Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Institut für
Technische Informatik und
Kommunikationsnetze

Understanding Permutations in Deep Neural Network

Semester Thesis

Yifan Lu
yiflu@student.ethz.ch

Computer Engineering and Networks Laboratory
Department of Information Technology and Electrical Engineering
ETH Zürich

Supervisors:
Zhongnan Qu
Prof. Dr. Lothar Thiele

July 4, 2021

Acknowledgements

I would like to thank my supervisor Zhongnan Qu for his dedicated support and guidance. Zhongnan continuously provided encouragement and was always willing and enthusiastic to assist in any way he could despite his busy schedules. I would also like to thank Prof. Dr. Lothar Thiele who gave me this opportunity to undertake this project on the topic of neural network compression, so that I could be exposed to this rapidly developing research area. Finally, many thanks to coworkers of the lab that have provided advice for this project.

Abstract

Training a neural network is synonymous with learning the values of the weights. In contrast, motivated by the observation that the weights follow very similar distributions before and after training, we hypothesised that training a neural network can be accomplished by reshuffling the weights without ever modifying their values. To assess our hypothesis, we propose *index optimizer* that aims to optimize the locations of weights rather than their values. With the help of straight-through estimator (STE)[1], our proposed optimizer achieves competitive performance on MNIST[2], CIFAR10[3] and ImageNet[4], under different initialization distributions including sparse weight initialization. This provides a different angle to address the problem of training the sparse neural network from scratch.

Additionally, we study the permutations of input and present a draft pipeline called *sortGAN* to recover the original image from randomly shuffled pixels. It shows the potential possibility to solve very large Jigsaw puzzles with generative adversarial nets (GAN) and might be full of practical value in archaeological and biological applications. [1]

[1]Code and experiment results are available at: https://gitlab.ethz.ch/tec/students/projects/2021/sa_yiflu/-/tree/master/workspace.

Contents

Chapter 1

Introduction

The recent advances of optimization techniques and great progresses in parallel computation have made deep neural network an irreplaceable choice to tackle problems from different fields, such as computer vision [5, 6] and natural language processing[7, 8]. Their vast scope of applications, spanning entertainment, health-care and education, have raised great demands to deploy deep neural networks on edge devices. Yet, the state-of-the-art architectures need extensive computational resources and considerable storage and memory bandwidth to be trained, which prevents them from being incorporated into mobile and embedded devices.

Numerous researches have been done to enable compression of over-parameterized neural network[9] by pruning parameters [10–14], weight quantization [15–20] and low-rank approximation[21–24]. The key insight behind these approaches is to encode the well-trained neural network into a more compressed form while maintaining the most important knowledge. However, some recent works find that the initialized neural network itself already contains some useful information without training [25–27].

Correspondingly, we observe that the weights inside neural network follow very similar distributions before and after training. Fig.1 shows the distribution of weights from LeNet5 [28] and VGG [6] before and after training respectively. Initialized with Kaiming normal distribution[29], the weights after training also follow a normal distribution with mean value around zero. Starting from the considerations that normal distributions with same mean values can be transformed with each other by multiplying a scalar and classification problems are robust under scaled parameters, we hypothesize that a classification network can be trained by reshuffling the weights without ever modifying weight values.

To verify our hypothesis, we propose *index optimizer* that trains neural network by reshuffling the weights without updating their values. Compared with *LaPerm* [30], we solve their problem of choosing sync period and improve the robustness and performance by borrowing STE from [1] and applying it during reshuffling procedure. Our proposed index optimizer achieves competitive

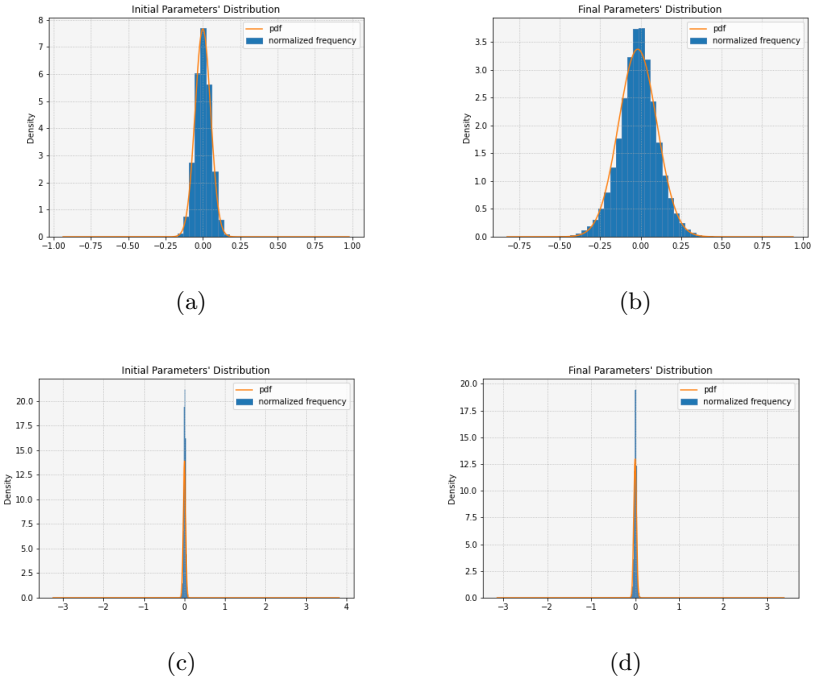
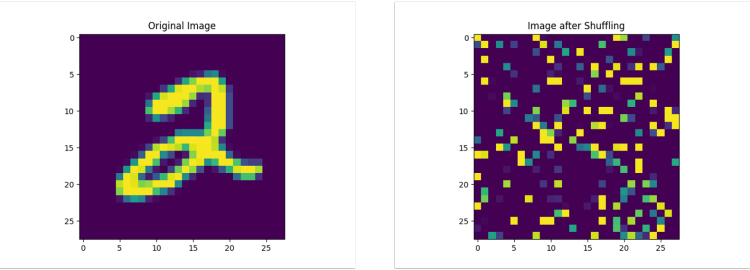


Figure 1.1: Distribution of weights. 1.1(a) and 1.1(c) show the initial weight distributions for LeNet5 and VGG. 1.1(c) and 1.1(d) show weight distributions after training for LeNet5 and VGG.

performance on MNIST[2], CIFAR10[3] and ImageNet[4] and is robust under various initialization distributions, including sparse weight initialization. This result provides us a new angle to tackle the difficulty faced by training sparse architectures from scratch [25, 26].

Apart from weight permutations, we also dive into input permutations, namely trying to recover original image from randomly shuffled pixels (see Fig.1.2). Based on GAN [31–33], we propose a prototype pipeline called *sortGAN* to rearrange the disordered pixels from the original image given class label. Although limitations exist, experiment results on MNIST and FashionMNIST[34] reveal the enormous potential of GAN to solve extremely complex Jigsaw puzzles.



(a) (b)

Figure 1.2: 1.2(a) is an image sample from MNIST dataset. 1.2(b) show the same image after randomly reshuffle the pixels. Our goal is to recover 1.2(a) given 1.2(b) and the class label.

The rest of the paper is structured as follows. In the second chapter, we list some related researches and compare our work with them. In chapter 3, we provide background knowledge used in this paper and then introduce our proposed index optimizer and sortGAN in chapter 4. Experiment results are shown and analyzed in chapter 5. In the end, we draw conclusions and raise some open questions for future work.

