4-27-2020

Deep Convolutional Networks without Learning the Classifier Layer

Zhongchao Qian
zq3684@rit.edu

Deep Convolutional Networks without Learning the Classifier Layer

by

Zhongchao Qian

B.Eng. Tianjin University, 2017

A thesis submitted in partial fulfillment of the
requirements for the degree of Master of Science
in the Chester F. Carlson Center for Imaging Science
College of Science
Rochester Institute of Technology

April 27, 2020

Signature of the Author ______________________________

Accepted by ______________________________
Coordinator, M.S. Degree Program Date

CHESTER F. CARLSON CENTER FOR IMAGING SCIENCE
COLLEGE OF SCIENCE
ROCHESTER INSTITUTE OF TECHNOLOGY
ROCHESTER, NEW YORK

CERTIFICATE OF APPROVAL

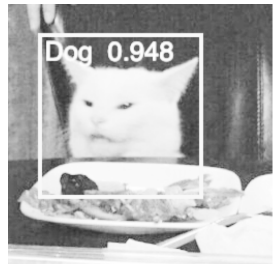
M.S. DEGREE THESIS

The M.S. Degree Thesis of Zhongchao Qian
has been examined and approved by the
thesis committee as satisfactory for the
thesis required for the
M.S. degree in Imaging Science

Dr. Christopher Kanan, Thesis Advisor   Date

Dr. Guoyu Lu   Date

Dr. Nathan Cahill   Date



To everyone trying to figure out why everything is or isn't working.

Deep Convolutional Networks without Learning the Classifier Layer

by

Zhongchao Qian

Submitted to the
Chester F. Carlson Center for Imaging Science
in partial fulfillment of the requirements
for the Master of Science Degree
at the Rochester Institute of Technology

Abstract

Deep convolutional neural networks (CNNs) are effective and popularly used in a wide variety of computer vision tasks, especially in image classification. Conventionally, they consist of a series of convolutional and pooling layers followed by one or more fully connected (FC) layers to produce the final output in image classification tasks. This design descends from traditional image classification machine learning models which use engineered feature extractors followed by a classifier, before the widespread application of deep CNNs. While this has been successful, in models trained for classifying datasets with a large number of categories, the fully connected layers often account for a large percentage of the network's parameters. For applications with memory constraints, such as mobile devices and embedded platforms, this is not ideal. Recently, a family of architectures that involve replacing the learned fully connected output layer with a fixed layer has been proposed as a way to achieve better efficiency. This research examines this idea, extends it further and demonstrates that fixed classifiers offer no additional benefit compared to simply removing the output layer along with its parameters. It also reveals that the typical approach of having a fully connected final output layer is inefficient in terms of parameter count. This work shows that it is possible to remove the entire fully connected layers thus reducing the model size up to 75% in some scenarios, while only making a small sacrifice in terms of model classification accuracy. In most cases, this method can achieve comparable performance to a traditionally learned fully connected classification output layer on the ImageNet-1K, CIFAR-100, Stanford Cars-196, and Oxford Flowers-102 datasets, while not having a fully connected output layer at all. In addition to comparable performance, the method featured in this research also provides feature visualization of deep CNNs at no additional cost.

Acknowledgements

I am very thankful to my advisor Dr. Christopher Kanan, for providing a lot of help, support, and guidance, in research work and life. Research is full of challenges and setbacks, and Dr. Kanan helped me overcome a lot of the difficulties. Also my thesis committee who provided suggestions and feedback. My time working at kLab has been pleasant and I would like to thank everyone in the lab. I also appreciate the help from all faculties and staff in the Center for Imaging Science. My friends and family provided a lot of emotional support and help. Finally DARPA who provided funding for the lifelong machine learning project which this work is part of.

