SuperShaper: Task-Agnostic Super Pre-training of BERT Models with Variable Hidden Dimensions

Vinod Ganesan\textsuperscript{1,3}\textsuperscript{*}, Gowtham Ramesh\textsuperscript{2}\textsuperscript{*}, Pratyush Kumar\textsuperscript{1,3}

\textsuperscript{1}Indian Institute of Technology, Madras \textsuperscript{2}Robert Bosch Center for Data Science and Artificial Intelligence, \textsuperscript{3}Microsoft Research, India

vinodg@cse.iitm.ac.in, gowtham.ramesh1@gmail.com, kumar.pratyush@microsoft.com

Abstract

Task-agnostic pre-training followed by task-specific fine-tuning is a default approach to train NLU models. Such models need to be deployed on devices across the cloud and the edge with varying resource and accuracy constraints. For a given task, repeating pre-training and fine-tuning across tens of devices is prohibitively expensive. We propose SuperShaper, a task agnostic pre-training approach which simultaneously pre-trains a large number of Transformer models by varying shapes, i.e., by varying the hidden dimensions across layers. This is enabled by a backbone network with linear bottleneck matrices around each Transformer layer which are sliced to generate differently shaped sub-networks. In spite of its simple design space and efficient implementation, SuperShaper discovers networks that effectively trade-off accuracy and model size: Discovered networks are more accurate than a range of hand-crafted and automatically searched networks on GLUE benchmarks. Further, we find two critical advantages of shape as a design variable for Neural Architecture Search (NAS): (a) heuristics of good shapes can be derived and networks found with these heuristics match and even improve on carefully searched networks across a range of parameter counts, and (b) the latency of networks across multiple CPUs and GPUs are insensitive to the shape and thus enable device-agnostic search. In summary, SuperShaper radically simplifies NAS for language models and discovers networks that generalize across tasks, parameter constraints, and devices.

1 Introduction

Computing over the last 60 years has swung multiple times between increased centralization and increased distribution. The last decade, with a surge in public and private cloud usage, has centralized compute and storage. But rising cloud costs, ever powerful client devices, and increased call for privacy are swinging the pendulum back to more compute on the edge. This ongoing transition is specifically relevant for compute-intensive AI inference. Caught between both ends of the swinging pendulum, AI model owners must ensure that their models are deployable on tens of different devices spanning CPU and GPU setups on cloud and client devices.

This wide deployment surface is particularly challenging for NLP applications for two reasons. First, NLU models, for instance for sentiment classification, are trained with a hybrid pipeline that combines pre-training and finetuning, where the former is significantly more expensive than the latter. For instance, pre-training BERT on C4 takes 1-5 days on 128 TPUv2s (Liu et al., 2021). In comparison, end-to-end training of GMLP (with similar model parameters) on ImageNet takes 4 hours (Liu et al., 2021). Second, models are usually deployed for multiple NLU tasks such as sentiment classification, paraphrase detection, question answering, natural language inference, etc. For every point in the product space of multiple tasks and multiple devices, different model variants have to be pre-trained and evaluated, which would be very expensive.

There are three broad solutions to address this optimization complexity. First, a single large language model, such as BERT, could be trained agnostic of task and device. Then for every device and task under consideration, the model can be sized down with techniques such as knowledge distillation (Tang et al., 2019; Turc et al., 2019a; Sanh et al., 2019a; Jiao et al., 2020), pruning (Michel et al., 2019; Goyal et al., 2020), quantization (Shen et al., 2020), factorization (Ma et al., 2019), etc. This approach does not generalize across devices or tasks and retains a large search complexity. Also, model compression is inherently lossy in not retaining all information learnt during pre-training. The second approach is to pretrain a single language model but simultaneously finetune many sub-networks of different sizes, in what is called super finetuning. Then for a chosen task and device an appropriately sized sub-network can be sampled from the super-network and deployed. The third approach takes this even further: Instead of pre-training one large language model, an entire family of language models is trained,

\textsuperscript{*}The first two authors have contributed equally.
in what is called super pre-training.

Over the last year, two works have studied task-specific super fine-tuning, namely DynaBERT (Hou et al., 2020) and YOCO-BERT (Zhang et al., 2021), while task-agnostic super-pre-training has been studied in NAS-BERT (Xu et al., 2021). In all three cases, starting from a BERT model, different models are trained as defined by an architectural search space. For instance, in DynaBERT, the number of attention heads and Transformer layers are changed, in NAS-BERT multi-headed attention, separable convolution, and feed-forward networks are independently scaled and combined, and in YOCO-BERT the number of layers, size of representations in FFN, and number of attention heads are changed. Given these large and multi-dimensional search spaces, each method proposes complex algorithms to super-train. DynaBERT proposes knowledge distillation and an importance based rewiring across layers to retain accurate sub-networks. NAS-BERT proposes reducing the search space by discretizing the network into blocks, and a heuristic based search space pruning. YOCO-BERT proposes an iterative explore-exploit algorithm that maintains a dynamic set of promising sub-networks which are trained simultaneously.

We propose an alternative approach to super pretraining language models, called SuperShaper. Like NAS-BERT, SuperShaper is task-agnostic and trains a family of language models. However, SuperShaper differs from existing methods in two crucial ways. First, the search space for SuperShaper is much simpler and includes only the hidden dimension of each Transformer layer, which we refer to as the shape of the network. This is enabled by modifying the BERT backbone with bottleneck matrices at the input and output of each layer, inspired from the efficient model MobileBERT (Sun et al., 2020). In each batch, differently shaped networks are randomly sampled by slicing the bottleneck matrices and trained. Though a single parameter per layer, the hidden dimension sensitively affects model capacity as the parameter count linearly depends on it. Second, the super-pre-training procedure is much simpler with SuperShaper and requires only sliced matrix multiplications on the bottleneck matrices, similar to the earliest techniques proposed for elastic training (Brock et al., 2018; Cai et al., 2019). This is radically simpler than existing NAS techniques which define complex design spaces, architecture modifications, and heuristics for managing the search space. Indeed, in a PyTorch codebase only about 20 lines of additional code are required to add SuperShaper functionality.

Despite the simple design space and efficient implementation, SuperShaper trains sub-networks that are competitive on GLUE tasks with BERT-base at the large end and with many compressed models (both hand-crafted and searched with NAS) at lower parameter counts. In the size interval of 60-660M parameters, the model found with SuperShaper has a higher average accuracy on GLUE than larger models identified with many successful techniques such as LayerDrop, DistillBERT, Bert-PKD, miniLM, Tiny-BERT, BERT-of-Theseus, PD-BERT, ELM, and YOCO-Bert. Only NAS-BERT, with its much larger search space and knowledge distillation reports a higher accuracy by 1%.