Chapter 2

Related Work

Deep Neural Network Compression Deep neural networks are typically over-parameterized and have significant redundancy[9]. The rising demands to realize an efficient on-device inference has attracted lots of academic attention and various approaches have been proposed such as pruning[10–14], low-rank approximation[21–24] and quantization[15–20]. The common ground behind these compression techniques is to transform the well-trained neural network into a more compressed representation that avoids important information from losing. Regarding this, our work is more related to recent ones [25–27]. Frankle *el at.*[25] explored a three-step iterative pruning strategy for finding sparse subnetworks that can be effectively trained from scratch when reset to their initialization. Zhou *et al.*[26] proposed an algorithm to identify randomly initialized neural network that achieves high accuracy without training. Ramanujan*et al.*[27] removed the stochasticity in Zhou's work and proposed edge-popup algorithm that boosts the performance significantly. Our work also focuses on distilling the information hidden in the initialization. Nevertheless, their work either iteratively updates the weight values using gradient-based optimization or maintains the initial weights without ever modifying their values and positions. In contrast, we attain the goals of training neural networks by permuting the initial weights.

LaPerm Qiu and Suda's work[30] is the closest one to ours. The idea behind both LaPerm and index optimizer is to train neural networks by permutation. However, we adopt STE and apply it to our index optimizer to improve the robustness and performance. Our index optimizer is much more stable and achieve much better performance for training sparse architecture from scratch. We also solve their problem of choosing hyper parameter, i.e. sync period k. Furthermore, we empirically prove the feasibility of LaPerm when the sync period $k = 1$, which opposes the statement in [30].

Jigsaw Puzzle Solving Jigsaw Puzzle is an $\mathcal{NP}$-complete problem. It has been commonly used as a pretext task in unsupervised learning. Doersch *et al.*[35]

pioneered this topic by proposing an architecture to solve 3×3 square-puzzles that predicts the relative position of adjacent fragments. Latterly, Noroozi *et al.*[36] proposed context-free network, a siamese-ennead CNN, to solve 3×3 square-puzzles by predicting the index assigned to Jigsaw puzzle permutations. Both of these two works treated Jigsaw puzzle as a classification problem, which restricts the number of possible re-assemblies. To remove this restriction, Wei *et al.*[37] proposed an iterative method combining predictions of pairwise relative position and absolute position. Deepzzle[38] is yet another work focused on image reassembly problem. They predicted the relative position as in [35] and then found the best reassembly using graph. Although theoretically it's possible to apply their approaches to larger Jigsaw puzzles, experiments appeared in the papers only involve 3×3 square Jigsaw puzzle. Our work can be understood as a 28×28 Jigsaw puzzle, which is much more complex and it can be potentially extended to Jigsaw puzzle with arbitrary shape. Besides, our goal is to recover the original image instead of a pretext task, therefore we also utilize class labels to provide additional information.

Chapter 3

Background

3.1 Neural Network

We consider a feed-forward network F of L fully-connected (`fc`) or convolutional (`conv`) layers. For simplicity, we omit the bias, batch normalization layers, pooling layers here.

Assume the input of the l-th layer is $\boldsymbol{x}_{l-1}$, and the output of the l-th layer is $\boldsymbol{x}_l$, which is also the input of the $(l+1)$-th layer. The weight tensor of the l-th layer is denoted as $\boldsymbol{w}_l$. Further assume the model outputs $\boldsymbol{y}$ given input data sample $\boldsymbol{x}$, i.e., $\boldsymbol{y} = F(\boldsymbol{w}; \boldsymbol{x})$. Note that $\boldsymbol{x}_0 = \boldsymbol{x}$ and $\boldsymbol{y} = \boldsymbol{x}_L$. For `conv` layers, $\boldsymbol{x}_{l-1} \in \mathbb{R}^{C_{l-1} \times H_{l-1} \times W_{l-1}}$, $\boldsymbol{x}_l \in \mathbb{R}^{C_l \times H_l \times W_l}$, and $\boldsymbol{w}_l \in \mathbb{R}^{C_l \times C_{l-1} \times S \times S}$, where H_l and W_l are the height and width of the l-th layer's output, C_l is the number of output channels, and S is the kernel size. For `fc` layers, $\boldsymbol{x}_{l-1} \in \mathbb{R}^{C_{l-1} \times 1}$, $\boldsymbol{x}_l \in \mathbb{R}^{C_l \times 1}$, and $\boldsymbol{w}_l \in \mathbb{R}^{C_l \times C_{l-1}}$. The entire architecture $F(\boldsymbol{w}; \boldsymbol{x})$ is trained based on the training dataset $\mathcal{D}^{train} = \{\boldsymbol{x}_i, \boldsymbol{z}_i\}_{i=1}^{I}$ and an appropriately chosen loss function $\mathcal{L} = \mathcal{L}(\boldsymbol{y}, \boldsymbol{z}) = \mathcal{L}(F(\boldsymbol{w}; \boldsymbol{x}); \boldsymbol{z}) = \mathcal{L}(\boldsymbol{w}; \mathcal{D}^{train})$. The objective of training is to minimize the value of the chosen loss function, i.e.

$$\min_{\boldsymbol{w}} \mathcal{L}(\boldsymbol{w}; \mathcal{D}^{train}). \tag{3.1}$$

For binary classification task, binary cross entropy (BCE) is the most commonly used loss function and it can be formulated as:

$$\mathcal{L}(\boldsymbol{y}, \boldsymbol{z}) = \sum_{i}^{I} [\boldsymbol{z}_i \cdot \log \boldsymbol{y}_i + (1 - \boldsymbol{z}_i) \log(1 - \boldsymbol{y}_i)]. \tag{3.2}$$

Otherwise for regression task, mean squared error (MSE) given in Eq.(3.3) is often used.

$$\mathcal{L}(\boldsymbol{y}, \boldsymbol{z}) = \sum_{i}^{I} (\boldsymbol{z}_i - \boldsymbol{y}_i)^2 \tag{3.3}$$

3.2 Kaiming Initialization

Weight initialization has been proven to be very important in accelerating optimization convergence and preventing layer activation outputs from exploding or vanishing during the course of a forward pass through a deep neural network[29, 39]. Our experiments are based on Kaiming initialization proposed by He *et al.* in paper [29], which is tailored for deep neural nets that use asymmetric, non-linear activation functions such as ReLU. He *et al.* derived the condition to avoid reducing or magnifying the magnitude of input signals exponentially:

$$\frac{n_l}{2} Var(\boldsymbol{w}_l^0) = 1, \tag{3.4}$$

where n_l represents the number of inputs and $\boldsymbol{w}_l^0$ represents the initial weight vector at layer l. This implies an initialization scheme of:

$$\boldsymbol{w}_l^0 \sim \mathcal{N}_k(0, \frac{2}{n_l}) \tag{3.5}$$

or

$$\boldsymbol{w}_l^0 \sim \mathcal{U}_k(-2\sqrt{\frac{3}{n_l}}, 2\sqrt{\frac{3}{n_l}}). \tag{3.6}$$

Eq.(3.5) is called *Kaiming normal distribution* and Eq.(3.6) *Kaiming uniform distribution.*

3.3 Gradient-based Optimization Algorithm

Our proposed index optimizer needs a gradient-based optimization algorithm to guide the reshuffling phase. We primarily integrate stochastic gradient descent (SGD)[40] and adaptive moment estimation (Adam)[41] into index optimizer and test them in our experiments.

SGD performs an update of weight numerical values for every mini-batch of n training examples. The updating rule can be summarized as:

$$\boldsymbol{w}^k = \boldsymbol{w}^{k-1} - lr \cdot \frac{\partial \mathcal{L}(\boldsymbol{w}^{k-1}, \mathcal{D}^k)}{\partial \boldsymbol{w}^{k-1}}, \tag{3.7}$$

where $\boldsymbol{w}^k$ indicates the weight vector during iteration k, lr indicates the learning rate, $\mathcal{L}$ is the loss function and $\mathcal{D}^k$ represents the mini-batch sampled from the training data at iteration k.

