TripLe: Revisiting Pretrained Model Reuse and Progressive Learning for Efficient Vision Transformer Scaling and Searching

Cheng Fu[†,‡], Hanxian Huang[†], Zixuan Jiang[‡], Yun Ni[‡], Lifeng Nai[‡], Gang Wu[‡],
Liqun Cheng[‡], Yanqi Zhou[‡], Sheng Li[‡], Andrew Li[‡], Jishen Zhao[‡]
UC San Diego[†], Google [‡]
{cfu,hah008}@ucsd.edu,
{zixuan, yunn, lnai, wgang, liquncheng, yanqiz, lsheng, andrewyli, jishenzhao}@google.com

Abstract

One promising way to accelerate transformer training is to reuse small pretrained models to initialize the transformer, as their existing representation power facilitates faster model convergence. Previous works designed expansion operators to scale up pretrained models to the target model before training. Yet, model functionality is difficult to preserve when scaling a transformer in all dimensions at once. Moreover, maintaining the pretrained optimizer states for weights is critical for model scaling, whereas the new weights added during expansion lack these states in pretrained models. To address these issues, we propose TripLe, which partially scales a model before training, while growing the rest of the new parameters during training by copying both the warmed-up weights with the optimizer states from existing weights. As such, the new parameters introduced during training will obtain their training states. Furthermore, through serializing the model scaling, the functionality of each expansion can be preserved. We evaluate TripLe in both single-trial model scaling and multi-trial neural architecture search (NAS). Due to the fast training convergence of TripLe, the proxy accuracy from TripLe better reveals the model quality compared to from-scratch training in multi-trial NAS. Experiments show that TripLe outperforms from-scratch training and knowledge distillation (KD) in both training time and task performance. TripLe can also be combined with KD to achieve an even higher task accuracy. For NAS, the model obtained from TripLe outperforms DeiT-B in task accuracy with 69% reduction in parameter size and FLOPs.

1. Introduction

Vision transformer (ViT) models are promising to achieve the state-of-the-art performance on various computer vision tasks [15, 19, 42, 43]. While the performance frontier keeps pushing forward, the training costs also scale with the growth of parameters. For example, Dehghani et al. [12] scaled a ViT to 22 billion parameters. Furthermore, recent research [31] shows that transformers require much more training steps and larger datasets to better generalize compared to convolutional neural networks (CNNs), imposing even more scaling costs of ViTs.

Among various transformer training acceleration and cost reduction techniques [23, 8, 49, 54, 39], one promising method is to reuse small pretrained models to initialize a large model before training. By scaling a small pretrained model with various expansion operators and using it to initialize the large model, the implicit knowledge facilitates faster model convergence. Previous studies such as bert2BERT [5] focused on preserving the functionality of the small pretrained transformer, when growing transformer width (i.e., hidden dimensions). A recent work learn-to-grow [47] learns linear mappings to scale the pretrained model by minimizing the task loss during model expansion.

However, we identify a set of critical limitations in the existing approaches through a comprehensive investigation (Sec 2.3). (**L1**) Maintaining the functionality during model scaling is not the only key factor to achieving a high speedup and final task accuracy. We identify that the other critical factor is retaining the *optimizer states* of the weights, because that preserves the direction of model updates when the functionality is preserved. Previous methods [47, 5] do not include these optimizer states during model scaling. (**L2**) Training discrepancies exist between the pretrained weights and the new weights because the new weights introduced during scaling do not have optimizer states built up before training. (**L3**) The functionality of a pretrained model is hardly preserved by simply expanding a small pretrained model in multiple dimensions all at once.

To address these limitations, we propose a new method, TripLe[1], which *partially expands a pretrained model be-*

[1]TripLe incorporates **three** principles: reusing pretrained models, pro-

fore training and *grows the rest of the parameters during training*. TripLe conducts the expansion of model width and depth in a serialized fashion. Tackling **L1**, we scale the width of the pretrained transformer with their optimizer states to initialize the scaled model and optimizer. So the pretrained weights will maintain their update directions during training. After a short warmup training phase, the new weights will obtain their training states. To address **L2**, we increase the depth of ViTs by copying the existing weights parameters and their training states. As such, the optimizer states obtained from the first stage can be leveraged by the second expansion, mitigating the training mismatch between new and pretrained weights. For **L3**, when serializing the width and depth expansion, each expansion stage will mostly preserve the functionality of the small pretrained model, enabling faster convergence of the training period.

TripLe offers the best of the two worlds – *pretrained model reuse* and *progressive learning*. Our exploration shows that they are two extreme cases for transformer scaling: the pretrained model reuse technique will scale a small pretrained model in multiple dimensions toward the target model before training, while progressive learning starts from a randomly initialized model and grows parameters during training until reaching the target large model. We observe that these methods in fact benefit each other on ViT training: reusing a pretrained ViT facilitates faster convergence of progressive learning at each stage, while progressive learning constructs the training states that help the ViT scaling. To further improve model quality, we augment TripLe with knowledge distillation (KD) [20]. Specifically, TripLe applies the pretrained model to initialize a large model and KD uses the pretrained model to provide teaching signals during training.

Experiments and Results. We evaluate TripLe with both *single-trial model scaling* and *multi-trial neural architecture search* (NAS). With single-trial training, we scale various ViTs, and compare the training time and task accuracy against from-scratch training and a variety of baseline model scaling methods. When scaling pretrained ViTs size by 8×, TripLe saves the training time up to 71.0%∼80.9% compared to from-scratch training. Other model scaling methods hardly achieve performance neutrality against from-scratch training on large ViT models. Moreover, with the same training budget as from-scratch training, a 44MB ViT (expanded from a 5MB ViT) outperforms both the official 86MB DeiT-B (by 0.2%) and KD alone (by 1.0%) in ImageNet-1k task accuracy. Combining TripLe with KD, the 44MB model shows a 1.8% higher task accuracy compared to using KD alone.

TripLe enhances NAS performance by finding a 27MB model that outperforms the task accuracy of the 86MB DeiT-B [42] and the model searched by traditional multi-trial NAS. One of the significant downsides of multi-trial search is a long searching time [57], as sampled models are always trained from scratch. To reduce training time, prior approaches typically use ∼10% of the final training time to approximate model accuracy. Yet, this proxy accuracy is too far from the final accuracy. Our method can be viewed as a new form of 'weight-sharing' designated for multi-trial NAS. TripLe allows each trial to start from a pretrained model (Figure 2). The proxy accuracy obtained from TripLe shows a higher correlation with the final task accuracy compared to training from scratch. After the model is searched, we further improve the model performance using TripLe during model evaluation.

gressive learning, and knowledge distillation.

Table 1. ViT [42] for evaluating TripLe. New model variants S_{L24}/B_{L24} only change the #layers of DeiT-S/DeiT-B.

Model (DeiT-)	hidden dim	#heads	#layers	#params	FLOPs (billions)	Top-1† Acc	Top-1‡ Acc
Ti	192	3	12	5M	2.16	72.0	72.1
S	384	6	12	22M	8.50	79.5	79.8
B	768	12	12	86M	33.72	81.0	81.8
L	1024	16	24	307M	119.36	82.1	82.2
S_{L24}	384	6	24	44M	16.87	80.0	-
B_{L24}	768	12	24	172M	67.21	81.4	-

† Top-1 Accuracy of our DeiT re-implementation with 300 epochs of training.
‡ Official Top-1 task accuracy under 300 epochs of training.

2. Scaling Vision Transformers

In this section, we introduce the Vision Transformers (ViTs) studied in this paper. Furthermore, we perform a comprehensive investigation on the performance of existing scaling operators.

2.1. Vision Transformer

Transformer [14] was first employed in vision tasks by Dosovitskiy et al. [15]. ViT first extracts features from raw image patches using a CNN and feeds the extracted features as the input to the transformer. DeiT [42] finds that the model can achieve a high task accuracy on ImageNet-1k [13] dataset when applying strong image augmentation with knowledge distillation [1]. Many follow-up works propose ViT variants for better task accuracy [43, 19, 44, 56, 6].

This study focuses on scaling pretrained ViTs, and our experiments reuse the DeiT architectures [42]. To study scaling ViTs in both depth and width, we also introduce two model variants namely DeiT-B_{L24} and DeiT-S_{L24} given in Table 1.

2.2. Revisiting Operators Scaling

Recent studies [5, 47] reuse small pretrained transformers to accelerate the large model training. These works introduce different expansion operators for transformer depth (i.e., number of layers) and width (i.e., hidden dimension). They then employ the expanded pretrained model to initialize the target model. We denote the width/depth expansion operator as γ/β. The large model Θ to be trained has

L transformer layers with hidden dimension D. The pretrained model θ has l layers with hidden dimension d.

