

Primer: Searching for Efficient Transformers for Language Modeling

David R. So, Wojciech Mańke, Hanxiao Liu, Zihang Dai, Noam Shazeer, Quoc V. Le
 Google Research, Brain Team
 {davidso, wojciechm, hanxiaol, zihangd, noam, qvl}@google.com

Abstract

Large Transformer models have been central to recent advances 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 Transformers by searching for a more efficient variant. Compared to previous approaches, our search is performed at a lower level, over the primitives that define a Transformer TensorFlow program. We identify an architecture, named Primer, that has a smaller training cost than the original Transformer and other variants for auto-regressive language modeling. Primer’s improvements can be mostly attributed to two simple modifications: squaring ReLU activations and adding a depthwise convolution layer after each Q, K, and V projection in self-attention.

Experiments show Primer’s gains over Transformer increase as compute scale grows and follow a power law with respect to quality at optimal model sizes. We also verify empirically that Primer can be dropped into different codebases to significantly speed up training without additional tuning. For example, at a 500M parameter size, Primer improves the original T5 architecture on C4 auto-regressive language modeling, reducing the training cost by 4X. Furthermore, the reduced training cost means Primer needs much less compute to reach a target one-shot performance. For instance, in a 1.9B parameter configuration similar to GPT-3 XL, Primer uses 1/3 of the training compute to achieve the same one-shot performance as Transformer. We open source our models and several comparisons in T5 to help with reproducibility.¹

1 Introduction

Transformers [1] have been used extensively in many NLP advances over the past few years (e.g., [2, 3, 4, 5, 6, 7]). With scaling, Transformers have produced increasingly better performance [3, 7, 8, 9], but the costs of training larger models have become prohibitively expensive.

In this paper, 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 computation graph [10]. Given a search space of TensorFlow programs, we use evolution [11, 12, 13, 14, 15, 16, 17] to search for models that achieve as low of a validation loss as possible given a fixed amount of training compute. An advantage of using TensorFlow programs as the search space is that it is easier to find simple low-level improvements to optimize Transformers. We focus on decoder-only auto-regressive language modeling (LM), because of its generality and success [18, 7, 19, 20, 21].²

The discovered model, named Primer (PRIMitives searched transformER), exhibits strong performance improvements over common Transformer variants on auto-regressive language modeling.

¹<https://github.com/google-research/google-research/tree/master/primer>

²We provide details of our primitives search in TensorFlow, but the same approach can also be applied to other deep learning libraries.

Our experiments show that Primer has the benefits of (1) achieving a target quality using a smaller training cost, (2) achieving higher quality given a fixed training cost, and (3) achieving a target quality using a smaller inference cost. These benefits are robust and hold across model sizes (20M to 1.9B parameters), across compute scales (10 to 10^5 accelerator hours), across datasets (LM1B, C4, PG19 [22]), across hardware platforms (TPUv2, TPUv3, TPUv4 and V100), across multiple Transformer codebases using default configurations (Tensor2Tensor, Lingvo, and T5) and across multiple model families (dense Transformers [1], sparse mixture-of-experts Switch Transformers [8], and Synthesizers [23]). We open source these comparisons to help with the reproducibility of our results.¹

Our main finding is that the compute savings of Primer over Transformers increase as training cost grows, when controlling for model size and quality. These savings follow a power law with respect to quality when using optimally sized models. To demonstrate Primer’s savings in an established training setup, we compare 500M 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 setting, Primer achieves an improvement of 0.9 perplexity given the same training cost, and reaches quality parity with the T5 baseline model using 4.2X less compute. We further demonstrate that Primer’s savings transfer to one-shot evaluations by comparing Primer to Transformer at 1.9B parameters in a setup similar to GPT-3 XL [7]. There, using 3X less training compute, Primer achieves similar performance to Transformer on both pretraining perplexity and downstream one-shot tasks.

Our analysis shows that the improvements of Primer over Transformer can be mostly attributed to two main modifications: squaring ReLU activations and adding a depthwise convolution layer after each Q, K, and V projection in self-attention. These two modifications are simple and can be 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.

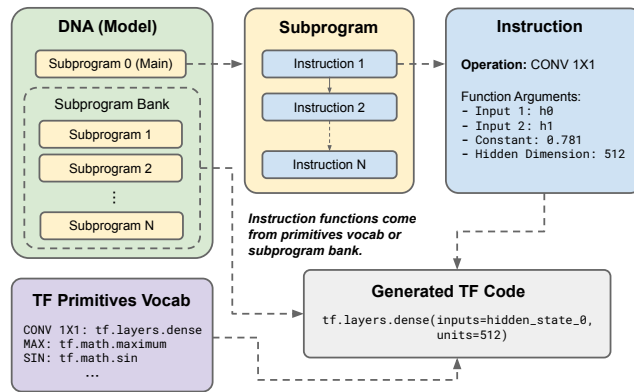


Figure 1: Overview of DNAs that define a decoder model program (i.e., an auto-regressive language model). Each DNA has a collection of subprograms, where SUBPROGRAM 0 is the MAIN() function entry point. Each subprogram is comprised of instructions, which are converted to lines of TensorFlow code. Instruction 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.

Figure 1 shows how programs are constructed in our search space. Each program is built from an 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*. The operation denotes 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 (see Appendix A.1 for details). It is worth emphasizing that high-level building blocks such as self-attention are not operations in the search space, but can be constructed from our low-level operations. 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. 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 by 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* [13] 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 programs' 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.

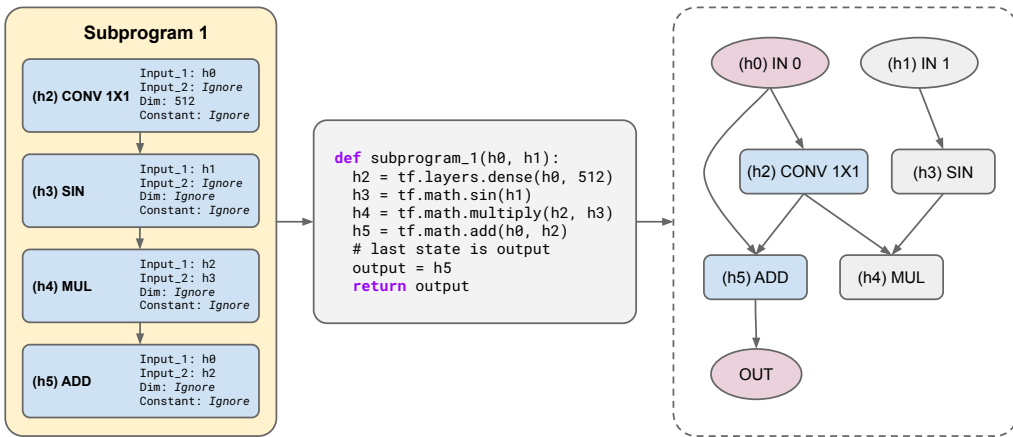


Figure 2: Example of a program converted into its corresponding TensorFlow graph. Nodes that do not contribute to the program output are not executed thanks to TensorFlow's deferred execution design.

Evolutionary Search: The goal of our evolutionary search is to find the most training efficient architecture in the 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) [24] in Tensor2Tensor [25]. This approach, which we call an *implicit efficiency objective* by fixed training budget, contrasts previous architecture search works that explicitly aim to reduce training or inference step time when optimizing for efficiency [26, 27, 28, 29]. 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 search, 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 [30] with hurdles [13]. We configure our hurdles using a 50th percentile passing bar and space them such that equal compute is invested in each hurdle band; this reduces the search cost by a factor of 6.25X compared to the same experiment with full model evaluations (see Appendix A.3 for more details). Additionally, 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 the search cost 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 Appendix A.4 for more search specifics, including mutation details and hyperparameters). We run our search for ~25K individuals and retrain the top 100 individuals on the search task to select the best one.

Hurdles: only 50% of the best models get trained after s steps (first hurdle band). The another 50% stops at the next band ...

Mutations and the evolution process is described in the appendix

Our search space is different from previous search spaces (see architecture search survey by [31]), which are often heavily biased such that random search performs well (see analysis by [32, 33, 34]). 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 [13] (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 constitute 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 that introduce bias through the search space. Although some works have also explored searching spaces that are open-ended like ours on miniature tasks [35], we demonstrate that our techniques can scale to full sized deep learning regimes (see Section 4).

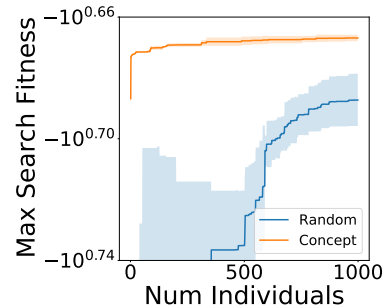


Figure 3: Small scale searches (10 hours on 10 TPUv2 chips) comparing conceptual initialization to random initialization. Our search space is open-ended enough that it is infeasible to search without strong initialization.

3 Primer

Primer: We name the discovered model *Primer*, which stands for PRIMitives searched transformER (See Appendix Figure 23 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 Section 4, we additionally show that Primer makes equally large gains when transferred to other codebases, training regimes, datasets, and downstream one-shot tasks.

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 [25]) and determine which Primer modifications are generally useful (Appendix Figure 26). 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 starting point for language modeling practitioners interested in using Primer. We now explain these modifications and then measure their empirical effectiveness.

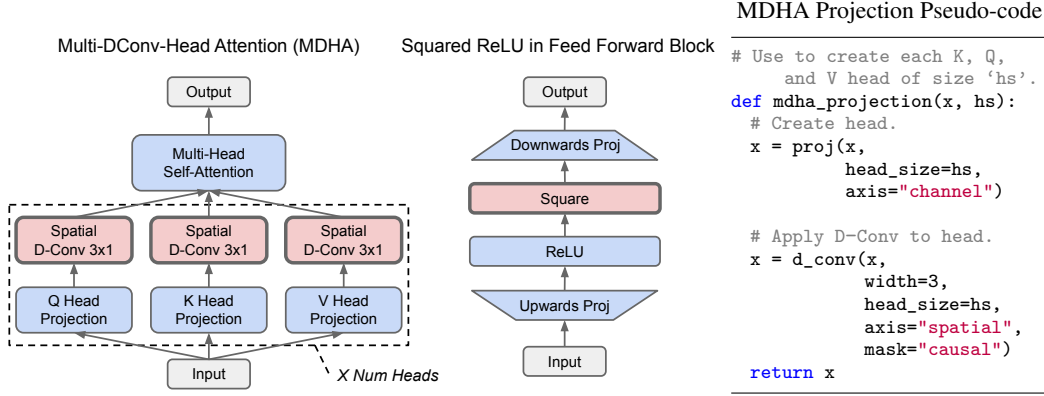


Figure 4: The two main modifications that give Primer most of its gains: depthwise convolution added to attention multi-head projections and squared ReLU activations. These modifications are easy to implement and transfer well across codebases. We call the model with just these two modifications *Primer-EZ*. 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 [36], 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 [37] can also be observed in other effective Transformer nonlinearities, such as GLU [38] variants like ReGLU [39] ($y = Ux \odot \max(Vx, 0)$ where \odot is an element-wise product) and point-wise activations like approximate GELU [40] ($y = 0.5x(1 + \tanh(\sqrt{2/\pi}(x + 0.044715x^3)))$). However, squared ReLU has drastically different asymptotics as $x \rightarrow \infty$ 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 weight 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).

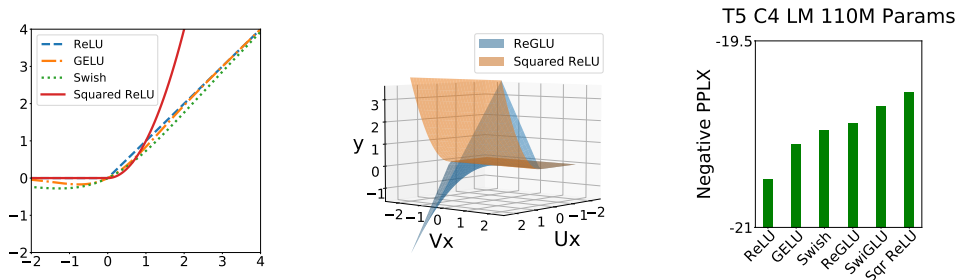


Figure 5: Left: Squared ReLU has starkly different asymptotics compared to other common activation functions. Center: Squared ReLU has significant overlap with GLU variants [39] that use activations with ReLU-like asymptotics, such as ReGLU and SwiGLU. Our experiments indicate that squared ReLU is better 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 (MDHA): 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 [41, 42], using them after each dense head projection has not been done to the best of our knowledge. MDHA is similar to Convolutional Attention [43], which uses separable convolution instead of depthwise convolution and does not apply convolution operations per attention head as we do.

Per channel convolution along the sequence dimension with a kernel size of 3

Point-wise is the projection

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:

- *Shared Q and K Depthwise Representation:* Primer shares some weight matrices for Q and K . K is created using the previously described MDHA 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 [44, 45]. 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 [46] that uses $x(x - \mu)$ instead of $(x - \mu)^2$, but we find this is not always effective.
- *12X Bottleneck Projection:* The discovered model 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 ($\sim 35M$ 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.
- *Post-Softmax Spatial Gating:* The 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.
- *Extraneous Modifications:* There are a handful of additional modifications that produce no meaningful difference in the discovered architecture. For example, hidden states being multiplied by -1.12. Verifying that these modifications neither help nor hurt quality, we exclude them from discussion in the main text and do not include them when experimenting with Primer. These extraneous modifications can still be found in Appendix A.5.

