ECAI 2024
U. Endriss et al. (Eds.)

doi:10.3233/FAIA240747

Differentiable Neural Network for Assembling Blocks

Zhiwei Liu[a], Qiang Lu[a,*], Yibo Zhao[a], Yanhong Zhao[b] and Jake Luo[c]

[a] Beijing Key Laboratory of Petroleum Data Mining, China University of Petroleum, Beijing, China
[b]Kunlun Digital Technology Co.,Ltd., Beijing, China
[c]Department of Health Informatics and Administration, University of Wisconsin Milwaukee, Milwaukee, United States

Abstract. The goal of assembly blocks is to select blocks from pre-trained neural network (NN) models and combine them into a new NN for a different dataset. By reusing the weights of these blocks, training the NN with the new dataset becomes cost-effective. To achieve this goal, we propose an end-to-end differentiable neural network called PA-DNN. PA-DNN consists of two modules: a partition NN module and an assembly NN module. For the new dataset, the partition NN module divides existing pre-trained NN models into blocks. The assembly NN module then selects some of these blocks and combines them into a new NN using a stitching component. To train PA-DNN, we design a score function that evaluates the performance of each new NN generated by PA-DNN. The evaluated value is used to train the partition NN module. Additionally, two loss functions are created to train the assembly NN module and the stitching component in the new NN, respectively. After the training process, PA-DNN infers a new NN, and only the stitching component of the NN is fine-tuned with the new dataset. Experiments show that, compared to manual models, neural architecture search, and the assembly model DeRy, PA-DNN can generate a more accurate and lightweight NN with lower training costs.

1 Introduction

Model repositories, such as HuggingFace [31] and MMPretrain [5], store numerous pre-trained neural network (NN) models manually crafted by AI experts. Although these pre-trained NN models perform well on their respective training datasets, their performance may degrade when encountering a new dataset. To adapt to the new dataset, these models need to be manually modified and/or retrained, which can be a waste of human and computational resources. The goal of assembling blocks is to reduce this waste by automatically selecting blocks from pre-trained NN models in the model repository and assembling them into a new NN. By keeping the weights of the assembled blocks, training the new NN with the new dataset becomes cost-effective.

Assembling NN blocks differs from neural architecture search (NAS) [27], which focuses on creating NNs with new structures. Instead, assembling NN blocks aims to reuse existing blocks. Traditional assembly NNs, such as NN2 [12] and SBNN [10], which originated from biological inspiration, build an NN by dynamically adjusting the connection weights between blocks. However, these methods need to deal with the interconnection of all blocks, which becomes infeasible for current model repositories that store many large NN models. Computing the interconnection is impossible because it requires all blocks from these NN models. To address this issue, the assembling block method DeRy [34] was proposed to reuse some blocks. DeRy first splits pre-trained NN models into blocks based on their structures. It then selects some blocks and combines them into an NN using stitching layers. However, the combined NN still requires full training to adapt to the target tasks. Recently, EvoLLM [1] used evolutionary algorithms to automatically combine blocks from different large language models. However, this split is static and not related to the new dataset, which could lead to suboptimal performance when building an accurate NN for the new dataset.

To reuse NN blocks and their weights, we propose an end-to-end differentiable NN model called PA-DNN. PA-DNN consists of two main components: (1) a partition NN module and (2) an assembly NN module, as shown in Fig. 1. The partition NN module includes an embedding block and n segmentation point distribution (SPD) blocks. The embedding block is extracted from a fixed NN model and is used to capture the features of the new dataset. Each SPD block learns the distribution of segmentation points in a pre-trained NN model from the model repository. The assembly NN module consists of K router blocks. Similar to the mixture of experts (MOE) approach used in large language models [9], each router block learns the distribution of choosing a split block for each layer based on the output of the previously chosen block.

PA-DNN utilizes the partition NN module to split pre-trained NN models into blocks based on the characteristics of the new dataset. The assembly NN module then selects and combines some of these blocks into a new NN. During training, PA-DNN employs a score function to evaluate the performance of the combined NN. The evaluation result is used as a loss function to guide the partition NN module in outputting the correct split blocks. Additionally, PA-DNN uses the training and validation loss functions [21] of the combined NN to train the assembly NN module and the stitching layers in the combined NN, respectively. After training, PA-DNN samples a combined NN and fine-tunes the stitching layers in the NN to adapt to the new dataset.

The main contributions of our work can be summarized as follows:

- We propose PA-DNN, an end-to-end differentiable NN model consisting of two differentiable NN modules: the partition NN module and the assembly NN module. To generate an accurate combined NN, the partition NN module learns the distribution of segmentation points, while the assembly NN module learns the distribution of split blocks. The end-to-end and differentiable nature of PA-DNN allows for easy adjustment of the combined NN

* Corresponding Author. Email: luqiang@cup.edu.cn.

