Neural Routing by Memory

Anonymous Author(s)
Affiliation
Address
email

Abstract

Recent Convolutional Neural Networks (CNNs) achieve significant success by stacking multiple convolutional blocks, named procedures in this paper, to extract semantic features. However, they use the same procedure sequence for all inputs in feature extraction, regardless of the intermediate features. This paper proposes a simple but effective idea of increasing the procedures and assigning similar intermediate features to the same specialized procedures in a divide-and-conquer fashion. It relieves each procedure's learning difficulty and thus leads to better performance. Specifically, we propose a routing-by-memory mechanism to existing CNN architectures. In each stage of the network, we introduce parallel Procedural Units (PUs). One PU consists of a memory head and a procedure. The memory head maintains a type of representative feature. For an intermediate feature, we search its most similar memory and forward it to the corresponding procedure in both training and testing. In this way, different procedures are specialized to allocated features and tackle them better. Networks with the proposed mechanism can be trained efficiently using a four-step training strategy. Experimental results show that our method improves VGGNet, ResNet, and EfficientNet's accuracy on Tiny ImageNet, ImageNet, and CIFAR-100 benchmarks with negligible extra computational cost.

1 Introduction

Human memory is often understood as an informational processing system. It plays an essential role in human intelligence and comprises a sensory processor, short-term memory, and long-term memory, which inspires many well-known machine learning models, such as Recurrent Neural Networks (RNN), Long Short-Term Memory (LSTM) [12], and Neural Turing Machine (NTM) [9]. Episodic memories, a type of long-term memory, are the collection of past personal experiences. They can be retrieved and exploited by brains when tackling problems we had met. They guide our execution of specific procedures we had done before. Inspired by the observation, we introduce the routing-by-memory mechanism to the neural network. It uses memories (summaries of seen features) to guide networks to process different features by different procedures in a divide-and-conquer fashion, relieving the learning difficulty and achieving better performance. This paper applies the mechanism to the feature extraction in CNNs and calls the network taking this mechanism as Routing-by-Memory Network (RMN). We will introduce conventional CNN and our RMN in what follows.

Recent Convolutional Neural Networks (CNNs) achieve state-of-the-art results on computer vision tasks by stacking convolutional blocks (e.g., residual block [11], inception block [30], and Dense block [15]), named procedures in this paper, to extract semantic features. However, they use the same stacked blocks to process different intermediate features, despite feature large variances. Since similar intermediate features can be processed using the same way, we propose a simple but effective idea of increasing the procedures and assigning similar intermediate features to the same specialized procedures named routing-by-memory mechanism. Different procedures are trained by different

Submitted to 35th Conference on Neural Information Processing Systems (NeurIPS 2021). Do not distribute.

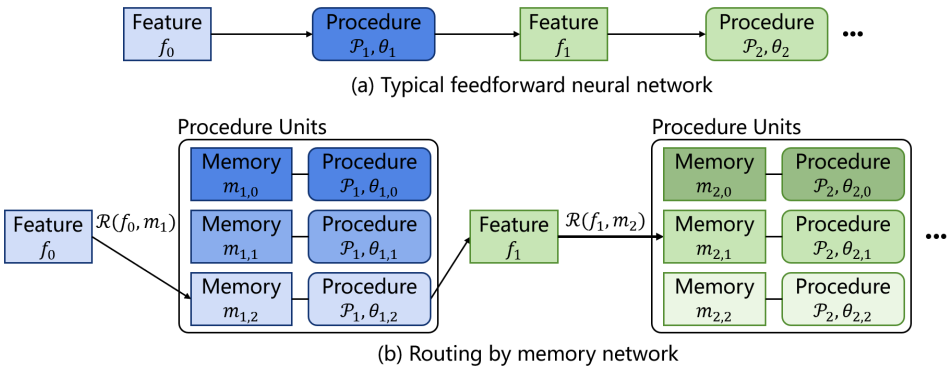

Figure 1: (a) Conventional feed-forward network. It processes the features by stacked procedures. (b) Our proposed Routing by Memory Network (RMN). It processes the features by input-dependent procedures. $\mathcal{R}$ is a routing function, and we use the nearest neighbor algorithm for it. m, $\mathcal{P}$, and θ denotes the memory, procedure architecture, and the parameters of the procedure, respectively. A Procedure Unit (PU) consists of a memory and a procedure. Given a feature that outputs from a previous stage, it searches its most similar memory and forwards the feature to the corresponding procedure. Different shades of color represent different kinds of features, memories, and procedures.

intermediate features, relieving each procedure's learning difficulty and improving performance. Besides, the routing is conducted by non-parametric nearest memory search, which does not bring extra routing networks and easy to train.

In our RMN, we introduce a simple but effective mechanism, routing by memory, to existing CNN architectures. Briefly, we introduce the Procedure Unit (PU) to process features (see Figure 1 for an illustration). It consists of a memory (a representative feature) with a procedure (some convolutional blocks). We use the memory to identify which type of features the corresponding procedure is expected to handle. Specifically, we split the network into different stages by downsampling layers. There are multiple PUs in each stage. All procedures within a stage use the same architecture but different parameters. In a stage, given an intermediate feature outputted from the previous stage, we will search its nearest memory and forward it to the corresponding procedure. In this way, different PUs are specialized to handle different types of features. Besides, in the procedures, we introduce a routing-dependent feature attention module, named Conditional Attention (CA), to improve the performance further with negligible extra computational cost.