4 Results

In our experiments, we compare Primer against three Transformer variants:

- *Vanilla Transformer:* The original Transformer [1] with ReLU activations and layer normalization [46] outside of the residual path.
- *Transformer+GELU:* A commonly used variant of the vanilla Transformer that uses a GELU [40] approximation activation function [2, 7].
- *Transformer++:* A Transformer with the following enhancements: RMS normalization [47], Swish activation [48] and a GLU multiplicative branch [38] in the feed forward inverted bottleneck (SviGLU) [39]. These modifications were benchmarked and shown to be effective in T5 [49].

We conduct our comparisons across three different codebases: Tensor2Tensor (T2T) [25], T5 [5], and Lingvo [50]. Tensor2Tensor is the codebase we use for searching and so a majority of our side-by-sides are done in T5 and Lingvo to prove transferability. In all cases, we use the default Transformer 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 four main experiments on auto-regressive language modeling. First, we will show that Primer outperforms the baseline models on the search task. Next, we will show that the relationship between Primer’s compute savings over Transformers and model quality follow a power law at optimal model sizes. These savings also transfer across datasets and codebases. Then, we will study Primer’s gains in an established training regime and show that it enables 4.2X compute savings at a 500M parameter size using full compute T5 training. Finally, we will demonstrate that these gains transfer to the pretraining and one-shot downstream task setup established by GPT-3 [7].

4.1 Search Task Comparison

We first analyze Primer’s performance on the search task: LM1B language modeling with sequence length 64, $\sim 35\text{M}$ model parameters, batches of 4096 tokens and 24 hours of training. We compare against the baseline models in both Tensor2Tensor (T2T) [25] 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 its *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 both Primer and Primer-EZ generalize to other hardware platforms and codebases.

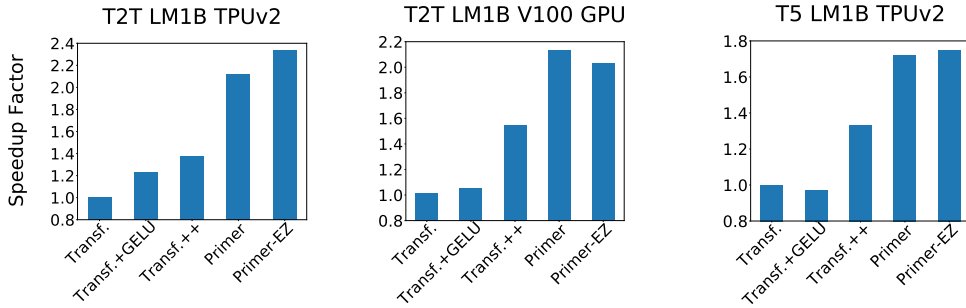


Figure 6: Comparison on the LM1B search task using 35M parameter models. “Speedup Factor” refers to the fraction of compute each model needs to reach quality parity with the vanilla Transformer trained for 24 hours. Primer and Primer-EZ both achieve over 1.7X speedup in all cases. Note that results transfer across codebases and different hardware.

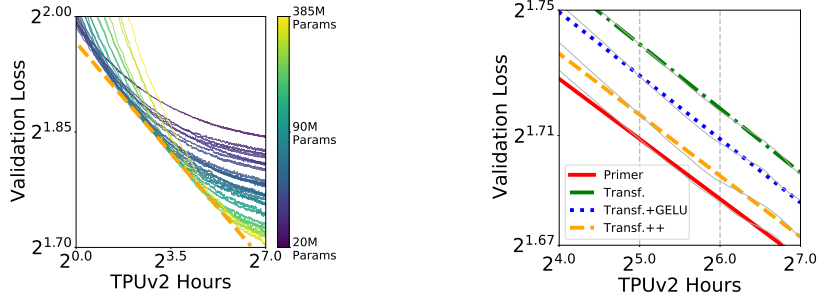


Figure 7: Left: When sweeping over optimal model sizes, the relationship between Transformer language model quality and training compute roughly obeys a power law [9]. Right: Comparison of these power law lines for varying Transformer modifications, fit with smoothed MSE loss. That the model lines are parallel to one another implies that compute savings by using superior modeling also scales as a power law with quality. Spacings between vertical dotted lines represent 2X differences in compute. Note that the x and y-axes in both plots are in log.

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 from 23M to 385M. The results, shown in Figure 7, corroborate previous claims that, at optimal parameters sizes, the relationship between compute and language model quality roughly follows a

power law [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 analyzing Primer’s inference performance across different decoding methods, serving platforms, datasets, etc., but that is beyond the scope of this work.

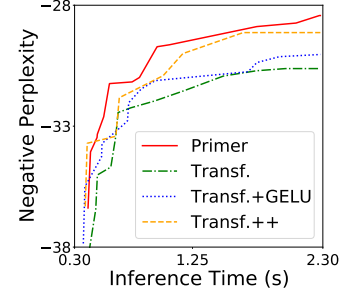


Figure 8: Pareto optimal inference comparison on LM1B. Primer demonstrates improved inference at a majority of target qualities. We observe these models have a 0.97 correlation between their train step and inference times.

4.2 Primer Transferability 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 [49, 5]; the batches are increased to 65K tokens, the sequence lengths are a longer 512, each decoder is 110M parameters ($d_{model} = 768$, $d_{ff} = 3072$, $L = 12$) and each model is trained to $\sim 525K$ steps on 4 TPUv3 chips. We also continue training each model to 1M steps to study the effect of larger compute budgets on Primer savings. The results, shown in Figure 9, indicate that the **Primer models are as strong in larger data, higher compute regimes**, as they are in the smaller LM1B regime. Compared to the vanilla baseline, Primer and Primer-EZ are at least 1.8X more efficient at the end of training on both PG19 and C4.

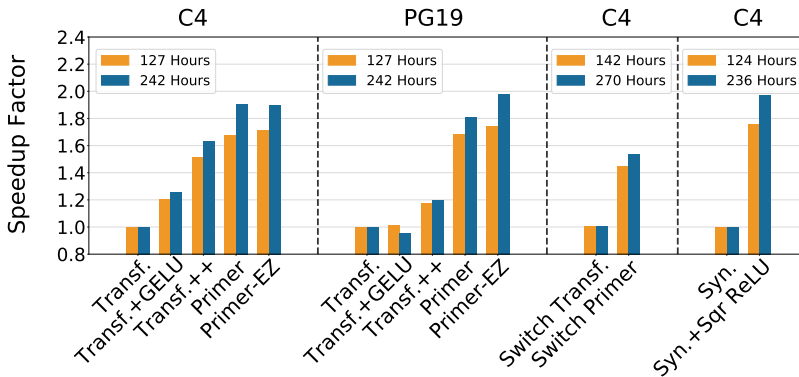


Figure 9: Comparison transferring Primer to larger datasets (C4 and PG19) and different model families (Switch Transformer and Synthesizer) in a different codebase (T5) with an order of magnitude more compute than the search task. Compared to the vanilla baseline, Primer and Primer-EZ are at least 1.8X more efficient at the end of training on both PG19 and C4. In all cases, the fraction of Primer compute savings increases as more compute is invested. Primer-EZ modifications also improve Switch Transformer (550M params) and Synthesizer (145M 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 525K and 1M steps. See Appendix A.10 for exact numbers.

Figure 9 also shows that the **Primer modifications are compatible with other efficient model families**, such as large sparse mixture-of-experts like Switch Transformer [8] and efficient Transformer approx-

improvements like Synthesizer [23]. For these experiments, we use the T5 implementations provided by Narang et al. [49]. The Primer-EZ techniques of added depthwise convolutions and squared ReLUs reduce Switch Transformer’s compute cost by a factor of 1.5X; this translates to a 0.6 perplexity improvement when controlling for compute (see Appendix A.10). Adding squared ReLUs to Synthesizer reduces training costs by a factor of 2.0X and improves perplexity by 0.7 when fully trained.

4.3 Large Scale T5 Auto-Regressive Language Model Training

In large scale compute configurations, the Primer compute savings ratios are even *higher*. To demonstrate Primer’s savings in an established high compute training setup, we scale up to the full T5 compute regime, copying Raffel et al. exactly [5]. This is the same as the C4 configuration in the previous section, but uses batches of $\sim 1\text{M}$ tokens, 64 TPUv3 chips and 537M parameters ($d_{\text{model}} = 1024$, $d_{\text{ff}} = 8192$, $L = 24$). Primer is 4.2X more compute efficient than the original T5 model and 2X more efficient than our strengthened Transformer++ baseline (Table 1).

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. This differs from the power law savings described in Section A.6. There, we use the optimal number of parameters 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. Note, this means that comparisons such as the ones given here can be “gamed” by investing more compute than is necessary for baseline models. It is for this reason that we use an exact replica of Raffel et al.’s [5] training regime: to demonstrate Primer’s savings in an already published training configuration.

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 537M 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.

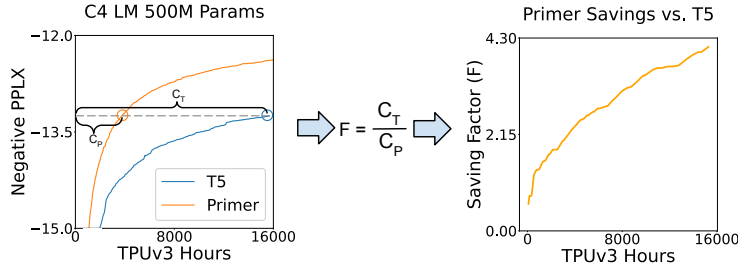


Figure 10: Compute savings of Primer vs. the original T5 architecture on C4 LM over time. The more compute invested in training, the higher the savings due to the asymptotic nature of both perplexity curves. Primer achieves the same performance as the original T5 architecture with 4.2X less compute.

4.4 Primer Transferability to Downstream One-Shot Tasks

In our final comparison, we demonstrate Primer’s improvements also hold in the *pretraining* and *one-shot downstream* task transfer regime. Recent trends in language modeling have moved towards training large models on large datasets, which is referred to as “pretraining.” These models are then transferred to unseen datasets and tasks, and, without much or any additional training, demonstrate the capacity to perform well on those “downstream” tasks [2, 51]. In the decoder-only auto-regressive language modeling configuration we study here, the most impressive results have been achieved by

GPT-3 [7], which showed that large language models can exhibit strong performance on unseen tasks given only one example – referred to as “one-shot” learning. In this section, we demonstrate that Primer’s training compute savings stretch beyond reaching a target pretraining perplexity and indeed transfer to downstream one-shot task performance.

To do this, we **replicate the GPT-3 pretraining and one-shot evaluation setup**.³ This replication is not exactly the same as the one used for GPT-3 because GPT-3 was not open sourced. Thus, these experiments are not meant to compare directly to GPT-3, as there are configuration differences. Instead, these experiments are used as a controlled comparison of the Transformer and Primer architectures. We conduct these experiments in the Lingvo codebase using a proprietary pretraining dataset. The downstream tasks are configured in the same one-shot way described by Brown et al. [7], with single prefix examples fed into each model with each task’s inputs. We compare (1) a baseline 1.9B parameter Transformer ($d_{model} = 2048$, $d_{ff} = 12288$, $L = 24$) with GELU activations, meant to approximate the GPT-3 XL architecture, and (2) a full Primer without shared QK representations, which only hurt performance according to Appendix A.7. Each model is trained using batches of $\sim 2M$ tokens using 512 TPUv4 chips for ~ 140 hours ($\sim 71.8K$ total accelerator hours or $\sim 1M$ train steps). We once again use the T5 training hyperparameters without any additional tuning.

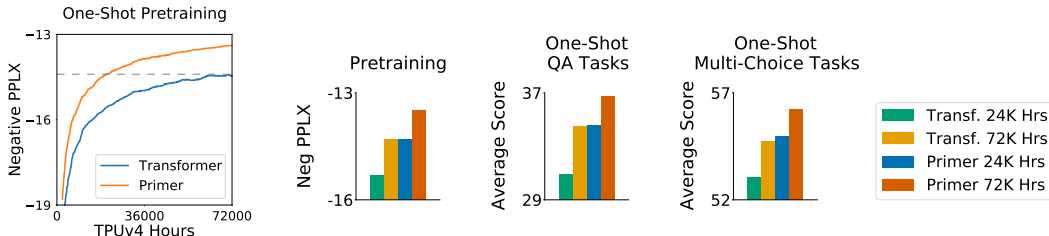


Figure 11: Comparison between **Transformer+GELU and Primer at 1.9B parameters** and varying training compute budgets on downstream one-shot tasks, similar to GPT-3. Primer achieves slightly better performance than Transformer when given 3X less pretraining compute and substantially better performance when given the same pretraining compute. Here we stop at 72K TPUv4 hours to roughly match the quality of GPT-3 XL, but the compute savings of Primer would be larger if we let the two models run longer (see Figure 10). Note, this is a crude comparison that uses averaged scores from the 27 one-shot tasks we evaluate. See Appendix A.11 (Table 6 and Figure 27) for the exact scores on each individual task.

Figure 11 shows that Primer achieves the same pretraining perplexity and one-shot downstream performance as Transformer+GELU while using **3X less compute**. Table 6 in the Appendix gives the exact performance numbers for each of the 27 evaluated downstream tasks. **Primer, despite using 3X less compute, outperforms Transformer+GELU on 5 tasks, does worse on 1 task, and performs equivalently on the remaining 21 tasks.** The same table shows that when given equivalent compute, Primer outperforms Transformer+GELU on 15 tasks, does worse on 2 tasks, and performs equivalently on the remaining 10 tasks. This result shows that not only can Primer improve language modeling perplexity, but the improvements also transfer to downstream NLP tasks.

