# Searching for Efficient Transformers for Language Modeling

**Anonymous Author(s)**
Affiliation
Address
email

## Abstract

Large Transformer language models have been central to many recent breakthroughs in natural language processing. The training and inference costs of these models, however, have grown rapidly and become prohibitively expensive. Here we aim to reduce the costs of Transformer models by searching for a more efficient alternative. In contrast to previous architecture search efforts that utilize high-level building blocks, we perform a granular search over low-level primitives that define a Transformer TensorFlow program. We identify an architecture, named Primer, that has a much lower training cost than the original Transformer and other variants for auto-regressive language modeling. Primer's improvements can be attributed to two simple modifications: squaring ReLU activations and adding depthwise convolution to attention multi-head projections. Our experiments show that Primer's gains over Transformer increase as compute scale increases and follow a power law function at optimal parameter sizes. As an example, a 500M parameter Primer can achieve the same auto-regressive language modeling performance as the original T5 architecture with 4.2X less compute.

## 1 Introduction

Transformers [1] have been key to many breakthroughs in NLP over the past few years (e.g., [2, 3, 4, 5, 6, 7]). Much of this success is also due to scale, given that unprecedented performance can be reached by simply increasing dataset size, model size and training compute budget [3, 7, 8]. For example, simply increasing Transformer model sizes has been shown to improve traditional NLP metrics in a predictable way, with models not yet reaching the theoretical ceiling [9]. However, the high training costs of these models make it difficult to push these results further.

Here we aim to reduce the training costs of Transformer language models. To this end, we propose searching for more efficient alternatives to Transformer by modifying its TensorFlow (TF) [10] computation graph. We mainly focus on decoder-only auto-regressive language modeling (LM), given its generality and success [11, 7, 12, 13, 14]. Given a search space of TensorFlow programs, we use evolution [15, 16, 17, 18, 19] to search for models that achieve as low of a validation loss as possible given fixed limited training time. An advantage of using TensorFlow programs as the search space is that it is easier to find simple low-level improvements to optimize Transformers.

The discovered model, Primer, exhibits strong performance improvements over common Transformer variants on auto-regressive language modeling. Our experiments show that Primer has the combined benefits of (1) achieving a target quality using less training cost, (2) achieving higher quality given a fixed training cost, and (3) achieving a target quality using less inference cost. The benefits are robust and hold across model sizes ($20M$ to $550M$ parameters), across compute scales (10 to $10^4$ accelerator hours), across datasets (LM1B, C4, PG19), across hardware platforms (TPUv2, TPUv3 and V100), across multiple Transformer codebases using default configurations (Tensor2Tensor

Submitted to 35th Conference on Neural Information Processing Systems (NeurIPS 2021). Do not distribute.

written in TensorFlow [10] and T5 in Mesh-TensorFlow [20]) and across multiple model families (dense Transformers [1], sparse mixture-of-experts Switch Transformers [8], and Synthesizers [21]).

Our main result is that the gain of Primer over Transformers increases as training cost increases, when controlling for model size. This gain also follows a power law function when using optimally sized models. In our largest experiment, we compare 500$M$ parameter Primer to the original T5 architecture, using the exact configuration used by Raffel et al. [5] applied to auto-regressive language modeling. In this large compute setting, Primer achieves an improvement of 0.9 PPLX given the same training cost, and reaches quality parity with the T5 baseline model using 4.2X less compute.

Our analysis shows that the improvements of Primer over Transformer can be attributed to two main modifications: squaring ReLU activations and adding depthwise convolution to attention multi-head projections. These two modifications are simple and can be easily dropped into existing Transformer codebases to obtain significant gains for auto-regressive language modeling.

## 2 Search Space and Search Method

**Searching Over TensorFlow Programs:** To construct a search space for Transformer alternatives, we use operations from TensorFlow (TF). In this search space, each program defines the stackable decoder block of an *auto-regressive language model*. Given input tensors $X \in \mathbb{R}^{n \times d}$ that represent sequences of length $n$ with embedding length $d$, our programs return tensors of the same shape. When stacked, their outputs represent next-token prediction embeddings at each sequence position. Our programs only specify model architectures and nothing else. In other words, the input and output embedding matrices themselves, as well as input preprocessing and weight optimization are not within the scope of our programs.

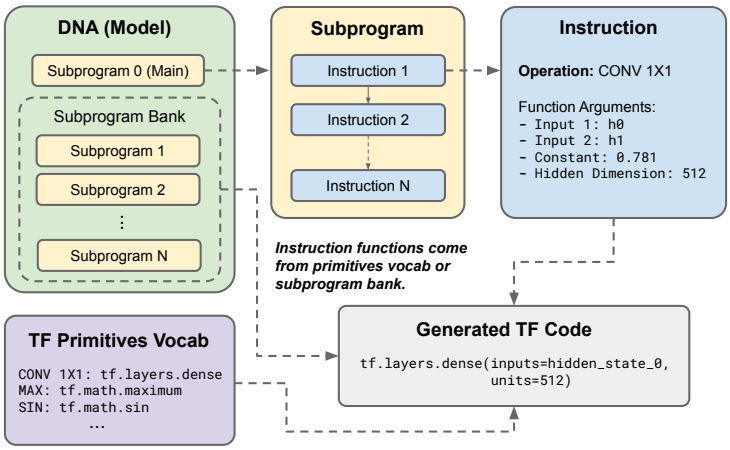



Figure 1: Overview of DNAs that define a decoder model program (i.e., an auto-regressive language model). Each DNA has a collection of subprograms, where SUBPROGRAM 0 is the MAIN() function entry point. Each subprogram is comprised of instructions, which are converted to lines of TensorFlow code. Instruction operations map to either basic TensorFlow library functions from the primitives vocabulary or one of the parent DNA's subprograms. The operation's arguments are filled using the parent instruction's argument set, which contains values for all potential operation arguments; arguments that are not used by a particular operation are simply ignored.