As the name suggest, while SGD use one universal learning rate, Adam adapt learning rate for each parameter. It keeps exponentially decaying averages of past gradients and squared gradients $\boldsymbol{m}^k$, $\boldsymbol{v}^k$ respectively:

$$\boldsymbol{m}^k = \beta_1 \boldsymbol{m}^{k-1} + (1 - \beta_1) \frac{\partial \mathcal{L}(\boldsymbol{w}^{k-1}, \mathcal{D}^k)}{\partial \boldsymbol{w}^{k-1}} \tag{3.8}$$

$$\boldsymbol{v}^k = \beta_2 \boldsymbol{v}^{k-1} + (1-\beta_2)(\frac{\partial \mathcal{L}(\boldsymbol{w}^{k-1}, \mathcal{D}^k)}{\partial \boldsymbol{w}^{k-1}})^2. \tag{3.9}$$

$\boldsymbol{m}^k$ and $\boldsymbol{v}^k$ are estimates of the first moment (mean) and the second moment (variance) of the gradients. As $\boldsymbol{m}^k$ and $\boldsymbol{v}^k$ are initialized as vectors of zeros, there exist a tendency for $\boldsymbol{m}^k$ and $\boldsymbol{v}^k$ to be biased towards zero, especially during the initial time steps and when the decay rates are small, i.e. β_1 and β_2 are close to one. Adam counteracts these biases by computing bias-corrected first and second moment estimates:

$$\hat{\boldsymbol{m}}^k = \frac{\boldsymbol{m}^k}{1-\beta_1^k} \tag{3.10}$$

$$\hat{\boldsymbol{v}}^k = \frac{\boldsymbol{v}^k}{1-\beta_2^k}. \tag{3.11}$$

Then parameters are updated according to the rule:

$$\boldsymbol{w}^k = \boldsymbol{w}^{k-1} - \frac{lr}{\sqrt{\hat{\boldsymbol{v}}^k} + \epsilon}\hat{\boldsymbol{m}}^k \tag{3.12}$$

3.4 Conditional Generative Adversarial Nets

GAN[31–33] is a group of different frameworks to train generative models in order to sidestep the difficulty of approximating many intractable probabilistic computations. Conditional generative adversarial nets (cGAN) is one among them that proposed in [32]. It extends both generator and discriminator of normal GAN architecture[31] to conditional models by feeding extra information $\mathbf{y}$ as additional input layer.

The architecture of cGAN is shown in Fig.3.1. In the generator the prior input noise $p_{\mathbf{z}}(\mathbf{z})$, and $\mathbf{y}$ are combined in joint hidden representation, while in the discriminator $\mathbf{x}$ and $\mathbf{y}$ are presented as inputs for the discriminative function. The objective function is:

$$\min_{\mathrm{G}} \max_{\mathrm{D}} \mathrm{V}(\mathrm{D},\mathrm{G}) = \mathbb{E}_{\boldsymbol{x}\sim p_{data}(\boldsymbol{x})}[\log \mathrm{D}(\boldsymbol{x}|\boldsymbol{y})] + \mathbb{E}_{\boldsymbol{z}\sim p_{\boldsymbol{z}}(\boldsymbol{z})}[\log(1-\mathrm{D}(\mathrm{G}(\boldsymbol{z}|\boldsymbol{y})))]. \tag{3.13}$$

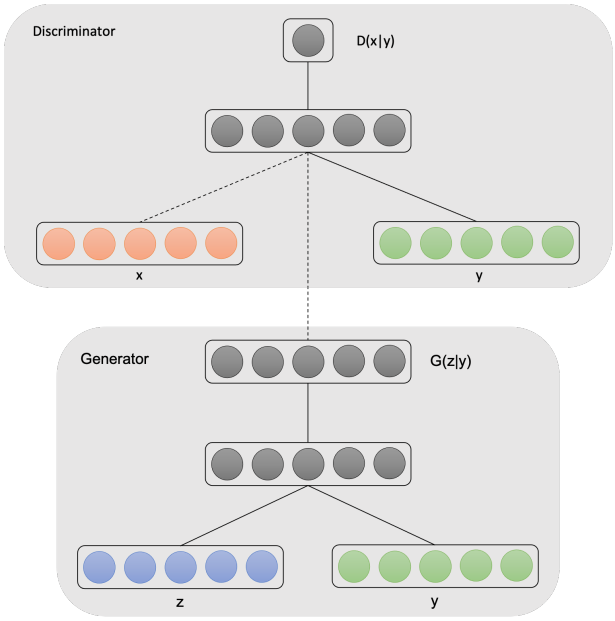


Figure 3.1: Architecture of cGAN

Chapter 4

Methodology

In this chapter, we present the proposed *index optimizer* and *sortGAN* in section 4.2 and section 4.3 respectively. In section 4.1, we introduce STE, which is of great importance for both index optimizer and sortGAN, and explain how we apply it in details.

4.1 Straight-Through Estimator

Training activation quantized neural networks involves minimizing a piecewise constant function whose gradient vanishes almost everywhere, which is undesirable for the standard back-propagation or chain rule. An empirical way to overcome this issue is to use STE[1] in the backward pass, so that the "gradient" through the modified chain rule becomes non-trivial.

Fig.4.1 shows the principle of STE. To make the gradients of threshold functions non-trivial, STE bypasses them in the backward propagation and passes on the incoming gradients as if the function was an identity function.

We adopt STE from activation quantized neural networks and apply it to both index optimizer and sortGAN by seeing the permutation procedure as quantization. More precisely, we reserve two kinds of weights — unconstrained and constrained, inside index optimizer. Unconstrained weights are updated normally by the chosen gradient-based optimization algorithm, whereas constrained weights are obtained by permuting the initial weights according to the order of unconstrained weights. This procedure can be understood as quantize arbitrary weight values to a specific ordering of the initial weights. During the forward pass constrained weights are used to make inferences, meanwhile during the backward propagation unconstrained weights are updated with gradients w.r.t. constrained weights and in turn guide the permutation of constrained weights.

Similarly for sortGAN, after new pixels are generated at output of the generator, original pixels are rearranged to be in agreement with the order of generated pixels, i.e. generated pixels are quantized using the original pixel values. During

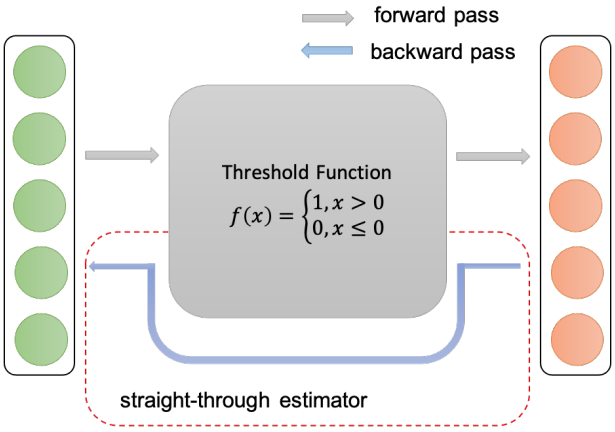


Figure 4.1: Working principle of STE.

the forward pass, rearranged original pixels are fed further into the discriminator. During the backward propagation, parameters inside the generator are updated with gradients w.r.t. rearranged original pixels.

4.2 Index Optimizer

Conventional gradient-based optimization algorithms train neural networks via updating numerical values of weights. Since we find that training only has impact on the variance of the weight distribution and classification tasks are robust under scaled parameters, we hypothesize that training of neural networks that aim at classification tasks can be accomplished by reshuffling the weights without modifying their values, i.e. by optimizing their positions instead of values.