Contents

List of Figures

List of Tables

Chapter 1

Introduction and Motivation

The strong performance of deep convolutional neural networks (CNNs) has enabled an enormous number of new computer vision applications. However, many state-of-the-art CNN architectures are ill-suited for deployment on mobile and embedded devices due to their high computational and memory requirements. The vast majority of CNN architectures are designed as having a feature extractor followed by a classifier. The feature extractor consists of convolutional layers and pooling operations, while the classifier is made up of one or more fully connected layers. This has been a common practice since the early days of deep CNNs, and it descends from traditional image classification methods. In the years before the first deep CNN won the ImageNet Large-Scale Visual Recognition Challenge (ILSVRC) challenge, winning methods as documented in [23, 30] used crafted feature extractors, followed by classifiers based on support-vector machines (SVM). In ILSVRC2012, the winning method AlexNet proposed in [18] is a deep convolutional neural network, which has a feature extractor, followed by a classifier using three fully connected layers with ReLU activation in between. In the years followed, popular architectures, for example the VGG family proposed in [34], all used multiple fully connected layers. The first work to change that is Network in Network proposed in [22], it uses a global average pooling (GAP) layer after the feature extractors, and only has a single fully connected layer for the classifier. In [37], GoogLeNet (Inception v1) uses this method and won ILSVRC2014. Since then, using global average pooling followed by a single fully connected layer has been the popular method of implementing the classifier.

A number of papers have developed methods for reducing the parameters in the feature extractor, for instance, in [18], AlexNet first implemented group convolutions, in depthwise separable convolutions introduced in Xception from [4], and squeeze and expand operations from SqueezeNet presented in [13], but little work has been done to reduce the parameters in the classifier's fully connected layers. Because the number of parameters in the classifier is typically proportional to the number of categories, the classifier can consume a large portion of the network's total parameters for large datasets. For example, in MobileNet v2 from [32], the fully connected layers consume 37% of the parameters in the CNN for ImageNet-1K classification.

A few existing works have studied how to reduce the number of parameters in a CNN's classifier for many-class datasets by using fixed output matrices [10, 29]. These methods initialize the weights, but do not update them during training, thus increasing the efficiency of models.

In this research, this idea is taken further. A fixed identity matrix is used as the classifier, which is equivalent to removing the classifier layer rather than having a feature extractor followed by a classifier. The convolutional layers are trained directly for classification and the traditional classification layer is entirely eliminated. This research shows that the number of parameters can be greatly reduced by rethinking the architecture design, as demonstrated in the bar plots for different architectures for ImageNet-1K classification in Figure 1.1. The green plot shows the total number of parameters for each architecture. As models get more efficient and compact, the final classifier accounts for more of the total parameters. The method presented in this work eliminates the need for a final fully connected (FC) layer for classification, significantly reducing memory requirements, especially in already efficient models.

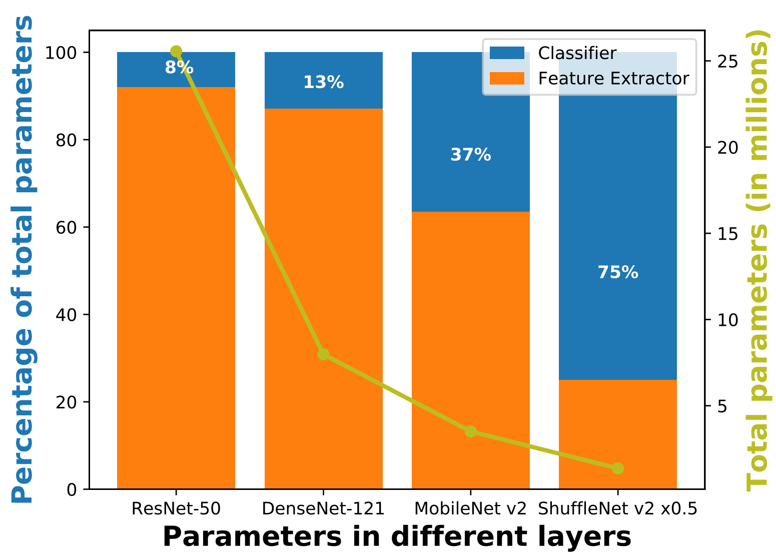


Figure 1.1: Bar plot showing the percentage of parameters in different parts of various deep CNN architectures.

This research features the following contributions: 1) It shows that the final convolutional layer can be modified in many widely used CNN architectures to enable the fully connected layer to be completely eliminated, with little loss in classification performance but with a large reduction in the total number of parameters for many-class datasets. 2) It compares the method against existing fixed classifier methods and achieve superior results, while being much simpler and more efficient. 3) It shows that the final classifier layer contributes little to overall model classification accuracy. Thus suggesting that using a fully connected layer is very inefficient and should be changed in future architecture designs for image classification. 4) It demonstrates that the method's final convolutional layers are interpretable without needing any additional computation or post-processing, which can be prohibitive on edge devices. This enables the CNN to be used for detection and localization without explicitly using techniques such as Class Activation Mapping (CAM), which was demonstrated in [41].

