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]

[1] https://github.com/google-research/google-research/tree/master/primer

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

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

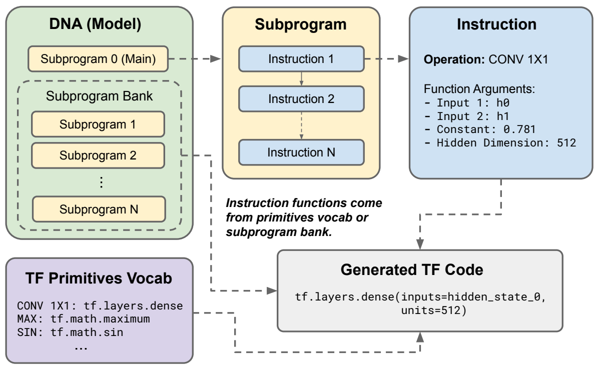


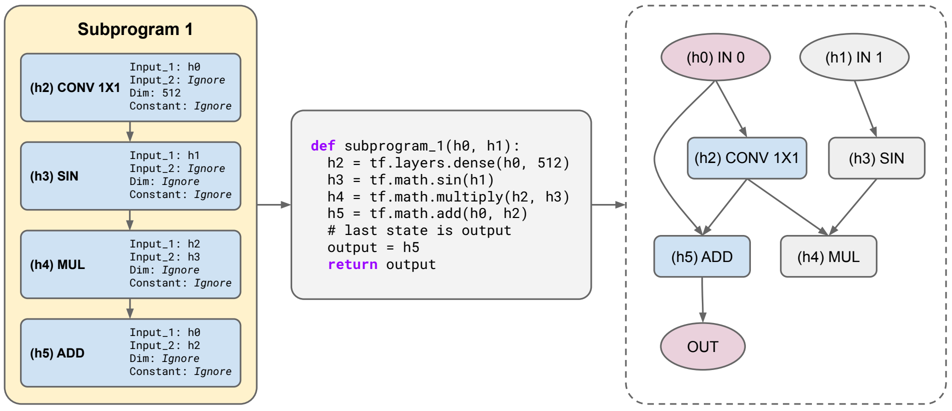
Figure 1: Overview of DNAs that define a decoder model program (i.e., an auto-regressive language model). Each DNA has a collection of subprograms, where SUBPROGRAM 0 is the MAIN() function entry point. Each subprogram is comprised of instructions, which are converted to lines of TensorFlow code. Instruction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



```
Input_1: h0
Input_2: Ignore
Dim: 512
Constant: Ignore

Input_1: h1
Input_2: Ignore
Dim: Ignore
Constant: Ignore

Input_1: h2
Input_2: h3
Dim: Ignore
Constant: Ignore

Input_1: h0
Input_2: h2
Dim: Ignore
Constant: Ignore
```

```
def subprogram_1(h0, h1):
  h2 = tf.layers.dense(h0, 512)
  h3 = tf.math.sin(h1)
  h4 = tf.math.multiply(h2, h3)
  h5 = tf.math.add(h0, h2)
  # last state is output
  output = h5
  return output
```

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 50^{th} 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 $\sim$25K individuals and retrain the top 100 individuals on the search task to select the best one.

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).

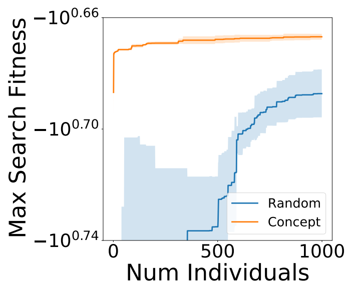
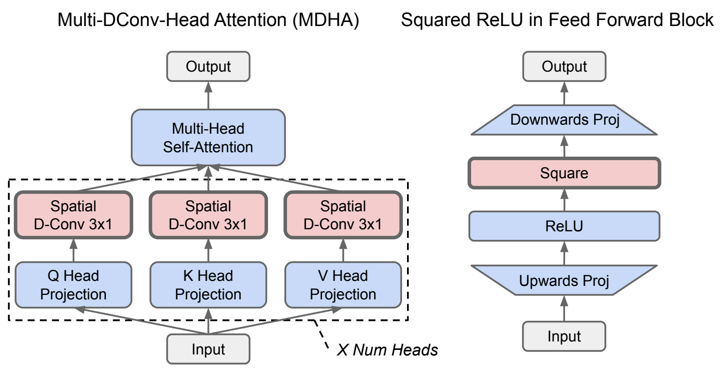


Figure 3: Small scale searches (10 hours on 10 TPUv2 chips) comparing conceptual initialization to random initialization. Our search space is open-ended enough that it is infeasible to search without strong initialization.

