LEARNING TO GROW PRETRAINED MODELS FOR EFFICIENT TRANSFORMER TRAINING

Anonymous authors
Paper under double-blind review

ABSTRACT

Scaling transformers has led to significant breakthroughs in many domains, leading to a paradigm in which larger versions of existing models are trained and released on a periodic basis. Curiously, new instances of such models are typically trained completely from scratch, despite the fact that they are often just scaled-up versions of their smaller counterparts. How can we use the implicit knowledge in the parameters of smaller, extant models to enable faster training of newer, larger models? This paper describes an approach for accelerating transformer training by *learning to grow* pretrained transformers, where we learn to linearly map the parameters of the smaller model to initialize the larger model. For tractable learning, we factorize the linear transformation as a composition of (linear) width- and depth-growth operators, and further employ a Kronecker factorization of these growth operators to encode architectural knowledge. Experiments across both language and vision transformers demonstrate that our LEarning to GrOw (LEGO) approach can save around 50% computational cost of training from scratch, while also consistently outperforming strong baselines that also reuse smaller pretrained models to initialize larger models. Our code will be made publicly available.

1 INTRODUCTION

The transformer architecture (Vaswani et al., 2017) has emerged as a general purpose architecture for modeling many structured domains (Devlin et al., 2019; Brown et al., 2020; Rives et al., 2021; Dosovitskiy et al., 2021; Touvron et al., 2021a). Perhaps more so than other architectures, the transformer empirically seems to have inductive biases that make it especially amenable to scaling (Rosenfeld et al., 2019; Kaplan et al., 2020), which has led to a paradigm in which larger versions of smaller, existing models are trained and released on a periodic basis (e.g., the GPT lineage of models (Radford et al., 2018; 2019; Brown et al., 2020)). New instances of such models are typically trained completely from scratch, despite the fact that they are often scaled-up versions of their smaller counterparts. Given the compute required to train even the smaller models, we argue that training each model from scratch is wasteful, and that prior knowledge implicit in the parameters of smaller, extant models should be leveraged to enable faster training of larger models.

One approach to this problem is through the lens of *model growth*, wherein a smaller model's pretrained parameters are used to initialize a subset of the larger model's parameters. While earlier works generally froze the parameters initialized from the pretrained model and only trained the new (randomly initialized) parameters (Fahlman & Lebiere, 1989; Fahlman, 1990; Gutstein et al., 2008), subsequent work has shown that copying a subset of the pretrained parameters to initialize the new parameters and then finetuning the entire network significantly accelerates training and sometimes even leads to better performance (Chen et al., 2015). When applied to modern transformers, these mechanisms roughly translate to a depth-expansion operator in which pretrained models are stacked (or combined with identity layers) to initialize deeper transformers (Gong et al., 2019; Yang et al., 2020), and a width-expansion operator in which the smaller model's matrices are copied to initialize the larger model's matrices (e.g., in block-diagonal fashion) (Chen et al., 2021b; Gu et al., 2020).

Noting the empirical effectiveness of such recipes, we observe that existing mechanisms generally do not have a learning component (e.g., randomly copying over neurons for width-expansion or stacking consecutive layers for depth-expansion). This paper instead proposes an efficient, data-driven approach for *learning to grow* transformers. In particular, our LEarning to GrOw (LEGO) approach frames the problem of initializing the larger model's parameters as learning a linear mapping from

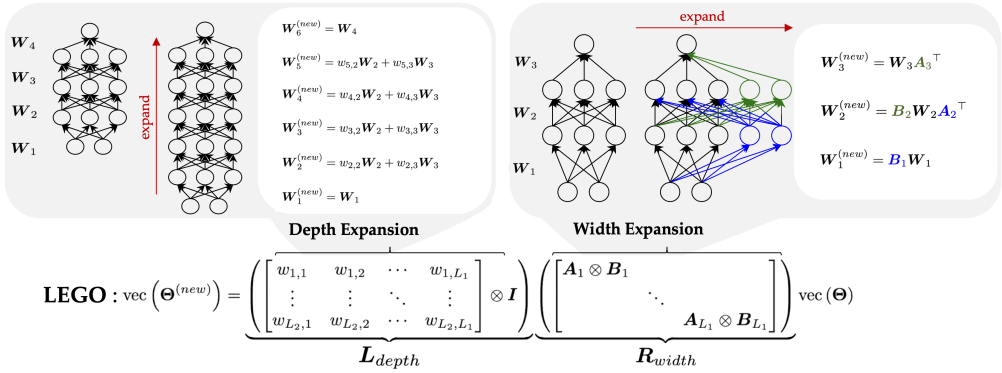


Figure 1: Our learning to grow (LEGO) framework accelerates training by using the weights of a smaller model $\boldsymbol{\Theta}$ to initialize the weights of the larger model $\boldsymbol{\Theta}^{(new)}$. The LEGO operator is parameterized as a sparse linear map $\boldsymbol{M}$ that can be decomposed into width- and depth-expansion operators. The width-operator $\boldsymbol{R}_{width}$ and depth-operator $\boldsymbol{L}_{depth}$ are structured matrices obtained from Kronecker products of smaller matrices which encode architectural knowledge by grouping parameters into layers and neurons. While we show the expansion operators for simple multi-layer perceptrons for illustrative purposes, in practice we apply LEGO to enable faster training of transformer networks. In our approach, we learn the LEGO matrix $\boldsymbol{M}$ with a 100 steps of SGD, use this to initialize the larger model, and then continue training as usual. Best viewed in color.

the smaller model's parameters, i.e., $\boldsymbol{\theta}^{(large)} = \boldsymbol{M}\boldsymbol{\theta}^{(small)}$ where $\boldsymbol{\theta}^{(small)}$ and $\boldsymbol{\theta}^{(large)}$ are the vectorized parameters of the small/large models. Of course, this mapping is completely intractable to learn without any restrictions on $\boldsymbol{M}$. We thus factorize the linear mapping to be a composition of sparse width- and depth-expansion operators, $\boldsymbol{M} = \boldsymbol{L}_{depth}\boldsymbol{R}_{width}$, where both width and depth matrices are further factorized to be a Kronecker product of smaller matrices that express architectural knowledge (e.g., through grouping parameters by layers and neurons). We show that our growth operators can represent existing approaches such as layer-stacking and neuron-copying as special cases. We find that with a small amount of learning on $\boldsymbol{M}$ (e.g., 100 gradient steps) to initialize the larger model, we can significantly accelerate training and outperform existing approaches for model growth with both vision and language transformers. Figure 1 illustrates our approach.

Targeting model families where there already exists a rich set of smaller pretrained models, extensive experiments on BERT (Devlin et al., 2019), RoBERTa (Liu et al., 2019), GPT2 (Radford et al., 2019), and ViT (Dosovitskiy et al., 2021; Touvron et al., 2021a;b) show that LEGO can consistently improve transformer training efficiency over the traditional way of training from scratch across domains and model sizes. For instance, LEGO saves 44.7% and 22.5% FLOPs for training BERT-Base and GPT2-Large from scratch by reusing pretrained smaller models that are half as big. Similarly, for vision transformers, when using DeiT-S (Touvron et al., 2021a) for initialization, LEGO yields 55% savings in FLOPs with no performance drop on ImageNet (Deng et al., 2009). These FLOPs savings directly translate to similar savings in wall clock. We also find that models trained using our proposed approach achieves very similar performance compared to the trained-from-scratch baselines when transferred to other downstream tasks (e.g., GLUE (Wang et al., 2018), SQuAD (Rajpurkar et al., 2016) for language models, and CIFAR (Krizhevsky et al., 2009), Flowers-102 (Nilsback & Zisserman, 2008), etc., for vision models.).

2 Related Work

Efficient Training. Efficient training of transformers has been studied from multiple perspectives. Some methods that are orthogonal to our work include mixed precision training (Shoeybi et al., 2019), large batch optimization (You et al., 2019), distributed training (Huang et al., 2019), and dropping layers (Zhang & He, 2020) or tokens (Hou et al., 2022). Knowledge inheritance (Qin et al., 2021) explores knowledge distillation during pre-training to efficiently learn larger transformers. Progressive training, which first trains a small transformer with few layers and then gradually expands by stacking layers, has also been applied to transformer training (Gong et al., 2019; Yang et al., 2020; Li et al., 2022; Shen et al., 2022). Net2Net Chen et al. (2015) uses function preserving transformations to grow width by copying neurons and depth by using identity layers. Recently, bert2BERT (Chen et al., 2021b) extends Net2Net to transformers. In contrast these approaches, our approach learns to (linearly) transform the parameters of a smaller model to initialize a larger model. While there is a line of work on learning to grow neural networks in a data-driven way, these methods are in general difficult to apply to modern-scale transformers since they (for example) involve

growing a single neuron at a time or employ expensive optimization/search procedures (Wei et al., 2016; Cai et al., 2018; Wu et al., 2019; 2021; Evci et al., 2022).

Network Initialization. Our work is also related to work on neural network initialization. Existing work in this line include controlling the norm of the parameters (Mishkin & Matas, 2015; Kilcher et al., 2018; Dai et al., 2019; Wu et al., 2019; Glorot & Bengio, 2010) or replacing the normalization layers (Brock et al., 2021; Zhang et al., 2019; Huang et al., 2020). MetaInit (Dauphin & Schoenholz, 2019) proposes an automatic network initialization method that optimizes the norms of weight tensors to minimize the gradient quotient on minibatches of random Gaussian samples. GradInit (Zhu et al., 2021) learns to initialize networks by adjusting norm of each network layer so that a single step optimization with prescribed hyperparameters results in the smallest possible loss value. We instead focus on taking advantage of smaller pretrained transformers to better initialize larger transformers, which remains an understudied problem.

Structured Matrices. Finally, our work is also related to structured matrices which are typically used to replace dense weight matrices for reducing training and inference computation cost. Representative works use sparse and low rank matrices (Chiu et al., 2021; Han et al., 2015), Chebyshev (Tang et al., 2019), Toeplitz (Sindhwani et al., 2015), Kronecker-product matrices (Zhang et al., 2015), and butterfly matrices (Dao et al., 2019) and its variants (Chen et al., 2021a). A unified framework to learn a broad family of structured matrices for small-footprint deep learning is presented in Sindhwani et al. (2015). Dao et al. (2022) propose the family of Monarch matrices, which inherit the expressiveness of butterfly matrices and achieves reasonable accuracy-efficiency tradeoffs in many sparse training applications. While our approach is inspired by these works, we propose to grow pretrained models by learning structured sparse linear operators with Kronecker factorization, which to our best knowledge has not been explored in the literature.