While SuperShaper finds competitive networks, we believe the key insight of SuperShaper is in the choice of the search space of shapes, the advantages it affords for deployability, and its implications for NAS in general. The first advantage relates to generalized insights on good shapes. With evolutionary algorithms (EAs) we search for sub-networks that are accurate and meet given parameter constraints. By analysing such networks, we identify heuristics of good shapes, which suggest a cigar-like shape. By applying these heuristics, we hand-craft sub-networks across a range of parameter counts. These hand-crafted sub-networks match and often exceed the performance of networks searched with EAs. Thus, Transformer shapes afford interpretable generalization of model compression across a range of parameter count constraints. No existing NAS technique for super-training or more generally in NLP, demonstrate such generalization. Second, we observe that the latency of networks as measured on CPUs and GPUs is insensitive to the shape of the network: For a given parameter count that meets the latency requirement on a device, the shape of the network is a free variable that can be optimized for accuracy. Thus, when deploying a model for a specific device, we need to compute the largest parameter count that is allowed by the latency constraint and then choose a network with a good shape for that parameter count. We demonstrate that this procedure is effective across GPUs and CPUs. In summary, SuperShaper is a simple yet effective technique to pretrain a range of task-agnostic language models to deploy appropriately sized NLP models on a range of devices. More broadly, our findings suggest that Neural Architecture Search (NAS) can be performed with radically simpler design spaces and implementations, which generalize across tasks, parameter counts, and devices.

2 Related Work

Over the years, a number of solutions have been proposed for efficient deployment of language models. These can be broadly grouped into the following categories.

2.1 Model Compression

In the context of language models, model compression has been widely applied to reduce computational complexity. For instance, (Wang et al., 2019) and (Ma et al., 2019) propose low-rank approximations of weight matrices for faster matrix multiplications. A number of efforts prune either attention heads (Michel et al., 2019),
tokens (Goyal et al., 2020; Wang et al., 2021b; Kim et al., 2021), or even complete layers (Fan et al., 2019; Sajjad et al., 2020) to propose smaller and faster transformer models. Recent work on lottery-ticket hypothesis (Frankle and Carbin, 2018) applied to BERT models (Prasanna et al., 2020; Chen et al., 2020c,d; Yu et al., 2019) has shown promise on finding good performing pruned models. Yet another successful approach for model compression is through quantization of weights and inputs to lower-precisions (Shen et al., 2020; Zafrir et al., 2019).

2.2 Knowledge Distillation

Knowledge distillation (KD) (Hinton et al., 2015) aims to compress the knowledge from a large teacher model to a compact and fast student model. This can be further categorized into task-specific and task-agnostic knowledge distillation. Task specific distillation (Tang et al., 2019; Turc et al., 2019a; Sun et al., 2019a; Chen et al., 2020a) requires to first fine-tune the teacher model for each downstream task and then perform distillation, whereas Task agnostic distillation (Sanh et al., 2019a; Jiao et al., 2020; Wang et al., 2020b; Sun et al., 2020) aims to obtain a general student model for the language modeling task which can directly be finetuned for all downstream tasks.

Generally, in KD the student models are trained with the soft targets obtained from the prediction layer of the teacher model with KL divergence or Mean square error (MSE) used as loss functions. Recent works have also tried aligning the representations across the embedding layer outputs (Sanh et al., 2019a; Jiao et al., 2020), hidden states (Jiao et al., 2020; Sun et al., 2020), self attention distributions (Wang et al., 2020b; Jiao et al., 2020).

2.3 Neural Architecture Search

Neural Architecture Search (Zoph and Le, 2017) aims to automate the design of neural networks by searching through a large space of network topologies and operators. The earlier efforts in computer vision (Zoph and Le, 2017; Liu et al., 2018) followed a costly bi-level optimization involving discrete outer loop search and continuous inner loop weight optimization. Soon, these were replaced with a more effective weight-sharing NAS approach (Cai et al., 2019; Yu et al., 2020; Wang et al., 2021a) that trains a large supernetwork followed by a careful search for optimal subnetworks.

In NLP, (Wang et al., 2020a) applied weight-sharing based NAS in the context of transformers to obtain efficient models for an end-to-end machine translation task. NAS is more challenging for language modeling that involves task-agnostic pretraining and task-specific finetuning. Most NAS efforts in NLP focus on task-specific finetuning (Gao et al., 2021; Chen et al., 2020b). Recent contemporary efforts focus on the challenging search for task-agnostic models by building a super-network using design strategies such as block-wise search, progressive shrinking and stochastic gradient optimization (Xu et al., 2021; Zhang et al., 2021).

In contrast, SuperShaper is a super-pretraining methodology to train a large number of task-agnostic and device-insensitive models in one-shot. The primary benefit of SuperShaper stems from its simple design-space. Many of the model-compression and knowledge-distillation efforts described here are complementary to SuperShaper and can be applied together to gain more benefits. The simplified design space also allows SuperShaper to effectively train a large number of models without requiring sophisticated training and search optimisations like other contemporary efforts (Xu et al., 2021; Zhang et al., 2021).

3 SuperShaper: The methodology

In this section we describe the methodology used in SuperShaper focusing on the backbone network, methods for pretraining and fine-tuning, and for searching subnetworks.

3.1 SuperShaper Backbone

A super pre-training procedure is characterized by a search space of networks. For a Transformer layer in BERT-based networks, different search spaces have been studied, including the number of attention heads, the number of neurons in the FFNs, the number of encoder layers, the use of other operators like separable convolution, etc. As discussed, SuperShaper searches over a single variable - the hidden dimensions for each layer.

In NLP, (Wang et al., 2020a) applied weight-sharing based NAS in the context of transformers to obtain efficient models for an end-to-end machine translation task. NAS is more challenging for language modeling that involves task-agnostic pretraining and task-specific finetuning. Most NAS efforts in NLP focus on task-specific finetuning (Gao et al., 2021; Chen et al., 2020b). Recent contemporary efforts focus on the challenging search for task-agnostic models by building a super-network using design strategies such as block-wise search, progressive shrinking and stochastic gradient optimization (Xu et al., 2021; Zhang et al., 2021).

In contrast, SuperShaper is a super-pretraining methodology to train a large number of task-agnostic and device-insensitive models in one-shot. The primary benefit of SuperShaper stems from its simple design-space. Many of the model-compression and knowledge-distillation efforts described here are complementary to SuperShaper and can be applied together to gain more benefits. The simplified design space also allows SuperShaper to effectively train a large number of models without requiring sophisticated training and search optimisations like other contemporary efforts (Xu et al., 2021; Zhang et al., 2021).

3 SuperShaper: The methodology

In this section we describe the methodology used in SuperShaper focusing on the backbone network, methods for pretraining and fine-tuning, and for searching subnetworks.

3.1 SuperShaper Backbone

A super pre-training procedure is characterized by a search space of networks. For a Transformer layer in BERT-based networks, different search spaces have been studied, including the number of attention heads, the number of neurons in the FFNs, the number of encoder layers, the use of other operators like separable convolution, etc. As discussed, SuperShaper searches over a single variable - the hidden dimensions for each layer. A standard Transformer layer (Vaswani et al., 2017) as visualized in Figure 1 (a) takes as input the representation per token in the sequence, passes it through a multi-headed self-attention module which is followed by an FFN module. Residual connections and layer norms are present after each operation. A

![Figure 1: A Transformer layer in (a) BERT, and (b) Backbone used in SuperShaper with bottleneck matrices.](image-url)
characteristic of such a layer is the size of the representations of each input token, namely the hidden dimension denoted \(d_h\). The hidden dimension captures the amount of information coming into the layer but also characterizes the parameter and computational complexity of that layer. For a typical Transformer layer based on BERT-base with 12 attention heads (with \(d_{\text{attn}}\) dimension), and FFNs (with \(d_{ff}\) dimension), the number of parameters in a layer is given by \(d_h \times (2 \times (2d_{\text{attn}} + d_{ff} + d_h))\) and the number of FLOPs is \(d_h \times (4 \times (2d_{\text{attn}} + d_{ff} + d_h)) + 2 \times \text{seq_len} \times d_{\text{attn}}\).