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.



MDHA Projection Pseudo-code

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

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

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

Squared ReLU: The most effective modification is the improvement from a ReLU activation to a squared ReLU activation in the Transformer's feed forward block. Rectified polynomials of varying degrees have been studied in the context of neural network activation functions [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).

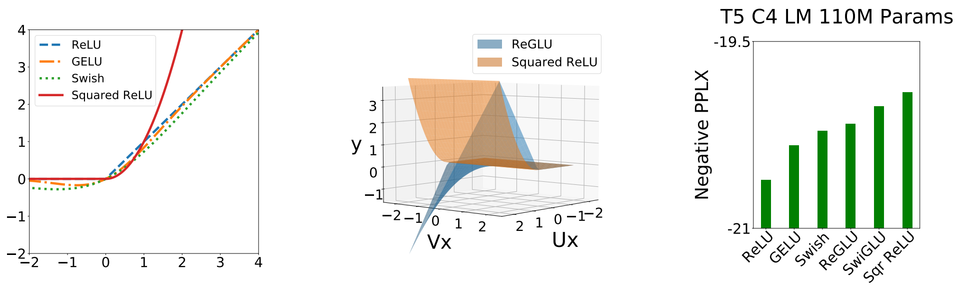


Figure 5: Left: Squared ReLU has starkly different asymptotics compared to other common activation functions. Center: Squared ReLU has significant overlap with GLU variants [39] that use activations with ReLU-like asymptotics, such as ReGLU and SwiGLU. Our experiments indicate that squared ReLU is better 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.

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 (~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 (SwiGLU) [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$35M 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.

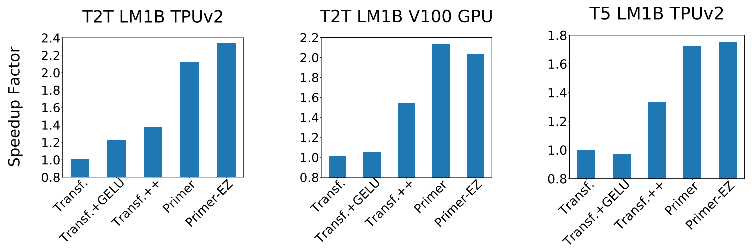
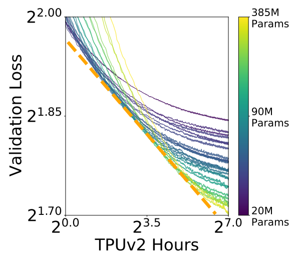


Figure 6: Comparison on the LM1B search task using 35M parameter models. "Speedup Factor" refers to the fraction of compute each model needs to reach quality parity with the vanilla Transformer trained for 24 hours. Primer and Primer-EZ both achieve over 1.7X speedup in all cases. Note that results transfer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



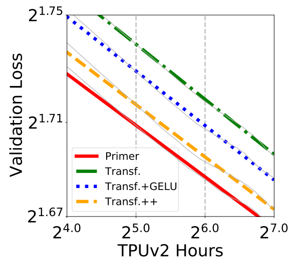


Figure 7: Left: When sweeping over optimal model sizes, the relationship between Transformer language model quality and training compute roughly obeys a power law [9]. Right: Comparison of these power law lines for varying Transformer modifications, fit with smoothed MSE loss. That the model lines are 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.

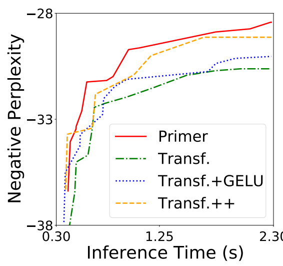


Figure 8: Pareto optimal inference comparison on LM1B. Primer demonstrates improved inference at a majority of target qualities. We observe these models have a 0.97 correlation between their train step and inference times.

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 ~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.