3 Proposed Approach

Notation. We denote the parameters of a neural network with L layers and D dimensions as $\boldsymbol{\Theta}_{L,D} = [\boldsymbol{W}_1 \quad \cdots \quad \boldsymbol{W}_L]^\top \in \mathbb{R}^{LD \times D}$, where $\boldsymbol{W}_l \in \mathbb{R}^{D \times D}$ denotes the weights for the l-th layer.[1] With slight abuse of notation, we denote the vectorization of $\boldsymbol{\Theta}_{L,D}$ as $\mathrm{vec}(\boldsymbol{\Theta}_{L,D})^\top = \left[\mathrm{vec}(\boldsymbol{W}_1)^\top \quad \cdots \quad \mathrm{vec}(\boldsymbol{W}_L)^\top\right]$.[2] Our goal is to re-use the parameters $\boldsymbol{\Theta} = \boldsymbol{\Theta}_{L_1,D_1}$ from a pretrained smaller model to initialize a large model $\boldsymbol{\Theta}^{(new)} = \boldsymbol{\Theta}_{L_2,D_2}$ through a *model growth operator* $M : \mathbb{R}^{L_1 D_1 \times D_1} \to \mathbb{R}^{L_2 D_2 \times D_2}$ that maps the weights of the smaller network to the weights of the larger one: $\boldsymbol{\Theta}^{(new)} = M(\boldsymbol{\Theta})$, where $L_1 < L_2$ and $D_1 < D_2$. After model growth, we adopt $\boldsymbol{\Theta}^{(new)}$ as the initialization of the large model and train it following standard recipes.

3.1 Existing Growth Operators

Existing works have separately established model growth operators M for depth ($L_1 < L_2, D_1 = D_2$) and width ($L_1 = L_2, D_1 < D_2$). We summarize these methods below.

Depth Expansion. StackBERT (Gong et al., 2019) proposes to duplicate the smaller model to double the depth, based on the observation that upper layers share similar functionality with the lower layers. In contrast, interpolation-based depth expansion methods (Chang et al., 2017; Dong et al., 2020) interleave every layer to form a deeper model, which can be roughly interpreted as simulating a finer grained solution to the original dynamical system from a neural ODE perspective (Chen et al., 2018). Letting $L_2 = kL_1$, the two methods' growth operators can be formulated as:

$$\text{(StackBERT)}\ \boldsymbol{W}_l^{(new)} = \boldsymbol{W}_{l \bmod L_1}, \quad \text{(Interpolation)}\ \boldsymbol{W}_i^{(new)} = \boldsymbol{W}_{\lfloor l/k \rfloor}, \quad \forall l \in [L_2]. \tag{1}$$

Width Expansion. Net2Net (Chen et al., 2015) expands width of neural networks by randomly copying neurons while preserving output values via normalization. This can be seen as growing a matrix associated with a particular layer by duplicating the columns and rows of its weight matrix. Suppose a layer has weight matrix $\boldsymbol{W}_l \in \mathbb{R}^{D_1 \times D_1}$.[3] To expand it to a matrix $\boldsymbol{W}_l^{(new)} \in \mathbb{R}^{D_2 \times D_2}$

[1] For notational brevity we assume that each hidden layer has same number of dimensions D, but LEGO can be straightforwardly generalized to layers with different dimensions (e.g., FFN layers of transformers).

[2] We therefore have $\mathrm{vec}(\boldsymbol{\Theta}_{L,D})^\top \in \mathbb{R}^{LD^2}$. Our approach is also agnostic with regard to vectorization order.

[3] We define a single layer as $f_l(\boldsymbol{x}) = \boldsymbol{W}_l \boldsymbol{x} + \boldsymbol{b}_l$, where the row number of $\boldsymbol{W}_l$ corresponds to the output dimension, and the column number of $\boldsymbol{W}_l$ corresponds to the input dimension.

($D_2 > D_1$), Net2Net copies $\boldsymbol{W}_l$ to its upper-left corner of $\boldsymbol{W}_l^{(new)}$, fills the new columns via a random selection matrix $\boldsymbol{S}_l$, and finally duplicates and normalizes rows according to the selection matrix from the previous layer. Formally, the growth operator of Net2Net can be written as:

$$\text{(Net2Net) } \boldsymbol{W}_l^{(new)} = \begin{bmatrix} \boldsymbol{I} \\ \boldsymbol{S}_{l-1}^\top \end{bmatrix} \boldsymbol{D}_l^{-1} \boldsymbol{W}_l \begin{bmatrix} \boldsymbol{I} & \boldsymbol{S}_l \end{bmatrix}, \quad \boldsymbol{D}_l = \operatorname{diag}(\boldsymbol{S}_{l-1}\mathbf{1}) + \boldsymbol{I}, \quad \forall l \in [L_2] \tag{2}$$

where $\boldsymbol{S}_l \in \{0,1\}^{D_1 \times (D_2 - D_1)}$ is a random selection matrix. The diagonal of $\boldsymbol{D}_l$ is a D_1-dimensional histogram, whose i-th entry indicates number of times i-th column of $\boldsymbol{W}_l$ was copied.

3.2 Learning to Grow

While existing operators have been empirically successful in accelerating transformer-based models such as BERT (Gong et al., 2019; Chen et al., 2021b), we observe that generally do not have a learning component and perform the depth- and width-expansions separately. In this section we introduce a general framework for LEarning to GrOw (LEGO) pretrained models, which generalizes existing operators by combining the width- and depth-growth operators in a data-driven way.

We can formulate the problem of initializing the weights of the larger model $\boldsymbol{\Theta}^{(new)}$ from the smaller model $\boldsymbol{\Theta}$ through the following optimization problem,

$$\arg\min_{M} \mathbb{E}_{\boldsymbol{x} \sim \mathcal{D}} \mathcal{L}(\boldsymbol{x}; \boldsymbol{\Theta}^{(new)}), \quad \text{subject to } \boldsymbol{\Theta}^{(new)} = M(\boldsymbol{\Theta}), \tag{3}$$

where $\mathcal{D}$ is the data distribution. It is of course intractable to optimize over the entire operator space, and thus we further simplify the function M to be a linear transformation, which results in the following formulation,

$$\operatorname{vec}(\boldsymbol{\Theta}^{(new)}) = \operatorname{vec}(M(\boldsymbol{\Theta})) = \boldsymbol{M} \operatorname{vec}(\boldsymbol{\Theta}), \quad \boldsymbol{M} \in \mathbb{R}^{L_2 D_2^2 \times L_1 D_1^2}. \tag{4}$$

This simplified objective is still completely infeasible to apply to contemporary neural networks where $L_1 D_1$ can easily be in the hundreds of millions. We therefore propose an efficient parameterization of $\boldsymbol{M}$ for tractable learning.

3.2.1 Decomposition along Depth and Width

Our first step is to decompose the LEGO operator as $\boldsymbol{M} = \boldsymbol{L}_{depth} \boldsymbol{R}_{width}$, where $\boldsymbol{L}_{depth}$ and $\boldsymbol{R}_{width}$ expand the depth and width of model separately. More concretely, we decompose $\boldsymbol{M}$ as

$$\boldsymbol{M} = \underbrace{\begin{bmatrix} \operatorname{diag}(\boldsymbol{\ell}_{1,1}) & \cdots & \operatorname{diag}(\boldsymbol{\ell}_{1,L_1}) \\ \vdots & \ddots & \vdots \\ \operatorname{diag}(\boldsymbol{\ell}_{L_2,1}) & \cdots & \operatorname{diag}(\boldsymbol{\ell}_{L_2,L_1}) \end{bmatrix}}_{\boldsymbol{L}_{depth}} \underbrace{\begin{bmatrix} \boldsymbol{R}_1 & & \\ & \ddots & \\ & & \boldsymbol{R}_{L_1} \end{bmatrix}}_{\boldsymbol{R}_{width}}. \tag{5}$$

where $\boldsymbol{R}_l \in \mathbb{R}^{D_2^2 \times D_1^2}$ and $\boldsymbol{\ell}_{i,j} \in \mathbb{R}^{D_2^2}$. $\boldsymbol{L}_{depth}$ is an array of diagonal matrices and $\boldsymbol{R}_{width}$ is a block-diagonal matrix, i.e., both matrices are highly structured and sparse. When applying $\boldsymbol{R}_{width}$ to weights $\operatorname{vec}(\boldsymbol{\Theta})$, the parameters of each layer will be transformed independently via $\operatorname{vec}(\boldsymbol{W}_l^{(new)}) = \boldsymbol{R}_l \operatorname{vec}(\boldsymbol{W}_l)$ and lifted to a higher dimension. The l-th row block of $\boldsymbol{L}_{depth}$ corresponds to growth operator of l-th layer, which amounts to linearly combining all layers of the smaller model via $\operatorname{vec}(\boldsymbol{W}_l^{(new)})_k = \sum_{l'=1}^{L_1} (\boldsymbol{\ell}_{l,l'})_k \operatorname{vec}(\boldsymbol{W}_l)_k$. By this factorization, we can effectively reduce complexity of LEGO operator from $\mathcal{O}(D_1^2 L_1 D_2^2 L_2)$ to $\mathcal{O}(D_1^2 D_2^2 L_1)$ and encode architectural knowledge by grouping parameters by layers. We show in Sec. 3.4 that this representation still enjoys high representation power due to its connection with Monarch matrices (Dao et al., 2022).

3.2.2 Parameter Sharing via Kronecker Factorization

The above LEGO operator still requires $\mathcal{O}(D_1^2 D_2^2 L_1)$ parameters for $\boldsymbol{R}_{width}$ and $\mathcal{O}(L_1 L_2 D_2^2)$ for $\boldsymbol{L}_{depth}$. The width operator $\boldsymbol{R}_{width}$ in this case still remains prohibitively expensive given that D_1 (and D_2) can easily be in the hundreds or thousands. In this section, we propose a Kronecker factorization to further reduce the number of learnable parameters for each growth operator.

Depth. For depth, we treat an entire layer as a single group and construct a new layer by combining existing layers, effectively tying parameters for all neurons in same layer. Formally, each block in

$\boldsymbol{L}_{depth}$ is simplified to be $\mathrm{diag}(\boldsymbol{\ell}_{i,j}) = w_{i,j}\boldsymbol{I}$. Then the entire matrix can be written as a Kronecker factorization, $\boldsymbol{L}_{depth} = \boldsymbol{w} \otimes \boldsymbol{I}$, where $\boldsymbol{w} \in \mathbb{R}^{L_2 \times L_1}$ is a matrix whose entry $w_{i,j}$ indicates blending weights of j-th layer of the small model to form i-th layer of the large model. This strategy reduces the number of parameters in $\boldsymbol{L}_{depth}$ to $\mathcal{O}(L_1 L_2)$, and is shown on left hand side of Figure 1.