Following the notations introduced in chapter 3, we suppose to train a feed-forward network $F(\boldsymbol{w}; \boldsymbol{x})$ with initial weights $\{\boldsymbol{w}_l^0\}_{l=1}^L$. The training problem Eq.(3.1) can be reformed as:

$$\min_{\boldsymbol{w}} \quad \mathcal{L}(\boldsymbol{w}; \mathcal{D}) \tag{4.1}$$

$$\text{s.t.} \quad \text{sort}(\boldsymbol{w}) = \text{sort}(\boldsymbol{w}^0) \tag{4.2}$$

To solve this constrained optimization problem, we propose index optimizer, whose pseudo-code is presented in Algorithm.1. Our proposed index optimizer keeps a sorted version of weight values in $\mathbf{w}_l^{\text{sorted}}$ and initialize unconstrained weights $\{\hat{\boldsymbol{w}}_l^0\}_{l=1}^L$ with $\{\boldsymbol{w}_l^0\}_{l=1}^L$ for each layer l before optimization. During each iteration k, a gradient-based optimizer opt is used to update the values of unconstrained weights $\{\hat{\boldsymbol{w}}_l^k\}_{l=1}^L$ (pseudo-code line.10), which are subsequently used

as references to permute the constrained weights $\{\boldsymbol{w}_l^k\}_{l=1}^L$ (pseudo-code line.11). opt can be chosen from any gradient-based optimization algorithm. In our experiment, we use SGD and Adam. Hence, opt can be replaced by the corresponding part from Eq.(3.7) or Eq.(3.12). To note is that, by seeing the reshuffling phase as a procedure of quantization, STE is applied at line.10. Namely, we reserve the unconstrained weight values $\{\hat{\boldsymbol{w}}_l^k\}_{l=1}^L$ but update them with gradients w.r.t constrained weight values $\{\boldsymbol{w}_l^0\}_{l=1}^L$.

Algorithm 1 Index Optimization

Input: Training dataset $\mathcal{D}^{\text{train}}$, initial weights $\boldsymbol{w}^0 = \{\boldsymbol{w}_l^0\}_{l=1}^L$,
Output: Optimized weights $\boldsymbol{w}^K = \{\boldsymbol{w}_l^K\}_{l=1}^L$ after K iterations

1: // Initialization
2: Initialize unconstrained weights $\hat{\boldsymbol{w}}_l^0 \leftarrow \boldsymbol{w}_l^0$, for $l \in \{1...L\}$
3: Initialize sorted weights $\mathbf{w}_l^{\text{sorted}} \leftarrow \text{sort}(\boldsymbol{w}_l^0)$, for $l \in \{1...L\}$
4: // Optimization loop
5: **for** $k \leftarrow 1, 2, ..., K$ **do**
6: Sample mini-batch $\mathcal{D}^k \sim \mathcal{D}^{\text{train}}$;
7: Get the loss $\mathcal{L}(\boldsymbol{w}^{k-1}; \mathcal{D}^k)$
8: Compute the gradient $\frac{\partial \mathcal{L}}{\partial \boldsymbol{w}^{k-1}}$
9: Approximate the gradient $\frac{\partial \mathcal{L}}{\partial \hat{\boldsymbol{w}}^{k-1}} \doteq \frac{\partial \mathcal{L}}{\partial \boldsymbol{w}^{k-1}}$
10: Optimize $\hat{\boldsymbol{w}}$ with a gradient-based optimizer $\hat{\boldsymbol{w}}^k = \hat{\boldsymbol{w}}^{k-1} + \text{opt}(\frac{\partial \mathcal{L}}{\partial \hat{\boldsymbol{w}}^{k-1}})$
11: $\boldsymbol{w}_l^k \leftarrow \mathbf{w}_l^{\text{sorted}}[\text{argsort}(\hat{\boldsymbol{w}}_l^k)]$ for $l \in \{1...L\}$
12: **end for**

4.3 sortGAN

We propose a draft architecture based on cGAN, which is shown in Fig.4.2. As the name *sortGAN* suggests, it intends to sort the disordered pixels to recover a meaningful image given the class label. Compared to cGAN, we substitute the randomly shuffled pixels of the original image for the input noise. Furthermore, we also add an additional STE between the generator and the discriminator, whose function is to match the generated pixels to original given pixels. It sorts both the generated pixels and the original disordered pixels according to their magnitude. Then pixels with the same index in the two sorted arrays are matched, so that the original disordered pixels can be re-sorted in accordance with the order of generated pixels. During the forward pass, the re-sorted original pixels are fed further into the discriminator. During the back propagation the gradients w.r.t the re-sorted original pixels are used to update weights inside the

generator. The objective function is modified from (3.13) to the following:

$$\begin{aligned}\min_{\mathrm{G}}\max_{\mathrm{D}} \mathrm{V}(\mathrm{D},\mathrm{G}) =& \mathbb{E}_{\boldsymbol{x}\sim p_{data}(\boldsymbol{x})}[\log \mathrm{D}(\boldsymbol{x}|\boldsymbol{y})] \\ &+ \mathbb{E}_{\boldsymbol{z}\sim p_{\boldsymbol{z}}(\boldsymbol{z})}[\log(1-\mathrm{D}(\mathrm{G}(\boldsymbol{z}|\boldsymbol{y})))] \\ &+ \mathbb{E}_{\boldsymbol{z}\sim p_{\boldsymbol{z}}(\boldsymbol{z})}[\mathrm{G}(\boldsymbol{z}|\boldsymbol{y}) - p_{\boldsymbol{x}}(\boldsymbol{x})]^2.\end{aligned} \tag{4.3}$$

We add MSE (Eq.(3.3)) to reduce the difference between the generated image and the original one, because our goal is trying to recover the original image instead of generating new samples.

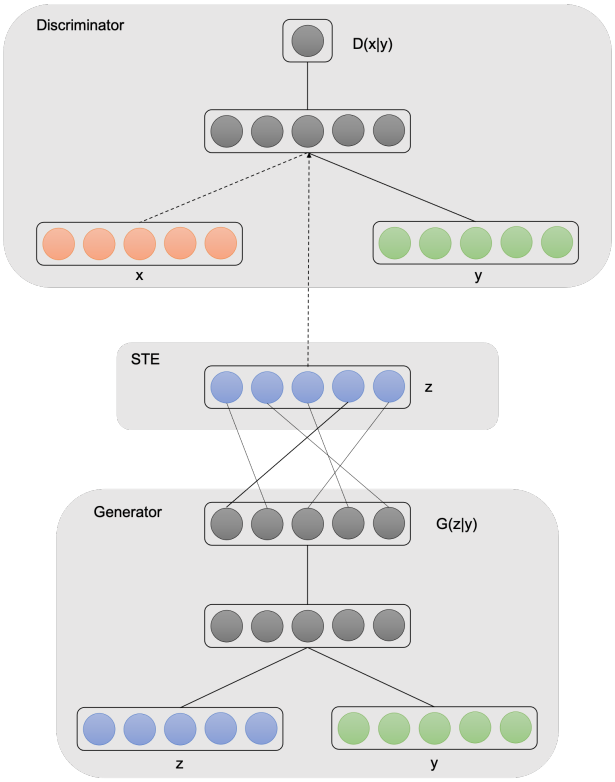


Figure 4.2: Architecture of sortGAN.

Chapter 5

Evaluations and Discussions

We have conducted a series of experiments to verify the feasibility of our proposed index optimizer and sortGAN. In this chapter, we report the detailed experimental setup and present the results.

