On the Transformer Growth for Progressive BERT Training

Xiaotao Gu♠♦* **Liyuan Liu**♠ **Hongkun Yu**♦ **Jing Li**♦ **Chen Chen**♦ **Jiawei Han**♠
♦ Google Research {hongkuny,jingli,chendouble}@google.com
♠ University of Illinois at Urbana-Champaign {xiaotao2,ll2,hanj}@illinois.edu

Abstract

Due to the excessive cost of large-scale language model pre-training, considerable efforts have been made to train BERT progressively—start from an inferior but low-cost model and gradually grow the model to increase the computational complexity. Our objective is to advance the understanding of Transformer growth and discover principles that guide progressive training. First, we find that similar to network architecture search, Transformer growth also favors compound scaling. Specifically, while existing methods only conduct network growth in a single dimension, we observe that it is beneficial to use compound growth operators and balance multiple dimensions (e.g., depth, width, and input length of the model). Moreover, we explore alternative growth operators in each dimension via controlled comparison to give operator selection practical guidance. In light of our analyses, the proposed method *CompoundGrow* speeds up BERT pre-training by 73.6% and 82.2% for the base and large models respectively, while achieving comparable performances[1].

1 Introduction

Thanks to the rapid increase of computing power, large-scale pre-training has been breaking the glass ceiling for natural language processing tasks (Liu et al., 2018; Peters et al., 2018; Devlin et al., 2019; Liu et al., 2019; Brown et al., 2020). However, with great power comes great challenges: the required excessive computational consumption significantly impedes the efficient iteration of both research exploration and industrial application. To lower the training cost, many attempts have been made to conduct *progressive training*, which starts from training an inferior but low-cost model, and gradually increases its resource consumption (Gong et al., 2019; Devlin et al., 2019). As elaborated in Section 5, two components are typically needed for designing such progressive training algorithms—the growth scheduler and the growth operator (Dong et al., 2020). The former controls when to conduct network growth, and the latter controls how to perform network growth. Here, our objectives are to better understand growth operators with a focus on Transformer models (Vaswani et al., 2017; Liu et al., 2020b), and specifically, to help design better progressive algorithms for BERT pre-training (Devlin et al., 2019). Specifically, we recognize the importance of using *compound growth operators* in our study, which balance different model dimensions (e.g., number of layers, the hidden size, and the input sequence length).

Regarding previous efforts made on Transformer growth, they mainly focus on one single model dimension: either the length (Devlin et al., 2019) or the depth (Gong et al., 2019). In this work, however, we find that *compound effect* plays a vital role in growing a model to different capacities, just like its importance in deciding network architectures under specific budgets (Tan and Le, 2019). Here, we show that growing a Transformer from both dimensions leads to better performance with less training cost, which verifies our intuition and shows the potential of using compound growth operators in progressive BERT training.

Further, we explore the potential choices of growth operators on each dimension. We conduct controlled experiments and comprehensive analyses to compare various available solutions. These analyses further guide the design of effective compound growth operators. Specifically, we observe that, on the length dimension, embedding pooling is more effective than directly truncating sentences. On the width dimension, parameter sharing outperforms low-rank approximation.

Guided by our analyses, we propose *CompoundGrow* by combining the most effective growth oper-

[*] Work done while interning at Google. Corresponding Author: Hongkun Yu and Xiaotao Gu.

[1] Code will be released at: https://github.com/google-research/google-research/tree/master/grow_bert

Proceedings of the 2021 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, pages 5174–5180
June 6–11, 2021. ©2021 Association for Computational Linguistics