Layer Stacking β_{stck} and Interpolation β_{inpt}. Layer stacking [18] and interpolation [4] are two common approaches to increasing the transformer depth. Specifically, the operation can be formulated as follows:

$$\beta_{stck} : W_i = w_{i \mod l}, \ \forall i \in \{1, ..., L\} \quad (1)$$

$$\beta_{inpt} : W_i = w_{\lfloor i/k \rfloor}, \ \forall i \in \{1, ..., L\}, k = \lfloor L/l \rfloor \quad (2)$$

Here, W_i denotes the initial weight of the i-th transformer layer in model Θ. k is the expansion ratio of layers. By duplicating the existing layers according to Eq.1-2, we can scale the pretrained model in depth.

Adding Identity Layers β_{ST}. Shen et al.[38] propose to add identity layers W_I to maintain the functionality of the pretrained model. We denote this layer as W_I where $W_I(x) = x$. Specifically, each transformer layer can be viewed as two sub-layers:

$$\begin{aligned} x' &= x + \text{Attention}(LN(x)) \\ y &= x' + \text{FFN}(LN(x')) \end{aligned} \quad (3)$$

The 'Attention' denotes the multi-head attention layer. 'FFN' denotes the feed-forward layers following the attention layer [45]. 'LN' denotes the layer normalization. When initializing the scale and bias in 'LN', 'Attention', and 'FFN' to 0, the output of Attention$(LN(x))$ and FFN$(LN(x'))$ will be 0 as well. In this way, the transformer layer has y and x equal. β_{ST} can be combined with layer β_{stck} or β_{inpt}, i.e., where to add these identity layers. They are denoted as β_{STstck} and β_{STinpt}, respectively.

bert2BERT γ_{b2B}: Net2Net [7] increases the width of neural networks by duplicating neurons randomly and maintaining their output values through normalization. For transformers, it is first applied in bert2BERT [5] for pretrained transformer scaling (Detailed in Appendix B.2).

Padding Zeros γ_{pad0}: A straightforward way to increase the transformer width is to pad zeros to the existing weights. The small pretrained weights are on the upper-left corner of the large layer, the rest parameters are all zero-initialized.

Special padding zeros γ_{ST}: Staged-training [38] proposes a new width expansion method. When scaling a dense layer, the width expansion can be written as:

$$\gamma_{ST}(w) = \begin{pmatrix} w & z \\ z & w \end{pmatrix} \quad (4)$$

Here, z is a $d \times d$ zero matrix. The scaled matrix has a size of $D \times D$, where $D = 2d$. (Equations for other parameters are detailed in Appendix B.2.)

Weight resizing γ_{inpt}: In this work, we propose a new baseline operator interpolation γ_{inpt} that treats the weight matrices as images and interpolates the matrices using image resizing methods, such as bicubic / bilinear [11] and etc. Empirically, we find bilinear outperforms other methods, so we apply it in γ_{inpt}.

Table 2. Relationships between different methods and expansion operators for different transformer scaling methods. Model Interpolation is a new baseline proposed in this paper.

Method	Notation	width	depth
bert2BERT [5]	b2B	γ_{b2B}	β_{stck}
Staged-Training [38]	ST	γ_{ST}	β_{STinpt}
Model Interpolation	Inpt	γ_{inpt}	β_{inpt}
Learn-to-grow [47]	LTG	γ_{ltg}	β_{ltg}
Pad Zero	Pad0	γ_{pad0}	β_{stck}

Learn-to-grow γ_{ltg} / β_{ltg}: Besides all the above methods, learn-to-grow [47] proposes to learn linear matrices that map the pretrained weights into larger weight matrices to preserve the functionality of the small pretrained model (Equations in Appendix B.2.). We denote its width and depth expansion operator as γ_{ltg} and β_{ltg}, respectively.

The combination of these operators forms all the existing methods for scaling pretrained transformers. The summary is given in Table 2.

2.3. Investigating Expansion Operators

We motivate TripLe with a detailed investigation on performance and inefficiencies of previous scaling operators.

Investigation 1: Which operators can preserve model functionality? Many different expansion methods, such as γ_{b2B}, γ_{ST} and β_{ST} claim they are functionality preserving. We rebuild these baselines and their initialized accuracy is given in Table 3 (marked in blue).

① **bert2BERT** (γ_{b2B}): We find γ_{b2B} can preserve transformer functionality under constraint. When applying γ_{b2B} to scale a $d \times d$ dense layer into $2d \times 2d$, the original output vector o can become $\bar{o} = \{\frac{o}{2}, \frac{o}{2}\}$. After another non-linear function F, the output $F(\bar{o})$ can be recovered back to $F(o)$ through another linear function when F satisfies:

$$F(x) = F(x/n) \cdot n, \quad n \in R, x \in R \quad (5)$$

As GeLU in ViT doesn't satisfy Eq.5, the functionality cannot be preserved. When switching GeLU to ReLU in FFN, we find the ViT functionality can be fully maintained using γ_{b2B} (Appendix B.2).

② γ_{ST} **and** β_{ST}: We find the operator γ_{ST}/β_{ST} can preserve the functionality of the ViT. β_{ST} is an identity layer and it is functionality preserving as discussed in Sec 2.2. For γ_{ST}, when expanding a $d \times d$ dense layer into $2d \times 2d$, The new dense layer has output $\bar{o} = \{o, o\}$ where o is the original output with a size of $d \times 1$ according to Eq.4. Because γ_{ST} does not scale o as γ_{b2B}, the output results GeLU($\bar{o}$) can be recovered back to GeLU(o) after another linear mapping.

③ β_{stck} **and** β_{inpt}: Empirically, we also find β_{stck} and β_{inpt} can preserve partial functionality. For example, when expanding S→S$_{L24}$, the expanded model with β_{stck}/β_{inpt} can achieve 65.56%/39.98% task accuracy, respectively. This indicates the initial task loss is small after scaling the pretrained model using β_{stck} / β_{inpt}.

Table 3. Performance comparison between different expansion operators under 30/60 training epochs (ep_{30}/ep_{60}). '-m' denotes ignoring the optimizer states. '+m' means we scale the optimizer states and use them to initialize the new optimizer.

models	width γ	depth β	Test accuracy					Δ_{ep60}
			init	-m		+m		
				ep30	ep60	ep30	ep60	
Ti→S	γ_{b2B}	-	0.00	72.90	**76.53**	72.76	76.64	+0.11
Ti→S	γ_{ST}	-	71.82	72.82	75.72	**74.88**	**77.29**	**+1.57**
Ti→S	γ_{pad0}	-	0.01	69.89	70.83	69.85	72.45	**+1.63**
Ti→S	γ_{inpt}	-	0.00	**73.04**	76.50	73.25	76.11	-0.39
S→S_{L24}	-	β_{stck}	65.56	79.10	80.74	80.02	**81.31**	**+0.57**
S→S_{L24}	-	β_{inpt}	38.98	**80.00**	**81.23**	**80.34**	81.26	+0.03
S→S_{L24}	-	β_{STstck}	79.49	78.46	79.47	78.67	79.52	+0.05
S→S_{L24}	-	β_{STinpt}	79.49	78.61	79.23	78.41	79.26	+0.03
Ti→S_{L24}	γ_{ST}	β_{stck}	0.01	75.73	78.21	76.23	78.62	+0.40
Ti→S_{L24}	TripLe		71.82	-	-	**76.52**	**79.23**	-

④ **Learn-to-grow γ_{ltg} and β_{ltg}**: Learn-to-grow focuses on reducing the task loss L at the beginning of the training as given in Eq.6 through learning linear mappings (β_{ltg}, γ_{ltg}) from the small pretrained model to the large one.

$$\arg\min_{\beta_{ltg}\gamma_{ltg}} E_{x\sim Dt} L(x;\Theta), \quad s.t. \Theta = \beta_{ltg}(\gamma_{ltg}(\theta)) \tag{6}$$

Here, Dt represents the data distribution, and Θ is a large model expanded from a small checkpoint θ. When expanding S→B, learn-to-grow can only achieve 72% initial task accuracy [47] compared to 79.5% task accuracy of the small pretrained model. As such, we conclude that γ_{ltg}/β_{ltg} can only preserve partial functionality.

Other scaling operators discussed in Sec. 2.2 are not functionally preserving.

Investigation 2: Does the initial accuracy of the scaled model matter to the final task performance?

We observe that the correlation between initial accuracy of the scaled model and final accuracy is weak. Specifically, we evaluate each width/depth operator using ImageNet-1k dataset. (Hyperparameters in Appendix A). The results are given in Table 3 (marked the columns in purple).