How to initialize and update memory and procedures is the main challenge when training our RMN. We propose an easily implement training strategy that includes four training steps, stem network training, procedure cloning, memory initialization, and routing-based training. In the stem network training step, we train a conventional CNN (e.g., ResNet and EfficentNet) by a few training epochs (i.e., early stopping) as a stem network. Then in the procedure cloning step, we generate multiple procedures in each stage of the network by cloning all learned procedures a preset number of times. In the memory initialization step, we first extract intermediate features for all stages of the network. Then we use representative features as initialized memory. This paper clusters features in each stage of the network and uses cluster centers as representative features. Finally, we continue the network training as the first step in the routing-based training step, but the data flow inside the network is decided by routing results. In this step, the memory is updated in a moving average fashion while other components are updated by the original optimization method used in the first stage. The overall training strategy is plug-and-play to existing CNNs' training approach since it does not change most original hyperparameters (e.g., the number of epochs and learning rate).

This paper takes VGGNet, ResNet, and EfficientNet as backbones to train RMN. According to the experimental results, RMN significantly improves original models' results on some benchmarks while not increasing the computational cost. We summarize our contributions as follows.

- We propose a new network mechanism, named routing by memory, of assigning intermediate features to specialized procedures, which relieves each procedure's learning difficulty and thus leads to better performance.

- The proposed mechanism is plug-and-play to existing CNN architectures under an effective training strategy with four steps: stem network training, procedure cloning, memory initialization, and routing-based training.
- We take VGGNet, ResNet, and EfficientNet as backbone networks to train our networks and achieve significant improvements in the accuracies on Tiny ImageNet, Imagenet, and CIFAR-100 benchmarks while not increasing the computational cost.

2 Related Works

2.1 Convolutional Neural Network Architecture

In 2012, AlexNet [20] outperforms handcrafted features engineering by a large margin in ImageNet image classification competition. It combines feature extraction and classification in a deep CNN model to extract discriminative semantic features. After that, CNNs gradually dominate most computer vision tasks, such as image classification, object detection, and semantic segmentation. Classic CNN architecture (e.g., AlexNet [20], ZFNet [37], and VGGNet [29]) extract features by stacking convolutional layers. In 2016, ResNet [11], the most successful CNN architectures in recent years, proposes to train very deep CNNs using skip connection and stacked residual blocks. After that, more and more efficient and effective stacked block-based CNNs (e.g., MobileNet[13], SENet [14], and EfficientNet [31]) were proposed. However, they use the same block sequence for all inputs in feature extraction, regardless of the intermediate features. This paper introduces the plug-and-play routing-by-memory mechanism to existing CNN architectures. It uses memory to guide different blocks tackles different features in a divide-and-conquer fashion, which boosts accuracy while not increasing the computational cost.

2.2 Conditional Computation

Conditional computation [3] refers to activating only some of the modules in a network in an input-dependent fashion. Recent researches introduce it to CNNs to accelerate the network inference. AIG [32], BlockDrop [35], and SkipNet [33] propose to learn the subset of blocks needed to process a given input. Since easy examples may not require deep layers' features to make the classification, SACT [7], Inside Cascaded CNN [38], and Dynamic Routing [23] propose to do input-dependent early stopping in network inference. These works aim to reduce the original models' computational costs while maintaining comparable accuracies.

2.3 Mixture of Experts

Mixture of Experts (MoE) ([17], [18]) was proposed three decades ago. It is related to conditional computation and introduces multiple experts (learners) to divide the problem space into homogeneous regions. Given an input, MoE often uses a routing or gating module to select the corresponding experts. In the deep learning era, sparsely-gated mixture-of-experts layer [28] first introduces MoE to LSTM for language modeling. As for CNNs, there are two categories of MOE, dynamic parameter and dynamic architecture, which are compatible. Dynamic parameter methods [36, 4, 5] learn input-dependent convolutional kernels' parameters using parameters generation networks. Dynamic architecture methods employ multiple sub-networks and build a dynamic computational graph by decision networks. NOEF [1] and HydraNets [24] divide the class label space and assign different sets of class labels to different sub-networks for handling. DeepMoE [34] proposes to increase the kernels and select each layer's features' channels by a decision network. Runtime Routing [25] proposes to learn an RNN-based decision network by reinforcement learning for multi-path networks. Dynamic Routing [22], Routing Networks [26], ExperGate [2], and HardMoE [10] extend MoE to semantic segmentation, multi-task learning, weakly-supervised learning and lifelong learning, respectively. These decision networks increase the training difficulty and bring extra computations. Besides, since they are non-differentiable, most methods have to modify gradient descent algorithms or employ reinforcement learning.

Our RMN solves routing in a new perspective, routing by memory, rather than designing complicated decision networks to decide the routing path. It is more straightforward and elegant. Besides, since the routing is based on non-parametric nearest memory search, RMN is lightweight, easy to train, and requires no modification in the back-propagation rule.