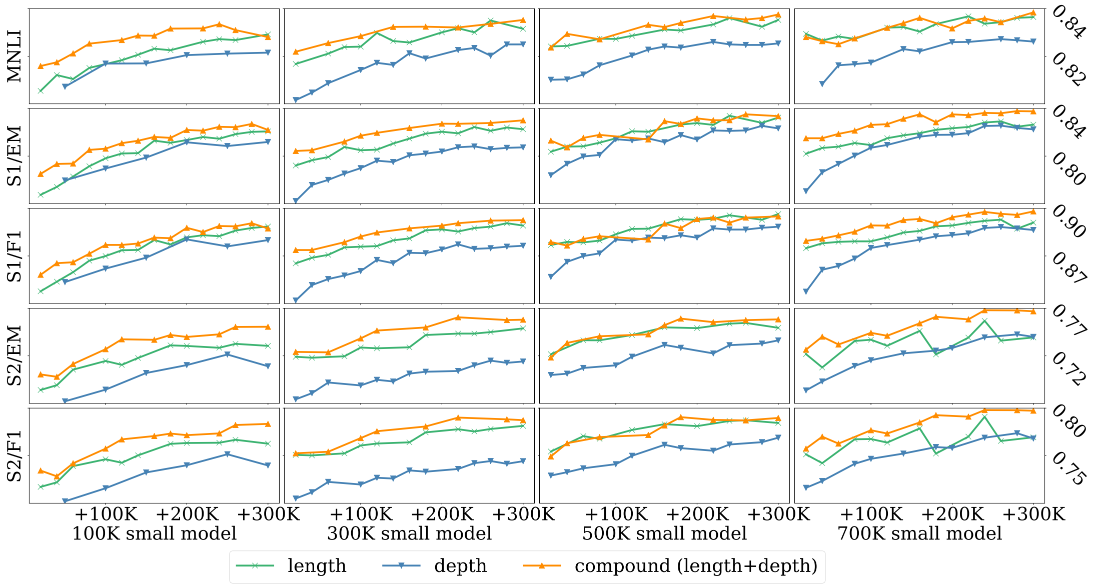

Figure 1: Comparison of single-dimensional operators and the compound operator with comparable cost. Y-axis indicates finetuning performances, including MNLI-match valid accuracy (MNLI), SQuaD v1.1 exact match score and F1 (S1/EM, S1/F1), and SQuaD v2.0 exact match score and F1 (S2/EM, S2/F1). X-axis stands for different training steps of the full model (12-layer BERT-base model with 512-token training data) in the last stage. Different columns represent different training steps for the small (low-cost) model. The three compared methods start from different small models: *depth* stands for a 3-layer model; *length* stands for training with 128-token training data; *compound* stands for a 6-layer model with 256-token training data.

```
Algorithm 1: Progressive Training.
$f$: network; $opt$: optimizer; $\mathcal{D}$: dataset.
$g_t$: the growth operator at stage $t$.
$T$: the total number of growing stages.
  $f_0 \leftarrow opt(f_0, \mathbf{x}, y)$
  for $t \in [1, T]$ do
      $f_t \leftarrow g_t(f_{t-1})$ for $\mathbf{x}, y \in \mathcal{D}$ do
          $f_t \leftarrow opt(f_t, \mathbf{x}, y)$
  return Final network $f_T$
```

ator on each dimension. Experiments on standard benchmarks show that, without sacrificing final performance, the final model speeds up the overall pre-training by 73.6% and 82.2% on BERT-base and BERT-large models respectively.

2 Progressive Compound Growth

Progressive Training. Algorithm 1 presents a generic setup for progressive training. In each training stage t, the corresponding growth operator g_t grows the model f. Then, f is updated by the optimizer *opt* before entering the next training step. Correspondingly, our goal is to maximize the final model performance after all training stages, which can be formulated as minimizing the empirical loss $\mathcal{L}$ over dataset $\mathcal{D}$:

$$\min_{g_t \in \mathcal{G}} \mathcal{L}(f_T) \quad s.t. \; f_t = opt\left(g_t(f_{t-1}), \mathcal{D}\right) \qquad (1)$$

Compound Effect. Existing progressive training methods only focus on one model dimension. For example, Gong et al. (2019) conduct Transformer growth by gradually increasing the network *depth*. Devlin et al. (2019) use shorter input sequence *length* at early stages. However, as studies in network architecture search have revealed (Tan and Le, 2019), growth operators that balance different model dimensions can achieve better performance than single-dimensional operators under the same budget. Note that our objective (Equation 1) is close to the objective of EfficientNet (Tan and Le, 2019), which aims to find the optimal network architecture by maximizing the model accuracy for a given resource budget:

$$\max_{d,w,r} \; Accuracy(\mathcal{N}(d, w, r))$$
$$s.t. \;\; Resource\_cost(\mathcal{N}) \leq \text{target\_budget},$$

where $\mathcal{N}(d, w, r)$ is a CNN network, d, w, r are coefficients to scale its depth, width, and resolution. In this work, we find that such a *compound effect* also plays a vital role in progressive BERT training. Intuitively, growing the network from more than one dimension creates larger potential to get better performance with less resource. Restricting the growth operator from handling all dimensions would lead to inferior performance, as

$\min_{g \in \mathcal{G}} \mathcal{L}(f_T) \geq \min_{g \in \mathcal{G} \cup \mathcal{G}^+} \mathcal{L}(f_T)$. The optimal value of the objective function (Equation 1) is bounded by the feasible set of the growth operator.