Width. For width, we decompose each diagonal block of width expansion operator $\boldsymbol{R}_{width}$ using the Kronecker factorization $\boldsymbol{R}_l = \boldsymbol{A}_l \otimes \boldsymbol{B}_l$, where $\boldsymbol{A}_l, \boldsymbol{B}_l \in \mathbb{R}^{D_2 \times D_1}$. Since $\mathrm{vec}(\boldsymbol{CAB}) = (\boldsymbol{B}^\top \otimes \boldsymbol{C})\,\mathrm{vec}(\boldsymbol{A})$ (Schacke, 2004), we then have,

$$\boldsymbol{R}_{width}\,\mathrm{vec}(\boldsymbol{\Theta}) = \begin{bmatrix} \boldsymbol{A}_1 \otimes \boldsymbol{B}_1 & & \\ & \ddots & \\ & & \boldsymbol{A}_{L_1} \otimes \boldsymbol{B}_{L_1} \end{bmatrix} \mathrm{vec}(\boldsymbol{\Theta}) \tag{6}$$

$$= \mathrm{vec}\left(\begin{bmatrix} \boldsymbol{B}_1 \boldsymbol{W}_1 \boldsymbol{A}_1^\top & \cdots & \boldsymbol{B}_{L_1} \boldsymbol{W}_{L_1} \boldsymbol{A}_{L_1}^\top \end{bmatrix}^\top \right). \tag{7}$$

Here we observe that $\boldsymbol{B}_l \boldsymbol{W}_l \boldsymbol{A}_l^\top$ performs in- and out-dimension expansion by $\boldsymbol{A}_l$ and $\boldsymbol{B}_l$, respectively. Each new column/row is a linear combination of columns/rows of small model's weight matrix. This factorization, which can be seen as grouping parameters by *neurons*, reduces number of parameters to $\mathcal{O}(L_1 D_1 D_2)$. Figure 1 (right) illustrates LEGO's width-expansion operator.

Altogether, we obtain the final parameterization of LEGO operator $\boldsymbol{M}$:

$$\boldsymbol{M} = \underbrace{\left(\begin{bmatrix} w_{1,1} & w_{1,2} & \cdots & w_{1,L_1} \\ \vdots & \vdots & \ddots & \vdots \\ w_{L_2,1} & w_{L_2,2} & \cdots & w_{L_2,L_1} \end{bmatrix} \otimes \boldsymbol{I} \right)}_{\text{Depth expansion}} \underbrace{\left(\begin{bmatrix} \boldsymbol{A}_1 \otimes \boldsymbol{B}_1 & & \\ & \ddots & \\ & & \boldsymbol{A}_{L_1} \otimes \boldsymbol{B}_{L_1} \end{bmatrix} \right)}_{\text{Width expansion}} \tag{8}$$

We can exploit the factorization to implement LEGO operator (Eq. 8) efficiently.

Overall Training. Concretely, LEGO expands a model in three steps: (1) for each layer, inserting new rows by linearly combining existing rows by $\boldsymbol{B}_l$, (2) for each layer, inserting new columns by linearly combining existing columns by $\boldsymbol{A}_l$, and then finally (3) reconstructing each layer by linearly combining the weight matrices with $\boldsymbol{w}$ along the depth. We then run a few steps (e.g., 100 iterations) of SGD to optimize $\boldsymbol{M}$, which has negligible compute cost relative to regular training. After obtaining $\boldsymbol{M}$, we initialize large model with $\boldsymbol{M}\,\mathrm{vec}(\boldsymbol{\Theta})$, and train parameters $\boldsymbol{\Theta}^{(new)}$ through SGD as usual. Algorithm 1 summarizes a forward pass of LEGO with transformer. Finally, as shown in Appendix A we note that StackBERT (Eq. 1), Interpolation (Eq. 1), and Net2Net (Eq. 2) are all special cases of LEGO (Eq. 8) with a particular setting of $\boldsymbol{L}_{depth}$ and $\boldsymbol{R}_{width}$.

3.3 Growing Transformers with LEGO

While LEGO can be applied to any multi-layer neural network architecture, in this paper we focus on using LEGO to grow transformers which have been shown to be particularly amenable to scaling. Below we briefly describe how LEGO is applied to the main transformer embedding/attention layers and defer further details (e.g., growing bias vectors, layer norm parameters) to the Appendix B.

Embedding Layer. Embedding layer can be regarded as a linear layer whose inputs are one-hot embeddings. We draw a learnable matrix $\boldsymbol{B}^{(emb)}$ to extend its output dimension. This embedding layer is also used as the final output layer for our transformer language modeling experiments.

Attention and Feedforward Layers. An attention layer consists of multi-head attention weights ($\boldsymbol{W}^Q, \boldsymbol{W}^K, \boldsymbol{W}^V$) and a linear projection ($\boldsymbol{W}^O$). Let $\boldsymbol{A}_l^k$ and $\boldsymbol{B}_l^k$ where $k \in \{Q, K, V, O\}$ be the in- and out-dimension expansion matrices (Eq. 6) for query, key, value, and projection in the l-th layer, respectively. To make sure the new input and output channels are aligned across modules, we tie LEGO operator as follows: (1) $\boldsymbol{A}_l^k = \boldsymbol{B}^{(emb)^\top}$ for $\forall k \in \{Q, K, V\}$, (2) $\boldsymbol{A}_l^O = (\boldsymbol{B}_l^{V^\top})$, (3) $\boldsymbol{B}_l^O = \boldsymbol{B}^{(emb)}$ for $\forall l \in [L_1]$, among which the last constraint is added to take into account residual connections (Chen et al., 2021b). We similarly tie parameters for the feed-forward networks. Specifically, we ensure $\boldsymbol{A}_l^{(fc1)} = \boldsymbol{B}^{(emb)\top}$, $\boldsymbol{A}_l^{(fc2)} = \boldsymbol{B}_l^{(fc1)\top}$ and $\boldsymbol{B}_l^{(fc2)} = \boldsymbol{B}^{(emb)}$.

Since transformers make heavy use of residual layers with skip connections, we found that simply using the same $\boldsymbol{B}^{(emb)}$ to parameterize $\boldsymbol{A}_l^k$ and $\boldsymbol{B}_l^k$ for many layers/modules worked well in prac-

tice. This reduces the number of learnable parameters even further and enables fast learning of $\boldsymbol{M}$ on a small amount of data (100 gradient steps).

3.4 Connection to Monarch Matrices

As shown in Sec. 3.2.1, our depth-width decomposition factorizes $\boldsymbol{M}$ into a multiplication of two structured sparse matrices. We examine the expressiveness of this factorized representation by relating it to Monarch matrices (Dao et al., 2022), defined below.

Definition 1. *Let the space of Monarch matrices be $\mathcal{M} \subseteq \mathbb{R}^{mn_1 \times mn_2}$. Then matrix $\boldsymbol{M} \in \mathcal{M}$ if $\boldsymbol{M} = \boldsymbol{P}_1 \boldsymbol{L} \boldsymbol{P}_2^\top \boldsymbol{R} = \boldsymbol{P}_1 \operatorname{diag}(\boldsymbol{L}_1, \cdots, \boldsymbol{L}_{n_1}) \boldsymbol{P}_2^\top \operatorname{diag}(\boldsymbol{R}_1, \cdots, \boldsymbol{R}_{n_2})$ where $\boldsymbol{L}_i \in \mathbb{R}^{b_1 \times b_2}$, $\boldsymbol{R}_i \in \mathbb{R}^{b_3 \times b_4}$ are dense rectangular matrices, and $n_1 b_2 = n_2 b_3$. $\boldsymbol{P}_1$ is the permutation $\pi(i) = (i - b_1 \lfloor \frac{i}{b_1} \rfloor - 1) n_1 + \lfloor \frac{i}{b_1} \rfloor + 1$ and $\boldsymbol{P}_2$ is the permutation $\pi(j) = (j - b_2 \lfloor \frac{j}{b_2} \rfloor - 1) n_1 + \lfloor \frac{j}{b_2} \rfloor + 1$.*

It is clear that the block-diagonal matrix $\boldsymbol{R}$ has the identical form to our width growing operator $\boldsymbol{R}_{width}$. By applying the permutation matrices $\boldsymbol{P}_1$ and $\boldsymbol{P}_2$ to $\boldsymbol{L}$, $\boldsymbol{L}$ is transformed into the exactly same form with our depth-growth operator $\boldsymbol{L}_{depth}$ in Eq. 5. This implies that our depth-width decomposition coincides with Monarch sparsification of dense matrices, which generalize butterfly matrices (Dao et al., 2019) and enjoy rich expressivity properties (Dao et al., 2020; 2022).

4 Experiments

We conduct experiments to answer three key research questions. Q1: To what extent LEGO can improve the training efficiency (FLOPs and wall time) of transformers compared to training from scratch and other growth operators? Q2: Can LEGO be universally effective across transformers from different domains (e.g., language and vision) and sizes? Q3: Can models trained using LEGO achieve similar performance compared to the baselines when transferred to other downstream task?

4.1 Experimental Setup

Datasets. We follow (Tan & Bansal, 2020) to use the English Wikipedia corpus[4] for training BERT (Devlin et al., 2019) and RoBERTa (Liu et al., 2019). We use the the public C4 (Raffel et al., 2020) dataset for training GPT2 (Radford et al., 2019). We use ImageNet (Deng et al., 2009) for training vision transformers. We use GLUE (Wang et al., 2018), SQuADv1.1 (Rajpurkar et al., 2016), and SQuADv2.0 (Rajpurkar et al., 2018) for evaluating pretrained BERT models. We test downstream performance of vision transformers (DeiT (Touvron et al., 2021a)) by performing transfer learning on 5 downstream image classification tasks, including CIFAR10 (Krizhevsky et al., 2009), CIFAR100 (Krizhevsky et al., 2009), Flowers102 (Nilsback & Zisserman, 2008), Stanford-Cars (Krause et al., 2013), and ChestXRay8 (Wang et al., 2017).

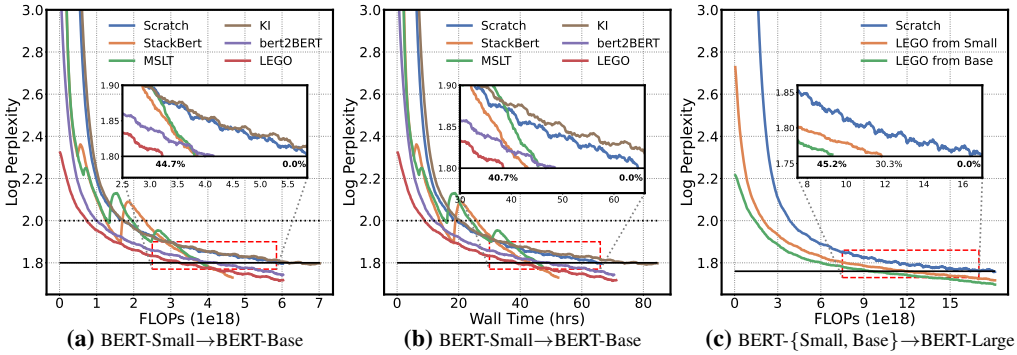
Models. We experiment with following transformers. (1) BERT-Small→BERT-Base, BERT-Base→BERT-Large, BERT-Small→BERT-Large for BERT, (2) RoBERTa-Small→RoBERTa-Base for RoBERTa, (3) GPT2-Base→GPT2-Large for GPT2, (4) DeiT-S→DeiT-B, and (5) CaiT-XS→CaiT-S for vision transformer.[5] More details are included in the Appendix C.