We now describe how programs are constructed in our search space (Figure 1). Each program is built from a evolutionary search *DNA*, which is an indexed collection of *subprograms*. SUBPROGRAM 0 is the MAIN() function that is the execution entry point, and the other subprograms are part of the DNA's *subprogram bank*. Each subprogram is an indexed array of *instructions* with no length constraints. An instruction is an *operation* with a set of input *arguments*. Operations denote the function that the instruction executes. Each operation maps to either a TF function from the *primitives vocabulary* or another subprogram in the DNA subprogram bank. The primitives vocabulary is comprised of simple primitive TF functions, such as ADD, LOG, and MATMUL. It is worth emphasizing that high-level build-

ing blocks such as self-attention are not an operation in the search space, but can be constructed from other low-level operations (see Appendix A.1 for details). The DNA's subprogram bank is comprised of additional programs that can be executed as functions by instructions. Each subprogram can only call subprograms with a higher index in the subprogram bank, which removes the possibility of cycles.

Each instruction's argument set contains a list of potential argument values for each instruction operation. The set of argument fields represents the union of fields that all the operation primitives use:

- *Input 1*: The index of the hidden state that will be used as the first tensor input for this function. The index of each hidden state is the index of the instruction that produced it, with the subprogram's input states at indexes 0 and 1. An example of an operation that uses this is SIN.
- *Input 2*: The index of the second tensor input. This is only used for operations that are binary with respect to tensor inputs. An example of an operation that uses this is ADD.
- *Constant*: A real valued constant. An example of an operation that uses this is MAX; *tf.math.maximum(x, C)* for $C = 0$ is how we express the Transformer's ReLU activation.
- *Dimension Size*: An integer representing the output dimension size for transformations that utilize weight matrices. An example of an operation that uses this is CONV 1X1, the dense projection used by the Transformer's attention projections and feed forward portions. See Appendix A.2 for how we employ relative dimensions [16] to resize our models.

Our search subprograms are converted to TF programs by converting each subprogram instruction to a corresponding line of TF code, one at a time in indexing order. To create the TF line, the instruction operation is mapped to the corresponding TF primitive function or DNA subprogram, and any relevant arguments are plugged in (see Appendix A.1 for the full TF primitives vocabulary, including argument mappings); the other arguments are ignored. The TF tensor that is generated by the final instruction is taken as the subprogram output. We do not use TF Eager and so a useful property of the constructed programs is that irrelevant nodes that do not contribute to the program outputs are ignored as per TF's original deferred execution design [10]. See Figure 2 for an illustration of how subprograms are converted to TF graphs and see Appendix A.2 for more details on how TF graphs are constructed, including how we handle causal masking.

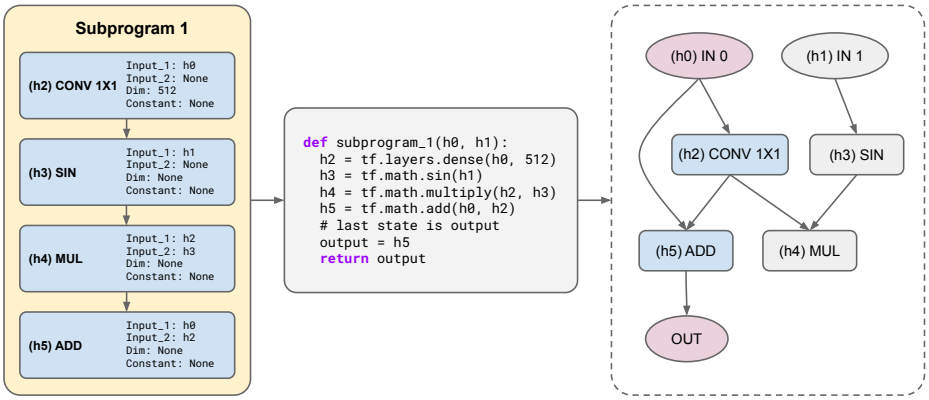



Figure 2: Example of a program converted into its corresponding TensorFlow graph. Nodes that do not contribute to the program output are not executed thanks to TensorFlow's deferred execution design.

**Evolutionary Search:** The goal of our evolutionary search is to find the most training efficient architecture in our search space. To do this, we give each model a fixed training budget (24 TPUv2 hours) and define its fitness as its perplexity on the One Billion Words Benchmark (LM1B) [22] in Tensor2Tensor [23]. This objective contrasts most previous architecture search works, which explicitly aim to reduce training or inference step time when optimizing for efficiency [24, 25, 26, 27]. Our objective is different in that the trade-off between step time and sample efficiency is implicit. For instance, a modification that doubles step time, but triples sample efficiency is a good modification in our space, as it ultimately makes the architecture more compute efficient. Indeed, the modifications we find to be most beneficial, squaring ReLUs and adding depthwise convolutions to attention,

increase training step time. However, they improve the sample efficiency of the model so much that they decrease the total compute needed to reach a target quality, by drastically reducing the number of training steps needed to get there.

The search algorithm we use is Regularized Evolution [28] with hurdles [16]. We configure our hurdles using a $50^{th}$ percentile passing bar and space them such that equal compute is invested in each hurdle band; this reduces the compute cost by a factor of 6.25X compared to the same experiment with full model evaluations (see Appendix A.3 for more details). We use 7 training hours as a proxy for a full day's training because a vanilla Transformer comes within 90% of its 24 hour training perplexity with just 7 hours of training. This reduces compute costs further by a factor of 3.43X, for a total compute reduction factor of 21.43X. So, although our target is to improve 24 hour performance, it only takes about 1.1 hours to evaluate an individual on average (see the Appendix A.4 for more search specifics, including mutation details and hyperparameters).