Among width expansion operators, γ_{ST} achieves the highest initial accuracy, however, it cannot achieve the best final accuracy across the baselines. The initial accuracy obtained using γ_{ST} can be compromised within a few training iterations. Our new baseline γ_{inpt} (not functionality preserving) achieves very similar final task accuracy compared to other baselines. For depth expansion operator, γ_{STstck}/γ_{STinpt}, can maintain model functionality. Yet, the identity transformer layer (Eq.3) poses a considerable challenge for the model to be trained properly.

Investigation 3: Effect of optimizer states in scaling pretrained ViTs. ViTs are mostly trained using AdamW [32]. That means the optimizer states also exist in the pretrained model. Specifically, during the model update, we have:

$$m_t = \beta_1 m_{t-1} + (1-\beta_1) g_t \tag{7}$$

$$v_t = \beta_2 v_{t-1} + (1-\beta_2) g_t^2 \tag{8}$$

$$\theta_{t+1} = \theta_t - \frac{\lambda(t)}{\sqrt{v_t}+\epsilon} \cdot m_t \tag{9}$$

Here, g_t is the first-order gradient of the weight parameter θ at time step t. $\lambda(t)$ is the learning rate scheduler. β_1 and β_2 are constant decay rates (0.9, 0.999). m and v are AdamW states with the same dimensions as θ. Previous work, such as learn-to-grow or bert2BERT neglects this momentum information.

To investigate whether m and v help reduce the training time, we also apply the same width and depth operators γ/β on m and v during model initialization. The performance differences are listed in Table 3 (marked in red).

The results show that retaining optimizer states improve the model quality in general, especially when functionality of the pretrained model is preserved or at least partially preserved. With width expansion, the functionality preserving γ_{ST} with momentum information outperforms the second-best baseline by 0.65%. With depth expansion, β_{stck} shows a 0.57% accuracy increase with momentum information. This is because functionality preserving reduces the initial task loss and maintains the gradient g_t in Eq.7-8 for pretrained weights. Together with m_{t-1} and v_{t-1}, the direction of pretrained weights update $\frac{m_t}{\sqrt{v_t}+\epsilon}$ is preserved. One exception is β_{ST}, because g_t in the growing layers changes due to zero-initialized LN and bias (Eq.3).

Investigation 4: Can we perfectly scale Ti to S_{L24} by combining the best scaling operators among γ and β? We expand the model from Ti→S_{L24} using the best-performed operator selected from the previous investigation, namely, γ_{ST} and β_{stck} (Table 3). Ti→S_{L24} shows a 2.7% accuracy drop compared to S→S_{L24}. This is due to the limitations of simply combining γ_{ST} and β_{stck}. (1) The functionality-preserving feature of γ_{ST} and β_{stck} is compromised, when combining them together. This incurs the advantage of applying optimizer states. (2) For Ti→S_{L24}, the weights grow from 5MB to 44MB. Among them, 24MB of the weights are *zeros* introduced during model expansion. These zero values have no training states in the pretrained model. In addition, zero values are under-trained. As such, a training discrepancy exists between the zeros and the pretrained parameters. The issue exacerbates once weight decay is applied, because the zero values will not be penalized at the beginning of training.

3. Method

Overview. We propose a new method to mitigate the aforementioned training mismatches by serializing the model expansion. As discussed above, simply combining the best-performed scaling operators β_{stck} and γ_{ST} together is sub-optimal. We propose TripLe that *scales width before training* and *grows model depth during training* (Figure 1(a)). TripLe maintains the functionality preserving feature of γ_{ST} and β_{stck} by serializing the scaling operations. Furthermore, with a short warmup training after conducting γ_{ST}, zero weights will obtain their training states that ben-

efit the subsequent depth expansion. In what follows, we discuss each step in detail.

Scaling Width Before Training. TripLe applies γ_{ST} to expand transformer width before the training. We extend γ_{ST} to the scenario where the expanded model width is not divisible by the small model width (Eq.10). This method is for more general model scaling (e.g., NAS).

$$\gamma_{ST}(w) = k \begin{Bmatrix} \begin{bmatrix} w & \dots & z & z_s \\ \vdots & \ddots & \vdots & \vdots \\ z & \dots & w & z_s \\ z_s^T & \dots & z_s^T & w_s \end{bmatrix}, k = \lfloor D/d \rfloor \end{Bmatrix} \quad (10)$$

Here, w is the $d \times d$ pretrained dense layer. w_s is downsampled from w with a size of $ds \times ds$, $ds = D \mod d$. z and z_s are zero matrices with different sizes. The scaled layer $\gamma_{ST}(w)$ has a size of $D \times D$. For parameters b with a dimension of $1 \times d$ in LN, weight bias, and classification token, we can conduct a similar procedure using Eq.11.

$$\gamma_{ST}(b) = (\overbrace{b \quad \dots \quad b}^{k} \quad b_s \;), k = \lfloor D/d \rfloor \quad (11)$$

Here, '$(\cdot)$' is the concatenation operation, $\gamma_{ST}(b)$ has a size of $1 \times D$. As m and v have the same dimension as their weight parameters, we apply the same operators (Eq.10-11) on m/v and use $\gamma_{ST}(m)$/$\gamma_{ST}(v)$ to initialize the AdamW optimizer. For the CNN in ViT with a dimension of $(d, \text{channel}, r, s)$, we flatten it along kernel dimension (d) and apply Eq.11. The CNN after scaling will be reshaped back to $(D, \text{channel}, r, s)$.

Empirically, we find this method can maintain partial functionality at the beginning of the training when D is not divisible by d.

Growing Model Depth During Training. As shown in Figure 1, before growing the depth of the model, we conduct warm-up training by keeping the pretrained model depth (Stage I). After Stage I, we directly copy-paste the weight parameters and optimizer states from the bottom transformer layers to the top layers. The number of layers will grow from l to $L = 2l$. In this stage (Stage II), we freeze the bottom l layers including the positional encoding and convolution layer. Lastly (Stage III), we unfreeze the bottom layer and train all the weights together.

The key difference between our approach and progressive learning[49] is that: (1) besides the weight parameters, we also copy the momentum information which is effective in speedup the training (Sec 2.3) (2) Traditional progressive learning requires a long training time for each stage to converge. However, when reusing a pretrained model, each stage can converge in a very short amount of time.

TripLe without Depth Expansion. When the depth of the small pretrained model and the target model is the same (l), we cannot copy-paste the warmed-up parameters from the bottom layer i ($i \in \{1, .., \frac{l}{2}\}$) to the top layer j ($j \in \{\frac{l}{2}+1, .., l\}$), as the top layers have already been initialized with pretrained weights. As discussed in Sec 2.3, zero weights (z) are under-trained compared to pretrained weight (w) after conducting γ_{ST} (Eq.10). As such, we warm up the bottom layers and the zeros at the top layers in Stage I (Figure 1(b)). Next in Stage II, we train only the matrices that are zero-initialized in the bottom layers (position z/z_s in Eq.10) and all the parameters in the top layers. In this way, zero weights have more training steps compared to the pretrained weights during warm-up stages, mitigating the problem that zeros are under-trained (Sec 2.3). Also, the training states for zero weights are established before training all parameters together. Empirically, we find this method improves the training speed and can achieve better task performance when scaling width only.

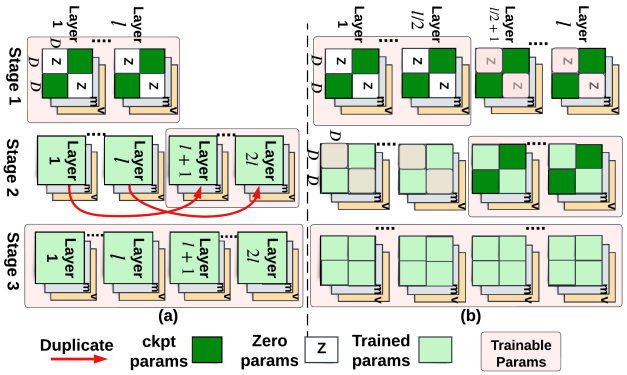


Figure 1. Training stages in TripLe when the model depth (l) (a) scales by $2\times$ and (b) is not scaling. We simplify the transformer layer into a single MLP. All the dense layers in the transformer layer are operated in the same fashion.

Combining TripLe with Knowledge Distillation. We also find *reusing pretrained models* methods can be combined with *Knowledge Distillation* (KD) to further improve the model performance. For KD, the pretrained models are employed to provide training signals; for methods in reusing pretrained models, the pretrained weights are used to initialize the large model. Based on our knowledge, we are the first work to combine them together.

We follow the KD method introduced in DeiT [42] and use the 'hard-label distillation' loss during training. Since our architectures are the same as DeiT, the integration is straightforward (Detailed in Appendix C.2). To make a fair comparison with the previous methods, we do not add KD during training unless specified.

4. TripLe for Multi-trial NAS

As one of use cases of model scaling is to enhance multi-trial NAS, we design a ViT search space and evaluate TripLe against traditional approaches as shown in Figure 2.

