DynaBERT: Dynamic BERT with Adaptive Width and Depth*

Lu Hou, Lifeng Shang, Xin Jiang, Qun Liu
Huawei Noah's Ark Lab
{houlu3, shang.lifeng, jiang.xin, qun.liu}@huawei.com

Abstract

The pre-trained language models like BERT and RoBERTa, though powerful in many natural language processing tasks, are both computational and memory expensive. To alleviate this problem, one approach is to compress them for specific tasks before deployment. However, recent works on BERT compression usually reduce the large BERT model to a fixed smaller size, and can not fully satisfy the requirements of different edge devices with various hardware performances. In this paper, we propose a novel dynamic BERT model (abbreviated as DynaBERT), which can run at adaptive width and depth. The training process of DynaBERT includes first training a width-adaptive BERT and then allows both adaptive width and depth, by distilling knowledge from the full-sized model to small sub-networks. Network rewiring is also used to keep the more important attention heads and neurons shared by more sub-networks. Comprehensive experiments under various efficiency constraints demonstrate that our proposed dynamic BERT (or RoBERTa) at its largest size has comparable performance as $\text{BERT}_{\text{BASE}}$ (or $\text{RoBERTa}_{\text{BASE}}$), while at smaller widths and depths consistently outperforms existing BERT compression methods.

1 Introduction

Recently, pre-trained language models based on the Transformer [22] structure like BERT [8] and RoBERTa [14] have achieved remarkable results on natural language processing tasks. However, these models have a lot of parameters, hindering their deployment on edge devices with limited storage, computation and energy consumption. The difficulty of deploying BERT to these devices lies in two aspects. Firstly, the hardware performances of various devices vary a lot, and it is infeasible to deploy one single BERT model to all kinds of edge devices. Different architectural configurations of the BERT model are desired. Secondly, the resource condition of one device under different circumstances can be quite different. For instance, on a mobile phone, when a large number of compute-intensive or storage-intensive programs are running, the resources that can be allocated to the current BERT model will be correspondingly fewer. Thus once the BERT model is deployed, dynamically selecting a part of the model (also referred as *sub-networks*) for inference based on the device's current resource condition is also desirable. Note that unless otherwise specified, the BERT model mentioned in this paper refers to a task-specific BERT rather than the pretrained model.

There have been some attempts to compress and accelerate inference of the Transformer-based models using low-rank approximation [15, 13], weight-sharing [7, 13], knowledge distillation [19, 21, 12], quantization [2, 29, 20] and pruning [18, 16, 17, 6, 23]. However, these methods usually compress the model to a fixed size and can not meet the requirements above. In [7, 10, 9], Transformer-based models with adaptive depth are proposed to dynamically select some of the Transformer layers during inference. However, these depth-adaptive models only consider compression and accelera-

*Work in progress.

tion in the depth direction. Some studies now show that the width direction also has high redundancy. For example, in [23, 17], it is shown that only a small number of the attention heads are required to achieve comparable accuracy as using all of them. Although there have been some works that train convolutional neural networks (CNNs) with adaptive width [28, 27, 26], the Transformer-based models with adaptive width have not been studied.

Adapting along only width or depth has a limited degree of flexibility and can only generate a limited number of architectural configurations. This can be restrictive for multiple performance requirements (e.g., latency, memory, and energy consumption) of multiple hardware platforms or different statuses of one particular platform. In this work, we offer flexibility in *both width and depth* of BERT to enable a significantly larger number of architectural configurations. This also enables better exploration of the balance between model accuracy and model size. Previously a once-for-all CNN with both adaptive width and depth is proposed in [4] by progressively shrinking the model in the width and depth directions. Specifically, the authors first train the CNN with the maximum kernel size, depth and width. Then they fine-tune the network to successively allow elastic kernel size, depth and width. Compared with CNNs, the BERT model is more complicated, which makes it impossible to directly apply the method in [6]. This complexity lies in that each Transformer layer includes both a Multi-Head Attention (MHA) mechanism and a position-wise Feed-forward Network (FFN) that perform transformations in two different dimensions (i.e., the sequence and the feature dimensions). Thus the width of a Transformer-based model can not be simply defined as the number of kernels as in CNNs. Moreover, successive training in first depth and then width can be sub-optimal since these two directions are hard to be disentangled, which may cause the knowledge learned in the depth direction to be forgotten after the width is trained to be adaptive.

In this paper, we propose a novel *dynamic* BERT, or DynaBERT for short, which can be executed at different widths and depths for specific tasks. The training process of DynaBERT includes first training a width-adaptive BERT (abbreviated as DynaBERT$_{\text{W}}$) and then allows both adaptive width and depth in DynaBERT. When training DynaBERT$_{\text{W}}$, we first *rewire* the connections in each Transformer layer based on the importance of the attention heads and neurons to ensure that, the most important heads and neurons are utilized by more sub-networks. Then we distill knowledge from a fixed teacher network to student sub-networks at equal or smaller widths in DynaBERT$_{\text{W}}$. After DynaBERT$_{\text{W}}$ is trained, we initialize from it to train DynaBERT with both adaptive width and depth. To avoid losing the elasticity already learned in the width direction, we use knowledge distillation in both the width and depth directions. We extensively evaluated the effectiveness of our proposed dynamic BERT on the GLUE benchmark under various efficiency constraints (#parameters, FLOPs, inference speed), using both BERT$_{\text{BASE}}$ and RoBERTa$_{\text{BASE}}$ as the backbone models. Under all deployment scenarios, our proposed dynamic BERT and RoBERTa at its largest size have comparable performance as BERT$_{\text{BASE}}$ and RoBERTa$_{\text{BASE}}$, while at smaller widths and depths consistently outperform existing BERT compression methods under the same efficiency constraint.

2 Related Work

In this section, we first describe the formulation of Transformer layer in BERT. Then we briefly review related work on compression of Transformer-based models.

2.1 Transformer Layer

The BERT model is built with Transformer Encoder layers [22], which capture long-term dependencies between input tokens by self-attention mechanism. Specifically, a standard Transformer layer contains a Multi-Head Attention (MHA) layer and a Feed-Forward Network (FFN).

For the t-th Transformer layer, suppose the input to it is $\mathbf{X} \in \mathbb{R}^{n \times d}$ where n and d are the sequence length and hidden state size. Suppose there are N_H attention heads in each layer, with head h parameterized by $\mathbf{W}_h^Q, \mathbf{W}_h^K, \mathbf{W}_h^V, \mathbf{W}_h^O \in \mathbb{R}^{d \times d_h}$ where $d_h = \frac{d}{N_H}$, and output computed as

$$\text{Attn}^h_{\mathbf{W}_h^Q, \mathbf{W}_h^K, \mathbf{W}_h^V, \mathbf{W}_h^O}(\mathbf{X}) = \text{Softmax}(\frac{\mathbf{Q}\mathbf{K}^\top}{\sqrt{d}})\mathbf{V}\mathbf{W}_h^{O\top} = \text{Softmax}(\frac{1}{\sqrt{d}}\mathbf{X}\mathbf{W}_h^Q\mathbf{W}_h^{K\top}\mathbf{X}^\top)\mathbf{X}\mathbf{W}_h^V\mathbf{W}_h^{O\top},$$

In multi-head attention, N_H heads are computed in parallel to get the final output [23]:

$$\text{MHAttn}_{\mathbf{W}^Q,\mathbf{W}^K,\mathbf{W}^V,\mathbf{W}^O}(\mathbf{X}) = \sum_{h=1}^{N_H} \text{Attn}^h_{\mathbf{W}_h^Q,\mathbf{W}_h^K,\mathbf{W}_h^V,\mathbf{W}_h^O}(\mathbf{X}). \tag{1}$$

The FFN layer is parameterized by two matrices $\mathbf{W}_1 \in \mathbb{R}^{d\times d_{ff}}$ and $\mathbf{W}_2 \in \mathbb{R}^{d_{ff}\times d}$ where d_{ff} is the number of neurons in the intermediate layer of FFN. With a slight abuse of notation, we still use $\mathbf{X} \in \mathbb{R}^{n\times d}$ to denote the input to FFN, the output is then computed as:

$$\text{Intermediate}(\mathbf{X}) = \text{GeLU}(\mathbf{X}\mathbf{W}_1 + \mathbf{b}_1); \quad \text{FFN}(\mathbf{X}) = \text{Intermediate}(\mathbf{X})\mathbf{W}_2 + \mathbf{b}_2,$$

where $\mathbf{b}_1, \mathbf{b}_2$ are the bias in the two linear layers.

2.2 Compression for Transformer/BERT

Transformer-based models can be compressed using low-rank approximation [13, 15], weight sharing [7, 13], distillation [19, 21, 12], quantization [20, 29, 3] or pruning [16, 6, 23, 17, 11].