Our search space is different from previous architecture search search spaces (see survey by [29]), which are often heavily biased such that random search performs well (see analysis by [30, 31, 32]). As our search space does not have this bias, 78% of random programs in our space with length equal to a Transformer program cannot train more than five minutes, due to numerical instability. Because of this open-endedness and abundance of degenerate programs, it is necessary to initialize the search population with copies of the Transformer [16] (input embedding size $d_{model} = 512$, feed forward upwards projection size $d_{ff} = 2048$, and number of layers $L = 6$) (Figure 3). To apply this initialization to our search space, we must determine how to divide the Transformer program into subprograms. To do this, we divide along the lines of the machine learning concepts that comprise it. For instance we create one subprogram each for self-attention, ReLU and layer norm, using commonly used implementations (see Appendix A.5 for the complete list). We call this method "conceptual initialization" because it introduces a bias to the search through initialization, while leaving the search space for evolution and the action space for mutations open-ended. This contrasts the large amount of previous works which introduce bias through the search space. Although some works have also explored searching spaces that are open ended like ours on miniature tasks [33], we demonstrate that our techniques can scale to full sized deep learning regimes (see Results).

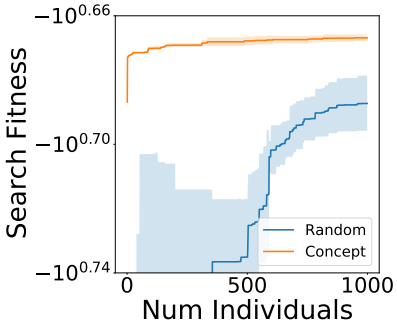


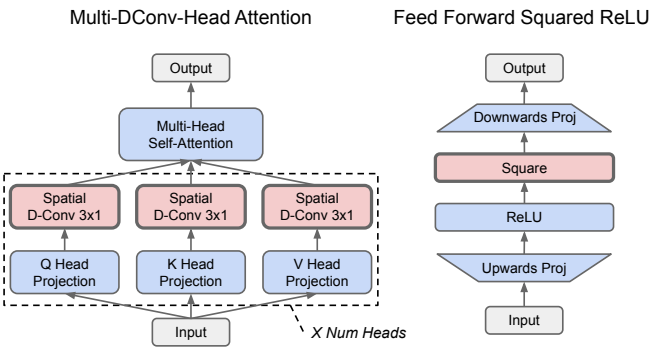

Figure 3: Small scale searches (10 hours on 10 TPUv2 chips) comparing conceptual initialization to random initialization. Our search space is open ended enough that it is infeasible to search without strong initialization.

We run our search for $\sim 25K$ individuals and retrain the top 100 individuals on the search task to select the best one.

## 3 Primer

**Primer:** We name the discovered model *Primer*, which stands for PRIMitives searched transformER (See Figure 21 for the full program). Primer shows significant improvement when retrained on the search task, requiring less than half the compute of Transformer to reach the same quality (Figure 6). In the Results section we additionally show that Primer makes equally large gains when transferred to other code bases, training regimes, and datasets.

**Primer-EZ:** A core motivation of this work is to develop simple techniques that can be easily adopted by language modeling practitioners. To accomplish this, we perform ablation tests across two codebases (T5 [5] and Tensor2Tensor [23]) and determine which of its modifications are generally useful (Appendix Figure 24). The two that produce the most robust improvements are squaring feed forward ReLUs and adding depthwise convolution to attention multi-head projections (Figure 4). We refer to a Transformer with just these two easy modifications as *Primer-EZ*; this is our recommended model for language modeling practitioners. We now explain these modifications and then measure their empirical effectiveness.




D-Conv Projection Pseudo-code

```
# Use to create each K, Q,
    and V head of size 'hs'.
def dconv_projection(x, hs):
  # Create head.
  x = proj(x,
           head_size=hs,
           axis="channel")

  # Apply D-Conv to head.
  x = d_conv(x,
             width=3,
             head_size=hs,
             axis="spatial",
             mask="causal")
  return x
```

Figure 4: The two main modifications that give Primer most of its gains: depthwise convolution added to attention multi-head projections and squared ReLU activations. These modifications are easy to implement and transfer well across codebases. We call the model with just these two modifications Primer-EZ. Blue indicates portions of the original Transformer and red signifies one of our proposed modifications.

**Squared ReLU:** The most effective modification is the improvement from a ReLU activation to a squared ReLU activation in the Transformer's feed forward block. Rectified polynomials of varying degrees have been studied in the context of neural network activation functions [34], but are not commonly used; to the best of our knowledge, this is the first time such rectified polynomial activations are demonstrated to be useful in Transformers. Interestingly, the effectiveness of higher order polynomials can also be observed in other effective Transformer nonlinearities, such as GLU variants like ReGLU [35] ($Y = XU \odot \max(XV, 0)$ where $\odot$ is an element-wise product) and point-wise activations like approximate GELU [36] ($y = 0.5x(1 + \tanh(\sqrt{2/\pi}(x + 0.044715x^3)))$). However, squared ReLU has drastically different asymptotics as $x \to \infty$ compared to the most commonly used activation functions: ReLU, GELU and Swish (Figure 5 left side). Squared ReLU does have significant overlap with ReGLU and in fact is equivalent when ReGLU's $U$ and $V$ matrices are the same and squared ReLU is immediately preceded by a linear transformation with weight matrix $U$. This leads us to believe that squared ReLUs capture the benefits of these GLU variants, while being simpler without additional parameters, and delivering better quality (Figure 5 right side).

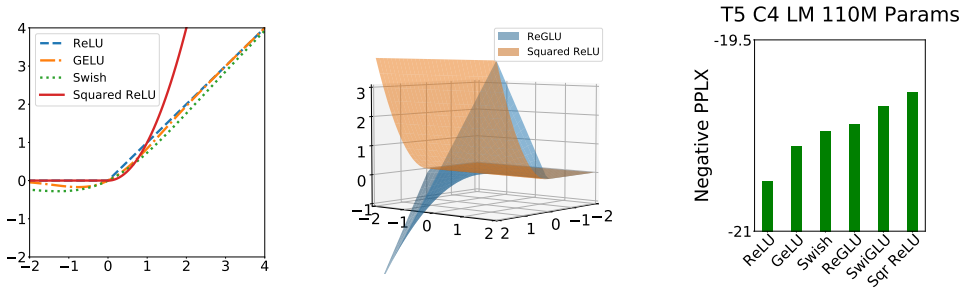



Figure 5: (Left) Squared ReLU has starkly different asymptotics compared to other common activation functions. (Center) Squared ReLU has significant overlap with GLU variants [35] that use activations with ReLU-like asymptotics, such as ReGLU and SwiGLU. Our experiments indicate that squared ReLU is better than these GLU variants in Transformer language models. (Right) Comparison of different nonlinearities in Transformers trained on C4 auto-regressive LM for $525K$ steps.