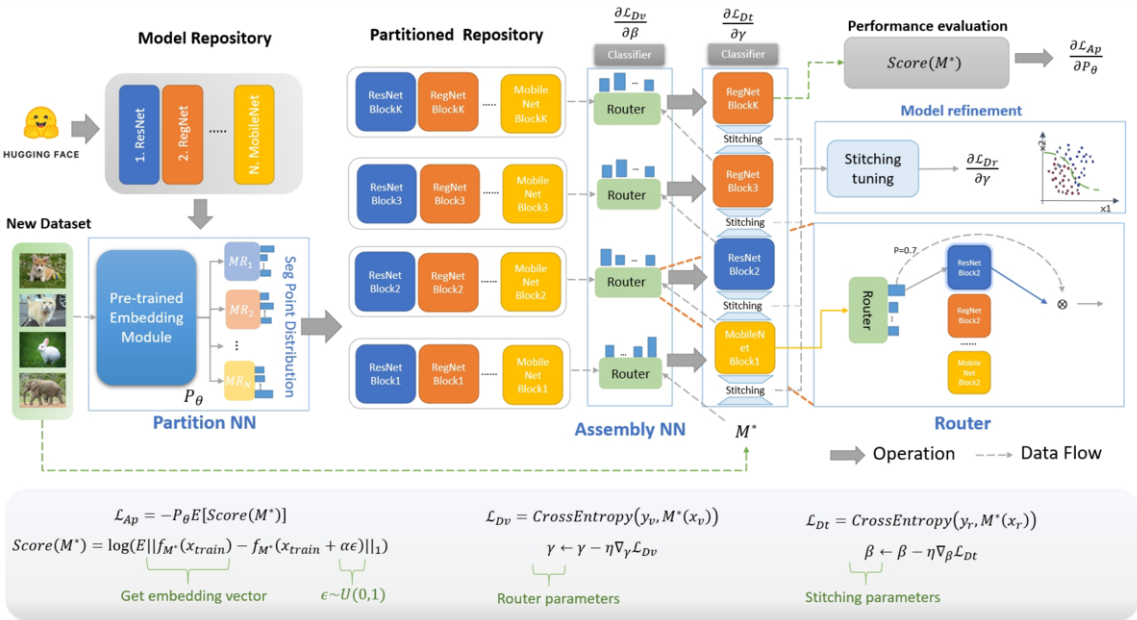

Figure 1. PA-DNN framework.

to fit the new dataset.

- We introduce a scoring method that efficiently evaluates the performance of the combined NN. This scoring method only requires the output of the combined NN on a subset of samples, eliminating the need for training the combined NN during evaluation.
- Experimental results demonstrate that PA-DNN outperforms nine baseline methods by generating a more accurate and lightweight NN with lower training costs. On the CIFAR10 benchmark, PA-DNN assembles an NN that achieves 95.22% accuracy by fine-tuning only 2.28M parameters, whereas the manually designed ResNet-152 requires training 58.35M parameters to achieve 95.37% accuracy. Similarly, on the CIFAR100 benchmark, PA-DNN's combined NN achieves 82.73% accuracy by fine-tuning 14.10M parameters, while ResNet-152 requires training 58.35M parameters to achieve 80.86% accuracy. By fine-tuning only the stitching network parameters, the combined NN incurs a lower training cost compared to full training. Experiments show that stitching layer fine-tuning reduces the training time by 23% compared to full training.

2 Related Work

2.1 Neural architecture search

Neural Architecture Search (NAS) [27] is a technique that aims to discover new neural network architectures within a predefined search space. The goal is to achieve high performance after full training. NAS is typically divided into two main categories: global network search [2, 37, 32] and cell network search [38, 25, 26].

Global network search involves directly exploring the entire architecture of a neural network. However, as the network depth increases, the search space expands exponentially. For instance, in ENAS [25], a neural network with a search depth of 12 can potentially generate around 10^{29} different network architectures, significantly increasing the search space. In contrast, cell network search focuses on identifying individual cell networks and constructs the final architecture by stacking multiple instances of these cell networks in a predefined manner. Unlike global network search, cell network search is not constrained by network depth, which effectively reduces the search space. However, due to the reduced search space, cell network search may not discover as diverse and rich network architectures as global network search.

Neural architecture search often comes with a high computational cost. For instance, NASNet [38] and AmoebaNet-A [26] required approximately 2,000 and 3,150 GPU days, respectively, to search for optimal network architectures using reinforcement learning and evolutionary algorithms. Although various techniques have been proposed to reduce search costs, such as accuracy prediction [20, 23] and differentiable search [21, 33, 7], the searched NN does not necessarily achieve the best accuracy after retraining, which can be attributed to the random initialization of network block weights. Unlike NAS, which aims to create new networks, neural network assembly focuses on reusing pre-trained network blocks to reduce search and training costs.

2.2 Assembly neural network

Assembly neural networks imitate the assembly and interconnection between neurons in the biological brain. Neurons are organized into blocks, with dense connections within blocks and relatively sparse connections between them. Assembly neural networks often possess self-organizing learning capabilities, automatically adjusting connection weights and network architecture based on input data characteristics. The Hebbian learning rule [4] states that "cells that fire together, wire together," meaning that connections between neurons that are activated simultaneously will be strengthened. In assembled neural networks, Hebbian learning rules adjust the connection weights between neurons, reinforcing the connections between co-activated neurons.

Goltsev [11] proposed a texture segmentation method utilizing assembled neural networks, employing competitive learning mechanisms and Hebbian learning rules to learn texture features in images. They also introduced a modular neural network approach based on

Hebbian learning rules [12], addressing issues such as low learning efficiency and poor generalization ability in traditional neural networks when handling complex tasks. Ferigo and Iacca [10] introduced a self-building neural network that utilizes Hebbian learning rules to adjust neuron connections and conducts network branch reduction operations based on connection weights.

Assembling neural networks update parameters with dense internal connections, making them unsuitable for large-scale pre-trained model repositories. In contrast, the proposed method exhibits sparsity characteristics, enabling its effective use in large-scale pre-trained model repositories.

Large language models (LLM) have recently achieved breakthrough results in various NLP fields. Akiba et al. [1] used evolutionary algorithms to merge multiple LLMs to build a new large language model and achieve greater performance improvements. Specifically, two model merging methods are adopted: directly reusing a certain module or summing the weights of multiple modules and then reusing them. However, this method only applies to homogeneous model collections and is unsuitable for heterogeneous network model collections.

DeRy et al. [34] split the heterogeneous network model repository into network blocks, selected some blocks for assembly, but fully trained the assembled models to adapt to the target data set. The model constructed by the PA-DNN method only needs to fine-tune the stitching layer to adapt to the target data set.