Chapter 2

Background work

This work relates to three main categories of existing work: **1) Alternative classifiers** which have been explored mainly for making the output layer more discriminative, or attempting to make the classifier more efficient, **2) Parameter reduction techniques** which range from the ground-up redesign of networks to post-trained pruning techniques, and **3) Model visualization** which are other techniques to provide visual interpretations to deep CNN models. These background work are discussed in detail in the following sections.

2.1 Alternative Classifiers

In [35], a study was conducted to understand what components of a CNN are absolutely necessary. They concluded that a CNN can be constructed using only convolution operations by demonstrating that the final fully connected output layer could be replaced by 1-by-1 point-wise convolutions; however, they did not consider that the entire classification layer could be removed.

A few existing works have studied how to reduce the number of parameters in a CNN's classifier for many-class datasets by using fixed output matrices [10, 29]. In [10], it was shown that any fixed orthogonal output matrix could be used to replace a learned output matrix with no reduction in performance. While this does not reduce the number of parameters or computational requirements, they then demonstrated that a Hadamard matrix could be used. A Hadamard matrix can be deterministically generated and does not need to be stored, thus enabling increased efficiency. However, it is not possible to construct a Hadamard matrix if the input to the classifier has fewer dimensions than the number of output categories because a Hadamard matrix's rows and columns are mutually orthogonal. This means for ResNet-18, which has 512-dimensional features input to the classifier, it would be limited to classifying at most 512 categories. This limitation was overcome in [29], which proposed a different method of creating a fixed output classifier. Their approach uses coordinate values of high-dimensional regular polytopes as rows of the fixed classifier weight matrix. While this approach works, it can be difficult to train, and it is used mainly to optimize for feature extraction.

It is not currently clear which fixed output matrix approach is best, and some of these methods still require the classifier's parameters to be stored, even if the parameters are not updated during training. In contrast, the approach in this research avoids using an explicit classification layer entirely, eliminating the problem of selecting and storing a fixed classifier weight matrix.

2.2 Parameter Reduction Techniques

A class of popular methods for reducing the number of parameters in the feature extractor is by using variants of convolution operations. Popular techniques include group convolutions, depth-wise convolutions, bottleneck modules, *etc.* Group convolutions split the convolution input and output channels into groups, where each group is a convolution operation independent of other groups [18]. By removing connections between channels belonging in different groups, it reduces parameters in the convolution by a factor equal to the number of channels. Depth-wise separable convolution is a two-step procedure. First, there is a group convolution where the number of input channels, output channels, and groups are all the same, followed by a point-wise convolution with the desired number of output channels [4]. In [9], bottleneck modules which has three layers of 1x1, 3x3, and 1x1 convolutions, using the point-wise convolutions to decrease and then increase the dimensions, reducing the parameters in the 3x3 convolution. Similar techniques are used in [13], the Fire module uses point-wise convolution to compress the number of channels first, then uses both 3x3 and 1x1 convolutions to expand to the desired number of channels.

Other methods for reducing the number of parameters are pruning and quantization. Pruning removes (zeros out) weights after training to promote sparsity, and a wide variety of pruning methods have been explored [1,7,11,20,21,24,36]. Quantization methods typically reduce the numeric precision of the weights after training, which can greatly reduce the number of parameters [5,14,15]. Both pruning and quantization are complementary to the method proposed in this research, which focuses on eliminating the classifier to reduce the number of parameters.

2.3 Model Visualization

One of the major complaints about CNNs is that they lack interpretability, leading to tools such as CAM [41], Grad-CAM [33], and Grad-CAM++ [3] being developed to better understand the features that led to the output of the classifier. These methods require additional post-processing computation after the model has been run to visualize the evidence used by the classifier to generate its output. In contrast, the approach in this research enables the CNN to be interpreted immediately, without any extra compute required.

Visualization of CNN model is already present when Le Cun developed LeNet-5 for handwritten digit recognition [19], showing the activations in each layer of the network. In [39], Zeiler introduced a method that visualizes intermediate feature layers in deep convolutional neural networks, giving some insight to the inner workings of deep convolutional neural networks.

Inverting the network is another technique that also provides insight to the network itself, and reveals that deep features contain information to reconstruct the input image [6, 26].