Traditional multi-trial NAS adopts an agent to sample a model and training hyperparameters from the search space. A worker starts training the model from scratch based on this selection. Different from traditional NAS, the worker

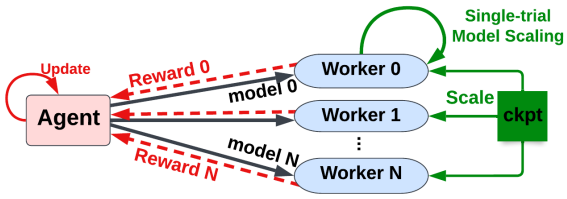


Figure 2. Leveraging TripLe in multi-trial NAS. The green blocks and arrows are key differences compared to traditional multi-trial NAS, while 'ckpt' denotes the small pretrained model.

Table 4. ViT search space for evaluating TripLe in NAS. 'Global' means all the layers are set to the same sampled parameter. 'Local' means the parameters are sampled for each layer.

Parameter name	Selections	Global/Local
hidden dimension (D)	[192, 384, 576, 768]	Global
Head factor (h_f)	[32, 64]	Local
FFN Expansion factor (e_f)	[2, 3, 4]	Local
Transformer layers (L)	[12, 13,..., 24]	Global
Learning rate (λ)	[5e-4, 1e-3, 4e-3]	Global
Weight decay	[0.05, 0.02, 0]	Global

in our approach will start training from a pretrained model leveraging TripLe. This section describes our search space, searching algorithm, and reward function.

Multi-trial Search Space. We build a neural architecture search space based on ViT architecture (Table 4). We search head factors h_f of each layer, hidden dimensions D, number of layers, and FFN expansion ratio e_f. The number of heads of each layer is D/h_f. The dimension of the FFN layer is $D \times e_f$. D, h_f and e_f are sampled independently. Furthermore, we search the learning rate and weight decay, which cannot be explored using one-shot algorithms. The cardinality of our search space is around 9.4e11.

Searching Algorithm and Reward function. We apply a regularized evolution algorithm [36] as the controller algorithm to optimize the search space of NAS. We do not choose to employ PPO method [37, 40, 57, 41] which requires a long training time for the agent to converge. We adopt the TuNAS reward [2] and our reward is defined as

$$Reward(\Theta) = Q(\Theta) + \epsilon|\frac{FLOPs(\Theta)}{FLOPs_0} - 1| \qquad (12)$$

Here, $Q(\cdot)$ indicates the quality (accuracy) of a candidate architecture Θ, $FLOPs(\Theta)$ is its FLOPs, $FLOPs_0$ is a problem-dependent FLOPs target, and hyperparameter $\epsilon < 0$ is the cost exponent.

5. Evaluation

We evaluate the performance of TripLe in both single-trial model scaling and multi-trial NAS.

Dataset and Hyperparameters. We evaluate TripLe using ImageNet-1k [13] for training ViTs. The ViT architectures are given in Table 1. We transfer the models trained using TripLe to various downstream tasks which include CIFAR10 [27], CIFAR100 [27], Flowers102 [33], Stanford-Cars [26] (results of given in Appendix F). All the experiments are done on dragonfish TPUs [24] with 8×8 topology. The validation/test sets are evaluated every 400 seconds on separate TPUs.

Table 5. Performance comparison between different model scaling methods. ep_{30} denotes the total training time is set to 30 epochs.

Model	Max ↓ Time	Method	Max ↓ FLOPs	Top-1 Test accuracy (%)				
				ep_{30}	ep_{60}	ep_{90}	ep_{120}	ep_{300}
B	0%	Scratch	0%	-	-	-	-	81.03
B	28.9%	MLST	36.7%	-	-	-	-	81.27
S→B	52.0%	LTG	55.4%	-	-	-	-	81.57
S→B	67.7%	b2B	68.2%	78.11	80.39	81.45	81.82	81.81
S→B	74.2%	Inpt	74.8%	78.98	80.89	81.80	81.75	81.75
S→B	78.6%	ST	80.0%	79.33	81.29	81.99	82.10	81.92
S→B	**81.4%**	TripLe	**82.1%**	**79.72**	**81.32**	**82.10**	**82.36**	**82.01**
S	0%	Scratch	0%	-	-	-	-	79.50
S	×	MLST	×	-	-	-	-	75.97
Ti→S	12.0%	b2B	12.2%	72.76	76.64	78.01	79.01	80.33
Ti→S	11.2%	Inpt	11.5%	73.25	76.11	77.78	78.59	80.54
Ti→S	12.0%	ST	13.8%	**74.88**	77.29	78.46	79.25	80.67
Ti→S	**17.4%**	TripLe	**18.4%**	74.67	**77.53**	**78.54**	**79.34**	**80.88**

We set the batch size to 4096, so the learning rate would be $\frac{batchsize}{512} * 0.0005 = 0.004$ according to the DeiT paper. Other hyperparameters are the same as DeiT [42] (Detailed in Appendix A.1). Our baseline methods are given in Table 3. For ST and Inpt, we also initialize the scaled model with optimizer states for a fair comparison. Specifically, we conduct the width/depth expansion on m and v using the same operators to expand the weights. MLST [49] is a progressive learning baseline for transformer training.

We set the training time t for each model scaling task to 30/60/90/120/300 epochs (denoted as ep_t), respectively. The final learning rates under different training times are always 0. The warm-up epochs are set to 5. For TripLe, the depth growth happens during the warm-up phase and Stage1/Stage2 takes 2.5/2.5 epochs, respectively.

5.1. Evaluation of Single-trial Models Scaling

Metrics. For evaluating the model quality, we report the Top-1 test accuracy on ImageNet-1k.

To measure the training cost for each method, we scan the minimum time required to match the *validation loss* of from-scratch training. And then, we use it to compute the maximum wall-time reduction **(Max ↓ Time)** and maximum training FLOPs reduction **(Max ↓ FLOPs)** for each method accordingly. When the method is unable to achieve the validation loss of from-scratch training under any settings, we report '×' in the corresponding table entry.

Evaluation on Expanding Width Only. We first perform width scaling only to evaluate our method given in Figure 1(b). As is shown in Table 5, TripLe outperforms other baselines in max training time reduction. For Ti→S/S→B, TripLe can save the 17.4%/82.1% maximum training time compared to 12.0%/78.6% of ST (the second-best baseline).

Besides, TripLe can achieve better task accuracy compared to baselines generally. When using 300 epochs for training, TripLe can outperform from-scratch training accuracy by 1.38%/0.98% for Ti→S/S→B. This indicates that

training more steps on the zero weights introduced by ST can mitigate the training mismatch between zero weights and pretrained weights. For learn-to-grow, the implementation is not open-source; so we report the max ↓ Time/FLOPs using the number reported in the paper.

Expanding Width and Depth Together. When expanding width and depth together, we apply our method given in Figure 1(a) that serializes the expansion of ViTs. The model will grow $8\times$ in parameter size. When choosing 40% (ep_{120}) of the total training time (300 epochs), the model obtained from TripLe outperforms ST by 0.88%/0.27%/0.69% and scratch training by 1.03%/0.99%/0.09% in task accuracy. This shows that serializing expansion operators can obtain better task accuracy under the same training budget. Besides the saving training cost, TripLe can also be employed to train ViT for better task accuracy compared to training from scratch.

Also, using progressive learning alone (i.e., MLST) cannot reach task performance of scratch training for B_{L24}/L under ep_{300}. The existing weights cannot be fully trained before expanding to a larger model, resulting in performance degradation.

Comparison and Combination of TripLe with Knowledge Distillation. As discussed in Sec 3, TripLe is orthogonal to KD which is another widely used technique to improve the transformer quality. In this subsection, we compare and combine TripLe with KD. For the teacher model in KD, we use a ResNet-101 with 79.33% test accuracy. The ResNet-101 must be trained using the same data augmentation techniques as DeiT.

The learning curves of TripLe and KD are given Figure 4. We observe that using TripLe can outperform the model trained using KD. For Ti→S_{L24}, the model achieves 82.0% test accuracy compared to 81.0% obtained from KD. When combining TripLe with KD, the model accuracy of S_{L24} can reach 82.8% test accuracy. This combination reveals that not only can we use the pretrained model to provide teaching signals in KD, but we can also use the small pretrained model to initialize the large model directly.

Sensitivity Analysis of TripLe. We gradually remove the design components from TripLe to validate their effects. The learning curves are given in Figure 3. When switching our depth expansion method to β_{stck} and conducting the expansion all at once before training (TripLe-copy), the performance gets worse and the model is overfitting in long-time training. This shows that conducting all the expansions before training incurs performance degradation. We further ignore the momentum information (TripLe-copy-m) during the model scaling and the results become even worse compared to the previous analysis. As such, maintaining the train states is critical for scaling pretrained models.