2.3 Model stitching

Model stitching typically involves using 1×1 convolutional layers to establish connections between distinct neural networks. Lenc and Vedaldi [18] effectively merged the initial and final layers of two pre-trained networks using a 1×1 convolutional layer, noting no significant performance degradation. Bansal et al. [3] demonstrated that stitching layers could connect models trained with varying architectures or training methodologies with minimal impact on their performance. Pan et al. [24] employed stitching layers to connect the front and rear segments of models from two distinct pre-trained model families, such as DeiT-Base and DeiT-Large [30]. Previous studies focused on the stitching of two models. Yang et al. [34] partitioned the heterogeneous pre-trained network model repository and reconstituted it using 1×1 convolutions. In contrast to prior research, the model stitching layer proposed in this paper does not have fixed hyperparameters, automatically constructing convolution or deconvolution layers based on input and output dimensions.

3 PA-DNN Architecture

3.1 Partition NN

The partition NN module is designed to automatically divide pre-trained network models into multiple blocks based on a validation dataset, denoted as D_v, which is a subset of the overall dataset D. The module consists of two main components: a pre-trained embedding block and n segmentation point distribution (SPD) blocks. The pre-trained embedding block can be any suitable pre-trained neural network model from the repository that has the same input format as the target dataset, such as the number of channels and image dimensions.

Each SPD block is a fully connected neural network that is responsible for splitting a pre-trained NN model with $k+1$ layers into multiple blocks. The SPD block has k outputs, where k is the number of potential segmentation points in the pre-trained NN model. Each

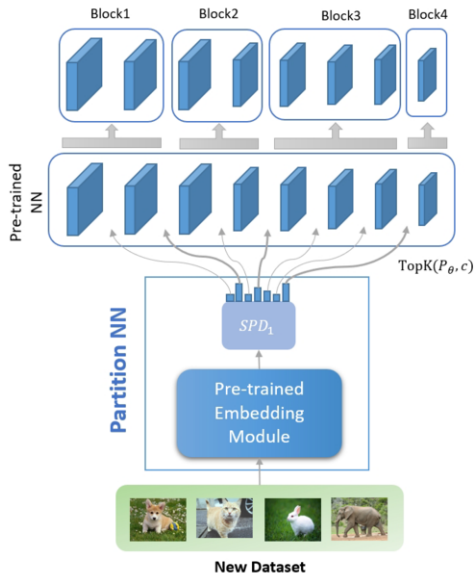

Figure 2. Network partitioning pipeline.

output o_i represents the probability P_i of separating the i-th layer and the $(i+1)$-th layer, which is calculated using the below function:

$$P_i = \frac{exp(o_i)}{\sum_{i=1}^{k} exp(o_i)}. \tag{1}$$

The function ensures that the sum of all probabilities is equal to 1 and that each probability is non-negative. The SPD block essentially learns the distribution of segmentation points for the pre-trained NN model.

To split a pre-trained neural network model into $c+1$ blocks, the top c probability values are selected from the k outputs of the SPD block. The pipeline for partitioning the pre-trained neural network is illustrated in Figure 2. For example, if the SPD output is $[0.1146, 0.1698, 0.0910, 0.1571, 0.1229, 0.1025, 0.2421]$ and $c = 3$, then the top three probability values of 0.1698, 0.1571, and 0.2421 are selected. This splits the neural network model into four blocks by separating the layers at indices 1, 3, and 6.

The Partition NN module divides n pre-trained NN models into $(c+1)\times n$ blocks, as shown in Figure 1. These blocks are represented by $M[c+1][n]$, where $M[i]$ is the set of the i-th blocks from the n pre-trained NN models.

3.2 Assembly NN

The Assembly NN module consists of $c+1$ router blocks and $c+1$ stitching blocks. The i-th router block is responsible for selecting one block from the block set $M[i]$ based on the output of the previous block selected by the $(i-1)$-th router block, as illustrated in Figure 3. The router block architecture is inspired by the Switch Transformer encoder block [9] and includes self-attention, softmax, and normalization layers. However, unlike the Switch Transformer encoder block, the input to the router block is a batch of data rather than a single data point. The self-attention mechanism in the router block focuses on capturing relationships between different data points within the batch. To aggregate these relationships, the router block introduces an average layer denoted by $'\oplus'$. The functionality of the router

block can be expressed by the following equation:

$$P_{Router} = Softmax(\frac{1}{n}\sum_{i=1}^{n} Attention(Norm(Flatten(H_1, H_2...H_n) \cdot \epsilon))) \quad (2)$$

where H represents a batch of previous block output, ϵ is a random noise to achieve the load balance of these candidate blocks [9].

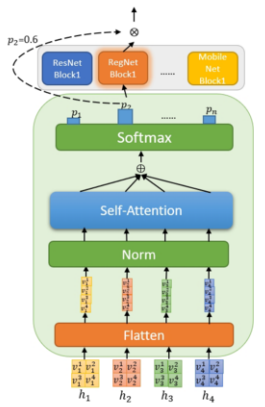

Figure 3. Router block.

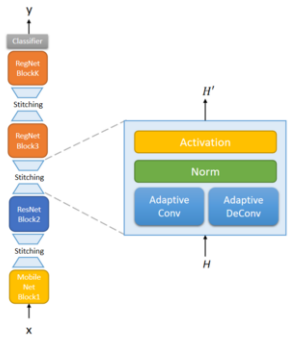

Figure 4. Stitching layer.

