Bespoke: A Block-Level Neural Network Optimization Framework for Low-Cost Deployment

Jong-Ryul Lee[1], Yong-Hyuk Moon[1,2]*

[1] Electronics and Telecommunications Research Institute (ETRI), Daejeon, Korea
[2] University of Science and Technology (UST), Daejeon, Korea
{jongryul.lee,yhmoon}@etri.re.kr

Abstract

As deep learning models become popular, there is a lot of need for deploying them to diverse device environments. Because it is costly to develop and optimize a neural network for every single environment, there is a line of research to search neural networks for multiple target environments efficiently. However, existing works for such a situation still suffer from requiring many GPUs and expensive costs. Motivated by this, we propose a novel neural network optimization framework named Bespoke for low-cost deployment. Our framework searches for a lightweight model by replacing parts of an original model with randomly selected alternatives, each of which comes from a pretrained neural network or the original model. In the practical sense, Bespoke has two significant merits. One is that it requires near zero cost for designing the search space of neural networks. The other merit is that it exploits the sub-networks of public pretrained neural networks, so the total cost is minimal compared to the existing works. We conduct experiments exploring Bespoke's the merits, and the results show that it finds efficient models for multiple targets with meager cost.

Introduction

Due to the great success of deep learning, neural networks are deployed into various environments such as smartphones, edge devices, and so forth. In addition, lots of techniques have been proposed for optimizing neural networks for fast inference and small memory usage (Park et al. 2020; Liu et al. 2021; Luo and Wu 2020; Yu, Mazaheri, and Jannesari 2021; Lee et al. 2020; Peng et al. 2019; Hinton, Vinyals, and Dean 2015; Yim et al. 2017).

One of the major issues about optimizing neural networks for efficient deployment is that it is not cheap to search for an optimized model for even a single target environment. If the target environment is changed, we need to search for another optimized model again. That is, if we have 100 target environments, we have to do optimization 100 times separately. This situation is not preferred in practice due to the expensive model search/training cost.

There is a line of research for addressing this issue. Cai et al. (2020) proposed an efficient method, called Once-For-All(OFA), producing non-trivial neural networks simultaneously for multiple target environments. Sahni et al. (2021) proposed a method called CompOFA which is more efficient than OFA in terms of training cost and search cost based on reducing the design space for model configuration. Since these methods do not utilize a pretrained network, they require too much time for training the full network. In order to address such an expensive cost, Molchanov et al. (2022) proposed a latency-aware network acceleration method called LANA based on a pretrained teacher model. This method dramatically reduces training and search costs with large design space. In LANA, for each layer in the teacher model, more than 100 alternative blocks (sub-networks) are trained to mimic it. After training them, LANA replaces ineffective layers in the teacher model with lightweight alternatives using integer programming. Even if LANA utilizes a pretrained teacher model, it requires designing alternative blocks. In addition, training many alternatives for every single layer may increase peak CPU/GPU memory usage. This limitation leads LANA to require $O(M)$ epochs for training the alternatives on ImageNet where M is the number of alternatives for each layer.

To significantly reduce the cost of searching for and retraining models for multiple target environments, we focus on public pretrained neural networks like models in Keras' applications[1]. Each of those networks was properly searched and trained with plentiful resources. Thus, instead of newly exploring large search space with a number of expensive GPUs, we devise a novel way of exploiting decent reusable blocks (sub-networks) in those networks. Our work starts from this rationale.

In this paper, we devise a framework named Bespoke efficiently optimizing neural networks for various target environments. This framework is supposed to help us to get fast and small neural networks with practically affordable cost, which a single modern GPU can handle. The illustration of comparing OFA, LANA, and Bespoke is depicted in Figure 1. Our framework has following notable features compared to previous methods like OFA and LANA.

- **Near Zero-Cost Model Design Space**: Our framework is based on the sub-networks of a teacher (original) model and those of pretrained models. Our framework

---

*Corresponding author.
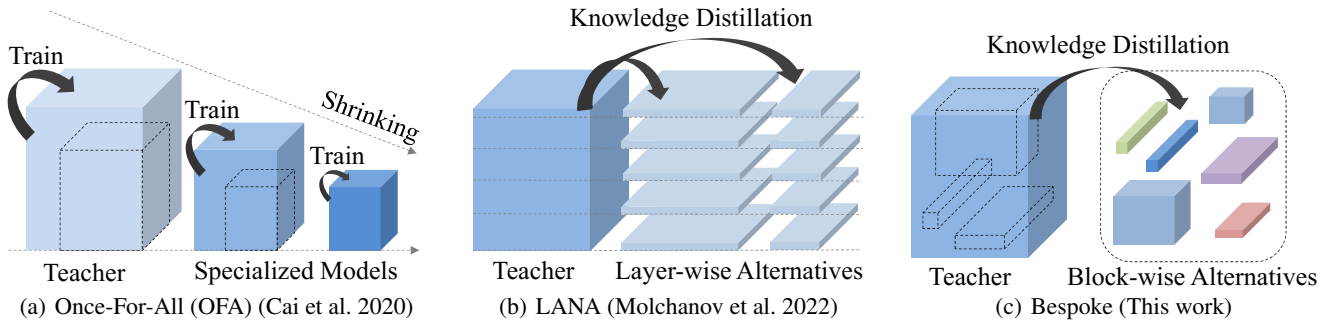
[1]https://keras.io/api/applications

Figure 1: Illustration of model optimization frameworks for multiple deployment targets

extracts uniformly at random sub-networks from the teacher and the pretrained networks, and uses them for constructing a lightweight model. Thus, we do not need to struggle to expand the design space with fancy blocks and numerous channel/spatial settings. Such blocks are already included in the pretrained neural networks which were well-trained with large-scale data. We can use them for a target task with a minor cost of knowledge distillation. To the best of our knowledge, this is the first work to utilize parts of such pretrained networks for student model construction.