In a standard BERT-like encoder the hidden dimension \(d_h\) of each layer is a constant, e.g., 768 for BERT-base. But with SuperShaper, we would like to explore sub-networks where layers have different hidden dimensions. The intuition behind this choice is that different layers may perform roles of varying importance. For instance, earlier layers manipulating the input embeddings and the final layers responsible for the output may require larger hidden dimensions. To enable this layer-wise variability of hidden dimensions, we take inspiration from MobileBERT (Sun et al., 2020) which proposed a bottleneck layer to compress the parameter size of BERT. Based on this, we modify the standard Transformer layer as shown in Figure 1 (b). The input and output of each layer are intermediated by multiplication with bottleneck matrices, which translate between the dimension of a token outside a layer (say 768) and the dimension of a token inside a layer (say 120). With this change, each layer can have differently sized bottleneck matrices such that the hidden dimension varies across layers.

For pre-training, we want to create a family of networks with varying shapes. To this end, we take inspiration from slicing matrices for elastic training (Cai et al., 2019). Specifically, each bottleneck matrix is sized at the largest size of 768 × 768. For sampling different sub-networks from this super-network, we slice the bottleneck matrix. For instance, to reduce the dimension of a layer to 120, we slice the bottleneck matrix at the input from 768 × 768 to 768 × 120. In other words, we only retain the first 120 columns in the input bottleneck matrix, and equivalently only the first 120 rows in the bottleneck matrix at the output. By so slicing the prefix columns and rows of the matrix we can generate differently shaped sub-networks. This slicing procedure is implemented with only about 20 lines of additional code in a PyTorch model as detailed in code snippet below. For a sliced matrix both forward and backward passes are similar to a standard neural network.

```python
class CustomLinear(nn.Linear):
    def __init__(self, super_in_dim, super_out_dim, bias=True, uniform=True, non_linear="linear"):
        self.samples = {}
    def set_sample_config(self, sample_in_dim, sample_out_dim):
        sample_weight = weight[:, :sample_in_dim]
        sample_weight = sample_weight[:, :sample_out_dim]
        self.samples["weight"] = sample_weight
        self.samples["bias"] = self.bias[... : self.samples["weight"], ... : self.samples["weight"]]

    def forward(self, x):
        # override the Forward pass to use the sampled weights and bias
        return F.linear(x, self.samples["weight"], self.samples["bias"])
```

### 3.2 Training with SuperShaper

We formalize some notion of the networks. We denote the SuperShaper backbone as \(T\) and any sub-network sliced from the backbone as \(T_S\) where \(S\) is the shape vector that represents the layer-wise hidden dimensions, \(S_i\) for layer \(i\). The set of all possible values of \(S\) denotes the design space \(D\). The smallest and largest sub-networks in \(D\) are denoted as \(T_{S^-}\) and \(T_{S^+}\), respectively, while a random sub-network is denoted as \(T_S\). To evaluate how a sub-network \(T_S\) has trained, we evaluate accuracy on Masked Language Modelling (MLM) task in terms of perplexity denoted \(P(T_S)\). Perplexity represents the uncertainty in predicting the representations of masked tokens in the MLM task.

We now describe the training procedure for SuperShaper: From a given design space \(D\), we sample \(n\) different shapes \(S\) and obtain \(T_S\) for each by the slicing technique described above. This sampling can be performed in two ways: (a) uniform random sampling from \(D\), and (b) random Sampling with sandwich rule (Yu and Huang, 2019), where in addition to (a) we also sample the largest and smallest sub-networks \(T_{S^-}\) and \(T_{S^+}\). Sandwich rule has been shown to perform better for weight-sharing NAS in computer vision (Yu and Huang, 2019; Yu et al., 2020; Wang et al., 2021a). For language modelling, we study both sampling methods and report our findings in Section 4. With the sampled sub-networks, gradient updates are computed and parameters are modified with a standard optimizer. Note that the sub-networks share a large number of their parameters, in particular the earlier rows and columns of the bottleneck matrices. Also matrices inside the layer (such as query, key, and value projection matrices) are shared. This parameter sharing is expected to enable generalization during training across the large space of sub-networks. We evaluate and provide empirical evidence for such generalization in Section 4.

### 3.3 Fine-tuning \(T_S\) from SuperShaper

The pre-training with SuperShaper is task-agnostic. To fine-tune a sampled sub-network \(T_S\) for a specific task, several options exist. First, we can sample \(T_S\) and fine-tune it directly on the task. Second, we can further pre-train \(T_S\) individually and then fine-tune on the task. Finally, we can randomly initialize the weights of \(T_S\) and pre-train from scratch before fine-tuning. We compare these options by fine-tuning on NLU 8 tasks – MNLI-m, QQP, QNLI, CoLA, SST-2, STS-B, RTE, MRPC –
from the GLUE benchmark (Wang et al., 2018). Across these tasks, we report the average performance for the three options as $\overline{G}_{\text{direct}}$, $\overline{G}_{\text{partial}}$, and $\overline{G}_{\text{scratch}}$ respectively.

3.4 Searching for optimal shapes

Once we have super pre-trained with a design space $D$, we can sample and deploy $T_S$ for any $S$, which can then be fine-tuned by methods described in the previous subsection. The set of sub-networks can be large: For instance, for a design space with 7 options for hidden dimensions at 12 layers, we have almost 14 billion sub-networks. The search question is to find an optimal shape from amongst this large set which meets specific constraints on accuracy, parameter count, or latency on devices. We adopt Evolutionary Algorithm (EA) from (Real et al., 2017) as a generic optimization technique, which starts with a population of solutions and over generations performs genetic operations such as mutation and crossover to create new solutions. At each generation the fittest solutions are retained based on defined metrics of fitness. For SuperShaper, the representation of sub-networks and genetic operations are natural since a sub-network is specified by a shape vector, equivalent to a genetic representation of an solution. For the fitness metrics, we use perplexity on language modelling and latency on a device. However, given that computing these metrics for thousands of candidate solutions would be expensive, we use trained predictors to approximate perplexity and latency as been studied elsewhere in NAS (Cai et al., 2019; Ganesan et al., 2020).

While EA with fitness predictors can search for sub-networks, the most desirable setting is to find sub-networks by applying a set of heuristics to decide the shape of each layer. We propose a technique to discover such heuristics and then use it to identify sub-networks for varying parameter count constraints. We report results on how these compare against EAs in Section 4.

4 Experimental Setup and Results

In this section, we detail the experimental setup and report a range of experimental findings to evaluate SuperShaper.

4.1 Experimental Setup

We first describe the pre-training, fine-tuning, and experimental setup used in all our experiments. 

**Design space** We slice the bottleneck matrix to produce Transformer layers of varying hidden dimensions. Our design space for hidden dimensions is $\{120, 240, 360, 480, 540, 600, 768\}$, which creates a design space $D$ of $7^{12}$ or about 14 billion sub-networks.