The stitching block constructs a stitching layer between two selected blocks, as shown in Figure 4. The stitching layer fulfills two requirements: preserving features and aligning different dimensional vectors. To preserve features, the stitching layer employs either convolution or deconvolution operations. If the input size of the current block is smaller than the output size of the previous block, the stitching layer applies convolution; otherwise, it applies deconvolution. To align different dimensional vectors, the parameters of the convolution or deconvolution operations are determined by the following function: $Conv(c_i, c_j, 3, \lceil w_i/w_j \rceil, \lceil h_i/h_j \rceil)$ or $DeConv(c_i, c_j, 3, \lceil w_j/w_i \rceil, \lceil h_j/h_i \rceil)$, where c_i, w_i and h_i are the number of channels, width, and height of the previous block output, respectively. c_i, w_i, and h_i are the current block input. $\lceil w_i/w_j \rceil$ is the step width while $\lceil h_i/h_j \rceil$) is the step height.

3.3 Training

The training of PA-DNN consists of three main steps: splitting pre-trained NN models (lines 14, 23), training the partition NN module (lines 17-21), and training the assembly NN module (lines 5-10), as outlined in Algorithm 1. These three steps are executed iteratively until PA-DNN converges.

3.3.1 Training Partition NN

The partition NN module P_θ generates the probability distribution of segmentation points, which effectively serves as the policy for splitting pre-trained NN models. The output of a combined NN M^* is considered the reward r for a segmentation operator sampled from the probability distribution P_θ. To train the partition NN module, we employ the Proximal Policy Optimization (PPO) algorithm with clipped objective [29], as defined by the following objective function:

$$\underset{\theta}{\operatorname{argmax}} \left(A(r)(min(\frac{P_\theta}{P_{\theta'}}, clip(\frac{P_\theta}{P_{\theta'}}, 1-\xi, 1+\xi)) \right) \quad (3)$$

Algorithm 1 PA-DNN algorithm

Input: $Dt = \{x_t, y_t\}, Dv = \{x_v, y_v\}$
Output: M^*

1: $P_\theta = PartitionNN$
2: $Q_\beta = AssemblyNN$
3: **Function** assembly(x_t, y_t, x_v, y_v):
4: /* training assembly NN module */
5: **while** Q_β is not converge **do**
6: $M^* = Q_\beta(x_v)$ (3.2)
7: $\gamma \leftarrow \gamma - \eta \nabla_\gamma \mathcal{L}(M^*(x_t), y_t)$ (6)
8: $\beta \leftarrow \beta - \eta \nabla_\beta \mathcal{L}(M^*(x_v), y_v)$(5)
9: **end while**
10: **return** M^*
11: **for** 1 to max_epoch **do**
12: **for** x_t, y_t, x_v, y_v in Dt, Dv **do**
13: /* splitting NNs */
14: $M[c+1][n] \leftarrow Sample(P_\theta(x_v), c)$
15: $M^* =$ assembly (x_t, y_t, x_v, y_v)
16: /* training partition NN module*/
17: $r = Score(M^*)$ (3.3.1)
18: **for** 1 to T **do**
19: $J(\theta) = -A(r)(min(\frac{P_\theta}{P_{\theta'}}, clip(\frac{P_\theta}{P_{\theta'}}, 1-\xi, 1+\xi))$ (3)
20: $\theta \leftarrow \theta - \eta \nabla_\theta J(\theta)$
21: **end for**
22: /* evaluate partition NN module */
23: $M[c+1][n] \leftarrow TopK(P_\theta(x_v), c)$
24: $M^* =$ assembly (x_t, y_t, x_v, y_v)
25: /* save best M^* by Score */
26: **end for**
27: **end for**
28: **return** M^*

where $P_{\theta'}$ is the probability of receiving reward r, $P_\theta / P_{\theta'}$ is importance sampling that evaluates the importance of P_θ to reward r, $A(r)$ represents the advantage function [28] of reward r, and ξ is a predefined threshold.

Obtaining the reward r of the combined NN is time-consuming due to the process of training the combined NN M^*. Inspired by Zen-Score [19], we propose a simple scoring method that obtains the reward r via some samples without training M^*. The scoring method is computed by the following Equation.

$$score(M^*) = log(\frac{1}{n}\sum_{i=1}^{n} ||f_{M^*}(x_i) - f_{M^*}(x_i + \alpha\epsilon)||_F), \quad s.t. \epsilon \sim U(0,1) \quad (4)$$

where x is a batch of samples in the training dataset D_t, M^* is the combined NN with random weights in stitching layers, f_{M^*} is the output from the last selected block in the combined NN. For the combined NN in Figure 1, f_{M^*} is the output of the k-th RegNet block segmented by partition NN module from pre-training NN models. α is a scaling factor, and ϵ is random noise. As $||f_{M^*}(x_i) - f_{M^*}(x_i + \alpha\epsilon)||_F$ is the feature distance between two outputs of f_{M^*} via sample x_i and its neighbor sample $x_i + \alpha\epsilon$, the small score shows the combined NN M^* generate similar features for similar input samples. So, the larger score is better.

3.3.2 Training Assembly NN

The goal of training the assembly NN module is to ensure that the combined NN M^* performs well on the validation dataset D_v after

Table 1. Comparison of Manual models, NAS, DeRy, and PA-DNN.