- **Low-Cost Preprocessing**: The randomly selected sub-networks are trained to mimic a part of the teacher via knowledge distillation in preprocessing. They can be simultaneously trained with low peak memory usage compared to OFA, CompOFA, and LANA.

- **Efficient Model Search**: Our framework deals with a neural network as a set of sub-networks. Each sub-network is extracted from the original network so that we can precisely measure the actual inference time of it. In addition, for evaluating the suitability of an alternative sub-network, our framework produces an induced neural network by rerouting the original network to use the alternative. Then, we compute the accuracy of the induced network for a validation dataset. The measured inference time and the accuracy lead us to make an incremental algorithm to evaluate a candidate neural network. Based on this, we devise an efficient model search method.

Related Work

Block-wise NAS. As our framework deals with a neural network in a block-wise way, there are several early works for block-wise architecture search/generation (Zhong et al. 2018; Li et al. 2020a). Such works find network architectures at block-level to effectively reduce search space. Zhong et al. (2018) proposed a block-wise network generation method with Q-learning, but it is still somewhat slow because it requires training a sampled model from scratch for evaluation. Li et al. (2020a) proposed another block-wise method to ensure that potential candidate networks are fully trained without additional training. Since the potential candidate networks are fully trained, the time for evaluating each architecture can be significantly reduced. For this, Li et

al. (2020a) used a teacher model and knowledge distillation with it. Compared to (Li et al. 2020a), we do not only use a teacher model for knowledge distillation but also use it as a base model to find a result model. In addition, while Li et al. (2020a) only considered sequentially connected blocks, this work does not have such a limitation regarding the form of a block. Since we reuse the components (blocks) in the teacher model with their weights, our framework can have lower costs for pretraining and searching than such NAS-based approaches, even if they are block-wisely formulated.

Hardware-aware NAS. Hardware-aware architecture search has been one of the popular topics in deep learning, so there are many existing works (Cai et al. 2020; Sahni et al. 2021; Yu et al. 2020; Moons et al. 2021; Abdelfattah et al. 2021; Molchanov et al. 2022). Those works usually aim to reduce model search space, predict the accuracy of a candidate model, or consider actual latency in the optimization process. In addition, some of such works were proposed to reduce the cost of deploying models to multiple hardware environments (Cai et al. 2020; Sahni et al. 2021; Moons et al. 2021; Molchanov et al. 2022). Moons et al. (2021) proposed a method named DONNA built on block-wise knowledge distillation for rapid architecture search with consideration of multiple hardware platforms. DONNA uses teacher blocks defined in a pre-trained reference model for distilling knowledge to student blocks, not for building student block architectures. Despite such many existing works, we think that there is an opportunity for improvement in terms of searching/training costs with the sub-networks in pretrained models.

Knowledge Distillation. Our framework includes knowledge distillation to train the sub-networks. Some early works for block-wise knowledge distillation which are related to our framework (Yim et al. 2017; Wang et al. 2018; Li et al. 2020b; Shen et al. 2021). Shen et al. (2021) proposed a block-wise distilling method for a student model which is given by model compression or an initialized lightweight model. In contrast to (Li et al. 2020a; Moons et al. 2021), they used a teacher model for making a student model. However, while our framework deals with any sub-network for a single input and a single output as a block, they used sequentially connected blocks at a high level. That is, student models produced by our framework are likely to be more diverse than those by (Shen et al. 2021).

Method

Bespoke consists of two steps: preprocessing and model search. Given a teacher (original) model, the preprocessing step finds sub-networks which will be used as alternatives. This step is described in Figure 2. The model search step constructs a student model by replacing some parts of the teacher model with the sub-networks. In this section, all the proofs are included in the supplemental material[2].

Formulation

For a neural network denoted by $\mathcal{N}$, the set of all layers in $\mathcal{N}$ is denoted by $L(\mathcal{N})$. Then, for any subset $X \subseteq L(\mathcal{N})$, we define a sub-network of $\mathcal{N}$ which consists of layers in X with the same connections in $\mathcal{N}$. Its inputs are defined over the connections from layers in $L(\mathcal{N}) \setminus X$ to X in $\mathcal{N}$. The outputs of the sub-network are similarly defined. For simplicity, we consider only sub-networks having a single input and a single output. We denote a set of sub-networks in $\mathcal{N}$ sampled in a certain way by $\Omega(\mathcal{N})$.

For a sub-network $\mathcal{S}$, the spatial change of $\mathcal{S}$ is defined to be the ratio between its spatial input size and its spatial output size. For example, if $\mathcal{S}$ has a convolution with stride 2, its spatial change is $\frac{1}{2}$. Then, for any two sub-networks $\mathcal{S}$ and $\mathcal{A}$, we define that if the spatial change of $\mathcal{S}$ is equal to that of $\mathcal{A}$ and the in/out channels of $\mathcal{A}$ are larger than or equal to those of $\mathcal{S}$, $\mathcal{A}$ is compatible with $\mathcal{S}$.

Suppose that we have a neural network $\mathcal{N}$ and a sub-network $\mathcal{S}$ of $\mathcal{N}$. Suppose that we have another sub-network $\mathcal{A}$ that is compatible with $\mathcal{S}$ and replace $\mathcal{S}$ with $\mathcal{A}$ for $\mathcal{N}$. $\mathcal{A}$ may come from $\mathcal{N}$ or another neural network. If the computational cost of $\mathcal{A}$ is smaller than that for $\mathcal{S}$, such a replacement can be seen as model acceleration/compression. One can worry a case that the input/output channel dimensions of $\mathcal{S}$ are smaller than those of $\mathcal{A}$. Even in this case, by pruning input/output channels of $\mathcal{A}$, we can make a pruned network $\mathcal{A}'$ from $\mathcal{A}$ having the same input/output channel dimensions as $\mathcal{S}$. Then, we can replace $\mathcal{S}$ with $\mathcal{A}'$ in $\mathcal{N}$ without computational violation. We say that a neural network produced by such a replacement is induced from $\mathcal{N}$.