5.1 Index Optimizer

Our proposed index optimizer achieves very competitive performance on MNIST, CIFAR10 and ImageNet and is feasible not only when initialized with normal distribution but also uniform distribution. It's known that sparse architectures are difficult to train from scratch[25, 26]. Nevertheless, our index optimizer can work properly and gain even better test accuracy than dense architecture if initialized sparsely, i.e. pruning 90% weights according to their magnitude before training.

We will show the experiment results together with how we design and conduct them in the next.

5.1.1 Ablation Studies

To understand the feasibility and stability of index optimizer, we perform ablation studies in respect of different granularity of permutation and diverse initialization distributions. Before diving deep into the experimental setup and results, we would like to introduce the notations and meaning behind them at first.

Granularity

In Algorithm.1, the default granularity of permutation is inside one layer (pseudocode line.11), i.e. the permutations take place among weights from same layer. In order to reduce the overhead of sorting, we also check the cases where permutations only happen in the 1st or 2nd dimension of weights.

According to the notations introduced in background chapter. Weights of `conv` layer l should be $\boldsymbol{w}_l \in \mathbb{R}^{C_l \times C_{l-1} \times S \times S}$. The default granularity means one parameter can be substituted with any other parameters inside $\boldsymbol{w}_l$, i.e. there are $C_l \times C_{l-1} \times S \times S$ choices. If the permutations happen inside 1st dimension, it means two parameters in exchange with each other must have the same first-dimension index. In other words, they are parameters from the same output channel and the number of potential choices to replace one parameter is $C_{l-1} \times S \times S$ now. Alike, if the scope of permutation is further shrunk within 2nd dimension, it means reshuffling takes places inside one kernel and the number of potential choices reduced to $S \times S$.

Initialization

We consider three commonly used initialization distributions in out experiments, i.e. Gaussian, uniform and sparse. We use Kaiming initialization and the explicit formula for Gaussian and uniform distribution can be found in Eq.(3.5) and Eq.(3.6). For simplicity, Eq.(3.5) and Eq.(3.6) are abbreviated to $\mathcal{N}_k$ and $\mathcal{U}_k$ afterwards. Sparse initialization means all parameters with values less than a specific threshold will be set to zero before the start of training. We realize sparse initialization by firstly initializing weights with $\mathcal{N}_k$ and then pruning 90% parameters according to their magnitudes.

5.1.2 Experimental Setup

We train LeNet5[28], VGG[6] and ResNet34[5] on MNIST, CIFAR10 and ImageNet with our index optimizer respectively. The mini-batch size is chosen to be 128 for MNIST and CIFAR10, while 1024 for ImageNet. To train LeNet5, we use Adam as inner gradient-based optimizer while for VGG and ResNet34 we use SGD. The initial learning rate of inner gradient-based optimizer is set to be $1e^{-3}$ and exponentially decayed with decay factor of 1.01. Momentum is also used and set to be 0.9 when using SGD. Besides, to avoid overfitting we stop the training if the test accuracy keeps decreasing for over 20 epochs.

We test the feasibility of index optimizer under Kaiming normal, Kaiming uniform and sparse initializations. Additionally, we also test different granularity to permute the weights. More details about ablation studies are provided in the former section.

5.1.3 Experiment Results

We summarize the top-1 test accuracy from experiments in the following tables. In Tab.5.1, we show the default case, i.e. permuting weights from same layer. $\mathcal{N}_k$ and $\mathcal{U}_k$ indicate the weights inside the architecture are initialized with Kaiming

normal or Kaiming uniform distribution respectively, whereas *sparse* means 90% of weights are pruned before training. The results of LaPerm are directly taken from [30]. In Tab.5.2 and Tab.5.3 results of finer permutation granularity are presented.

	LeNet5 & MNIST			VGG & CIFAR10			ResNet34 & ImageNet		
	$\mathcal{N}_k$	$\mathcal{U}_k$	sparse	$\mathcal{N}_k$	$\mathcal{U}_k$	sparse	$\mathcal{N}_k$	$\mathcal{U}_k$	sparse
LaPerm	99.72	99.76	77.33	⁄	88.85	77.98	⁄	⁄	⁄
Index Optimizer (without STE)	99.49	99.16	96.56	92.44	93.85	90.92	70.93	⁄	⁄
Index Optimizer	99.49	99.34	99.18	90.94	93.01	92.55	⁄	⁄	⁄

Table 5.1: Results of index optimizer. Permutations are allowed inside one layer.

	LeNet5 & MNIST			VGG & CIFAR10		
	$\mathcal{N}_k$	$\mathcal{U}_k$	sparse	$\mathcal{N}_k$	$\mathcal{U}_k$	sparse
Index Optimizer (without STE)	98.77	98.96	92.91	87.77	91.40	87.06
Index Optimizer	99.51	99.26	99.41	90.98	92.53	91.62

Table 5.2: Results of index optimizer. Permutations are allowed inside one output channel.

	LeNet5 & MNIST			VGG & CIFAR10		
	$\mathcal{N}_k$	$\mathcal{U}_k$	sparse	$\mathcal{N}_k$	$\mathcal{U}_k$	sparse
Index Optimizer (without STE)	98.63	95.16	69.65	83.67	90.72	71.33
Index Optimizer	99.48	99.27	99.36	90.24	91.86	90.42

Table 5.3: Results of index optimizer. Permutations are allowed inside one kernel.

As can be seen from Tab.5.1, our proposed index optimizer achieves competitive accuracy and is feasible with both normal distribution and uniform distribution. Besides, it's also possible to train sparse architecture from the start with index optimizer without harming the accuracy. Since sparse architectures are difficult to train with conventional gradient-based optimization algorithms, our work offers an alternative approach to tackle this issue. We list also results of index optimizer without STE, which corresponds to LaPerm with sync period $k = 1$. The results contradict the statement in paper [30] that LaPerm shows significant degradation in its performance when $k = 1$. Moreover, the performance remains fairly stable under finer permutation granularity. Although, there exists a tiny drop for sparse architectures. For dense architecture, it has almost no impact.

5.2 sortGAN

Input permutations have great practical value in relation to archaeology, chemistry and biology, for instance to restore cultural heritage, or to analyze genome. Our goal here is trying to recover an image from its randomly reshuffled pixels. It can be seen as a special sorting task and we find it's challenging to guide the sorting by an individual classification neural network (see Appendix.B for more details). Based on GAN, we propose a prototype architecture called *sortGAN* to attain the goal.