Method	Training Params (M)	Acc on CIFAR-10 (%)	AUC on CIFAR-10 (%)	Acc on CIFAR-100 (%)	AUC on CIFAR-100 (%)
ResNet-34	21.34	94.08	99.52	77.98	99.18
ResNet-50	23.51	94.17	99.59	78.63	99.15
ResNet-101	42.52	94.81	99.66	79.87	99.19
ResNet-152	58.35	95.37	99.68	80.86	98.67
RegNetY-800mf	6.30	95.35	99.73	79.53	99.28
RegNetY-3.2GF	20.60	96.05	99.71	80.81	99.39
MobileNet-V3-Large	5.40	94.07	99.72	75.50	99.04
Manual model * [34]	25.43 ± 6.68	94.84 ± 0.27	99.66 ± 0.03	79.03 ± 0.66	99.13 ± 0.08
AmoebaNet-A	3.2	96.66	-	81.07	-
NASNet-A	3.3	97.35	-	83.18	-
PNAS	3.2	96.59	-	80.47	-
ENAS	4.6	96.46	-	80.57	-
DARTS(1st)	3.4	97.00	-	82.46	-
DARTS(2nd)	3.3	97.24	-	83.03	-
NAS * [26, 38, 20, 25, 21]	3.5 ± 0.20	96.88 ± 0.14	-	81.80 ± 0.46	-
DeRy(4,10,3)	7.64	95.84	99.83	82.67	99.56
DeRy(4,20,5)	14.19	96.32	99.71	83.10	99.21
DeRy(4,30,6)	24.89	96.42	99.71	84.05	99.67
DeRy(4,50,10)	40.41	97.07	99.80	84.25	99.63
DeRy * [34]	21.78 ± 6.20	96.41 ± 0.22	99.76 ± 0.03	83.52 ± 0.33	99.52 ± 0.09
PA-DNN(run 1)	7.78	94.44	99.70	-	-
PA-DNN(run 2)	1.34	94.58	99.76	-	-
PA-DNN(run 3)	1.65	95.05	99.77	-	-
PA-DNN(run 4)	2.28	95.25	99.59	-	-
PA-DNN(run CIFAR10)	3.26 ± 1.32	94.83 ± 0.17	99.71 ± 0.04	-	-
PA-DNN(run 1)_full-train	40.54	96.55	99.72	-	-
PA-DNN(run 2)_full-train	12.62	96.57	99.80	-	-
PA-DNN(run 3)_full-train	12.61	96.77	99.80	-	-
PA-DNN(run 4)_full-train	9.90	96.75	99.86	-	-
PA-DNN(run CIFAR10)_full-train	18.92 ± 6.27	96.66 ± 0.05	99.80 ± 0.02	-	-
PA-DNN(run 5)	3.40	-	-	79.39	99.46
PA-DNN(run 6)	10.13	-	-	80.60	99.53
PA-DNN(run 7)	17.31	-	-	81.52	99.47
PA-DNN(run 8)	14.11	-	-	82.73	99.51
PA-DNN(run CIFAR100)	11.24 ± 2.60	-	-	81.06 ± 0.61	99.49 ± 0.01
PA-DNN(run 5)_full-train	15.19	-	-	82.91	99.41
PA-DNN(run 6)_full-train	31.22	-	-	83.02	99.45
PA-DNN(run 7)_full-train	34.84	-	-	82.88	99.63
PA-DNN(run 8)_full-train	31.73	-	-	83.91	99.66
PA-DNN(run CIFAR100)_full-train	28.25 ± 3.83	-	-	83.18 ± 0.21	99.54 ± 0.05

* The results of manual models, NAS, and DeRy are from the corresponding papers.

being trained on the training dataset D_t. To achieve this, we utilize the validation dataset D_v to compute the loss function of M^*, which is then used to update the weights in the assembly NN module. This process guides the selection of candidate blocks for the combined NN. On the other hand, the training dataset D_t is employed to update the weights of the stitching layers, as proposed in [21]. Consequently, training the assembly NN module does not involve updating the weights of the selected blocks. The process of updating weights in different components of the NN is described by the following equations:

$$\beta \leftarrow \beta - \eta \nabla_{\beta} \mathcal{L}(y_v, M^*(x_v)) \tag{5}$$

$$\gamma \leftarrow \gamma - \eta \nabla_{\gamma} \mathcal{L}(y_t, M^*(x_t)) \tag{6}$$

where (x_v, y_v)(or (x_t, y_t)) is the batch of data in D_v (or D_t), β are weights in the assembly NN module, γ are weights in stitching layers, η is the learning rate, and $\mathcal{L}$ denotes the cross entropy loss function.

Table 2. Model Repository Setting.

Id	Model Architecture	Pre-trained Dataset	Number of layers
1	ResNet-34	ImageNet-1k	16
2	ResNet-50	CIFAR10	16
3	ResNet-50	CIFAR100	16
4	ResNet-101	ImageNet-1k	20
5	ResNet-152	ImageNet-1k	24
6	RegNetY_3_2gf	ImageNet-1k	18
7	RegNet_Y_800mf	ImageNet-1k	14
8	MobileNet_v3_Large	ImageNet-1k	15

4 EXPERIMENTAL

4.1 Experiment Setup

To facilitate our experiments, we compiled a repository of pre-trained models by acquiring eight convolutional network architectures from TorchVision[1] and MMPretrain[2]. This repository encompasses vari-

[1] https://pytorch.org/vision/stable/models.html
[2] https://mmpretrain.readthedocs.io/en/latest/index.html

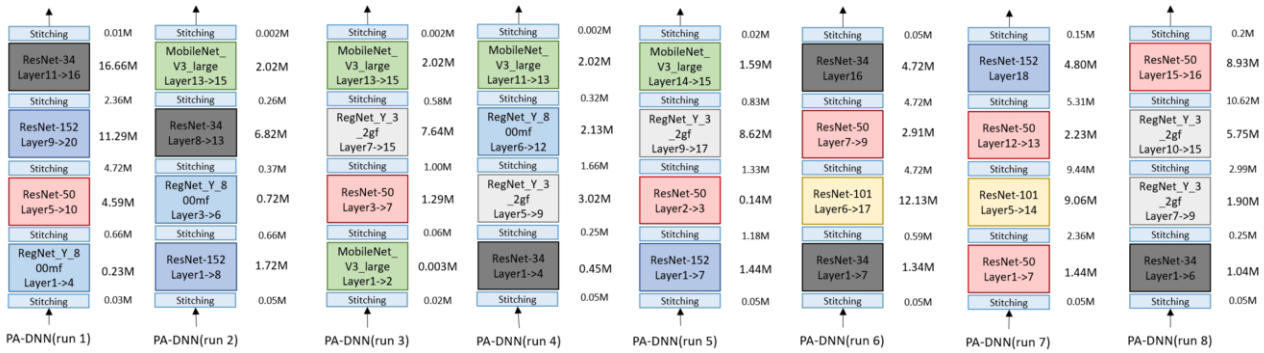

Figure 5. Combined NNs found by PA-DNN. The first four NNs are found in CIFAR-10 while the last four NNs found in CIFAR-100.

ous models, specifically ResNet, MobileNet, and RegNet, which are detailed in Table 2. These models have been previously trained on three major datasets: ImageNet-1k [6], CIFAR-10 [17], and CIFAR-100 [17].

