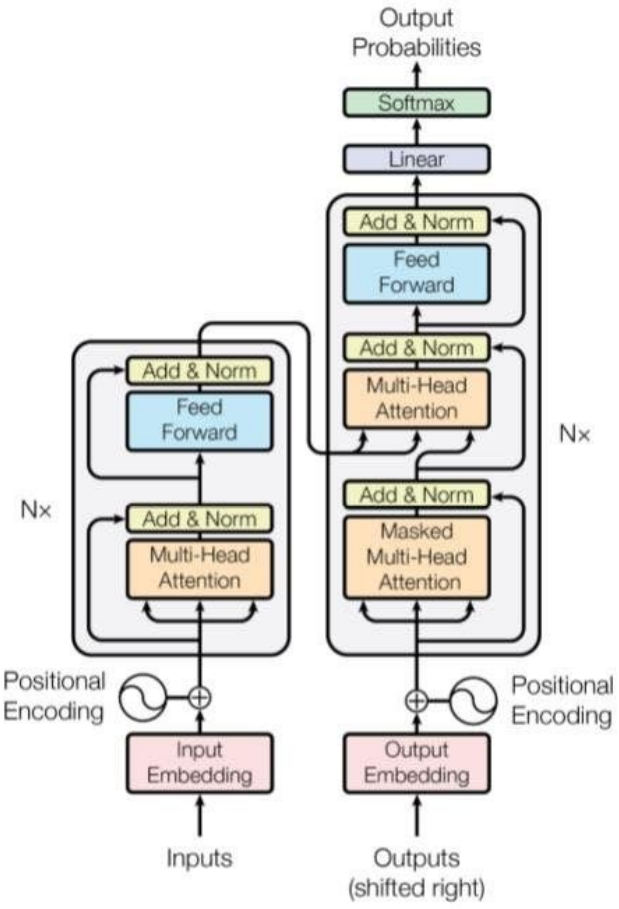
$$W_i^V \in \mathbb{R}^{d_{model}, d_v}$$

W_i^V \in \mathbb{R}^{d_{model}, d_v}

$$W^o \in \mathbb{R}^{h \cdot d_v, d_{model}}$$

W^o \in \mathbb{R}^{h \cdot d_v, d_{model}}

In particular, in the “Attention is all you need” paper, $h = 8$, $d_k = d_v = \frac{d_{model}}{h} = 64$, $d_{model} = 512$.
In particular, in the “Attention is all you need” paper, h = 8, d_k = d_v = \frac{d_{model}}{h} = 64, d_{model} = 512.



This is the final structure of the Transformer.
Where $N = 6$, $FeedForw(x) = max(0, xW_1 + b_1)W_2 + b_2$, and the hidden layer dimension $d_{ff} = 2048$.
Where N = 6, Feed Forw (x) = max(0, xW_1+b_1)W_2 + b_2, and the hidden layer dimension d_{ff} = 2048.

The decoder's second multi-headed attention block takes Q from the previous layer and K, V from the encoder output. The decoder has an output of dimension d_{model}, the final linear layer converts it to a logits vector of dimension $d_{vocabulary}$. The softmax function outputs a probability distribution.

GPT (Jun 2018)

Introduction

We explore a semi-supervised approach for language understanding tasks using a combination of unsupervised pre-training and supervised fine-tuning. First, we use a language modeling objective on the unlabeled data to learn the initial parameters of a neural network model. Subsequently, we adapt these parameters to a target task using the corresponding supervised objective.

Model

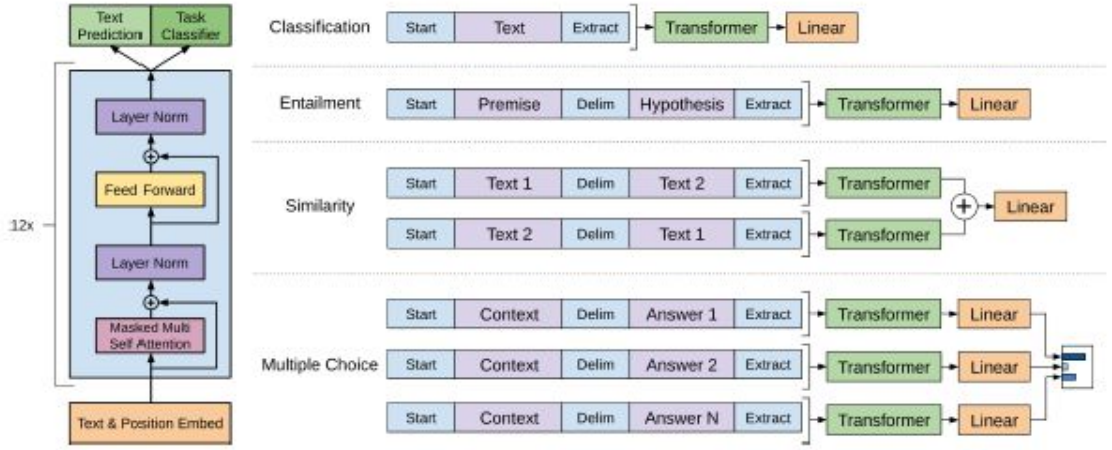


Figure 1: (**left**) Transformer architecture and training objectives used in this work. (**right**) Input transformations for fine-tuning on different tasks. We convert all structured inputs into token sequences to be processed by our pre-trained model, followed by a linear+softmax layer.

12-layer decoder-only Transformer with masked self-attention heads (768 dimensional states and 12 attention heads). For the position-wise feed-forward networks, we used 3072 dimensional inner states. We used the Adam optimization scheme with a max learning rate of 2.5e-4. The learning rate was increased linearly from zero over the first 2000 updates and annealed to 0 using a cosine schedule. We train for 100 epochs on mini batches of 64 randomly sampled, contiguous sequences of 512 tokens. We used a bytepair encoding (BPE) vocabulary. For the activation function, we used the Gaussian Error Linear Unit (GELU). We used learned position embeddings instead of the sinusoidal version proposed in the original work.

Fine tuning

Unless specified, we reuse the hyperparameter settings from unsupervised pre-training. We add dropout to the classifier with a rate of 0.1. For most tasks, we use a learning rate of 6.25e-5 and a batch size of 32. Our model finetunes quickly and 3 epochs of training was sufficient for most cases. We use a linear learning rate decay schedule with warm up over 0.2% of training. λ was set to 0.5.

Unsupervised pre-training

Given an unsupervised corpus of tokens $U = \{u_1, ..., u_n\}$, we use a standard language modeling objective to maximize the following likelihood:

$$L_1(U) = \sum_i logP(u_i|u_{i-k},,u_{i-1};\Theta)$$

L_1(U) = \sum_i log P(u_i | u_{i-k}, …, u_{i-1}; \Theta)

where k is the size of the context window, and the conditional probability P is modeled using a neural network with parameters Θ. These parameters are trained using stochastic gradient descent. In our experiments, we use a multi-layer Transformer decoder for the language model, which is a variant of the transformer. This model applies a multi-headed self-attention operation over the input context tokens followed by position-wise feedforward layers to produce an output distribution over target tokens:

$$h_0 = UW_e + W_p$$
$$h_l = \text{transformer block}(h_{l-1}) \quad \forall i \in [1, n]$$
$$P(u) = softmax(h_n W_e^T)$$

h_0 = UW_e + W_p

h_l = \text{transformer_block}(h_{l-1})\quad \forall i \in [1, n]

P(u) = softmax(h_nW_e^T)

where $U = (u_{-k}, ..., u_{-1})$ is the context vector of tokens, n is the number of layers, W_e is the token embedding matrix, and W_p is the position embedding matrix.

Supervised fine-tuning

After training, we adapt the parameters to the supervised target task. We assume a labeled dataset C, where each instance consists of a sequence of input tokens, $x^1,,x^m$, along with a label y. The inputs are passed through our pre-trained model to obtain the final transformer block's activation h_l^m, which is then fed into an added linear output layer with parameters W_y to predict y:

$$P(y|x^1,,x^m) = softmax(h_l^m W_y)$$

P(y | x^1, …, x^m) = softmax(h_l^mW_y)

This gives us the following objective to maximize:

$$L_2(C) = \sum_{(x,y)} logP(y|x^1,,x^m)$$

L_2(C) = \sum_{(x, y)} log P (y | x^1, …, x^m)

We additionally found that including language modeling as an auxiliary objective to the fine-tuning helped learning by (a) improving generalization of the supervised model, and (b) accelerating convergence. Specifically, we optimize the following objective (with weight λ):

$$L_3(C) = L_2(C) + \lambda \cdot L_1(C)$$

L_3(C) = L_2(C) + \lambda \cdot L_1(C)

GPT2 (Feb 2019)

Model

We use a Transformer (Vaswani et al., 2017) based architecture for our LMs. The model largely follows the details of the OpenAI GPT model with few modifications. Layer normalization (Ba et al., 2016) was moved to the input of each sub-block, similar to a pre-activation residual network (He et al., 2016) and an additional layer normalization was added after the final self-attention block. A modified initialization which accounts for the accumulation on the residual path with model depth is used. We scale the weights of residual layers at initialization by a factor of $1/\sqrt{N}$ where N is the number of residual layers. The vocabulary is expanded to 50,257. We also increase the context size from 512 to 1024 tokens and a larger batch size of 512 is used.