Suppose that we have a teacher neural network $\mathcal{T}$ which we want to optimize. Let us consider a set of public pretrained networks, denoted by $\mathbf{P}$. Then, suppose that we have sub-networks in $\mathcal{T}$ as well as sub-networks in the pretrained networks of $\mathbf{P}$. By manipulating such sub-networks, we propose a novel structure named a model house $\mathsf{H} = (\Omega(\mathcal{T}), \mathbf{A})$ where $\mathbf{A}$ is a set of sub-networks each of which is derived from $\mathcal{T}$ or a neural network in $\mathbf{P}$. $\mathbf{A}$ is called the alternative set, and every element in it is compatible with a sub-network in $\Omega(\mathcal{T})$. Given a model house H, Our framework finds a lightweight model $\mathcal{M}^*$ which is induced from $\mathcal{T}$ as follows:

$$\mathcal{M}^* = \underset{\mathcal{M} \in satisfy(\mathsf{H}, R)}{\arg\min} Loss_{val}(\mathcal{M}), \qquad (1)$$

where $Loss_{val}(\mathcal{M})$ is the task loss of a (neural network) model $\mathcal{M}$ upon the validation dataset, R represents a certain requirement, and $satisfy(\mathsf{H}, R)$ is a set of candidate neural networks induced from $\mathcal{T}$ with H which satisfy R.
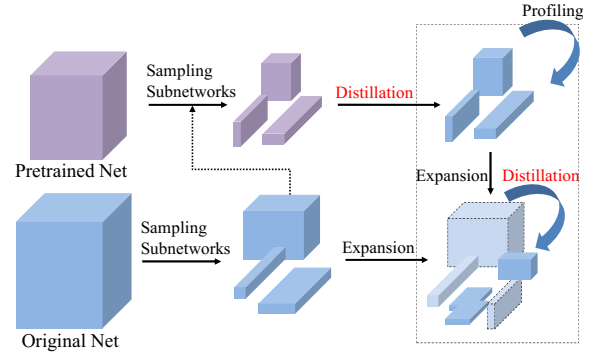
[2] https://arxiv.org/abs/2303.01913

Figure 2: The flow of the preprocessing step

Algorithm 1: Modified Depth-First Search($l_0, \mathcal{T}$)

Input: l_0: a layer in $\mathcal{T}$, $\mathcal{T}$: a neural network
1 **begin**
2   Initialize a stack S with l_0;
3   Initialize a vector δ as the zero vector in $\mathbb{R}^{|\mathcal{T}|}$;
4   **while** S *is not empty* **do**
5         $v := \text{pop}(S)$;
6   mark v as being visited;
7   **for** $u \in \mathbf{N}_{out}(v)$ **do**
8   Increase $\delta[u]$ by 1;
9   **if** $\delta[u] = |\mathbf{N}_{in}(u)|$ **then**
10   Push u into S;

Preprocessing

Let us introduce how to find effective sub-networks for a model house H and how to process them.

Enumeration. Let us introduce a way of finding a sub-network in a (teacher) neural network $\mathcal{T}$. Note that it is not trivial to find such a sub-network in $\mathcal{T}$ due to complex connections between layers such as skip connections. To address this issue, we use a modified version of the Depth-First Search (DFS). The modified algorithm starts from any layer in $\mathcal{T}$, and it can move from a current layer l to its outbound neighbors only if all the inbound layers of l were already visited. For understanding, we provide the detailed procedure of the modified DFS in Algorithm 1. In this algorithm, $|\mathcal{T}|$ represents the number of layers in $\mathcal{T}$. In addition, for any layer l, $\mathbf{N}_{\text{out}}(l)$ is the outbound layers of l, and $\mathbf{N}_{\text{in}}(l)$ is the inbound layers of l. By maintaining a vector δ, we can make u wait until all the inbound layers of u are visited.

Suppose that the modified DFS algorithm started from a layer l_0. Let us denote the $(i+1)$-th layer popped from S by l_i and the set of popped layers before l_i by P_i. Then, the modified DFS algorithm guarantees the following lemma.

Lemma 1. *Suppose that l_0 has a single input and l_i has a single output. If l_i is the only element in the stack when l_i is being popped from the traversing stack, the sub-network defined by $P_i \cup \{l_i\}$ from $\mathcal{T}$ whose l_i is the output layer has the single input and the single output.*

(a) Simple skip connection   (b) Nested skip connection   (c) Sequentially connected blocks
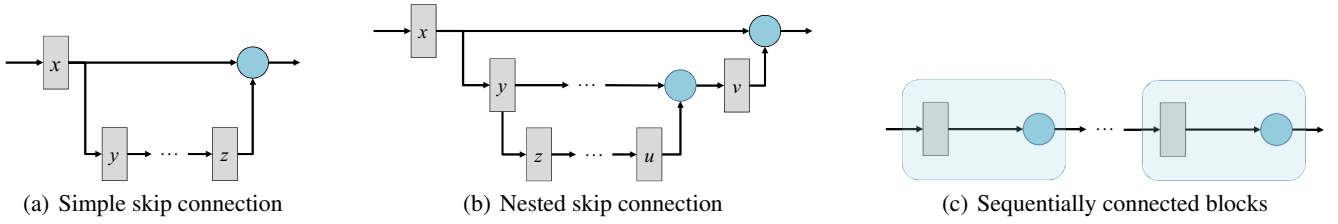
Figure 3: Illustration of example sub-networks having a single input and a single output with skip connections. A rectangle represents a layer and a circle represents an element-wise operation like addition and multiplication.

The key point of this lemma is the completeness that every layer in $P_i \cup \{l_i\}$ is connected from or to another in the set except l_0 and l_i. Based on this lemma, we can derive the following theorem.