Primer’s Return on Investment: The compute savings in this large-scale experiment demonstrate the return on investment for the Primer search. The search for Primer itself cost $\sim 2.14E+21$ FLOPs. Training Transformer for this experiment cost $\sim 2.96E+22$ FLOPs, which means the compute saved by Primer to reach the same performance is $\sim 1.98E+22$ FLOPs. Thus, for this single training, the return on investment for the architecture search is roughly 9.24X. Note that the search cost is a one-time cost, and Primer can be reused in future trainings to save more compute. More details on energy cost and carbon emission estimates can be found in Appendix A.13.

5 Conclusion

Limitations: There are limitations to this study. First, our model parameter sweeps are approximately an order of magnitude smaller than the sweeps performed in the original study by Kaplan et

³The development of the training dataset and evaluation pipeline used in this section is its own standalone work. Full details of such work will soon be released in a separate technical report.

al. [9]. Likewise, although our large-scale models use a significant amount of compute, they are still orders of magnitude smaller than state-of-the-art models such as the full-scale GPT-3 [7].

Another limitation is that we focus primarily on decoder-only models, while encoder-only [2, 3, 4] and encoder-decoder sequence models [52, 1, 6] are still widely used. In Appendix A.12, we perform encoder-decoder masked language modeling comparisons in T5, but do not study the results in significant depth. The main finding there is that, although Primer modifications improve upon vanilla Transformer, they perform only as well as Transformer++. This result suggests that architectural modifications that work well for decoder-only auto-regressive language models may not necessarily be as effective for encoder-based masked language models. Developing an architecture that also works well for masked language models is a topic of our future research.

Practical Discussion: The main motivation of this work is to develop simple and practical changes to Transformers that can be easily adopted. To that end, we provide answers to some questions that practitioners may ask:

- *Are the Primer training compute savings going to be the same in all setups?* No. Across our own provided experiments, Primer yields various compute savings. This is because the compute savings depend on hardware specifics, deep learning library operation speeds, model sample efficiencies on specific tasks, and other factors that may vary across setups. We use the exact replica of T5 training as a demonstration of what savings look like in an established configuration (4.2X), but expect results to vary across configurations.
- *Can Primer improve BERT [2]?* This work has focused on the specific task of auto-regressive language modeling, which, with the development of GPT-3, proves to be important for both traditional NLP applications as well as generative applications. We have only briefly investigated Primer’s application to masked language modeling and encoder-decoder models (Appendix A.12). Our investigations show that, while Primer improves upon vanilla Transformer, it is not obviously better than Transformer++. Thus, modifications that work well for auto-regressive language modeling may not be as effective for masked language modeling. Future work could investigate if the Primer modifications can be integrated into encoder-decoder and encoder-only models in a more effective way that can improve models like BERT. Future work could also apply the search method described here to finding better encoder-based masked language models.
- *Do hyperparameter configurations need to be retuned to use Primer?* Our intention is for Primer modifications to not require any additional hyperparameter tuning. To that end, in our experiments we did not tune any hyperparameters, and instead used the Transformer hyperparameters from established libraries. However, Primer may work even better with additional tuning.
- *Is Primer-EZ better than Primer?* In our comparison experiments, we find that Primer-EZ is sometimes better than Primer in the T5 codebase. However, in application to other codebases, such as Lingvo and T2T, we find that the full Primer can give improved performance over Primer-EZ. Thus, we recommend that practitioners first try using Primer-EZ for its ease of implementation and then move on to implementing the full Primer if they are interested in achieving further gains.

Recommendations and Future Directions: We recommend the adoption of Primer and Primer-EZ for *auto-regressive* language modeling because of their strong performance, simplicity, and robustness to hyperparameter and codebase changes. To prove their potential, we simply dropped them into established codebases and, without any changes, showed that they can give significant performance boosts. Furthermore, in practice, additional tuning could further improve their performance.

We also hope our work encourages more research into the development of efficient Transformers. For example, an important finding of this study is that small changes to activation functions can yield more efficient training. In the effort to reduce the cost of Transformers, more investment in the development of such simple changes could be a promising area for future exploration.

Acknowledgements

We thank Zhen Xu for his help with infrastructure. We also thank Gabriel Bender, Hallie Cramer, Andrew Dai, Nan Du, Yanping Huang, Daphne Ippolito, Norm Jouppi, Lluís-Miquel Munguia, Sharan Narang, Ruoming Pang, David Patterson, Yanqi Zhou, and the Google Brain Team for their help and feedback.

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. *arXiv preprint arXiv:2001.08361*, 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] 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.
- [12] Hanxiao Liu, Karen Simonyan, Oriol Vinyals, Chrisantha Fernando, and Koray Kavukcuoglu. Hierarchical representations for efficient architecture search. In *ICLR*, 2018.
- [13] David R. So, Chen Liang, and Quoc V. Le. The evolved transformer. In *ICML*, 2019.
- [14] Hanxiao Liu, Andrew Brock, Karen Simonyan, and Quoc V Le. Evolving normalization-activation layers. In *NeurIPS*, 2020.
- [15] Xin Yao. Evolving artificial neural networks. *Proceedings of the IEEE*, 87(9):1423–1447, 1999.
- [16] Jurgen Schmidhuber. Evolutionary principles in self-referential learning. (on learning how to learn: The meta-meta-... hook.). Diploma thesis, Technische Universität München, Germany, 1987.
- [17] Kenneth Stanley, Jeff Clune, Joel Lehman, and Risto Miikkulainen. Designing neural networks through neuroevolution. *Nature Machine Intelligence*, 1:24–35, 2019.
- [18] Alec Radford, Jeffrey Wu, R. Child, David Luan, Dario Amodei, and Ilya Sutskever. Language models are unsupervised multitask learners. In *Technical report, OpenAI*, 2019.

- [19] Timo Schick and Hinrich Schütze. It’s not just size that matters: Small language models are also few-shot learners. *arXiv preprint arXiv:2009.07118*, 2021.
- [20] Sinong Wang, Han Fang, Madian Khabsa, Hanzi Mao, and Hao Ma. Entailment as few-shot learner. *arXiv preprint arXiv:2104.14690*, 2021.
- [21] Tianyu Gao, Adam Fisch, and Danqi Chen. Making pre-trained language models better few-shot learners. *arXiv preprint arXiv:2012.15723*, 2020.
- [22] Jack W. Rae, Anna Potapenko, Siddhant M. Jayakumar, and T. Lillicrap. Compressive transformers for long-range sequence modelling. *ArXiv*, abs/1911.05507, 2020.
- [23] 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.
- [24] Ciprian Chelba, Tomas Mikolov, Mike Schuster, Qi Ge, Thorsten Brants, Philipp Koehn, and Tony Robinson. One billion word benchmark for measuring progress in statistical language modeling. In *Interspeech*, 2014.
- [25] 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. *arXiv preprint arXiv:1803.07416*, 2018.
- [26] Mingxing Tan and Quoc Le. Efficientnet: Rethinking model scaling for convolutional neural networks. In *ICML*, 2019.
- [27] 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.
- [28] Han Cai, Ligeng Zhu, and Song Han. Proxylessnas: Direct neural architecture search on target task and hardware. *arXiv preprint:1812.00332*, 2019.
- [29] Thomas Elsken, Jan Hendrik Metzen, and Frank Hutter. Efficient multi-objective neural architecture search via lamarckian evolution. In *ICLR*, 2019.
- [30] Esteban Real, Alok Aggarwal, Yanping Huang, and Quoc V. Le. Regularized evolution for image classifier architecture search. In *AAAI*, 2019.
- [31] Thomas Elsken, Jan Hendrik Metzen, and Frank Hutter. Neural architecture search: A survey. *Journal of Machine Learning Research*, 20(55):1–21, 2019.
- [32] Liam Li and Ameet S. Talwalkar. Random search and reproducibility for neural architecture search. In *UAI*, 2019.
- [33] Kaicheng Yu, Christian Sciuto, Martin Jaggi, Claudiu Musat, and Mathieu Salzmann. Evaluating the search phase of neural architecture search. In *ICLR*, 2020.
- [34] 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.
- [35] Esteban Real, Chen Liang, David R. So, and Quoc V. Le. Automl-zero: Evolving machine learning algorithms from scratch. In *ICML*, 2020.
- [36] Dmitry Krotov and John J. Hopfield. Dense associative memory for pattern recognition. In *Advances in Neural Information Processing Systems*, 2016.
- [37] Siddhant M. Jayakumar, Jacob Menick, Wojciech M. Czarnecki, Jonathan Schwarz, Jack W. Rae, Simon Osindero, Y. Teh, Tim Harley, and Razvan Pascanu. Multiplicative interactions and where to find them. In *ICLR*, 2020.
- [38] Yann N Dauphin, Angela Fan, Michael Auli, and David Grangier. Language modeling with gated convolutional networks. In *ICML*, 2017.
- [39] Noam Shazeer. Glu variants improve transformer. *arXiv preprint arXiv:2002.05202*, 2020.
- [40] Dan Hendrycks and Kevin Gimpel. Bridging nonlinearities and stochastic regularizers with gaussian error linear units. *arXiv preprint arXiv:1606.08415*, 2016.
- [41] 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.

- [42] 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.
- [43] Haiping Wu, Bin Xiao, Noel Codella, Mengchen Liu, Xiyang Dai, Lu Yuan, and Lei Zhang. CvT: Introducing convolutions to vision transformers. *arXiv preprint arXiv:2103.15808*, 2021.
- [44] Alexei Baevski and Michael Auli. Adaptive input representations for neural language modeling. In *arXiv preprint arXiv:1809.10853*, 2019.
- [45] 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.
- [46] Jimmy Ba, Jamie Kiros, and Geoffrey E. Hinton. Layer normalization. *arXiv preprint arXiv:1607.06450*, 2016.
- [47] Biao Zhang and Rico Sennrich. Root mean square layer normalization. In *NeurIPS*, 2019.
- [48] Prajit Ramachandran, Barret Zoph, and Quoc V. Le. Searching for activation functions. *arXiv preprint arXiv:1710.05941*, 2018.
- [49] Sharan Narang, Hyung Won Chung, Yi Tay, William Fedus, Thibault F  vry, 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.
- [50] Jonathan Shen, P. Nguyen, Yonghui Wu, Z. Chen, M. Chen, Ye Jia, Anjuli Kannan, T. Sainath, Yuan Cao, C. Chiu, Yanzhang He, J. Chorowski, Smit Hinsu, S. Laurenzo, James Qin, Orhan Firat, Wolfgang Macherey, Suyog Gupta, Ankur Bapna, Shuyuan Zhang, Ruoming Pang, Ron J. Weiss, Rohit Prabhavalkar, Qiao Liang, Benoit Jacob, Bowen Liang, HyoukJoong Lee, Ciprian Chelba, S  bastien Jean, Bo Li, M. Johnson, Rohan Anil, Rajat Tibrewal, Xiaobing Liu, Akiko Eriguchi, Navdeep Jaitly, Naveen Ari, Colin Cherry, Parisa Haghani, Otavio Good, Youlong Cheng, R.   lvarez, Isaac Caswell, Wei-Ning Hsu, Zongheng Yang, Kuan-Chieh Wang, Ekaterina Gonina, Katrin Tomanek, Ben Vanik, Zelin Wu, Llion Jones, M. Schuster, Y. Huang, Dehao Chen, Kazuki Irie, George F. Foster, John Richardson, Uri Alon, and E. al. Lingvo: a modular and scalable framework for sequence-to-sequence modeling. *ArXiv*, abs/1902.08295, 2019.
- [51] Andrew M. Dai and Quoc V. Le. Semi-supervised sequence learning. In *Advances in Neural Information Processing Systems*, 2015.
- [52] Ilya Sutskever, Oriol Vinyals, and Quoc V Le. Sequence to sequence learning with neural networks. In *Advances in Neural Information Processing Systems*, 2014.
- [53] Zohar Karnin, Tomer Koren, and Oren Somekh. Almost optimal exploration in multi-armed bandits. In *Proceedings of the 30th International Conference on Machine Learning*, 2013.
- [54] 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.
- [55] Thomas Helmuth, N. McPhee, and L. Spector. Program synthesis using uniform mutation by addition and deletion. *Proceedings of the Genetic and Evolutionary Computation Conference*, 2018.
- [56] Noam M. Shazeer and Mitchell Stern. Adafactor: Adaptive learning rates with sublinear memory cost. *arXiv preprint arXiv:1804.04235*, 2018.
- [57] Peter Shaw, Jakob Uszkoreit, and Ashish Vaswani. Self-attention with relative position representations. In *NAACL-HLT*, 2018.
- [58] Taku Kudo and J. Richardson. Sentencepiece: A simple and language independent subword tokenizer and detokenizer for neural text processing. In *EMNLP*, 2018.
- [59] David Patterson, Joseph Gonzalez, Quoc V. Le, Chen Liang, Llu  s-Miquel Mungu  a, D. Rothchild, David R. So, Maud Texier, and J. Dean. Carbon emissions and large neural network training. *arXiv preprint arXiv:2104.10350*, 2021.
- [60] 24/7 carbon-free energy: Methodologies and metrics. <https://www.gstatic.com/gumdrop/sustainability/24x7-carbon-free-energy-methodologies-metrics.pdf>. Accessed: 2021-09-14.