The smallest model is equivalent to the original GPT, and the second smallest equivalent to the largest model from BERT (Devlin et al., 2018). Our largest model, which we call GPT-2, has over an order of magnitude more parameters than GPT.

Summarization

We test GPT-2's ability to perform summarization on the CNN and Daily Mail dataset (Nallapati et al., 2016). To induce summarization behavior we add the text TL;DR: after the article and generate 100 tokens with Top-k random sampling (Fan et al., 2018) with k = 2 which reduces repetition and encourages more abstractive summaries than greedy decoding. We use the first 3 generated sentences in these 100 tokens as the summary.

While qualitatively the generations resemble summaries, they often focus on recent content from the article or confuse specific details such as how many cars were involved in a crash or whether a logo was on a hat or shirt. On the commonly reported ROUGE 1, 2, L metrics the generated summaries only begin to approach the performance of classic neural baselines and just barely outperforms selecting 3 random sentences from the article. GPT-2's performance drops by 6.4 points on the aggregate metric when the task hint is removed which demonstrates the ability to invoke task specific behavior in a language model with natural language.

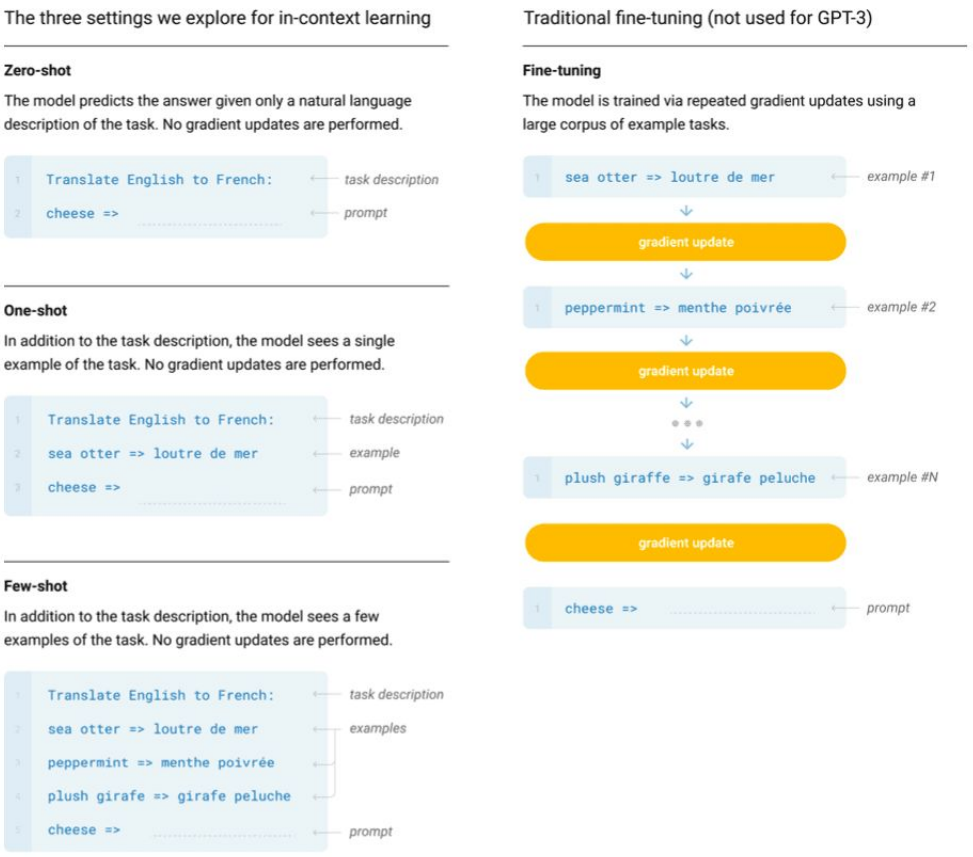
GPT3 (Jun 2020)

Introduction

In this paper, we train a 175 billion parameter autoregressive language model, which we call GPT-3, and measuring its in-context learning abilities. Specifically, we evaluate GPT-3 on over two dozen NLP datasets, as well as several novel tasks designed to test rapid adaptation to tasks unlikely to be directly contained in the training set. For each task, we evaluate GPT-3 under 3 conditions: (a) "few-shot learning", or in-context learning where we allow as many demonstrations as will fit into the model's context window (typically 10 to 100), (b) "one-shot learning", where we allow only one demonstration, and (c) "zero-shot" learning, where no demonstrations are allowed and only an instruction in natural language is given to

the model. GPT-3 could also in principle be evaluated in the traditional fine-tuning setting, but we leave this to future work.
Few-shot learning also improves dramatically with model size. **We emphasize that these "learning" curves involve no gradient updates or fine-tuning, just increasing numbers of demonstrations given as conditioning.**
Broadly, on NLP tasks GPT-3 achieves promising results in the zero-shot and one-shot settings, and in the few-shot setting is sometimes competitive with or even occasionally surpasses state-of-the-art.

Zero, one, few shots



In this paper we focus on zero-shot, one-shot and few-shot, with the aim of comparing them not as competing alternatives, but as different problem settings which offer a varying trade-off between performance on specific benchmarks and sample efficiency. We especially highlight the few-shot results as many of them are only slightly behind state-of-the-art fine-tuned models. Ultimately, however, one-shot, or even sometimes zero-shot, seem like the fairest comparisons to human performance, and are important targets for future work.

Model

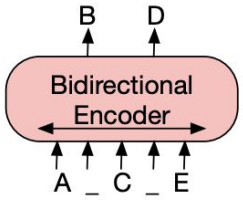
We use the same model and architecture as GPT-2, including the modified initialization, pre-normalization, and reversible tokenization described therein, with

the exception that we use alternating dense and locally banded sparse attention patterns in the layers of the transformer, similar to the Sparse Transformer.
All models use a context window of $n_{ctx} = 2048$ tokens.

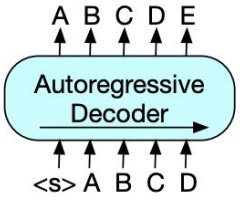
Model Name	n_{params}	n_{layers}	d_{model}	n_{heads}	d_{head}	Batch Size	Learning Rate
GPT-3 Small	125M	12	768	12	64	0.5M	6.0×10^{-4}
GPT-3 Medium	350M	24	1024	16	64	0.5M	3.0×10^{-4}
GPT-3 Large	760M	24	1536	16	96	0.5M	2.5×10^{-4}
GPT-3 XL	1.3B	24	2048	24	128	1M	2.0×10^{-4}
GPT-3 2.7B	2.7B	32	2560	32	80	1M	1.6×10^{-4}
GPT-3 6.7B	6.7B	32	4096	32	128	2M	1.2×10^{-4}
GPT-3 13B	13.0B	40	5140	40	128	2M	1.0×10^{-4}
GPT-3 175B or "GPT-3"	175.0B	96	12288	96	128	3.2M	0.6×10^{-4}

Table 2.1: Sizes, architectures, and learning hyper-parameters (batch size in tokens and learning rate) of the models which we trained. All models were trained for a total of 300 billion tokens.

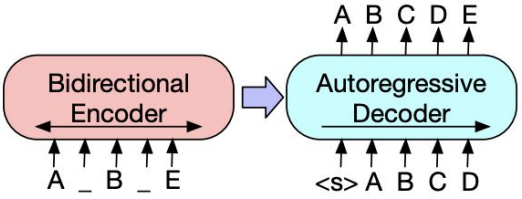
BART (Oct 2019)



(a) BERT: Random tokens are replaced with masks, and the document is encoded bidirectionally. Missing tokens are predicted independently, so BERT cannot easily be used for generation.



(b) GPT: Tokens are predicted auto-regressively, meaning GPT can be used for generation. However words can only condition on leftward context, so it cannot learn bidirectional interactions.



(c) BART: Inputs to the encoder need not be aligned with decoder outputs, allowing arbitary noise transformations. Here, a document has been corrupted by replacing spans of text with mask symbols. The corrupted document (left) is encoded with a bidirectional model, and then the likelihood of the original document (right) is calculated with an autoregressive decoder. For fine-tuning, an uncorrupted document is input to both the encoder and decoder, and we use representations from the final hidden state of the decoder.

Figure 1: A schematic comparison of BART with BERT (Devlin et al., 2019) and GPT (Radford et al., 2018).