Low-rank approximation approximates the weight matrix by the multiplication of two matrices with lower rank. ALBERT [13] uses low-rank approximation for the word embeddings of BERT model. Tensorized Transformer [15] shows that the output of MHA can be linearly represented by a group of orthonormal base vectors, and multi-linear attention is used to compress the model.

Weight sharing shares parameters across layers in one network. Universal Transformer [7] shares parameters across layers. Compared to the standard Transformer, it gets better performance on language modeling and subject-verb agreement. Deep Equilibrium Models [1] reach an equilibrium point where the input and output of a certain layer stay the same. ALBERT [13] shows that sharing weights across layers stabilizes network parameters and achieves better performance than the original BERT with fewer parameters. Though these weight sharing methods significantly reduce the model size, the inference remains slow.

Distillation transfers knowledge from a big teacher model to a smaller compact model. DistilBERT [19] pre-trains a smaller general-purpose BERT with the use of distillation loss over the soft logits and the hidden states between the student and teacher networks. BERT-PKD [21] uses distillation loss on multiple intermediate layers. TinyBERT [12] successively uses a general distillation and a task-specific distillation over the embedding, attention matrices and output of each Transformer layer together to build task-specific small models.

Quantization represents each weight value using low bits. QBERT [20] uses the second-order information to determine the number of bits for each layer, and more bits are assigned for layers with steeper curvature. They also use group quantization for different weights in MHA. Fully-quantized Transformer [3] uses uniform min-max quantization for computational expensive operations in the Transformer. Weights are bucketed before quantization to reduce quantization error. Q8BERT [29] performs symmetric 8-bit linear quantization on BERT through quantization-aware training.

Pruning removes unimportant connections or neurons in the network. In [11], a magnitude-based pruning method is used to prune unimportant connections during the pre-training phase, while "info deletion" is used to recover some wrongly pruned weights during the fine-tuning phase on the downstream task. In [6], sparse self-attention is introduced during fine-tuning, by replacing the softmax function with a controllable sparse transformation. In [16], gates are put on attention heads, the neurons in the intermediate layer of FFN, and the embeddings to individually eliminate parts of the Transformer. It is shown in [17, 23] that a large percentage of attention heads can be removed without significantly impacting performance. In LayerDrop [10], structured dropout is used to prune Transformer layers for efficient inference. In [9], the encoder-decoder Transformer models are trained to make output predictions at different layers of the decoder for a particular sequence.

However, most of these existing methods can only compress the original model to a specific size, without considering the efficiency constraints of different hardwares or different statuses of one certain hardware. Though Transformer-based models with adaptive depth are proposed in [7, 10,

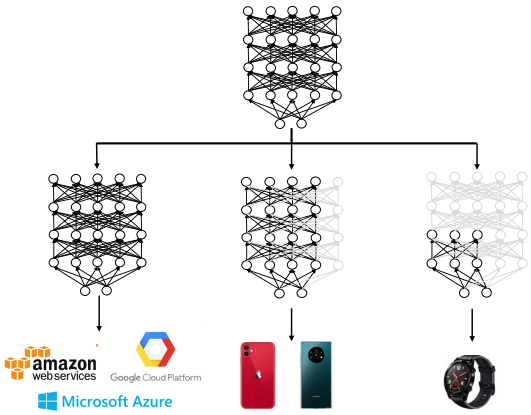


Figure 1: A network with adaptive width and depth. One single model can run at different depths and widths to satisfy various deployment requirements.

9], considering compression and acceleration only in the depth direction can be limited. Recent studies show that the width direction of the Transformer-based models also has high redundancy. For example, it is shown in [23, 17] that comparable accuracy can be well maintained even when many attention heads are pruned.

3 Method

In this section, we elaborate the training method of our DynaBERT model. The training process includes two stages. We first train a width-adaptive DynaBERT$_\text{W}$ in Section 3.1 and then train the both width- and depth-adaptive DynaBERT in Section 3.2.

3.1 Training DynaBERT$_\text{W}$ with Adaptive Width

Before describing the training process, we first need to define the width of BERT model. Compared to CNNs stacked with regular convolutional layers, the BERT model stacked with Transformer layers is much more complicated. In each Transformer layer, the computation of the MHA contains the linear transformation and multiplications of keys, queries, values for multiple heads. Moreover, the MHA and the FFN in each Transformer layer perform transformations in different dimensions, making it hard to trivially determine the width of the Transformer layer.

3.1.1 Using Attention Heads and Intermediate Neurons in FFN to Adapt the Width

Following [17], we divide the computation of the MHA into the computations for each attention head as in (1). Thus the width of the MHA can be decided by the number of attention heads. The width of the FFN can be decided by the number of neurons in the intermediate layer. We do not adapt the number of neurons in the embedding dimension because they are connected through skip connections across all Transformer layers and cannot be flexibly scaled for one particular Transformer layer. Therefore, for a Transformer layer, we adapt its width by varying the number of attention heads of the MHA and neurons in the intermediate layer of the FFN.

In each Transformer layer, when the width multiplier is m_w, the MHA retains the leftmost $\lfloor m_w N_H \rfloor$ attention heads, and the FFN intermediate layer retains the leftmost $\lfloor m_w d_{ff} \rfloor$ neurons. In this case, each Transformer layer is roughly compressed by the ratio m_w. Note that this is not strictly equal because layer normalization and biases in linear layers also have a small fraction of parameters. Different Transformer layers, or the attention heads and the neurons in the same layer, can also have different width multipliers. In this paper, for simplicity, we focus on using the same width multiplier for the attention heads and neurons in all Transformer layers.

3.1.2 Network Rewiring

To fully utilize the network capacity, the more important heads or neurons should be shared across more sub-networks. Since we use the leftmost $\lfloor m_w N_H \rfloor$ attention heads and $\lfloor m_w d_{ff} \rfloor$ neurons, before training the width-adaptive network, we rank the attention heads and neurons according to their importance in the initial BERT model. Following [18, 23], we compute the importance score of a head or neuron based on the variation in the loss if we remove it. Denote the output of one attention head as $\mathbf{h}$, the importance of this head I_h to the loss $\mathcal{L}$ can be estimated using the first-order Taylor expansion as

$$I_{\mathbf{h}} = |\mathcal{L}_{\mathbf{h}} - \mathcal{L}_{\mathbf{h}=\mathbf{0}}| = \left| \mathcal{L}_{\mathbf{h}} - (\mathcal{L}_{\mathbf{h}} - \frac{\partial \mathcal{L}}{\partial \mathbf{h}}(\mathbf{h} - \mathbf{0}) + R_{\mathbf{h}=\mathbf{0}}) \right| \approx \left| \frac{\partial \mathcal{L}}{\partial \mathbf{h}} \mathbf{h} \right|$$

if we ignore the remainder $R_{\mathbf{h}=\mathbf{0}}$. It is shown in [23] that for head $\mathbf{h}$ in MHA, its importance $\left|\frac{\partial \mathcal{L}}{\partial \mathbf{h}}\mathbf{h}\right|$ can be computed using the gradient w.r.t. its mask variable m as

$$\left| \frac{\partial \mathcal{L}}{\partial \mathbf{h}} \mathbf{h} \right| = \left| \frac{\partial \mathcal{L}}{\partial m} \right|. \tag{2}$$

Similarly, for a neuron in the intermediate layer of FFN, denote the set of weights connected to it as $\mathbf{w} = \{w_1, w_2, \cdots, w_K\}$, its importance can be estimated by

$$\left| \frac{\partial \mathcal{L}}{\partial \mathbf{w}} \mathbf{w} \right| = \left| \sum_{i=1}^{K} \frac{\partial \mathcal{L}}{\partial w_i} w_i \right|. \tag{3}$$

Empirically, we use the development set to calculate the importance of the attention heads and neurons according to (2) and (3). Then we arrange the attention heads and neurons in the width direction from highest to lowest (Figure 2) and rewire the corresponding connections. The detailed process is shown in Algorithm 1.

