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†, which partially expands a pretrained model be-
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 \textbf{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 \textbf{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 \textbf{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 – \textit{pretrained model reuse} and \textit{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) \cite{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 \textit{single-trial model scaling} and \textit{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 \(8 \times\), 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 \cite{42} and the model searched by traditional progressive learning, and knowledge distillation.

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

This study focuses on scaling pretrained ViTs, and our experiments reuse the DeiT architectures \cite{42}. To study scaling ViTs in both depth and width, we also introduce two model variants namely DeiT-B\(_{L,24}\) and DeiT-S\(_{L,24}\) given in Table 1.

#### 2.2. Revisiting Operators Scaling

Recent studies \cite{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 \(\gamma/\beta\). The large model \(\Theta\) to be trained has

<table>
<thead>
<tr>
<th>Model (DeiT)</th>
<th>hidden dim</th>
<th>#heads</th>
<th>#layers</th>
<th>#params (billions)</th>
<th>FLOPs (billions)</th>
<th>Top-1′ Acc</th>
<th>Top-1 Acc</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ti</td>
<td>192</td>
<td>3</td>
<td>12</td>
<td>5M</td>
<td>2.16</td>
<td>72.0</td>
<td>72.1</td>
</tr>
<tr>
<td>S</td>
<td>384</td>
<td>6</td>
<td>12</td>
<td>22M</td>
<td>8.50</td>
<td>79.5</td>
<td>79.8</td>
</tr>
<tr>
<td>B</td>
<td>768</td>
<td>12</td>
<td>12</td>
<td>86M</td>
<td>33.72</td>
<td>81.0</td>
<td>81.8</td>
</tr>
<tr>
<td>L</td>
<td>1024</td>
<td>16</td>
<td>24</td>
<td>307M</td>
<td>119.36</td>
<td>82.1</td>
<td>82.2</td>
</tr>
</tbody>
</table>

† Top-1 Accuracy of our DeiT re-implementation with 300 epochs of training.
‡ Official Top-1 accuracy under 300 epochs of training.
L transformer layers with hidden dimension D. The pre-trained model Θ has l layers with hidden dimension d.

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

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

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

Here, $W_i$ denotes the initial weight of the i-th transformer layer in model $\Theta$. $k$ 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 $\beta_{\text{ST}}$. 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:

$$x' = x + \text{Attention}(LN(x))$$

$$y = x' + \text{FFN}(LN(x'))$$

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. $\beta_{\text{ST}}$ can be combined with layer $\beta_{\text{stack}}$ or $\beta_{\text{inpt}}$, i.e., where to add these identity layers. They are denoted as $\beta_{\text{ST}_{\text{stack}}}$ and $\beta_{\text{ST}_{\text{inpt}}}$, respectively.

bert2BERT $\gamma_{\text{bert2BERT}}$: 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 $\gamma_{\text{pad0}}$: 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 $\gamma_{\text{ST}}$: Staged-training [38] proposes a new width expansion method. When scaling a dense layer, the width expansion can be written as:

$$\gamma_{\text{ST}}(w) = \left( \begin{array}{ccc} w & z \\ z & w \end{array} \right) \quad (4)$$

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 $\gamma_{\text{inpt}}$: In this work, we propose a new baseline operator interpolation $\gamma_{\text{inpt}}$ 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 $\gamma_{\text{inpt}}$.