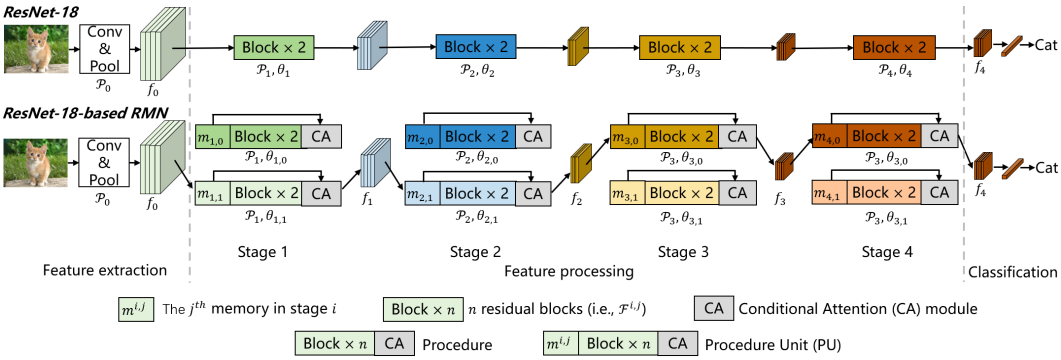

Figure 2: An illustration of ResNet-18 and ResNet-18-based 2-way RMN. We divide the network into three phases: (preliminary) feature extraction, feature processing, and classification. In the feature processing phase, we split the network into four stages by downsampling layers. RMN introduces the routing-by-memory mechanism to each stage. It uses memory to guide forwarding different features to their corresponding specialized procedures. Different colors denote different stages. Different shades of color represent different kinds of features, memories, and procedures within a stage.

3 Routing by Memory Network

This section elaborates the architecture and training strategy of our RMN.

3.1 Network Architecture

Our RMN is built by integrating the routing-by-memory mechanism into an existing CNN architecture. In this paper, we elaborate our methods mainly on the task of image classification. According to the functions, we treat the processing of a common CNN as three separated phases: (preliminary) feature extraction, feature processing, and final classification. Our RMN enhances the intermediate features in the feature processing phase by incorporating the routing-by-memory mechanism. In the following sections, we take ResNet-18 [11] as an example backbone to present how to integrate the routing-by-memory mechanism into a conventional CNN(see Figure 2).

3.1.1 Resnet

In this section, we elaborate the ResNet-18. Given an input image $\mathcal{X}$, in the first (preliminary) feature extraction phase, its feature maps f_0 is generated by a series of operations $\mathcal{P}_0$ consisting of convolution, pooling, and batch normalization, which can be formulated as follows:

$$f_0 = \mathcal{P}_0(\mathcal{X}, \theta_0), \tag{1}$$

where θ_0 denotes the learnable parameters in $\mathcal{P}$. Low-level features such as edges and corners are represented in these preliminary feature maps.

In the feature processing phase, multiple residual blocks are utilized to process the feature map f^0 for concerning the information in higher semantic levels, such as animals' fur and furniture's texture. We group the blocks into different stages according to their output feature maps' dimensions and resolutions (i.e., divide the stages by down-sampling operations). For example, as shown in Figure 2, we divide blocks into four stages by grouping two adjacent blocks because their feature maps' shapes are the same. We represent the operations (i.e., convolution blocks) in the ith stage as $\mathcal{P}_i$ with its parameters as θ_i. The feature maps outputted from i^{th} stage are f_i. Specifically, we formulate the processing of the ith stage as follows:

$$f_i = \mathcal{P}_i(f_{i-1}, \theta_i), \qquad i = 1, 2, 3, 4, \tag{2}$$

The ResNet ends with a global average pooling layer and a 1000-way (the number of classes, 1000 in ImageNet) fully-connected layer with an argmax (softmax in the model training phase) function. They are used to classify the given image in the classification phase. We formulate this phase as follows:

$$\mathcal{Y} = \underset{c \in [0,1,\ldots,C-1]}{\arg\max} \; (\mathrm{GAP}(f_4) W_c^T + b), \tag{3}$$

where GAP refers to the global average pooling in the classification phase. W and b are learnable weights and biases of the fully connected layer (i.e., classifier). c is the class index, and C is the number of classes. $\mathcal{Y}$ denotes the output label.

3.1.2 Resnet-Based RMN

In this section, we take the ResNet-18 as a backbone network to introduce our RMN. In the Resnet-18-based RMN, we propose Procedure Units (PU) in the feature processing phase. There are multiple PUs in each stage, and we use N-way RMN to represent how many PUs per stage in our RMN. All PUs within a stage uses the same architecture but will learn different parameters due to the routing mechanism. Each PU consists of two modules, including memory and procedure. The memory is a representative feature that is learned from features by a moving average fashion in the training phase (see Section 3.2 for details). We use global average pooling for memory to reduce the storage cost. In the ith stage, given the feature f_{i-1} and memories $m_i = [m_{i,0}, m_{i,2}, ..., m_{i,N-1}]$, we first do the routing $\mathcal{R}$ by the nearest neighbor searching as follows:

$$\mathcal{R}(m_i, f_{i-1}) = \underset{j \in [0,1,...,N-1]}{\arg\min} (\mathcal{D}(m_{i,j}, \mathrm{GAP}(f_{i-1}))), \tag{4}$$

where $m_{i,j}$ denotes the jth PU's memory in the ith stage. To reduce the storage cost, we apply global average pooling GAP to f_{i-1} before routing. $\mathcal{D}$ refers to a distance measurement metric, and we use Euclidean distance in this paper. Since $\mathcal{D}$ is non-parametric and fast, the computational cost of routing is negligible and can be omitted.

After the routing, we forward the features to the corresponding procedures. A procedure consists of residual blocks and an optional Conditional Attention (CA) module. The CA module is an optional module introduced to improve the accuracy further by a little additional computational cost. Without the CA module, RMN re-formulates the ith stage of the feature processing phase as follows:

$$f_i = \mathcal{P}_i(f_{i-1}, \theta_{i,\mathcal{R}(m_i, f_{i-1})}), \qquad i = 1, 2, 3, 4, \tag{5}$$

where $\theta_{i,0}, \theta_{i,0}, ..., \theta_{i,N-1}$ refers to N sets of parameters for $\mathcal{P}_i$.