Zhou *et al.* discovered that a deep CNN for image classification can also be used for object detection [40], in the same forward pass calculation. Later they proposed class activation map [41] (CAM), the technique was introduced as a way to visualize which portion of the image a CNN used to make a prediction. It requires using global average pooling in models, and needs extra calculations to produce the CAM. The method in this work can output CAM, during the inference stage in a single forward pass as well, but can do it directly without any other additional calculations.

Chapter 3

Methods

In this research, four methods for implementing the classifier are evaluated: 1. using a learned fully connected classifier 2. using a fixed orthogonal projection; 3. using a fixed Hadamard projection; and 4. removing the fully connected layer, which is equivalent to using a fixed identity matrix for projection and setting the bias term to zero. First, the conventional method of using a fully connected classifier will be explained. Then the two fixed projection methods [10] will also be explained. Finally the classifier implementation featured in this work will be demonstrated. All three fixed classifiers will be compared against a learned fully connected classifier, and against each other, to evaluate their effects on the model.

3.1 Learned Fully Connected Classifier

In typical deep neural networks for single-class image classification, the last layer is a fully connected layer of affine transformation, and all its parameters are learned.

First, a few variables will be defined:

- Let $f(\cdot)$ be the feature extractor.
- Let $c(\cdot)$ be the classifier.
- Let $\mathbf{x} \in \mathbb{R}^{3\times h\times w}$ be the input to the model, assuming the input is an 3 color channel RGB image and h, w is its height and width.
- Let n_c be the number of output channels from the feature extractor.
- Let f_h and f_w be the height and width of the output from the feature extractor.
- Let $\mathbf{f} \in \mathbb{R}^{n_c\times f_h\times f_w}$ be the output feature map, $\mathbf{f} = f(\mathbf{x})$.
- Let n_k be the number of output categories.
- Let $\mathbf{h}$ and $\mathbf{h}_i$ be intermediate results between layers in the classifier $c(\cdot)$.
- Let $\mathbf{y} \in \mathbb{R}^{n_k}$ be the output of the model, $\mathbf{y} = c(\mathbf{h})$.

In earlier deep convolutional neural networks, for convenience we will use the AlexNet architecture proposed in [18] as an example, $\mathbf{f} \in \mathbb{R}^{256\times 6\times 6}$ is the result of a non-global max pooling operation of kernel size 3×3 in the end of its feature extractor $f(\cdot)$. This feature map $\mathbf{f}$ is then flattened into a vector $\mathbf{h_0} \in \mathbb{R}^{9,216}$. Then it goes through multiple affine transformations followed by non-linear activations, to finally produce the output $\mathbf{y} \in \mathbb{R}^{1000}$ as shown in Equation 3.1 below

$$\begin{aligned}
\mathbf{h_1} &= \mathrm{ReLU}\left(\mathbf{W_1 h_0} + \mathbf{b_1}\right) \\
\mathbf{h_2} &= \mathrm{ReLU}\left(\mathbf{W_2 h_1} + \mathbf{b_2}\right) \\
\mathbf{y} &= \mathrm{softmax}\left(\mathbf{W_3 h_2} + \mathbf{b_3}\right).
\end{aligned} \tag{3.1}$$

In the case of AlexNet, the weights for the affine transformations are $\mathbf{W_1} \in \mathbb{R}^{4{,}096\times 9{,}216}$, $\mathbf{W_2} \in \mathbb{R}^{4{,}096\times 4{,}096}$, $\mathbf{W_3} \in \mathbb{R}^{1{,}000\times 4{,}096}$; the biases $\mathbf{b}_{1\ldots 3}$ are of dimensions 4096, 4096, and 1024 respectively. The final non-linear activation is softmax$(\cdot)$, in order to produce the final output, which is the classification likelihood for each potential category.

In more recent architectures, the classifier is a single affine transformation, and its input is produced from a global average pooling (GAP) operation:

$$\mathbf{h} = \frac{1}{f_h \times f_w} \sum_{f_h} \sum_{f_w} \mathbf{f}. \tag{3.2}$$

By averaging the elements in each channel, we are able to obtain $\mathbf{h} \in \mathbb{R}^{n_c}$ as the input to the affine transformation and obtain the output:

$$\mathbf{y} = \text{softmax}\left(\mathbf{Wh} + \mathbf{b}\right). \tag{3.3}$$

It is intuitive that $\mathbf{W} \in \mathbb{R}^{n_k \times n_c}$ and $\mathbf{b} \in \mathbb{R}^{n_k}$. Through the use of GAP, the classifier is still able to use information from the entire feature map, while consuming way less parameters.