Theorem 1. *All the possible sub-networks of a neural network $\mathcal{T}$ having a single input and a single output can be enumerated by running the modified DFS algorithm multiple times.*

For understanding, Figure 3 depicts sub-networks that can be found with the modified DFS algorithm. An example of the simple skip connection is a sub-network with residual connections in ResNet (He et al. 2016). An example of the nested skip connections is a sub-network with a residual connection and a squeeze-and-excitation module in EfficientNet (Tan and Le 2019). Our enumeration algorithm can also find a sub-network including multiple blocks with skip connections depicted in Figure 3(c).

Sampling. Because a neural network usually has more than 100 layers, enumerating all possible sub-networks is simply prohibitive with respect to time and space. Another decent option is to randomly sample sub-networks. Given a neural network $\mathcal{N}$, the sub-network random sampling algorithm works as follows. First, it collects all individual layers in $\mathcal{N}$. Then, it uniformly at random selects a layer among them and conducts the modified DFS algorithm. Whenever l_i is being popped from the stack only having l_i, the algorithm stores a pair of l_i and P_i. After the traversal, our algorithm uniformly at random selects one among the stored pairs and derives a sub-network as the return. In practice, for this random selection, we use a part of the stored pairs close to l_0 to avoid selecting over-sized sub-networks. The size ratio of the part to the entire stored pairs is parameterized to r. We denote the sub-network sampling algorithm with a neural network $\mathcal{N}$ by *SubnetSampling($\mathcal{N}$)*.

The sub-network sampling algorithm is used to get the sub-networks of a teacher network and pretrained networks. Let us denote a set of sampled sub-networks of pretrained networks in $\mathbf{P}$ by $\Omega(\mathbf{P})$. Algorithm 2 describes the overall procedure to find $\Omega(\mathcal{T})$ and $\Omega(\mathbf{P})$ with *SubnetSampling(·)*. In Lines 3-5, it computes sub-networks in $\mathcal{T}$. In Lines 6-12, for a randomly sampled sub-network $\mathcal{S} \in \Omega_{\mathcal{T}}$, it uniformly at random samples a pretrained neural network $\mathcal{P}$ and gets a sub-network $\mathcal{P}'$ of it. We store a mapping from $\mathcal{P}'$ to $\mathcal{S}$ into *TMap* for our search algorithm.

Further Approximation. In order to expand the search space of our framework, we generate more sub-networks

---

Algorithm 2: *Construct($\mathcal{T}, \mathbf{P}, n_{\mathcal{T}}, n_{\mathbf{P}}$)*

Input: $\mathcal{T}$: the teacher network, $\mathbf{P}$: the set of pretrained networks, $n_{\mathcal{T}}$: the number of sampled sub-networks in $\mathcal{T}$, $n_{\mathbf{P}}$: the number of sampled sub-networks over $\mathbf{P}$

Output: $\Omega(\mathcal{T})$: a set of sampled sub-networks in $\mathcal{T}$, $\Omega(\mathbf{P})$: a set of sampled sub-networks in the neural networks of $\mathbf{P}$, *TMap*: a map

```
1 begin
2     Ω_T := {}, Ω_P := {}, initialize TMap;
3     while |Ω_T| < n_T do
4         S := SubnetSampling(T);
5         Add S into Ω_T;
6     while |Ω_P| < n_P do
7         Randomly sample S ∈ Ω_T;
8         Randomly sample P ∈ P;
9         P' := SubnetSampling(P);
10        if P' is compatible with S then
11            Add P' into Ω_P;
12            TMap[P'] := S;
13    return Ω_T, Ω_P, TMap;
```

and add them to $\mathbf{A}$. One straightforward approach is to apply model compression techniques such as channel pruning and decomposition to the sub-networks in $\Omega(\mathcal{T})$ and $\Omega(\mathbf{P})$. For this purpose, we use a variant of a discrepancy-aware channel pruning method (Luo and Wu 2020) named CURL. This variant simply uses mean-squared-error as the objective function, while the original version uses KL divergence.

Since a channel pruning method changes the number of the in/out channels of sub-networks, one can ask about how to handle the compatibility between sub-networks. While every alternative should have the same in/out shapes of a teacher layer in (Molchanov et al. 2022), we permit the changes of an alternative's in/out shapes by introducing channel-wise masking layers.

To implement this, for any alternative sub-network $\mathcal{A} \in \mathbf{A}$, we append an input masking layer M_{in} and output masking layer M_{out} after the input layer and the output layer, respectively. If a channel pruning method makes channel-wise masking information for the in/out channels of $\mathcal{A}$, it is stored in M_{in} and M_{out}. The masked version of $\mathcal{A}$ is used instead of $\mathcal{A}$ when $\mathcal{A}$ is selected as a replacement. Suppose that a

resulting student model has such masked sub-networks. After fine-tuning the student model, we prune internal channels in it according to the masking information in masking layers. Keeping the masking layers during fine-tuning is essential for intermediate feature-level knowledge distillation. For understanding, we provide figures explaining this procedure in the supplemental material.

We denote a set of networks added by the expansion process from $\Omega(\mathcal{T})$ and $\Omega(\mathbf{P})$ by $\Omega^*(\mathcal{T}, \mathbf{P})$. Then, the alternative set $\mathbf{A}$ is defined as,

$$\mathbf{A} = \Omega(\mathcal{T}) \cup \Omega(\mathbf{P}) \cup \Omega^*(\mathcal{T}, \mathbf{P}). \qquad (2)$$