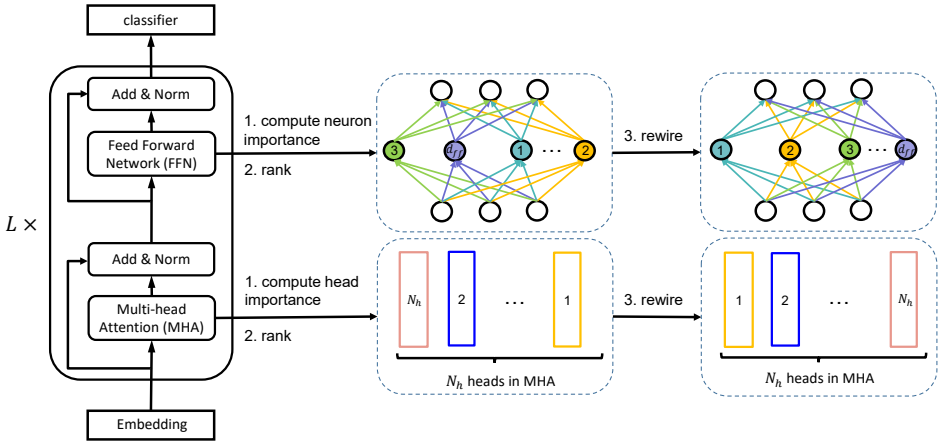


Figure 2: Rewire connections in BERT based on the importance of attention heads and neurons. The left side is a BERT model with L Transformer layers. The middle shows attention heads in MHA and neurons in the intermediate layer in FFN ranked by their importance. The right side shows rewiring connections in each Transformer layer based on the importance of the heads and neurons.

3.1.3 Training with Adaptive Width

After the connections of the BERT model are rewired according to Algorithm 1, we use knowledge distillation to train DynaBERT$_{\text{W}}$. Specifically, we use the rewired BERT model as the fixed teacher network, and to initialize DynaBERT$_{\text{W}}$. Then we distill the knowledge from the fixed teacher model to student sub-networks at different widths in DynaBERT$_{\text{W}}$ (Figure 3).

Algorithm 1 Rewire the network according to the importance of attention heads and neurons.

initialize: development set, trained BERT on downstream task.
Clear gradients of weights $optimizer.zero\_grad()$.
for $iter = 1, ..., T_{val}$ **do**
    Get next mini-batch of data and label.
    Forward propagation, get predicted labels and compute loss $\mathcal{L}$.
    Accumulate gradients $\mathcal{L}.backward()$.
end for
Calculate the importance of attention heads and neurons according to (2) and (3).
Rewire the network according the importance.

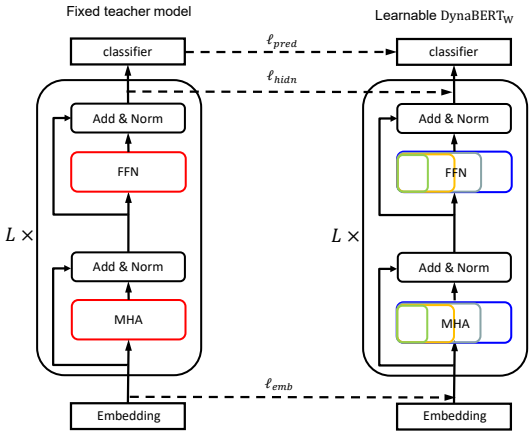


Figure 3: Using knowledge distillation (dashed lines) to transfer the knowledge from a fixed teacher model to student sub-networks at different widths in DynaBERT$_W$. Distillation loss is computed by matching the logits, embedding and hidden states between the teacher and students. The width multipliers used are $[1.0, 0.75, 0.5, 0.25]$.

Take the classification task as an example, we transfer the knowledge in the logits, embedding (i.e., the output of the embedding layer), and hidden states (i.e. the output of each Transformer layer) of all L Transformer layers from the teacher model to student sub-networks. Specifically, suppose the logits, embedding and hidden states of the fixed teacher model at the maximum width are $\mathbf{y}, \mathbf{E}$ and $\mathbf{H}$ respectively, while those for a student sub-network with width multiplier m_w are $\mathbf{y}^{(m_w)}, \mathbf{E}^{(m_w)}$ and $\mathbf{H}^{(m_w)}$ respectively. Here $\mathbf{E}, \mathbf{E}^{(m_w)} \in \mathbb{R}^{n\times d}$ and $\mathbf{H}, \mathbf{H}^{(m_w)} \in \mathbb{R}^{L\times n\times d}$. For the sub-network with width multiplier m_w, its distillation loss function contains three parts. The first part is the distillation loss on the logits ℓ_{pred}, which distills the knowledge from the teacher model's logits to the student through soft cross-entropy loss

$$\ell_{pred}(\mathbf{y}^{(m_w)}, \mathbf{y}) = -\text{softmax}(\mathbf{y}) \cdot \text{log\_softmax}(\mathbf{y}^{(m_w)}). \tag{4}$$

The second part is the distillation loss ℓ_{emb} on the embedding, which distills the knowledge from the teacher model's embedding $\mathbf{E} \in \mathbb{R}^{n\times d}$ to the student $\mathbf{E}^{(m_w)} \in \mathbb{R}^{n\times d}$

$$\ell_{emb}(\mathbf{E}^{(m_w)}, \mathbf{E}) = \text{MSE}(\mathbf{E}^{(m_w)}, \mathbf{E}). \tag{5}$$

The last part is the distillation loss ℓ_{hidn} on the hidden states which makes the output of each transformer layer of the student sub-network $\mathbf{H}_l^{(m_w)}$ mimic $\mathbf{H}_l$ from the teacher model. The loss is computed over all L Transformer layers

$$\ell_{hidn}(\mathbf{H}^{(m_w)}, \mathbf{H}) = \sum_{l=1}^{L} \text{MSE}(\mathbf{H}_l^{(m_w)}, \mathbf{H}_l). \tag{6}$$

Combining (4) - (6), the objective is

$$\mathcal{L} = \lambda_1 \ell_{pred}(\mathbf{y}^{(m_w)}, \mathbf{y}) + \lambda_2(\ell_{emb}(\mathbf{E}^{(m_w)}, \mathbf{E}) + \ell_{hidn}(\mathbf{H}^{(m_w)}, \mathbf{H})), \tag{7}$$

where λ_1 and λ_2 are the scaling parameters that control the weights of different loss terms. Note that we use the same scaling parameter for ℓ_{emb} and ℓ_{hidn}, because the embedding has the same dimension and similar scale as the hidden states. In our experiments, we choose $(\lambda_1, \lambda_2) = (1, 0.1)$ because $\ell_{emb} + \ell_{hidn}$ is around one magnitude larger than ℓ_{pred} empirically. The detailed process is shown in Algorithm 2. To provide more task-specific data for distillation learning, we use the data augmentation method from TinyBERT [12]. Since the labels of augmented data are not always correct, we do not consider the loss between predicted labels and ground-truth labels.

Algorithm 2 Train DynaBERT$_{\text{W}}$ with adaptive width.

input: Training set, width multiplier list $widthList$.
initialize: Initialize a fixed teacher model and a trainable student model $studentModel$ with the rewired model from Algorithm 1.
for $iter = 1, ..., T_{train}$ **do**
  Get next mini-batch of data.
  Execute the teacher model and get the logits $\mathbf{y}$, embedding $\mathbf{E}$, hidden states $\mathbf{H}$.
  Clear gradients in the student model $studentModel.zero\_grad()$.
  for width multiplier m_w in $widthList$ **do**
    Execute sub-network with width multiplier m_w, get the logits $\mathbf{y}^{(m_w)}$, embedding $\mathbf{E}^{(m_w)}$ and hidden states $\mathbf{H}^{(m_w)}$. Compute loss $\mathcal{L}$ using (7).
    Accumulate gradients $\mathcal{L}.backward()$.
  end for
end for
Update the parameters with the accumulated gradients.

3.2 Training DynaBERT with Adaptive Width and Depth

After the training of DynaBERT$_{\text{W}}$ using Algorithm 2, we further use knowledge distillation to train DynaBERT with both adaptive width and depth. Specifically, we use the trained DynaBERT$_{\text{W}}$ model from Section 3.1 as the fixed teacher model, and to initialize the DynaBERT model. Then we distill the knowledge from the fixed teacher model at the maximum depth to student sub-networks at equal or lower depths (Figure 4).

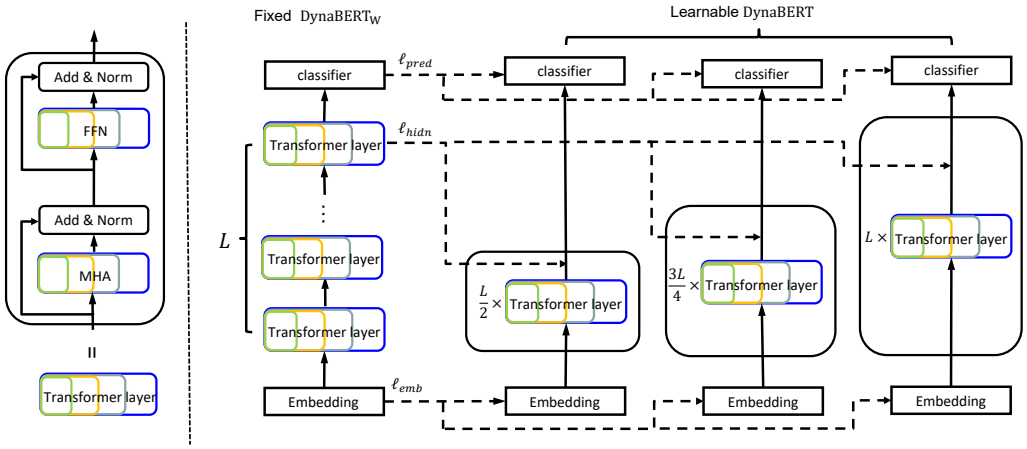


Figure 4: Using knowledge distillation (dashed lines) to transfer the knowledge from a fixed width-adaptive teacher model with the maximum depth to student sub-networks at different depths in the both width- and depth-adaptive model. Distillation loss is computed by matching the logits, embedding, and hidden states between the teacher and students. The width multipliers and depth multipliers used are $[1.0, 0.75, 0.5, 0.25]$ and $[1.0, 0.75, 0.5]$, respectively.