Introduction

Pre-training has two stages: (1) text is corrupted with an arbitrary noising function, and (2) a sequence-to-sequence model is learned to reconstruct the original text. BART uses a standard Transformer based neural machine translation architecture.

Model
BART uses the standard sequence-to-sequence Transformer architecture from (Vaswani et al., 2017), except, following GPT, that we modify ReLU activation functions to GeLUs (Hendrycks & Gimpel, 2016) and initialise parameters from N(0, 0.02). For our base model, we use 6 layers in the encoder and decoder, and for our large model we use 12 layers in each.

Pre training
BART is trained by corrupting documents and then optimizing a reconstruction loss, i.e. the cross-entropy between the decoder's output and the original document. Unlike existing denoising autoencoders, which are tailored to specific noising schemes, BART allows us to apply any type of document corruption. In the extreme case, where all information about the source is lost, BART is equivalent to a language model.

The transformations we used are:
Token masking, token deletion (random tokens are deleted from the input. In contrast to token masking, the model must decide which positions are missing inputs), text infilling (a number of text spans are sampled, with span lengths drawn from a Poisson distribution ($\lambda = 3$). Each span is replaced with a single [MASK] token. 0-length spans correspond to the insertion of [MASK] tokens. Text infilling teaches the model to predict how many tokens are missing from a span), sentence permutation (a document is divided into sentences based on full stops, and these sentences are shuffled in a random order), document rotation (a token is chosen uniformly at random, and the document is rotated so that it begins with that token. This task trains the model to identify the start of the document).

T5 (Oct 2019)

Introduction
The basic idea underlying our work is to treat every text processing problem as a text-to-text problem, i.e. taking text as input and producing new text as output.
To specify which task the model should perform, we add a task-specific (text) prefix to the original input sequence before feeding it to the model.

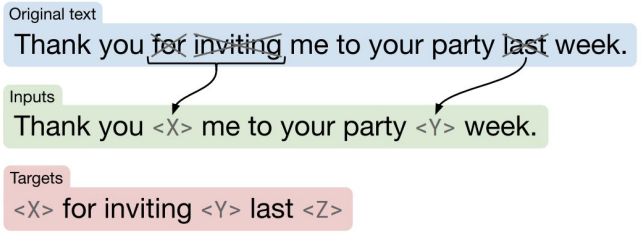
Base model
Due to its increasing ubiquity, all of the models we study are based on the Transformer architecture. We do not deviate significantly from this architecture as originally proposed.

While the original Transformer used a sinusoidal position signal or learned position embeddings, it has recently become more common to use relative position embeddings (Shaw et al., 2018; Huang et al., 2018a). Instead of using a fixed embedding for each position, relative position embeddings produce a different learned embedding according to the offset between the "key" and "query" being compared in the self-attention mechanism. We use a simplified form of position embeddings where each "embedding" is simply a scalar that is added to the corresponding logit used for computing the attention weights.

Encoder-only models like BERT (Devlin et al., 2018) are designed to produce a single prediction per input token or a single prediction for an entire input sequence. This makes them applicable for classification or span prediction tasks but not for generative tasks like translation or abstractive summarization.

Both the encoder and decoder consist of 12 blocks (each block comprising self-attention, optional encoder-decoder attention, and a feed-forward network). The feed-forward networks in each block consist of a dense layer with an output dimensionality of $d_{ff} = 3072$ followed by a ReLU nonlinearity and another dense layer. The key and value matrices of all attention mechanisms have an inner dimensionality of dkv = 64 and all attention mechanisms have 12 heads. All other sub-layers and embeddings have a dimensionality of $d_{model} = 768$. For regularization, we use a dropout probability of 0.1 everywhere dropout is applied in the model.
The other versions are: small, large, 3B and 11B.

Unsupervised objective



Schematic of the objective we use in our baseline model. In this example, we process the sentence “Thank you for inviting me to your party last week.” The words “for”, “inviting” and “last” (marked with an ×) are randomly chosen for corruption. Each consecutive span of corrupted tokens is replaced by a sentinel token (shown as <X> and <Y>) that is unique over the example. Since “for” and “inviting” occur consecutively, they are replaced by a single sentinel <X>. The output sequence then consists of the dropped-out spans, delimited by the sentinel tokens used to replace them in the input plus a final sentinel token <Z>.

We obtain additional speedup by specifically corrupting spans of tokens rather than corrupting individual tokens in an i.i.d. manner. To test this idea, we consider an objective that specifically corrupts contiguous, randomly-spaced spans of tokens. We use a mean span length of 3 and corrupt 15% of the original sequence.

Training and fine-tuning

We always train using standard maximum likelihood, i.e. using teacher forcing (Williams and Zipser, 1989) and a cross-entropy loss. For optimization, we use AdaFactor (Shazeer and Stern, 2018). At test time, we use greedy decoding (i.e. choosing the highest-probability logit at every timestep). We pre-train each model for 1M steps on C4 before fine-tuning. We use a maximum sequence length of 512 and a batch size of 2^{11} sequences.

Our models are fine-tuned for 2^{18} = 262,144 steps on all tasks. This value was chosen as a trade-off between the high-resource tasks (i.e. those with large data sets), which benefit from additional fine-tuning, and low-resource tasks (smaller data sets), which overfit quickly. During fine-tuning, we continue using batches with 128 length-512 sequences (i.e. 2^{16} tokens per batch). We use a constant learning rate of 0.001 when fine-tuning. We save a checkpoint every 5,000 steps and report results on the model checkpoint corresponding to the highest validation performance.

For tasks with long output sequences, we found improved performance from using beam search (Sutskever et al., 2014). Specifically, we use a beam width of 4 and a length penalty of α = 0.6 (Wu et al., 2016) for the WMT translation and CNN/DM summarization tasks.

Vocabulary

We use SentencePiece (Kudo and Richardson, 2018) to encode text as WordPiece tokens (Sennrich et al., 2015; Kudo, 2018). For all experiments, we use a vocabulary of 32,000 wordpieces.

PEGASUS (Dec 2019)

Model

PEGASUS-base had L = 12, H = 768, F = 3072, A = 12 and PEGASUS-large had L = 16, H = 1024, F = 4096, A = 16, where L denotes the number of layers for encoder and decoder (i.e. Transformer blocks), H for the hidden size, F for the feed-forward layer size and A for the number of self-attention heads. We pre-trained PEGASUS-base with a batch size of 256 and PEGASUS-large with a batch size of 8192.

We used sinusoidal positional encoding following Vaswani et al. (2017). For optimization, both pre-training and fine-tuning used Adafactor (Shazeer & Stern, 2018) with square root learning rate decay and dropout rate of 0.1. We used greedy-decoding and used beam-search with a length-penalty, α, as in Wu et al. (2016) for the final large model.
PEGASUS-base max input tokens: 512. PEGASUS-large max input tokens: 1024.

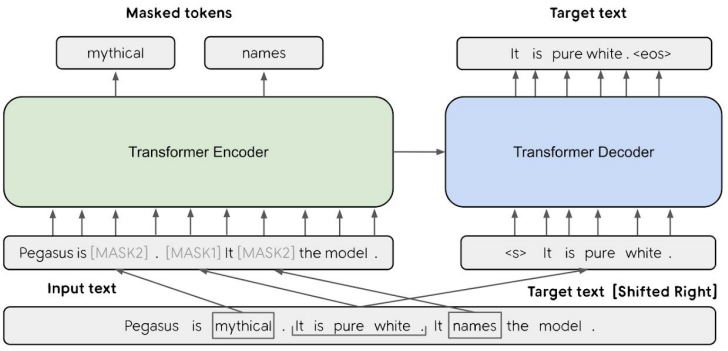


Figure 1: The base architecture of PEGASUS is a standard Transformer encoder-decoder. Both GSG and MLM are applied simultaneously to this example as pre-training objectives. Originally there are three sentences. One sentence is masked with `[MASK1]` and used as target generation text (GSG). The other two sentences remain in the input, but some tokens are randomly masked by `[MASK2]` (MLM).

Unsupervised objective

In contrast to MASS, UniLM, BART and T5, the proposed PEGASUS masks multiple whole sentences rather than smaller continuous text spans.

Inspired by recent success in masking words and contiguous spans (Joshi et al., 2019; Raffel et al., 2019), we select and mask whole sentences from documents, and concatenate the gap-sentences into a pseudo-summary. The corresponding position of each selected gap sentence is replaced by a mask token [MASK1] to inform the model. Gap sentences ratio, or GSR, refers to the number of selected gap sentences to the total number of sentences in the document. When scaling up to PEGASUS-large, we chose an effective GSR of 30%.

We consider 3 primary strategies for selecting m gap sentences without replacement from a document: random, lead and principal. The principal method selects top-m scored sentences according to importance. As a proxy for importance we compute ROUGE1-F1 (Lin, 2004) between the sentence and the rest of the document, $s_i = rouge(x_i, D \setminus \{x_i\}), \quad \forall i$. s_i = rouge(x_i, D \setminus \{x_i\}), \quad \forall i.