Sub-Network Training and Profiling. Let us discuss how to train the sub-networks in $\mathbf{A}$ to use them for producing a student model. We train them via knowledge distillation as Molchanov et al. (2022) did. Recall that each alternative sub-network $\mathcal{A} \in \mathbf{A}$ has a corresponding sub-network $\mathcal{S} \in \Omega(\mathcal{T})$. Thus, for each sub-network $\mathcal{A} \in \mathbf{A}$, we can construct a distillation loss defined by the mean-squared-error between the output of $\mathcal{A}$ and that of $\mathcal{S}$. By minimizing such a MSE loss by gradient descent optimization, we can train the sub-networks in $\mathbf{A}$. Formally, the mean-square-error loss for training alternatives is defined as follows:

$$\min_{W_{\mathbf{A}}} \sum_{x \in X_{\text{sample}}} \sum_{\mathcal{S} \in \Omega(\mathcal{T})} \sum_{\mathcal{A} \in \mathbf{A}_{\mathcal{S}}} (f_{\mathcal{S}}(x^*) - f_{\mathcal{A}}(x^*))^2, \qquad (3)$$

where $W_{\mathbf{A}}$ is the set of all weight parameters over $\mathbf{A}$, X_{sample} is a set of sampled input data, $\mathbf{A}_{\mathcal{S}}$ is a set of alternatives compatible with $\mathcal{S}$, $f_{\mathcal{S}}$ is the functional form of $\mathcal{S}$, and x^* is the output of the previous layer of $\mathcal{S}$ in $\mathcal{T}$. It is noteworthy that if we have enough memory size for a GPU, all the sub-networks in $\mathbf{A}$ can be trained together in a single distillation process. If GPU memory is insufficient, they can be trained over several distillation processes.

Requirement and Profiling. A requirement can be related to FLOPs, actual latency, memory usage, and so forth. In order to produce a student model satisfying a requirement, we measure such metrics for every alternative sub-network after the sub-network training step.

Model Search

Given a requirement and a model house $\mathbf{H} = (\Omega(\mathcal{T}), \mathbf{A})$, our search algorithm for a student model is based on simulated annealing. Simulating annealing is a meta-heuristic algorithm for optimizing a function within computational budget. In order to exploit simulated annealing, we need to define three things: an initial solution, a function to define a next candidate, and the objective function. For simulated annealing, a solution $\mathbf{S}$ is defined to be a set of sub-networks in $\mathbf{A}$, and we say that $\mathbf{S}$ is feasible when sub-networks in $\mathcal{T}$ corresponded to the sub-networks in $\mathbf{S}$ by *TMap* are not overlapped. In this work, the initial solution is a solution selected by a simple greedy method. For a solution $\mathbf{S}$, the score function for the greedy method is defined as,

$$\Delta acc(\mathbf{S}) = \sum_{\mathcal{A} \in \mathbf{S}} \Delta acc(\mathcal{A}, TMap[\mathcal{A}], \mathcal{T}), \qquad (4)$$

$$score(\mathbf{S}) = \max \left\{ 1.0, \frac{metric(\mathcal{N}_{\mathbf{S}})}{R} \right\} + \lambda \Delta acc(\mathbf{S}), \quad (5)$$

where $\Delta acc(\mathcal{A}, TMap[\mathcal{A}], \mathcal{T})$ represents the accuracy loss when $TMap[\mathcal{A}]$ is replaced with $\mathcal{A}$ for $\mathcal{T}$. The values for $\Delta acc(\cdot)$ can be calculated in preprocessing. $metric(\cdot)$ is a metric function to evaluate the inference time (e.g., latency) of a candidate student model, and $\mathcal{N}_{\mathbf{S}}$ is a candidate student model induced by $\mathbf{S}$. R is an objective value in terms of $metric(\cdot)$. We define $score(\cdot)$ handling a single requirement (metric) for simplicity, but it can be easily extended to multiple requirements. This score function is also used in the annealing process. Note that since constructing $\mathcal{N}_{\mathbf{S}}$ with alternatives is time-consuming, we take another strategy for computing it. $metric(\mathcal{N}_{\mathbf{S}})$ is approximated incrementally from $metric(\mathcal{T})$ by adding $metric(\mathcal{A})$ and subtracting $metric(TMap[\mathcal{A}])$ for each sub-network $\mathcal{A} \in \mathbf{S}$.

The remaining thing to complete the annealing process is to compute the next candidate solution. Given a current solution $\mathbf{S}$, we first copy it to $\mathbf{S}'$ and simply uniformly at random remove a sub-network from it. Then, we repeatedly add a randomly selected sub-network in $\mathbf{A}$ into $\mathbf{S}'$ until the selected sub-network violates the feasibility of $\mathbf{S}'$. After that, $\mathbf{S}'$ becomes a next candidate solution.

Retraining with Distillation. After getting a student model, we retrain it with knowledge distillation. Consider that the final solution of the simulated annealing process is denoted by $\mathbf{S}^*$. This distillation process works with the mean-square-error loss between the outputs corresponding to the sub-networks of $\mathbf{S}^*$ in $\mathcal{N}_{\mathbf{S}^*}$ and their corresponding sub-networks in $\mathcal{T}$.

Experiments

Implementation Detail

Datasets. We use two representative datasets: CIFAR-100 (Krizhevsky 2009) and ImageNet-1K (ImageNet) (Russakovsky et al. 2015). CIFAR-100 has 50K training images and 10K validation images with 100 classes. ImageNet has 1,281K training images and 50K validation images with 1000 classes.

Models. As original (teacher) models, we use Efficient-Net (Tan and Le 2019). As pretrained models, we do not only use them, but also ResNet50V2 (He et al. 2016), MobileNet (Howard et al. 2017), MobileNetV2 (Sandler et al. 2018), MobileNetV3 (Howard et al. 2019), Inception-ResNet (Szegedy et al. 2017), ResNetRS (Bello et al. 2021), and EfficientNetV2 (Tan and Le 2021). Those models were already pretrained and are available at Keras' applications.

It should be noticed that for CIFAR-100, EfficientNet and MobileNetV3 are transformed for transfer learning to additionally include a global average pooling layer, a dropout layer and a fully-connected layer. That is why they are slower for CIFAR-100 than for ImageNet.

Computation Environment. Each experiment for CIFAR-100 is conducted with NVIDIA GeForce Titan RTX, while that for ImageNet is conducted with NVIDIA RTX A5000.