A Appendix

A.1 TensorFlow Primitives Vocabulary

Name	TF Function	Argument Mapping			
		Input 1	Input 2	Constant	Dim Size
ADD	tf.math.add	x	y	-	-
DIFFERENCE	tf.math.subtract	x	y	-	-
DIVIDE	tf.math.divide	x	y	-	-
MULTIPLY	tf.math.multiply	x	y	-	-
ABS ROOT	tf.math.sqrt(tf.abs(x))	x	-	-	-
SQUARE	tf.math.square	x	-	-	-
EXP	tf.exp	x	-	-	-
LOG	tf.log(tf.abs(x))	x	-	-	-
C MUL	tf.math.multiply	x	-	y	-
ABS	tf.abs	x	-	-	-
RECIP	tf.math.reciprocal_no_nan	x	-	-	-
SIGN	tf.sign	x	-	-	-
COS	tf.cos	x	-	-	-
SIN	tf.sin	x	-	-	-
TANH	tf.tanh	x	-	-	-
MAX	tf.math.maximum	x	-	y	-
MIN	tf.math.minimum	x	-	y	-
SCALE	x+tf.Variable()	x	-	-	-
SHIFT	x*tf.Variable()	x	-	-	-
SIGMOID	tf.sigmoid	x	-	-	-
MASK	tf.linalg.band_part	input	-	-	-
CUM PROD	tf.math.cumprod	x	-	-	-
CUM SUM	tf.math.cumsum	x	-	-	-
RED MEAN	tf.reduce_mean	input_tensor	-	-	-
RED SUM	tf.reduce_sum	input_tensor	-	-	-
RED MIN	tf.reduce_min	input_tensor	-	-	-
RED MAX	tf.reduce_max	input_tensor	-	-	-
RED PROD	tf.reduce_prod	input_tensor	-	-	-
MAT MUL	tf.matmul	a	b	-	-
T-MAT MUL	tf.matmul(transpose_b=True)	a	b	-	-
CONV 1X1	tf.layers.dense	inputs	-	-	units
CONV 3X1	tf.nn.conv1d	input	-	-	filters
CONV 7X1	tf.nn.conv1d	input	-	-	filters
CONV 15X1	tf.nn.conv1d	input	-	-	filters
CONV 31X1	tf.nn.conv1d	input	-	-	filters
DCONV 3X1	tf.nn.depthwise_conv2d	input	-	-	filters
DCONV 7X1	tf.nn.depthwise_conv2d	input	-	-	filters
DCONV 15X1	tf.nn.depthwise_conv2d	input	-	-	filters
DCONV 31X1	tf.nn.depthwise_conv2d	input	-	-	filters

Table 2: TensorFlow (TF) Primitives Vocabulary. “Name” is the name of the operation in our search space. “TF Function” is the TensorFlow function that the name is mapped to when a DNA instruction is being converted to a line of TensorFlow code. “Argument Mapping” describes how the values in a DNA’s argument set are mapped to the corresponding TensorFlow function arguments. This vocabulary is largely constructed from the lowest level TF operations needed to create Transformers (see Appendix A.5). We additionally extend those operations to include adjacent operations; for example, we extend MAX to also include MIN, extend RED SUM to include RED PRODUCT, and extend CONV 1X1 to include CONV 3X1. We also add commonly used math primitives such as SIN and ABS.

A.2 Constructing TensorFlow Graphs

TensorFlow graphs are built from DNA programs as described in Section 2 of the main text. Here we provide additional implementation details.

Relative Dimensions: We use relative dimensions [13] instead of absolute dimensions for each instruction’s “dimension size” argument. This allows us to resize the models to fit within our parameter limits (32M to 38M parameters). The vocabulary for these relative dimensions is [1, 2, 4, 8, 12, 16, 24, 32, 48, 64]. This vocabulary was not tuned.

Values Bank: For “constant” and “dimension size” argument fields, we create a shared bank of values that each instruction references. The constants bank holds 2 values and the dimension sizes bank holds 6 values; these numbers were not tuned. Instead of each instruction possessing their own individual values for these arguments, they instead hold an index to these shared banks. This allows multiple instructions to share the same value and to change simultaneously when that value is changed. For example, each of the individual attention multi-head projections for Q , K and V start off sharing the same output dimension size so that they all change simultaneously if that value changes. See A.4 for an example of how these bank values are mutated.

Causal Masking: An important part of teacher-forced language model training is that positions cannot “see” the token they are trying to predict. Each position should only get information from previous positions, otherwise the model will be degenerate when the targets are not provided. To enforce this causal constraint we add additional overhead to operations that move information spatially to mask out any information from future positions. For example, when applying convolutions we follow the standard practice of shifting the inputs spatially by $(\text{KERNEL WIDTH} - 1)$ so that each position only receives information from previous positions.

Branching: To enable multi-head capabilities for the Transformer search seed, we add a meta argument to our instructions called “branching.” This argument can take any value in [1, 2, 4, 8, 16] and determines how many times that instruction is executed in parallel, with the resulting tensors being concatenated together along their embedding axes. Branching can be used with any of the TensorFlow primitives as well as with any of a DNA’s subprograms. This allows us to initialize the search with multi-head self-attention by branching SUBPROGRAM 1 (self-attention) 8 times (see Appendix A.5 for subprogram implementations). Primer does not utilize this branching capability in any meaningful way, beyond using the initialized multi-head attention.

Resolving Dimension Mismatches: We do not constrain how tensor dimensions can be mutated and so programs may be invalid because they perform binary operations on tensors with incompatible sizes. For example, a program may describe adding together two tensors with differing embedding sizes. To resolve these dimension mismatch issues we deterministically pseudorandomly set one of the tensor dimensions to match the other.

A.3 Halving Hurdles

We configure our hurdles [13] such that the top 50% of individuals passes each hurdle, according to fitness. We space the hurdles in such a way that the expected amount of compute devoted to training each hurdle band is roughly equal at the end of the search. That is, given that our maximum amount of training compute for an individual is 7 hours or 25,200 seconds (s), we construct hurdles at the 812.9s, 2438.7s, 5690.3s, and 12,193.5s marks. Thus, 1/5 of the compute budget is devoted to training every individual up to the first hurdle (812.9s), 1/5 of the compute budget is devoted to training the $\sim 50\%$ of individuals that are trained from the first to the second hurdle ($2438.7s - 812.9s = 1625.8s$), 1/5 of the compute budget is devoted to training the $\sim 25\%$ of individuals that are trained from the second to the third hurdle ($5690.3s - 2438.7s = 3251.6s$), etc. This configuration strategy, which we refer to as “halving hurdles,” requires setting only one hyperparameter, the number of hurdles, and removes the need to set hurdle threshold values and comparison steps, as has been previously done [13, 35]. We choose four hurdles because five hurdles would require the first hurdle to be anchored at less than ten minutes of training, which we find empirically to be too noisy of a signal. Using hurdles in this way decreases the average train time per model to 4064s or about 1 hour and 8 minutes, reducing the compute cost by a factor of $\sim 6.2X$.

This strategy is not unlike bandit algorithms such as Successive Halving[53] and Hyperband[54], however we do not use a static population of individuals created a priori, but integrate our halving with the changing evolutionary population.

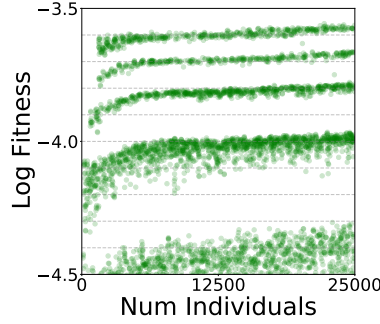


Figure 12: Halving hurdles from our Primer search. Each dot represents the final fitness of an individual generated by evolution. Different “bands” form because each hurdle has a different training allowance. All bands see improvement over time, meaning that the median fitness improves for all compute allowances. This correlation between a model’s performances at different training budgets allows us to reduce our total search cost by roughly a factor of 6.2X.

A.4 Evolution Search Details

We use Regularized Evolution [30] with a population size of 100 and a tournament selection size of 10. These values were not tuned. The mutations we use are as follows.

Mutations: To create new candidates in our search, we uniform randomly select a *parent* from our search population and apply a single *mutation* to it. We employ five different mutation types (selections and decisions are performed uniform randomly unless specified otherwise):

- *Delete:* Remove an instruction from a subprogram.
- *Insert:* Create an instruction and insert it into a subprogram.
- *Delete and Insert:* Perform a delete mutation followed by an insert mutation [55].
- *Mutate Field:* Select a field from an instruction and change its value.
- *Swap:* Swap the position of two instructions in a randomly selected subprogram. The input tensors for each instruction are also swapped so that the net effect is switching the positions of the instructions in the compute graph.
- *Mutate Bank Value:* Change the value of a relative tensor dimension or constant in the corresponding bank. The values for relative tensor dimensions are selected from their vocabulary (see Appendix A.2). The values for constants are changed according to $c_{new} := c_{prev} \cdot 10^X + Y$ for previous value c_{prev} , new value c_{new} and random variables $X, Y \sim N(0, 1)$.

After a mutation is applied, we run a light check to see if the resulting candidate’s compute graph is exactly equivalent to the parent’s compute graph. If it is, we perform another mutation.

A.5 Transformer and Primer Program Comparisons

Here we present the programs for both the Transformer seed and the discovered Primer model. Table 3 is a key that maps operation names to graph symbols for subsequent graphs. Figures 13 to 22 depict the subprograms for each model with the Primer changes highlighted in orange. Figure 23 depicts the full compute graphs for each model, with all subprograms resolved to their constituent primitives. Figures 24 and 25 depict the DNA programs for Transformer and Primer with all subprograms resolved and all instruction bank values plugged in.

Name	Graphing symbol
ADD	+
DIFFERENCE	−
DIVIDE	÷
MULTIPLY	×
ABS ROOT	$\sqrt{\quad}$
SQUARE	x^2
EXP	e^x
LOG	Log
C MUL	$\times C$
ABS	$ x $
RECIP	Recip
SIGN	Sign
COS	Cos
SIN	Sin
TANH	Tanh
MAX	Max
MIN	Min
SCALE	Scale
SHIFT	Shift
SIGMOID	Sigm
MASK	Mask
CUM PROD	Cum Prod
CUM SUM	Cum Sum
RED MEAN	Reduce Mean
RED SUM	Reduce Sum
RED MIN	Reduce Min
RED MAX	Reduce Max
RED PROD	Reduce Prod
MAT MUL	$M \times N$
T-MAT MUL	$M \times N^T$
CONV 1X1	Conv 1x1
CONV 3X1	Conv 3x1
CONV 7X1	Conv 7x1
CONV 15X1	Conv 15x1
CONV 31X1	Conv 31x1
DCONV 3X1	D-wise 3x1
DCONV 7X1	D-wise 7x1
DCONV 15X1	D-wise 15x1
DCONV 31X1	D-wise 31x1

Table 3: Key for primitives mapped to corresponding symbols used in the following graphs.

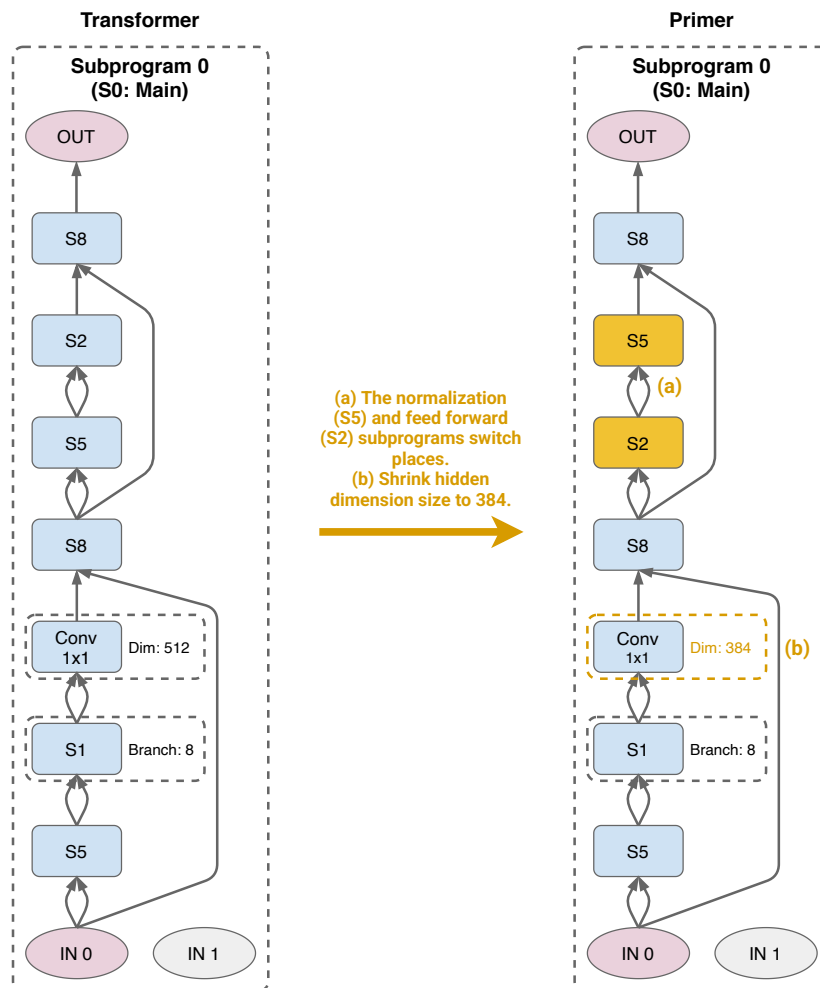


Figure 13: Main subprograms. Changes are highlighted in orange.



Figure 14: Attention subprograms. Changes are highlighted in orange.