Baselines. We compare our approach with the following baselines. (1) training from scratch baseline where we train the larger transformer without using any smaller pretrained models, (2) progressive training methods designed for growing depth in transformers (StackBERT (Gong et al., 2019) and MSLT (Yang et al., 2020)), (3) bert2BERT (Chen et al., 2021b) that extends Net2Net (Chen et al., 2015) for width expansion and stacking for depth expansion, (4) KI (Qin et al., 2021) which uses distillation for transferring knowledge from the smaller model to the larger model.

Implementation Details. We always use 100 gradient steps to learn the LEGO operator for all models, which is negligible in terms of FLOPs/wall time compared to full training after initialization. We train both BERT and RoBERTa models for 400K steps with a warmup of 10K steps. We remove the next-sentence prediction task (Liu et al., 2019) and use a fixed sequence length of 128 for pretraining both models. For BERT, we use a batch size of 256 and a learning rate of $2e^{-4}$, while we use a batch size of 1024 and a learning rate of $8e^{-4}$ for training RoBERTa models. Following (Shen et al., 2022), we train GPT2 models with a batch size of 384 and sequence length

[4]BERT (Devlin et al., 2019) also uses Toronto Book Corpus (Zhu et al., 2015) which no longer publicly available. We exclude it in our work to ensure reproducibility.

[5]BERT-Small has 6 layers with 512 hidden dimensions, while other named models are their usual sizes.



(a) BERT-Small→BERT-Base **(b)** BERT-Small→BERT-Base **(c)** BERT-{Small, Base}→BERT-Large

Figure 2: Results on BERT. (a-b) shows validation log perplexity vs. FLOPs and wall time respectively, in training BERT-Base by reusing BERT-Small of almost half of the size. (c) shows log perplexity vs. FLOPs in growing BERT-Small and BERT-Base to BERT-Large. The solid line indicates the final perplexity of large model trained from scratch, while the dotted line represents performance of small models. LEGO offers about 45% savings in FLOPs and 40% savings in wall time over BERT-Base training from scratch. Our approach is also flexible in reusing either BERT-Small or BERT-Base for accelerating BERT-Large training.

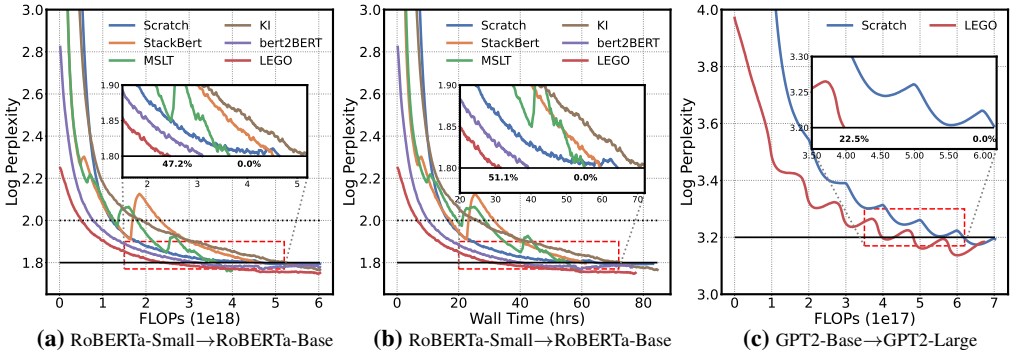
Table 1: Downstream Performance on GLUE and SQuAD. All of the results are based on BERT-Base models trained using different baselines. LEGO achieves similar or even better performance than the original training from scratch baseline on several downstream tasks, despite improving training efficiency.

Method	Savings (FLOPs)	Savings (Walltime)	SST-2 (Acc.)	MNLI (Acc.)	MRPC (Acc.)	CoLA (Acc.)	QNLI (Acc.)	QQP (Acc.)	STS-B (Acc.)	SQuADv1.1 (F1/EM)	SQuADv2.0 (F1/EM)	Avg. GLUE	Avg. SQuAD
Scratch	–	–	88.19	78.43	85.78	62.09	87.06	87.18	86.99	86.55 / 77.31	71.31 / 67.07	82.25	78.79 / 72.19
StackBERT	34.1%	33.3%	88.99	79.72	85.29	59.09	87.28	89.17	86.97	86.50 / 77.42	71.32 / 67.41	82.36	78.91 / 72.41
MSLT	34.9%	30.0%	88.53	78.10	82.60	64.76	83.58	88.54	85.89	86.07 / 76.73	70.68 / 67.17	81.72	78.47 / 71.95
KI	-5.7%	-13.9%	88.65	78.83	83.50	64.86	86.25	88.96	87.09	84.93 / 76.29	71.09 / 67.41	82.59	78.01 / 71.85
bert2BERT	29.0%	25.1%	88.30	80.05	85.54	61.73	88.16	86.18	87.00	86.24 / 77.09	71.52 / 66.85	82.42	78.88 / 71.97
LEGO	44.7%	40.7%	88.42	79.29	84.31	62.09	88.07	88.81	87.00	86.28 / 77.45	71.24 / 67.17	82.57	78.76 / 72.31

of 1024. For vision transformer, we build our models based on DeiT (Touvron et al., 2021a) and CaiT (Touvron et al., 2021b), and apply their default hyper-parameters for training on ImageNet dataset. We train all our vision transformers for 300 epochs with a batch size of 1024. For transfer learning with BERT/RoBERTa, we follow (Tan & Bansal, 2020), we train for 3 epochs with a learning rate of $1e^{-4}$ and a batch-size of 32 for all tasks in GLUE. On SQuAD v1.1 and SQuAD 2.0, we fine-tune for 2 epochs with a learning rate of $5e^{-5}$ and a batch size of 12. We run both GLUE and SQuAD evaluations three times with different random seeds and report the mean numbers. For transfer learning experiments on DeiT, we finetune the pretrained models with 1000 epochs, batch size 768, learning rate 0.01, and use the same data augmentation in training on ImageNet. We use the same pretraining data and experimental settings for all the baselines (including our approach) for a fair comparison. Note that we include the additional compute required for training LEGO in all our tables and figures. However, since our LEGO operator is parameter efficient and we only train it for 100 steps, the influence on visualization and quantitative saving percentages is negligible.

4.2 Results and Analysis

BERT. Figure 2 shows comparison between the different baselines for training BERT models. As seen from Figure 2(a), LEGO saves 44.7% computational cost (FLOPs) of training BERT-Base (12 layers, 768 dimensions) from scratch by reusing BERT-Small which is almost half of the size. On the same setting, LEGO offers 40.7% savings in wall time compared to training from scratch (Figure 2(b)). Among the compared methods, StackBERT is the most competitive in terms of both FLOPs and wall time. However, our data-driven approach to growth obtains +10.6% and +7.2% improvements in FLOPs and wall time on top of StackBERT. Similarly, LEGO significantly outperforms the recent bert2BERT method which saves about 30% computational costs. We observe that KI does not provide any real savings in training as it requires additional computation for knowledge distillation. Furthermore, Figure 2(c) shows that our LEGO approach is flexible in growing either BERT-Small or BERT-Base for accelerating BERT-Large training. As expected, reusing BERT-Base instead of BERT-Small leads to 14.9% more savings in FLOPs (45.2% vs 30.3%) as BERT-Base contains more implicit knowledge in its parameters. Finally, Table 3 shows the per-task performance of different BERT-Base models on both GLUE and SQuAD benchmarks. Despite being trained significantly faster than scratch, BERT trained with our LEGO approach achieve very similar performance comparing to the baselines on both benchmarks.

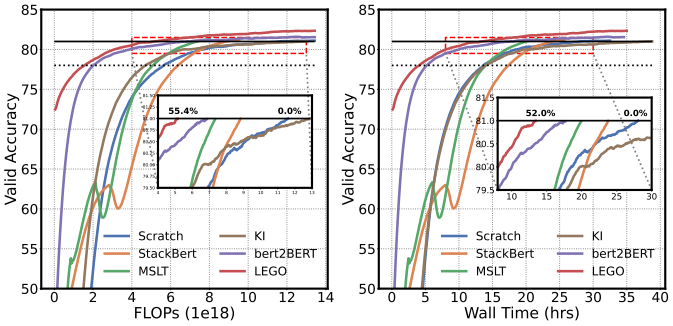


(a) RoBERTa-Small→RoBERTa-Base **(b)** RoBERTa-Small→RoBERTa-Base **(c)** GPT2-Base→GPT2-Large

Figure 3: Results on RoBERTa and GPT2. LEGO reduces FLOPs by 47.2% and 22.5% for RoBERTa-Base and GPT2-Large training, demonstrating its effectiveness across different training strategies and architectures.

RoBERTa and GPT2. Figure 3(a-b) shows the results on RoBERTa, whose training recipe uses larger batch size and learning rate than BERT. LEGO similarly accelerates RoBERTa training, which indicates that our method is robust to optimization hyperparameters. On GPT2, LEGO saves 22.5% computation cost of training GPT2-Large (345M parameters) by reusing GPT2-Base (117M parameters) (Figure 3(c)). These consistent improvements clearly show that LEGO is universally effective for accelerating training of transformers across different model architectures and sizes.

Vision Transformer. Figure 4 shows that by using DeiT-S, LEGO can save 55.4% FLOPs and 52% GPU wall time to reach the same performance of 81% on ImageNet. Interestingly, the model initialized by our data-driven growth operator (w/ only 100 gradient steps of tuning) can already achieve 72% accuracy at the beginning of training and leads to the final accuracy of 81.7% at the end of the training. Compared to the next best method, bert2BERT, LEGO obtains more than 15% savings, which once again demonstrates the effectiveness of our approach in growing vision transformers as well. Tab. 2 shows that fine-tuning results on downstream tasks are also on-par performance with model trained from scratch, showing that LEGO does not harm the model generalization capability when transferred to downstream datasets. We also find similar savings in CaiT-XS→CaiT-S where LEGO saves FLOPs by 52.6% and wall time by 46.1% over training CaiT-S from scratch on ImageNet (see Appendix H for more details).