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

<table>
<thead>
<tr>
<th>Method</th>
<th>Notation</th>
<th>width</th>
<th>depth</th>
</tr>
</thead>
<tbody>
<tr>
<td>bert2BERT [5]</td>
<td>$\beta_{\text{bert2BERT}}$</td>
<td>$\gamma_{\text{bert2BERT}}$</td>
<td>$\beta_{\text{ST}}$</td>
</tr>
<tr>
<td>Staged-Training [38]</td>
<td>ST</td>
<td>$\gamma_{\text{ST}}$</td>
<td>$\beta_{\text{ST}_{\text{stack}}}$</td>
</tr>
<tr>
<td>Model Interpolation</td>
<td>inpt</td>
<td>$\gamma_{\text{inpt}}$</td>
<td>$\beta_{\text{inpt}}$</td>
</tr>
<tr>
<td>Learn-to-grow [47]</td>
<td>LTG</td>
<td>$\gamma_{\text{ltg}}$</td>
<td>$\beta_{\text{ltg}}$</td>
</tr>
<tr>
<td>Pad Zero</td>
<td>Pad0</td>
<td>$\gamma_{\text{pad0}}$</td>
<td>$\beta_{\text{pad0}}$</td>
</tr>
</tbody>
</table>

Learn-to-grow $\gamma_{\text{ltg}}$ / $\beta_{\text{ltg}}$: 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 $\gamma_{\text{ltg}}$ and $\beta_{\text{ltg}}$, 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 $\gamma_{\text{bert2BERT}}$, $\gamma_{\text{ST}}$ and $\beta_{\text{ST}}$ claim they are functionality preserving. We rebuild these baselines and their initialized accuracy is given in Table 3 (marked in blue).

1. bert2BERT $\gamma_{\text{bert2BERT}}$: We find $\gamma_{\text{bert2BERT}}$ can preserve transformer functionality under constraint. When applying $\gamma_{\text{bert2BERT}}$ to scale a $d \times d$ dense layer into $2d \times 2d$, the original output vector $o$ can become $\tilde{o} = \{\frac{1}{2}, \frac{1}{2}\}$. After another non-linear function $F$, the output $F(\tilde{o})$ 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 FNN, we find the ViT functionality can be fully maintained using $\gamma_{\text{bert2BERT}}$ (Appendix B.2).

2. $\gamma_{\text{ST}}$ and $\beta_{\text{ST}}$: We find the operator $\gamma_{\text{ST}}/\beta_{\text{ST}}$ can preserve the functionality of the ViT. $\beta_{\text{ST}}$ is an identity layer and it is functionality preserving as discussed in Sec 2.2. For $\gamma_{\text{ST}}$, when expanding a $d \times d$ dense layer into $2d \times 2d$. The new dense layer has output $\tilde{o} = \{o, o\}$ where $o$ is the original output with a size of $d \times 1$ according to Eq.4. Because $\gamma_{\text{ST}}$ does not scale $o$ as $\gamma_{\text{bert2BERT}}$, the output results GeLU($\tilde{o}$) can be recovered back to GeLU($o$) after another linear mapping.

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

\[ m_t = \beta_1 m_{t-1} + (1 - \beta_1) g_t \]  
\[ v_t = \beta_2 v_{t-1} + (1 - \beta_2) g_t^2 \]  
\[ \theta_{t+1} = \theta_t - \frac{\lambda(t)}{\sqrt{v_t} + \epsilon} \cdot m_t \]  

Here, \( g_t \) is the first-order gradient of the weight parameter \( \theta \) at time step \( t \). \( \lambda(t) \) is the learning rate scheduler. \( \beta_1 \) and \( \beta_2 \) are constant decay rates (0.9, 0.999). \( m \) and \( v \) are AdamW states with the same dimensions as \( \theta \). 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 \( \gamma / \beta \) 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 \( \gamma ST \) with momentum information outperforms the second-best baseline by 0.65%. With depth expansion, \( \beta \) 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 \( \frac{m_t}{v_t} \) is preserved. One exception is \( \beta \), 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 \( L_{24} \) by combining the best scaling operators among \( \gamma \) and \( \beta \)? We expand the model from Ti\( \rightarrow S L_{24} \) using the best-performed operator selected from the previous investigation, namely, \( \gamma ST \) and \( \beta \) (Table 3). Ti\( \rightarrow S L_{24} \) shows a 2.7% accuracy drop compared to S\( \rightarrow S L_{24} \). This is due to the limitations of simply combining \( \gamma ST \) and \( \beta \). (1) The functionality-preserving feature of \( \gamma ST \) and \( \beta \) is compromised, when combining them together. This incurs the advantage of applying optimizer states. (2) For Ti\( \rightarrow S L_{24} \), 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 \( \beta \) and \( \gamma ST \) together is suboptimal. We propose TripLe that scales width before training and grows model depth during training (Figure 1(a)). TripLe maintains the functionality preserving feature of \( \gamma ST \) and \( \beta \) by serializing the scaling operations. Furthermore, with a short warmup training after conducting \( \gamma ST \), zero weights will obtain their training states that ben-
Scaling Width Before Training. TripLe applies $\gamma_{ST}$ to expand transformer width before the training. We extend $\gamma_{ST}$ 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).