During the training of DynaBERT, to avoid catastrophic forgetting of learned elasticity in the width direction, we still train over different widths in each iteration. For width multiplier m_w, the objec-

tive function of the student sub-network with depth multiplier m_d still contains three parts like in Section 3.1.3. These three parts make the logits $\mathbf{y}^{(m_w,m_d)}$, embedding $\mathbf{E}^{(m_w,m_d)}$ and hidden states $\mathbf{H}^{(m_w,m_d)}$ mimic $\mathbf{y}^{(m_w)}$, $\mathbf{E}^{(m_w)}$ and $\mathbf{H}^{(m_w)}$ from the teacher model with the maximum depth. When the depth multiplier is smaller than 1, the student will have fewer Transformer layers than the teacher. In this case, following [12, 10, 19], we simply drop layers evenly to get a balanced network. Specifically, for depth multiplier m_d, which means we prune layers with a rate $(1 - m_d)$, we drop the layers at depth d which satisfies $\text{mod}(d + 1, \frac{1}{m_d}) \equiv 0$. We use $d + 1$ here instead of d in [10] because we want to keep the last layer which is shown to be important in [25]. For instance, for a BERT model with 12 Transformer layers indexed by $1, 2, 3, \cdots, 12$. When $m_d = 0.75$, we drop the layers at depth $3, 7, 11$. The loss is computed over all the kept layers.

$$\ell_{hidn}(\mathbf{H}^{(m_w,m_d)}, \mathbf{H}^{(m_w)}) = \sum_{l \in \text{keptLayers}} \text{MSE}(\mathbf{H}_l^{(m_w,m_d)}, \mathbf{H}_l^{(m_w)}).$$

Thus the loss can be written as

$$\mathcal{L} = \lambda_1 \ell_{pred}(\mathbf{y}^{(m_w,m_d)}, \mathbf{y}^{(m_w)}) + \lambda_2(\ell_{emb}(\mathbf{E}^{(m_w,m_d)}, \mathbf{E}^{(m_w)}) + \ell_{hidn}(\mathbf{H}^{(m_w,m_d)}, \mathbf{H}^{(m_w)})). \quad (8)$$

For simplicity, we do not tune λ_1, λ_2 and choose $(\lambda_1, \lambda_2) = (1, 1)$ in our experiments. The training procedure can be found in Algorithm 3. After the unsupervised training, one can further fine-tune the network using the cross-entropy loss between the predicted labels and the ground-truth labels. This step improves the performance on some data sets empirically (details can be found in the Section 4.3). In this paper, we use the model with the higher average validation accuracy over all widths and depths between with and without fine-tuning.

Algorithm 3 Train DynaBERT with adaptive width and depth.

input: Training set, width multiplier list $widthList$, depth multiplier list $depthList$.
initialize: Initialize a fixed teacher model and a trainable student model $studentModel$ with the width-adaptive model trained from Algorithm 2.
for $iter = 1, ..., T_{train}$ **do**
  Get next mini-batch of data.
  for width multiplier m_w in $widthList$ **do**
    Execute the teacher model with width multiplier m_w and the maximum depth, append the logits $\mathbf{y}^{(m_w)}$ to $\mathcal{Y}$, the embedding $\mathbf{E}^{(m_w)}$ to $\mathcal{E}$, and the hidden states $\mathbf{H}^{(m_w)}$ to $\mathcal{H}$.
  end for
  Clear gradients in the student model $studentModel.zero\_grad()$.
  for depth multiplier m_d in $depthList$ **do**
    for width multiplier m_w in $widthList$ **do**
      Execute sub-network of the student model with width multiplier m_w and depth multiplier m_d, get logits $\mathbf{y}^{(m_w)}$, embedding $\mathbf{E}^{(m_w)}$ and $\mathbf{H}^{(m_w)}$.
      Based on $\mathbf{y}^{(m_w)}, \mathbf{E}^{(m_w)}$ and $\mathbf{H}^{(m_w)}$ with current width multiplier m_w in $\mathcal{Y}, \mathcal{E}$ and $\mathcal{H}$, compute loss $\mathcal{L}$ using (8).
      Accumulate gradients $\mathcal{L}.backward()$.
    end for
  end for
end for
Update the parameters with the accumulated gradients.

4 Experiment

In this section, we evaluate the efficacy of the proposed DynaBERT on the General Language Understanding Evaluation (GLUE) tasks [24] using both $\text{BERT}_{\text{BASE}}$ [8] and $\text{RoBERTa}_{\text{BASE}}$ [14] as the backbone models. The corresponding width- and depth-adaptive BERT and RoBERTa models are named as DynaBERT and DynaRoBERTa, respectively. In the following, we elaborate our experiment settings in Section 4.1. Then in Section 4.2, we show the performance of our proposed DynaBERT and DynaRoBERTa at different widths and depths, and compare with other compression methods under different resource constraints. Finally, in Section 4.3, we show the importance of the proposed network rewiring (Section 3.1.1), knowledge distillation and data augmentation in the training of $\text{DynaBERT}_{\text{W}}$ (Section 3.1) and DynaBERT (Section 3.2).

4.1 Setting

Data. We conduct comprehensive experiments on the official GLUE benchmark [24], which is a collection of diverse natural language understanding tasks, including textual entailment (RTE and MNLI), question answering (QNLI), similarity and paraphrase (MRPC, QQP, STS-B), sentiment analysis (SST-2) and linguistic acceptability (CoLA). For MNLI, we experiment on both the matched (MNLI-m) and mismatched (MNLI-mm) sections.

Training Details. The DynaBERT and DynaRoBERTa models have the same maximum size as the $\text{BERT}_{\text{BASE}}$ and $\text{RoBERTa}_{\text{BASE}}$ models, respectively. For $\text{BERT}_{\text{BASE}}$ and $\text{RoBERTa}_{\text{BASE}}$, the number of Transformer layers is $L = 12$, the hidden state size is $d = 768$. In each Transformer layer, the number of heads in MHA is $N_H = 12$, and the number of neurons in the intermediate layer in FFN is $d_{ff} = 3072$. The list of width multipliers is $[1.0, 0.75, 0.5, 0.25]$, and the list of depth multipliers is $[1.0, 0.75, 0.5]$. There are a total of $4 \times 3 = 12$ different configurations of sub-networks. The detailed hyperparameters for the width-adaptive training, both width- and depth-adaptive training and the final fine-tuning stages can be found in Table 8 in Appendix A.

Reporting Results. Results on development set are used for evaluation. The metrics are Spearman correlation for STS-B, Matthews correlation for CoLA and accuracy for the other tasks. We compare our proposed DynaBERT and DynaRoBERTa with the following methods: (1) $\text{BERT}_{\text{BASE}}$ [8], (2) $\text{RoBERTa}_{\text{BASE}}$ [14], (3) DistilBERT [19], (4) TinyBERT [12], and (5) LayerDrop [10]. We evaluate the efficacy of our proposed DynaBERT and DynaRoBERTa under different efficiency constraints, including #parameters, FLOPs, the latency on NVIDIA K40 GPU and on Kirin 810 A76 ARM CPU (details can be found in Appendix B).

4.2 Main Results

Results on the GLUE benchmark. In Table 1, we show the evaluation results of sub-networks derived from the proposed DynaBERT and DynaRoBERTa with different width and depth multipliers. The Transformer layers are nearly compressed by rate $m_w \times m_d$, if we do not count the parameters in layer normalization and linear layer bias which are negligible.

Table 1: Results on development set of our proposed DynaBERT and DynaRoBERTa with different width and depth multipliers (m_w, m_d). The highest accuracy among 12 different configurations in each block is highlighted.