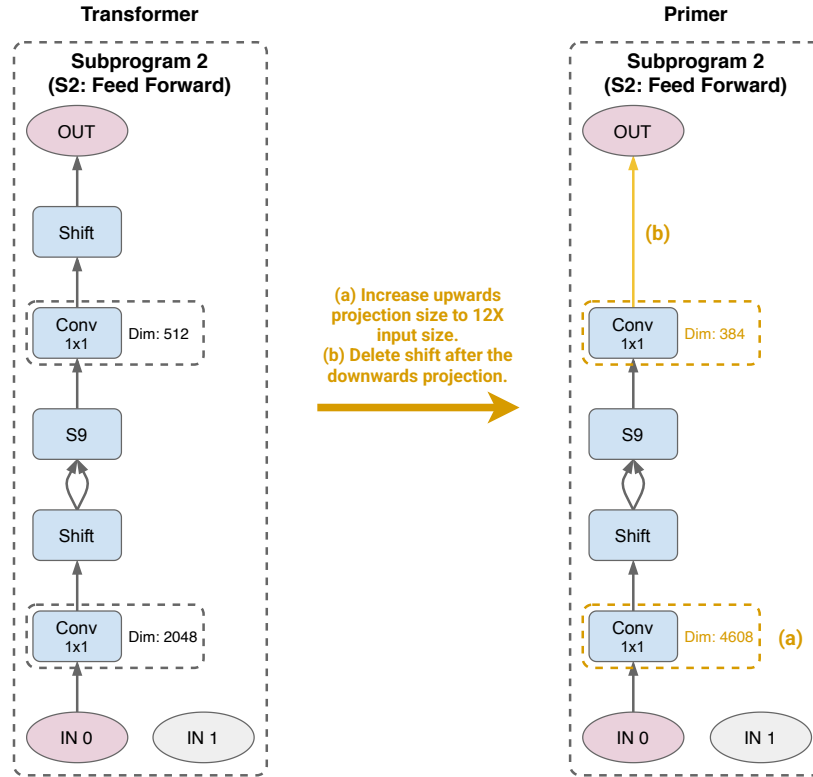


Figure 15: Feed forward subprograms. Changes are highlighted in orange.

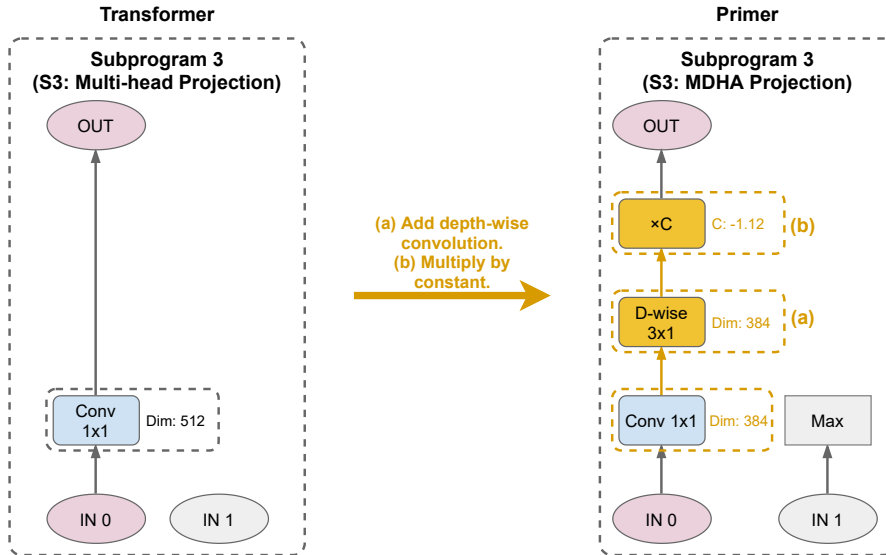


Figure 16: Multi-head projection subprograms. Changes are highlighted in orange.

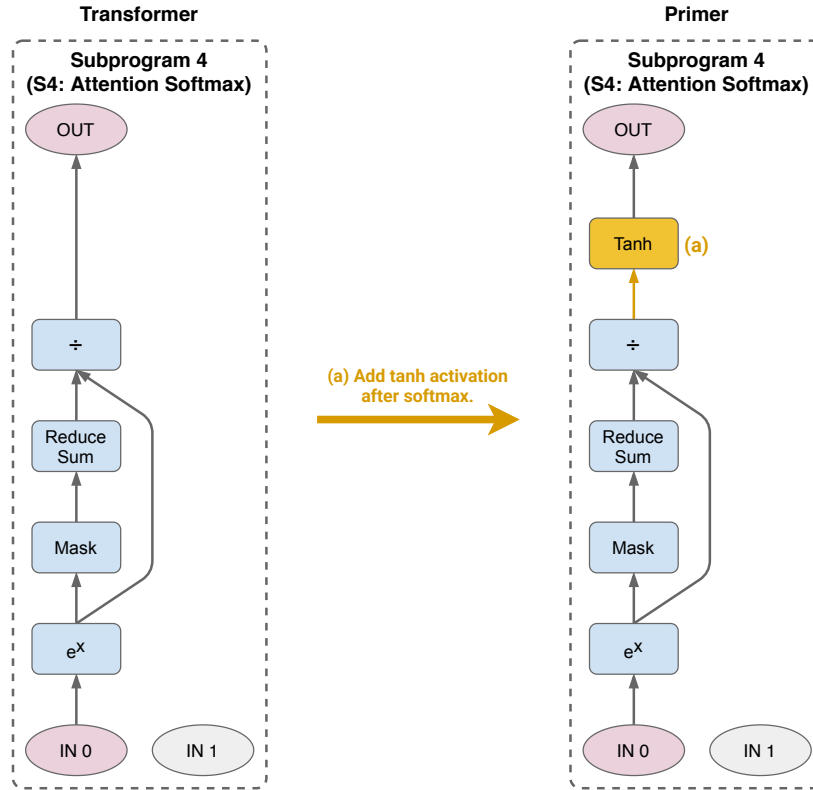


Figure 17: Sigmoid subprograms. Changes are highlighted in orange.

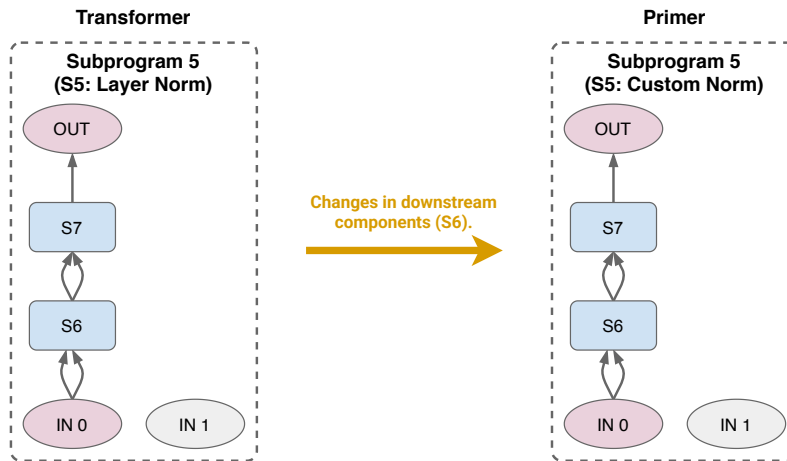


Figure 18: Normalization subprograms. Changes to this subprogram are realized in downstream changes to S6.

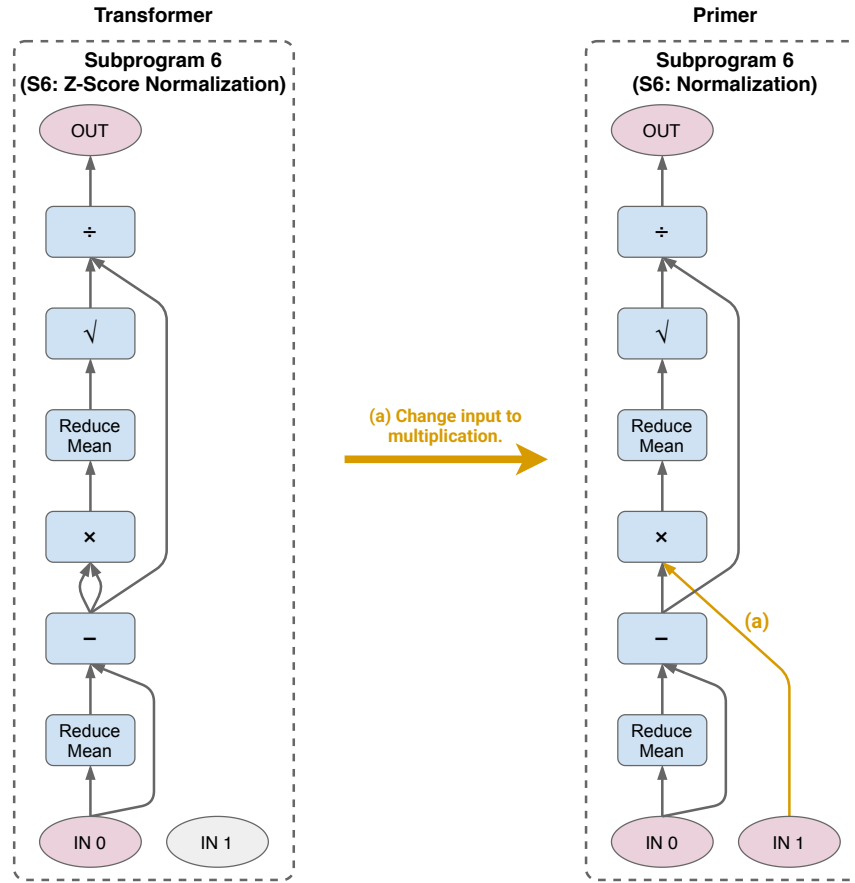


Figure 19: Z-score normalization subprograms. Changes are highlighted in orange.

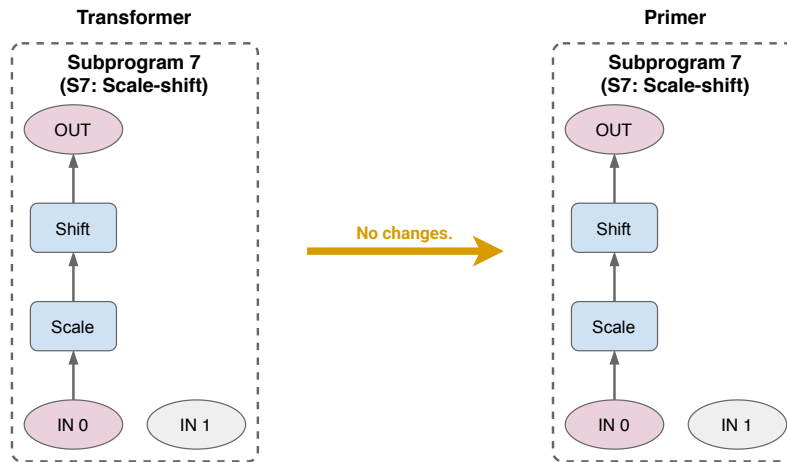


Figure 20: Scale-shift subprograms. No changes here.

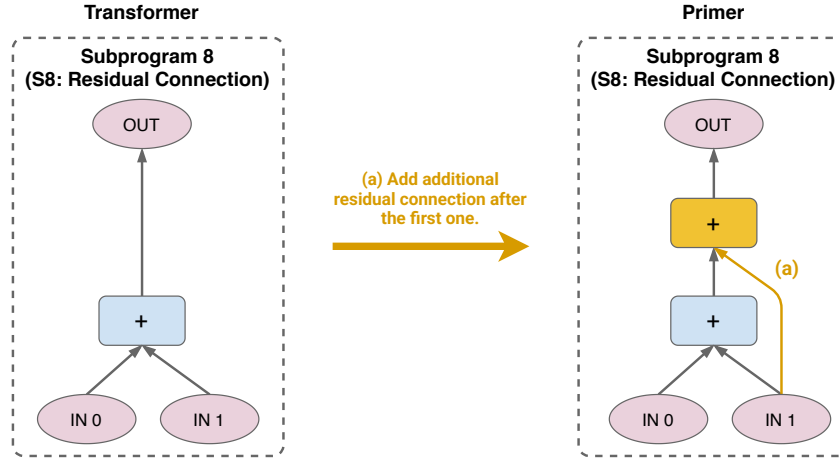


Figure 21: Residual connection subprograms. This change is essentially a functional no-op.

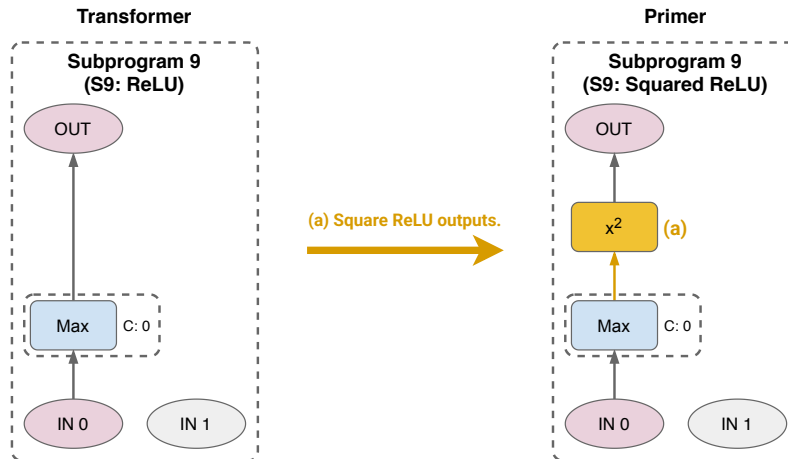


Figure 22: Activation function subprograms. Changes are highlighted in orange.