**Multi-DConv-Head Attention:** Another effective modification is adding 3x1 depthwise convolutions after each of the multi-head projections for query $Q$, key $K$ and value $V$ in self-attention. These depthwise convolutions are performed over the spatial dimension of each dense projection's output. Interestingly, this ordering of pointwise followed by depthwise convolution is the reverse of typical separable convolution, which we find to be less effective in Appendix A.6. We also find

that wider depthwise convolution and standard convolution not only do not improve performance, but in several cases hurt it. Although depthwise convolutions have been used for Transformers before [37, 38], using them after each dense head projection has not been done to the best of our knowledge. Multi-DConv-Head Attention is not unlike the Convolutional Attentions proposed by Wu et al. [39] for vision, however there they use traditional separable convolution ordering for pointwise and depthwise transformations and do not limit depthwise convolutions scope per-head, as we do.

**Other Modifications:** The other Primer modifications are less effective. Graphs for each modification can be found in Appendix A.5 and an ablation study can be found in Appendix A.7. We briefly describe the modifications and their usefulnesses here:

- *12X Bottleneck Projection*: Primer uses a smaller $d_{model}$ size of 384 (compared to the baseline's 512) and a larger $d_{ff}$ size of 4608 (compared to the baseline's 2048). We find this larger projection improves results dramatically at smaller sizes ($\sim$35*M* parameters), but is less effective for larger models, as has been previously noted [9]. For this reason we do not include this modification when referencing Primer or Primer-EZ.
- *Shared Q and K Depthwise Representation*: Primer shares some weight matrices for $Q$ and $K$. $K$ is created using the previously described depthwise convolution projection and $Q = KW$ for learnable weight matrix $W \in \mathbb{R}^{d \times d}$. We find that this generally hurts performance.
- *Pre and Post Normalization*: The standard practice for Transformers has become putting normalization before both the self-attention and feed forward transformations [40, 41]. Primer uses normalization before self-attention but applies the second normalization *after* the feed forward transformation. We find this is helpful in some but not all cases.
- *Custom Normalization*: Primer uses a modified version of layer normalization [42] that uses $x(x - \mu)$ instead of $(x - \mu)^2$, but we do not find this is generally effective.
- *Post-Softmax Spatial Gating*: The original discovered model has a set of per-channel learnable scalars after the attention softmax, which improves perplexity for fixed length sequences. However, these scalars cannot be applied to variable sequence lengths and so we do not include this modification in Primer for our experiments.

# 4 Results

In our experiments, we compare Primer against three main baseline architectures:

- *Vanilla Transformer*: The original Transformer [1] with RELU activations and layer normalization [42] outside of the residual path.
- *Transformer + GELU*: A commonly used variant of the vanilla Transformer that uses a GELU [43] approximation activation function [2, 7].
- *Transformer++*: A Transformer with the following enhancements: RMS normalization [44], Swish activation [45] and a GLU multiplicative branch [36] in the feed forward inverted bottleneck (SwiGLU) [35]. These modifications were benchmarked and shown to be effective in T5 [46].

We conduct our comparisons across two different codebases, Tensor2Tensor (T2T) [23] and T5 [5]. Tensor2Tensor is the codebase we use for searching and so a majority of our side-by-sides are done in T5 to prove transferability. In all cases, we use the default hyperparameters for each codebase, with regularization disabled. See Appendix A.8 for more hyperparameter details.

In the following sections, we will present our results in three main experiments with auto-regressive language modeling. First, we will show that Primer outperforms these baseline models on the search task. Next, we will show that Primer enjoys power law savings over Transformers at optimal model sizes. The gain also transfers across datasets and codebases. Finally, we will show that Primer enables 4.2X compute savings at a 500M parameter size using full compute T5 training.

## 4.1 Search Task Comparison

We first analyze Primer's performance on the search task: LM1B language modeling with sequence length 64, $\sim$35M model parameters, batches of 4096 tokens and 24 hours of training. We compare

against the baseline models in both Tensor2Tensor (T2T) [23] and T5 [5] and on TPUv2s and V100 GPUs. We grade each model's performance according to how much faster it reaches the vanilla Transformer's final quality, which we will refer to as *speedup factor*. Figure 6 shows that Primer provides a speedup factor of 1.7X or more over Transformer in all cases. Figure 6 also shows that Primer-EZ generalizes to other hardware platforms and code bases more effectively than Primer. Appendix A.7 provides an ablation study of the importance of different modifications in Primer including those that give rise to Primer-EZ.

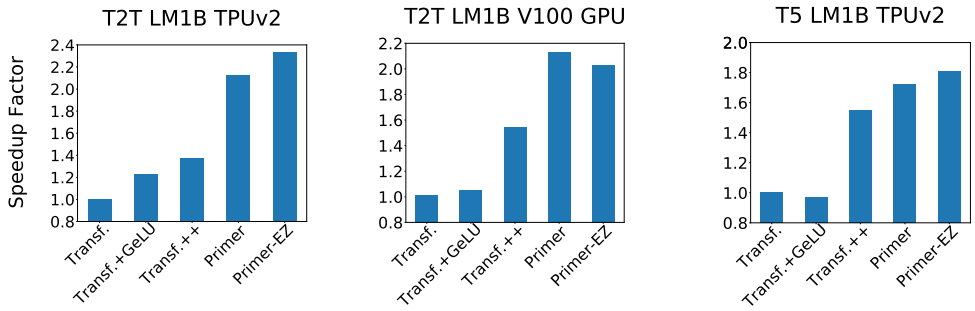



Figure 6: Comparison on the LM1B search task using 35*M* parameter models. "Speedup Factor" refers to how much less compute each method needs to reach quality parity with the baseline trained for 24 hours. Primer and Primer-EZ both achieve over 1.7X speedup in all cases. Note that results transfer across code bases and different hardware.