4.2 Experimental Setting

The CIFAR-10 and CIFAR-100 datasets were employed as the benchmark for image classification in our experiments. Each dataset comprises 50,000 training images and 10,000 test images. The training images are equally divided into two subsets: a training set and a validation set, each containing 25,000 images.

For the training of PA-DNN, we adopted a batch size of 64. The partition NN module is trained using the Adam optimizer [15] with a learning rate of 5×10^{-5}, while the assembly NN module employs the Stochastic Gradient Descent (SGD) optimizer with the learning rate 0.01. The stitching layers in the combined neural network are fine-tuned using the AdamW optimizer [22] with the learning rate 1×10^{-3}. To enhance model robustness and generalization, we also applied data augmentation techniques, specifically Mixup [36] and CutMix [35], during the training of PA-DNN.

PA-DNN runs one each of two benchmarks four times. For each combined NN found by PA-DNN, fine-tuning was performed on stitching layers and the entire network, respectively, denoted as **PA-DNN** and **PA-DNN_full-train**. PA-DNN is compared with nine baseline methods. These methods fall into three categories: the manual models –ResNet[13], RegNet[16] and MobileNet[14], the NAS method –AmoebaNet-A[26], NASNet-A[38], PNAS[20], ENAS[25] and DARTS[21], and the assembly block method –DeRy[34].

4.3 Results

Our evaluation involved a comparative analysis between manual models, NAS methods, DeRy models, and PA-DNN combined NNs. The evaluation metrics include search cost, number of training parameters, the number of parameters, model accuracy, and AUC on the CIFAR datasets, as detailed in Table 1. The combined NN found by PA-DNN is illustrated in Figure. 5.

Compared with manual models, PA-DNN achieves combined NN with lower training costs, as shown in Figure 6. For example, for the CIFAR-10 dataset, the PA-DNN (run 3) obtains an accuracy of 95.05% by updating only 1.65 million parameters. In contrast, the traditional manual model, ResNet-152, reached a slightly higher accuracy of 95.37% but required training a substantial 58.35 million parameters. For the CIFAR-100 dataset, PA-DNN (run 8) gets an accuracy of 82.73% by training only 14.10 million parameters, surpassing the performance of more extensive models like ResNet-152, which achieved an accuracy of 80.86% with 58.35 million parameters. However, the combined NNs found by PA-DNN could be large, leading to high inference costs. For example, the PA-DNN (run 3) parameters are 12.61M, while the number of MobileNet parameters is 5.40 M.

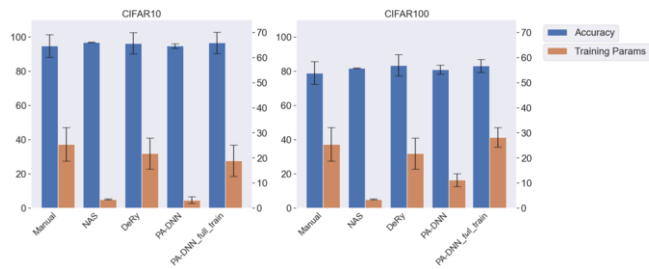

Figure 6. Accuracy and training parameters.

Although the combined NN by PA-DNN is usually larger than NAS, PA-DNN has a smaller search cost and better performance on the CIFAR-100 dataset. NAS autonomously constructs each layer of an NN, generally resulting in the NN with a smaller number of parameters. This method incurs significant computational costs, as evidenced by the extensive GPU days required for training, such as 3150 GPU days for AmoebaNet-A [26] and 2000 GPU days for NASNet-A [38], as shown in Table 3. As the complexity of models increases, the search space expands exponentially, leading to increased computational demands. For instance, ENAS, which performs a global architecture search, although faster, involves densely connected modules that consume considerable GPU resources. Similarly, DARTS [21], a differentiable NAS, offers quicker search capabilities but suffers from the drawback of updating all parameters simultaneously during the search process. In contrast, PA-DNN utilizes pre-trained model blocks, which typically possess more parameters, resulting in generally larger combined NN compared to those designed by NAS. Despite this, PA-DNN requires a significantly lower number of parameters to be trained, contributing to reduced computational demands relative to NAS methods. This efficiency makes PA-DNN particularly advantageous for scenarios where computational resources are a limiting factor.

PA-DNN has a smaller training cost than DeRy. For instance,

Table 3. Comparison of NAS and PA-DNN in the training cost.

Method	Search Cost (GPU days)
AmoebaNet-A	3150
NASNet-A	2000
PNAS	225
ENAS	0.5
DARTS(1st)	0.4
DARTS(2nd)	1
PA-DNN	1.8
PA-DNN_full-train	2.1

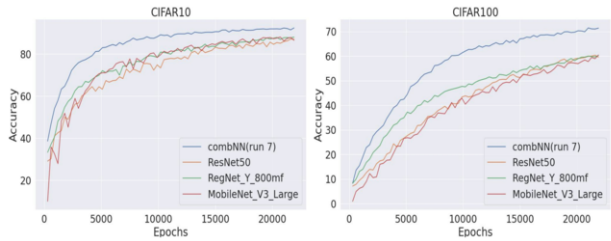

Figure 7. Validation accuracy on CIFAR.

when trained on the CIFAR-10 dataset, PA-DNN (run 3) updated 1.65M parameters and achieved an accuracy of 95.05%. In contrast, DeRy(4,10,3) trained 7.64M parameters but only achieved an accuracy of 95.84%. Furthermore, the average accuracy of PA-DNN_full-train is better than that of DeRy. However, when it comes to the CIFAR-100 dataset, the advantages of PA-DNN are not obvious. This is because the performance scoring method of the CIFAR-100 dataset is unstable.