In either case, the weight matrices $\mathbf{W}$ and biases $\mathbf{b}$ are optimized during back-propagation using gradient descent.

3.2 Fixed Orthogonal Classifier

In a fixed orthogonal classifier [10], everything is the same as using a learned fully connected classifier, except for the weight matrix $\mathbf{W}$, which is initialized using a specific matrix, and during back-propagation, it is not updated.

To obtain the weight matrix $\mathbf{W}$, a semi-orthogonal matrix is randomly generated. An orthogonal matrix is defined as a square matrix $\mathbf{Q}$, where $\mathbf{Q}\mathbf{Q}^T = \mathbf{Q}^T\mathbf{Q} = \mathbf{I}$, and $\mathbf{I}$ is an identity matrix. In the case of a semi-orthogonal matrix, the matrix is no longer square. A matrix $\mathbf{W}$ is semi-orthogonal if either $\mathbf{W}^T\mathbf{W} = \mathbf{I}$ or $\mathbf{W}\mathbf{W}^T = \mathbf{I}$.

Given the semi-orthogonality, in the case of $n_c \geq n_k$, the rows of the weight matrix $\mathbf{W}$ are mutually orthogonal; in the case of $n_c < n_k$ the columns are mutually orthogonal, but the rows are not.

In fixed orthogonal classifiers, the weight matrix is not updated during training and is semi-orthogonal, hence its name.

3.3 Fixed Hadamard Classifier

In fixed Hadamard classifiers [10], the weight matrix is also fixed (i.e., not updated), and it is initialized from a Hadamard matrix. In this case, the Hadamard matrix is constructed using Sylvester's construction. Let $\mathbf{H}_1$ be a Hadamard matrix of order 1, defined as

$$\mathbf{H}_1 = \begin{bmatrix} 1 \end{bmatrix}. \tag{3.4}$$

Let k be any non-negative integer greater than 1. Higher order Hadamard matrices of order 2^k can be constructed using Hadamard matrices of the lower order 2^{k-1}, given as,

$$\mathbf{H}_{2^k} = \begin{bmatrix} \mathbf{H}_{2^{k-1}} & \mathbf{H}_{2^{k-1}} \\ \mathbf{H}_{2^{k-1}} & -\mathbf{H}_{2^{k-1}} \end{bmatrix}. \tag{3.5}$$

By iterating this process, we can obtain Hadamard matrices of order 1, 2, 4, ..., 2^k.

To construct the weight matrix, we would need to obtain a Hadamard matrix of order

2^k, where $k = \lceil \log_2 \max(n_c, n_k) \rceil$. Then the matrix is truncated to fit the size of the input and output, by taking its first n_c rows and first n_k columns.

For instance, if we have 3 output channels from the feature extractor $f(\cdot)$, i.e. $n_c = 3$, and we have 2 output categories, i.e. $n_k = 2$ then we know the input to the classifier $\mathbf{h} \in \mathbb{R}^3$ and the desired output is $\mathbf{y} \in \mathbb{R}^2$. To construct the weight matrix we can calculate $k = \lceil \log_2 \max(3, 2) \rceil = \lceil \log_2 3 \rceil = 2$, therefore we need to construct a Hadamard matrix of order $2^2 = 4$.

Using Sylvester's construction, we have

$$\mathbf{H}_1 = \begin{bmatrix} 1 \end{bmatrix},$$

$$\mathbf{H}_2 = \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix},$$

and finally

$$\mathbf{H}_4 = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & -1 & 1 & -1 \\ 1 & 1 & -1 & -1 \\ 1 & -1 & -1 & 1 \end{bmatrix}.$$

Then we can truncate $\mathbf{H}_4$ to obtain

$$\mathbf{W} = \begin{bmatrix} 1 & 1 & 1 \\ 1 & -1 & 1 \end{bmatrix}.$$

Then the output is obtained using the following calculation:

$$\mathbf{y} = \alpha \mathbf{W}\mathbf{h} + \mathbf{b} \ , \tag{3.6}$$

where α is a learned scalar parameter that is updated during back-propagation and $\mathbf{h}$ is the input to the classifier.