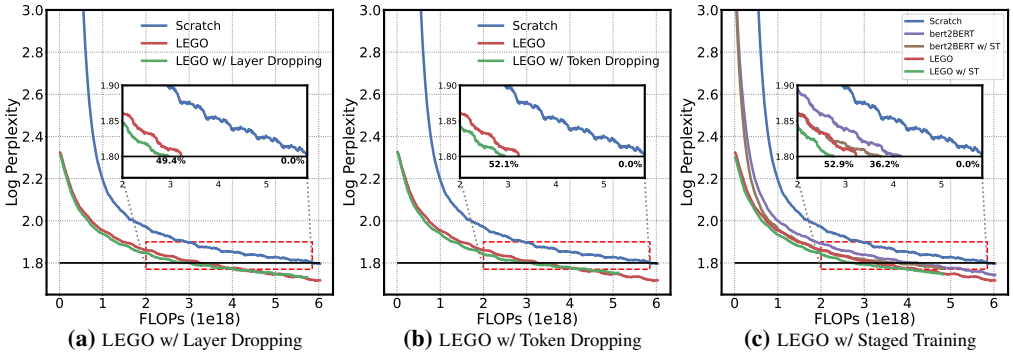
(a) DeiT-S→DeiT-B **(b)** DeiT-S→DeiT-B

Figure 4: Results on DeiT. (a) accuracy vs. flops and (b) accuracy vs. wall time, for training DeiT-B. LEGO saves flops and wall time by more than 50% over training from scratch on ImageNet.

Combining with Other Training Strategies. We also validate that our LEGO approach can be effectively combined with other orthogonal strategies such as layer dropping (Zhang & He, 2020), token dropping (Hou et al., 2022), and staged training (Chen et al., 2021b). More details are included in Appendix D. Figure 5 shows that LEGO can be combined with other training techniques to improve the computational savings by 4.7%, 7.4%, and 8.2% with layer dropping, token dropping and staged training, respectively. Following (Chen et al., 2021b), we also apply staged training strategy to bert2BERT and observe that LEGO still outperforms bert2BERT with staged training by 16.7% (see Figure 5(c)).

Table 2: Transfer Learning Performance of DeiT-B. Despite being trained twice faster, DeiT-B model trained using LEGO are very competitive with original training from scratch baseline on all downstream tasks.

Method	FLOPs	Walltime	ImageNet	CIFAR10	CIFAR100	Flowers	Cars	ChestXRay8
Scratch	–	–	81.10	99.09	90.76	97.79	92.06	55.81
StackBERT	23.8%	15.1%	81.21	99.11	90.80	97.56	92.09	55.77
MSLT	36.7%	28.9%	81.27	99.07	90.21	97.71	92.11	55.79
KI	−11.2%	−36.8%	81.01	98.94	90.32	97.81	92.08	55.80
bert2BERT	40.8%	37.0%	81.59	99.14	90.69	97.67	92.15	55.82
LEGO	55.4%	52.0%	81.71	99.12	90.74	97.77	92.09	55.82

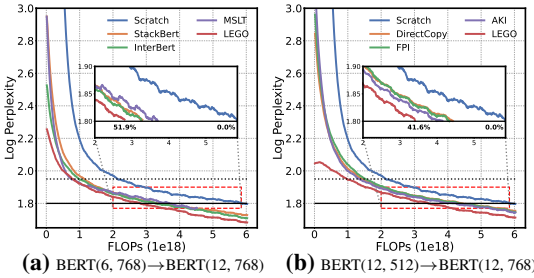


(a) LEGO w/ Layer Dropping **(b)** LEGO w/ Token Dropping **(c)** LEGO w/ Staged Training

Figure 5: LEGO with Orthogonal Training Strategies. Our approach can be combined with (a) layer dropping, (b) token dropping, and (c) staged training (ST), for further accelerate BERT training.

4.3 Ablation Studies

Depth Only Expansion. In this experiment, we examine the effectiveness of our proposed depth expansion operator ($\boldsymbol{L}_{depth}$) by only growing depth of a BERT from 6 layers to 12 layers, i.e, (BERT(6, 768)→BERT(12, 768). We compare with stacking (StackBERT, Gong et al., 2019), Interpolation (InterBERT, Chang et al., 2017; Dong et al., 2020) (see Eq. 1), and MSLT (Yang et al., 2020). For LEGO, we only apply its $\boldsymbol{L}_{depth}$ component to the pre-trained model weights. Results in Figure 6(a) shows that a data-driven approach works well even when just growing across the depth dimension.



(a) BERT(6, 768)→BERT(12, 768) **(b)** BERT(12, 512)→BERT(12, 768)

Figure 6: Results on Depth Only and Width Only Extension. LEGO saves computation cost by 51.7% and 41.6% over the original training from scratch, while extending only depth or width separately.

Width Only Expansion. We also verify the effectiveness of $\boldsymbol{R}_{width}$ by only extending BERT width from 512 to 768, i.e., (BERT(12, 512)→BERT(12, 768). We compare LEGO based initialization with direct copy (Wei et al., 2016), function preserving initialization (FPI, Chen et al., 2015), and advanced knowledge initialization (AKI, Chen et al., 2021b). LEGO's width expansion component outperforms all other methods, as shown in Figure 6(b).

Table 3: Effect of number of gradient steps. "+FLOPs" stands for additional flops (in 10^{15}).

# of Steps	+FLOPs	Savings
100	3.61	44.7%
500	18.06	44.5%
1000	36.13	44.2%
10000	361.30	38.9%

Learning Steps. Our main experiments just use 100 gradient steps to grow. We tune our LEGO operator on the pretraining set for 100, 500, 1000, and 10000 steps and compute the additional FLOPs for BERT-Small→BERT-Base training. Table 3 shows that training LEGO operator within 1000 steps result in the identical model convergence (reaching 1.8 PPL at 215K steps). This suggests tuning model weights under the linear constraints of LEGO can achieve a faster convergence. Training LEGO operator for more than 10000 steps can provide a model with slightly faster convergence (214K steps), but results in less savings overall.

5 Conclusion

We introduced an efficient data-driven approach for accelerating training by learning to grow transformers. We initialize the larger transformer's parameters as a linear mapping from the smaller pretrained model's parameters and factorize the mapping to be a composition of sparse width- and depth-expansion operators with a Kronecker factorization that groups parameters into layers and neurons. We demonstrate the effectiveness of our proposed approach on both language and vision transformers of different sizes, outperforming several competing methods. While our compute resources prevented us from applying LEGO to even larger transformers, it would be interesting to see if this can be applied on top of extremely large models such as GPT3 (Brown et al., 2020).

REFERENCES

Andy Brock, Soham De, Samuel L Smith, and Karen Simonyan. High-performance large-scale image recognition without normalization. In *International Conference on Machine Learning*, pp. 1059–1071, 2021.

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 *Proceedings of NeurIPS*, 2020.

Han Cai, Tianyao Chen, Weinan Zhang, Yong Yu, and Jun Wang. Efficient architecture search by network transformation. In *Proceedings of AAAI*, 2018.

Bo Chang, Lili Meng, Eldad Haber, Frederick Tung, and David Begert. Multi-level residual networks from dynamical systems view. *arXiv preprint arXiv:1710.10348*, 2017.

Beidi Chen, Tri Dao, Kaizhao Liang, Jiaming Yang, Zhao Song, Atri Rudra, and Christopher Re. Pixelated butterfly: Simple and efficient sparse training for neural network models. *arXiv preprint arXiv:2112.00029*, 2021a.

Cheng Chen, Yichun Yin, Lifeng Shang, Xin Jiang, Yujia Qin, Fengyu Wang, Zhi Wang, Xiao Chen, Zhiyuan Liu, and Qun Liu. bert2bert: Towards reusable pretrained language models. *arXiv preprint arXiv:2110.07143*, 2021b.

Ricky TQ Chen, Yulia Rubanova, Jesse Bettencourt, and David K Duvenaud. Neural ordinary differential equations. *Advances in neural information processing systems*, 31, 2018.

Tianqi Chen, Ian Goodfellow, and Jonathon Shlens. Net2net: Accelerating learning via knowledge transfer. *arXiv preprint arXiv:1511.05641*, 2015.

Justin Chiu, Yuntian Deng, and Alexander Rush. Low-rank constraints for fast inference in structured models. *Advances in Neural Information Processing Systems*, 34:2887–2898, 2021.

Xiaoliang Dai, Hongxu Yin, and Niraj K Jha. Nest: A neural network synthesis tool based on a grow-and-prune paradigm. *IEEE Transactions on Computers*, 68(10):1487–1497, 2019.

Tri Dao, Albert Gu, Matthew Eichhorn, Atri Rudra, and Christopher Ré. Learning fast algorithms for linear transforms using butterfly factorizations. In *International conference on machine learning*, pp. 1517–1527, 2019.

Tri Dao, Nimit S Sohoni, Albert Gu, Matthew Eichhorn, Amit Blonder, Megan Leszczynski, Atri Rudra, and Christopher Ré. Kaleidoscope: An efficient, learnable representation for all structured linear maps. *arXiv preprint arXiv:2012.14966*, 2020.

Tri Dao, Beidi Chen, Nimit S Sohoni, Arjun Desai, Michael Poli, Jessica Grogan, Alexander Liu, Aniruddh Rao, Atri Rudra, and Christopher Ré. Monarch: Expressive structured matrices for efficient and accurate training. In *International Conference on Machine Learning*, pp. 4690–4721, 2022.

Yann N Dauphin and Samuel Schoenholz. Metainit: Initializing learning by learning to initialize. *Advances in Neural Information Processing Systems*, 32, 2019.

Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. Imagenet: A large-scale hierarchical image database. In *2009 IEEE conference on computer vision and pattern recognition*, pp. 248–255, 2009.

Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. In *Proceedings of NAACL*, 2019.

Chengyu Dong, Liyuan Liu, Zichao Li, and Jingbo Shang. Towards adaptive residual network training: A neural-ode perspective. In *International conference on machine learning*, pp. 2616–2626. PMLR, 2020.

Alexey Dosovitskiy, Lucas Beyer, Alexander Kolesnikov, Dirk Weissenborn, Xiaohua Zhai, Thomas Unterthiner, Mostafa Dehghani, Matthias Minderer, Georg Heigold, Sylvain Gelly, Jakob Uszkoreit, and Neil Houlsby. An Image is Worth 16x16 Words: Transformers for Image Recognition at Scale. In *Proceedings of ICLR*, 2021.

Utku Evci, Max Vladymyrov, Thomas Unterthiner, Bart van Merriënboer, and Fabian Pedregosa. Gradmax: Growing neural networks using gradient information. *arXiv preprint arXiv:2201.05125*, 2022.

Scott Fahlman. The recurrent cascade-correlation architecture. In *Advances in Neural Information Processing Systems*, 1990.

Scott Fahlman and Christian Lebiere. The cascade-correlation learning architecture. In *Advances in Neural Information Processing Systems*, 1989.

Xavier Glorot and Yoshua Bengio. Understanding the difficulty of training deep feedforward neural networks. In *Proceedings of the thirteenth international conference on artificial intelligence and statistics*, pp. 249–256, 2010.

Linyuan Gong, Di He, Zhuohan Li, Tao Qin, Liwei Wang, and Tieyan Liu. Efficient training of bert by progressively stacking. In *International conference on machine learning*, pp. 2337–2346, 2019.