Since the combined NN found by PA-DNN reserves weights of NN blocks from the repository of pre-trained models, training the combined NN only updates the weights of stitching layers. So, the training has high efficiency. For example, when training the combined NN found by PA-DNN on the 7th run, ResNet50, RegNet_Y_800mf, and MobileNet_V3_Large, the combined NN exhibits superior convergence and performance, as shown in Figure 7.

4.4 Performance score verification

The goal of the performance score function is to predict the accuracy of the combined NN without training. To verify the effectiveness of the performance scoring method, we conducted experiments based on the NAS-Bench-201[8]. NAS-Bench-201 is a NAS dataset that contains 15,625 different model architectures with random parameters and accuracy on the datasets CIFAR10, CIFAR100, and ImageNet16-120. 100 models are selected from NAS-bench-201 with random initialized parameters. Then, their performance scores are computed by 2048 image data in CIFAR10 and CIFAR100. Experimental results show that the score function outcome is positively correlated with the accuracy of the trained model, i.e., models with higher performance scores have higher accuracy, as shown in Figure 8. It proves the effectiveness of the performance scoring method, which can guide the training of the partition NN module.

5 Discussion

As indicated in Table 1, PA-DNN is capable of discovering a more compact combined model with lower training costs. Unlike manual models that rely on human-designed architectures and lack specific optimizations for new datasets, PA-DNN can discover thinner models through the partitioning capabilities of its partition NN module.

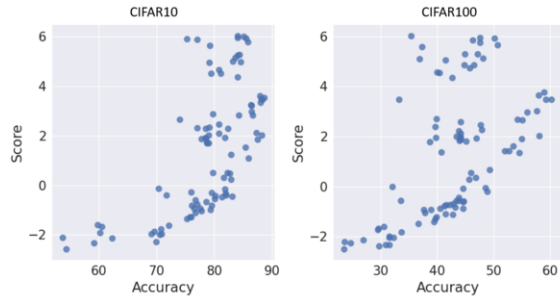

Figure 8. The relationship between model accuracy and score function.

This module divides models into smaller blocks, which are then dynamically selected by the assembly NN module based on dataset specifics, resulting in better overall performance.

In contrast to NAS, which often involves extensive search costs due to its broad and deep search spaces, PA-DNN benefits from the differentiability architecture and the sparsity of its assembly NN module. This leads to a reduced search cost, making PA-DNN more efficient in finding the suitable architecture of the combined NN.

PA-DNN also shows advantages over the DeRy method in terms of training costs. PA-DNN's innovative stitching layer design, which offers superior feature conversion capabilities, allows the system to fine-tune only the stitching layer parameters rather than retraining all parameters as DeRy requires. This approach significantly reduces the computational expense.

However, PA-DNN does not always hold an absolute advantage in terms of accuracy. Since most parameters of the combined NN are frozen, and only a select few within the stitching layers are updated, the accuracy may not always reach the highest potential. Enhancements in stitching layer design or additional fine-tuning of pre-trained blocks could potentially improve results. Additionally, the high variance in accuracy observed in PA-DNN models, as shown in Figure 6, might be attributed to two main factors: the sparsity in policy sampling and the instability in performance scoring. The large sampling space in our experiments—24 probabilities out of 139—could be better managed with policy optimization techniques suited for extensive action spaces. Moreover, as depicted in Figure 8, the current performance scoring method's instability could lead to inaccuracies in model evaluation, affecting the policy updates. Adopting more stable performance evaluation techniques could further enhance the effectiveness of PA-DNN.

6 Conclusion

We propose a novel method called PA-DNN that uses an end-to-end differentiable neural network to assemble NN blocks from the NN model repository. PA-DNN consists of two main components: the partition NN module and the assembly NN module. The two components are responsible for model partitioning and network assembly, respectively. Compared to manual models, NAS methods, and DeRy models, PA-DNN is more efficient in discovering thinner combined NNs with smaller search and training costs and better model performance. Although the combined NN has fewer parameters, it still exhibits high accuracy. Our future research is dedicated to designing more powerful stitching layer structures and more accurate model performance scoring methods. Additionally, we will extend our research on neural network assembly to multi-task and multi-modal scenarios.

References

[1] T. Akiba, M. Shing, Y. Tang, Q. Sun, and D. Ha. Evolutionary optimization of model merging recipes. *arXiv preprint arXiv:2403.13187*, 2024.

[2] B. Baker, O. Gupta, N. Naik, and R. Raskar. Designing neural network architectures using reinforcement learning. *arXiv preprint arXiv:1611.02167*, 2016.

[3] Y. Bansal, P. Nakkiran, and B. Barak. Revisiting model stitching to compare neural representations. *Advances in neural information processing systems*, 34:225–236, 2021.

[4] T. H. Brown, E. W. Kairiss, and C. L. Keenan. Hebbian synapses: biophysical mechanisms and algorithms. *Annual review of neuroscience*, 13(1):475–511, 1990.

[5] M. Contributors. Openmmlab's pre-training toolbox and benchmark, 2023.

[6] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei. Imagenet: A large-scale hierarchical image database. In *2009 IEEE conference on computer vision and pattern recognition*, pages 248–255. Ieee, 2009.