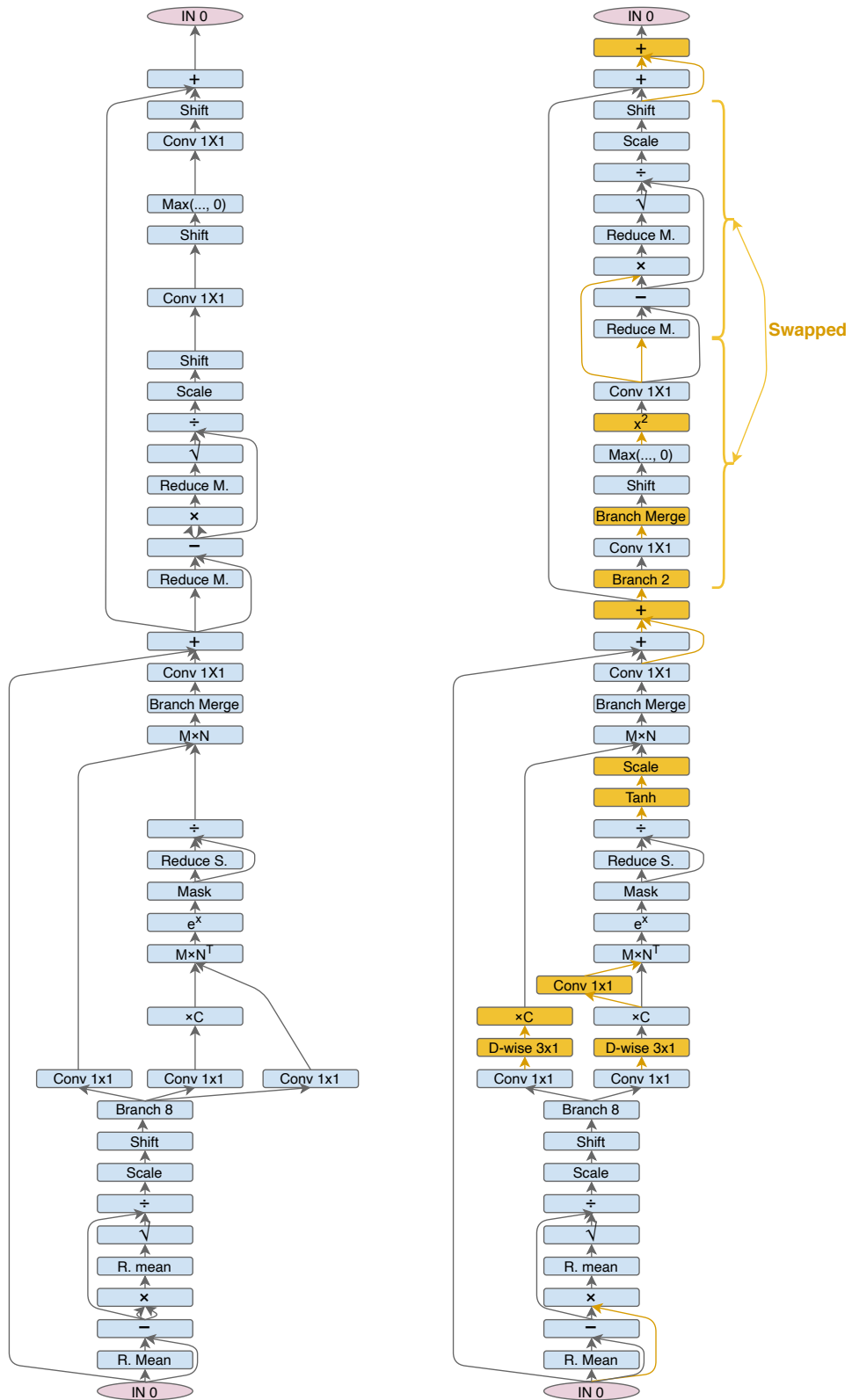


Figure 23: Comparison of Transformer (Left) and Primer (Right) programs, with all subprograms resolved to their constituent primitives. Primer differences are highlighted in orange.

TRANSFORMER

(0)	INPUT				
(1)	INPUT				
(2)	REDUCE_MEAN	In0: 0	In1: 0	Dim: 128	C: 0.00
(3)	DIFFERENCE	In0: 0	In1: 2	Dim: 128	C: 0.00
(4)	MULTIPLY	In0: 3	In1: 3	Dim: 128	C: 0.00
(5)	REDUCE_MEAN	In0: 4	In1: 4	Dim: 128	C: 0.00
(6)	ABS_SQUARE_ROOT	In0: 5	In1: 5	Dim: 128	C: 0.00
(7)	DIVIDE	In0: 3	In1: 6	Dim: 128	C: 0.00
(8)	SCALE	In0: 7	In1: 7	Dim: 128	C: 0.00
(9)	SHIFT	In0: 8	In1: 8	Dim: 128	C: 0.00
(10)	BRANCH_8_INPUT_1	In0: 9	In1: 9	Dim: 128	C: 0.00
(11)	BRANCH_8_INPUT_2	In0: 9	In1: 9	Dim: 128	C: 0.00
(12)	DENSE	In0: 10	In1: 10	Dim: 64	C: 0.00
(13)	DENSE	In0: 10	In1: 10	Dim: 64	C: 0.00
(14)	CONSTANT_MUL	In0: 13	In1: 13	Dim: 128	C: 0.12
(15)	DENSE	In0: 10	In1: 10	Dim: 64	C: 0.00
(16)	TRANSPOSE_MAT_MUL	In0: 14	In1: 12	Dim: 128	C: 0.00
(17)	EXP	In0: 16	In1: 16	Dim: 128	C: 0.00
(18)	EMBEDDING_MASK	In0: 17	In1: 17	Dim: 128	C: 0.00
(19)	REDUCE_SUM	In0: 18	In1: 18	Dim: 128	C: 0.00
(20)	DIVIDE	In0: 18	In1: 19	Dim: 128	C: 0.00
(21)	MAT_MUL	In0: 20	In1: 15	Dim: 128	C: 0.00
(22)	BRANCH_MERGE	In0: 21	In1: 21	Dim: 512	C: 0.00
(23)	DENSE	In0: 22	In1: 22	Dim: 512	C: 0.00
(24)	ADD	In0: 0	In1: 23	Dim: 128	C: 0.00
(25)	REDUCE_MEAN	In0: 24	In1: 24	Dim: 128	C: 0.00
(26)	DIFFERENCE	In0: 24	In1: 25	Dim: 128	C: 0.00
(27)	MULTIPLY	In0: 26	In1: 26	Dim: 128	C: 0.00
(28)	REDUCE_MEAN	In0: 27	In1: 27	Dim: 128	C: 0.00
(29)	ABS_SQUARE_ROOT	In0: 28	In1: 28	Dim: 128	C: 0.00
(30)	DIVIDE	In0: 26	In1: 29	Dim: 128	C: 0.00
(31)	SCALE	In0: 30	In1: 30	Dim: 128	C: 0.00
(32)	SHIFT	In0: 31	In1: 31	Dim: 128	C: 0.00
(33)	DENSE	In0: 32	In1: 32	Dim: 2048	C: 0.00
(34)	SHIFT	In0: 33	In1: 33	Dim: 128	C: 0.00
(35)	MAX	In0: 34	In1: 34	Dim: 128	C: 0.00
(36)	DENSE	In0: 35	In1: 35	Dim: 512	C: 0.00
(37)	SHIFT	In0: 36	In1: 36	Dim: 128	C: 0.00
(38)	ADD	In0: 24	In1: 37	Dim: 128	C: 0.00

Figure 24: List of instructions for Transformer program, with all subprograms resolved to their constituent primitives.

PRIMER

(0)	INPUT				
(1)	INPUT				
(2)	REDUCE_MEAN	In0: 0	In1: 0	Dim: 768	C: -1.12
(3)	DIFFERENCE	In0: 0	In1: 2	Dim: 768	C: -1.12
(4)	MULTIPLY	In0: 3	In1: 0	Dim: 768	C: -1.12
(5)	REDUCE_MEAN	In0: 4	In1: 4	Dim: 768	C: -1.12
(6)	ABS_SQUARE_ROOT	In0: 5	In1: 5	Dim: 768	C: -1.12
(7)	DIVIDE	In0: 3	In1: 6	Dim: 768	C: -1.12
(8)	SCALE	In0: 7	In1: 7	Dim: 768	C: -1.12
(9)	SHIFT	In0: 8	In1: 8	Dim: 384	C: -0.57
(10)	BRANCH_8_INPUT_1	In0: 9	In1: 9	Dim: 768	C: -1.12
(11)	BRANCH_8_INPUT_2	In0: 9	In1: 9	Dim: 768	C: -1.12
(12)	MAX	In0: 10	In1: 10	Dim: 768	C: -0.57
(13)	DENSE	In0: 10	In1: 10	Dim: 48	C: -1.12
(14)	DEPTHWISE_CONV_3X1	In0: 13	In1: 10	Dim: 384	C: -1.12
(15)	CONSTANT_MUL	In0: 14	In1: 14	Dim: 384	C: -1.12
(16)	MAX	In0: 11	In1: 11	Dim: 768	C: -0.57
(17)	DENSE	In0: 10	In1: 10	Dim: 48	C: -1.12
(18)	DEPTHWISE_CONV_3X1	In0: 17	In1: 10	Dim: 384	C: -1.12
(19)	CONSTANT_MUL	In0: 18	In1: 18	Dim: 384	C: -1.12
(20)	DENSE	In0: 19	In1: 11	Dim: 48	C: -1.12
(21)	MAX	In0: 10	In1: 10	Dim: 768	C: -0.57
(22)	DENSE	In0: 10	In1: 10	Dim: 48	C: -1.12
(23)	DEPTHWISE_CONV_3X1	In0: 22	In1: 10	Dim: 384	C: -1.12
(24)	CONSTANT_MUL	In0: 23	In1: 23	Dim: 384	C: -1.12
(25)	TRANSPOSE_MAT_MUL	In0: 20	In1: 19	Dim: 768	C: -1.12
(26)	EXP	In0: 25	In1: 25	Dim: 768	C: -1.12
(27)	EMBEDDING_MASK	In0: 26	In1: 26	Dim: 768	C: -1.12
(28)	REDUCE_SUM	In0: 27	In1: 27	Dim: 768	C: -1.12
(29)	DIVIDE	In0: 27	In1: 28	Dim: 768	C: -1.12
(30)	TANH	In0: 29	In1: 25	Dim: 384	C: -1.12
(31)	SCALE	In0: 30	In1: 19	Dim: 384	C: -1.12
(32)	MAT_MUL	In0: 31	In1: 24	Dim: 768	C: -1.12
(33)	BRANCH_MERGE	In0: 32	In1: 32	Dim: 384	C: -1.12
(34)	DENSE	In0: 33	In1: 33	Dim: 384	C: -1.12
(35)	ADD	In0: 0	In1: 34	Dim: 768	C: -1.12
(36)	ADD	In0: 35	In1: 34	Dim: 768	C: -1.12
(37)	BRANCH_2_INPUT_1	In0: 36	In1: 36	Dim: 2304	C: -1.12
(38)	BRANCH_2_INPUT_2	In0: 36	In1: 36	Dim: 2304	C: -1.12
(39)	DENSE	In0: 37	In1: 38	Dim: 2304	C: -1.12
(40)	BRANCH_MERGE	In0: 39	In1: 39	Dim: 4608	C: -1.12
(41)	SHIFT	In0: 40	In1: 40	Dim: 768	C: -1.12
(42)	MAX	In0: 41	In1: 41	Dim: 768	C: -0.57
(43)	SQUARE	In0: 42	In1: 41	Dim: 768	C: -1.12
(44)	DENSE	In0: 43	In1: 43	Dim: 384	C: -1.12
(45)	REDUCE_MEAN	In0: 44	In1: 44	Dim: 768	C: -1.12
(46)	DIFFERENCE	In0: 44	In1: 45	Dim: 768	C: -1.12
(47)	MULTIPLY	In0: 46	In1: 44	Dim: 768	C: -1.12
(48)	REDUCE_MEAN	In0: 47	In1: 47	Dim: 768	C: -1.12
(49)	ABS_SQUARE_ROOT	In0: 48	In1: 48	Dim: 768	C: -1.12
(50)	DIVIDE	In0: 46	In1: 49	Dim: 768	C: -1.12
(51)	SCALE	In0: 50	In1: 50	Dim: 768	C: -1.12
(52)	SHIFT	In0: 51	In1: 51	Dim: 384	C: -0.57
(53)	ADD	In0: 36	In1: 52	Dim: 768	C: -1.12
(54)	ADD	In0: 53	In1: 52	Dim: 768	C: -1.12

Figure 25: List of instructions for Primer program, with all subprograms resolved to their constituent primitives.