Method		CoLA			STS-B			MRPC			RTE		
$\text{BERT}_{\text{BASE}}$		58.1			89.8			87.7			71.1		
	(m_w, m_d)	1.0x	0.75x	0.5x	1.0x	0.75x	0.5x	1.0x	0.75x	0.5x	1.0x	0.75x	0.5x
	1.0x	59.7	59.1	54.6	**90.1**	89.5	88.6	86.3	85.8	85.0	72.2	71.8	66.1
DynaBERT	0.75x	**60.8**	59.6	53.2	90.0	89.4	88.5	**86.5**	85.5	84.1	71.8	**73.3**	65.7
	0.5x	58.4	56.8	48.5	89.8	89.2	88.2	84.8	84.1	83.1	72.2	72.2	67.9
	0.25x	50.9	51.6	43.7	89.2	88.3	87.0	83.8	83.8	81.4	68.6	68.6	63.2
		MNLI-(m/mm)			QQP			QNLI			SST-2		
$\text{BERT}_{\text{BASE}}$		84.8/84.9			90.9			92.0			92.9		
	(m_w, m_d)	1.0x	0.75x	0.5x	1.0x	0.75x	0.5x	1.0x	0.75x	0.5x	1.0x	0.75x	0.5x
	1.0x	**84.9/85.5**	84.4/85.1	83.7/84.6	**91.4**	**91.4**	91.1	92.1	91.7	90.6	93.2	93.3	92.7
DynaBERT	0.75x	84.7/85.5	84.3/85.2	83.6/84.4	**91.4**	91.3	91.2	92.2	91.8	90.7	93.0	93.1	92.8
	0.5x	84.7/85.2	84.2/84.7	83.0/83.6	91.3	91.2	91.0	**92.2**	91.5	90.0	**93.3**	92.7	91.6
	0.25x	83.9/84.2	83.4/83.7	82.0/82.3	90.7	91.1	90.4	91.5	90.8	88.5	92.8	92.0	92.0
		CoLA			STS-B			MRPC			RTE		
$\text{RoBERTa}_{\text{BASE}}$		65.1			91.2			90.7			81.2		
	(m_w, m_d)	1.0x	0.75x	0.5x	1.0x	0.75x	0.5x	1.0x	0.75x	0.5x	1.0x	0.75x	0.5x
	1.0x	63.6	61.0.7	59.5	**91.3**	91.0	90.0	88.7	89.7	88.5	**82.3**	78.7	72.9
DynaRoBERTa	0.75x	**63.7**	61.4	54.9	91.0	90.7	89.7	90.0	89.2	88.2	79.4	77.3	70.8
	0.5x	61.3	58.1	52.9	90.3	90.1	88.9	**90.4**	90.0	86.5	75.1	73.6	71.5
	0.25x	54.2	46.7	39.8	89.6	89.2	87.5	88.2	88.0	84.3	70.0	70.0	66.8
		MNLI-(m/mm)			QQP			QNLI			SST-2		
$\text{RoBERTa}_{\text{BASE}}$		87.5/87.5			91.8			93.1			95.2		
	(m_w, m_d)	1.0x	0.75x	0.5x	1.0x	0.75x	0.5x	1.0x	0.75x	0.5x	1.0x	0.75x	0.5x
	1.0x	**88.3/87.6**	87.7/87.2	86.2/85.8	92.0	**92.0**	91.7	**92.9**	92.5	91.4	**95.1**	94.3	93.3
DynaRoBERTa	0.75x	88.0/87.3	87.5/86.7	85.8/85.4	91.9	91.8	91.6	92.8	92.4	91.3	94.6	94.3	93.3
	0.5x	87.1/86.4	86.8/85.9	84.8/84.2	91.7	91.5	91.2	92.3	91.9	90.8	93.6	94.2	92.9
	0.25x	84.6/84.7	84.0/83.7	82.1/82.0	91.2	91.0	90.5	90.9	90.9	89.3	93.9	93.2	91.6

From Table 1, the proposed DynaBERT (or DynaRoBERTa) achieves comparable performances as $\text{BERT}_{\text{BASE}}$ (or $\text{RoBERTa}_{\text{BASE}}$) with the same or smaller size. For most tasks, the sub-network of

DynaBERT or DynaRoBERTa with the maximum depth and width does not necessarily have the best performance, indicating that there exists redundancy in the original BERT or RoBERTa model. Indeed the width and depth of the model for most tasks can be reduced without performance drop with the proposed method. Another observation is that using one certain width multiplier usually has higher accuracy than using the same depth multiplier. This indicates that compared to the depth direction, the width direction is more robust to compression. Sub-networks from DynaRoBERTa most of the time perform significantly better than those from DynaBERT under the same depth and width.

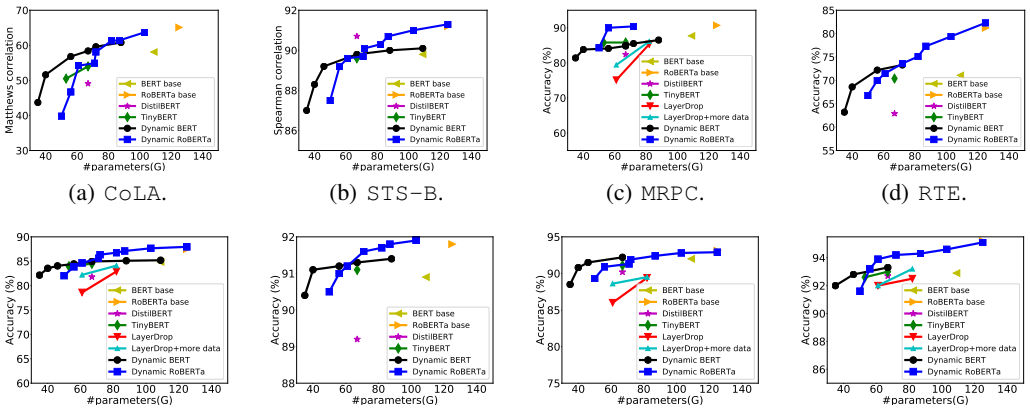
We also show the test set results in Table 2. Again, the proposed DynaBERT achieves comparable accuracy than $\text{BERT}_{\text{BASE}}$ with the same size. Interestingly, the proposed DynaRoBERTa outperforms $\text{RoBERTa}_{\text{BASE}}$ on seven out of eight tasks. A possible reason is that allowing adaptive width and depth increases the training difficulty and acts as regularization, and so contributes positively to the performance.

Table 2: Results on test set of our proposed DynaBERT and DynaRoBERTa. Note that the evaluation metric for QQP and MRPC here is "F1".

	MNLI-m	MNLI-mm	QQP	QNLI	SST-2	CoLA	STS-B	MRPC	RTE
$\text{BERT}_{\text{BASE}}$	84.6	83.6	71.9	90.7	93.4	51.5	85.2	87.5	69.6
DynaBERT $(m_w, m_d = 1, 1)$	84.5	84.1	72.1	91.3	93.0	54.9	84.4	87.9	69.9
$\text{RoBERTa}_{\text{BASE}}$	86.0	85.4	70.9	92.5	94.6	50.5	88.1	90.0	73.0
DynaRoBERTa $(m_w, m_d = 1, 1)$	86.9	86.7	71.9	92.5	94.7	54.1	88.4	90.8	73.7

Comparison with Other Methods. Figures 5-8 show the comparison of our proposed DynaBERT and DynaRoBERTa with other compression methods under different efficiency constraints (i.e., #parameters, FLOPs, inference speed) on different hardware platforms (i.e., NVIDIA K40 GPU and Kirin 810 ARM CPU). Note that each number of TinyBERT and DistilBERT is run using a different model, while different numbers of LayerDrop and our proposed DynaBERT/DynaRoBERTa are run using different sub-networks within one model.

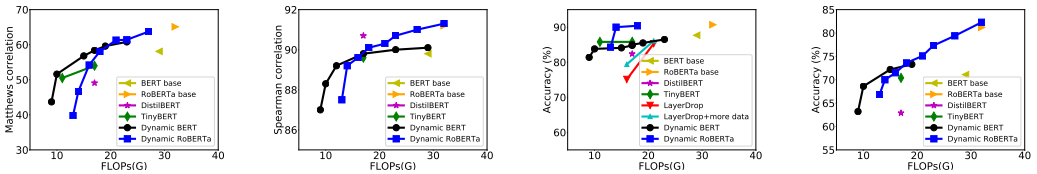
As can be seen, the proposed DynaBERT and DynaRoBERTa achieve comparable accuracy as $\text{BERT}_{\text{BASE}}$ and $\text{RoBERTa}_{\text{BASE}}$, but require fewer or the same number of parameters, FLOPs or lower latency on GPU or ARM CPU. Under the same efficiency constraint, sub-networks extracted from our proposed DynaBERT outperform DistilBERT on all data sets except STS-B under #parameters, and outperforms TinyBERT on all data sets except MRPC; Sub-networks extracted from DynaRoBERTa outperforms LayerDrop by a large margin. Our proposed method even consistently outperforms LayerDrop trained with much more data. We speculate that it is because LayerDrop only allows flexibility on the depth, while ours enable flexibility in both width and depth, which generates a significantly larger number of architectural configurations and better explores of the balance between model accuracy and size.