In this formulation sentences are scored independently (Ind) and the top m selected. We also consider selecting them sequentially (Seq) as in Nallapati et al. (2017) by greedily maximizing the ROUGE1-F1 between selected sentences, $S\{x_i\}$, and remaining sentences, $D \setminus (S\{x_i\})$.

When calculating ROUGE1-F1, we also consider n-grams as a set (Uniq) instead of double-counting identical n-grams as in the original implementation (Orig). This results in four variants of the principal sentence selection strategy, choosing Ind/Seq and Orig/Uniq options.

Following BERT, we select 15% tokens in the input text, and the selected tokens are (1) 80% of time replaced by a mask token [MASK2], or (2) 10% of time replaced by a random token, or (3) 10% of time unchanged. We apply MLM to train the Transformer encoder as the sole pre-training objective or along with GSG. However, we found that MLM does not improve downstream tasks at large number of pre-training steps, and chose not to include MLM in the final model PEGASUS-large. Ind-Orig and Seq-Uniq were consistently better (or similar) than Random and Lead. The results suggest choosing principal sentences works best for downstream summarization tasks, and we chose Ind-Orig for the PEGASUS-large.

Training

For pre-training we considered two large text corpora: C4 (350M web pages) and HugeNews (1.5B articles). For downstream summarization, we only used public abstractive summarization datasets, and access them through TensorFlow Summarization Datasets. We used train/validation/test ratio of 80/10/10 if no split was provided, and 10% train split as validation if there was no validation split.

Pre-training on HugeNews was more effective than C4 on the two news downstream datasets, while the non-news informal datasets (WikiHow and Reddit TIFU) prefer the pre-training on C4. This suggests pre-training models transfer more effectively to downstream tasks when their domains are aligned better.

Zero and low-resource summarization

In real-world practice, it is often difficult to collect a large number of supervised examples to train or fine-tune a summarization model. To simulate the low-resource summarization setting, we picked the first 10k (k = 1, 2, 3, 4) training examples from each dataset to fine-tune PEGASUS-large (HugeNews). We fine-tuned the models up to 2000 steps with batch size 256, learning rate 0.0005, and picked the checkpoint with best validation performance.

PEGASUS-large (mixed,stochastic), ***present on HuggingFace***

The PEGASUS-large (mixed,stochastic) model includes the changes: (1) The model was pre-trained on the mixture of C4 and HugeNews weighted by their number of examples. (2) The model dynamically chose gap sentences ratio uniformly between 15%-45%. (3) Importance sentences were stochastically sampled with 20% uniform noise on their scores. (4) The model was pre-trained for 1.5M steps instead of 500k steps, as we observed slower convergence of pre-training perplexity. (5) The SentencePiece tokenizer was updated to encode the newline character.

PROPHETNET (Jan 2020)

Our ProphetNet is based on Transformer encoder-decoder architecture. There are two goals when designing ProphetNet: (a) the model should be able to simultaneously predict the future n-gram at each time step in an efficient way during the training phase, and (b) the model can be easily converted to predict the next token only as original Seq2Seq model for inference or fine-tuning phase.

Future N-gram prediction

ProphetNet mainly changes the original Seq2Seq optimization of predicting next single token as $p(y_t|y_{<t}, x)$ into $p(y_{t:t+n-1}|y_{<t}, x)$ at each time step t, where $y_{t:t+n-1}$ denotes the next continuous n future tokens. In other words, the next n future tokens are predicted simultaneously.

Based on Transformer Seq2Seq architecture, ProphetNet contains a multi-layer Transformer encoder with the multi-head self-attention mechanism and a multi-layer Transformer decoder with the proposed multi-head n-stream self-attention mechanism. Given a source sequence $x = (x_1, ..., x_M)$, ProphetNet encodes the x into a sequence representation, which is the same as the original Transformer encoder:

$$H_{enc} = \text{Encoder}(x_1, ..., x_M)$$

H_{enc} = \text{Encoder}(x_1, ... , x_M)

where H_{enc} denotes the source sequence representations.

On the decoder side, instead of predicting only the next token at each time step like the original Transformer decoder, ProphetNet decoder predicts n future tokens simultaneously as we mentioned above:

$$p(y_t|y_{<t}, x), , p(y_{t+n-1}|y_{<t}, x) = \text{Decoder}(y_{<t}, H_{enc}) \quad 2 \leq n \leq N$$

p(y_t | y_{<t}, x), …, p(y_{t+n-1} | y_{<t}, x) = \text{Decoder}(y_{<t}, H_{enc})\quad 2 \leq n \leq N

where the decoder outputs N probability at each time step. The future n-gram prediction objective can be further formalized as

$$\mathcal{L} = -\sum_{n=0}^{N-1} \alpha_n \cdot \left(\sum_{t=1}^{T-n} log p_\Theta(y_{t+n}|y_{<t}, x) \right)$$

\mathcal{L} = -\sum_{n=0}^{N-1} \alpha_n \cdot \left(\sum_{t=1}^{T-n} log p_{\Theta}(y_{t+n} | y_{<t}, x) \right)

The above future n-gram prediction objective can be seen to consist of two parts: (a) the conditional LM loss which is the same as the original teacher forcing, and (b) the N−1 future token prediction losses which force the model to predict the future target tokens. The future n-gram prediction loss explicitly encourages the model to plan for future token prediction and prevent overfitting on strong local correlations. Furthermore, we assign the different weights α_n to each loss as the trade-off between the traditional language modeling and future n-gram prediction.

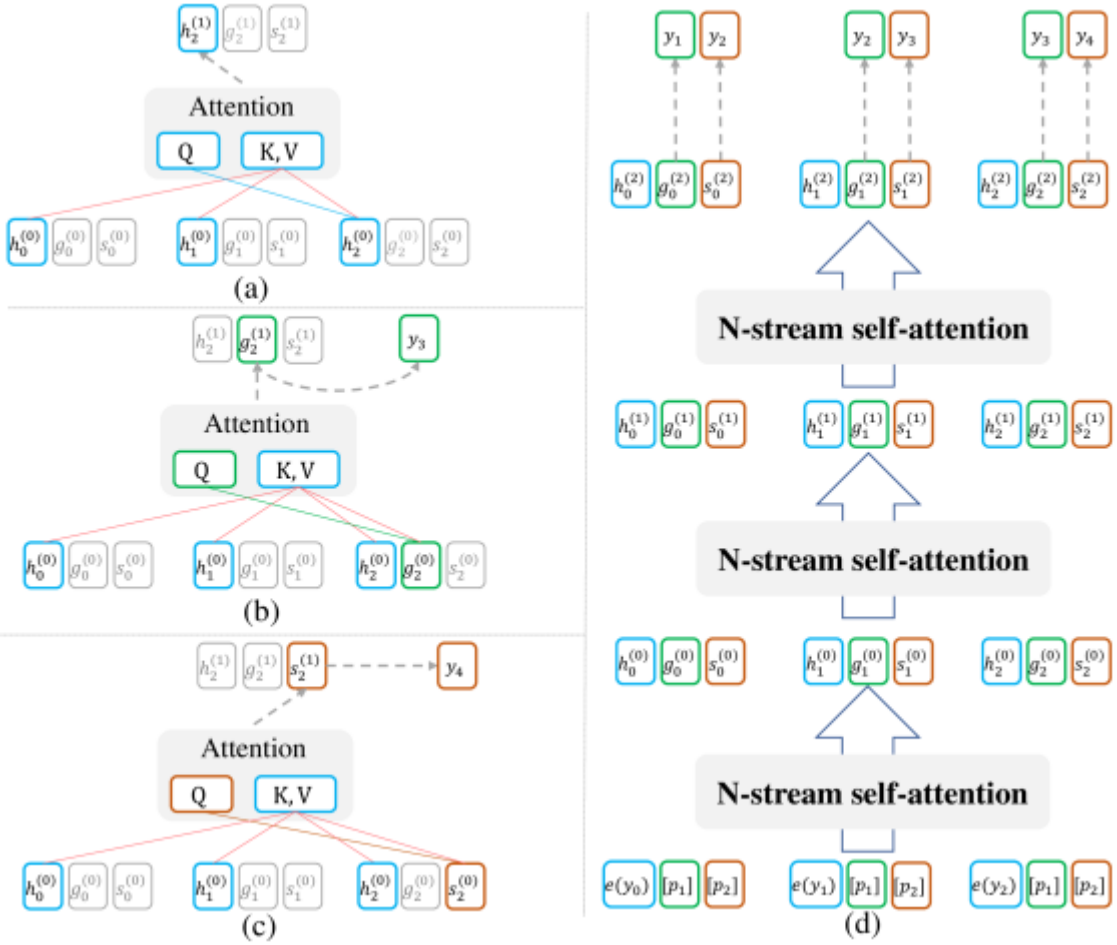


Figure: N-stream self-attention mechanism which contains a main stream self-attention and n predicting stream self-attention. For simplicity sake, we take 2-stream self-attention (n = 2) as an example here. Figure (a) presents the attention process of the main stream self-attention. Figure (b) and Figure (c) show the attention process of 1-st predicting stream and 2-nd predicting stream, respectively. Figure (d) shows the inputs, outputs, and the whole multi-layer n-stream self-attention.