Xiaotao Gu, Liyuan Liu, Hongkun Yu, Jing Li, Chen Chen, and Jiawei Han. On the transformer growth for progressive bert training. *arXiv preprint arXiv:2010.12562*, 2020.

Steven Gutstein, Olac Fuentes, , and Eric Freudenthal. Knowledge transfer in deep convolutional neural nets. In *Proceedings of International Journal on Artificial Intelligence Tools*, 2008.

Song Han, Huizi Mao, and William J Dally. Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding. *arXiv preprint arXiv:1510.00149*, 2015.

Le Hou, Richard Yuanzhe Pang, Tianyi Zhou, Yuexin Wu, Xinying Song, Xiaodan Song, and Denny Zhou. Token dropping for efficient bert pretraining. *arXiv preprint arXiv:2203.13240*, 2022.

Neil Houlsby, Andrei Giurgiu, Stanislaw Jastrzebski, Bruna Morrone, Quentin De Laroussilhe, Andrea Gesmundo, Mona Attariyan, and Sylvain Gelly. Parameter-efficient transfer learning for nlp. In *International Conference on Machine Learning*, 2019.

Xiao Shi Huang, Felipe Perez, Jimmy Ba, and Maksims Volkovs. Improving transformer optimization through better initialization. In *International Conference on Machine Learning*, pp. 4475–4483, 2020.

Yanping Huang, Youlong Cheng, Ankur Bapna, Orhan Firat, Dehao Chen, Mia Chen, HyoukJoong Lee, Jiquan Ngiam, Quoc V Le, Yonghui Wu, et al. Gpipe: Efficient training of giant neural networks using pipeline parallelism. *Advances in neural information processing systems*, 32, 2019.

Menglin Jia, Luming Tang, Bor-Chun Chen, Claire Cardie, Serge Belongie, Bharath Hariharan, and Ser-Nam Lim. Visual prompt tuning. *arXiv preprint arXiv:2203.12119*, 2022.

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.

Yannic Kilcher, Gary Bécigneul, and Thomas Hofmann. Escaping flat areas via function-preserving structural network modifications. 2018.

Jonathan Krause, Michael Stark, Jia Deng, and Li Fei-Fei. 3d object representations for fine-grained categorization. In *Proceedings of the IEEE international conference on computer vision workshops*, pp. 554–561, 2013.

Alex Krizhevsky, Geoffrey Hinton, et al. Learning multiple layers of features from tiny images. 2009.

Brian Lester, Rami Al-Rfou, and Noah Constant. The power of scale for parameter-efficient prompt tuning. *arXiv preprint arXiv:2104.08691*, 2021.

Changlin Li, Bohan Zhuang, Guangrun Wang, Xiaodan Liang, Xiaojun Chang, and Yi Yang. Automated progressive learning for efficient training of vision transformers. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pp. 12486–12496, 2022.

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.

Dmytro Mishkin and Jiri Matas. All you need is a good init. *arXiv preprint arXiv:1511.06422*, 2015.

Maria-Elena Nilsback and Andrew Zisserman. Automated flower classification over a large number of classes. In *2008 Sixth Indian Conference on Computer Vision, Graphics & Image Processing*, pp. 722–729. IEEE, 2008.

Jonas Pfeiffer, Aishwarya Kamath, Andreas Rücklé, Kyunghyun Cho, and Iryna Gurevych. Adapterfusion: Non-destructive task composition for transfer learning. *arXiv preprint arXiv:2005.00247*, 2020.

Yujia Qin, Yankai Lin, Jing Yi, Jiajie Zhang, Xu Han, Zhengyan Zhang, Yusheng Su, Zhiyuan Liu, Peng Li, Maosong Sun, et al. Knowledge inheritance for pre-trained language models. *arXiv preprint arXiv:2105.13880*, 2021.

Alec Radford, Karthik Narasimhan, Tim Salimans, and Ilya Sutskever. Improving language understanding by generative pre-training. 2018.

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

Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, Peter J Liu, et al. Exploring the limits of transfer learning with a unified text-to-text transformer. *J. Mach. Learn. Res.*, 21(140):1–67, 2020.

Pranav Rajpurkar, Jian Zhang, Konstantin Lopyrev, and Percy Liang. Squad: 100,000+ questions for machine comprehension of text. *arXiv preprint arXiv:1606.05250*, 2016.

Pranav Rajpurkar, Robin Jia, and Percy Liang. Know what you don't know: Unanswerable questions for squad. *arXiv preprint arXiv:1806.03822*, 2018.

Alexander Rives, Joshua Meier, Tom Sercu, Siddharth Goyal, Zeming Lin, Jason Liu, Demi Guo, Myle Ott, C. Lawrence Zitnick, Jerry Ma, and Rob Fergus. Biological structure and function emerge from scaling unsupervised learning to 250 million protein sequences. *Proceedings of the National Academy of Sciences*, 118(15), 2021. ISSN 0027-8424. doi: 10.1073/pnas.2016239118.

Jonathan S Rosenfeld, Amir Rosenfeld, Yonatan Belinkov, and Nir Shavit. A constructive prediction of the generalization error across scales. *arXiv preprint arXiv:1909.12673*, 2019.

Kathrin Schacke. On the kronecker product. *Master's thesis, University of Waterloo*, 2004.

Sheng Shen, Pete Walsh, Kurt Keutzer, Jesse Dodge, Matthew Peters, and Iz Beltagy. Staged training for transformer language models. *arXiv preprint arXiv:2203.06211*, 2022.

Mohammad Shoeybi, Mostofa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, and Bryan Catanzaro. Megatron-lm: Training multi-billion parameter language models using model parallelism. *arXiv preprint arXiv:1909.08053*, 2019.

Vikas Sindhwani, Tara Sainath, and Sanjiv Kumar. Structured transforms for small-footprint deep learning. *Advances in Neural Information Processing Systems*, 28, 2015.

Hao Tan and Mohit Bansal. Vokenization: Improving language understanding with contextualized, visual-grounded supervision. *arXiv preprint arXiv:2010.06775*, 2020.

Shanshan Tang, Bo Li, and Haijun Yu. Chebnet: Efficient and stable constructions of deep neural networks with rectified power units using chebyshev approximations. *arXiv preprint arXiv:1911.05467*, 2019.

Hugo Touvron, Matthieu Cord, Matthijs Douze, Francisco Massa, Alexandre Sablayrolles, and Hervé Jégou. Training data-efficient image transformers & distillation through attention. In *International Conference on Machine Learning*, pp. 10347–10357, 2021a.

Hugo Touvron, Matthieu Cord, Alexandre Sablayrolles, Gabriel Synnaeve, and Hervé Jégou. Going deeper with image transformers. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, 2021b.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is All You Need. In *Proceedings of NeurIPS*, 2017.

Alex Wang, Amanpreet Singh, Julian Michael, Felix Hill, Omer Levy, and Samuel Bowman. Glue: A multi-task benchmark and analysis platform for natural language understanding. In *Proceedings of the 2018 EMNLP Workshop BlackboxNLP: Analyzing and Interpreting Neural Networks for NLP*, pp. 353–355, 2018.

Xiaosong Wang, Yifan Peng, Le Lu, Zhiyong Lu, Mohammadhadi Bagheri, and Ronald M Summers. Chestx-ray8: Hospital-scale chest x-ray database and benchmarks on weakly-supervised classification and localization of common thorax diseases. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 2097–2106, 2017.

Tao Wei, Changhu Wang, Yong Rui, and Chang Wen Chen. Network morphism. In Maria Florina Balcan and Kilian Q. Weinberger (eds.), *Proceedings of The 33rd International Conference on Machine Learning*, pp. 564–572, 2016.

Lemeng Wu, Dilin Wang, and Qiang Liu. Splitting steepest descent for growing neural architectures. *Advances in neural information processing systems*, 32, 2019.

Lemeng Wu, Dilin Wang, Peter Stone, and Qiang Liu. Firefly neural architecture descent: a general approach for growing neural networks. *Advances in neural information processing systems*, 2021.

Cheng Yang, Shengnan Wang, Chao Yang, Yuechuan Li, Ru He, and Jingqiao Zhang. Progressively stacking 2.0: A multi-stage layerwise training method for bert training speedup. *arXiv preprint arXiv:2011.13635*, 2020.

Yang You, Jing Li, Sashank Reddi, Jonathan Hseu, Sanjiv Kumar, Srinadh Bhojanapalli, Xiaodan Song, James Demmel, Kurt Keutzer, and Cho-Jui Hsieh. Large batch optimization for deep learning: Training bert in 76 minutes. *arXiv preprint arXiv:1904.00962*, 2019.

Hongyi Zhang, Yann N Dauphin, and Tengyu Ma. Fixup initialization: Residual learning without normalization. *arXiv preprint arXiv:1901.09321*, 2019.

Minjia Zhang and Yuxiong He. Accelerating training of transformer-based language models with progressive layer dropping. *Advances in Neural Information Processing Systems*, 33:14011–14023, 2020.

Xu Zhang, Felix X Yu, Ruiqi Guo, Sanjiv Kumar, Shengjin Wang, and Shi-Fu Chang. Fast orthogonal projection based on kronecker product. In *Proceedings of the IEEE International Conference on Computer Vision*, pp. 2929–2937, 2015.

Chen Zhu, Renkun Ni, Zheng Xu, Kezhi Kong, W Ronny Huang, and Tom Goldstein. Gradinit: Learning to initialize neural networks for stable and efficient training. *Advances in Neural Information Processing Systems*, 34:16410–16422, 2021.

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

A UNIVERSALITY OF LEGO OPERATOR

Proposition 1. *StackBERT (Eq. 1), Interpolation (Eq. 1), and Net2Net (Eq. 2) are all the special cases of the LEGO operator (Eq. 8).*

Proof. We prove Proposition 1 by constructing parameters in $\boldsymbol{L}_{depth}$ and $\boldsymbol{R}_{width}$.

Stacking. Stacking based methods (Gong et al., 2019; Yang et al., 2020) duplicate the entire lower blocks on top of the small model to the form new layers (Eq. 1). Formally, we show this operation can be done by the following operator:

$$\boldsymbol{M} = \underbrace{\begin{bmatrix} \boldsymbol{I} & & \\ & \boldsymbol{I} & \\ & & \ddots \\ \boldsymbol{I} & & \\ & \boldsymbol{I} & \\ & & \ddots \end{bmatrix}}_{\boldsymbol{L}_{depth}} \underbrace{\begin{bmatrix} \boldsymbol{I} & & \\ & \ddots & \\ & & \boldsymbol{I} \end{bmatrix}}_{\boldsymbol{R}_{width}} \tag{9}$$

Interpolation. Interpolation based methods (Chang et al., 2017; Dong et al., 2020) interleave each layer for twice. We can construct the following matrix to achieve layer interpolation (Eq. 1).

$$\boldsymbol{M} = \underbrace{\begin{bmatrix} \boldsymbol{I} & & \\ \boldsymbol{I} & & \\ & \boldsymbol{I} & \\ & \boldsymbol{I} & \\ & & \ddots \\ & & \ddots \end{bmatrix}}_{\boldsymbol{L}_{depth}} \underbrace{\begin{bmatrix} \boldsymbol{I} & & \\ & \ddots & \\ & & \boldsymbol{I} \end{bmatrix}}_{\boldsymbol{R}_{width}} \tag{10}$$

We remark that any rearrangement of layers to construct new layers (mathematically a permutation of existing layers with replacement) can be constructed in a similar way. Thus, our LEGO operator is a super set of them all.