**Super-pretraining** We initialize our backbone with BERT-base-cased model trained on Wikipedia and BookCorpus. We then super pre-train the backbone using Masked language modeling over the C4 RealNews dataset (Raffel et al., 2019) with batch size 2048, max sequence length 128, for 175K steps (or 26 epochs) on 8 A100 GPUs. The complete set of hyperparameters used is shown in Table 1.

| Data                      | C4/RealNews |
|---------------------------|-------------|
| Max sequence length       | 128         |
| Batch size                | 2048        |
| Peak learning rate        | 2e-5        |
| Number of steps           | 175K        |
| Warmup steps              | 10K         |
| Hidden dropout            | 0           |
| GeLU dropout              | 0           |
| Attention dropout         | 0           |
| Learning rate decay       | Linear      |
| Optimizer                 | AdamW       |
| Adam $\epsilon$           | 1e-6        |
| Adam $(\beta_1, \beta_2)$ | (0.9, 0.999) |
| Weight decay              | 0.01        |
| Gradient clipping         | 0           |

Table 1: Hyperparameters for MLM super pre-training on C4 RealNews. Super pre-training was done on 8 A100 GPUs

**Fine tuning.** Similar to (Xu et al., 2021), we evaluate the effectiveness of SuperShaper by pre-training all our compressed models from scratch and later fine-tuning them on 8 GLUE tasks. We report performance metrics on the dev version of the benchmark. For RTE, MRPC and STS-B, we start with a model fine-tuned on MNLI similar to (Liu et al., 2019; Xu et al., 2021). For metrics, we report Matthews correlation for CoLA (Wang et al., 2018), Spearman correlation for STS-B (Wang et al., 2018) and accuracy for all other tasks. For MNLI-m (Wang et al., 2018), we report accuracy on the matched set. Following (Devlin et al., 2019; Xu et al., 2021; Zhang et al., 2021; Hou et al., 2020), we also exclude the problematic WNLI dataset. For all the datasets in GLUE, we use the official train and dev splits and download the datasets from HuggingFace datasets\(^1\). The complete set of hyperparameters for fine-tuning can be found in Table 2.

**Evolutionary Algorithm (EA).** For EA, we adapt the algorithm presented in (Real et al., 2017). We choose a population size of 100, mutation probability of 0.4, and the ratio of parent size to mutation or crossover size as 1. We bound the search algorithm to 300 iterations.

**Fitness Predictors.** For perplexity predictor, we randomly sample 10,000 sub-networks and evaluate their perplexity as measured on C4-RealNews dataset. We use this dataset to build the predictor based on XGBoost model (Chen et al., 2015). For latency predictor, we sample 1,000 – 4,000 sub-networks and evaluate their latency on the chosen device. We again train a XGBoost model to predict latency from this dataset. We consider 3 devices - 2 GPUs 1080Ti and K80, and a server class single-core Xeon CPU. The GPUs were

\(^1\)https://huggingface.co/datasets/glue
Table 2: Hyperparameters for fine-tuning on GLUE

|                     | CoLA | Other GLUE tasks |
|---------------------|------|------------------|
| Batch size          | 16, 32 | 32 |
| Weight decay        | 0, 0.1 | 0 |
| Warmup steps        | 0, 400 | 0 |
| Max sequence length | 128 |                  |
| Peak learning rate  | 5e-5 |                  |
| Number of epochs    | 10 |                  |
| Hidden dropout      | 0 |                  |
| GeLU dropout        | 0 |                  |
| Attention dropout   | 0 |                  |
| Learning rate decay | Linear |                |
| Optimizer           | AdamW |                |
| Adam $\epsilon$     | 1e-6 |                |
| Adam ($\beta_1$, $\beta_2$) | (0.9, 0.999) |              |
| Gradient clipping   | 0 |                  |

evaluated with a batch size of 128 while the CPU was evaluated with batch size 1.

4.2 Pre-training with SuperShaper

![Figure 2: The perplexity trajectory of BERT-base and SuperShaper backbone on MLM task for C4 Real News](image)

Training the backbone.

We first pre-train our proposed backbone network that is based on BERT-base but with additional bottleneck matrices on the C4-RealNews dataset. We initialize the bottleneck structures with identity weights and zero-bias. The corresponding plot of perplexity on language modelling is for BERT-base and our backbone are shown in Section 4.2. The SuperShaper backbone mimics the performance of BERT-base thereby providing a good starting point for pre-training of multiple sub-networks with SuperShaper.

Effect of sub-network sampling rule.

In computer vision, sandwich rule is widely used in the context of weight-sharing NAS (Yu and Huang, 2019). We super pre-train the trained backbone network with the sandwich rule. The corresponding loss trajectory for largest, smallest, and randomly sampled sub-networks are shown in Figure 3(a). Clearly, the larger network has a lower perplexity, but the super pre-training ensures that a range of networks are simultaneously trained on the MLM task. Specifically, randomly sampled subnetworks shown as $T_{S}$, even though not sampled as frequently as the smallest subnetwork, have a lower perplexity. This provides evidence of generalization during super pre-training.

We now compare the sandwich sampling rule with fully randomised sampling. We plot the perplexity of 4 networks: the largest, smallest, and two other intermediate networks in Figure 3(b)-(e). Sandwich sampling always samples the largest and smallest and thus the perplexity on these networks is significantly lower with sandwich sampling than random sampling. On the other two networks, sandwich sampling is able to tightly match the perplexity obtained with random sampling, even though stochastically sandwich rule has fewer gradient updates to these two networks than random sampling. This suggests that sandwich sampling effectively combines good extremum sub-networks with reasonably good intermediate sub-networks. In all subsequent experiments, we use sandwich sampling.

Visualizing bottleneck matrices.

We initialize the bottleneck matrices to identity. After super pre-training, we visualize these matrices to understand the role of sliced training of sub-networks. We take the softmax of the principal diagonal of the two bottleneck matrices, and plot them for all layers in Figure 4. We clearly observe that the entries show a banded pattern with boundaries at the shapes in our design space: 120, 240, 360, 480, 540, 600, and 768. This implies that super pre-training learns different linear projections of 768 dimensional input representation to the chosen hidden dimensions.

Effectiveness of super pre-training.

We ask two questions towards evaluating the effectiveness of super pre-training: (a) Is the relative performance of sampled sub-networks on the MLM perplexity ($\overline{G}_{\text{direct}}$) correlated with performance of the same sub-networks when pre-trained individually from scratch ($\overline{G}_{\text{scratch}}$), and (b) Does the super pre-training afford sub-networks an advantage when being fine-tuned for tasks? For the first question, we sample a set of sub-networks $T_S$ of both varying (33-96M) and similar (63-65M) parameter counts, and plot $\overline{G}_{\text{direct}}$ and $\overline{G}_{\text{scratch}}$ in Figure 5 (a). We notice that $\overline{G}_{\text{direct}}$ and $\overline{G}_{\text{scratch}}$ are highly correlated with a Spearman correlation coefficient of 0.954. This implies that the sub-network’s measured MLM perplexity after super pre-training is a good proxy for final performance. We also observe that networks sampled at the similar parameter count (63-65M) have varying performance suggesting the sensitive role of shape in accuracy.