Empirical Verification. For empirical verification, we compare existing single-dimensional growth operators in model depth and length with the corresponding compound operator that balances both dimensions. For all three compared growth operators, their configurations are adjusted to make sure they have the same model after growth, and their low-cost models have empirically comparable training costs. As to the training, we first train the low-cost model for 100/300/500/700K steps, and then grow the model to a standard BERT-base model for another 300K steps training. For models trained with different steps/growth operators, we compare their performance after finetuning on MNLI, SQuaD v1.1, and SQuaD v2.0 respectively.

As Figure 1 shows, across different settings (columns) and metrics (rows), the compound operator consistently outperforms or at least achieves comparable results with single-dimensional operators. The observation meets our intuition: to achieve same speedup, the compound method can distribute the reduction on training cost to different dimensions, and achieve better performance.

3 Explore Possible Growth Operators

After verifying the importance of compound growing, we conduct more analysis to provide guidance for growth operator design.

3.1 Length Dimension

Data Truncation first limits the maximum length of input sequences by truncating the training sentences to a shorter length, and then train the model on full-length data. Note that shorter input sequences usually come with less masked tokens to predict in each sentence. For instance, Devlin et al. (2019) first use sentences of at most 128 tokens (with 20 masked tokens) before training on data of 512 tokens (with 76 masked tokens). The major issue of this data truncation operator is the incomplete update of position embeddings. The model needs to learn embeddings for the extra positions from scratch at the last stage.

Embedding Pooling. Inspired by the idea of multigrid training in the vision domain (Wu et al., 2020), we train the model with "low-resolution text" through embedding pooling over unmasked tokens. Compared with data truncation, this method leaves the training data intact and can update all position embeddings. Specifically, since the output length of self-attention modules is decided by the length of query vectors, we only conduct pooling on query vectors in the first self-attention layer and keep key/value vectors intact.

As shown in the first group of Table 1, data truncation (sequence length$=256$) and mean pooling ($k=2$) has similar performance on MNLI and SQuAD v1.1, while mean pooling outperforms data truncation on SQuAD v2.0.

3.2 Width Dimension

On the width dimension, we focus our study on the feedforward network module (FFN). Similar to gradually increasing the network depth, one can also gradually increase the network width for Transformer growth. Specifically, the FFN module can be formed as $f(xW_1)W_2$, where $f(\cdot)$ is the activation function, $W_1 \in \mathbb{R}^{D\times H}$ and $W_2 \in \mathbb{R}^{H\times D}$ are parameters, D and H are the embedding size and the hidden size respectively.

Matrix Factorization. A straightforward method is to approach the original weight matrix $W_i \in \mathbb{R}^{m\times n}$ by the product of two small matrices $W_{i1} \in \mathbb{R}^{m\times h}$ and $W_{i2} \in \mathbb{R}^{h\times n}$ in the early training stage. In the late stage of training, we would recover W_i as $W_{i1} \times W_{i2}$ and unleash the full potential.

Parameter Sharing. Instead of decomposing original weight matrices with low-rank approximation, we try to employ parameter sharing by spliting the matrix into multiple blocks and sharing parameters across different blocks. Formally, for input x,

$$f(xW_1)W_2 = f(x[W_1', \ldots, W_1'])\begin{bmatrix} W_2'/k \\ \ldots \\ W_2'/k \end{bmatrix} = f(xW_1')W_2'. \tag{2}$$

Specifically, in the early training stage, we replace W_1 and W_2 with smaller matrices $W_1' \in \mathbb{R}^{D\times \frac{H}{k}}$ and $W_2' \in \mathbb{R}^{\frac{H}{k}\times D}$. Then, at the growth step, we vertically duplicate (share) W_1' for k times along the dimension with size H/k as the new W_1. W_2 is generated similarly. Similar to matrix factorization, this setting also preserves the output after the growth. Random noise is added to W_1 and W_2 by the dropout layers in FFN, so that the shared small matrices will have different outputs and gradients in later training steps (Chen et al., 2015).