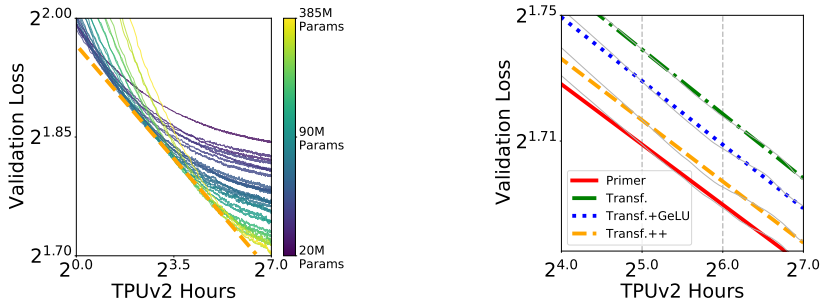



Figure 7: (Left) When sweeping over optimal parameter sizes, the relationship between Transformer language model quality and training compute roughly obeys a power law [9]. (Right) Comparison of these power law lines for varying Transformer modifications, fit with smoothed MSE loss. That the model lines are parallel to one another imply that compute savings by using superior modeling also follows a power law. Spacings between vertical dotted lines represent 2X differences in compute.

Next we study the scaling laws of Primer. Here we compare Primer to our baselines over many sizes by training each model using every permutation of $L \in \{6, 9, 12\}$ layers, $d_{model} \in \{384, 512, 1024\}$ initial embedding size, and $p \in \{4, 8, 12\}$ feed forward upwards projection ratio, creating a parameter range of 23*M* to 385*M*. The results, shown in Figure 7, corroborates previous claims that, at optimal parameters sizes, the relationship between compute and language model quality roughly follows power laws [9]. That is, the relationship between validation loss, $l$, and training compute, $c$, follows the relationship $l = ac^{-k}$, for empirical constants $a$ and $k$. This is represented as a line in double log space (Figure 7): $\log l = -k \log c + \log a$. However, these lines are not the same for each architecture. The lines are roughly parallel but shifted up and down. In Appendix A.9 we show that, given a vertical spacing of $\log b^k$, parallel lines such as these indicate compute savings, $s$, for superior modeling also follow a power law of the form $l = a_1(1 - 1/b)^k s^{-k}$. The intuition behind this is that $b$ is a constant compute reduction factor for all $l$ and thus a power law investment of training compute with relation to $l$ results in a power law savings with relation to $l$ as well (see Appendix A.9).

Primer also has the capacity to improve inference, despite our search focusing on training compute. Figure 8 shows a Pareto front comparison of quality vs. inference, when using feed forward pass timing as a proxy for inference. We use forward pass timing as a proxy for inference because there are multiple ways to decode a language model, each with varying compute costs. A more in depth study

could be conducted studying Primer's inference performance across different decoding methods, serving platforms, datasets, etc., but that is beyond the scope of this work.

### 4.2 Transferability of Primer to Other Codebases, Datasets, and Model Types

We now study Primer's ability to transfer to larger datasets, PG19 and C4, in another codebase, T5. We additionally scale up to a higher compute regime that has been used as a proxy for large scale training by previous studies [46, 5]; the batches are increased to 65$K$ tokens, the sequence lengths are a longer 512, each decoder is 110$M$ parameters ($d_{model} = 768$, $d_{ff} = 3072$, $L = 12$) and each model is trained to $\sim$525$K$ steps on 4 TPUv3 chips. We also continue training each model to 1$M$ steps to study the effect of larger compute budgets on Primer savings. The results, shown in Figure 9, indicate that Primer-EZ is as strong in larger data, higher compute regimes, as it is in the smaller LM1B regime. Compared to the vanilla baseline, Primer-EZ is $\sim$2X more efficient at the end of training on both PG19 and C4.

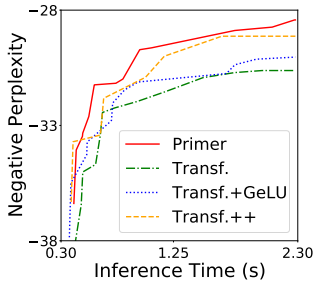



Figure 8: Pareto optimal inference comparison on LM1B. Primer demonstrates improved inference when controlling for quality at a majority of inference times. We observe these models have a 0.97 correlation between their train step and inference times.

We also show that Primer is compatible with other efficient model families, such as large sparse mixture-of-experts like Switch Transformer [8] and efficient Transformer approximations like Synthesizer [21] (Figure 9). For these, we use the T5 implementations provided by Narang et al. [46]. The Primer-EZ techniques of added depthwise convolutions and squared ReLUs reduce Switch Transformer compute cost by a factor of 1.5; this translates to a 0.6 PPLX improvement when controlling for compute (see Appendix A.10). Adding squared ReLUs to Synthesizer reduces costs by a factor of 1.9 and improves perplexity by 0.8 when fully trained.

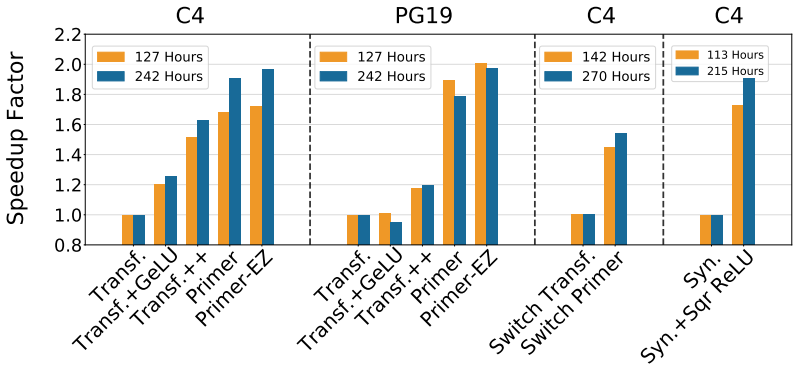



Figure 9: Comparison transferring Primer to larger datasets (C4 and PG19) and different model families (Switch Transformer and Synthesizer) in a different code base (T5) with an order of magnitude more compute than the search task. In the smaller dataset case, PG19, Primer-EZ outperforms the baselines with half as much compute. In the larger dataset case, C4, the fraction of Primer-EZ compute savings increases as more compute is invested. Primer-EZ modifications also improve Switch Transformer (550$M$ params) and Synthesizer (145$M$ params), showing that it is compatible with other efficient methods. Compute budgets are selected according to how long it takes each baseline to train for 525$K$ and 1$M$ steps. See Appendix A.10 for exact numbers.