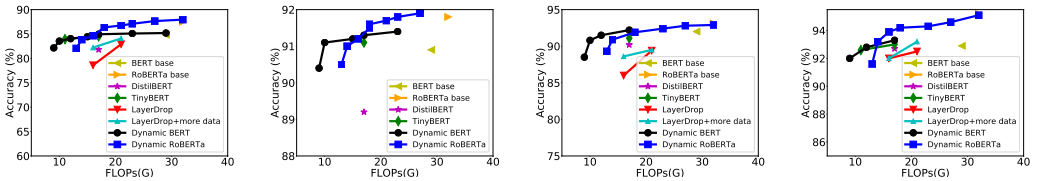
(a) CoLA. (b) STS-B. (c) MRPC. (d) RTE.

(e) MNLI. (f) QQP. (g) QNLI. (h) SST-2.

Figure 5: Comparison of #parameters(G) between our proposed DynaBERT and DynaRoBERTa and other methods. Average accuracy of MNLI-m and MNLI-mm is plotted.

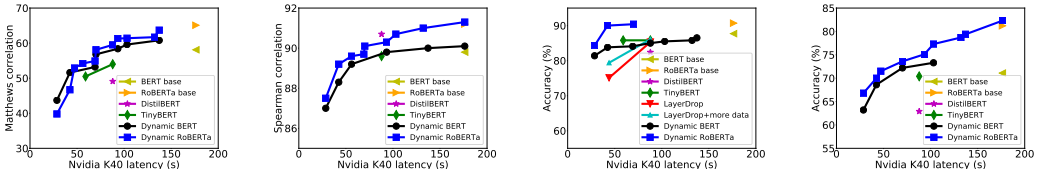


(a) CoLA. (b) STS-B. (c) MRPC. (d) RTE.

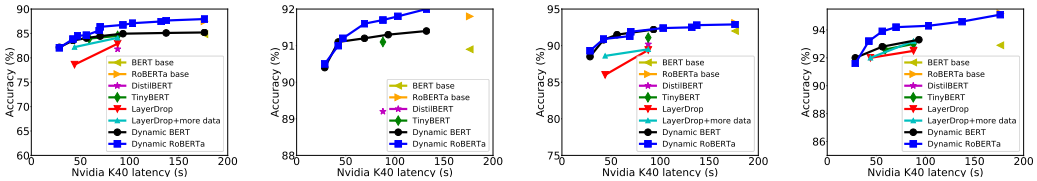


(e) MNLI. (f) QQP. (g) QNLI. (h) SST-2.

Figure 6: Comparison of FLOPs(G) between our proposed DynaBERT and DynaRoBERTa and other methods. Average accuracy of MNLI-m and MNLI-mm is plotted.

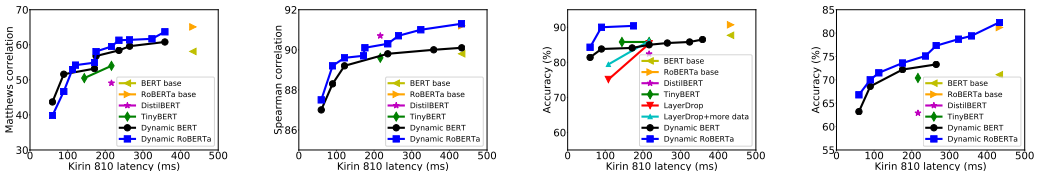


(a) CoLA. (b) STS-B. (c) MRPC. (d) RTE.

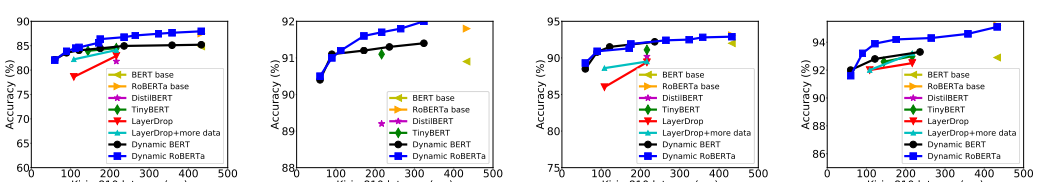


(e) MNLI. (f) QQP. (g) QNLI. (h) SST-2.

Figure 7: Comparison of NVIDIA K40 GPU latency(s) between our proposed DynaBERT and DynaRoBERTa and other methods. The latency is the running time of 100 batches with batch size of 128 and sequence length of 128. Average accuracy of MNLI-m and MNLI-mm is plotted.



(a) CoLA. (b) STS-B. (c) MRPC. (d) RTE.



(e) MNLI. (f) QQP. (g) QNLI. (h) SST-2.

Figure 8: Comparison of Kirin 810 ARM CPU latency(ms) between our proposed DynaBERT and DynaRoBERTa and other methods. The latency is tested with batch size of 1 and sequence length of 128. Average accuracy of MNLI-m and MNLI-mm is plotted.

4.3 Ablation Study

In this section, we do ablation study in the training of DynaBERT$_{\text{W}}$ and DynaBERT.

Training DynaBERT$_{\text{W}}$ with Adaptive Width. We evaluate the importance of network rewiring, knowledge distillation and data augmentation in the training of DynaBERT$_{\text{W}}$ in Table 3. DynaBERT$_{\text{W}}$ trained without network rewiring, knowledge distillation and data augmentation is called "vanilla DynaBERT$_{\text{W}}$". We also compare DynaBERT$_{\text{W}}$ against the baseline of using separate networks, each of which is initialized from the BERT$_{\text{BASE}}$ with a certain width multiplier $m_w \in [1.0, 0.75, 0.5, 0.25]$, and then fine-tuned on the downstream task. Note the difference between the two is that the former trains one single network, but executes at different widths, while the later trains four different networks with different width multipliers.

From Table 3, DynaBERT$_{\text{W}}$ has comparable performance as the separate network baseline at its largest width and has significantly better performance at smaller widths. The smaller the width, the more significant the accuracy gain. After network rewiring, the average development set accuracy on the GLUE benchmark is over 2 points higher than the counterpart without rewiring. The accuracy gain is larger when the width of the model is smaller. With knowledge distillation and data augmentation, the average accuracy is further improved by around 1.5 points.

Table 3: Ablation study in the training of DynaBERT$_{\text{W}}$. Results on the development set are reported. The highest average accuracy over four width multipliers is highlighted.

	m_w	MNLI-m	MNLI-mm	QQP	QNLI	SST-2	CoLA	STS-B	MRPC	RTE	avg.
Separate network	1.0x	84.8	84.9	90.9	92.0	92.9	58.1	89.8	87.7	71.1	83.6
	0.75x	84.2	84.1	90.6	89.7	92.9	48.0	87.2	82.8	66.1	80.6
	0.5x	81.7	81.7	89.7	86	91.4	37.2	84.5	75.5	55.2	75.9
	0.25x	77.9	77.9	89.9	83.7	86.7	14.7	77.4	71.3	57.4	70.8
	avg.	82.2	82.2	90.3	87.8	91.0	39.9	84.6	78.8	61.6	77.6
Vanilla DynaBERT$_{\text{W}}$	1.0x	84.5	85.1	91.3	91.7	92.9	58.1	89.9	83.3	69.3	82.9
	0.75x	83.5	84.0	91.1	90.1	91.7	54.5	88.7	82.6	65.7	81.3
	0.5x	82.1	82.3	90.7	88.9	91.6	46.9	87.3	83.1	61	79.3
	0.25x	78.6	78.4	89.1	85.6	88.5	16.4	83.5	72.8	60.6	72.6
	avg.	82.2	82.5	90.6	89.1	91.2	44.0	87.4	80.5	64.2	79.0
+ Network rewiring	1.0x	84.9	84.9	91.4	91.6	91.9	56.3	90.0	84.6	70.0	82.8
	0.75x	84.3	84.2	91.3	91.7	92.4	56.4	89.9	86.0	71.1	83.0
	0.5x	82.9	82.9	91.0	90.6	91.9	47.7	89.2	84.1	71.5	81.3
	0.25x	80.4	80.0	90.0	87.8	90.4	45.1	87.3	80.4	66.0	78.6
	avg.	83.1	83.0	90.9	90.4	91.7	51.4	89.1	83.8	**69.7**	81.4
+ Distillation and Data Augmentation	1.0x	85.1	85.4	91.1	92.5	92.9	59.0	90.0	86.0	70.0	83.5
	0.75x	84.9	85.6	91.1	92.4	93.1	57.9	90.0	87.0	70.8	83.6
	0.5x	84.4	84.9	91.0	92.3	93.0	56.7	89.9	87.3	71.5	83.4
	0.25x	83.4	83.8	90.6	91.2	91.7	49.9	89.0	84.1	65.7	81.0
	avg.	**84.5**	**84.9**	**91.0**	**92.1**	**92.7**	**55.9**	**89.7**	**86.1**	69.5	82.9

Training DynaBERT with Adaptive Width and Depth. We evaluate the effect of knowledge distillation, data augmentation and final fine-tuning in the training of DynaBERT on four of GLUE data sets SST-2, CoLA, MRPC and RTE in Table 4. The DynaBERT trained without knowledge distillation, data augmentation and final fine-tuning is called "vanilla DynaBERT".