Net2Net. Since we show in Eq. 6, the Kronecker factorization on $\boldsymbol{R}_l$ amounts to decomposing the general growth operator into in-dimension and out-dimension expansion. We can construct Net2Net (Chen et al., 2015) based growth by simply letting:

$$\boldsymbol{L}_{depth} = \boldsymbol{I} \in \mathbb{R}^{L_1 D_2 \times L_2 D_2}, \quad \boldsymbol{R}_{width} = \begin{bmatrix} \boldsymbol{A}_1 \otimes \boldsymbol{B}_1 & & \\ & \ddots & \\ & & \boldsymbol{A}_{L_1} \otimes \boldsymbol{B}_{L_1} \end{bmatrix} \tag{11}$$

$$\boldsymbol{A}_l = \begin{bmatrix} \boldsymbol{I} \\ \widetilde{\boldsymbol{S}}_{l-1} \end{bmatrix}, \quad \boldsymbol{B}_l = \begin{bmatrix} \boldsymbol{I} \\ \boldsymbol{S}_{l,} \end{bmatrix} \tag{12}$$

where $\boldsymbol{S}_l \in \{0,1\}^{(D_2 - D1) \times D_1}$ is a selection matrix to enlarge the out dimension, and $\widetilde{\boldsymbol{S}}_{l-1} = \boldsymbol{S}_{l-1} \operatorname{diag}(\boldsymbol{1}^\top \boldsymbol{S}_{l-1})^{-1}$ copies the selection from $\boldsymbol{S}_{l-1}$ with normalization to guarantee functionality preserving in expansion. □

B GROWING TRANSFORMERS WITH LEGO

Transformer consists of embedding layer, multi-block attention layer, and an output layer. The core ingredient attention block consists of a Multi-Head Attention (MHA) module followed by a FeedForward Network (FFN), with a skip connection across the both blocks. Applying LEGO requires the following considerations:

Embedding Layers. For both language and vision transformers, embedding layer can be regarded as a linear layer, whose inputs are one-hot embeddings in language models. We draw a learnable matrix $\boldsymbol{B}^{(emb)}$ to extend its output dimension.

Multi-Head Attention Block. An attention layer in transformer consists of multi-head attention weights ($\boldsymbol{W}^Q, \boldsymbol{W}^K, \boldsymbol{W}^V$) and a linear projection ($\boldsymbol{W}^O$). Let $\boldsymbol{A}_l^k$ and $\boldsymbol{B}_l^k$ with $k \in \{Q, K, V, O\}$ be the in- and out-dimension expansion matrices (Eq. 6) for query, key, value, and projection in the l-th layer, respectively. Applying $\boldsymbol{B}_l^k$ to $\boldsymbol{W}^k$ ($k \in Q, K, V$) constructs new heads by weighted summing rows of all existing heads. To make sure the new input and output channels are aligned across modules, we tie our LEGO operator with the following scheme: (1) $\boldsymbol{A}_l^k = \boldsymbol{B}^{(emb)\top}$ for $\forall k \in \{Q, K, V\}$, (2) $\boldsymbol{A}_l^O = \boldsymbol{B}_l^{(V)\top}$, (3) $\boldsymbol{B}_l^O = \boldsymbol{B}^{(emb)}$ for $\forall l \in [L_1]$, among which the last constraint is added for residual connection (Chen et al., 2021b). Both Bias terms and layer normalization inherit the associated linear transformations' out-dimension expansion matrices to grow the width. For depth expansion, each module independently combines the same module from other layers (Eq. 8) with learnable coefficients $\boldsymbol{w}$.

Feed-Forward Networks. Each attention block is followed by a two-layer FFN. Let $\boldsymbol{A}_l^k$ and $\boldsymbol{B}_l^k$ with $k \in \{fc1, fc2\}$ be the in- and out-dimension expansion matrices (Eq. 6) for the first and second FFN layer in the l-th layer, respectively. Similarly, we tie the parameters for feed-forward networks: $\boldsymbol{A}_l^{(fc1)} = \boldsymbol{B}^{(emb)\top}$, $\boldsymbol{A}_l^{(fc2)} = \boldsymbol{B}_l^{(fc1)\top}$ and $\boldsymbol{B}_l^{(fc2)} = \boldsymbol{B}^{(emb)}$.

Output Layer. For output head, we ensure $\boldsymbol{A}^{(out)} = \boldsymbol{B}^{(emb)\top}$, since the output dimension of attention layers are always aligned with the $\boldsymbol{B}^{(emb)}$ by our construction. The output layer does not need out-dimension expansion.

Algorithm 1 summarizes a forward pass of LEGO with transformer.

C Model Configurations

We summarize the settings of different transformer models used for our experiments in Table 4. For BERT and RoBERTa, we re-use the code base provided by (Tan & Bansal, 2020). For GTP2, we follow the model configuration of OpenAI and use the pre-training code provided by (Shen et al., 2022). For DeiT, we use their official codebase (Touvron et al., 2021a).

Table 4: Configuration of different transformers.

	BERT-Small	BERT-Base	RoBERTa-Small	RoBERTa-Base	GPT2-Base	GPT2-Large		DeiT-S	DeiT-B
# layers	6	12	6	12	12	24	# layers	12	12
# hidden	512	768	512	768	768	1024	# hidden	384	768
# heads	8	12	8	12	12	16	# heads	6	12
# vocab	30522	30522	50265	50265	50257	50257	input res.	224	224
seq. length	128	128	128	128	1024	1024	patch size	16	16

D Orthogonal Training Strategies

For layer dropping, we follow the same progressive dropping rate schedule with (Zhang & He, 2020), and set the maximum dropping rate to 0.1 to recover the performance. For token dropping, we randomly set 15% tokens aside in the middle layers. In the first 50k steps of staged training, only a sub-network is activated and trained, and afterwards, we do full-model training for 350k steps.

E GLUE Performance using AdapterFusion

Our LEGO is mainly proposed for improving efficiency of the pre-training stage and hence is compatible with various fine-tuning schemes like full model finetuning, adapters (Houlsby et al., 2019; Pfeiffer et al., 2020) or prompt tuning (Lester et al., 2021; Jia et al., 2022) for adaptation to downstream tasks. For full model finetuning, Table 1 and Table 2 show that despite being trained significantly faster than scratch, BERT and DeiT trained with our LEGO approach respectively, achieve

Algorithm 1 A forward pass of LEGO with transformer.

Input: A small transformer with hidden D_1 and number of layer L_1. Denote the embedding layer as $\boldsymbol{W}^{(emb)} \in \mathbb{R}^{D_1 \times E}$, attention layers as $\boldsymbol{W}_l^Q, \boldsymbol{W}^K, \boldsymbol{W}_l^V, \boldsymbol{W}_l^O \in \mathbb{R}^{D_1, \times D_1}$, FFN layers as $\boldsymbol{W}_l^{(fc1)} \in \mathbb{R}^{4D_1 \times D_1}$, $\boldsymbol{W}_l^{(fc2)} \in \mathbb{R}^{D_1 \times 4D_1}$, LayerNorm layers as $\boldsymbol{W}_l^{(ln1)} \in \mathbb{R}^{D_1 \times 4D_1}$, $\boldsymbol{W}_l^{(ln1)}, \boldsymbol{W}_l^{(ln2)} \in \mathbb{R}^{D_1}, \forall l \in [L_1]$, the output head $\boldsymbol{W}^{(out)} \in \mathbb{R}^{C \times D_1}$

Output: A large transformer with hidden D_2 and number of layer L_2. Denote the weight matrices as $\boldsymbol{\Omega}$ with the corresponding superscripts as the small model.

$\boldsymbol{\Omega}^{(emb)} \leftarrow \boldsymbol{B}^{(emb)} \boldsymbol{W}^{(emb)}$

for $l = 1, \cdots, L_1$ **do** ▷ Width Expansion

  $\boldsymbol{\Omega}_l^Q \leftarrow \boldsymbol{B}^{(Q)} \boldsymbol{W}_l^Q \boldsymbol{B}^{(emb)\top}$

  $\boldsymbol{\Omega}_l^K \leftarrow \boldsymbol{B}^{(K)} \boldsymbol{W}_l^K \boldsymbol{B}^{(emb)\top}$

  $\boldsymbol{\Omega}_l^V \leftarrow \boldsymbol{B}^{(V)} \boldsymbol{W}_l^V \boldsymbol{B}^{(emb)\top}$

  $\boldsymbol{\Omega}_l^O \leftarrow \boldsymbol{B}^{(emb)} \boldsymbol{W}_l^V \boldsymbol{B}^{(V)\top}$

  $\boldsymbol{\Omega}_l^{(ln1)} \leftarrow \boldsymbol{B}^{(emb)} \boldsymbol{W}_l^{(ln1)}$

  $\boldsymbol{\Omega}_l^{(fc1)} \leftarrow \boldsymbol{B}^{(fc1)} \boldsymbol{W}_l^V \boldsymbol{B}^{(emb)\top}$

  $\boldsymbol{\Omega}_l^{(fc2)} \leftarrow \boldsymbol{B}^{(emb)} \boldsymbol{W}_l^V \boldsymbol{B}^{(fc1)\top}$

  $\boldsymbol{\Omega}_l^{(ln2)} \leftarrow \boldsymbol{B}^{(emb)} \boldsymbol{W}_l^{(ln2)}$

end for

for $l = 1, \cdots, L_2$ **do** ▷ Depth Expansion

  $\boldsymbol{\Omega}_l^Q \leftarrow \sum_{j=1}^{L_1} w_{l,j}^Q \boldsymbol{\Omega}_j^Q$

  $\boldsymbol{\Omega}_l^K \leftarrow \sum_{j=1}^{L_1} w_{l,j}^K \boldsymbol{\Omega}_j^K$

  $\boldsymbol{\Omega}_l^V \leftarrow \sum_{j=1}^{L_1} w_{l,j}^V \boldsymbol{\Omega}_j^V$

  $\boldsymbol{\Omega}_l^O \leftarrow \sum_{j=1}^{L_1} w_{l,j}^O \boldsymbol{\Omega}_j^O$

  $\boldsymbol{\Omega}_l^{(ln1)} \leftarrow \sum_{j=1}^{L_1} w_{l,j}^{(ln1)} \boldsymbol{\Omega}_j^{(ln1)}$