### 4.3 Large Scale T5 Auto-Regressive Language Model Training

In large scale compute configurations, the Primer compute savings ratios are *higher*. In our final comparison, we scale up to the full T5 compute regime, copying Raffel et al. exactly [5]. This is the same as the configuration in the previous section, but uses batch sizes of 1$M$ tokens, 64 TPUv3 chips and a parameters size of 537$M$ ($d_{model} = 1024$, $d_{ff} = 8192$, $L = 24$). Primer is 4.2X more compute efficient than the original T5 model and 2X more efficient than our strengthened Transformer++ baseline.

The reason why savings are even better here is because, at fixed sizes, more compute invested yields higher Primer compute savings. Figure 10 shows how the fraction of compute Primer needs to achieve parity with the original T5 architecture shrinks as the models are trained for longer; this is due to the asymptotic nature of both the control and variable perplexity training curves (Figure 10). This differs from the power law savings described in Section A.6. There, we use optimal parameter sizing for each compute budget, and so the compute saving factor, $b$, remains constant. For fixed model sizes, the compute saving factor grows as more compute is invested, meaning that compute savings can exceed the power law estimation. That we are using an exact replica of Raffel et al.'s [5] training regime shows that these gains can be easily achieved by dropping Primer into existing setups, without any further tuning.

| Model | Steps | TPUv3 Hours | PPLX |
|---|---|---|---|
| Original T5 | 1M | 15.7K | 13.25 |
| T5++ | 251K | 4.6K | 13.25 |
| Primer | 207K | **3.8K** | 13.25 |
| T5++ | 1M | 16.5K | 12.69 |
| Primer | 480K | **8.3K** | 12.69 |
| Primer | 1M | 17.3K | 12.35 |

Table 1: Comparison in compute usage to reach target qualities on C4 LM at 537$M$ parameters using the full T5 compute scale. Target qualities are selected according to the final performances of the baseline models. Primer achieves the same quality as the original T5 architecture using 4.2X less compute.

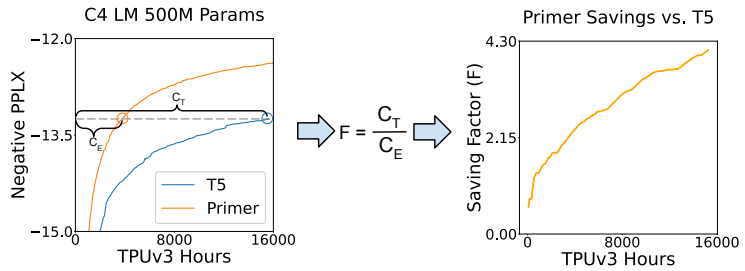



Figure 10: Compute savings of Primer vs. original T5 architecture on C4 LM over time. The more compute invested in training, the higher the savings due to the asymptotic nature of both perplexity curves. Primer achieves the same performance as the original T5 architecture with 4.2X less compute.

## 5 Conclusion

**Limitations:** There are limitations to this study. Firstly, our cross-parameter model sweeps are approximately an order of magnitude smaller than the sweeps performed in the original study by Kaplan et al. [9]. By a similar token, although our large scale experiments use a significant amount of compute, it is still orders of magnitude smaller than state-of-the-art models such as GPT-3 [7].

Another limitations is that we focus primarily on decoder-only models, while encoder-only [2, 3, 4] or encoder-decoder sequence models [47, 1, 6] are still widely used. In Appendix A.11 we perform encoder-decoder masked language modeling comparisons in T5 and show that Primer modifications yield improvements there as well, but we do not study these improvements to the same degree.

**Recommendations and Future Directions:** We recommend the adoption of Primer-EZ by language modeling practitioners for its strong performance, ease of implementation and robustness to hyperparameter and code base changes. To demonstrate its potential, we simply dropped Primer-EZ into established code bases and, without any changes, demonstrated it can give a significant performance boost. However, in practice, additional tuning could also further improve its performance.

One of the findings of this study has been that small changes to activation functions can yield more efficient training. In the effort to reduce the cost of training large language models, more investment in the development of easy to implement changes like that could be a promising area for future exploration.

This study focused primarily on the development of modifications to Transformer decoders. Similar studies that focus on improving encoders and encoder-decoder models could potentially yield equal improvements in the study of other sequence problems, such as masked language modeling.

# References


[1] Ashish Vaswani, Noam M. Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Advances in Neural Information Processing Systems*, 2017.

[2] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. In *NAACL*, 2018.

[3] Zhilin Yang, Zihang Dai, Yiming Yang, Jaime Carbonell, Ruslan Salakhutdinov, and Quoc V Le. Xlnet: Generalized autoregressive pretraining for language understanding. In *Advances in Neural Information Processing Systems*, 2019.

[4] Yinhan Liu, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, Mike Lewis, Luke Zettlemoyer, and Veselin Stoyanov. Roberta: A robustly optimized bert pretraining approach. *arXiv preprint arXiv:1907.11692*, 2019.

[5] Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J. Liu. Exploring the limits of transfer learning with a unified text-to-text transformer. *Journal of Machine Learning Research*, 21(140):1–67, 2020.

[6] Daniel Adiwardana, Minh-Thang Luong, David R. So, J. Hall, Noah Fiedel, R. Thoppilan, Z. Yang, Apoorv Kulshreshtha, Gaurav Nemade, Yifeng Lu, and Quoc V. Le. Towards a human-like open-domain chatbot. *arXiv preprint arXiv:2001.09977*, 2020.