$$
\gamma_{ST}(w) = \left\{ \begin{array}{c}
w \ldots \ z \ z_a \\
\vdots \quad \vdots \quad \vdots \\
z \quad \vdots \quad \vdots \\
z \quad w \quad z_a \\
z^T \quad z^T \quad w_a \\
\end{array} \right\}, \; k = \lceil D/d \rceil
$$

Here, $w$ is the $d \times d$ pretrained dense layer. $w_a$ is down-sampled from $w$ with a size of $ds \times ds$, $ds = D \mod d$. $z$ and $z_a$ 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.

$$
\gamma_{ST}(b) = (b \ldots b \ b_a), \; k = \lceil D/d \rceil
$$

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 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 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 $i$ ($i \in \{1, \ldots, \frac{l}{2}\}$) to the top layer $j$.

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...
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

$$
\text{Reward}(\Theta) = Q(\Theta) + \epsilon \frac{FLOPs(\Theta)}{FLOPs_{s_0}} - 1 \quad (12)
$$

Here, $Q(\cdot)$ indicates the quality (accuracy) of a candidate architecture $\Theta$, $FLOPs(\Theta)$ is its FLOPs, $FLOPs_{s_0}$ 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 $8 \times 8$ topology. The validation/test sets are evaluated every 400 seconds on separate TPUs.

We set the batch size to 4096, so the learning rate would be $\frac{\text{Batch size}}{512} \times 0.0005 = 0.004$ 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 $ep$), 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 ($\text{Max ↓ Time}$) and maximum training FLOPs reduction ($\text{Max ↓ FLOPs}$) 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 \( L \), 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 \( B_{L,24} \) 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 \( Ti \rightarrow S_{L,24} \), the model achieves 82.0\% test accuracy compared to 81.0\% obtained from KD. When combining TripLe with KD, the model accuracy of \( S_{L,24} \) 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 \( \beta_{stack} \) 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. Keep-