From Table 4, with knowledge distillation and data augmentation, the average accuracy of smaller depth is significantly improved compared to vanilla counterpart on all four data sets. Additional fine-tuning further improves the average accuracy on all three depth multipliers on SST-2, CoLA and two on RTE, but harms the performance on MRPC. Empirically, we choose the model with higher average accuracy between with and without fine-tuning.

5 Discussion

5.1 Comparison of Conventional Distillation and Inplace Distillation

To train width-adaptive CNNs, in [27], inplace distillation is used to boost the performance. In this distillation, sub-network with the maximum width is the teacher while sub-networks with smaller widths in the same model are students. Training loss includes the loss from both the teacher network

Table 4: Ablation study in the training of DynaBERT. Results on the development set are reported. The highest average accuracy over four width multipliers for each depth multiplier is highlighted.

		SST-2			CoLA			MRPC			RTE		
	(m_w, m_d)	1.0x	0.75x	0.5x	1.0x	0.75x	0.5x	1.0x	0.75x	0.5x	1.0x	0.75x	0.5x
Vanilla DynaBERT	1.0x	92.0	91.6	90.9	58.5	57.7	42.9	85.3	83.8	78.4	67.9	66.8	66.4
	0.75x	92.3	91.6	91.1	57.9	56.4	42.4	86.0	83.1	78.7	69.0	66.8	63.9
	0.5x	91.9	91.9	90.6	55.9	53.3	40.6	86.0	83.1	79.7	68.2	65.0	63.9
	0.25x	91.6	91.3	89.0	52.0	50.0	27.6	83.1	80.4	77.5	65.3	63.5	60.3
	avg.	92.0	91.6	90.4	56.1	54.4	38.4	85.1	82.6	78.6	67.6	65.5	63.6
	(m_w, m_d)	1.0x	0.75x	0.5x	1.0x	0.75x	0.5x	1.0x	0.75x	0.5x	1.0x	0.75x	0.5x
+ Distillation and Data augmentation	1.0x	92.9	93.3	92.7	57.1	56.7	52.6	86.3	85.8	85.0	72.2	70.4	66.1
	0.75x	93.1	93.1	92.1	57.7	55.4	51.9	86.5	85.5	84.1	72.6	72.2	64.6
	0.5x	92.9	92.1	91.3	54.1	53.7	47.5	84.8	84.1	83.1	72.9	72.6	66.1
	0.25x	92.5	91.7	91.6	50.7	51.0	44.6	83.8	83.8	81.4	67.5	67.9	62.5
	avg.	92.9	92.6	91.9	54.9	54.2	49.2	**85.4**	**84.8**	**83.4**	**71.3**	70.8	64.8
	(m_w, m_d)	1.0x	0.75x	0.5x	1.0x	0.75x	0.5x	1.0x	0.75x	0.5x	1.0x	0.75x	0.5x
+ Fine-tuning	1.0x	93.2	93.3	92.7	59.7	59.1	54.6	84.1	83.6	82.6	72.2	71.8	66.1
	0.75x	93.0	93.1	92.8	60.8	59.6	53.2	84.8	83.6	82.8	71.8	73.3	65.7
	0.5x	93.3	92.7	91.6	58.4	56.8	48.5	83.6	83.3	82.6	72.2	72.2	67.9
	0.25x	92.8	92.0	92.0	50.9	51.6	43.7	82.6	83.6	81.1	68.6	68.6	63.2
	avg.	**93.1**	**92.8**	**92.3**	**57.5**	**56.8**	**50.0**	83.8	83.5	82.3	71.2	**71.5**	**65.7**

and the student network. In this section, we also adapt inplace distillation to train DynaBERT$_\text{W}$ and compare it with the conventional distillation used in Section 3.1 For inplace distillation, the loss for the student is the distillation loss over logit, embedding and hidden states with the teacher. The loss of the teacher is the distillation loss over logits, embedding and hidden states with a fixed fine-tuned task-specific BERT.

Table 5 shows the comparison of these two kinds of distillation. As can be seen, inplace distillation has higher average accuracy on MRPC and CoLA when training DynaBERT$_\text{W}$. However, in the training of DynaBERT, the model initialized with the inplace distillation can lead to even worse performance than that with the conventional distillation.

Table 5: Comparison of results on development set between using conventional distillation and inplace distillation. For DynaBERT$_\text{W}$, the average accuracy over four width multipliers are reported. For DynaBERT, the average accuracy over four width multipliers and three depth multipliers are reported. The higher accuracy in each group is highlighted.

	Distillation type	SST-2	CoLA	MRPC	RTE
DynaBERT$_\text{W}$	Conventional	92.7	**55.9**	86.1	69.5
	Inplace	92.6	**55.9**	**87.0**	**70.0**
DynaBERT	Conventional	**92.7**	**54.8**	83.2	**69.5**
	Inplace	92.5	54.5	**84.3**	69.0

5.2 Different Methods to Train DynaBERT$_\text{W}$

For DynaBERT$_\text{W}$ (Section 3.1), we rewire the network only once before training by alternating over four different width multipliers. In this section, we also tried adapting the following two methods in training width-adaptive CNNs to BERT: (1) using progressive rewiring as in [4] which progressively rewires the network as more width multipliers are supported; and (2) universally slimmable training [27] which randomly sample some width multipliers in each iteration.

Progressive Rewiring. Instead of rewiring the network only once before training, "progressive rewiring" progressively rewires the network as more width multipliers are supported throughout the training. Specifically, for four width multipliers $[1.0, 0.75, 0.5, 0.25]$, progressive rewiring first sorts the attention heads and neurons and rewires the corresponding connections before training to support width multipliers $[1.0, 0.75]$. Then the attention heads and neurons are sorted and the network is rewired again before supporting $[1.0, 0.75, 0.5]$. Finally, the network is again sorted and rewired before supporting all four width multipliers. For "progressive rewiring", we tune the initial learning rate from $\{2\times10^{-5}, 1\times10^{-5}, 2\times10^{-5}, 5\times10^{-6}, 2\times10^{-6}\}$ and pick the best-performing initial learning rate 1×10^{-5}. Table 6 shows the development set accuracy on the GLUE benchmark

for using progressive rewiring. Since progressive rewiring requires progressive training and is time-consuming, we do not use data augmentation and distillation, and use cross-entropy loss between predicted labels and the ground-truth labels as the training loss. By comparing with Table 3, using progressive rewiring has no significant gain over rewiring only once.

Table 6: Results on development set on the GLUE benchmark using progressive rewiring in training DynaBERT$_{\text{W}}$.

m_w	MNLI-m	MNLI-mm	QQP	QNLI	SST-2	CoLA	STS-B	MRPC	RTE	avg.
1.0x	84.6	84.5	91.5	91.6	92.4	57.4	90.1	86.5	70.0	83.2
0.75x	83.6	84.0	91.2	91.4	91.7	56.6	89.7	84.8	70.8	82.6
0.5x	82.5	82.9	91.0	90.8	91.9	52.2	89.1	84.1	72.9	81.9
0.25x	78.3	79.7	89.9	87.9	90.4	45.1	87.6	82.4	67.5	78.8
avg.	82.3	82.8	90.9	90.4	91.6	52.8	89.1	84.5	70.3	81.6

Universally Slimmable Training. Instead of using a pre-defined list of width multipliers, universally slimmable training [27] samples several width multipliers in each training iteration. Following [27], we also use inplace distillation for universally slimmable training. For universally slimmable training, we tune (λ_1, λ_2) in $\{(1,1),(1,0),(0,1),(1,0.1),(0.1,1),(0.1,0.1)\}$ on MRPC and choose the best-performing one $(\lambda_1, \lambda_2) = (0.1, 0.1)$. The corresponding results for can be found in Table 7. For better comparison with using pre-defined width multipliers, we also report results when the width multipliers are $[1.0, 0.75, 0.5, 0.25]$. We also do not use data augmentation here. By comparing with Table 3, there is no significant difference between using universally slimmable training and the alternative training as used in Algorithm 2.