A.6 Exact LM1B Numbers

Model	Params	Train Steps	Step/Sec	PPLX	Speedup
	Tensor2Tensor, TPUv2				
Vanilla Transformer	35M	1.9M	22.4	35.44 +/- 0.30	-
Transformer+GELU	35M	1.9M	22.4	35.00 +/- 0.12	1.23 +/- 0.07
Transformer++	35M	1.9M	22.0	34.87 +/- 0.46	1.37 +/- 0.24
Primer	34M	1.9M	21.7	33.77 +/- 0.15	2.12 +/- 0.09
Primer-EZ	35M	1.8M	21.0	33.53 +/- 0.09	2.34 +/- 0.04
Transformer+MDHA	35M	1.8M	21.0	34.26 +/- 0.12	1.76 +/- 0.06
Transformer+Sep Conv	35M	1.8M	21.0	34.34 +/- 0.10	1.54 +/- 0.05
	Tensor2Tensor, V100				
Vanilla Transformer	35M	1.3M	15.4	37.19 +/- 0.07	-
Transformer+GELU	35M	1.2M	14.1	37.11 +/- 0.02	1.05 +/- 0.02
Transformer++	35M	1.3M	14.7	36.23 +/- 0.11	1.54 +/- 0.05
Primer	34M	1.2M	13.8	35.06 +/- 0.15	2.13 +/- 0.11
Primer-EZ	35M	1.1M	13.3	35.16 +/- 0.13	2.03 +/- 0.09
	T5, TPUv2				
Vanilla Transformer	35M	2.1M	23.9	23.30 +/- 0.02	-
Transformer+GELU	35M	2.1M	23.8	23.39 +/- 0.02	0.97 +/- 0.03
Transformer++	35M	2.1M	24.2	23.04 +/- 0.02	1.33 +/- 0.05
Evolved Transformer	38M	1.6M	18.7	23.08 +/- 0.02	1.23 +/- 0.02
Primer	36M	2.0M	22.9	22.71 +/- 0.03	1.72 +/- 0.01
Primer-EZ	36M	2.0M	22.5	22.62 +/- 0.02	1.75 +/- 0.03

Table 4: Comparison on the search task, auto-regressive language modeling on LM1B, across two different hardware platforms (TPUv2s and V100 GPUs) and two different libraries (Tensor2Tensor and T5), using those libraries’ default hyperparameters. This table contains the precise numbers for Figure 6. “Speedup” describes the fraction of compute used by each model to achieve the same results as the vanilla Transformer baseline trained with the full compute budget. Even though Primer was developed in Tensor2Tensor using TPUv2s, it shows strong performance on GPU and in T5. Perplexity is reported with respect to each library’s default tokenization.

A.7 Ablation and Insertion Studies

One of the core motivations of this work is to develop simple and robust Transformer modifications. To that end, we study the individual effectiveness of each Primer modification, described in Section 3 of the main text. We measure this effectiveness using insertion and ablation studies. In the insertion studies we add each modification in isolation to a vanilla Transformer. In the ablation studies we remove each modification from Primer one at a time. We are interested in how these modifications affect performance not just in our search library, Tensor2Tensor, but also in other libraries. Thus, we perform these insertion and ablation studies in a different library, T5, as a well, and use modification transferability as the key guiding metric for our modeling recommendations.

The results of these studies are shown in Figure 26. “Normalized PPLX Delta” describes the degree to which a modification helps or hurts performance. For baseline perplexity, P_b , and modification perplexity, P_m , “Normalized PPLX Delta” is defined as $\frac{P_b - P_m}{P_b}$ in the insertion study and $\frac{P_m - P_b}{P_b}$ for the ablation study. These definitions differ so that a positive value always indicates that the modification is good and a negative value always indicates that the modification is bad. Three techniques are beneficial in all scenarios. The first is “12X proj,” which increases the size of the Transformer feed forward upwards projection while controlling for parameters. We find this works well for smaller models but is not useful at larger sizes. The second two, MDHA and squared ReLUs, are the defining modifications of Primer-EZ, a simpler model that captures much of the gains of the full Primer.

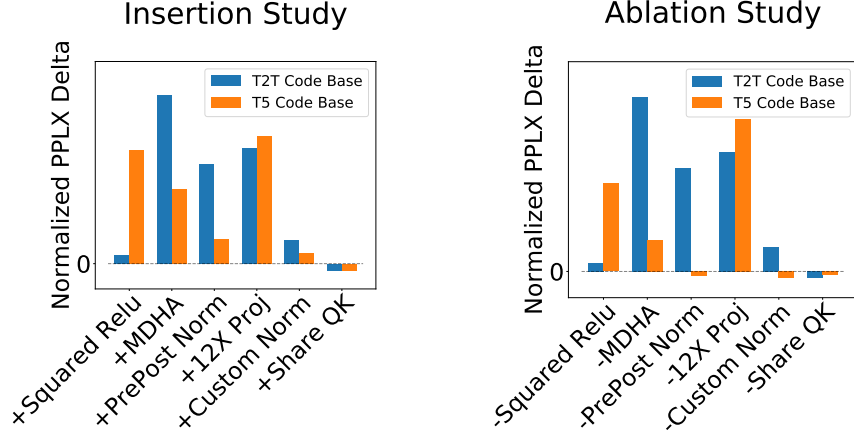


Figure 26: Investigation into transferability of Primer modifications on LM1B at ~ 35 M parameters. In the “Insertion Study” we insert each of the modifications into a vanilla Transformer. In the “Ablation Study,” we remove each modification from Primer. “Normalized PPLX Delta” indicates the degree to which the treated models are affected by these modifications; values are normalized to be comparable across code bases and so that positive values indicate beneficial techniques in both studies. Likewise, negative values indicate harmful techniques in both studies.

A.8 Full Training Details

In all experiments, we use previously published hyperparameter settings that were tuned for Transformer, with regularization disabled and no additional tuning for Primer. In Tensor2Tensor (T2T) these are the TRANSFORMER_TPU hyperparameters and in T5 and Lingvo these are the open-sourced parameters used in previous T5 studies [5, 49]. They both specify an Adafactor optimizer [56], with 10K warmup steps at a learning rate of 0.01, followed by reciprocal square root learning rate decay. T2T uses positional embeddings and subword tokenization, while T5 and Lingvo use relative attention [57] and SentencePieces [58].

For LM1B, we use the T2T default settings of max sequence length of 64 and batches of 4096 tokens; this is appropriate because LM1B has an average sequence length of roughly 32. For C4 and PG19, we use the T5 default of a max sequence length of 512. For one-shot pretraining, we use a max sequence length of 1024. In Section 4.2 we use batches of 65K tokens, in Section 4.3 we use batches of 1M tokens, and in Section 4.4 we use batches of 2M tokens.

A.9 Power Law Compute Savings Derivations

In Section 4.1 of the main text, we reproduce the results of Kaplan et al. [9] and show that, at optimal parameter sizing, the relationship between language model quality and training compute follows a power law: $l = ac^{-k}$, where l is validation loss, c is training compute, and a and k are empirical constants. 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 we compare. The lines are roughly parallel but shifted up and down. Thus, defining the shift between two architectures’ lines as $\log b^k$, we can derive the relationship of their training costs as:

$$\begin{aligned}
 -k \log c_0 + \log a_0 &= -k \log c_1 + \log a_0 + \log b^k \\
 -k \log c_0 &= -k \log c_1 + \log b^k \\
 c_0^{-k} &= b^k c_1^{-k} \\
 c_0 &= c_1 / b
 \end{aligned}$$

where b is a consistent reduction factor regardless of l . Compute savings, s , for using a superior architecture can now be calculated as:

$$\begin{aligned}
s &= c_1 - c_0 \\
s &= c_1 - c_1/b = c_1(1 - 1/b) \\
&\text{or} \\
c_1 &= \frac{s}{1 - 1/b}
\end{aligned}$$

Plugging this into the original power law relationship for c_1 we get:

$$\begin{aligned}
l &= a_1 \left(\frac{s}{1 - 1/b} \right)^{-k} \\
l &= a_1 (1 - 1/b)^k s^{-k}
\end{aligned}$$

Thus, the relationship between quality and compute savings yielded by an improved architecture also follows a power law with coefficient $a_1(1 - 1/b)^k$. This relationship is intuitive when recognizing that the compute reduction factor b is consistent for all values of 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.

A.10 Exact T5 Numbers for Medium Sized Experiments

		Baseline Compute @525K			Baseline Compute @1M		
Model	Params	Steps	PPLX	Speedup	Steps	PPLX	Speedup
	C4						
Vanilla Transformer	110M	525K	20.61	-	1M	19.82	-
Transformer+GELU	110M	524K	20.34	1.20	998K	19.58	1.26
Transformer++	110M	524K	20.03	1.52	998K	19.28	1.64
Evolved Transformer	110M	351K	20.79	0.89	668K	19.84	0.98
Primer	110M	483K	19.82	1.68	920K	19.07	1.91
Primer-EZ	110M	471K	19.83	1.71	896K	19.07	1.90
Switch Transformer	550M	525K	17.16	-	1M	16.32	-
Switch Primer	550M	474K	16.56	1.45	900K	15.82	1.56
Synthesizer	145M	525K	20.35	-	1M	19.57	-
+ Squared ReLU	145M	523K	19.55	1.74	996K	18.83	1.96
	PG19						
Vanilla Transformer	110M	525K	16.39	-	1M	15.83	-
Transformer+GELU	110M	524K	16.35	1.01	998K	15.84	0.95
Transformer++	110M	524K	16.15	1.18	998K	15.64	1.20
Primer	110M	483K	15.96	1.68	920K	15.31	1.81
Primer-EZ	110M	471K	15.84	1.74	896K	15.37	1.98

Table 5: Language modeling comparison on larger datasets, transferring Primer to the T5 codebase. In this transferred regime, Primer improves upon all baselines. Furthermore, Primer-EZ not only reaches parity with Primer, but in some cases, surpasses it. Switch Transformer and Synthesizer also benefit from the Primer-EZ modifications. Compute budget comparison points are chosen according to how long it takes vanilla baselines to reach 525K and 1M training steps. Perplexities are given with respect to SentencePieces. This table has the precise numbers for Figure 9.

A.11 Performance on Individual One-Shot Tasks

Task	Metric	Transf. 1/3	Transf. Full	Primer 1/3	Primer Full	GPT-3 XL
Pretraining	pplx	15.3	14.3	14.3	13.5	-
<i>Question Answering Tasks</i>						
TriviaQA	acc	22.5 ± 0.4	26.8 ± 0.5	27.5 ± 0.4	32.2 ± 0.5	26.5
WebQs	acc	9.1 ± 0.5	9.6 ± 0.4	9.8 ± 0.8	10.4 ± 0.3	9.2
NQs	acc	5.8 ± 0.2	6.7 ± 0.2	7.8 ± 0.5	9.1 ± 0.3	5.4
SQuADv2	f1	54.2 ± 2.4	65.4 ± 2.9	64.2 ± 3.7	67.8 ± 1.2	54
CoQa	f1	52.5 ± 1.1	57.7 ± 1.2	59.1 ± 0.9	61.2 ± 0.7	66.1
DROP	f1	21.5 ± 0.4	23.4 ± 0.2	24.8 ± 0.5	26.5 ± 0.2	23
Quac	f1	30.1 ± 0.5	30.9 ± 0.7	28.9 ± 0.9	30.2 ± 0.7	32.3
LAMBADA	acc	51.5 ± 0.9	55.2 ± 1.3	54.5 ± 1.1	56.8 ± 0.9	58.3
QA Average	avg	30.9	34.5	34.6	36.8	34.3
<i>Multi-Choice Schema Tasks</i>						
HellaSwag	acc	55.7 ± 0.3	59.5 ± 0.2	60.2 ± 0.3	63.3 ± 0.2	53.5
StoryCloze	acc	75.2 ± 0.3	75.9 ± 0.4	76.9 ± 0.2	77.5 ± 0.3	74.2
Winogrande	acc	55.4 ± 0.3	58.4 ± 0.4	58.8 ± 0.3	60.4 ± 0.2	59.1
PIQA	acc	72.6 ± 0.5	72.6 ± 0.3	73.7 ± 0.5	75.0 ± 0.4	74.4
ARC (Challenge)	acc	32.7 ± 0.4	34.4 ± 0.3	35.6 ± 0.9	37.4 ± 0.4	36.4
ARC (Easy)	acc	64.5 ± 0.5	64.9 ± 0.5	65.6 ± 0.6	67.5 ± 0.5	55.9
OpenBookQA	acc	45.3 ± 0.9	46.8 ± 0.8	47.9 ± 0.4	49.3 ± 0.5	46.4
ANLI R1	acc	33.9 ± 1.2	35.5 ± 0.2	35.5 ± 0.4	34.8 ± 0.3	34.6
ANLI R2	acc	33.5 ± 0.7	33.4 ± 0.5	34.5 ± 0.6	33.5 ± 0.4	32.7
ANLI R3	acc	34.5 ± 0.7	35.2 ± 0.1	33.0 ± 0.3	33.8 ± 0.5	33.9
ReCoRD	acc	84.8 ± 0.1	86.3 ± 0.2	85.8 ± 0.3	86.7 ± 0.0	83
WSC	acc	67.4 ± 0.8	66.8 ± 1.2	69.3 ± 1.3	68.9 ± 1.2	62.5
BoolQ	acc	58.9 ± 1.1	63.6 ± 2.1	60.7 ± 0.8	64.7 ± 2.0	63.7
CB	acc	56.3 ± 2.5	53.0 ± 2.7	55.4 ± 3.3	56.6 ± 9.6	48.2
RTE	acc	48.4 ± 1.2	53.6 ± 2.5	54.3 ± 1.5	52.9 ± 2.8	49.5
COPA	acc	80.2 ± 3.2	87.2 ± 1.2	84.8 ± 1.5	87.5 ± 1.1	74
WiC	acc	51.6 ± 0.2	51.0 ± 0.5	51.7 ± 0.1	51.8 ± 0.1	49.2
RACE-h	acc	39.4 ± 0.4	40.8 ± 0.4	40.4 ± 0.4	43.7 ± 0.3	42
RACE-m	acc	50.0 ± 1.0	52.6 ± 0.4	51.8 ± 0.8	54.0 ± 0.4	55.2
Multi-Choice Average	avg	53.1	54.7	55	56.2	54.1

Table 6: Comparison between Transformer+GELU and Primer at 1.9B parameters on downstream one-shot tasks at 1/3 and full pretraining compute budgets. One-shot sample means and standard deviations are computed using the evaluated performance of 5 weight checkpoints. **Bold numbers** denote improved one-shot performance and **shaded numbers** denote worse one-shot performance compared to Transformer with full compute that is statistically significant under an independent t-test with p-value threshold 0.05. Primer achieves the same performance as Transformer when given 1/3 the training compute and stronger performance on a majority of tasks when given the same training compute. GPT-3 XL [7] scores are provided as a grounding reference point; they should not be closely compared to our results as the models have different pretraining configurations.