---

<table>
<thead>
<tr>
<th>Method</th>
<th>Max ( ep )</th>
<th>FLOPs</th>
<th>Trip-1 Test accuracy (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Scratch</td>
<td>0%</td>
<td>-</td>
<td>79.97</td>
</tr>
<tr>
<td>MLST</td>
<td>30%</td>
<td>601.0</td>
<td>80.42</td>
</tr>
<tr>
<td>Scratch</td>
<td>70%</td>
<td>79.98</td>
<td>80.14</td>
</tr>
<tr>
<td>B_{L,24}</td>
<td>80.70</td>
<td>80.28</td>
<td></td>
</tr>
<tr>
<td>TripLe</td>
<td>80.77</td>
<td>82.23</td>
<td></td>
</tr>
<tr>
<td>B_{L,24}</td>
<td>80.40</td>
<td>80.20</td>
<td></td>
</tr>
<tr>
<td>TripLe</td>
<td>80.70</td>
<td>82.23</td>
<td></td>
</tr>
<tr>
<td>B_{L,24}</td>
<td>80.77</td>
<td>82.23</td>
<td></td>
</tr>
<tr>
<td>TripLe</td>
<td>80.40</td>
<td>80.20</td>
<td></td>
</tr>
<tr>
<td>B_{L,24}</td>
<td>80.70</td>
<td>82.23</td>
<td></td>
</tr>
<tr>
<td>TripLe</td>
<td>80.77</td>
<td>82.23</td>
<td></td>
</tr>
<tr>
<td>B_{L,24}</td>
<td>80.40</td>
<td>80.20</td>
<td></td>
</tr>
<tr>
<td>TripLe</td>
<td>80.70</td>
<td>82.23</td>
<td></td>
</tr>
<tr>
<td>B_{L,24}</td>
<td>80.77</td>
<td>82.23</td>
<td></td>
</tr>
<tr>
<td>TripLe</td>
<td>80.40</td>
<td>80.20</td>
<td></td>
</tr>
<tr>
<td>B_{L,24}</td>
<td>80.70</td>
<td>82.23</td>
<td></td>
</tr>
<tr>
<td>TripLe</td>
<td>80.77</td>
<td>82.23</td>
<td></td>
</tr>
<tr>
<td>B_{L,24}</td>
<td>80.40</td>
<td>80.20</td>
<td></td>
</tr>
<tr>
<td>TripLe</td>
<td>80.70</td>
<td>82.23</td>
<td></td>
</tr>
<tr>
<td>B_{L,24}</td>
<td>80.77</td>
<td>82.23</td>
<td></td>
</tr>
<tr>
<td>TripLe</td>
<td>80.40</td>
<td>80.20</td>
<td></td>
</tr>
<tr>
<td>B_{L,24}</td>
<td>80.70</td>
<td>82.23</td>
<td></td>
</tr>
<tr>
<td>TripLe</td>
<td>80.77</td>
<td>82.23</td>
<td></td>
</tr>
<tr>
<td>B_{L,24}</td>
<td>80.40</td>
<td>80.20</td>
<td></td>
</tr>
<tr>
<td>TripLe</td>
<td>80.70</td>
<td>82.23</td>
<td></td>
</tr>
<tr>
<td>B_{L,24}</td>
<td>80.77</td>
<td>82.23</td>
<td></td>
</tr>
<tr>
<td>TripLe</td>
<td>80.40</td>
<td>80.20</td>
<td></td>
</tr>
<tr>
<td>B_{L,24}</td>
<td>80.70</td>
<td>82.23</td>
<td></td>
</tr>
<tr>
<td>TripLe</td>
<td>80.77</td>
<td>82.23</td>
<td></td>
</tr>
</tbody>
</table>

---

5.2. Evaluation of TripLe on Multi-Task 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-task 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 300 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-task 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.
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_{ep=30}) and scratch training (Scratch_{ep=30}). We set the FLOPs target (FLOPs_{S0}) 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).

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

<table>
<thead>
<tr>
<th>Method</th>
<th>Search method</th>
<th>Evaluation method</th>
<th>Params (MB)</th>
<th>FLOPs (Million)</th>
<th>Test Acc (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>DeiT-S (ours)</td>
<td>Scratch_{ep=30}</td>
<td>Scratch_{ep=30}</td>
<td>22</td>
<td>8495</td>
<td>79.5</td>
</tr>
<tr>
<td>DeiT-B (ours)</td>
<td>Scratch_{ep=30}</td>
<td>Scratch_{ep=30}</td>
<td>86</td>
<td>33722</td>
<td>81.0</td>
</tr>
<tr>
<td>ViT-scratch</td>
<td>Scratch_{ep=30}</td>
<td>TripLe_{ep=30}</td>
<td>30</td>
<td>11409</td>
<td>79.5</td>
</tr>
<tr>
<td>ViT-scratch</td>
<td>Scratch_{ep=30}</td>
<td>TripLe_{ep=30}</td>
<td>30</td>
<td>11409</td>
<td>80.8</td>
</tr>
<tr>
<td>ViT-TripLe</td>
<td>TripLe_{ep=30}</td>
<td>TripLe_{ep=30}</td>
<td>27</td>
<td>10416</td>
<td>79.7</td>
</tr>
<tr>
<td>ViT-TripLe</td>
<td>TripLe_{ep=30}</td>
<td>TripLe_{ep=30}</td>
<td>27</td>
<td>10416</td>
<td>81.1</td>
</tr>
</tbody>
</table>

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.