5.2.1 Experimental Setup

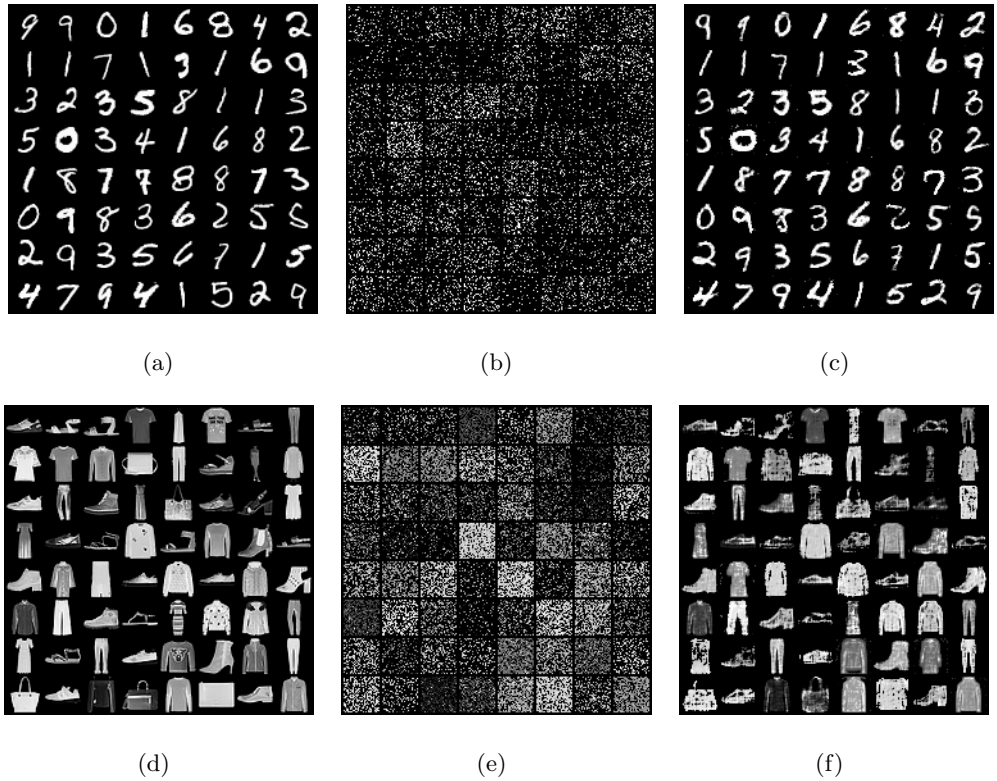
sortGAN (see Fig.4.2) is constructed of a generator net, a discriminator net and STE in between. Both the generator net and the discriminator net are implemented with multilayer perceptron (MLP). The generator consists of five fully-connected layers with hidden layer sizes 128, 256, 512 and 1024 respectively and outputs a 784-dimensional vector corresponding to the pixels of the generated image. Likewise, the discriminator is composed of three fully-connected layers with same hidden layer size 512 and one additional to output the logit.

During the experiment, we shuffle the pixels from the original image and arrange them into one-dimensional features, which are subsequently joint with the class label encoded as one-hot vector. The generator is fed with this joint one-dimensional features and outputs generated pixels that will guide the re-sorting procedure of original disordered pixels. BCE (Eq.(3.2)) is used for training the discriminator, whereas for training the generator both BCE and MSE (Eq.(3.3)) are used together. The whole architecture is trained by Adam with mini-batch size 64 and learning rate equals $2e^{-3}$.

5.2.2 Experiment Results

We conduct experiments on two datasets, namely MNIST and FashionMNIST, and then arbitrarily choose one mini-batch to show the results after 100 epochs (see Fig.5.1). The three columns present original images, pixels after random shuffling and recovered images separately.

From the results we can see that sortGAN can recover the features of original images to some extent. For instance, the two 9 image samples shown in the first line of Fig.5.1(c) are different even though their class labels are the same. The circle sizes of the two 9 on the recovered images highly coincide to that of original images. However, limitations of our preliminary model are revealed in Fig.5.1(f). Details like the patterns on clothes can not be fully recovered. Besides, there are some scattered noise pixels.



(a) (b) (c)

(d) (e) (f)

Figure 5.1: Results of sortGAN. 5.1(a) and 5.1(d) show original images from datasets. 5.1(b) and 5.1(e) show randomly disordered pixels. 5.1(c) and 5.1(f) show recovered results.

Chapter 6

Conclusions and Future Work

In conclusion, starting from the observation that training neural networks only affects the variance of the weight distribution, we propose *index optimizer* that trains a neural network by permuting randomly-initialized weights without altering their numerical values and verify its efficiency under different conditions. Additionally, we study the permutations of input and propose a preliminary architecture called *sortGAN* to recover original image from randomly shuffled pixels. The contributions of this work can be summarized as:

- We propose index optimizer that can be potentially used for lightweight storage and transmission of deep neural networks. In comparison with LaPerm, our index optimizer is more stable and attains competitive performance. In addition, it also solves hyper-parameter choosing problem of LaPerm.
- Our proposed index optimizer is also feasible when the initial weights are sparse (90% parameters are zeros). It provides an approach to meet the challenge of training sparse architectures from scratch.
- We propose a preliminary pipeline called sortGAN and it can to some extent recover original image from randomly shuffled pixels.

Despite the above contributions, there remain some open questions waiting to be answered and limitations needed to be eliminated in the future work. Our experiments in section 5.1.3 are restricted on classification problems. Meanwhile, although efficiency of index optimizer is empirically verified, it lacks strong theoretical explanations. Due to time limitation, experiments for pruning and finer-granularity are not executed on ImageNet. Moreover, the recovered result of sortGAN is still unsatisfactory when the target image is relatively complex and it demands additional mechanism to tackle RGB images.

Appendix A

Preliminary Experiments

In this chapter, we show supplementary results from preliminary experiments. In chapter 1, we demonstrate with empirical evidence that if initialized with Gaussian distribution, weights from the entire architecture follow very similar distributions before and after training. However, in Algorithm.1 we permute weights inside one layer instead of entire architecture. To provide a more sufficient explanation, we present the distribution of parameters inside each layer. Fig.A.1 and Fig.A.3 show the initial layer weight distributions of LeNet5 and VGG, while Fig.A.2 and Fig.A.4 show the layer weight distributions after training. We can draw similar conclusion from the figures that training only affects the variance of the weight distributions inside each layer.

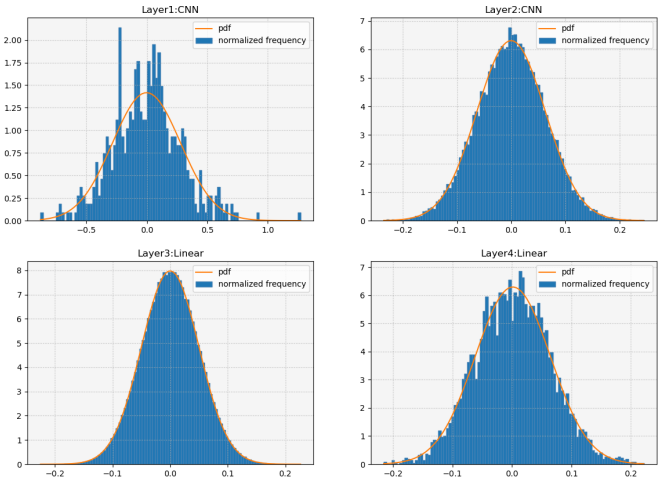


Figure A.1: Initial layer weight distributions of LeNet5.

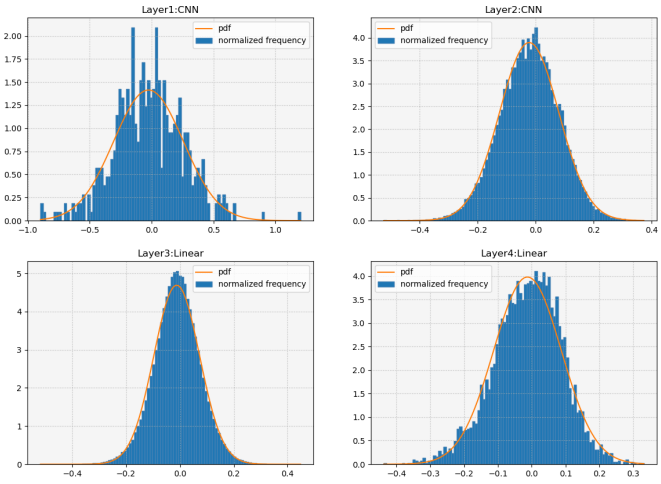


Figure A.2: Layer weight distributions of LeNet5 after training.