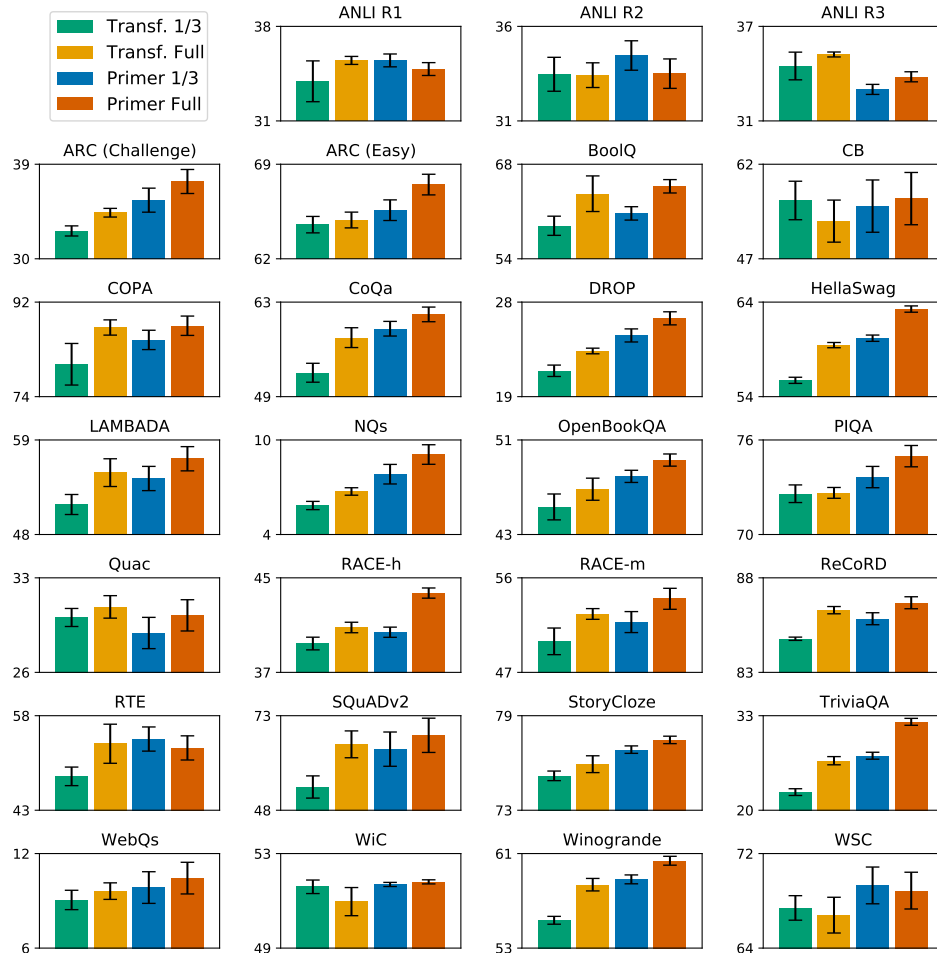


Figure 27: Comparison between Transformer+GELU and Primer at 1.9B parameters on downstream one-shot tasks at 1/3 and full pretraining compute budgets. 95% confidence intervals are provided according to an independent t-test, using a sample of 5 pretraining weight checkpoints. Primer achieves roughly the same performance as Transformer when given 1/3 the pretraining compute and stronger performance on a majority of tasks when given the same pretraining compute. Exact numbers are presented in Table 6.

A.12 Masked Language Modeling

Encoder-decoder style masked language modeling (MLM) is not the focus of this work. However, because it was the focus of the original T5 project, we include MLM comparisons here for completeness (Table 7). Specifically, we use the exact comparison configuration used by Narang et al.[49], who benchmarked several Transformer variants; the one difference is that we only run model training one time, since this regime is not the focus of our study. For “Primer-EZ Decoder” we use a Transformer++ encoder and a Primer-EZ decoder. Our treatments demonstrate that the Primer-EZ modifications have the capacity to improve encoder-decoder MLM models, but perhaps to a lesser degree, when compared to Transformer++. We believe this indicates that decoder-only LM and encoder-decoder MLM benefit from different modeling decisions – something that could be studied in future works. We also believe that running our search on encoder-decoder MLM directly could yield modifications that are more beneficial for this task.

Model	Params	Pretraining Log PPLX	SGLUE	XSum	WebQ
Vanilla Transformer*	223M	1.838	70.97	17.78	23.02
Transformer+GeLU*	223M	1.838	73.67	17.86	25.13
Transformer++	224M	1.792	75.65	17.90	25.92
Primer-EZ Decoder	224M	1.787	76.69	17.87	24.87

Table 7: **Masked language modeling** comparison on C4 in T5 with encoder-decoder style models. These results are run in the exact same configuration as Narang et al. [49], although we only run our models once, as MLM is not the focus of our work. * indicates rows that are taken from that study.

A.13 Carbon Emission Estimates

Following the recommendations of Patterson et al. [59], we release the carbon emission estimates for our largest experiments.

To estimate the carbon emissions^{4,5} for our **architecture search**, we build off of the measurements taken by Patterson et al. Their emissions estimate for architecture search is 3.2 MTCO₂e for 1360 days of TPUv2 usage [59]. Here, we use **1145.8 days of TPUv2** compute for our search. Additionally, the PUE for our data center⁶ at the time of our search was 1.08 instead of 1.10, and its net carbon intensity average was 0.336 MTCO₂e/MWh instead of 0.431 MTCO₂e/MWh.^{7,8} Thus, the proportional emissions estimate for our architecture search experiments is 3.2 MTCO₂e $\times \frac{1145.8}{1360} \times \frac{1.08}{1.10} \times \frac{336}{431} = 2.06$ MTCO₂e. For comparison, **a round trip plane ticket from San Francisco to New York for a single passenger is ~ 1.2 MTCO₂e** [59] and so our search costs roughly 1.72 such plane tickets.

We follow the same process of building off of the Patterson et al. measurements to estimate emissions for our large scale T5 experiments. The Patterson et al. emissions estimate for 11B parameter T5 is 46.7 tCO₂e for 10,249 days of TPUv3 usage. Our T5 models are smaller, and so only require 687.5 TPUv3 days to train on average. We run 3 trainings (Primer, original T5 and T5++) to show Primer’s improvements over baselines, yielding a total of 2062.5 TPUv3 days. When we ran our experiments, the data center⁹ PUE was 1.10 instead of 1.12 and its net carbon intensity average was 0.540 MTCO₂e/MWh instead of 0.545 MTCO₂e/MWh. Thus, the proportional total estimate for these T5 model trainings is 46.7 MTCO₂e $\times \frac{2062.5}{10,249} \times \frac{1.10}{1.12} \times \frac{540}{545} = 8.54$ MTCO₂e.

To estimate the emissions of our one-shot pretrainings in Lingvo, we measure system average power in the same manner as Patterson et al. [59]. Including memory, network interface, fans, and host CPU, the average power per TPUv4 chip is 343W. We use the same equation as Patterson et al. to calculate

⁴Our CO₂e accounting methodology for data center net carbon intensity does not currently fit the Greenhouse Gas (GHG) protocol for emissions reporting (Scope 2 and 3 for electricity). This deviation is due to a change in methodology where Google uses hourly life cycle emission factors, while the GHG Protocol generally relies on annual operating emission factor data. Google chooses to share these modified metrics as part of our 24/7 carbon-free energy (CFE) program, focused on our goal of achieving 100% 24/7 local CFE by 2030. Google’s target for 2030 goes beyond the traditional Scope 2 rules to restrict both the location and the accounting period. This means that, instead of anywhere in a continent, the CFE purchase should be on the same geographically local grid; and instead of the accounting period being one year, the accounting should be within the same hour.

⁵While electricity consumption is relatively straightforward, strategies to reduce greenhouse gas emissions are not. For details on the distinction between conventional carbon offsets, Google’s goal for 2030 of 24/7 CFE for its global data centers and campuses, and what it is doing now to set the groundwork for 2030, please see Appendix B of Patterson et al. [59].

⁶Each data center is located within a Regional Grid, which is the geographic basis for Google’s 24/7 CFE goals. For our data center in Georgia, the Regional Grid is the Southern Company balancing authority.

⁷The net carbon intensity at a particular data center is based on accounting for hourly emission reductions via real time, local carbon-free energy purchases. This is calculated using the 24/7 carbon-free energy methodology, which can be reviewed in greater depth in “24/7 Carbon-Free Energy: Methodologies and Metrics” [60].

⁸The carbon intensity values utilized in this paper are at the annual 2020 grid level for each data center in which the models were run.

⁹For our data center in Taipei, for purposes of Google’s 24/7 CFE accounting, the Regional Grid is Taiwan.

CO₂e for our 2 large scale pretrainings: $2 * 343W * 71,800h * 1.08(PUE) * 0.055 \text{ MTCO}_2\text{e/MWh} = 29.26 \text{ MTCO}_2\text{e}$.¹⁰

The emission cost for our large scale T5 and one-shot comparisons are higher than the cost of the architecture search itself. We invest in these large scale comparisons to demonstrate the potential savings of our efficient modifications. For instance, the savings for using Primer over Transformer described in Section 4.4 of the main text equates to 9.75 MTCO₂e, which alone is $\sim 4.7X$ the cost of the architecture search. Note, differences in hardware setups affect these savings. For example, the one-shot models were trained in Oklahoma, which has favorable MTCO₂e/MWh when compared to Georgia, where the Primer search was conducted. Viewing compute in terms of FLOPs, to remove these hardware-specific factors, Primer’s savings in the one-shot experiments are 9.24X the cost of the search itself, as described in Section 4.4 of the main text. Thus, the architecture search yields returns on investment, even at our relatively small comparison sizes, which are roughly 100X smaller than the full scale GPT-3 [7].

A.14 Comparison to Evolved Transformer

Model	<i>LM1B</i>		<i>C4</i>	
	Params	PPLX @ 1.5M Steps	Params	PPLX @ 1M Steps
Vanilla Transformer	35M	23.45	110M	19.82
Transformer+GELU	35M	23.68	110M	19.58
Transformer++	35M	23.35	110M	19.29
Evolved Transformer	38M	23.11	110M	19.37
Primer	36M	22.97	110M	18.99
Primer-EZ	36M	22.89	110M	18.93

Table 8: Auto-regressive language modeling comparison between Primer and various baselines, including the Evolved Transformer, controlling for training steps in T5. These are the same experiments featured in Tables 4 and 5, but with the data presented to compare sample efficiency instead of training compute efficiency.

This work builds off of the Evolved Transformer [13], which also sought to discover improved sequence models using architecture search. Compute efficiency comparisons to the Evolved Transformer architecture are provided in T5 on LM1B in Table 4 and on C4 in Table 5. Sample efficiency comparisons to the Evolved Transformer architecture are offered in Table 8 on those same experiments. In this section we discuss these comparisons and how they highlight the improvements of our Primer search over the Evolved Transformer search.

Firstly, our Primer search aims to improve training compute efficiency, which yields more practical results than the sample efficiency objective of So et al. [13], who controlled for number of train steps when evaluating models. Evolved Transformer is effective in this controlled-train-step regime when comparing to other baselines, as shown in Table 8. When controlling for number of training steps in this way, Evolved Transformer is roughly on par with Transformer++ on C4 and is better than Transformer++ on LM1B. However, Evolved Transformer is substantially slower than all other models (see Tables 4 and 5) because it is deeper; we follow the same scaling policy as So et al. of adding additional layers to control for parameters, given that an Evolved Transformer layer has significantly less parameters than a standard Transformer layer. Evolved Transformer’s slowness counteracts its sample efficiency and for this reason its speedup factor is diminished on LM1B and less than 1.0 (indicating a slowdown over vanilla Transformer) on C4 (see Tables 4 and 5). This limits Evolved Transformer’s practicality. In contrast, Primer is designed to specifically address this shortcoming and thus delivers the practical result of substantial compute savings.

The open-ended nature of the Primer search also allows for effective modifications that were not available to the Evolved Transformer search. In fact, *none* of the Primer modifications (see Section 3) can be represented in the Evolved Transformer search space, aside from resizing hidden dimension

¹⁰For our data center in Oklahoma, for purposes of Google’s 24/7 CFE accounting, the Regional Grid is the Southwest Power Pool (SPP) Independent System Operator.

sizes. This is because the Evolved Transformer search space followed a rigid ordering of components and used a vocabulary of unalterable high level building blocks. For example, normalization always preceded weighted transformations and, although there were different weighted transformations to choose from such as self-attention and GLU, those transformations could not be modified by the search. In contrast, the Primer search space allows for the modification of all initialized modules – such as weighted transformations, activation functions and normalization functions – as well as allows for macro-level reordering, such as moving normalization after weighted transformations. We believe that this difference in openness is what allowed Primer to develop definitively superior modifications, as demonstrated not only by improved compute efficiency, but also by improved sample efficiency (Table 8), which is what Evolved Transformer was meant to optimize.