**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 super-network 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


A. Training Hyperparameters


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 $\gamma_{ST}$ with momentum information outperforms other baselines.

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

bert2BERT ($\gamma_{b2b}$): 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 $d_{o} = d_{in}^Tw_0$. $w_0$ has a dimension of $2 \times 2$ and $d_{in}$ has a dimension of $2 \times 1$. After layer scaling, $w_0'$ has a size of $4 \times 4$.

$$
\begin{array}{c}
  d_{in} = \begin{bmatrix} a \\ b \end{bmatrix}, \quad w_0 = \begin{bmatrix} o \\ q \\ p \\ r \end{bmatrix}, \quad d_{o} = d_{in}^Tw_0 = \begin{bmatrix} a_o \\ b_o \end{bmatrix}
\end{array}
$$

(13)

When expanding the weight matrix $w_0$ given in Eq. 13 from a $2 \times 2$ matrix into a $4 \times 4$ 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:

$$
\begin{array}{c}
  d'_{in} = \begin{bmatrix} a \\ b \\ a \\ a \end{bmatrix}, \quad w'_0 = \begin{bmatrix} o \\ q \\ q \\ r \\ r \\ r \\ 3 \\ 3 \\ 3 \\ 3 \end{bmatrix}
\end{array}
$$

(14)

As is shown above, the result $d'_{in}^Tw'_0 = d_{in}^Tw_0$ does not change during the expansion.

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

$$
\begin{array}{c}
  d''_{in} = \begin{bmatrix} a \\ b \\ a \\ a \\ a \\ a \\ a \\ a \end{bmatrix}, \quad w''_0 = \begin{bmatrix} o \\ q \\ q \\ q \\ r \\ r \\ r \\ 3 \\ 3 \\ 3 \\ 3 \end{bmatrix}
\end{array}
$$

(15)

The final output $o'' = d''_{in}^Tw''_0$ would be $o'' = [a_o, b_o, a_o, b_o]$. 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.

$$
LayerNorm(o) = \frac{(o'' - \mu_o)}{\sigma_o} \odot W_{LN}^o + b_{LN}
$$

(16)

However, if the next layer is LayerNorm (Eq 16). The mean ($\mu_o$) and variance ($\sigma_o$) 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. $W_{LN}^o$ and $b_{LN}$.

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

Learn-to-grow..learn-to-share [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 $\gamma_{ltg}$ and $\beta_{ltg}$, respectively.

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

(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 $W'_i$ into $W$.

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

(18)

Here $P_i$ is a $1 \times l$ vector. $i$ is the layer numbers 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.

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>4096</td>
<td>4e-3</td>
<td>cosine</td>
<td>5</td>
<td>0.1</td>
<td>0.0</td>
<td>0.1</td>
<td>x</td>
<td>x</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

Table 10. Transfer learning results on various datasets.

<table>
<thead>
<tr>
<th>Model</th>
<th>Params</th>
<th>FLOPs</th>
<th>CF-10</th>
<th>CF-100</th>
<th>Cars</th>
<th>Flowers</th>
</tr>
</thead>
<tbody>
<tr>
<td>DeiT-B (official)</td>
<td>86M</td>
<td>33.7B</td>
<td>99.1</td>
<td>90.8</td>
<td>92.1</td>
<td>98.4</td>
</tr>
<tr>
<td>S→B, LTG</td>
<td>86M</td>
<td>33.7B</td>
<td>99.1</td>
<td>90.7</td>
<td>92.1</td>
<td>97.8</td>
</tr>
<tr>
<td>S→B, TripLe-ep300</td>
<td>86M</td>
<td>33.7B</td>
<td>99.1</td>
<td>90.8</td>
<td>92.2</td>
<td>98.4</td>
</tr>
</tbody>
</table>

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.