For studying the second question, we pre-train and then fine-tune the varying parameter count sub-
Figure 3: (a) Loss trajectory of $T^+_S$, $T^-_S$ and $T_S$ networks, (b)-(d) Perplexity trajectory of $T^-_S$, two randomly sampled $T^-_S$, and $T^+_S$ respectively for random-sampling and sandwich rule

Figure 4: Visualization of input and output Bottleneck matrix diagonals for all the 12 layers.

networks (33-96M) in two ways (a) by retaining the weights learnt during super pre-training ($G_{\text{partial}}$), and (b) starting with random initialization $G_{\text{scratch}}$. We plot two quantities in Figure 5 (b): the amount of pre-training time saved with (a) and the additional GLUE score obtained with (a). We observe that models with fewer parameters (30-50M) show significant savings in the pre-training time (up to 6.6×) and simultaneously benefit from improved GLUE accuracy (up to 3%). The gains on both axes for larger models are smaller. This suggests that smaller models whose parameters receive more weight updates due to sharing of the earlier rows and columns across sub-networks benefit more from super pre-training. This is encouraging because most effort in deployability is concerned with models of smaller size.

4.3 Comparing sub-networks with other methods

Comparing with BERT-base.
As a first baseline, we search for a SuperShaper-Base model with EA with a constraint of 100M parameters and obtain a model with 96M parameters. This model is comparable against uncompressed BERT-base model which has 110M parameters. We compare the GLUE performance $G_{\text{scratch}}$ of SuperShaper-Base with two of the top reported results on BERT-Base (Xu et al., 2021; Sanh et al., 2019b). From Table 3, we find that the average GLUE score across the two reported BERT-base baselines (84.4% and 83%) is similar to SuperShaper-Base (83.7%). Thus, SuperShaper-Base performs competitively with the uncompressed BERT-base with fewer parameters (96M vs 110M).

Comparing with compressed models
We now compare against state-of-the-art compressed models either hand-crafted or found by NAS algorithms (see Table 4). Since several of these models are in the range of 60-67M, we search for a sub-network from SuperShaper with a parameter constraint of 66M. The task-wise performance of the obtained sub-network is reported in Table 4. SuperShaper outperforms many prominent hand-crafted or compressed networks proposed over the last two years by a significant margin. Across NAS-based methods, SuperShaper performs competitively despite a much simpler design space. Only NAS-BERT reports a higher accuracy, which may be attributed to the use of novel operators such as separable convolution in the design space. Also, NAS-BERT and DynaBERT use knowledge distillation and data augmentation. These methods are orthogonal to shaping and can be combined with our approach. In summary, we establish that SuperShaper with a simple design space and efficient super pre-training implementation performs competitively in compressing models to a given parameter count.
Figure 5: (a) SuperShaper provides a fast and accurate proxy for sub-network perplexity, and (b) \(G_{\text{partial}}\) inherited sub-networks only require fraction of pre-training cost (in blue) i.e. 1.3-6.6x reduction to reach optimum. This comes at a higher average gain in GLUE score (in red).

| Model | Params | MNLI-m | QQP | QNLI | CoLA | SST-2 | STS-B | RTE | MRPC | Average GLUE |
|-------|--------|--------|-----|------|------|-------|-------|-----|------|--------------|
| BERT-Base (from NAS-BERT) | 110M | 85.2 | 91 | 91.3 | 61 | 92.9 | 90.3 | 76 | 87.7 | 84.4 |
| BERT-Base (from DistilBERT) | 110M | 86.7 | 89.6 | 91.8 | 56.3 | 92.7 | 89 | 69.3 | 88.6 | 83 |
| SuperShaper-Base | 96M | 83.9 | 90.86 | 90.92 | 56.58 | 92.89 | 88.3 | 77.98 | 88.48 | 83.7 |

Table 3: Comparing SuperShaper with BERT-Base models.

| Model | Params | MNLI-m | QQP | QNLI | CoLA | SST-2 | STS-B | RTE | MRPC | Average GLUE |
|-------|--------|--------|-----|------|------|-------|-------|-----|------|--------------|
| LayerDrop (Fan et al., 2019) | 66M | 80.7 | 88.3 | 88.4 | 45.4 | 90.7 | - | 65.2 | 85.9 | 77.8 |
| DistilBERT (Sanh et al., 2019b) | 66M | 82.2 | 88.5 | 89.2 | 51.3 | 91.3 | 86.9 | 59.9 | 87.5 | 79.6 |
| Bert-PKD (Sun et al., 2019b) | 66M | 81.5 | 70.7 | 89.0 | - | 92.0 | - | 65.5 | 85.0 | 80.6 |
| MiniLM (Wang et al., 2020b) | 66M | 84.0 | 91.0 | 91.0 | 49.2 | 92.0 | - | 71.5 | 88.4 | 81.0 |
| Ta-TinyBERT (Jiao et al., 2020) | 67M | 83.5 | 90.6 | 90.5 | 42.8 | 91.6 | 86.5 | 72.2 | 88.4 | 80.8 |
| Tiny-BERT (Jiao et al., 2020) | 66M | 84.6 | 89.1 | 90.4 | 51.1 | 93.1 | 83.7 | 70.0 | 82.6 | 80.6 |
| BERT-of-Theseus (Xu et al., 2020) | 66M | 82.3 | 89.6 | 89.5 | 51.1 | 91.5 | 88.7 | 68.2 | - | 80.1 |
| PD-BERT (Turc et al., 2019b) | 66M | 82.5 | 90.7 | 89.4 | - | 91.1 | - | 66.7 | 84.9 | 84.2 |
| ELM (Jiao et al., 2021) | 60M | 84.2 | 91.1 | 90.8 | 54.2 | 92.7 | 88.9 | 72.2 | 89.0 | 82.9 |
| NAS-BERT+ (Xu et al., 2021) | 60M | 83.3 | 90.9 | 91.3 | 55.6 | 92.0 | 88.6 | 78.5 | 87.5 | 83.5 |
| DynaBERT† (Hou et al., 2020) | 60M | 84.2 | 91.2 | 91.5 | 56.8 | 92.7 | 89.2 | 72.2 | 84.1 | 82.8 |
| YOCO-bert (Zhang et al., 2021) | 59-67M | 82.6 | 90.5 | 87.2 | 59.8 | 92.8 | - | 72.9 | 90.3 | 82.3 |
| SuperShaper evo-search (ours) | 63M | 82.2 | 90.2 | 88.1 | 53.0 | 91.9 | 87.6 | 79.1 | 89.5 | 82.7 |
| SuperShaper heuristic-shaped (ours) | 61M | 82 | 90.3 | 88.4 | 52.6 | 91.6 | 87.8 | 77.6 | 86.5 | 82.1 |

Table 4: Comparison of SuperShaper with 60-67M parameter constraint models on development set of GLUE. † indicates models trained with data augmentation, * indicates model trained without knowledge distillation in the fine-tuning stage.

| Model | Params (M) | \(G_{\text{partial}}\) | \(G_{\text{scratch}}\) | MNLI-m | QQP | QNLI | CoLA | SST-2 | STS-B | RTE | MRPC | Average GLUE |
|-------|-------------|-----------------|--------------|--------|-----|------|------|-------|-------|-----|------|--------------|
| 33    | 10.82       | 12.44           | 73.45       | 84.71 | 80.52 | 10.27 | 85.32 | 82.65 | 65.70 | 82.11 | 70.59 |
| 53    | 8.59        | 6.02            | 79.40       | 89.51 | 86.38 | 33.85 | 89.11 | 86.66 | 68.23 | 84.56 | 77.21 |
| 63    | 7.09        | 4.55            | 82.23       | 90.18 | 88.05 | 53.00 | 91.86 | 87.63 | 79.06 | 89.46 | 82.68 |
| 69    | 6.62        | 4.28            | 82.74       | 90.45 | 89.54 | 54.98 | 91.28 | 88.42 | 77.98 | 87.75 | 82.89 |
| 80    | 6.17        | 4.02            | 83.05       | 90.56 | 89.22 | 54.87 | 93.10 | 88.46 | 80.14 | 87.75 | 83.39 |
| 90.5  | 5.83        | 3.79            | 83.06       | 90.51 | 88.72 | 58.87 | 91.51 | 88.47 | 77.26 | 88.97 | 83.42 |
| 96    | 5.65        | 3.73            | 83.90       | 90.86 | 90.92 | 56.58 | 92.89 | 88.30 | 77.98 | 88.48 | 83.74 |

Table 5: Performance of best models from parameter-constrained evolutionary search.