[7] Tom B Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. Language models are few-shot learners. In *Advances in Neural Information Processing Systems*, 2020.

[8] William Fedus, Barret Zoph, and Noam M. Shazeer. Switch transformers: Scaling to trillion parameter models with simple and efficient sparsity. *arXiv preprint arXiv:2101.03961*, 2021.

[9] Jared Kaplan, Sam McCandlish, Tom Henighan, Tom B. Brown, Benjamin Chess, Rewon Child, Scott Gray, Alec Radford, Jeffrey Wu, and Dario Amodei. Scaling laws for neural language models, 2020.

[10] Martín Abadi, P. Barham, J. Chen, Z. Chen, Andy Davis, J. Dean, M. Devin, Sanjay Ghemawat, Geoffrey Irving, M. Isard, M. Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, D. Murray, Benoit Steiner, P. Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Y. Yu, and Xiaoqiang Zhang. Tensorflow: A system for large-scale machine learning. In *OSDI*, 2016.

[11] Alec Radford, Jeffrey Wu, R. Child, David Luan, Dario Amodei, and Ilya Sutskever. Language models are unsupervised multitask learners. In *Technical report, OpenAI*, 2019.

[12] Timo Schick and Hinrich Schütze. It's not just size that matters: Small language models are also few-shot learners, 2021.

[13] Sinong Wang, Han Fang, Madian Khabsa, Hanzi Mao, and Hao Ma. Entailment as few-shot learner. *ArXiv*, abs/2104.14690, 2021.

[14] Tianyu Gao, Adam Fisch, and Danqi Chen. Making pre-trained language models better few-shot learners. *ArXiv*, abs/2012.15723, 2020.

[15] Esteban Real, Sherry Moore, Andrew Selle, Saurabh Saxena, Yutaka Leon Suematsu, Quoc V. Le, and Alex Kurakin. Large-scale evolution of image classifiers. In *ICML*, 2017.

[16] David R. So, Chen Liang, and Quoc V. Le. The evolved transformer. In *ICML*, 2019.

[17] Xin Yao. Evolving artificial neural networks. *Proceedings of the IEEE*, 87(9):1423–1447, 1999.

[18] Jurgen Schmidhuber. Evolutionary principles in self-referential learning. (on learning how to learn: The meta-meta-... hook.). Diploma thesis, Technische Universitat Munchen, Germany, 1987.

[19] Kenneth Stanley, Jeff Clune, Joel Lehman, and Risto Miikkulainen. Designing neural networks through neuroevolution. *Nature Machine Intelligence*, 1:24–35, 2019.

[20] Noam M. Shazeer, Youlong Cheng, Niki Parmar, Dustin Tran, Ashish Vaswani, Penporn Koanantakool, Peter Hawkins, H. Lee, Mingsheng Hong, C. Young, Ryan Sepassi, and Blake A. Hechtman. Mesh-tensorflow: Deep learning for supercomputers. In *NeurIPS*, 2018.

[21] Yi Tay, Dara Bahri, Donald Metzler, Da-Cheng Juan, Zhe Zhao, and Che Zheng. Synthesizer: Rethinking self-attention in transformer models. *arXiv preprint arXiv:2005.00743*, 2020.

[22] Ciprian Chelba, Tomas Mikolov, Mike Schuster, Qi Ge, Thorsten Brants, Phillipp Koehn, and Tony Robinson. One billion word benchmark for measuring progress in statistical language modeling. In *Interspeech*, 2014.

[23] Ashish Vaswani, Samy Bengio, Eugene Brevdo, Francois Chollet, Aidan N. Gomez, Stephan Gouws, Llion Jones, Łukasz Kaiser, Nal Kalchbrenner, Niki Parmar, Ryan Sepassi, Noam Shazeer, and Jakob Uszkoreit. Tensor2tensor for neural machine translation. *CoRR*, abs/1803.07416, 2018.

[24] Mingxing Tan and Quoc Le. Efficientnet: Rethinking model scaling for convolutional neural networks. In *ICML*, 2019.

[25] Mingxing Tan, Bo Chen, Ruoming Pang, Vijay Vasudevan, and Quoc V. Le. Mnasnet: Platform-aware neural architecture search for mobile. *2019 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 2815–2823, 2019.

[26] Han Cai, Ligeng Zhu, and Song Han. Proxylessnas: Direct neural architecture search on target task and hardware. *ArXiv*, abs/1812.00332, 2019.

[27] Thomas Elsken, Jan Hendrik Metzen, and Frank Hutter. Efficient multi-objective neural architecture search via lamarckian evolution, 2019.

[28] Esteban Real, Alok Aggarwal, Yanping Huang, and Quoc V. Le. Regularized evolution for image classifier architecture search. In *AAAI*, 2019.

[29] Thomas Elsken, Jan Hendrik Metzen, and Frank Hutter. Neural architecture search: A survey. *Journal of Machine Learning Research*, 20(55):1–21, 2019.

[30] Liam Li and Ameet S. Talwalkar. Random search and reproducibility for neural architecture search. In *UAI*, 2019.

[31] Kaicheng Yu, Christian Sciuto, Martin Jaggi, Claudiu Musat, and Mathieu Salzmann. Evaluating the search phase of neural architecture search. In *ICLR*, 2020.

[32] Gabriel Bender, Hanxiao Liu, Bo Chen, Grace Chu, Shuyang Cheng, Pieter-Jan Kindermans, and Quoc V. Le. Can weight sharing outperform random architecture search? An investigation with tunas. In *CVPR*, 2020.

[33] Esteban Real, Chen Liang, David R. So, and Quoc V. Le. Automl-zero: Evolving machine learning algorithms from scratch. In *ICML*, 2020.

[34] Dmitry Krotov and John J. Hopfield. Dense associative memory for pattern recognition. In *Advances in Neural Information Processing Systems*, 2016.

[35] Noam Shazeer. Glu variants improve transformer. *arXiv preprint arXiv:2002.05202*, 2020.