CPU latency and GPU latency are measured with ONNX Runtime[3]. For reference, Intel i9-10900X is used for CIFAR-100, while AMD EPYC 7543 32-Core processor is used for ImageNet.

---

[3]https://github.com/onnx/onnx

Methods	CIFAR-100			ImageNet		
	Top-1 Acc. (%)	Latency (ms)	Params (M)	Top-1 Acc. (%)	Latency (ms)	Params (M)
EfficientNetB2	88.01	72.78	7.91	79.98	28.40	9.18
EfficientNetB0	86.38	41.28	4.18	77.19	18.63	5.33
Bespoke-EB2-Rand	79.45	26.11	6.98	71.78	16.72	9.58
Bespoke-EB2-Only	84.47	33.07	4.39	65.62	13.74	6.30
Bespoke-EB2	**85.45**	**31.82**	**7.43**	**78.61**	**17.55**	**9.39**
MobileNetV3-Large	84.33	24.71	3.09	75.68	13.54	5.51
MobileNetV3-Small	81.98	11.98	1.00	68.21	6.99	2.55
CURL	81.49	28.65	1.89	71.21	17.20	2.37
Bespoke-EB0	**84.86**	**22.05**	**3.78**	**73.93**	**9.39**	**6.77**

Table 1: Overall Results (CPU Latency)

Implementation. We implement all things with TensorFlow Keras because it is easy to implement the sub-network extraction and injection with Keras. Our training code is based on NVIDIA Deep Learning Examples[4].

EfficientNet and MobileNetV3 are fine-tuned for CIFAR-100 with the learning rate is 0.0001 and cosine annealing (Loshchilov and Hutter 2017). We used CutMix (Yun et al. 2019) and MixUp (Zhang et al. 2018) together for fine-tuning them, but not for retraining a student model.

The learning rate for training alternative sub-networks and fine-tuning a student model after the model search step is 0.02. We used the SGD optimizer for training the alternatives, and the AdaBelief optimizer (Zhuang et al. 2020) for fine-tuning the student model. We used the same hyper-parameter values for the AdaBelief optimizer depending on datasets as described in (Zhuang et al. 2020). The number of epochs for training the alternatives is one for ImageNet and 20 for CIFAR-100, respectively. In addition, the number of epochs for fine-tuning the student model is 15 for ImageNet and 30 for CIFAR-100, respectively.

Recall r is the ratio parameter to control the maximum size of sampled sub-networks. It is set to 20% for ImageNet and 30% for CIFAR-100. $n_{\mathcal{T}}$ is the parameter for $|\Omega(\mathcal{T})|$ and $n_{\mathbf{P}}$ is the parameter for $|\Omega(\mathbf{P})|$. For all the experiments, $n_{\mathcal{T}}$ and $n_{\mathbf{P}}$ are fixed to 100 and 200, respectively. In addition, $|\Omega^*(\mathcal{T}, \mathbf{P})|$, which is the number of sub-networks added by the expansion process, is also fixed to 200.

Results

Building a model house and searching for a student model are repeated 5 times. Thus, each reported latency/accuracy for Bespoke is the average over 25 measured values.
Overall Inference Results. CURL (Luo and Wu 2020) is a channel pruning method which gives a global score to each channel in a network. Based on the score information, we can make models with various scales. Thus, we compare our framework with it in terms of latency and accuracy.

The overall results are presented in Table 1, Table 2, and Table 3. In these tables, the results of applying Bespoke to EfficientNet-B2 (EfficientNet-B0) are presented with Bespoke-EB2 (Bespoke-EB0). 'Rand' represents the results of models built by random selection on the same

Methods	Top-1 Acc. (%)	Latency (ms)
EfficientNetB2	88.01	5.78
EfficientNetB0	86.38	4.06
Bespoke-EB2-Rand	79.45	5.36
Bespoke-EB2-Only	87.30	5.15
Bespoke-EB2	**87.01**	**3.08**
MobileNetV3-Large	84.33	3.04
MobileNetV3-Small	81.98	2.52
CURL	81.49	4.02
Bespoke-EB0	**86.18**	**3.19**

Table 2: GPU Latency (CIFAR-100)

Methods	Top-1 Acc. (%)	Latency (ms)
EfficientNetB2	79.98	3.55
EfficientNetB0	77.19	2.29
Bespoke-EB2-Rand	71.78	2.55
Bespoke-EB2-Only	74.22	2.83
Bespoke-EB2	**78.08**	**2.16**
MobileNetV3-Large	75.68	1.82
MobileNetV3-Small	68.21	1.29
CURL	71.21	1.81
Bespoke-EB0	**72.75**	**2.40**

Table 3: GPU Latency (ImageNet)

model house of Bespoke-EB2. 'Only' represents the results of making Bespoke only use the sub-networks of a teacher. It is noteworthy that the accuracy of Bespoke-EB2 is higher than that of EfficientNet-B0 for ImageNet while they have similar CPU latency. This means that Bespoke can effectively scale down EfficientNet-B2 in terms of CPU latency. The accuracy of Bespoke-EB2 for CIFAR-100 is lower than that of EfficientNet-B0, but Bespoke-EB2 is more efficient than EfficientNet-B0. Compared to MobileNetV3 models, which were proposed for fast inference, Bespoke-EB2 is somewhat slower than them. However, it can still be more accurate than the MobileNetV3 models. Bespoke-EB0 is faster than MobileNetV3-Large with slightly less accuracy. Note that the accuracy gap between MobileNetV3-Large and Bespoke-EB2 is larger than the gap between MobileNetV3-Large and Bespoke-EB0, while the latency gap is not. In addition, the latency difference between Bespoke-EB0 and MobileNetV3-Small is marginal, but the accuracy gap is significant. This means that the stu-