<table>
<thead>
<tr>
<th>Model</th>
<th>Params</th>
<th>FLOPs</th>
<th>CF-10</th>
<th>CF-100</th>
<th>Cars</th>
<th>Flowers</th>
</tr>
</thead>
<tbody>
<tr>
<td>DeiT-B (official)</td>
<td>86M</td>
<td>33.7B</td>
<td>99.1</td>
<td>90.8</td>
<td>92.1</td>
<td>98.4</td>
</tr>
<tr>
<td>S→B, LTG</td>
<td>86M</td>
<td>33.7B</td>
<td>99.1</td>
<td>90.7</td>
<td>92.1</td>
<td>97.8</td>
</tr>
<tr>
<td>S→B, TripLe-ep300</td>
<td>86M</td>
<td>33.7B</td>
<td>99.1</td>
<td>90.8</td>
<td>92.2</td>
<td>98.4</td>
</tr>
</tbody>
</table>

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.

D. Learning Curve of NAS

For each trial, both TripLe-NAS and multi-trial NAS conduct 30 epochs of training. The learning 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.

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.

\[
L_{\text{hardDistill}}^{\text{global}} = \frac{1}{2} L_{CE}(\psi(o_s), y) + \frac{1}{2} L_{CE}(\psi(o_t), y_t) \tag{19}
\]

\(\psi\) is the softmax function. \(L_{CE}\) is the cross-entropy loss. During model evaluation under KD, the prediction comes from the combination of both \(o_s\) and \(o_t\): \(\bar{y} = \arg \max_c \psi\left(\frac{o_s + o_t}{2}\right)\).

When we disable the knowledge distillation, we follow the official DeiT implementation\(^2\) for training and the loss is given as Eq 20.

\[
L_{global} = \frac{1}{2} L_{CE}(\psi\left(\frac{o_s + o_t}{2}\right), y) \tag{20}
\]

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 \(S\rightarrow B\), 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 \(\gamma_{ST}\) alone to scale \(S\rightarrow B\) can achieve the pretrained DeiT-S accuracy (79%) at step 0. \(\gamma_{Pad}\) 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.
Figure 6. Training Ti→S with 30 epochs using different width expansion methods, i.e., $\gamma_{b2b}$, $\gamma_{ST}$, $\gamma_{pad0}$, $\gamma_{intp}$. ‘+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.

<table>
<thead>
<tr>
<th>Model</th>
<th>Params</th>
<th>FLOPs</th>
<th>hidden dim</th>
<th>Layers</th>
<th>hl</th>
<th>ef</th>
<th>wd</th>
<th>lr</th>
</tr>
</thead>
<tbody>
<tr>
<td>ViT-TripLe</td>
<td>27M</td>
<td>10416M</td>
<td>384</td>
<td>19</td>
<td>[32,32,64,64,32,32,32,32,32,32,64]</td>
<td>[2,4,2,2,2,4,2,2,2,4,2]</td>
<td>0.05</td>
<td>4e-3</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>[32,64,32,32,64,32,32]</td>
<td>[4,4,2,2,2,4,2]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ViT-Scratch</td>
<td>30M</td>
<td>11409M</td>
<td>384</td>
<td>19</td>
<td>[32,32,64,32,32,64,32,64,64,64]</td>
<td>[3,4,4,2,3,4,2,3,4,4,2]</td>
<td>0.05</td>
<td>4e-3</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>[32,32,32,32,32,32,32,32]</td>
<td>[4,4,2,2,2,4,2]</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 7. Task performance when trained with (1) TripLe_{ep30} (2) TripLe_{ep120} (3) Scratch_{ep30} (4) Scratch_{ep30}. 

[Image of graphs and tables as shown in the document]