We then introduce the CA module. The same channels of features outputted from different procedure units may represent different semantic meanings. For example, the first convolution kernel of a procedure may focus on animals' fur. However, the first convolution kernel of another procedure may focus on furniture's texture. The inconsistent semantic meaning of different features increases the learning difficultly. Inspired by the position coding in ViT [6], we introduce the CA module to do routing-dependent channel-wise attention to the features, relieving the inconsistency adaptively. CA module consists of a conditional input, a routing result, and a Squeeze-and-Excitation (SE) module [14] (see Figure 3 for an illustration of the CA module). Specifically, it first computes the scalar vector $s = [s_1, s_2, ..., s_K]$ that has the same number of channels as $f_i = [f_{i_1}, f_{i_2}, ..., f_{i_K}]$. This operation can be formulated as:

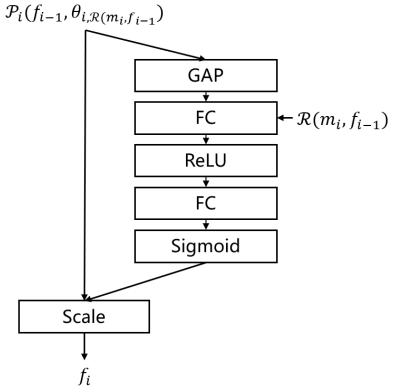

Figure 3: An illustration of Conditional Attention (CA) module. CA is based on the Squeeze-and-Excitation (SE) module but concatenates the squeezed feature with the routing result. Thus, it can do routing-dependent channel-wise attention.

$$s = Excite(Concat(Squeeze(f_i), \mathcal{R}(m_i, f_{i-1}))), \tag{6}$$

where $Squeeze$ denotes using a global average pooling followed with a fully connected layer to squeeze the feature dimension. $Excite$ denotes using a fully connected layer and a Sigmoid function to expand the feature dimension and compute attention values. $Concat$ denotes channel-wise concatenation operation. Please note that all CA modules share parameters within a stage.

Then we scale the features f_i according to the scalar vector s by channel-wise multiplication (we use $*$ to represent it). The CA module can be formulated as follows:

$$CA(f_i) = s * f_i, \tag{7}$$

In summary, when using the CA module, RMN re-formulates the ith stage of the feature processing phase as follows:

$$f_i = s * \mathcal{P}_i(f_{i-1}, \theta_{i,\mathcal{R}(m_i, f_{i-1})}), \quad i = 1, 2, 3, 4 \tag{8}$$

3.2 Training Strategy

In this section, we take the ResNet-18-based RMN as an example to introduce the four-step training strategy. Unlike the conventional training pipeline, after a few training epochs, we insert two steps to expand procedures and initialize memories. In the last step, we update memories in a moving average fashion. The training strategy is plug-and-play to existing CNNs' training approach since it does not change most original hyperparameters (e.g., the number of epochs and learning rate) and can use a standard gradient descent algorithm.

3.2.1 Stem Network Training

In the first step, we aim to train a conventional CNN (e.g., VGG, ResNet, and EfficientNet), named stem network in this paper, to extract reasonable features for memory initialization. Specifically, in this step, we train the standard ResNet-18 by a few epochs (i.e., early stopping) and will resume the training in the fourth step. If using CA modules, we use random routing results $r \sim \mathcal{U}(1, N)$ for each CA module, where $\mathcal{U}$ denotes the discrete uniform distribution.

3.2.2 Procedure Cloning

We clone the procedures in the second step. Specifically, we clone the parameters of all procedures by N times (i.e., assign θ_i to $\theta_{i,0}$,$\theta_{i,2}$,...,$\theta_{i,N-1}$). In this way, we can build N branch procedure units in each stage of the feature processing phase. Please note that we also clone the momentum of the gradient in this step to maintain the training's consistency in the fourth phase.

3.2.3 Memory Initialization

We initialize the memories using representative features. In this paper, we compute the representative features by cluster analysis. Specifically, in the ith stage of feature processing phase,we extract each training sample's f_i. Then we use the Euclidean distance-based k-means clustering algorithm for the features and get N clusters. Then we initialize each memory in m_{i+1} by each cluster' center. In this way, different memories can dominate different kinds of features.

3.2.4 Routing-Based Training

In this step, we resume the model training. For the memory, inspired by the update of mean value in Batch Normalization (BN) [16], we update the memory in a moving-average fashion. Specifically, in a training iteration and in the ith feature processing stage, given H samples that are routed to j^{th} PU, we have their features as $f_i^1, ..., f_i^H$. we update the memory as follows:

$$m_{i,j} = \alpha m_{i,j} + (1 - \alpha) \frac{\sum_{h=1}^{H} \mathrm{GAP}(f_i^h)}{H} \tag{9}$$

where α is a hyper-parameter and denotes the momentum.

In this step, the computational graph (i.e., data flow) is dynamic due to the routing mechanism. But, the routing-by-memory mechanism uses the non-parametric nearest neighbor algorithm to select the procedures, and the memory is updated in a moving-average fashion. So, we can use the original stochastic gradient descent algorithm with softmax cross-entropy loss function to train our RMN as the original ResNet-18.

4 Experiments

In this section, we will first introduce our experimental setup. Then we present ablation experiments on our proposed main components and hyper-parameters. Finally, we show our results on some image classification benchmarks.

We do supplementary experiments about memory visualization, runtime analysis, comparisons with other related MOE methods, and generalization on other vision tasks in our ***supplementary materials***.