Methods	Preprocess	Search	Retrain
OFA	1205	40	75
DNA	320	14	450
DONNA	1920+1500	1500	50
LANA	197	< 1h	100
Bespoke	$|\mathbf{A}|/G = 4$	< 0.5h	15

Table 4: Cost Analysis for ImageNet (Epochs)

Methods	Top-1 Acc. (%)	Speed-up
EfficientNet-B0	77.19	1.0
LANA-EB2-0.4 (CPU)	78.11	1.19
LANA-EB2-0.5 (CPU)	78.87	0.98
Bespoke-EB2 (CPU)	78.61	1.06
LANA-EB2-0.45 (GPU)	79.71	1.10
Bespoke-EB2 (GPU)	78.08	1.06

Table 5: Comparison with LANA (ImageNet)

Ratio	Top-1 Acc. (%)	CPU Latency (ms)
1.0	85.45	31.82
0.75	86.23	37.34
0.5	86.23	44.50

Table 6: Random Partial Alternative Set Test (CIFAR-100, Bespoke-B2)

dent models of Bespoke have a better trade-off between accuracy and latency than those of MobileNetV3.

Models found by Bespoke-EB2-Rand have inferior accuracy compared to the other competitors. This result supports that selecting good alternatives is important for Bespoke, and the search algorithm of Bespoke effectively works. In addition, we observed that models found by Bespoke-EB2-Only are less competitive than those by Bespoke-EB2. From this result, the public pretrained networks are actually helpful for constructing fast and accurate models.

Meanwhile, Bespoke-EB0 is somewhat behind in terms of a trade-off between accuracy and GPU latency, especially for ImageNet. This may happen with the quality of the sampled sub-networks. Bespoke-EB0 will produce better results with a more number of sampled sub-networks and a more number of epochs for training them.

Since CURL does not exploit actual latency, its latency gain is insignificant. Recall that Bespoke utilizes the variant of CURL to expand the alternative set. This observation implies that the latency gain of Bespoke mainly comes from replacing expensive sub-networks with fast alternatives, not the expansion. We expect that Bespoke can find faster models with latency-aware compression methods.

Cost analysis. We analyze the cost for preprocessing, searching, and retraining. The results for OFA (Cai et al. 2020), DNA (Li et al. 2020a), DONNA (Moons et al. 2021), and LANA (Molchanov et al. 2022) come from (Molchanov et al. 2022). Note that 'preprocess' includes pretraining process and 'search' is the cost for searching for a single target.

The result is described in Table 4. Note that G is the number of sub-networks that can be trained with a single GPU in parallel. In the experiments, G is 100 for ImageNet. Since each sub-network is pretrained with one epoch, the entire cost for pretraining the alternatives is $|\mathbf{A}|/G$. The preprocessing step includes applying the channel pruning method, but its cost is negligible. Note that the preprocessing cost of LANA is also calculated with parallelism. From this table, we can see that Bespoke requires significantly less costs than the other methods for all the steps. Especially, due to the concept of the block-level replacement, the preprocessing cost of Bespoke is an order of magnitude smaller than that of any other method.

Comparison with LANA. We provide an analysis to compare Bespoke with LANA (Molchanov et al. 2022), which is described in Table 5. Because it is not possible to set up the same environment with LANA, we use relative speed-up from EfficientNet-B0. The model found by LANA is more accurate than that by Bespoke for GPU latency with faster inference latency. On the other hand, Bespoke-EB2 is not

dominated by the models of LANA in terms of accuracy and CPU latency. This result is quite promising because Bespoke requires much less cost for preprocessing, but there is a situation that Bespoke is comparable to LANA.

Random sub-alternative set test. We conducted experiments evaluating Bespoke with random subsets of alternative set $\mathbf{A}$ (Table 6). The ratio is the size ratio from a random subset to $\mathbf{A}$. Bespoke finds an inefficient student model with the random subset of half size, which is slower than EfficientNet-B0. This is because many efficient and effective sub-networks are not included in the subset. This result supports the importance of a good alternative set.

Conclusions

This paper proposes an efficient neural network optimization framework named Bespoke, which works with sub-networks in an original network and public pretrained networks. The model design space of Bespoke is defined by those sub-networks. Based on this feature, it can make a lightweight model for a target environment at a very low cost. The results of the experiments support the assumption that such sub-networks can actually be helpful in reducing searching and retraining costs. Compared to LANA, Bespoke can find a lightweight model having comparable accuracy and CPU latency. Its entire cost is much lower than that required by LANA and the other competitors.

For future work, we consider making Bespoke support other valuable tasks such as pose estimation and image segmentation. In addition, by adding more recent pretrained neural architectures into Bespoke, we want to see how Bespoke improves with them.

Acknowledgments

References

Abdelfattah, M. S.; Mehrotra, A.; Dudziak, Ł.; and Lane, N. D. 2021. Zero-Cost Proxies for Lightweight {NAS}. In *ICLR*.

Bello, I.; Fedus, W.; Du, X.; Cubuk, E. D.; Srinivas, A.; Lin, T.-Y.; Shlens, J.; and Zoph, B. 2021. Revisiting ResNets: Improved Training and Scaling Strategies. In *NeurIPS*.

Cai, H.; Gan, C.; Wang, T.; Zhang, Z.; and Han, S. 2020. Once-for-All: Train One Network and Specialize it for Efficient Deployment. In *ICLR*.

He, K.; Zhang, X.; Ren, S.; and Sun, J. 2016. Identity Mappings in Deep Residual Networks. In *ECCV*, 630–645.

Hinton, G.; Vinyals, O.; and Dean, J. 2015. Distilling the Knowledge in a Neural Network. *arXiv preprint arXiv:1503.02531*.

Howard, A.; Sandler, M.; Chu, G.; Chen, L.-C.; Chen, B.; Tan, M.; Wang, W.; Zhu, Y.; Pang, R.; Vasudevan, V.; Le, Q. V.; and Adam, H. 2019. Searching for MobileNetV3. In *ICCV*, 1314–1324.