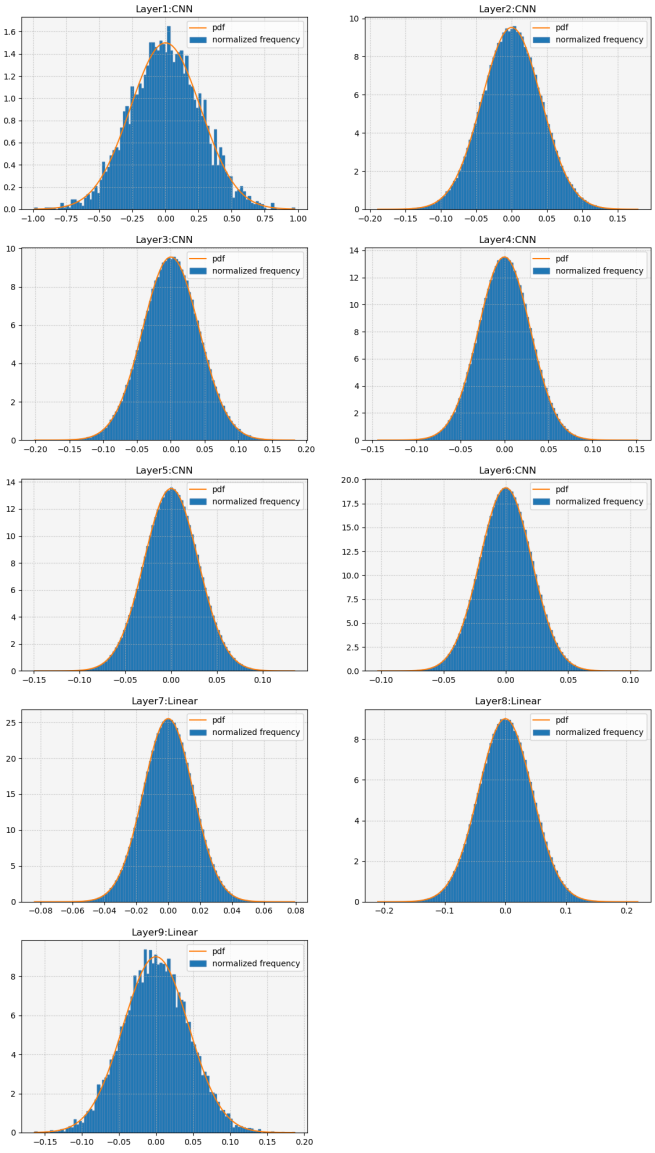


Figure A.3: Initial layer weight distributions of VGG.

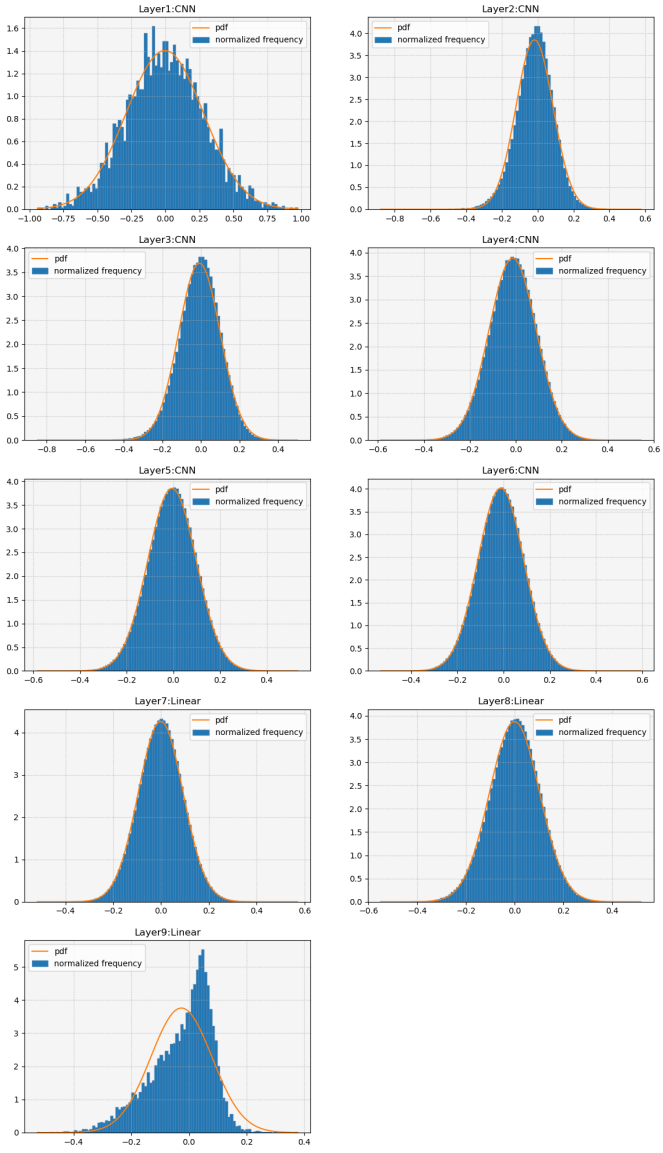


Figure A.4: Layer weight distributions of VGG after training.

Appendix B

Input Permutation

In this chapter, we share the results of experiments where we guide the permutation of input pixels by using trained neural networks reversely. Our goal here is the same as in section 5.2, i.e. given the class label and randomly disordered pixels, we would like to recover the original image.

We train LeNet5 on MNIST and then freeze the parameters inside the model. By treating input pixels as parameters, we use our proposed index optimizer to update the positions of them. We also add some tricks to boost the performance:

Bagging We train three models on MNIST separately. The second and third models are trained by freezing all parameters of the first model except the ones from the input layer, i.e. the differences of parameters among the three trained models lie on the input layer. We then fix the parameters inside these three trained models and use them together to update the positions of input pixels.

Additional Loss Functions Apart from the original cross-entropy loss function targeted at classification, we add two additional loss functions to aid the re-sorting of disordered pixels. One of them penalizes non-black pixels on the boundary. The other aims at gathering pixels of same gray scale together, i.e. it penalizes abrupt intensity change between one pixel and its neighbours.

Prototype of Discriminator We train a model to distinguish original images from the randomly disordered ones. After we fix the parameters inside the model, it is used to optimize the input pixels so that the re-sorted pixels can form a reasonable image that can be classified as original image.

We pick the first sample from each class and present the recovered results in Fig.B.1. Despite various tricks we used, the recovered results are still inadequate due to the complexity of the task.