4.1 Experimental Setup

In this section, we first introduce the dataset we used. Then we introduce the network architectures. Finally, we present the training details of our RMN.

4.1.1 Data

In this paper, we take three image classification benchmarks, Tiny ImageNet [21], ImageNet 2012 [27], and CIFAR-100 [19] to evaluate our method. ImageNet 2012 consists of 1.2M training images and 50,000 validation images for 1000 classes. Tiny ImageNet is a subset of ImageNet. It consists of 200 classes, and each class has 500 training images, 50 validation images. CIFAR-100 consists of 100 classes, and each class has 500 training images, 100 validation images. We use Tiny ImageNet for ablation experiments in Section 4.2. In Section 4.3, we show our results of all three benchmarks.

4.1.2 Network Architectures

Our RMN can be applied to most existing CNNs architectures, and we take some widely-used architectures, VGG, ResNet, and EfficientNet in our experiments. In the ablation experiments, we take ResNet-18 as the backbone network to train our RMN. In the section of evaluations on all benchmarks, we present the results of using all architectures. For VGG-16, we split the network by pooling layers to different stages and build the PUs. We also use batch normalization for VGG-16. Since the image resolutions of Tiny ImageNet and CIFAR-100 are 64×64, and 32×32, respectively, we drop the first two down-sampling operations for ResNet and EfficientNet when training on these two benchmarks. For VGGNet, we drop the first two down-sampling operations on Tiny ImageNet but keep them on CIFAR-100. For the ImageNet, we use the resolution of 224×224.

4.1.3 Training Details

Learning rate. We first follow the warmup strategy [8] to increase the learning rate from 1e-5 to 0.48 in the first five epochs. Then we use the cosine learning rate strategy for the rest of the epochs, and we decrease the learning rate to 1e-5 in the final epoch.

Training epochs. For CIFAR-100 and Tiny ImageNet, the total number of training epochs for VGG-16, ResNet-18, ResNet-50, and EfficientNet-B0 are 120, 120, 200, and 300, respectively. For ImageNet, we use 160, 160, 220, and 400 epochs for them, respectively. We take the first 40 epochs as the step of stem network training.

The number of PUs. In this paper, N denotes the number of PUs per feature processing stage. High N can improve the model capacity but bring more parameters. Seeing Section 4.2 for the ablation experiments on N. We set $N = 8$ for accuracy and cost trade-off in other experiments. Please also note that, compared with memory consumption on feature extraction, the parameters consume little memory (less than 1% overall consumption when inference on ImageNet with the batch size as 100). So, our extra GPU memory consumption is negligible. Besides, extra parameters bring extra storage costs. But storage is abundant in real-world applications since the disks are cheap.

Batch Size. We use batch size 256 for the all baseline networks. For our RMN, in each feature processing stage, there are multiple PUs, and different features from the same batch are fed to different PUs. So, the batch size of each PU is much smaller than the total batch size, and thus we have to enlarge the batch size to train our RMN. Simply, we multiply the original batch size (256) by $N/2$ though the number of data among different PUs are imbalanced, which means we use $N = 8$ and batch size 1024 in this paper. Please see Section 4.2 for the ablation experiment of batch size.

Momentum in Memory Updating. The momentum α is a hyper-parameter used to make the memory updating smooth and stable. However, too high momentum will make the memory out of date, while too small momentum leads the memory updating unstable. We set it to 0.9 in our paper. Please see Section 4.2 for the its ablation experiment.

Others. Regarding other training details, we use stochastic gradient descent with momentum 0.9 and weight decay 1e-5 for ResNet and VGGNet. We use RMSprop optimizer for EfficientNet. We use Synchronized Batch Normalization (SyncBN) supported by the Nvidia APEX library. For CA modules, we use the reduction ratio of 16. For data augmentation, we use random augmentation introduced in ResNet [11] paper. We use 8$\times$V100 (32GB memory version) with PyTorch.

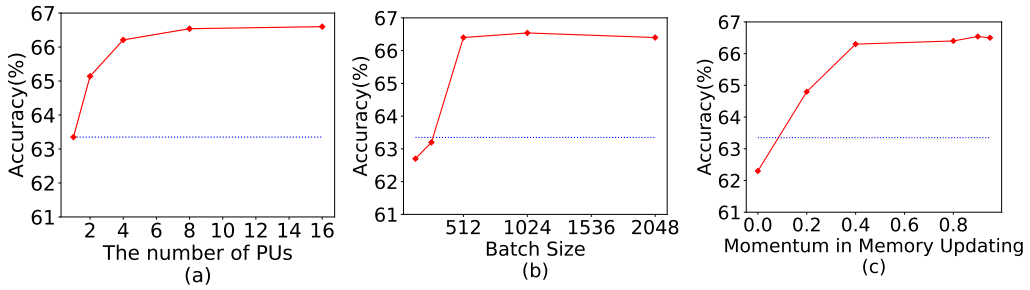

Figure 4: Experimental results on Tiny ImageNet. The blue dashed line denotes the baseline ResNet-18 with CA modules (without conditional routing inputs). (a) More PUs bring higher accuracy, and the accuracy got converged on 8 PUs, so we set it to 8 in other experiments. (b) Since RMN assigns different samples to different PUs, it requires a large batch size to maintain the training stability. We set it to 1024 in other experiments. (c) Higher momentum makes the memory updating more smooth and stable and leads to better results.