Pre-training

We only consider token span masking which is the same as the MASS. We mask out some token spans of the original text as the encoder input, and the model learns to recover the masked tokens. Besides, unlike MASS learns to recover one next token at each time step, ProphetNet learns to recover the next n future tokens within each masked token span.

We pre-train the ProphetNet which contains 12-layer encoder and 12-layer decoder with 1024 embedding/hidden size and 4096 feed-forward filter size. The batch size and training steps are set to 1024 and 500K, respectively. We use Adam optimizer (Kingma & Ba, 2015) with a learning rate of 3 E−4 for pre-training. Considering the training cost, we set the n to be 2 for ProphetNet in the following experiments.

The input length of ProphetNet is set to 512.

Fine-tuning

During inference, we limit the length of the output to between 45 and 110 tokens with 1.2 length penalty. We set beam size to 5 and remove the duplicated trigrams in beam search (Fan et al., 2017).

Fine-tuned on summarization and **question generation**.

Hierarchical Transformers for Multi-Doc Summarization (May 2019)

Aside from the difficulties in obtaining training data, a major obstacle to the application of end-to-end models to multi-document summarization is the sheer size and number of source documents which can be very large. As a result, it is practically infeasible (given memory limitations of current hardware) to train a model which encodes all of them into vectors and subsequently generates a summary from them. Liu et al. (2018) propose a two-stage architecture, where an extractive model first selects a subset of salient passages, and subsequently an abstractive model generates the summary while conditioning on the extracted subset. Although the model of Liu et al. (2018) takes an important first step towards abstractive multi-document summarization, it still considers the multiple input documents as a concatenated flat sequence, being agnostic of the hierarchical structures and the relations that might exist among documents.

Training Data

The input to a hypothetical system is the title of a Wikipedia article and a collection of source documents, while the output is the Wikipedia article's first section. Source documents are web pages cited in the References section of the Wikipedia article and the top 10 search results returned by Google (with the title of the article as the query). Since source documents could be relatively long, they are split into multiple paragraphs by line-breaks. More formally, given title T, and L input paragraphs $\{P_1, ..., P_L\}$ (retrieved from Wikipedia citations and a search engine), the task is to generate the lead section D of the Wikipedia article.

Since the input paragraphs are numerous and possibly lengthy, instead of directly applying an abstractive system, we first rank them and summarize the L'-best ones. Our summarizer follows the very successful encoder-decoder architecture.

On average, each input has 525 paragraphs, and each paragraph has 70.1 tokens. The average length of the target summary is 139.4 tokens.

Model

Instead of treating the selected paragraphs as a very long sequence, we develop a hierarchical model based on the Transformer architecture (Vaswani et al., 2017) to capture inter-paragraph relations.

For both ranking and summarization stages, we encode source paragraphs and target summaries using subword tokenization with SentencePiece.

During decoding we use beam search with beam size 5 and length penalty with α = 0.4 (Wu et al., 2016); we decode until an end-of-sequence token is reached.

Drawbacks
The architecture is not present on HuggingFace. The code provided on GitHub is for training and should be studied and modified to obtain a generative model.

REFORMER (Jan 2020)

Introduction
We introduce the Reformer model which solves these problems using the following techniques:

- Reversible layers, first introduced in Gomez et al. (2017), enable storing only a single copy of activations in the whole model.
- Splitting activations inside feed-forward layers and processing them in chunks
- Approximate attention computation based on locality-sensitive hashing.

Splitting activations in fact only affects the implementation; it is numerically identical to the layers used in the Transformer. Applying reversible residuals instead of the standard ones does change the model but has a negligible effect on training in all configurations we experimented with. Finally, locality-sensitive hashing in attention is a more major change that can influence the training dynamics, depending on the number of concurrent hashes used. We study this parameter and find a value which is both efficient to use and yields results very close to full attention.

Model
Memory-efficient attention: it is important to note that the QK^T matrix does not need to be fully materialized in memory. The attention can indeed be computed for each query q_i separately, only calculating

$$\text{softmax}\left(\frac{q_i K^T}{\sqrt{d_k}}\right)$$

\text{softmax}\left(\frac{q_iK^T}{\sqrt{d_k}}\right)

once in memory, and then re-computing it on the backward pass when needed for d_k gradients. This way of computing attention may be less efficient but it only uses memory proportional to length. We use this memory-efficient implementation of attention to run the full-attention baselines presented in the experimental section.

Shared-QK: for models with LSH attention, we want queries and keys (Q and K) to be identical. This is easily achieved by using the same linear layer to go from A to Q and K, and a separate one for V. We call a model that behaves like this a shared-QK

Transformer. It turns out that sharing QK does not affect the performance of Transformer.

Locality sensitive hashing: As already mentioned, the main issue is the term QK^T. But note that we are actually only interested in softmax(QK^T). Since softmax is dominated by the largest elements, for each query q_i we only need to focus on the keys in K that are closest to q_i. For example, if K is of length 64K, for each q_i we could only consider a small subset of, say, the 32 or 64 closest keys. That is much more efficient, but how can we find the nearest neighbors among the keys? This problem can be solved by locality-sensitive hashing (LSH).

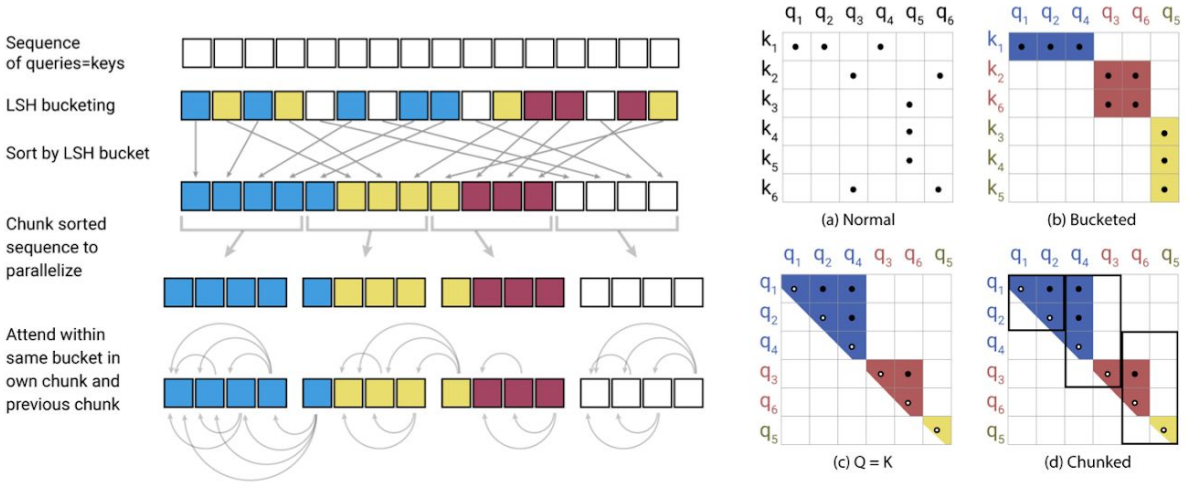


Figure 2: Simplified depiction of LSH Attention showing the hash-bucketing, sorting, and chunking steps and the resulting causal attentions. (a-d) Attention matrices for these varieties of attention.

Figure 2(a-b) shows a schematic comparison of full-attention with a hashed variant. Part (a) depicts that the attention matrix for full attention is typically sparse, but the computation does not take advantage of this sparsity. In (b), the queries and keys have been sorted according to their hash bucket. Since similar items fall in the same bucket with high probability, the full attention pattern can be approximated by only allowing attention within each bucket. In the sorted attention matrix, pairs from the same bucket will cluster near the diagonal (as depicted in Figure 2c). We can follow a batching approach where chunks of m consecutive queries (after sorting) attend to each other, and one chunk back (Figure 2d).

LONGFORMER (Apr 2020)

Very difficult to understand, will spend more time on this.

LINFORMER (Jul 2020)

Prior work has proposed several techniques for improving the efficiency of self-attention. One popular technique is introducing sparsity into attention layers (Child et al., 2019; Qiu et al., 2019; **Beltagy et al., 2020 (Longformer)**) by having each token attend to only a subset of tokens in the whole sequence. This reduces the overall complexity of the attention mechanism to $O(n\sqrt{n})$ (Child et al., 2019). However, as shown in Qiu et al. (2019), this approach suffers from a large performance drop with limited efficiency gains, i.e., a 2% drop with only 20% speed up.