Sensitivity to Training Times. In Table 6, we present the task performance as a function of training time. Keeping the original training budget with model scaling methods incurs performance degradation. That is because scaling pretrained model achieves faster training convergence. In this scenario, training the large model for too long will result in model over-fitting. For training from scratch, the overfitting doesn't occur until the model is trained for 400 epochs [42]. As such, reducing the training time when scaling a pretrained model is necessary.

Table 6. Performance comparison between different model scaling methods. ep_{30} denotes the total training time is set to 30 epochs. Ti→S_{L24} denotes scaling DeiT-Ti to DeiT-S_{L24}. The pretrained model accuracy is given in Table 1.

models	Max ↓ Time	Method	Max ↓ FLOPs	Top-1 Test accuracy (%)				
				ep_{30}	ep_{60}	ep_{90}	ep_{120}	ep_{300}
S_{L24}	0%	Scratch	0%	-	-	-	-	79.97
S_{L24}	39.3%	MLST	41.6%	-	-	-	-	80.42
Ti→ S_{L24}	58.9%	b2B	60.1%	75.28	78.63	79.98	80.14	80.54
Ti→ S_{L24}	68.6%	Inpt	70.1%	75.72	78.93	80.40	80.64	81.12
Ti→ S_{L24}	67.9%	ST	70.0%	76.23	78.62	80.10	80.22	79.65
Ti→ S_{L24}	**71.0%**	TripLe	**72.1%**	**76.52**	**79.23**	**80.68**	**81.10**	**82.04**
B_{L24}	0%	Scratch	0%	-	-	-	-	**81.37**
B_{L24}	×	MLST	×	-	-	-	-	78.84
S→ B_{L24}	×	b2B	×	79.02	80.89	81.33	81.22	79.39
S→ B_{L24}	79.9%	Inpt	80.4%	80.40	82.29	**82.39**	82.23	80.28
S→ B_{L24}	79.9%	ST	80.4%	80.70	82.34	82.23	81.99	79.94
S→ B_{L24}	**80.9%**	TripLe	**81.7%**	**80.77**	**82.53**	82.36	**82.26**	80.60
L	0%	Scratch	0%	-	-	-	-	**82.12**
L	×	MLST	×	-	-	-	-	49.63
B→ L	×	b2B	×	78.02	81.64	81.71	81.55	80.01
L	×	Inpt	×	81.23	81.55	81.28	81.20	80.59
B→ L	×	ST	×	80.88	81.45	81.65	81.50	79.84
B→ L	**73.3%**	TripLe	**74.7%**	**81.23**	**82.04**	**82.22**	**82.19**	81.40

Table 7. Kendall-tau correlation between different methods across 15 trials.

Method	Scratch_{ep30}	TripLe_{ep120}	Scratch_{ep300}
Scratch_{ep300}	0.221	**0.318**	-
TripLe_{ep120}	0.789	**0.865**	0.363

5.2. Evaluation of TripLe on Multi-trial Search

We intend to answer two questions in this section: (1) *Does the proxy accuracy obtained from TripLe show a higher correlation to the final task accuracy?* (2) *Can TripLe find better models compared to multi-trial search?*

The pretrained model we reuse for the sampled models is DeiT-Ti. Our searching method is implemented inside the symbolic programming library named PyGlove [34].

Ranking Score Comparison. We randomize 15 models from our search space (Table 4) and evaluate the Kendall-tau [25] correlation between the proxy accuracy and the task accuracy. Specifically, the models are trained: (1) from scratch for 30 epochs. (Scratch_{ep30}) (2) from the pre-trained model for 30 epochs. (TripLe_{ep30}) (3) from scratch for 300 epochs (Scratch_{ep300}). (4) from pretrained model for 120 epochs (TripLe_{ep120}). The results are given in Table 7.

When using 10% of the total training time (i.e., 300 epochs) for each trial, traditional multi-trial search only shows 0.221 Kendall-tau correlation. This indicates that correlation between the proxy accuracy (scratch_{30}) and the final training accuracy (scratch_{300}) is weak. On the other hand, TripLe shows a higher correlation to the final model

Figure 3. Sensitivity analysis of TripLe for (a) Ti→S$_{L24}$ (b) S→B$_{L24}$ (c) B→L under ep$_{30}$/ep$_{60}$/ep$_{120}$/ep$_{300}$. '-copy' denotes scaling both width and depth together before training. '-m' denotes ignoring momentum information from the pretrained model.

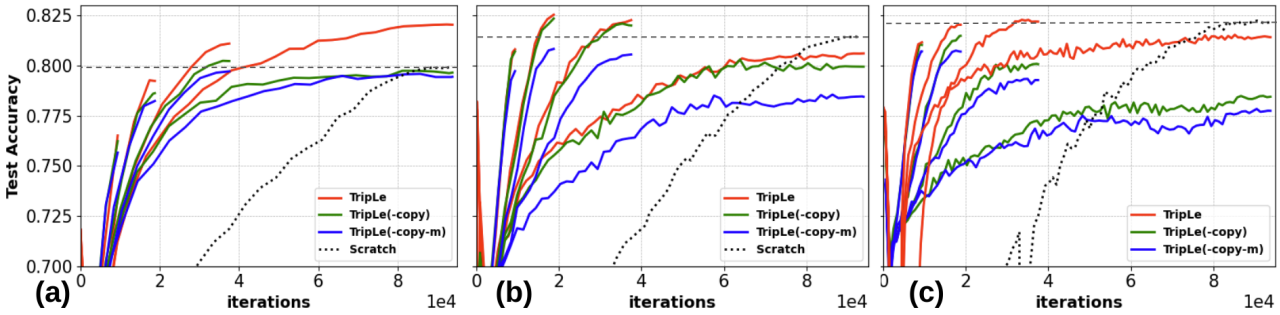
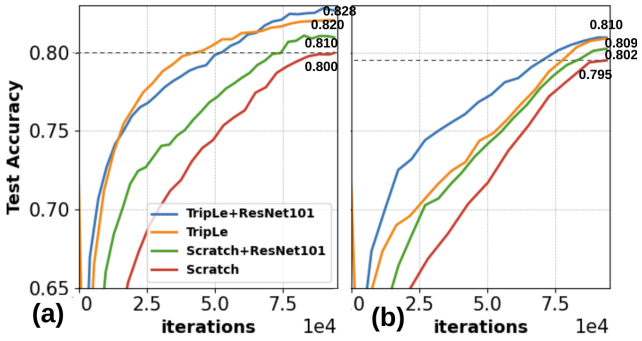


Figure 4. Comparison of TripLe with knowledge distillation under 300 epochs of training for (a) Ti→S$_{L24}$ and (b) Ti→S.



performance trained under TripLe$_{120}$ and scratch$_{300}$.

We also evaluate the correlation between scratch$_{300}$ and TripLe$_{120}$, it shows a 0.363 Kendall-tau correction. This can be interpreted as the model that is suitable for scratch training may not fit for TripLe. Also, it can come from random seed selection [51] in the scratch training. For TripLe, the initialization weights are fixed.

Searched Model Comparison. We conduct 200 trials for both multi-trial NAS and NAS with TripLe (Learning curve in Appendix D). For each trial, we conduct 30 epochs of training using TripLe (TripLe$_{ep30}$) and scratch training (Scratch$_{ep30}$). We set the FLOPs target ($FLOPs_0$) to 10000M and other hyperparameters for the regularized evolutionary and our reward function are given in Appendix A.2. As shown in Table 8, the model (ViT-TripLe) searched using NAS with TripLe can obtain 81.1% accuracy. ViT-TripLe outperform our re-implement 86MB DeiT-B in task accuracy with 69%/69% reduction in parameter size and inference FLOPs. On the other hand, the model searched by traditional NAS can achieve 80.8% task accuracy (Detailed architectures in Appendix F).

6. Related Work

Reusing Pretrained Model and Progressive Learning. Methods that reuse pretrained models assume the small pretrained model pre-exists before starting the training. The smaller model will be scaled up to initialize the large model [5, 47]. For progressive learning, these methods assume the small pretrained model does not exist. The models are initialized randomly and grow towards the target model during training [18, 49, 38, 28, 16]. In this work, we assume the pretrained model exists before training and also employ progressive learning to grow the model during training. We compare both lines of work in Sec 5.

Table 8. Comparison results of using TripLe in multi-trial NAS and traditional multi-trial NAS.