Models for various parameter count constraints.
We now apply EA to search for sub-networks at varying parameter count, ranging from 40 to 110M. To understand the effectiveness of EA search, we sample 10,000 random sub-networks and compute their perplexity. We then plot these points along with the
### Table 6: Hidden dimensions of templatized shapes and their corresponding perplexities for $G_{\text{scratch}}$ and $G_{\text{direct}}$.  

| Shapes               | Params (M) | $G_{\text{direct}}$ | $G_{\text{scratch}}$ | L1 | L2 | L3 | L4 | L5 | L6 | L7 | L8 | L9 | L10 | L11 | L12 |
|----------------------|------------|----------------------|----------------------|----|----|----|----|----|----|----|----|----|----|-----|-----|-----|
| Evo Search 1         | 65         | 6.86                 | 4.45                 | 480| 360| 360| 240| 240| 360| 480| 360| 480| 360 | 480 | 540 |
| Evo Search 2         | 63         | 7.09                 | 4.55                 | 480| 240| 360| 240| 360| 540| 480| 360| 360 | 360 | 360 | 480 |
| Lower Triangle       | 64         | 7.31                 | 4.67                 | 120| 120| 360| 360| 360| 360| 480| 480| 360 | 600 | 768 |
| Random               | 64         | 7.49                 | 4.91                 | 480| 360| 360| 360| 360| 360| 360| 480| 540 | 360 | 360 | 540 |
| Rectangle            | 58         | 7.5                  | 4.72                 | 360| 360| 360| 360| 360| 360| 360| 360| 360 | 360 | 360 | 360 |
| Inverted Diamond     | 65         | 8.12                 | 4.93                 | 768| 600| 360| 240| 240| 120| 120| 240| 240 | 360 | 600 | 768 |
| Bottle               | 64         | 8.31                 | 4.9                  | 120| 120| 120| 120| 120| 120| 120| 120| 120 | 120 | 120 | 120 |
| Diamond              | 64         | 8.36                 | 5.13                 | 768| 600| 540| 480| 480| 360| 768| 540 | 480 | 360 | 240 | 240 |
| Upper Triangle       | 64         | 8.43                 | 5.16                 | 600| 540| 480| 360| 360| 360| 360| 480| 540 | 360 | 360 | 360 |
| Inverted Bottle      | 64         | 9.22                 | 5.37                 | 768| 600| 360| 600| 540| 360| 480| 540 | 360 | 120 | 120 | 120 |

Table 7: Shape difference positively correlates with $G_{\text{direct}}$ difference across a wide parameter range

$$\text{Diff} = \|S_1 - S_2\|$$ (1)

Models with templatized shapes.

We evaluate the performance of the following templatized shapes in the 63-65M parameter range: lower triangle, upper triangle, rectangle, diamond, inverted diamond, bottle, and inverted bottle. Details of the hidden dimensions and sub-network perplexity for each network are reported in Table 6. We observe that lower triangle has the lowest perplexity (7.31) while inverted bottle (9.22) has the highest. This wide range reiterates that shape sensitively affects performance. Further, we observe that larger number of parameters towards the later layers benefits model performance.

**Shape difference vs performance.** To further study the effect of shape on performance, we test if the shape difference between random subnetworks and an optimal subnetwork (determined by evolutionary search) in the same parameter range, correlates with their differences in performance. The shape difference between two subnetworks with shapes $S_1$ and $S_2$ and their respective difference in performance ($G_{\text{direct}}$) is characterised by their L2 norm of their differences:

$$\text{Diff} = \|S_1 - S_2\|$$ (1)

We choose points across different parameter ranges (50-100M) from the 10,000 random sampled subnetworks from section Section 4 and compute their shape and performance differences with the optimal evolutionary-search model. The Spearman and Pearson correlation coefficient (Myers and Sirois, 2004; Benesty et al., 2009) across the shape and performance L2

Figure 6: Evolutionary search finds accurate models, while models based on simple heuristics perform competitively.

4.4 Shape analysis of Super-Networks.

In contrast to other NAS techniques, the design space of SuperShaper is interpretable - the network shape. We can thus ask the question: Are there good shapes across sizes?
we derive a set of heuristics layers and the first layer. Based on these observations, layer’s shape and find it to be highest for the last few compute the feature importance (see Figure 7(a)) of each heuristically shaped models. Based on the above patterns of what is a good shape: (a) a large dimension in the last layer, (b) moderately large dimension in the first layer, (c) low dimensions in early middle layers (2-5), and (d) moderate dimensions in later middle layers (6-11). We characterize this as a cigar-like shape.

**Heuristically shaped models.** Based on the above heuristics, we hand-shape sub-networks with the following algorithm: (a) construct a reference model $T_S$, following the heuristics at a given parameter range (60-65M), (b) for a target parameter count, scale the shape $S_i$ of every layer linearly, (c) for early middle layers, round down the scaled $S_i$ and for remaining layers round up $S_i$ to the nearest configuration in $D$. Based on this algorithm, we identify sub-networks across the parameter count with cigar-like shapes as shown in Figure 10. We evaluate these hand-crafted sub-networks on perplexity and find that they are competitive and even outperform sub-networks searched with EAs (see Figure 6). We also pre-train and evaluate one of the heuristic models with a parameter count of 61M on the Glue tasks (see Table 4). We observe that, similar to our evolutionary-search subnetwork (63M), the heuristic model outperforms prominent hand-crafted or compressed networks and only lags behind NAS-BERT and DynaBERT. This strongly demonstrates the generalization of the derived heuristics across model size. To the best of our knowledge, this is the first such generalization demonstrated for NAS.

**Feature importance from optimal sub-networks.**

From the analysis of sub-networks searched by EA and the templated shapes, we find that accurate networks have more parameters in later layers. We analyse this using the perplexity predictor trained to estimate $\mathcal{G}_\text{partial}$ given the shape. For this predictor, we compute the feature importance (see Figure 7(a)) of each layer’s shape and find it to be highest for the last few layers and the first layer. Based on these observations, we derive a set of heuristics of what is a good shape: (a) a large dimension in the last layer, (b) moderately large dimension in the first layer, (c) low dimensions in early middle layers (2-5), and (d) moderate dimensions in later middle layers (6-11). We characterize this as a cigar-like shape.