  $\boldsymbol{\Omega}_l^{(fc1)} \leftarrow \sum_{j=1}^{L_1} w_{l,j}^{(fc1)} \boldsymbol{\Omega}_j^{(fc1)}$

  $\boldsymbol{\Omega}_l^{(fc2)} \leftarrow \sum_{j=1}^{L_1} w_{l,j}^{(fc2)} \boldsymbol{\Omega}_j^{(fc2)}$

  $\boldsymbol{\Omega}_l^{(ln2)} \leftarrow \sum_{j=1}^{L_1} w_{l,j}^{(ln2)} \boldsymbol{\Omega}_j^{(ln2)}$

end for

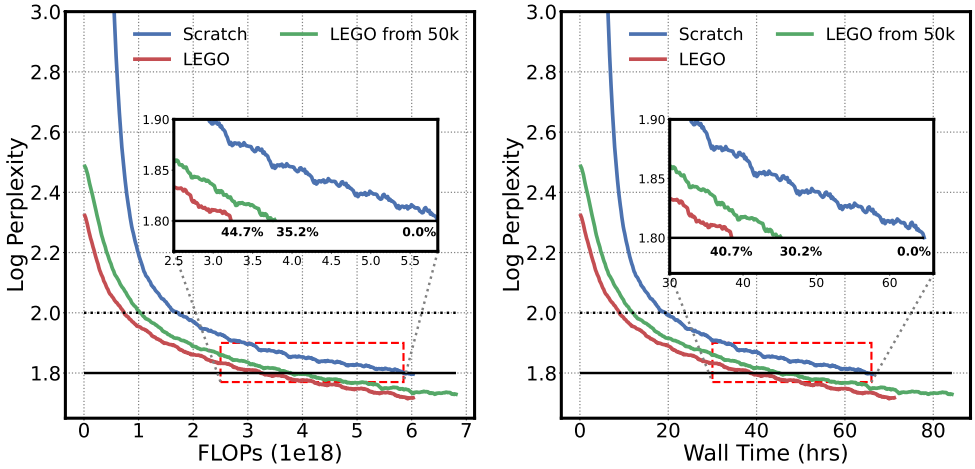
$\boldsymbol{\Omega}^{(out)} \leftarrow \boldsymbol{W}^{(out)} \boldsymbol{B}^{(emb)\top}$

Train transformer with parameters $\boldsymbol{\Omega}$.

Table 5: Downstream performance using AdapterFusion (Pfeiffer et al., 2020) on GLUE Benchmark. All of the results are based on BERT-Base models trained using different baselines.

Method	Savings (FLOPs)	Savings (Walltime)	SST-2 (Acc.)	MNLI (Acc.)	MRPC (Acc.)	CoLA (Acc.)	QNLI (Acc.)	QQP (Acc.)	STS-B (Acc.)	Average (Acc.)
Scratch	–	–	88.41	78.60	86.02	62.39	87.62	88.02	86.52	82.51
StackBERT	34.1%	33.3%	88.78	79.80	85.43	59.56	87.71	89.19	86.27	82.39
MSLT	34.9%	30.0%	88.41	78.35	83.15	63.97	86.19	88.20	86.42	82.10
KI	-5.7%	-13.9%	88.94	78.84	84.00	64.61	86.75	88.19	87.93	82.75
bert2BERT	29.0%	25.1%	88.47	80.53	85.50	62.33	88.57	86.72	87.10	82.75
LEGO	44.7%	40.5%	88.45	80.01	84.67	63.05	88.06	88.92	87.00	82.88

very similar performance comparing to the baselines on downstream tasks. We also test BERT-Base models trained using different baselines by using adapterfusion (Pfeiffer et al., 2020) instead of full finetuning on GLUE benchmark. Table 5 shows that LEGO also achieves on-par performance with model trained from scratch under adapter-based tuning with 44.7% savings in FLOPs abd 40.5% savings in wall time. This shows that LEGO does not harm the model generalization capability when adapters are used as a parameter-efficient finetuning strategy for transferring a trained model to downstream datasets.



(a) BERT-Small→BERT-Base **(b)** BERT-Small→BERT-Base

Figure 7: Results on BERT-Base by reusing BERT-Small trained for 50k steps. Instead of training BERT-Base from fully trained BERT-Small, we run LEGO on BERT-Small trained with 50k steps. LEGO offers about 35.2% savings in FLOPs and 30.2% savings in wall time over the BERT-Base training from scratch.

Table 6: Downstream GLUE performance of StackBERT and LEGO at the end of training. All of the results are based on BERT-Base model trained using BERT-Small as the smaller model.

Method	SST-2 (Acc.)	MNLI (Acc.)	MRPC (Acc.)	CoLA (Acc.)	QNLI (Acc.)	QQP (Acc.)	STS-B (Acc.)	Average (Acc.)
StackBERT	88.57	79.84	85.06	61.21	88.25	89.49	87.02	82.78
LEGO	88.55	79.89	89.48	61.54	88.60	88.63	87.14	82.76

F REUSING SMALLER MODELS TRAINED FOR ONLY FEW STEPS

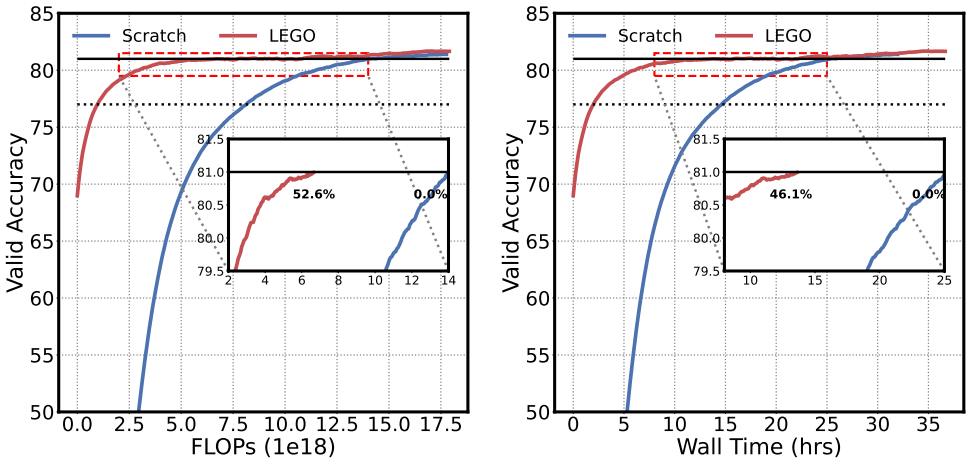
LEGO focuses on utilizing the knowledge of smaller models that have already been pretrained and available. In this section, we investigate how LEGO can leverage smaller existing models that are only trained for few steps to accelerate training of a larger model. We perform an experiment on BERT-Base by reusing a BERT-Small trained for only 50k steps instead full training for 220k steps as used in our experiments. Figure 7 shows that LEGO can still save 35.2% savings in FLOPs and 30.2% savings in wall time over the BERT-Base training from scratch.

G STACKBERT VS LEGO AT THE END OF BERT-BASE TRAINING

Table 1 shows that BERT trained with our LEGO approach achieve very similar performance compared to StackBERT on GLUE benchmark at the point where it reaches the log perplexity of the model trained from scratch. In this section, we analyze the performance of StackBERT and LEGO inn detail for training a BERT-Base by reusing BERT-Small. Figure 2 shows that StackBERT achieves lower log perplexity when FLOPs $> 4e^{18}$. However, both LEGO and StackBERT leads to very similar log perplexity (1.71 vs 1.72) at the end of training. We also check the downstream GLUE performance of both models at the end of training and find that they obtain very similar performance across all the tasks, as shown in Table 6.

H RESULTS ON CAIT

In addition to DeiT (Touvron et al., 2021a), we perform additional experiments with CaiT (Touvron et al., 2021b) on ImageNet and find that while reusing CaiT-XS, LEGO offers about 52.6% savings in FLOPs and 46.1% savings in wall time over the CaiT-S training from scratch (see Figure 8).



(a) CaiT-XS→CaiT-S **(b)** CaiT-XS→CaiT-S

Figure 8: Results on CaiT. (a) accuracy vs. flops and (b) accuracy vs. wall time, for training CaiT-S. LEGO saves flops by 52.6% and wall time by 46.1% over training from scratch on ImageNet.

Table 7: GLUE performance of different LEGO models. All of the results are based on BERT-Base models with BERT-Small as the base model for LEGO optimization.

Method	SST-2 (Acc.)	MNLI (Acc.)	MRPC (Acc.)	CoLA (Acc.)	QNLI (Acc.)	QQP (Acc.)	STS-B (Acc.)	Average (Acc.)
BERT-Small (Scratch)	87.21	77.56	82.11	59.93	85.06	85.82	84.99	80.38
BERT-Base (LEGO Init)	88.15	77.62	82.53	60.70	85.79	86.65	85.83	81.04
BERT-Base (LEGO Init + Pretrain)	88.42	79.29	84.31	62.09	88.07	88.81	87.00	82.57
BERT-Base (Scratch)	88.19	78.43	85.78	62.09	87.06	87.18	86.99	82.25

This once again corroborates that our proposed LEGO is universally effective for different vision transformers irrespective of the architecture and sizes.

I GLUE PERFORMANCE WITH LEGO INITIALIZATION

Table 1 shows the downstream fine-tuning results on GLUE and SQuAD for pretrained BERT models using different baselines. The numerical results reflect that despite being trained significantly faster than scratch, BERT trained with our LEGO approach achieve very similar performance comparing to the baselines on both benchmarks. In this section, we perform an additional experiment by directly finetuning a BERT-Base model initialized by LEGO (from BERT-Small) without any pre-training and observe that LEGO initialized model can benefit downstream tasks compared to a BERT-Small model trained from scratch (1st row vs 2nd row). Table 7 summarizes the results.

J SCALABILITY TO LARGE MODELS WITH BILLION+ PARAMETERS

Our extensive experiments on BERT (Devlin et al., 2019), RoBERTa (Liu et al., 2019), GPT2 (Radford et al., 2019), DeiT (Touvron et al., 2021a) and CaiT (Touvron et al., 2021b) show that LEGO can consistently improve transformer training efficiency over the traditional way of training from scratch across domains and model sizes. One interesting future direction of our work is scaling LEGO to very large models with parameters more than 100B, such as GPT3 (Brown et al., 2020). While we currently do not possess the compute resources for this extreme large-scale study, we perform a preliminary experiment on GPT2-1.5B (Radford et al., 2019) by using GPT2-Large as the initialization. We train for 15k steps on C4 dataset (Raffel et al., 2020) and find that our proposed LEGO saves about 39% computation cost (FLOPs) of training GPT2-1.5B from scratch to reach the same log perplexity (which is 3.3). We believe that it is imperative to study the extent to which the benefits of LEGO remain at the scale on which the modern large language models are trained. We hope to cover this in our future work.