	Search method	Evaluation method	Params (MB)	FLOPs (Million)	Test Acc (%)
DeiT-S (ours)	-	Scratch$_{ep300}$	22	8495	79.5
DeiT-B (ours)	-	Scratch$_{ep300}$	86	33722	81.0
ViT-scratch	Scratch$_{ep30}$	TripLe$_{ep120}$	30	11409	79.5
ViT-scratch	Scratch$_{ep30}$	TripLe$_{ep300}$	30	11409	80.8
ViT-TripLe	TripLe$_{ep30}$	TripLe$_{ep120}$	27	10416	79.7
ViT-TripLe	TripLe$_{ep30}$	TripLe$_{ep300}$	27	10416	81.1

Efficient Transformer Learning. Besides, existing methods for the efficient transformer training techniques, such as pipeline parallelism [39, 23], large batch optimization [50], and layer dropping [54] are orthogonal to TripLe and works in reusing pretrained model. Some of the techniques are designed for NLP tasks, such as Electra [9] or token dropping [22], which cannot be directly applied in ViTs training. Knowledge distillation can also improve the quality and reducing training time [35, 42] . In this work, we compare and combine our approach with KD.

Neural Architecture Search. Recent advances in one-shot NAS leverage the idea of weight-sharing and train a supernetwork that contains all the possible model selections [48, 10, 30, 46, 17, 2]. For multi-trial NAS [57, 40, 41, 29], a controller samples candidate architectures and each one is trained from scratch. One shot is way faster than multi-trial method. However, one-shot cannot search training recipes and activation functions [10]. Also, one-shot incurs regularization conflict [17] that can hardly be resolved. In this work, we leverage TripLe to improve the performance of multi-trial NAS.

7. Conclusions

We propose TripLe, a method for scaling pretrained ViT to reduce the training time and improve task performance. Naïvely scaling the ViT once in multiple dimensions can

hardly preserve the functionality of the pretrained model. Besides, the new parameters introduced during scaling are under-trained and do not have their training states established. As such, TripLe scales the width of the model and optimizer states before training. During training, TripLe grows the depth by copying the warmed-up weights and optimizer states from existing layers. In this way, each expansion can mostly preserve functionality and the new weights in depth expansion can also obtain their training states from the previous expansion stage.

In single-trial model scaling, TripLe not only reduces the training time of scaling ViTs but also achieves even better task accuracy compared to the baseline methods. In multi-trial NAS, the proxy accuracy obtained from TripLe shows a higher correlation to their final performance. Besides, the searched model with TripLe outperforms the counterpart obtained using traditional NAS in task accuracy.

References

[1] Samira Abnar, Mostafa Dehghani, and Willem Zuidema. Transferring inductive biases through knowledge distillation. *arXiv preprint arXiv:2006.00555*, 2020. 2

[2] 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 *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 14323–14332, 2020. 6, 8

[3] Maxim Berman, Hervé Jégou, Andrea Vedaldi, Iasonas Kokkinos, and Matthijs Douze. Multigrain: a unified image embedding for classes and instances. *arXiv preprint arXiv:1902.05509*, 2019. 12

[4] 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. 3

[5] 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*, 2021. 1, 2, 3, 8

[6] Chun-Fu Richard Chen, Quanfu Fan, and Rameswar Panda. Crossvit: Cross-attention multi-scale vision transformer for image classification. In *Proceedings of the IEEE/CVF international conference on computer vision*, pages 357–366, 2021. 2

[7] Tianqi Chen, Ian Goodfellow, and Jonathon Shlens. Net2net: Accelerating learning via knowledge transfer. *arXiv preprint arXiv:1511.05641*, 2015. 3

[8] Wuyang Chen, Wei Huang, Xianzhi Du, Xiaodan Song, Zhangyang Wang, and Denny Zhou. Auto-scaling vision transformers without training. In *International Conference on Learning Representations*, 2022. 1

[9] Kevin Clark, Minh-Thang Luong, Quoc V Le, and Christopher D Manning. Electra: Pre-training text encoders as discriminators rather than generators. *arXiv preprint arXiv:2003.10555*, 2020. 8

[10] Xiaoliang Dai, Alvin Wan, Peizhao Zhang, Bichen Wu, Zijian He, Zhen Wei, Kan Chen, Yuandong Tian, Matthew Yu, Peter Vajda, et al. Fbnetv3: Joint architecture-recipe search using predictor pretraining. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 16276–16285, 2021. 8

[11] Philip J Davis. *Interpolation and approximation*. Courier Corporation, 1975. 3

[12] Mostafa Dehghani, Josip Djolonga, Basil Mustafa, Piotr Padlewski, Jonathan Heek, Justin Gilmer, Andreas Steiner, Mathilde Caron, Robert Geirhos, Ibrahim Alabdulmohsin, et al. Scaling vision transformers to 22 billion parameters. *arXiv preprint arXiv:2302.05442*, 2023. 1

[13] 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*, pages 248–255. Ieee, 2009. 2, 6

[14] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: Pre-training of deep bidirectional transformers for language understanding. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*, pages 4171–4186, Minneapolis, Minnesota, June 2019. Association for Computational Linguistics. 2, 12

[15] Alexey Dosovitskiy, Lucas Beyer, Alexander Kolesnikov, Dirk Weissenborn, Xiaohua Zhai, Thomas Unterthiner, Mostafa Dehghani, Matthias Minderer, Georg Heigold, Sylvain Gelly, et al. An image is worth 16x16 words: Transformers for image recognition at scale. *arXiv preprint arXiv:2010.11929*, 2020. 1, 2

[16] 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. 8

[17] Chengyue Gong, Dilin Wang, Meng Li, Xinlei Chen, Zhicheng Yan, Yuandong Tian, Vikas Chandra, et al. Nasvit: Neural architecture search for efficient vision transformers with gradient conflict aware supernet training. In *International Conference on Learning Representations*, 2021. 8

[18] 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*, pages 2337–2346. PMLR, 2019. 3, 8

[19] Benjamin Graham, Alaaeldin El-Nouby, Hugo Touvron, Pierre Stock, Armand Joulin, Hervé Jégou, and Matthijs Douze. Levit: a vision transformer in convnet's clothing for faster inference. In *Proceedings of the IEEE/CVF international conference on computer vision*, pages 12259–12269, 2021. 1, 2

[20] Geoffrey Hinton, Oriol Vinyals, and Jeff Dean. Distilling the knowledge in a neural network. *arXiv preprint arXiv:1503.02531*, 2015. 2

[21] Elad Hoffer, Tal Ben-Nun, Itay Hubara, Niv Giladi, Torsten Hoefler, and Daniel Soudry. Augment your batch: Improving generalization through instance repetition. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 8129–8138, 2020. 12

[22] 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. 8

[23] 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. 1, 8

[24] Norman P Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, et al. In-datacenter performance analysis of a tensor processing unit. In *Proceedings of the 44th annual international symposium on computer architecture*, pages 1–12, 2017. 6

[25] Maurice G Kendall. A new measure of rank correlation. *Biometrika*, 30(1/2):81–93, 1938. 7

[26] Jonathan Krause, Michael Stark, Jia Deng, and Li Fei-Fei. 3d object representations for fine-grained categorization. In *4th International IEEE Workshop on 3D Representation and Recognition (3dRR-13)*, Sydney, Australia, 2013. 6

[27] Alex Krizhevsky, Geoffrey Hinton, et al. Learning multiple layers of features from tiny images. 2009. 6

[28] 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*, pages 12486–12496, 2022. 8

[29] Sheng Li, Mingxing Tan, Ruoming Pang, Andrew Li, Liqun Cheng, Quoc V Le, and Norman P Jouppi. Searching for fast model families on datacenter accelerators. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 8085–8095, 2021. 8

[30] Hanxiao Liu, Karen Simonyan, and Yiming Yang. DARTS: Differentiable architecture search. In *International Conference on Learning Representations*, 2019. 8

[31] Zhuang Liu, Hanzi Mao, Chao-Yuan Wu, Christoph Feichtenhofer, Trevor Darrell, and Saining Xie. A convnet for the 2020s. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 11976–11986, 2022. 1

[32] Ilya Loshchilov and Frank Hutter. Decoupled weight decay regularization. *arXiv preprint arXiv:1711.05101*, 2017. 4

[33] 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*, pages 722–729. IEEE, 2008. 6

[34] Daiyi Peng, Xuanyi Dong, Esteban Real, Mingxing Tan, Yifeng Lu, Gabriel Bender, Hanxiao Liu, Adam Kraft, Chen Liang, and Quoc Le. Pyglove: Symbolic programming for automated machine learning. In *Advances in Neural Information Processing Systems (NeurIPS)*, volume 33, pages 96–108, 2020. 7

[35] 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. 8

[36] Esteban Real, Alok Aggarwal, Yanping Huang, and Quoc V Le. Regularized evolution for image classifier architecture search. In *Proceedings of the aaai conference on artificial intelligence*, volume 33, pages 4780–4789, 2019. 6, 12