As the second group of Table 1 shows, parameter sharing has significant superiority over matrix fac-

Table 1: Empirical comparison among growth operators. For each operator, a low-cost model is first trained for 700K steps, then grown to the original BERT model for another 300K steps training.

	BERT$_{\text{base}}$					BERT$_{\text{large}}$				
	MNLI	**SQuAD v1.1**		**SQuAD v2.0**		**MNLI**	**SQuAD v1.1**		**SQuAD v2.0**	
	Acc.	**EM**	**F1**	**EM**	**F1**	**Acc.**	**EM**	**F1**	**EM**	**F1**
Data Truncation	83.72	82.72	90.00	76.06	79.18	85.80	85.51	92.18	79.56	82.57
Embed Pooling	84.04	82.96	90.16	76.83	79.88	85.88	85.07	91.95	80.86	83.69
FFN Factorization	83.53	82.21	89.45	75.27	78.11	85.96	85.66	92.10	79.35	82.38
FFN Share Param.	83.92	83.02	89.91	75.83	78.56	86.28	85.60	92.02	80.92	83.85

torization with comparable budgets (k=4 for parameter sharing and h=0.2D for matrix factorization).

3.3 Depth Dimension

Transformer growth in the depth dimension has been thoroughly discussed in literature (Gong et al., 2019; Li et al., 2020). Our observation in this dimension is consistent with their conclusions. In experiments we also compare compound growth with the standard progressive stacking method.

Discussion. From the perspective of implementation, compound growth introduces little additional engineering effort compared with progressive stacking. Specifically, the growth step of progressive stacking basically copies the parameters of the small model to corresponding layers of the full model. The growth on the width dimension is a similar parameter copying process for the fully connected layers, while the growth on the length dimension removes the embedding pooling layer without changing any model parameters.

4 Experiment

Experiment Setups. We train the original BERT models following the same settings in (Devlin et al., 2019) with 256 batch size and 512-token data. All compared models will finally grow to the original model, and keep the total number of training steps to 1M. We evaluate the final model on the GLUE benchmark (Wang et al., 2018) including 9 subtasks, and the two versions of SQuAD (Rajpurkar et al., 2018) datasets for question answering. More detailed experiment settings can be found in the appendix for reproduction.

Compared Methods. Previous studies have rarely focused on progressive Transformer growth for BERT training, and progressive Transformer stacking (Gong et al., 2019) is the only directly comparable method to the best of our knowledge. We apply their method on the official BERT model with the same training setting, learning rate schedule and hardware as our method. We set the training schedule as 300K steps with ¼ number of layers, 400K steps with ½ number of layers, and 300K steps with the full model.

Our Method. For *CompoundGrow*, we apply treatments on three dimensions for the low-cost model: (1) mean embedding pooling with size 2 on the length dimension; (2) parameter sharing with $k = 2$ on FFN modules on the width dimension; (3) stacking on the depth dimension. Following the same setting as compared methods, we try to equally distribute the 1M training steps. We train the model with all treatments with ¼ number of layers and ½ number of layers for 200K steps respectively, and then stack it to full layers with treatments on the width and length dimensions for another 300K steps. At the last stage, we train the full model for 300K steps, just like the compared method.

Results. Table 2 shows the speedup of different models. We estimate the inference FLOPs for compared models and get their real training time from the Tensorflow profiler [2]. On the BERT-base model, stacking and *CompoundGrow* speeds up pre-training by 68.7% and 107.1% respectively in FLOPs, 64.9% and 73.6% respectively on walltime. On the BERT-large model, stacking and *CompoundGrow* speeds up pre-training by 70.7% and 111.4% respectively in FLOPs, 69.7% and 82.2% respectively on walltime. Though *CompoundGrow* is significantly faster, on development sets of MNLI and SQuaD, the compared methods do not have significantly different finetuning performance from the original BERT models.