[36] Yann N Dauphin, Angela Fan, Michael Auli, and David Grangier. Language modeling with gated convolutional networks. In *ICML*, 2017.

[37] Adams Wei Yu, David Dohan, Minh-Thang Luong, Rui Zhao, Kai Chen, Mohammad Norouzi, and Quoc V. Le. QANet: Combining local convolution with global self-attention for reading comprehension. In *ICLR*, 2018.

[38] Anmol Gulati, James Qin, Chung-Cheng Chiu, Niki Parmar, Yu Zhang, Jiahui Yu, Wei Han, Shibo Wang, Zhengdong Zhang, Yonghui Wu, and Ruoming Pang. Conformer: Convolution-augmented transformer for speech recognition. In *Interspeech*, 2020.

[39] Haiping Wu, Bin Xiao, Noel Codella, Mengchen Liu, Xiyang Dai, Lu Yuan, and Lei Zhang. Cvt: Introducing convolutions to vision transformers. *ArXiv*, abs/2103.15808, 2021.

[40] Alexei Baevski and Michael Auli. Adaptive input representations for neural language modeling. In *arXiv preprint arXiv:1809.10853*, 2019.

[41] Ruibin Xiong, Yunchang Yang, Di He, Kai Zheng, S. Zheng, Chen Xing, Huishuai Zhang, Yanyan Lan, L. Wang, and T. Liu. On layer normalization in the transformer architecture. *arXiv preprint arXiv:2002.04745*, 2020.

[42] Jimmy Ba, Jamie Kiros, and Geoffrey E. Hinton. Layer normalization. *arXiv preprint arXiv:1607.06450*, 2016.

[43] Dan Hendrycks and Kevin Gimpel. Bridging nonlinearities and stochastic regularizers with gaussian error linear units. *arXiv preprint arXiv:1606.08415*, 2016.

[44] Biao Zhang and Rico Sennrich. Root mean square layer normalization. In *NeurIPS*, 2019.

[45] Prajit Ramachandran, Barret Zoph, and Quoc V. Le. Searching for activation functions. *arXiv preprint arXiv:1710.05941*, 2018.

[46] Sharan Narang, Hyung Won Chung, Yi Tay, William Fedus, Thibault Févry, Michael Matena, Karishma Malkan, Noah Fiedel, Noam Shazeer, Zhenzhong Lan, Yanqi Zhou, Wei Li, Nan Ding, Jake Marcus, Adam Roberts, and Colin Raffel. Do transformer modifications transfer across implementations and applications? *arXiv preprint arXiv:2102.11972*, 2021.

[47] Ilya Sutskever, Oriol Vinyals, and Quoc V Le. Sequence to sequence learning with neural networks. In *Advances in Neural Information Processing Systems*, 2014.

[48] Lisha Li, Kevin Jamieson, Giulia DeSalvo, Afshin Rostamizadeh, and Ameet Talwalkar. Hyperband: A novel bandit-based approach to hyperparameter optimization. *Journal of Machine Learning Research*, 18(185):1–52, 2018.

[49] Zohar Karnin, Tomer Koren, and Oren Somekh. Almost optimal exploration in multi-armed bandits. In *Proceedings of the 30th International Conference on Machine Learning*.

[50] David Patterson, Joseph Gonzalez, Quoc V. Le, Chen Liang, Lluís-Miquel Munguía, D. Rothchild, David R. So, Maud Texier, and J. Dean. Carbon emissions and large neural network training. *ArXiv*, abs/2104.10350, 2021.

# Checklist

1. For all authors...
   (a) Do the main claims made in the abstract and introduction accurately reflect the paper's contributions and scope? [Yes] All claims in the abstract are backed by empirical results in the Results section of the paper.
   (b) Did you describe the limitations of your work? [Yes] See the "Limitations" portion of the Conclusion.
   (c) Did you discuss any potential negative societal impacts of your work? [Yes] In Appendix A.12 we release the carbon emission estimates for our work.
   (d) Have you read the ethics review guidelines and ensured that your paper conforms to them? [Yes] We have read the ethics review guidelines and do not feel that our paper breaches them.
2. If you are including theoretical results...
   (a) Did you state the full set of assumptions of all theoretical results? [N/A]
   (b) Did you include complete proofs of all theoretical results? [N/A]
3. If you ran experiments...
   (a) Did you include the code, data, and instructions needed to reproduce the main experimental results (either in the supplemental material or as a URL)? [Yes] We include compute graphs and pseudocode for Primer and Primer-EZ. Furthermore we will open source model implementations compatible with T5 if accepted that allow exactly reproducing our experiment trainings there.
   (b) Did you specify all the training details (e.g., data splits, hyperparameters, how they were chosen)? [Yes] These are described in both the Results section and supplementary materials. We did not do any hyperparameter tuning. We instead used the hyperparameters from existing open source libraries.
   (c) Did you report error bars (e.g., with respect to the random seed after running experiments multiple times)? [Yes] Tables with full data that describe the figures in the main text are provided in the Appendix with error bars.
   (d) Did you include the total amount of compute and the type of resources used (e.g., type of GPUs, internal cluster, or cloud provider)? [Yes] We explicitly state the number of accelerator hours we run each experiment for, in addition to the accelerators they were run on.
4. If you are using existing assets (e.g., code, data, models) or curating/releasing new assets...
   (a) If your work uses existing assets, did you cite the creators? [Yes] We use the T2T and T5 open source libraries and cite them.
   (b) Did you mention the license of the assets? [N/A]
   (c) Did you include any new assets either in the supplemental material or as a URL? [N/A]
   (d) Did you discuss whether and how consent was obtained from people whose data you're using/curating? [N/A]
   (e) Did you discuss whether the data you are using/curating contains personally identifiable information or offensive content? [N/A]
5. If you used crowdsourcing or conducted research with human subjects...
   (a) Did you include the full text of instructions given to participants and screenshots, if applicable? [N/A]
   (b) Did you describe any potential participant risks, with links to Institutional Review Board (IRB) approvals, if applicable? [N/A]
   (c) Did you include the estimated hourly wage paid to participants and the total amount spent on participant compensation? [N/A]