More recently, the **Reformer** (Kitaev et al., 2020) used locally-sensitive hashing (LSH) to reduce the self-attention complexity to O(n log(n)). However, in practice, the Reformer's efficiency gains only appear on sequences with length > 2048 (Figure 5 in Kitaev et al. (2020)). Furthermore, the Reformer's multi-round hashing approach actually increases the number of sequential operations, which further undermines their final efficiency gains.

Our approach is inspired by the key observation that self-attention is low rank. More precisely, we show both theoretically and empirically that the stochastic matrix formed by self-attention can be approximated by a low-rank matrix. Empowered by this observation, we introduce a novel mechanism that reduces self-attention to an O(n) operation in both space- and time-complexity.

Model

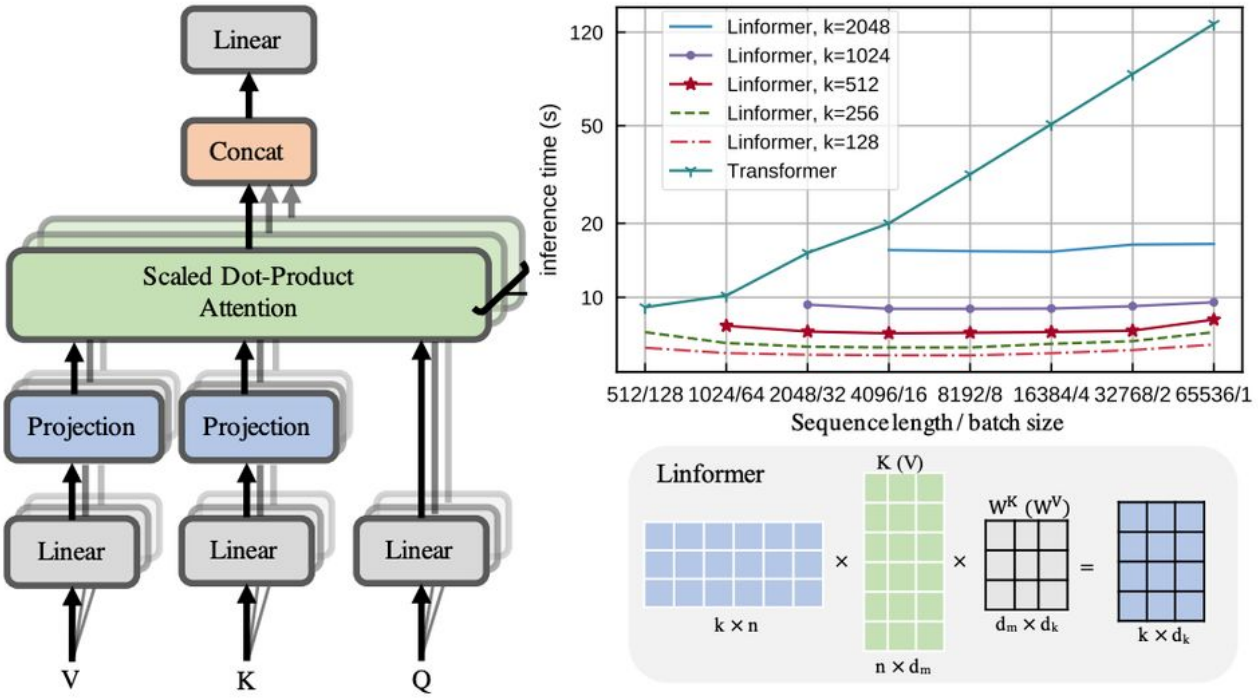


Figure 2: Left and bottom-right show architecture and example of our proposed multihead linear self-attention. Top right shows inference time vs. sequence length for various Linformer models.

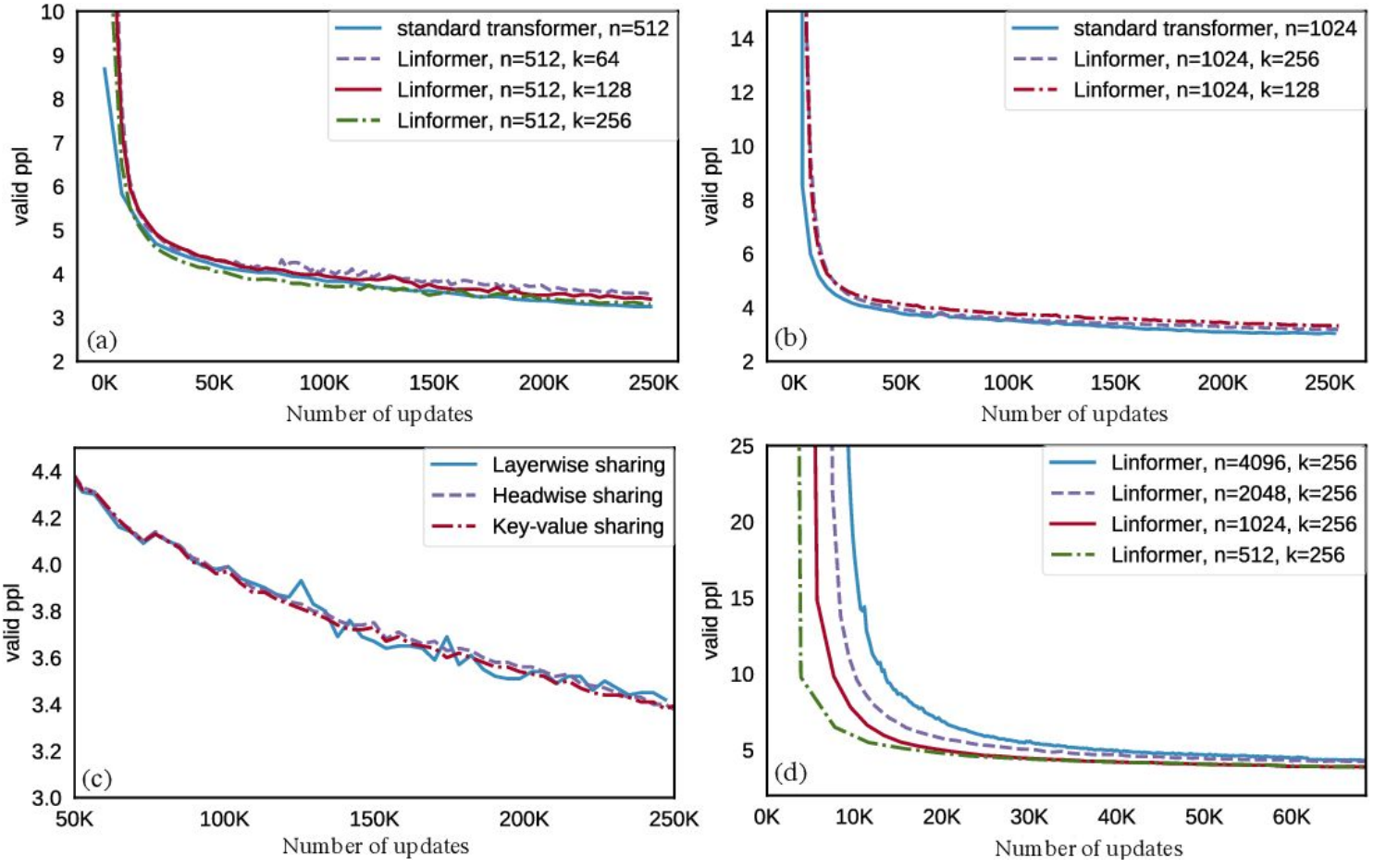
The main idea of our proposed linear self-attention (Figure 2) is to add two linear projection matrices $E_i, F_i \in \mathbb{R}^{nxk}$ when computing key and value. We first project the original (n x d)-dimensional key and value layers KW_i^K and VW_i^V into (k x d)-dimensional projected key and value layers. We then compute an (n x k)-dimensional context mapping matrix P using scaled dot-product attention.

$$head_i = \text{Attention}(QW_i^Q, E_i KW_i^K, F_i VW_i^V) = \text{softmax}\left(\frac{QW_i^Q (E_i KW_i^K)^T}{\sqrt{d_k}}\right) \cdot F_i VW_i^V$$

head_i = \text{Attention}(QW_i^Q, E_iKW_i^K, F_iVW_i^V) = \text{softmax}\left(\frac{QW_i^Q(E_iKW_i^K)^T}{\sqrt{d_k}} \right) \cdot F_iVW_i^V

Finally, we compute context embeddings for each headi using $P \cdot (F_i VW_i^V)$. Note the above operations only require O(nk) time and space complexity. Thus, if we can choose a very small projected dimension k, such that k << n, then we can significantly reduce the memory and space consumption.