Table 3 shows the test performance on the GLUE benchmark. Both compared methods achieve at least the same performance as the original BERT model. While *CompoundGrow* saves more training time, it achieves the same performance with stacking on the large model. On the base model, stacking is better in terms of average GLUE score, mainly

[2] https://www.tensorflow.org/guide/profiler

Table 2: The pre-training speedup and finetuning performance on dev sets of MNLI and SQuaD. M/MM stands for matched/mismatched accuracy for MNLI. EM/F1 represents exact match score and F1 score for SQuaD. The FLOPs are estimated for forward pass operations, while the walltime is real training time profiled by the TensorFlow profiler from a distributed multi-host setting.

	speedup	**speedup**	**MNLI Acc.**		**SQuAD v1.1**		**SQuAD v2.0**	
	(FLOPs)	**(walltime)**	**M**	**MM**	**EM**	**F1**	**EM**	**F1**
BERT$_{\textbf{BASE}}$	–	–	84.4	84.4	83.3	90.2	77.4	80.4
Stack$_{\textbf{BASE}}$	+68.7%	+64.9%	84.5	84.9	83.5	90.5	77.1	80.3
Compound$_{\textbf{BASE}}$	**+107.1%**	**+73.6%**	84.7	84.7	83.8	90.3	77.0	80.0
BERT$_{\textbf{LARGE}}$	–	–	86.3	86.4	86.2	92.7	81.0	84.3
Stack$_{\textbf{LARGE}}$	+70.7%	+69.7%	86.9	87.3	86.3	92.6	81.7	84.7
Compound$_{\textbf{LARGE}}$	**+111.4%**	**+82.2%**	87.3	86.8	85.8	92.4	82.4	85.3

Table 3: The test performance on the GLUE benchmark with metrics described in the original paper (Wang et al., 2018), the higher the better. Compound stands for the proposed method with speedup shown in Table 2.

	CoLA	**SST-2**	**MRPC**	**SST-B**	**QQP**	**MNLI-m/mm**	**QNLI**	**RTE**	**WNLI**	**GLUE**
BERT$_{\textbf{BASE}}$	52.1	93.5	88.9/84.8	87.1/85.8	71.2/89.2	84.6/83.4	90.5	66.4	65.1	78.3
Stack$_{\textbf{BASE}}$	57.3	92.8	89.4/85.6	85.4/84.1	71.0/89.1	84.7/83.5	91.4	69.9	63.7	79.1
Compound$_{\textbf{BASE}}$	50.1	92.6	89.1/85.2	85.4/83.9	70.9/88.9	84.6/83.6	91.3	70.1	65.1	78.3
BERT$_{\textbf{LARGE}}$	60.5	94.9	89.3/85.4	87.6/86.5	72.1/89.3	86.7/85.9	92.7	70.1	65.1	80.5
Stack$_{\textbf{LARGE}}$	62.2	94.3	89.9/85.9	86.0/85.0	71.2/88.9	86.9/86.3	93.0	75.2	65.1	81.1
Compound$_{\textbf{LARGE}}$	61.2	94.2	90.2/86.7	86.4/85.7	71.4/89.2	87.2/86.1	93.6	73.3	65.8	81.1

due to its advantage on the CoLA dataset. Such an unusual gap on CoLA might be caused by its relatively small volume and corresponding random variance (Dodge et al., 2020). On the larger and more robust MNLI dataset, the compared methods achieve almost the same score.

5 Related Work

Progressive training was originally proposed to improve training stability, which starts from an efficient and small model and gradually increase the model capacity (Simonyan and Zisserman, 2014). Recent study leverages this paradigm to accelerate model training. For example, multi-level residual network (Chang et al., 2018) explores the possibility of augmenting network depth in a dynamic system of view and transforms each layer into two subsequent layers. AutoGrow (Wen et al., 2020) attempts to automate the discover of proper depth to achieve near-optimal performance on different datasets. LipGrow (Dong et al., 2020) proposes a learning algorithm with an automatic growing scheduler for convolution nets. At the same time, many studies have been conducted on the model growing operators. Network Morphism (Wei et al., 2016, 2017) manages to grow a layer to multiple layers with the represented function intact. Net2net (Chen et al., 2015) is a successful application to transfer knowledge to a wider network with function-preserving initialization. Similar ideas can be discovered in many network architectures, including progressive growing of GAN (Karras et al., 2017) and Adaptive Computation Time (Graves, 2016; Jernite et al., 2016).