**4.5 Device-specific efficient models.**

We now discuss searching for sub-networks based on latency on a device. We train latency predictors for 3 devices - 1080Ti, K80, and a Xeon PC with high $R^2$ values of 0.993, 0.988 and 0.892 respectively as shown by the fitness curves in Figure 9. The feature importance of these predictors strongly favours total parameters (96.5%) and only very weakly depends on layer dimensions as shown in Figure 7(b),(c) and (d). This is a crucial insight: the shape of the network for a given parameter count is a free variable that can be optimized for accuracy. Thus for deployment on a device, we need to identify the right parameter count that meets the latency constraint while the shape can be identified with EA or the heuristics we have laid out.

We run EA under two settings - parameter constraints and latency constraints for all devices. We also evaluate the hand-crafted models. The latency and perplexity of these models are shown in Figure 8. As can be seen, all three techniques result in similar performance. This corroborates that latency is insensitive to shape and that the heuristics identify competitive networks.

In summary, we showed that SuperShaper effectively generalizes training across sub-networks, and finds competitive networks at various sizes. This training on language models enables generalization across tasks. Further we derived a set of simple rules to shape a network which is competitive with EA search, thereby easily generalizing the search across model size. And finally we established that latency on devices is insensitive to shapes and thus EA search on parameter count or hand-crafted networks generalize across devices. Thus, with a simple and effective superpretraining procedure we identify sub-networks that generalize across tasks, model sizes, and devices.

**5 Conclusions and Future Work**

To address the problem of deploying NLU models across a range of devices, we propose SuperShaper a NAS technique to pretrain language models by shaping Transformer layers. SuperShaper identifies networks that outperform state-of-the-art model compres-
Figure 8: Perplexity vs Latency for optimal models searched using EA with parameter and latency constrained and for heuristically shaped models across: (a) 1080Ti GPU, (b) Xeon CPU, and (c) K80 GPU

Figure 9: The latency predictors are very accurate with $R^2$ scores of 0.993, 0.988, and 0.892 respectively.

Figure 10: Heuristically shaped models have a cigar-like shape.

Figure 11: Heuristically shaped models have a cigar-like shape

Figure 12: Heuristically shaped models have a cigar-like shape

Figure 13: Heuristically shaped models have a cigar-like shape

Figure 14: Heuristically shaped models have a cigar-like shape

Figure 15: Heuristically shaped models have a cigar-like shape

Figure 16: Heuristically shaped models have a cigar-like shape

Figure 17: Heuristically shaped models have a cigar-like shape

Figure 18: Heuristically shaped models have a cigar-like shape

Figure 19: Heuristically shaped models have a cigar-like shape

Figure 20: Heuristically shaped models have a cigar-like shape

6 Acknowledgements

We thank AI4Bharat and Robert Bosch Centre for Data Science and Artificial Intelligence, IIT Madras for their help with compute resources for this project. We thank wandb (Biewald, 2020) for providing a very useful experiment tracking tool that helped accelerate our progress. We thank Vamsi Krishna for useful discussions in the early stages of this project.

References

Jacob Benesty, Jingdong Chen, Yiteng Huang, and Israel Cohen. 2009. Pearson correlation coefficient. In Noise reduction in speech processing, pages 1–4. Springer.

Lukas Biewald. 2020. Experiment tracking with weights and biases. Software available from wandb.com.

Andrew Brock, Theo Lim, JM Ritchie, and Nick Weston. 2018. Smash: One-shot model architecture search through hypernetworks. In International Conference on Learning Representations.

Han Cai, Chuang Gan, Tianzhe Wang, Zhekai Zhang, and Song Han. 2019. Once-for-all: Train one network and specialize it for efficient deployment. In International Conference on Learning Representations.

Daoyuan Chen, Yaliang Li, Minghui Qiu, Zhen Wang, Bofang Li, Bolin Ding, Hongbo Deng, Jun Huang, Wei Lin, and Jingren Zhou. 2020a. Adabert: Task-adaptive bert compression with differentiable neural architecture search. In Proceedings of the Twenty-Ninth International Joint Conference on Artificial Intelligence, IJCAI-20, pages 2463–2469. International Joint Conferences on Artificial Intelligence Organization. Main track.

Daoyuan Chen, Yaliang Li, Minghui Qiu, Zhen Wang, Bofang Li, Bolin Ding, Hongbo Deng, Jun Huang, Wei Lin, and Jingren Zhou. 2020b. Adabert: Task-adaptive bert compression with differentiable neural architecture search. Cell, 2(3):4.
Tianlong Chen, Jonathan Frankle, Shiyu Chang, Sijia Liu, Yang Zhang, Zhangyang Wang, and Michael Carbin. 2020c. The lottery ticket hypothesis for pre-trained bert networks. arXiv preprint arXiv:2007.12223.

Tianqi Chen, Tong He, Michael Benesty, Vadim Khotilovich, Yuan Tang, Hyunsu Cho, et al. 2015. Xgboost: extreme gradient boosting. R package version 0.4-2, 1(4):1–4.

Xiaoan Chen, Yu Cheng, Shuohang Wang, Zhe Gan, Zhangyang Wang, and Jingjing Liu. 2020d. Early-bert: Efficient bert training via early-bird lottery tickets. arXiv preprint arXiv:2101.00063.

Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. Bert: Pre-training of deep bidirectional transformers for language understanding. In NAACL-HLT (1).

Angela Fan, Edouard Grave, and Armand Joulin. 2019. Reducing transformer depth on demand with structured dropout. arXiv preprint arXiv:1909.11556.

Jonathan Frankle and Michael Carbin. 2018. The lottery ticket hypothesis: Finding sparse, trainable neural networks. arXiv preprint arXiv:1803.03635.

Vinod Ganesan, Surya Selvam, Sanchari Sen, Pratyush Kumar, and Anand Raghunathan. 2020. A case for generalizable dnn cost models for mobile devices. In 2020 IEEE International Symposium on Workload Characterization (IISWC), pages 169–180. IEEE.

Jiahui Gao, Hang Xu, Xiaozhe Ren, Philip LH Yu, Xiaodan Liang, Xin Jiang, Zhenguo Li, et al. 2021. Autobert-zero: Evolving bert backbone from scratch. arXiv preprint arXiv:2107.07445.

Saurabh Goyal, Anamitra Roy Choudhury, Saurabh Raje, Venkatesan Chakaravarthy, Yogish Sabharwal, and Ashish Verma. 2020. Power-bert: Accelerating bert inference via progressive word-vector elimination. In International Conference on Machine Learning, pages 3690–3699. PMLR.

Geoffrey E. Hinton, Oriol Vinyals, and J. Dean. 2015. Distilling the knowledge in a neural network. ArXiv, abs/1503.02531.

Lu Hou, Zhiqi Huang, Lifeng Shang, Xin Jiang, Xiao Chen, and Qun Liu. 2020. Dynabert: Dynamic bert with adaptive width and depth. In Advances in Neural Information Processing Systems, volume 33, pages 9782–9793. Curran Associates, Inc.

Xiaoe Jiao, Huating Chang, Yichun Yin, Lifeng Shang, Xin Jiang, Xiao Chen, Linlin Li, Fang Wang, and Qun Liu. 2021. Improving task-agnostic bert distillation with layer mapping search. Neurocomputing, 461:194–203.