[37] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*, 2017. 6

[38] Sheng Shen, Pete Walsh, Kurt Keutzer, Jesse Dodge, Matthew Peters, and Iz Beltagy. Staged training for transformer language models. In *International Conference on Machine Learning*, pages 19893–19908. PMLR, 2022. 3, 8

[39] 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. 1, 8

[40] Mingxing Tan, Bo Chen, Ruoming Pang, Vijay Vasudevan, Mark Sandler, Andrew Howard, and Quoc V Le. Mnasnet: Platform-aware neural architecture search for mobile. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 2820–2828, 2019. 6, 8

[41] Mingxing Tan and Quoc Le. Efficientnet: Rethinking model scaling for convolutional neural networks. In *International Conference on Machine Learning*, pages 6105–6114. PMLR, 2019. 6, 8

[42] 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*, pages 10347–10357. PMLR, 2021. 1, 2, 5, 6, 7, 8

[43] Hugo Touvron, Matthieu Cord, and Hervé Jégou. Deit iii: Revenge of the vit. In *Computer Vision–ECCV 2022: 17th European Conference, Tel Aviv, Israel, October 23–27, 2022, Proceedings, Part XXIV*, pages 516–533. Springer, 2022. 1, 2

[44] 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*, pages 32–42, 2021. 2

[45] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. *Advances in neural information processing systems*, 30, 2017. 3

[46] Alvin Wan, Xiaoliang Dai, Peizhao Zhang, Zijian He, Yuandong Tian, Saining Xie, Bichen Wu, Matthew Yu, Tao Xu, Kan Chen, Peter Vajda, and Joseph E. Gonzalez. Fbnetv2: Differentiable neural architecture search for spatial and channel dimensions. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2020. 8

[47] Peihao Wang, Rameswar Panda, Lucas Torroba Hennigen, Philip Greengard, Leonid Karlinsky, Rogerio Feris, David Daniel Cox, Zhangyang Wang, and Yoon Kim. Learning to grow pretrained models for efficient transformer train-

ing. In *International Conference on Learning Representations*, 2023. 1, 2, 3, 4, 8, 12

[48] Bichen Wu, Xiaoliang Dai, Peizhao Zhang, Yanghan Wang, Fei Sun, Yiming Wu, Yuandong Tian, Peter Vajda, Yangqing Jia, and Kurt Keutzer. Fbnet: Hardware-aware efficient convnet design via differentiable neural architecture search. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 10734–10742, 2019. 8

[49] 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. 1, 5, 6, 8

[50] 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. 8

[51] Kaicheng Yu, René Ranftl, and Mathieu Salzmann. An analysis of super-net heuristics in weight-sharing nas. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 44(11):8110–8124, 2021. 8

[52] Sangdoo Yun, Dongyoon Han, Seong Joon Oh, Sanghyuk Chun, Junsuk Choe, and Youngjoon Yoo. Cutmix: Regularization strategy to train strong classifiers with localizable features. In *Proceedings of the IEEE/CVF international conference on computer vision*, pages 6023–6032, 2019. 13

[53] Hongyi Zhang, Moustapha Cisse, Yann N Dauphin, and David Lopez-Paz. mixup: Beyond empirical risk minimization. *arXiv preprint arXiv:1710.09412*, 2017. 13

[54] 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. 1, 8

[55] Zhun Zhong, Liang Zheng, Guoliang Kang, Shaozi Li, and Yi Yang. Random erasing data augmentation. In *Proceedings of the AAAI conference on artificial intelligence*, volume 34, pages 13001–13008, 2020. 12

[56] Daquan Zhou, Bingyi Kang, Xiaojie Jin, Linjie Yang, Xiaochen Lian, Zihang Jiang, Qibin Hou, and Jiashi Feng. Deepvit: Towards deeper vision transformer. *arXiv preprint arXiv:2103.11886*, 2021. 2

[57] Barret Zoph and Quoc V Le. Neural architecture search with reinforcement learning. *arXiv preprint arXiv:1611.01578*, 2016. 2, 6, 8

A. Training Hyperparameters

A.1. Hyperparameters for Single-trial Model Scaling.

Our training hyperparameters are the same as DeiT-B [14] as given in Table 9. We find using repeat augmentation [3, 21] and erasing augmentation [55] doesn't show any performance improvement. As such, we do not use them in the training phase.

A.2. Hyperparameters for NAS

The regularized evolution algorithm discussed in Sec 4 identical as AmoebaNet [36]. We set the population size set to 50 and the tournament size set to 10. The mutation probabilities are uniform and are identical to [36]. For the reward function, exponent $\epsilon = -0.07$, FLOPs target is set to $FLOPs_0 = 10000M$.

B. Details of Baseline Scaling Operators

B.1. Learning Curve of Scaling Operators

The learning curve for different expansion operators is given in Figure 6. The γ_{ST} with momentum information outperforms other baselines.

B.2. Details Explanations for bert2BERT and Learn-to-share

bert2BERT (γ_{b2B}): We use a simple example to illustrate the key idea of bert2BERT here. Assuming we are expanding the first layer w_0 and the input feature vector is d_{in}, the output of the matrix would be $d_o = d_{in}^T w_0$. w_0 has a dimension of 2×2 and d_{in} has a dimension of 2×1. After layer scaling, w_0'' has a size of 4×4.

$$d_{in} = \begin{bmatrix} a \\ b \end{bmatrix}, \quad w_0 = \begin{bmatrix} o & p \\ q & r \end{bmatrix}, \quad d_o^T = d_{in}^T w_0 = \begin{bmatrix} a_o \\ b_o \end{bmatrix} \tag{13}$$

When expanding the weight matrix w_0 given in Eq. 13 from a 2×2 matrix into a 4×4 matrix, we first expand the input dimensions.

We randomly select two rows, e.g., the first row, and duplicate them. Then, we normalize these rows based on the number of duplication. The corresponding input features will be duplicated in the same fashion without normalization. The result dense layers are given as follow:

$$d'_{in} = \begin{bmatrix} a \\ b \\ a \\ a \end{bmatrix}, \quad w'_0 = \begin{bmatrix} \frac{o}{3} & \frac{p}{3} \\ q & r \\ \frac{o}{3} & \frac{p}{3} \\ \frac{o}{3} & \frac{p}{3} \end{bmatrix} \tag{14}$$

As is shown above, the result ${d'_{in}}^T w'_0 = d_{in}^T w_0$ does not change during the expansion.

Next, we randomly select two columns, e.g., the second column, and duplicate the it without normalization.

$$d''_{in} = \begin{bmatrix} a \\ b \\ a \\ a \end{bmatrix}, \quad w''_0 = \begin{bmatrix} \frac{o}{3} & \frac{p}{3} & \frac{p}{3} & \frac{p}{3} \\ q & r & r & r \\ \frac{o}{3} & \frac{p}{3} & \frac{p}{3} & \frac{p}{3} \\ \frac{o}{3} & \frac{p}{3} & \frac{p}{3} & \frac{p}{3} \end{bmatrix} \tag{15}$$

The final output ($o'' = {d''_{in}}^T w''_0$) would be $o'' = [a_o, b_o, b_o, b_o]$. For the following layer w_1, the input is determined and thus the policy of row duplication is determined as well. For w_1, we continue the same procedure for expanding column (i.e., random select columns and duplicate them). And so on, the model functionality can be preserved.

$$LayerNorm(o) = \frac{(o'' - \mu_o)}{\sigma_o} \odot W^{LN} + b^{LN} \tag{16}$$

However, if the next layer is LayerNorm (Eq 16). The mean (μ_o) and variance (σ_o) of the output o changes. $\odot$ denotes the element-wise multiplication. During expansion, we don't know the relationship between a_o and b_o, so bert2BERT cannot preserve functionality through changing the LN scale and LN-bias, i.e. W^{LN} and b^{LN}.

On the other hand, γ_{ST} will yield output $o'' = [a_o, b_o, a_o, b_o]$. The mean μ_o and the variance σ_o of the output vector does not change.

Learn-to-grow. learn-to-grow [47] proposes to learn linear matrices that map the pretrained weights into larger weight matrices to preserve the functionality of the small pretrained model. We denote its width and depth expansion operator as γ_{ltg} and β_{ltg}, respectively.

$$W'_i = \gamma_{ltg}(w_i) = H_i w_i H_i^T, \quad i \in \{1, ..., l\} \tag{17}$$

Here, H_i ($D \times d$) is a trainable linear layer that maps the dense layer w_i into W'_i. w_i has a dimension of $d \times d$ and W'_i has a size of $D \times D$. For layer normalization and weight bias with a dimension of $d \times 1$, the expansion is similar to Eq 17 [47].