Perplexity Results



BIGBIRD (Jul 2020)

Introduction

While we know that self-attention and Transformers are useful, our theoretical understanding is rudimentary. What aspects of the self-attention model are necessary for its performance? What can we say about the expressivity of Transformers and similar models? Apriori, it was not even clear from the design if the proposed self-attention mechanism was as effective as RNNs. For example, the self-attention does not even obey sequence order as it is permutation equivariant. Can we achieve the empirical benefits of a fully quadratic self-attention scheme using fewer inner-products? Do these sparse attention mechanisms preserve the expressivity and flexibility of the original network?

Architecture

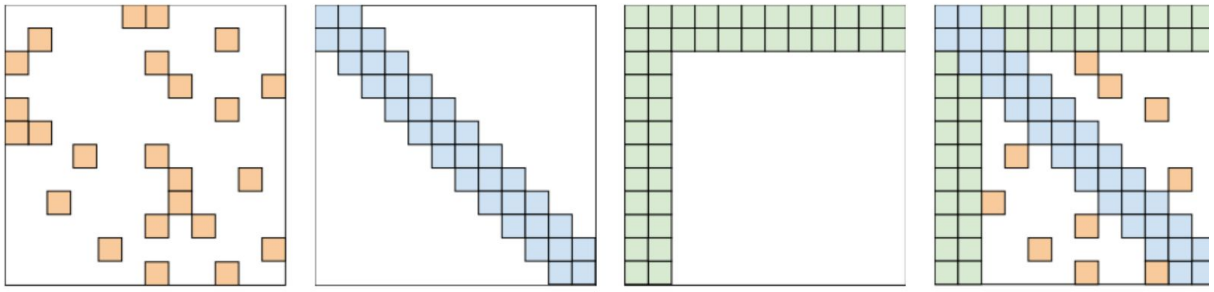
In this section, we describe the BigBird model using the generalised attention mechanism that is used in each layer of transformer operating on an input sequence $X = (x_1, ..., x_n) \in \mathbb{R}^{nxd}$. The generalized attention mechanism is described by a directed graph D whose vertex set is $[n] = \{1, ..., n\}$. The set of arcs (directed edges) represent the set of inner products that the attention mechanism will consider. Let $N(i)$ denote the out-neighbors set of node i in D, then the ith output vector of the generalized attention mechanism is defined as

$$\text{ATTN}_D(X_i) = x_i + \sum_{h=1}^{H} \sigma\left(Q_h(x_i) K_h(X_{N(i)})^T\right) \cdot V_h(X_{N(i)})$$

\text{ATTN}_D(X_i) = x_i + \sum_{h = 1}^H \sigma \left(Q_h(x_i)K_h(X_{N(i)})^T \right) \cdot V_h(X_{N(i)})

where $Q_h, K_h : \mathbb{R}^d \rightarrow \mathbb{R}^m$ Q_h, K_h : \mathbb{R}^d \rightarrow \mathbb{R}^m are query and key functions respectively, $V_h : \mathbb{R}^d \rightarrow \mathbb{R}^d$ V_h : \mathbb{R}^d \rightarrow \mathbb{R}^d is a value function, σ is a scoring function (e.g. softmax or hardmax) and H denotes the number of heads. Also note $X_{N(i)}$ corresponds to the matrix formed by only stacking $\{x_j : j \in N(i)\}$ and not all the inputs.

If D is the complete digraph, we recover the full quadratic attention mechanism of Vaswani et al. To simplify our exposition, we will operate on the adjacency matrix A of the graph D. To elaborate, $A \in [0, 1]^{nn}$ with $A(i, j) = 1$ if query i attends to key j and is zero otherwise. For example, when A is the ones matrix (as in BERT), it leads to quadratic complexity, since all tokens attend on every other token. This view of self-attention as a fully connected graph allows us to exploit existing graph theory to help reduce its complexity.



(a) Random attention (b) Window attention (c) Global Attention (d) BIGBIRD

We believe sparse random graphs for attention mechanism should have two desiderata: small average path length between nodes and a notion of locality. Let us consider the simplest random graph construction, In such a random graph the shortest path between any two nodes is logarithmic in the number of nodes. Thus, we propose a sparse attention where each query attends over r random number of keys. Mathematically this means that $A(i, \cdot) = 1$ for r randomly chosen keys (see Fig. 1a). To capture the local structures in the context, in BigBird, we define a sliding window attention, so that during self attention of width w, query at location i attends from $i - w/2$ to $i + w/2$ keys. In our notation, $A(i, i - w/2 : i + w/2) = 1$ (see Fig. 1b). The final piece of BigBird is inspired from our theoretical analysis (Sec. 3), which is critical for empirical performance. More specifically, our theory utilizes the importance of "global tokens" (tokens that attend to all tokens in the sequence and to whom all tokens attend to (see Fig. 1c).

The final attention mechanism for BigBird (Fig. 1d) has all three of these properties: queries attend to r random keys, each query attends to w/2 tokens to the left of its location and w/2 to the right of its location and they contain g global tokens (The global tokens can be from existing tokens or extra added tokens).

Encoder-Decoder Setup

For an encoder-decoder setup, one can easily see that both suffer from quadratic complexity due to the full self attention. We focus on introducing the sparse attention mechanism of BigBird only at the encoder side. This is because, in practical generative applications, the length of output sequence is typically small as compared to the input. For summarization, it is more efficient to use sparse attention mechanism for the encoder and full self-attention for the decoder.

Summarization

In this paper we focus on abstractive summarization of long documents where using a longer contextual encoder should improve performance. The reasons are twofold: First, the salient content can be evenly distributed in the long document, not just in the first 512 tokens. Second, longer documents exhibit a richer discourse structure and summaries are considerably more abstractive, thereby observing more context helps. As has been pointed out recently, pre-training helps in generative tasks, we warm start from our general purpose MLM pre-training on base-sized models as well as utilizing state-of-the-art summarization specific pre-training from Pegasus on large-sized models. Modeling longer context brings significant improvement. Along with hyperparameters, we also present results on shorter but more widespread

datasets in App. E.4, which shows that using sparse attention does not hamper performance either.

Parameter		Base: BIGBIRD-RoBERTa	Large: BIGBIRD-Pegasus
Block length, b		64	64
Global token location		ITC	ITC
# of global token, g		$2 \times b$	$2 \times b$
Window length, w		$3 \times b$	$3 \times b$
# of random token, r		$3 \times b$	$3 \times b$
Max. encoder sequence length	BBC-XSUM:	1024	1024
	CNN/DM:	2048	2048
	Others:	3072	3072
Max. decoder sequence length	BBC-XSUM:	64	64
	CNN/DM:	128	128
	Others:	256	256
Beam size		5	5
Length penalty	BBC-XSUM:	0.7	0.7
	Others:	0.8	0.8
# of heads		12	16
# of hidden layers		12	16
Hidden layer size		768	1024
Batch size		128	128
Loss		teacher forced cross-entropy	teacher forced cross-entropy
Activation layer		gelu	gelu
Dropout prob		0.1	0.1
Attention dropout prob		0.1	0.1
Optimizer		Adam	Adafactor
Learning rate		1×10^{-5}	1×10^{-4}
Compute resources		4×4 TPUv3	4×8 TPUv3

Table 18: Encoder hyperparameters for Summarization. We use full attention in decoder

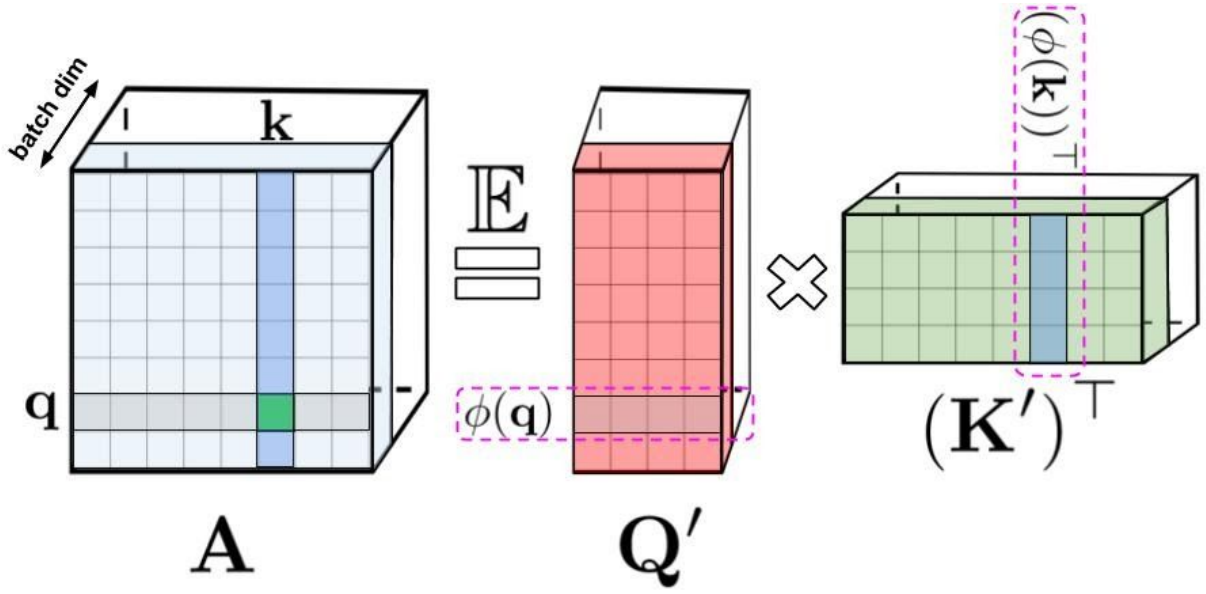
PERFORMER (Sep 2020)