4.2 Ablation Experiments

4.2.1 The Number of PUs

The key component in our RMN is the PU. So N, the number of PUs, is a critical hyper-parameter. More PUs will improve the model's capacity and leads to higher accuracy. From Figure 4 (1), we can see that the accuracy increase along with larger N and got converged when $N = 8$. However, larger N increases the number of parameters and requires a larger batch size (batch size$= 256 \times N/2$). We make an accuracy and cost trade-off and set N to 8 in other experiments.

4.2.2 Batch Size

Our RMN introduces multiple branches with different PUs in the network, and the examples are assigned to the different branches. So, RMN requires a larger batch size than conventional CNN. According to the results in Figure 4 (b), too small batch sizes (128 or 256) bring even lower accuracy than the baseline, which is because the memory updating and routing are unstable. Thus, network training becomes more difficult. Moreover, with the 128 batch size, we found about 40% PUs died after 50 training epochs, which means these PUs' memories are far from current features distribution, and thus no data can be assigned to these PUs. We consider that the small number of data assigned to the PUs will make the memory updating unstable and draw the corresponding memory features to the outliers. Too large batch size (2048) also leads to not good results. Simply, we multiply the original batch size (256) by $N/2$ though the numbers of data among different PUs are imbalanced, which means we set the batch size to 1024 in other experiments.

4.2.3 Momentum in Memory Updating

We use the moving average to update the memories. Thus the momentum α is a key hyper-parameter. Too small α will lead the memories to change quickly. The higher the α is, the more stable and smooth the memory updating becomes, while too large α causes a very slow updating. According to Figure 4 (c), we set the α to 0.9 in other experiments.

4.2.4 Conditional Attention

Conditional Attention (CA) modules are optional modules to further improve accuracy by introducing little extra computational cost. According to the results in Table 1, it improves the accuracy by 2.2%. We also conduct ablation on the conditional routing inputs and found it can improve the accuracy by 1.0%, demonstrating conditional inputs' effectiveness. We also show the results

Model	Acc.	#Params	#FLOPs
ResNet-18	61.6%	11.3M	2.25B
ResNet-18 with CA Modules†	63.3%	11.4M	2.26B
Our RMN without CA Modules	64.3%	89.4M	2.25B
Our RMN with CA Modules†	65.5%	89.5M	2.26B
Our RMN with CA Modules	66.5%	89.5M	2.26B

Table 1: Ablations of the CA module on Tiny ImageNet. †denotes does not use conditional routing inputs.

of ResNet-18 with CA modules (does not use conditional routing inputs) for a reference, which is equivalent to SE module.

4.3 Evaluation on Benchmarks

We evaluate our methods on three image classification benchmarks: Tiny ImageNet, ImageNet, and CIFAR-100. According to the evaluation results (Table 2) on Tiny ImageNet, our RMN significantly improves the accuracy while not increasing the computational cost. Besides, the improvements for VGGNet and ResNet are much more significant than EfficientNet. It is because EfficientNet's architecture is already well-designed, compact, and contains SE modules. Thus, our method may bring some redundancies, but it still improves the accuracy by 1.8%. Seeing from the results (Table 3) on the more challenging benchmark, ImageNet, our method still performs well. Moreover, using our RMN, ResNet-50 can outperform EfficientNet-B0. For the CIFAR-100 benchmark, although the image resolution is small, 32×32, our method can still bring considerable improvements. Moreover, ResNet-18-based RMN achieves impressive accuracy and even outperforms the original ResNet-50.

Model	Acc.	#Params	#FLOPs
VGG-16 [29]	63.3%	135.0M	14.10B
ResNet-18 [11]	61.6%	11.3M	2.25B
ResNet-50 [11]	67.2%	23.9M	5.25B
EfficientNet-B0 [31]	67.0%	4.3M	0.47B
Our RMN (VGG-16) *	65.1%	237.7M	14.10B
Our RMN (ResNet-18)*	64.3%	89.4M	2.25B
Our RMN (ResNet-50)*	69.1%	188.4M	5.25B
Our RMN (EfficientNet-B0)*	68.5%	32.4M	0.47B
Our RMN (VGG-16)	66.5%	237.8M	14.11B
Our RMN (ResNet-18)	66.4%	89.5M	2.26B
Our RMN (ResNet-50)	69.7%	189.1M	5.26B
Our RMN (EfficientNet-B0)	68.8%	32.5M	0.47B

Table 2: Evaluation on Tiny-ImageNet. * denotes without CA modules.

Model	Acc.	#Params	#FLOPs
VGG-16 [29]	72.9%	138.3M	15.51B
ResNet-18 [11]	70.7%	11.7M	1.81B
ResNet-50 [11]	76.1%	25.6M	4.11B
EfficientNet-B0 [31]	76.5%	5.3M	0.39B
Our RMN (VGG-16)*	74.8%	241.0M	15.51B
Our RMN (ResNet-18)*	72.7%	89.8M	1.81B
Our RMN (ResNet-50)*	77.8%	190.0M	4.11B
Our RMN (EfficientNet-B0)*	77.9%	33.4M	0.39B
Our RMN (VGG-16)	76.1%	241.1M	15.52B
Our RMN (ResNet-18)	73.8%	89.9M	1.82B
Our RMN (ResNet-50)	78.3%	190.7M	4.12B
Our RMN (EfficientNet-B0)	78.1%	33.5M	0.40B

Table 3: Evaluation on ImageNet. * denotes without CA modules.