After width expansion, learn-to-share trains another set of linear mappings for depth expansion that expand W' into W:

$$W_i = \beta_{ltg}(w_i) = \Sigma_{j=1}^{l} P_{i,j} W'_j, \quad i \in \{1, ..., L\} \tag{18}$$

Here P_i is a $1 \times l$ vector. l is the number layers in the pretrained model; L is the number of layers in the scaled model. This means the expanded layer W_i is the weighted sum of W'_j where $j \in \{1, ..., l\}$,.

The linear mappings (H, P) are introduced to scale every dense layers in the scaled ViT. These mappings contain

Table 9. Hyperparameters for model scaling experiments. The hyperparameters are identical to DeiT-B. We find batch augmentation [?] and Erasing are not useful to increase the final task accuracy.

Search method	Search method	Learning rate decay	Warmup epoch	Label smoothing	Dropout	Drop path	Repeat Aug	Gradient clip	RandAug [?]	Mixup [53]	Cutmix [52]	Erasing [?]
4096	4e-3	cosine	5	0.1	0.0	0.1	×	×	✓	✓	✓	✓

a large number of parameters and requires a prohibitively expensive hardware memory for training. Some techniques are proposed in the paper to reduce the number of parameters, such as Kronecker factorization.

In this paper, we find the objective of training these linear mappings is the same as the training the scaled model (Eq 6). For S→B, learn-to-grow can achieve 72% initial accuracy. Specifically, learn-to-grow trains the linear mapping H, P for around 200 steps and scale the model according to Eq 17-18. However, using γ_{ST} alone to scale S→B can achieve the pretrained DeiT-S accuracy (79%) at step 0. γ_{Pad0} can achieve 73% accuracy with 200 steps of model training. This means training these linear mappings for increasing the initial accuracy is redundant. Besides, as discussed in Sec 2.3, we argue that the initial accuracy is not the key for a successful model scaling.

C. Combine TripLe with KD

As we reuse the DeiT architectures, the output has two parts: (1) the output logits of distillation head o_t and (2) the output logits of classification head o_s. Assuming the output logits of teacher model is Z_t, the corresponding teaching label would be $y_t = \arg\max_c Z_t(c)$. When KD is applied, the hard loss is defined as Eq 19.

$$\mathcal{L}_{global}^{hardDistill} = \frac{1}{2}\mathcal{L}_{CE}(\psi(o_s), y) + \frac{1}{2}\mathcal{L}_{CE}(\psi(o_t), y_t) \quad (19)$$

ψ is the softmax function. $\mathcal{L}_{CE}$ is the cross-entropy loss. During model evaluation under KD, the prediction comes from the combination of both o_s and o_t: $\bar{y} = \arg\max_c \frac{o_s+o_t}{2}(c)$.

When we disable the knowledge distillation, we follow the official DeiT implementation[2] for training and the loss is given as Eq 20.

$$\mathcal{L}_{global} = \frac{1}{2}\mathcal{L}_{CE}(\psi(\frac{o_s + o_t}{2}), y) \quad (20)$$

D. Learning Curve of NAS

For each trial, both TripLe-NAS and multi-trial NAS conduct 30 epochs of training. The leraning curve of the agent during searching phase is given in Figure 5. Generally, both multi-trial and TripLe-NAS gradually increases reward over time. The learning curve of TripLe is more stable compared to multi-trial.

[2]https://github.com/facebookresearch/deit

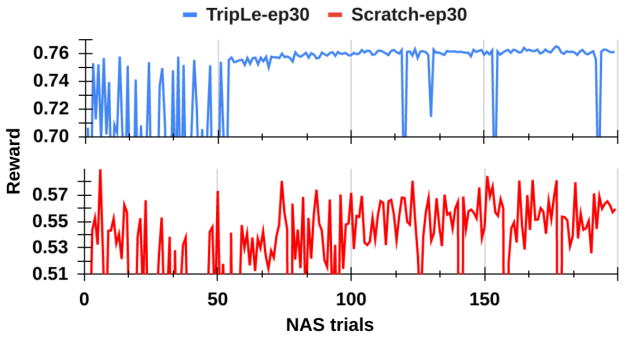


Figure 5. Learning Curve of the agents during NAS when each sample is trained with (1) TripLe$_{ep30}$ (2) Scratch$_{ep30}$.

Table 10. Transfer learning results on various datasets.

Model	Params	FLOPs	CF-10	CF-100	Cars	Flowers
DeiT-B (official)	86M	33.7B	99.1	90.8	92.1	98.4
S→B, LTG	86M	33.7B	99.1	90.7	92.1	97.8
S→B, TripLe$_{ep300}$	86M	33.7B	99.1	90.8	92.2	98.4

E. Model Transfer Learning

Table E shows the transfer learning results of ViT-TripLe and ViT-Scratch. For the downstream tasks, the inputs are resized into 224×224.

F. Searched architectures.

Table F shows the models searched using NAS with TripLe and traditional multi-trial NAS.

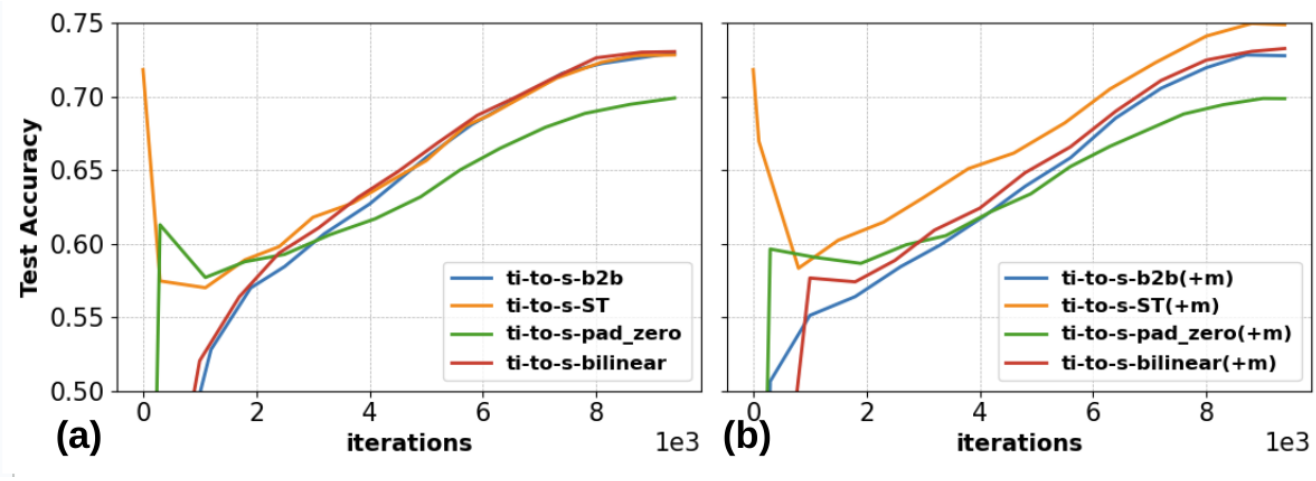


Figure 6. Training Ti→S with 30 epochs using different width expansion methods, i.e., γ_{b2B}, γ_{ST}, γ_{pad0}, γ_{intp}. '+m' denotes we also employ optimizer states in the pretrained model as discussed in Sec 2.3.

Table 11. Searched Architectures from (1) multi-trial NAS with TripLe and (2) traditional multi-trial NAS.

Model	Params	FLOPs	hidden dim	Layers	hf	ef	wd	lr
ViT-TripLe	27M	10416M	384	19	[32,32, 64,64,64,32,32,32,32,32,32,64] [32, 64,32,32,64,32,32]	[2,4,2,2,2,4,4,2,2,4,2,2] [4,4,2,2,2,4,2]	0.05	4e-3
ViT-Scratch	30M	11409M	384	19	[32,32,64,32,32,64,32,64,32,64,64,64] [32,32,32,32,32,32,32]	[3,4,4,2,3,4,2,3,4,4,4,2] [4,4,2,2,2,4,2]	0.05	4e-3

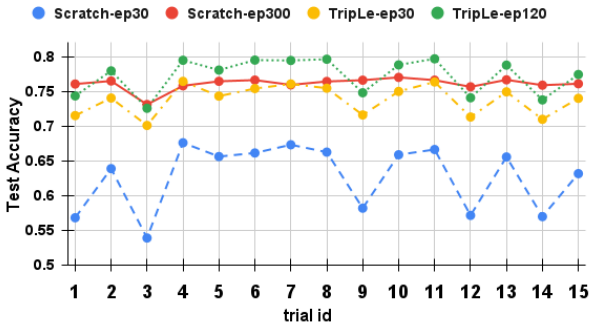


Figure 7. Task performance when trained with (1) TripLe$_{ep30}$(2) TripLe$_{ep120}$ (3) Scratch$_{ep30}$ (4) Scratch$_{ep30}$.