[7] X. Dong and Y. Yang. Searching for a robust neural architecture in four gpu hours. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 1761–1770, 2019.

[8] X. Dong and Y. Yang. Nas-bench-201: Extending the scope of reproducible neural architecture search. *arXiv preprint arXiv:2001.00326*, 2020.

[9] W. Fedus, B. Zoph, and N. Shazeer. Switch transformers: Scaling to trillion parameter models with simple and efficient sparsity. *The Journal of Machine Learning Research*, 23(1):5232–5270, 2022.

[10] A. Ferigo and G. Iacca. Self-building neural networks. *arXiv preprint arXiv:2304.01086*, 2023.

[11] A. Goltsev. An assembly neural network for texture segmentation. *Neural Networks*, 9(4):643–653, 1996.

[12] A. Goltsev and V. Gritsenko. Modular neural networks with hebbian learning rule. *Neurocomputing*, 72(10-12):2477–2482, 2009.

[13] K. He, X. Zhang, S. Ren, and J. Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

[14] A. Howard, M. Sandler, G. Chu, L.-C. Chen, B. Chen, M. Tan, W. Wang, Y. Zhu, R. Pang, V. Vasudevan, et al. Searching for mobilenetv3. In *Proceedings of the IEEE/CVF international conference on computer vision*, pages 1314–1324, 2019.

[15] D. P. Kingma and J. Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[16] J. Krause, M. Stark, J. Deng, and L. Fei-Fei. 3d object representations for fine-grained categorization. In *Proceedings of the IEEE international conference on computer vision workshops*, pages 554–561, 2013.

[17] A. Krizhevsky, G. Hinton, et al. Learning multiple layers of features from tiny images. 2009.

[18] K. Lenc and A. Vedaldi. Understanding image representations by measuring their equivariance and equivalence. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 991–999, 2015.

[19] M. Lin, P. Wang, Z. Sun, H. Chen, X. Sun, Q. Qian, H. Li, and R. Jin. Zen-nas: A zero-shot nas for high-performance image recognition. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pages 347–356, 2021.

[20] C. Liu, B. Zoph, M. Neumann, J. Shlens, W. Hua, L.-J. Li, L. Fei-Fei, A. Yuille, J. Huang, and K. Murphy. Progressive neural architecture search. In *Proceedings of the European conference on computer vision (ECCV)*, pages 19–34, 2018.

[21] H. Liu, K. Simonyan, and Y. Yang. Darts: Differentiable architecture search. *arXiv preprint arXiv:1806.09055*, 2018.

[22] I. Loshchilov and F. Hutter. Decoupled weight decay regularization. *arXiv preprint arXiv:1711.05101*, 2017.

[23] J. Mellor, J. Turner, A. Storkey, and E. J. Crowley. Neural architecture search without training. In *International Conference on Machine Learning*, pages 7588–7598. PMLR, 2021.

[24] Z. Pan, J. Cai, and B. Zhuang. Stitchable neural networks. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 16102–16112, 2023.

[25] H. Pham, M. Guan, B. Zoph, Q. Le, and J. Dean. Efficient neural architecture search via parameters sharing. In *International conference on machine learning*, pages 4095–4104. PMLR, 2018.

[26] E. Real, A. Aggarwal, Y. Huang, and Q. V. Le. Regularized evolution for image classifier architecture search. In *Proceedings of the aaai conference on artificial intelligence*, volume 33, pages 4780–4789, 2019.

[27] P. Ren, Y. Xiao, X. Chang, P.-Y. Huang, Z. Li, X. Chen, and X. Wang. A comprehensive survey of neural architecture search: Challenges and solutions. *ACM Computing Surveys (CSUR)*, 54(4):1–34, 2021.

[28] J. Schulman, P. Moritz, S. Levine, M. Jordan, and P. Abbeel. High-dimensional continuous control using generalized advantage estimation. *arXiv preprint arXiv:1506.02438*, 2015.

[29] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*, 2017.

[30] H. Touvron, M. Cord, M. Douze, F. Massa, A. Sablayrolles, and H. Jégou. Training data-efficient image transformers & distillation through attention. In *International conference on machine learning*, pages 10347–10357. PMLR, 2021.

[31] T. Wolf, L. Debut, V. Sanh, J. Chaumond, C. Delangue, A. Moi, P. Cistac, T. Rault, R. Louf, M. Funtowicz, et al. Huggingface's transformers: State-of-the-art natural language processing. *arXiv preprint arXiv:1910.03771*, 2019.

[32] L. Xie and A. Yuille. Genetic cnn. In *Proceedings of the IEEE international conference on computer vision*, pages 1379–1388, 2017.

[33] S. Xie, H. Zheng, C. Liu, and L. Lin. Snas: stochastic neural architecture search. *arXiv preprint arXiv:1812.09926*, 2018.

[34] X. Yang, D. Zhou, S. Liu, J. Ye, and X. Wang. Deep model reassembly. *Advances in neural information processing systems*, 35:25739–25753, 2022.

[35] S. Yun, D. Han, S. J. Oh, S. Chun, J. Choe, and Y. Yoo. Cutmix: Regularization strategy to train strong classifiers with localizable features. In *Proceedings of the IEEE/CVF international conference on computer vision*, pages 6023–6032, 2019.

[36] H. Zhang, M. Cisse, Y. N. Dauphin, and D. Lopez-Paz. mixup: Beyond empirical risk minimization. *arXiv preprint arXiv:1710.09412*, 2017.

[37] B. Zoph and Q. V. Le. Neural architecture search with reinforcement learning. *arXiv preprint arXiv:1611.01578*, 2016.

[38] B. Zoph, V. Vasudevan, J. Shlens, and Q. V. Le. Learning transferable architectures for scalable image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 8697–8710, 2018.