Model	Acc.	#Params	#FLOPs
VGG-16 [29]	73.2%	33.8M	0.43B
ResNet-18[11]	75.5%	11.2M	0.56B
ResNet-50[11]	77.6%	23.7M	1.31B
EfficientNet-B0[31]	78.0%	4.2M	0.12B
Our RMN (VGG-16)*	74.6%	136.7M	0.43B
Our RMN (ResNet-18)*	76.8%	88.3M	0.56B
Our RMN (ResNet-50)*	78.4%	183.8M	1.31B
Our RMN (EfficientNet-B0)*	78.6%	32.3M	0.12B
Our RMN (VGG-16)	75.8%	136.8M	0.45B
Our RMN (ResNet-18)	77.9%	88.4M	0.57B
Our RMN (ResNet-50)	78.9%	184.6M	1.32B
Our RMN (EfficientNet-B0)	78.8%	32.4M	0.12B

Table 4: Evaluation on CIFAR-100. * denotes without CA modules.

5 Conclusion

In this paper, we proposed a specific mechanism, routing by memory, for conventional feed-forward networks. We integrated it with the existing CNN architectures and formed the Routing by Memory Network (RMN). Specifically, it introduces the Procedure Unit (PU) to the CNNs, which consists of a memory (a representative feature) with a procedure (some convolutional blocks). We employed memories to forward different features to their expert PUs. Networks with the proposed mechanism can be trained efficiently using a four-step training strategy. According to the results on Tiny ImageNet, ImageNet, and CIFAR-100, our RMN significantly improves the VGG-16, ResNet-18, ResNet-50, and EfficientNet-B0 while not increasing the computational cost.

Limitation analysis. The main limitations of our method are extra parameters and larger training batch sizes. But they are not critical. First, extra parameters are inevitable for MOE-related methods. However, as we discuss in Section 4.1.3, compared with memory consumption on feature extraction, the model parameters consume little memory. Besides, in many real scenarios, the model storage cost is also unimportant. Second, the larger training batch size ($4\times$ for 8-way RMN) is necessary, but V100 (32GB) is enough to train most existing CNN models. Besides, some techniques can use larger batch sizes in limited memory, such as gradient accumulation, memory-saving CNN training framework (e.g., MXNet), and dynamic memory allocation.

References

[1] Karim Ahmed, Mohammad Haris Baig, and Lorenzo Torresani. Network of experts for large-scale image categorization. In *European Conference on Computer Vision*, pages 516–532. Springer, 2016.

[2] Rahaf Aljundi, Punarjay Chakravarty, and Tinne Tuytelaars. Expert gate: Lifelong learning with a network of experts. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 3366–3375, 2017.

[3] Yoshua Bengio, Nicholas Léonard, and Aaron Courville. Estimating or propagating gradients through stochastic neurons for conditional computation. *arXiv preprint arXiv:1308.3432*, 2013.

[4] Yinpeng Chen, Xiyang Dai, Mengchen Liu, Dongdong Chen, Lu Yuan, and Zicheng Liu. Dynamic convolution: Attention over convolution kernels. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 11030–11039, 2020.

[5] Zhourong Chen, Yang Li, Samy Bengio, and Si Si. You look twice: Gaternet for dynamic filter selection in cnns. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 9172–9180, 2019.

[6] Alexey Dosovitskiy, Lucas Beyer, Alexander Kolesnikov, Dirk Weissenborn, Xiaohua Zhai, Thomas Unterthiner, Mostafa Dehghani, Matthias Minderer, Georg Heigold, Sylvain Gelly, et al. An image is worth 16x16 words: Transformers for image recognition at scale. *arXiv preprint arXiv:2010.11929*, 2020.

[7] Michael Figurnov, Maxwell D Collins, Yukun Zhu, Li Zhang, Jonathan Huang, Dmitry Vetrov, and Ruslan Salakhutdinov. Spatially adaptive computation time for residual networks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 1039–1048, 2017.

[8] Priya Goyal, Piotr Dollár, Ross Girshick, Pieter Noordhuis, Lukasz Wesolowski, Aapo Kyrola, Andrew Tulloch, Yangqing Jia, and Kaiming He. Accurate, large minibatch sgd: Training imagenet in 1 hour. *arXiv preprint arXiv:1706.02677*, 2017.

[9] Alex Graves, Greg Wayne, and Ivo Danihelka. Neural turing machines. *arXiv preprint arXiv:1410.5401*, 2014.

[10] Sam Gross, Marc'Aurelio Ranzato, and Arthur Szlam. Hard mixtures of experts for large scale weakly supervised vision. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 6865–6873, 2017.

[11] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

[12] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.

[13] Andrew G Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. Mobilenets: Efficient convolutional neural networks for mobile vision applications. *arXiv preprint arXiv:1704.04861*, 2017.

[14] Jie Hu, Li Shen, and Gang Sun. Squeeze-and-excitation networks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 7132–7141, 2018.

[15] Gao Huang, Zhuang Liu, Laurens Van Der Maaten, and Kilian Q Weinberger. Densely connected convolutional networks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 4700–4708, 2017.

[16] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. In *International conference on machine learning*, pages 448–456. PMLR, 2015.

[17] Robert A Jacobs, Michael I Jordan, Steven J Nowlan, and Geoffrey E Hinton. Adaptive mixtures of local experts. *Neural computation*, 3(1):79–87, 1991.

[18] Michael I Jordan and Robert A Jacobs. Hierarchical mixtures of experts and the em algorithm. *Neural computation*, 6(2):181–214, 1994.

[19] Alex Krizhevsky, Geoffrey Hinton, et al. Learning multiple layers of features from tiny images. 2009.

[20] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. *Advances in neural information processing systems*, 25:1097–1105, 2012.