Introduction

We introduce the first Transformer architectures, Performers, capable of provably accurate and practical estimation of regular (softmax) full-rank attention, but of only linear space and time complexity and not relying on any priors such as sparsity or low-rankness. Performers use the Fast Attention Via positive Orthogonal Random features (FAVOR+) mechanism, leveraging new methods for approximating softmax and Gaussian kernels, which we propose. We believe these methods are of independent interest, contributing to the theory of scalable kernel methods. Consequently, Performers are the first linear architectures fully compatible (via small amounts of fine-tuning) with regular Transformers, providing strong theoretical guarantees: unbiased or nearly-unbiased estimation of the attention matrix, uniform convergence and lower variance of the approximation.

Architecture

In the original attention mechanism, the query and key inputs, corresponding respectively to rows and columns of a matrix, are multiplied together and passed through a softmax operation to form an attention matrix, which stores the similarity scores. Note that in this method, one cannot decompose the query-key product back into its original query and key components after passing it into the nonlinear softmax operation. However, it is possible to decompose the attention matrix back to a product of random nonlinear functions of the original queries and keys, otherwise known as random features, which allows one to encode the similarity information in a more efficient manner.

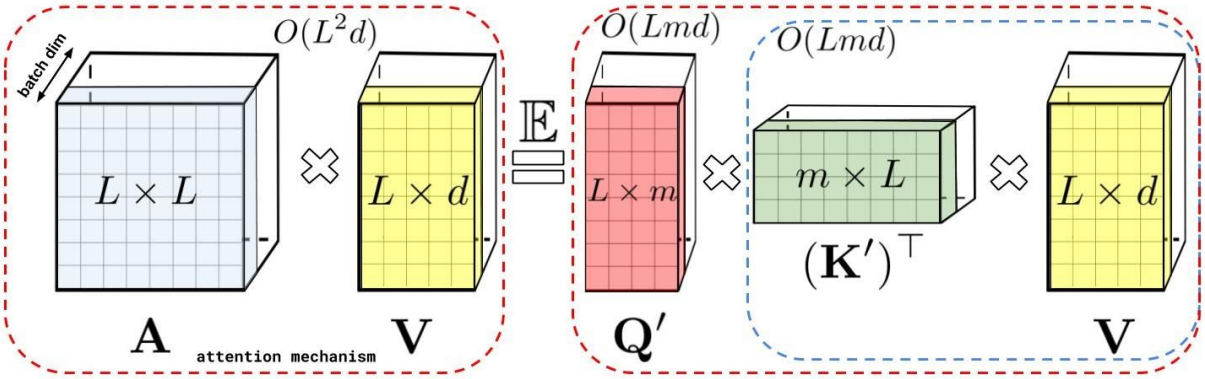


LHS: The standard attention matrix, which contains all similarity scores for every pair of entries, formed by a softmax operation on the query and keys, denoted by **q** and **k**. **RHS:** The standard attention matrix can be approximated via lower-rank randomized matrices **Q′** and **K′** with rows encoding potentially randomized nonlinear functions of the original queries/keys. For the regular softmax-attention, the transformation is very compact and involves an exponential function as well as random Gaussian projections.

Regular softmax-attention can be seen as a special case with these nonlinear functions defined by exponential functions and Gaussian projections. Note that we can also reason inversely, by implementing more general nonlinear functions first, implicitly defining other types of similarity measures, or kernels, on the query-key product. We frame this as generalized attention, based on earlier work in kernel methods. Although for most kernels, closed-form formulae do not exist, our mechanism can still be applied since it does not rely on them.

The decomposition described above allows one to store the implicit attention matrix with linear, rather than quadratic, memory complexity. One can also obtain a *linear time* attention mechanism using this decomposition. While the original attention mechanism multiplies the stored attention matrix with the *value* input to obtain the final result, after decomposing the attention matrix, one can rearrange matrix

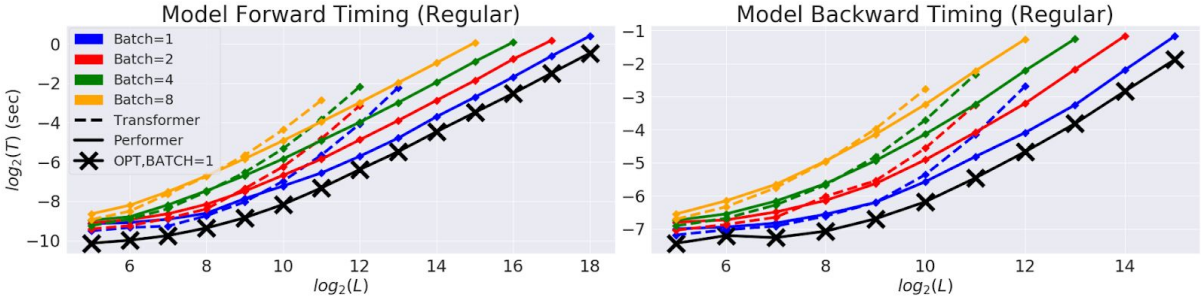
multiplications to approximate the result of the regular attention mechanism, without explicitly constructing the quadratic-sized attention matrix. This ultimately leads to *FAVOR+*.



Left: Standard attention module computation, where the final desired result is computed by performing a matrix multiplication with the attention matrix **A** and value tensor **V**. **Right:** By decoupling matrices **Q′** and **K′** used in lower rank decomposition of **A** and conducting matrix multiplications in the order indicated by dashed-boxes, we obtain a linear attention mechanism, never explicitly constructing **A** or its approximation.

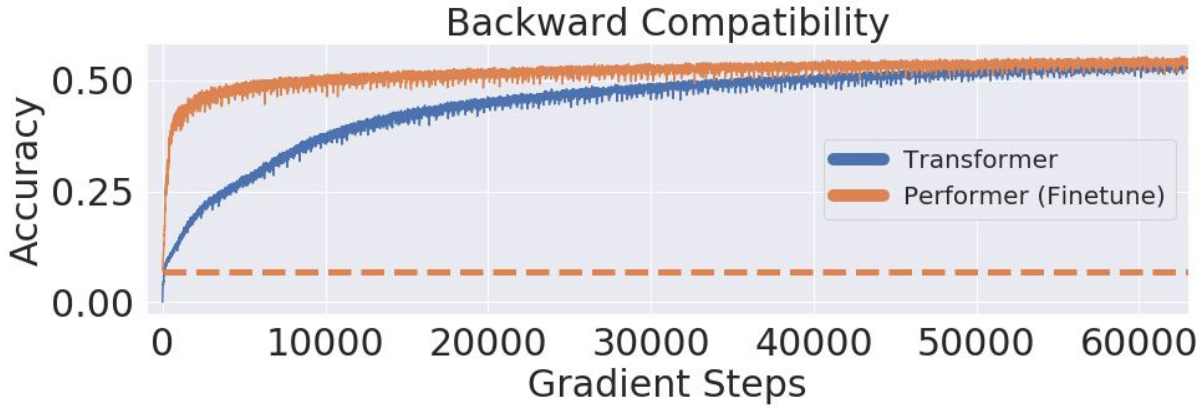
Results

We first benchmark the space- and time-complexity of the Performer and show that the attention speedups and memory reductions are empirically nearly optimal, i.e., very close to simply not using an attention mechanism at all in the model.



Bidirectional timing for the regular Transformer model in log-log plot with time (T) and length (L). Lines end at the limit of GPU memory. The black line (X) denotes the maximum possible memory compression and speedups when using a “dummy” attention block, which essentially bypasses attention calculations and demonstrates the maximum possible efficiency of the model. The Performer model is nearly able to reach this optimal performance in the attention component.

We further show that the Performer, using our unbiased softmax approximation, is backwards compatible with pretrained Transformer models after a bit of fine-tuning, which could potentially lower energy costs by improving inference speed, without having to fully retrain pre-existing models.


Backward Compatibility
Accuracy
0.50
0.25
0.00
0
10000
20000
30000
40000
50000
60000
Gradient Steps
Transformer
Performer (Finetune)