As large-scale pre-training keeps advancing the state-of-the-art (Devlin et al., 2019; Radford, 2018), their overwhelming computational consumption becomes the major burden towards further developing more powerful models (Brown et al., 2020). Preliminary application of progressive training has been made on Transformer pre-training. (Devlin et al., 2019) designs two-stage training with a reduced sequence length for the first 90% of updates. (Gong et al., 2019) stack shallow model trained weights to initialize a deeper model, which grows the BERT-base model on the depth dimension and achieves 25% shorter training time.

6 Conclusion

In this work we empirically verify the importance of balancing different dimensions in Transformer growth and propose compound growth operators, which integrates operators for more than one dimension. Moreover, we conduct controlled experiments on various design choices of growth operators, which provides a practical guidance to algorithm design. Our final model speeds up the training of the BERT-base and BERT-large models by 73.6% and 82.2% in walltime respectively while achieving comparable performance.

References

T. Brown, B. Mann, Nick Ryder, Melanie Subbiah, J. Kaplan, P. Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, G. Krüger, Tom Henighan, R. Child, Aditya Ramesh, D. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, E. Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, J. Clark, Christopher Berner, Sam McCandlish, A. Radford, Ilya Sutskever, and Dario Amodei. 2020. Language models are few-shot learners. In *NIPS*.

Bo Chang, Lili Meng, Eldad Haber, Frederick Tung, and David Begert. 2018. Multi-level residual networks from dynamical systems view. In *ICLR*.

Chen Chen, Xianzhi Du, Le Hou, Jaeyoun Kim, Pengchong Jin, Jing Li, Yeqing Li, Abdullah Rashwan, and Hongkun Yu. 2020. Tensorflow official model garden.

Tianqi Chen, Ian Goodfellow, and Jonathon Shlens. 2015. Net2net: Accelerating learning via knowledge transfer. In *ICLR*.

Zihang Dai, Guokun Lai, Yiming Yang, and Quoc V Le. 2020. Funnel-transformer: Filtering out sequential redundancy for efficient language processing. *arXiv preprint arXiv:2006.03236*.

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

Jesse Dodge, Gabriel Ilharco, Roy Schwartz, Ali Farhadi, Hannaneh Hajishirzi, and Noah Smith. 2020. Fine-tuning pretrained language models: Weight initializations, data orders, and early stopping. *arXiv preprint arXiv:2002.06305*.

Chengyu Dong, Liyuan Liu, Zichao Li, and Jingbo Shang. 2020. Towards adaptive residual network training: A neural-ode perspective. In *ICML*.

Linyuan Gong, D. He, Zhuohan Li, T. Qin, Liwei Wang, and T. Liu. 2019. Efficient training of bert by progressively stacking. In *ICML*.

Alex Graves. 2016. Adaptive computation time for recurrent neural networks. *arXiv preprint arXiv:1603.08983*.

Yacine Jernite, Edouard Grave, Armand Joulin, and Tomas Mikolov. 2016. Variable computation in recurrent neural networks. *arXiv preprint arXiv:1611.06188*.

Tero Karras, Timo Aila, Samuli Laine, and Jaakko Lehtinen. 2017. Progressive growing of gans for improved quality, stability, and variation. In *ICLR*.

Bei Li, Ziyang Wang, Hui Liu, Yufan Jiang, Quan Du, Tong Xiao, Huizhen Wang, and Jingbo Zhu. 2020. Shallow-to-deep training for neural machine translation. In *EMNLP 2020*.

Liyuan Liu, Haoming Jiang, Pengcheng He, W. Chen, Xiaodong Liu, Jianfeng Gao, and J. Han. 2020a. On the variance of the adaptive learning rate and beyond. In *ICLR*.

Liyuan Liu, X. Liu, Jianfeng Gao, Weizhu Chen, and J. Han. 2020b. Understanding the difficulty of training transformers. In *EMNLP*.

Liyuan Liu, Jingbo Shang, Xiang Ren, Frank Fangzheng Xu, Huan Gui, Jian Peng, and Jiawei Han. 2018. Empower sequence labeling with task-aware neural language model. In *AAAI*.

Y. Liu, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, M. Lewis, Luke Zettlemoyer, and Veselin Stoyanov. 2019. Roberta: A robustly optimized bert pretraining approach. *ArXiv*, abs/1907.11692.