[21] Ya Le and Xuan Yang. Tiny imagenet visual recognition challenge. *CS 231N*, 7:7, 2015.

[22] Yanwei Li, Lin Song, Yukang Chen, Zeming Li, Xiangyu Zhang, Xingang Wang, and Jian Sun. Learning dynamic routing for semantic segmentation. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 8553–8562, 2020.

[23] Mason McGill and Pietro Perona. Deciding how to decide: Dynamic routing in artificial neural networks. In *International Conference on Machine Learning*, pages 2363–2372. PMLR, 2017.

[24] Ravi Teja Mullapudi, William R Mark, Noam Shazeer, and Kayvon Fatahalian. Hydranets: Specialized dynamic architectures for efficient inference. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 8080–8089, 2018.

[25] Yongming Rao, Jiwen Lu, Ji Lin, and Jie Zhou. Runtime network routing for efficient image classification. *IEEE transactions on pattern analysis and machine intelligence*, 41(10):2291–2304, 2018.

[26] Clemens Rosenbaum, Tim Klinger, and Matthew Riemer. Routing networks: Adaptive selection of non-linear functions for multi-task learning. *arXiv preprint arXiv:1711.01239*, 2017.

[27] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, et al. Imagenet large scale visual recognition challenge. *International journal of computer vision*, 115(3):211–252, 2015.

[28] Noam Shazeer, Azalia Mirhoseini, Krzysztof Maziarz, Andy Davis, Quoc Le, Geoffrey Hinton, and Jeff Dean. Outrageously large neural networks: The sparsely-gated mixture-of-experts layer. *arXiv preprint arXiv:1701.06538*, 2017.

[29] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014.

[30] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. Going deeper with convolutions. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 1–9, 2015.

[31] Mingxing Tan and Quoc Le. Efficientnet: Rethinking model scaling for convolutional neural networks. In *International Conference on Machine Learning*, pages 6105–6114. PMLR, 2019.

[32] Andreas Veit and Serge Belongie. Convolutional networks with adaptive inference graphs. In *Proceedings of the European Conference on Computer Vision (ECCV)*, pages 3–18, 2018.

[33] Xin Wang, Fisher Yu, Zi-Yi Dou, Trevor Darrell, and Joseph E Gonzalez. Skipnet: Learning dynamic routing in convolutional networks. In *Proceedings of the European Conference on Computer Vision (ECCV)*, pages 409–424, 2018.

[34] Xin Wang, Fisher Yu, Lisa Dunlap, Yi-An Ma, Ruth Wang, Azalia Mirhoseini, Trevor Darrell, and Joseph E Gonzalez. Deep mixture of experts via shallow embedding. In *Uncertainty in Artificial Intelligence*, pages 552–562. PMLR, 2020.

[35] Zuxuan Wu, Tushar Nagarajan, Abhishek Kumar, Steven Rennie, Larry S Davis, Kristen Grauman, and Rogerio Feris. Blockdrop: Dynamic inference paths in residual networks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 8817–8826, 2018.

[36] Brandon Yang, Gabriel Bender, Quoc V Le, and Jiquan Ngiam. Condconv: Conditionally parameterized convolutions for efficient inference. *arXiv preprint arXiv:1904.04971*, 2019.

[37] Matthew D Zeiler and Rob Fergus. Visualizing and understanding convolutional networks. In *European conference on computer vision*, pages 818–833. Springer, 2014.

[38] Kaipeng Zhang, Zhanpeng Zhang, Hao Wang, Zhifeng Li, Yu Qiao, and Wei Liu. Detecting faces using inside cascaded contextual cnn. In *Proceedings of the IEEE International Conference on Computer Vision*, pages 3171–3179, 2017.

Checklist

1. For all authors...
   (a) Do the main claims made in the abstract and introduction accurately reflect the paper's contributions and scope? [Yes]
   (b) Did you describe the limitations of your work? [Yes]
   (c) Did you discuss any potential negative societal impacts of your work? [N/A]
   (d) Have you read the ethics review guidelines and ensured that your paper conforms to them? [Yes]
2. If you are including theoretical results...
   (a) Did you state the full set of assumptions of all theoretical results? [N/A]
   (b) Did you include complete proofs of all theoretical results? [N/A]
3. If you ran experiments...
   (a) Did you include the code, data, and instructions needed to reproduce the main experimental results (either in the supplemental material or as a URL)? [No]
   (b) Did you specify all the training details (e.g., data splits, hyperparameters, how they were chosen)? [Yes]
   (c) Did you report error bars (e.g., with respect to the random seed after running experiments multiple times)? [No]
   (d) Did you include the total amount of compute and the type of resources used (e.g., type of GPUs, internal cluster, or cloud provider)? [Yes]
4. If you are using existing assets (e.g., code, data, models) or curating/releasing new assets...
   (a) If your work uses existing assets, did you cite the creators? [Yes]
   (b) Did you mention the license of the assets? [No]
   (c) Did you include any new assets either in the supplemental material or as a URL? [No]
   (d) Did you discuss whether and how consent was obtained from people whose data you're using/curating? [No]
   (e) Did you discuss whether the data you are using/curating contains personally identifiable information or offensive content? [No]
5. If you used crowdsourcing or conducted research with human subjects...
   (a) Did you include the full text of instructions given to participants and screenshots, if applicable? [N/A]
   (b) Did you describe any potential participant risks, with links to Institutional Review Board (IRB) approvals, if applicable? [N/A]
   (c) Did you include the estimated hourly wage paid to participants and the total amount spent on participant compensation? [N/A]