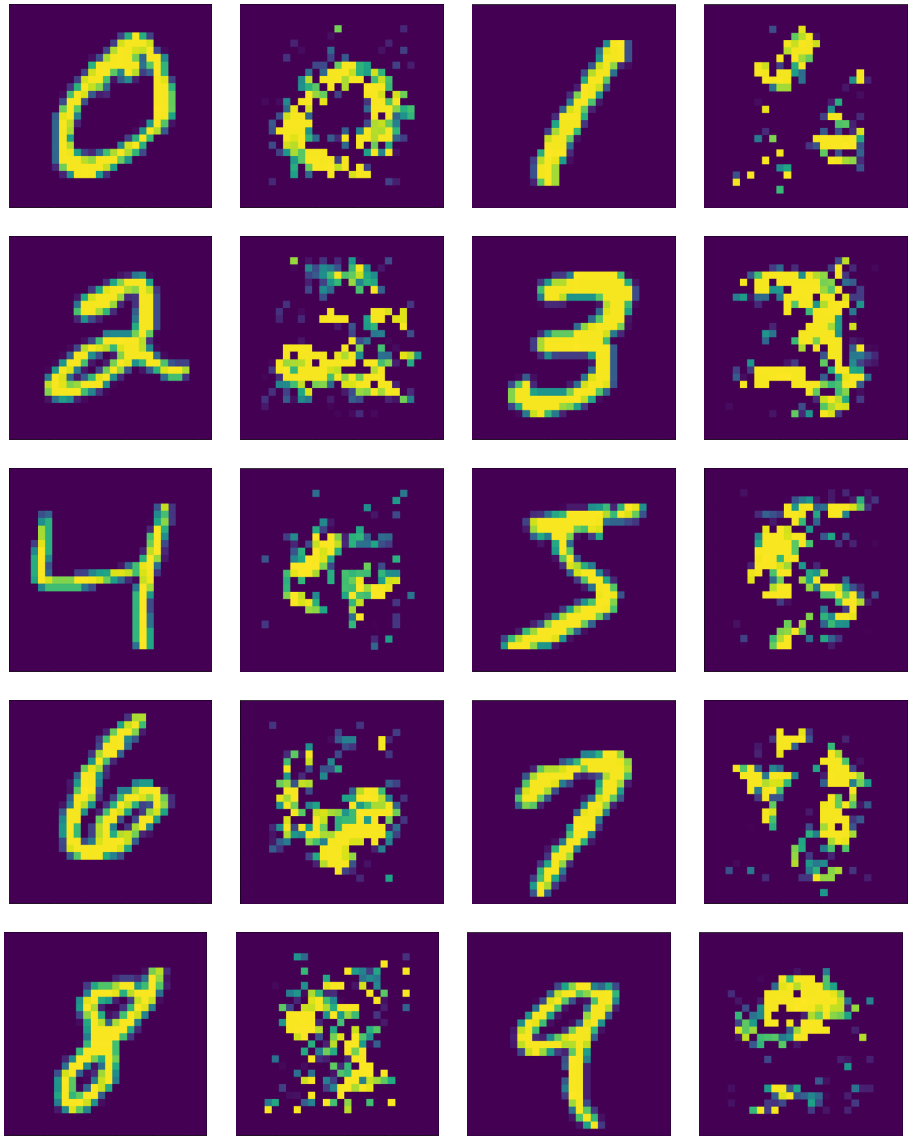


Figure B.1: Experiment results. The first and third columns show the original image samples from MNIST. The second and the last columns show the recovered results of the corresponding original image sample.

Bibliography

[1] Yoshua Bengio, Nicholas Léonard, and Aaron Courville. Estimating or propagating gradients through stochastic neurons for conditional computation, 2013.

[2] Yann LeCun and Corinna Cortes. MNIST handwritten digit database. 2010.

[3] Alex Krizhevsky. Learning multiple layers of features from tiny images. Technical report, 2009.

[4] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei. ImageNet: A Large-Scale Hierarchical Image Database. In *CVPR09*, 2009.

[5] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition, 2015.

[6] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition, 2015.

[7] Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. Language models are unsupervised multitask learners. 2018.

[8] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding, 2019.

[9] Misha Denil, Babak Shakibi, Laurent Dinh, Marc'Aurelio Ranzato, and Nando de Freitas. Predicting parameters in deep learning, 2014.

[10] Yiwen Guo, Anbang Yao, and Yurong Chen. Dynamic network surgery for efficient dnns, 2016.

[11] Song Han, Jeff Pool, John Tran, and William J. Dally. Learning both weights and connections for efficient neural networks, 2015.

[12] Xin Dong, Shangyu Chen, and Sinno Jialin Pan. Learning to prune deep neural networks via layer-wise optimal brain surgeon, 2017.

[13] Babak Hassibi and David Stork. Second order derivatives for network pruning: Optimal brain surgeon. In *Advances in Neural Information Processing Systems*, volume 5, 1993.

[14] Yann LeCun, John Denker, and Sara Solla. Optimal brain damage. In *Advances in Neural Information Processing Systems*, volume 2, 1990.

[15] Itay Hubara, Matthieu Courbariaux, Daniel Soudry, Ran El-Yaniv, and Yoshua Bengio. Binarized neural networks. In *Advances in Neural Information Processing Systems*, volume 29, 2016.

[16] Brais Martinez, Jing Yang, Adrian Bulat, and Georgios Tzimiropoulos. Training binary neural networks with real-to-binary convolutions, 2020.

[17] Ziwei Wang, Jiwen Lu, Chenxin Tao, Jie Zhou, and Qi Tian. Learning channel-wise interactions for binary convolutional neural networks. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2019.

[18] Chenzhuo Zhu, Song Han, Huizi Mao, and William J. Dally. Trained ternary quantization, 2017.

[19] Yunchao Gong, Liu Liu, Ming Yang, and Lubomir Bourdev. Compressing deep convolutional networks using vector quantization, 2014.

[20] Jiaxiang Wu, Cong Leng, Yuhang Wang, Qinghao Hu, and Jian Cheng. Quantized convolutional neural networks for mobile devices, 2016.

[21] Emily Denton, Wojciech Zaremba, Joan Bruna, Yann LeCun, and Rob Fergus. Exploiting linear structure within convolutional networks for efficient evaluation, 2014.

[22] Max Jaderberg, Andrea Vedaldi, and Andrew Zisserman. Speeding up convolutional neural networks with low rank expansions, 2014.

[23] Vadim Lebedev, Yaroslav Ganin, Maksim Rakhuba, Ivan Oseledets, and Victor Lempitsky. Speeding-up convolutional neural networks using finetuned cp-decomposition, 2015.

[24] Alexander Novikov, Dmitry Podoprikhin, Anton Osokin, and Dmitry Vetrov. Tensorizing neural networks, 2015.

[25] Jonathan Frankle and Michael Carbin. The lottery ticket hypothesis: Finding sparse, trainable neural networks, 2019.

[26] Hattie Zhou, Janice Lan, Rosanne Liu, and Jason Yosinski. Deconstructing lottery tickets: Zeros, signs, and the supermask, 2020.

[27] Vivek Ramanujan, Mitchell Wortsman, Aniruddha Kembhavi, Ali Farhadi, and Mohammad Rastegari. What's hidden in a randomly weighted neural network?, 2020.

[28] Yann Lecun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-based learning applied to document recognition. In *Proceedings of the IEEE*, pages 2278–2324, 1998.

[29] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Delving deep into rectifiers: Surpassing human-level performance on imagenet classification, 2015.

[30] Yushi Qiu and Reiji Suda. Train-by-reconnect: Decoupling locations of weights from their values, 2020.

[31] Ian J. Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. Generative adversarial networks, 2014.

[32] Mehdi Mirza and Simon Osindero. Conditional generative adversarial nets, 2014.

[33] Alec Radford, Luke Metz, and Soumith Chintala. Unsupervised representation learning with deep convolutional generative adversarial networks, 2016.

[34] Han Xiao, Kashif Rasul, and Roland Vollgraf. Fashion-mnist: a novel image dataset for benchmarking machine learning algorithms, 2017.

[35] Carl Doersch, Abhinav Gupta, and Alexei A. Efros. Unsupervised visual representation learning by context prediction, 2016.

[36] Mehdi Noroozi and Paolo Favaro. Unsupervised learning of visual representations by solving jigsaw puzzles, 2017.

[37] Chen Wei, Lingxi Xie, Xutong Ren, Yingda Xia, Chi Su, Jiaying Liu, Qi Tian, and Alan L. Yuille. Iterative reorganization with weak spatial constraints: Solving arbitrary jigsaw puzzles for unsupervised representation learning, 2018.

[38] Marie-Morgane Paumard, David Picard, and Hedi Tabia. Deepzzle: Solving visual jigsaw puzzles with deep learning and shortest path optimization. *IEEE Transactions on Image Processing*, 29:3569–3581, 2020.

[39] Xavier Glorot and Yoshua Bengio. Understanding the difficulty of training deep feedforward neural networks. In *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics*, pages 249–256, 2010.

[40] J. Kiefer and J. Wolfowitz. Stochastic Estimation of the Maximum of a Regression Function. *The Annals of Mathematical Statistics*, 23(3):462 – 466, 1952.

[41] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization, 2017.