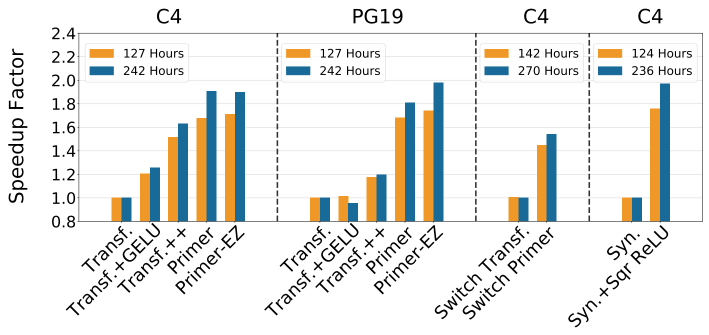


Figure 9: Comparison transferring Primer to larger datasets (C4 and PG19) and different model families (Switch Transformer and Synthesizer) in a different codebase (T5) with an order of magnitude more compute than the search task. Compared to the vanilla baseline, Primer and Primer-EZ are at least 1.8X 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 approximations 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$1M tokens, 64 TPUv3 chips and 537M parameters ($d_{model} = 1024$, $d_{ff} = 8192$, $L = 24$). Primer is 4.2X more compute efficient than the original T5 model and 2X more efficient than our strengthened Transformer++ baseline (Table 1).

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.

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.

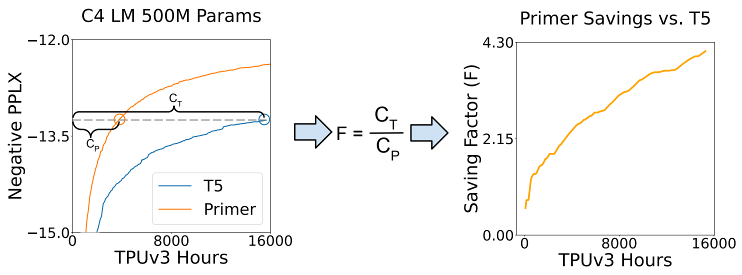


Figure 10: Compute savings of Primer vs. the original T5 architecture on C4 LM over time. The more compute invested in training, the higher the savings due to the asymptotic nature of both perplexity curves. Primer achieves the same performance as the original T5 architecture with 4.2X less compute.

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. 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 $\sim$140 hours ($\sim$71.8K total accelerator hours or $\sim$1M train steps). We once again use the T5 training hyperparemeters without any additional tuning.

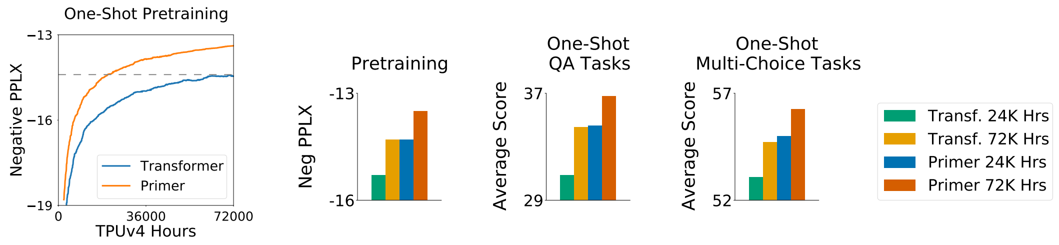


Figure 11: Comparison between Transformer+GELU and Primer at 1.9B parameters and varying training compute budgets on downstream one-shot tasks, similar to GPT-3. Primer achieves slightly better performance than Transformer when given 3X less pretraining compute and substantially better performance when 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 exact task scores.

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 Transfomer+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.

5 Conclusion

Limitations: There are limitations to this study. First, although our comparisons are at substantial sizes, they are still orders of magnitude smaller than state-of-the-art models such as the full-scale GPT-3 [7]. Second, we focus primarily on decoder-only models, while encoder-based sequence models are still widely used [1, 2, 3, 4, 6, 52]. 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 better encoders is a topic of our future research.

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, Lluis-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 Universitat Munchen, 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, Phillipp 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 Févry, Michael Matena, Karishma Malkan, Noah Fiedel, Noam Shazeer, Zhenzhong Lan, Yanqi Zhou, Wei Li, Nan Ding, Jake Marcus, Adam Roberts, and Colin Raffel. Do transformer modifications transfer across implementations and applications? *arXiv preprint arXiv:2102.11972*, 2021.

[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, Sébastien 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. Álvarez, 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, Lluís-Miquel Munguía, 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.