Xiaoe Jiao, Yichun Yin, Lifeng Shang, Xin Jiang, Xiao Chen, Linlin Li, Fang Wang, and Qun Liu. 2020. TinyBERT: Distilling BERT for Natural Language Understanding. In Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: Findings.

Sehoon Kim, Sheng Shen, David Thorsley, Amir Gholami, Joseph Hassoun, and Kurt Keutzer. 2021. Learned token pruning for transformers. arXiv preprint arXiv:2107.00910.

Hanxiao Liu, Zihang Dui, David R. So, and Quoc V. Le. 2021. Pay attention to mlps. ArXiv, abs/2105.08050.

Hanxiao Liu, Karen Simonyan, and Yiming Yang. 2018. Darts: Differentiable architecture search. In International Conference on Learning Representations.

Yinhan Liu, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, Mike Lewis, Luke Zettlemoyer, and Veselin Stoyanov. 2019. Roberta: A robustly optimized bert pretraining approach. arXiv preprint arXiv:1907.11692.

Xindian Ma, Peng Zhang, Shuai Zhang, Nan Duan, Yuexiao Hou, Ming Zhou, and Dawei Song. 2019. Are sixteen heads really better than one? arXiv preprint arXiv:1905.10650.

Leann Myers and Maria J Sirois. 2004. Spearman correlation coefficients, differences between. Encyclopedia of statistical sciences, 12.

Sai Prasanna, Anna Rogers, and Anna Rumshisky. 2020. When bert plays the lottery, all tickets are winning. arXiv preprint arXiv:2005.00561.

Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J Liu. 2019. Exploring the limits of transfer learning with a unified text-to-text transformer. arXiv preprint arXiv:1910.10683.

Esteban Real, Sherry Moore, Andrew Selle, Saurabh Saxena, Yutaka Leon Suematsu, Jie Tan, Quoc V Le, and Alexey Kurakin. 2017. Large-scale evolution of image classifiers. In International Conference on Machine Learning, pages 2902–2911. PMLR.

Hassan Sajjad, Fahim Dalvi, Nadir Durrani, and Preslav Nakov. 2020. Poor man’s bert: Smaller and faster transformer models. arXiv e-prints, pages arXiv–2004.

Victor Sanh, Lysandre Debut, Julien Chaumond, and Thomas Wolf. 2019a. Distilbert, a distilled version of bert: smaller, faster, cheaper and lighter. ArXiv, abs/1910.01108.

Victor Sanh, Lysandre Debut, Julien Chaumond, and Thomas Wolf. 2019b. Distilbert, a distilled version of bert: smaller, faster, cheaper and lighter. arXiv preprint arXiv:1910.01108.
Sheng Shen, Zhen Dong, Jiayu Ye, Linjian Ma, Zhewei Yao, Amir Gholami, Michael W Mahoney, and Kurt Keutzer. 2020. Q-bert: Persian based ultra low precision quantization of bert. In Proceedings of the AAAI Conference on Artificial Intelligence, volume 34, pages 8815–8821.

Siqi Sun, Yu Cheng, Zhe Gan, and Jingjing Liu. 2019a. Patient knowledge distillation for BERT model compression. In Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP), pages 4323–4332, Hong Kong, China. Association for Computational Linguistics.

Siqi Sun, Yu Cheng, Zhe Gan, and Jingjing Liu. 2019b. Patient knowledge distillation for bert model compression. In Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP), pages 4323–4332.

Zhiqing Sun, Hongkun Yu, Xiaodan Song, Renjie Liu, Yiming Yang, and Denny Zhou. 2020. Mobile-BERT: a compact task-agnostic BERT for resource-limited devices. In Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics, pages 2158–2170, Online. Association for Computational Linguistics.

Raphael Tang, Yao Lu, L. Liu, Lili Mou, Olga Vechtomova, and Jimmy J. Lin. 2019. Distilling task-specific knowledge from bert into simple neural networks. ArXiv, abs/1903.12136.

Iulia Turc, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019a. Well-read students learn better: The impact of student initialization on knowledge distillation. ArXiv, abs/1908.08962.

Iulia Turc, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019b. Well-read students learn better: The impact of student initialization on knowledge distillation. arXiv preprint arXiv:1908.08962, 13.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. In Advances in neural information processing systems, pages 5998–6008.

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

Dilin Wang, Meng Li, Chengyue Gong, and Vikas Chandra. 2021a. Attentionenas: Improving neural architecture search via attentive sampling. In Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition, pages 6418–6427.

Hanrui Wang, Zhanghao Wu, Zhijian Liu, Han Cai, Ligeng Zhu, Chuang Gan, and Song Han. 2020a. Hat: Hardware-aware transformers for efficient natural language processing. arXiv preprint arXiv:2005.14187.

Hanrui Wang, Zhekai Zhang, and Song Han. 2021b. Spatten: Efficient sparse attention architecture with cascade token and head pruning. In 2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA), pages 97–110, IEEE.

Wenhui Wang, Furu Wei, Li Dong, Hangbo Bao, Nan Yang, and Ming Zhou. 2020b. Minilm: Deep self-attention distillation for task-agnostic compression of pre-trained transformers. In Advances in Neural Information Processing Systems.

Ziheng Wang, Jeremy Wohlwend, and Tao Lei. 2019. Structured pruning of large language models. arXiv preprint arXiv:1910.04732.

Canwen Xu, Wanchunshu Zhou, Tao Ge, Furu Wei, and Ming Zhou. 2020. BERT-of-Theseus: Compressing BERT by Progressive Module Replacing. In Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP), pages 7859–7869.

Jin Xu, Xu Tan, Renqian Luo, Kaitao Song, Jian Li, Tao Qin, and Tie-Yan Liu. 2021. Nas-bert: Task-agnostic and adaptive-size bert compression with neural architecture search. In Proceedings of the 27th ACM SIGKDD Conference on Knowledge Discovery & Data Mining.

Haonan Yu, Sergey Edunov, Yuandong Tian, and Ari S Morcos. 2019. Playing the lottery with rewards and multiple languages: lottery tickets in rl and nlp. In International Conference on Learning Representations.

Jiahui Yu and Thomas S Huang. 2019. Universally slimmable networks and improved training techniques. In Proceedings of the IEEE/CVF International Conference on Computer Vision, pages 1803–1811.

Jiahui Yu, Pengchong Jin, Hanxiao Liu, Gabriel Bender, Pieter-Jan Kindermans, Mingxing Tan, Thomas Huang, Xiaodan Song, Ruoming Pang, and Quoc Le. 2020. Bignas: Scaling up neural architecture search with big single-stage models. In European Conference on Computer Vision, pages 702–717, Springer.

Ofrir Zafrir, Guy Boudoukh, Peter Izsak, and Moshe Wasserblat. 2019. Q8bert: Quantized 8bit bert. arXiv preprint arXiv:1910.06188.

Shaojun Zhang, Xiawu Zheng, Chenyi Yang, Yuchao Li, Yan Wang, Fei Chao, Mengdi Wang, Shen Li, Jun Yang, and Rongrong Ji. 2021. You only compress once: Towards effective and elastic bert compression via exploit-explore stochastic nature gradient. arXiv preprint arXiv:2106.02435.
Barret Zoph and Quoc V. Le. 2017. Neural architecture search with reinforcement learning. In 5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings. OpenReview.net.