Howard, A. G.; Zhu, M.; Chen, B.; Kalenichenko, D.; Wang, W.; Weyand, T.; Andreetto, M.; and Adam, H. 2017. MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications. *CoRR*, abs/1704.04861.

Krizhevsky, A. 2009. Learning Multiple Layers of Features from Tiny Images. *Technical Report TR-2009, University of Toronto*.

Lee, C.; Hong, S.; Hong, S.; and Kim, T. 2020. Performance analysis of local exit for distributed deep neural networks over cloud and edge computing. *ETRI Journal*, 42(5): 658–668.

Li, C.; Peng, J.; Yuan, L.; Wang, G.; Liang, X.; Lin, L.; and Chang, X. 2020a. Block-Wisely Supervised Neural Architecture Search With Knowledge Distillation. In *CVPR*, 1986–1995.

Li, T.; Li, J.; Liu, Z.; and Zhang, C. 2020b. Few Sample Knowledge Distillation for Efficient Network Compression. In *CVPR*, 14627–14635.

Liu, L.; Zhang, S.; Kuang, Z.; Zhou, A.; Xue, J.-H.; Wang, X.; Chen, Y.; Yang, W.; Liao, Q.; and Zhang, W. 2021. Group Fisher Pruning for Practical Network Compression. In *ICML*, volume 139, 7021–7032.

Loshchilov, I.; and Hutter, F. 2017. SGDR: Stochastic Gradient Descent with Warm Restarts. In *ICLR*.

Luo, J.-H.; and Wu, J. 2020. Neural Network Pruning With Residual-Connections and Limited-Data. In *CVPR*, 1455–1464.

Molchanov, P.; Hall, J.; Yin, H.; Kautz, J.; Fusi, N.; and Vahdat, A. 2022. LANA: Latency Aware Network Acceleration. In *ECCV*, 137–156.

Moons, B.; Noorzad, P.; Skliar, A.; Mariani, G.; Mehta, D.; Lott, C.; and Blankevoort, T. 2021. Distilling Optimal Neural Networks: Rapid Search in Diverse Spaces. In *ICCV*, 12209–12218.

Park, S.; Lee, J.; Mo, S.; and Shin, J. 2020. Lookahead: A Far-sighted Alternative of Magnitude-based Pruning. In *ICLR*.

Peng, H.; Wu, J.; Chen, S.; and Huang, J. 2019. Collaborative Channel Pruning for Deep Networks. In *ICML*, volume 97, 5113–5122.

Russakovsky, O.; Deng, J.; Su, H.; Krause, J.; Satheesh, S.; Ma, S.; Huang, Z.; Karpathy, A.; Khosla, A.; Bernstein, M.; Berg, A. C.; and Fei-Fei, L. 2015. ImageNet Large Scale Visual Recognition Challenge. *International Journal of Computer Vision (IJCV)*, 115(3): 211–252.

Sahni, M.; Varshini, S.; Khare, A.; and Tumanov, A. 2021. Comp{OFA} – Compound Once-For-All Networks for Faster Multi-Platform Deployment. In *ICLR*.

Sandler, M.; Howard, A.; Zhu, M.; Zhmoginov, A.; and Chen, L.-C. 2018. MobileNetV2: Inverted Residuals and Linear Bottlenecks. *arXiv preprint arXiv:1801.04381*.

Shen, C.; Wang, X.; Yin, Y.; Song, J.; Luo, S.; and Song, M. 2021. Progressive Network Grafting for Few-Shot Knowledge Distillation. In *AAAI*, volume 35, 2541–2549.

Szegedy, C.; Ioffe, S.; Vanhoucke, V.; and Alemi, A. A. 2017. Inception-v4, Inception-ResNet and the Impact of Residual Connections on Learning. In *AAAI*, 4278–4284.

Tan, M.; and Le, Q. 2019. EfficientNet: Rethinking Model Scaling for Convolutional Neural Networks. In *ICML*, volume 97, 6105–6114.

Tan, M.; and Le, Q. V. 2021. EfficientNetV2: Smaller Models and Faster Training. *arXiv preprint arXiv:2104.00298*.

Wang, H.; Zhao, H.; Li, X.; and Tan, X. 2018. Progressive Blockwise Knowledge Distillation for Neural Network Acceleration. In *IJCAI*, 2769–2775.

Yim, J.; Joo, D.; Bae, J.; and Kim, J. 2017. A Gift from Knowledge Distillation: Fast Optimization, Network Minimization and Transfer Learning. In *CVPR*, 7130–7138.

Yu, J.; Jin, P.; Liu, H.; Bender, G.; Kindermans, P.-J.; Tan, M.; Huang, T.; Song, X.; Pang, R.; and Le, Q. 2020. BigNAS: Scaling up Neural Architecture Search with Big Single-Stage Models. In *ECCV*, 702–717.

Yu, S.; Mazaheri, A.; and Jannesari, A. 2021. Auto Graph Encoder-Decoder for Neural Network Pruning. In *ICCV*, 6362–6372.

Yun, S.; Han, D.; Oh, S. J.; Chun, S.; Choe, J.; and Yoo, Y. 2019. CutMix: Regularization Strategy to Train Strong Classifiers With Localizable Features. In *ICCV*, 6022–6031.

Zhang, H.; Cisse, M.; Dauphin, Y. N.; and Lopez-Paz, D. 2018. mixup: Beyond Empirical Risk Minimization. In *ICLR*.

Zhong, Z.; Yan, J.; Wu, W.; Shao, J.; and Liu, C. 2018. Practical Block-Wise Neural Network Architecture Generation. In *CVPR*, 2423–2432.

Zhuang, J.; Tang, T.; Ding, Y.; Tatikonda, S. C.; Dvornek, N.; Papademetris, X.; and Duncan, J. 2020. AdaBelief Optimizer: Adapting Stepsizes by the Belief in Observed Gradients. In *NeurIPS*.