Ilya Loshchilov and Frank Hutter. 2019. Decoupled weight decay regularization. In *ICLR*.

Matthew E Peters, Mark Neumann, Mohit Iyyer, Matt Gardner, Christopher Clark, Kenton Lee, and Luke Zettlemoyer. 2018. Deep contextualized word representations. In *NAACL-HLT*.

A. Radford. 2018. Improving language understanding by generative pre-training.

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

Karen Simonyan and Andrew Zisserman. 2014. Very deep convolutional networks for large-scale image recognition. In *ICLR*.

Mingxing Tan and Quoc Le. 2019. Efficientnet: Rethinking model scaling for convolutional neural networks. In *ICML*.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, L. Kaiser, and Illia Polosukhin. 2017. Attention is all you need. In *NIPS*.

Alex Wang, Amanpreet Singh, Julian Michael, Felix Hill, Omer Levy, and Samuel R Bowman. 2018. Glue: A multi-task benchmark and analysis platform for natural language understanding. In *ICLR*.

Tao Wei, Changhu Wang, and Chang Wen Chen. 2017. Modularized morphing of neural networks. *arXiv preprint arXiv:1701.03281*.

Tao Wei, Changhu Wang, Yong Rui, and Chang Wen Chen. 2016. Network morphism. In *ICML*.

Wei Wen, Feng Yan, Yiran Chen, and Hai Li. 2020. Autogrow: Automatic layer growing in deep convolutional networks. In *KDD*.

Chao-Yuan Wu, Ross Girshick, Kaiming He, Christoph Feichtenhofer, and Philipp Krahenbuhl. 2020. A multigrid method for efficiently training video models. In *CVPR*.

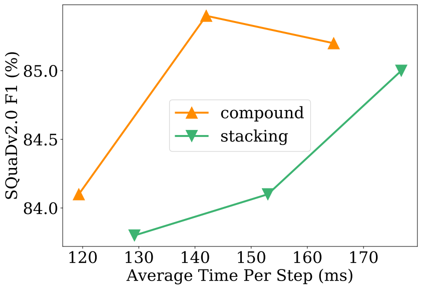

Figure 2: Compare the speed-performance trade-off of stacking and *CompoundGrow* on BERT$_{\text{large}}$. The three data points in each curve is generated with 300K/500K/700K low-cost training steps, respectively.

A Experiment Details

All our models are implemented based on the TensorFlow implementation[3] of BERT (Chen et al., 2020) and trained on TPU v3 with 64 chips. We keep the original WordPieceTokenizer and original position embeddings (instead of relative position encoding used in (Dai et al., 2020)). Following (Devlin et al., 2019), we use the English Wikipedia corpus and the BookCorpus for pre-training. For each finetuning task, we search hyperparameters from following candidates: batch size=16/32/64, learning rate=3e-4/1e-4/5e-5/3e-5.

Optimization. The original BERT models use the AdamW (Loshchilov and Hutter, 2019) optimizer with learning rate decay from 0.0001 to 0 and 10K steps of warmup (Liu et al., 2020a). At the start of each progressive training stage, the learning rate is reset to 0.0001 and keeps decaying as the original schedule.

Baseline Implementation. We apply the compared stacking method (Gong et al., 2019) on the official BERT model with the same training setting, learning rate schedule and hardware as our method, and achieves better performance than the reported numbers in the original paper. To further unleash the potential of the compared method, we adjust their original training schedule to 300K steps with ¼ number of layers, 400K steps with ½ number of layers, and 300K steps with the full model. The new training schedule is much faster than the reported one (speedup from the reported +25% to +64.9%) and still gives better final performance than the original paper. This is the fastest stacking model we can get without performance drop.

B Further Comparison Between *CompoundGrow* and Stacking

To have a deeper understanding of the compared methods, we study their speed-performance trade-off by adjusting the training schedule. Specifically, each time we reduce 200K low-cost training steps for both models, and compare their validation F1 score on SQuaDv2.0. As Figure 2 shows, *CompoundGrow* has clear performance advantage when given comparable training budgets, which further verifies our hypothesis.

[3] https://github.com/tensorflow/models/blob/master/official/nlp/modeling/models/bert_pretrainer.py