Table 7: Results on development set on the GLUE benchmark using universal slimmable training in training DynaBERT$_{\text{W}}$.

m_w	MNLI-m	MNLI-mm	QQP	QNLI	SST-2	CoLA	STS-B	MRPC	RTE	avg.
1.0x	84.6	85.0	91.2	91.7	92.4	59.7	90.0	85.3	69.0	83.2
0.75x	84.0	84.5	91.1	91.3	92.5	56.7	90.0	85.3	70.4	82.9
0.5x	82.2	82.6	90.7	90.5	91.1	52.1	89.2	85.3	71.5	81.7
0.25x	79.7	79.5	89.3	87.5	90.1	36.4	87.3	79.4	67.5	77.4
avg.	82.6	82.9	90.6	90.3	91.5	51.2	89.1	83.8	69.6	81.3

6 Conclusion and Future Work

In this paper, we propose a novel model called DynaBERT which can run at different widths and depths. Experiments on various tasks show that under the same efficiency constraint, sub-networks extracted from the proposed DynaBERT consistently achieve better performance than the other BERT compression methods. In this work, we only adapt the width in the number of attention heads in MHA and neurons in the intermediate layer in FFN. In the future, we would also like to change the hidden state size to further compress the model and accelerate inference. Another possible direction is that models which share weights across Transformer layers like ALBERT and universal Transformers might be more suited for adaptive depth. This work focuses on training a dynamic BERT on specific tasks, in the future, we would also like to apply the proposed method to the pre-training stage.

References

[1] S. Bai, J. Z. Kolter, and V. Koltun. Deep equilibrium models. In Advances in Neural Information Processing Systems, pages 688–699, 2019.

[2] A. Bhandare, V. Sripathi, D. Karkada, V. Menon, S. Choi, K. Datta, and V. Saletore. Efficient 8-bit quantization of transformer neural machine language translation model. Preprint arXiv:1906.00532, 2019.

[3] A. Bie, B. Venkitesh, J. Monteiro, M. Haidar, M. Rezagholizadeh, et al. Fully quantizing a simplified transformer for end-to-end speech recognition. Preprint arXiv:1911.03604, 2019.

[4] H. Cai, C. Gan, T. Wang, Z. Zhang, and S. Han. Once for all: Train one network and specialize it for efficient deployment. In International Conference on Learning Representations, 2020.

[5] K. Clark, M. Luong, Q. V. Le, and C. D. Manning. Electra: Pre-training text encoders as discriminators rather than generators. In International Conference on Learning Representations, 2019.

[6] B. Cui, Y. Li, M. Chen, and Z. Zhang. Fine-tune BERT with sparse self-attention mechanism. In Conference on Empirical Methods in Natural Language Processing, 2019.

[7] M. Dehghani, S. Gouws, O. Vinyals, J. Uszkoreit, and L. Kaiser. Universal transformers. In International Conference on Learning Representations, 2019.

[8] J. Devlin, M. Chang, K. Lee, and K. Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. In North American Chapter of the Association for Computational Linguistics, pages 4171–4186, 2019.

[9] M. Elbayad, J. Gu, E. Grave, and M. Auli. Depth-adaptive transformer. In International Conference on Learning Representations, 2020.

[10] A. Fan, E. Grave, and A. Joulin. Reducing transformer depth on demand with structured dropout. In International Conference on Learning Representations, 2019.

[11] M. A. Gordon, K. Duh, and N. Andrews. Compressing bert: Studying the effects of weight pruning on transfer learning. Preprint arXiv:2002.08307, 2020.

[12] X. Jiao, Y. Yin, L. Shang, X. Jiang, X. Chen, L. Li, F. Wang, and Q. Liu. Tinybert: Distilling bert for natural language understanding. Preprint arXiv:1909.10351, 2019.

[13] Z. Lan, M. Chen, S. Goodman, K. Gimpel, P. Sharma, and R. Soricut. Albert: A lite bert for self-supervised learning of language representations. In International Conference on Learning Representations, 2020.

[14] Y. Liu, M. Ott, N. Goyal, J. Du, M. Joshi, D. Chen, O. Levy, M. Lewis, L. Zettlemoyer, and V. Stoyanov. Roberta: A robustly optimized bert pretraining approach. Preprint arXiv:1907.11692, 2019.

[15] X. Ma, P. Zhang, S. Zhang, N. Duan, Y. Hou, D. Song, and M. Zhou. A tensorized transformer for language modeling. In Advances in Neural Information Processing Systems, 2019.

[16] J.S. McCarley. Pruning a bert-based question answering model. Preprint arXiv:1910.06360, 2019.

[17] P. Michel, O. Levy, and G. Neubig. Are sixteen heads really better than one? In Advances in Neural Information Processing Systems, pages 14014–14024, 2019.

[18] P. Molchanov, S. Tyree, T. Karras, T. Aila, and J. Kautz. Pruning convolutional neural networks for resource efficient inference. In International Conference on Learning Representations, 2017.

[19] V. Sanh, L. Debut, J. Chaumond, and T. Wolf. Distilbert, a distilled version of bert: smaller, faster, cheaper and lighter. Preprint arXiv:1910.01108, 2019.

[20] S. Shen, Z. Dong, J. Ye, L. Ma, Z. Yao, A. Gholami, M. W. Mahoney, and K. Keutzer. Q-bert: Hessian based ultra low precision quantization of bert. In AAAI Conference on Artificial Intelligence, 2020.

[21] S. Sun, Y. Cheng, Z. Gan, and J. Liu. Patient knowledge distillation for bert model compression. In Conference on Empirical Methods in Natural Language Processing, pages 4314–4323, 2019.

[22] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin. Attention is all you need. In Advances in neural information processing systems, pages 5998–6008, 2017.

[23] E. Voita, D. Talbot, F. Moiseev, R. Sennrich, and I. Titov. Analyzing multi-head self-attention: Specialized heads do the heavy lifting, the rest can be pruned. In Annual Conference of the Association for Computational Linguistics, 2019.

[24] A. Wang, A. Singh, J. Michael, F. Hill, O. Levy, and S. R. Bowman. Glue: A multi-task benchmark and analysis platform for natural language understanding. In International Conference on Learning Representations, 2019.

[25] W. Wang, F. Wei, L. Dong, H. Bao, N. Yang, and M. Zhou. Minilm: Deep self-attention distillation for task-agnostic compression of pre-trained transformers. Preprint arXiv:2002.10957, 2020.

[26] J. Yu and T. S. Huang. Network slimming by slimmable networks: Towards one-shot architecture search for channel numbers. Preprint arXiv:1903.11728, 2019.

[27] J. Yu and T. S. Huang. Universally slimmable networks and improved training techniques. In IEEE International Conference on Computer Vision, pages 1803–1811, 2019.

[28] J. Yu, L. Yang, N. Xu, J. Yang, and T. Huang. Slimmable neural networks. In International Conference on Learning Representations, 2018.

[29] O. Zafrir, G. Boudoukh, P. Izsak, and M. Wasserblat. Q8bert: Quantized 8bit bert. Preprint arXiv:1910.06188, 2019.

A Hyperparameters

The detailed hyperparameters for the width-adaptive training in Section 3.1, both width- and depth-adaptive training and the final fine-tuning in Section 3.2 are shown in Table 8.

Table 8: Hyperparameters for different stages in training DynaBERT and DynaRoBERTa on the GLUE benchmark.

	Width-adaptive	Width- and depth-adaptive	Final fine-tuning
Data augmentation	y	y	n
Distillation	y	y	n
λ_1, λ_2	1.0, 0.1	1.0,1.0	-
Width mutipliers	[1.0, 0.75,0.5,0.25]	[1.0, 0.75,0.5,0.25]	[1.0, 0.75,0.5,0.25]
Depth multipliers	1	[1.0, 0.75,0.5]	[1.0, 0.75,0.5]
Batch Size	32	32	32
Learning Rate	$2e-5$	$2e-5$	$2e-5$
Warmup Steps	0	0	0
Learning Rate Decay	Linear	Linear	Linear
Weight Decay	0	0	0
Gradient Clipping	1	1	1
Dropout	0.1	0.1	0.1
Attention Dropout	0.1	0.1	0.1
Max Epochs (MNLI, QQP)	1	1	3
Max Epochs (other datasets)	3	3	3
Logging steps (MNLI, QQP)	500	500	50
Logging steps (other datasets)	50	50	10

B Storage, FLOPs and Latency

To count the floating point operations (FLOPs), we follow the setting in [5] and infer FLOPs with batch size of 1 and sequence length of 128. To evaluate the inference speed on GPU, we follow [12], and experiment on the `QNLI` training set with batch size of 128 and sequence length of 128. The numbers are the average running time of 100 batches on a NVIDIA K40 GPU. To evaluate the inference speed on CPU, we experiment on Kirin 810 A76 ARM CPU with batch size of 1 and sequence length of 128.