The fixed Hadamard classifier using this construction has a limitation. It cannot produce effective outputs when the output dimension is larger than that of the input. For instance when using it in ResNet-18 for classification of ImageNet-1K, the input is a vector of 512 dimensions, while the output needs to be 1000 dimensions. Here $\mathbf{W}$ has 1000 rows and 512 columns, and it is apparent that rows 513 through 1000 are identical to rows 1 through 488, resulting in the same intermediate results for all these items. The final results only differ because $\mathbf{b}$ could be different. This is also very apparent from observing the first two columns of $\mathbf{H}_4$ constructed earlier, the first two elements from rows 1 and 3, or rows 2 and 4, are the same.

3.4 Fixed Identity Classifier

This is the method featured in this work. Here, the final fully connected layer is completely removed, and the output from the global average pooling layer is directly used to compute classification scores. The global average pooling layer is immediately after the last convolution layer in the feature extractor. By removing the FC layer, it greatly reduces the number of parameters in the network. A depiction of this method is shown in Figure 3.1. Implementation wise, it is equivalent to setting the weight matrix $\mathbf{W}$ as an identity matrix $\mathbf{I}$, where all the elements on the diagonal are 1 and all other elements are 0. This matrix is not updated throughout training. The bias term, **b**, is also dropped.

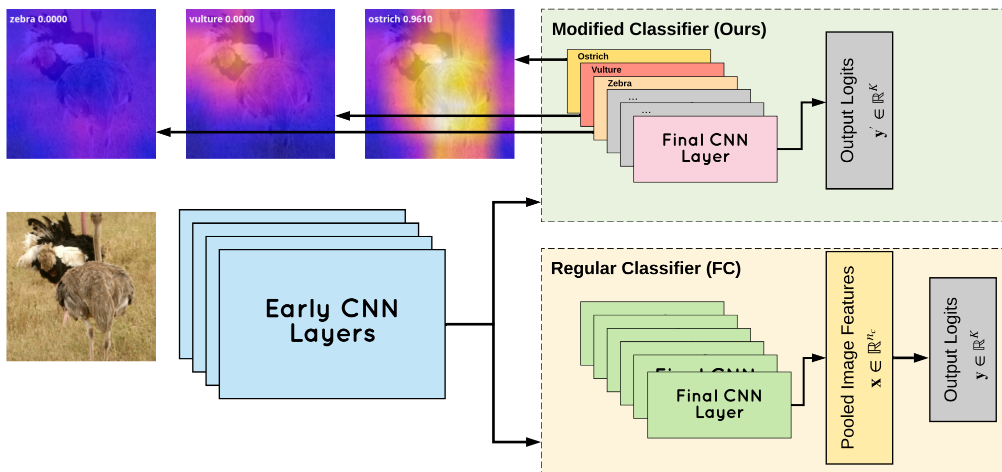


Figure 3.1: A depiction of the fixed identity classifier method.

This method offers an additional benefit: since each channel in the output of the final CNN layer represents an output class, it enables the outputs to be visualized immediately, similar to class activation mappings (CAM) [41]. Contrary to CAM which requires post-processing intermediate results from the neural network, this method can obtain these visualizations without any extra compute, during the forward pass (inference), along with obtaining the classification scores. The visualization results are demonstrated in Figure 3.1, using an image of ostrich from ImageNet-1K test set. As shown in the figure, they can be directly visualized to represent class-specific visualizations. The model produces high activation for regions with the correct class (ostrich), low activation for an unrelated class (zebra), and regions containing background objects (vulture).

This method suffers the same limitation as a fixed Hadamard classifier: it is unable to handle cases where the number of classification categories is greater than the number of channels from the last convolution layer. However, this research is not promoting the method as a drop-in replacement on existing architectures, it serves as a proxy tool to study the final classifier layer in current image classification architectures, and a possible method to design classifiers for future efficient architectures.

Chapter 4

Experiments

To demonstrate the effectiveness of fixed identity classifiers, the methods are evaluated across a variety of base architectures and datasets. All experiments are implemented using the Python programming language on PyTorch, an open-source machine learning framework.

4.1 Architectures

Several common residual networks, as well as mobile architectures that contain far fewer parameters, are chosen as the base architectures:

- **ResNet-18** – The ResNet-18 architecture is a common residual network consisting of 18 layers and skip connections to help gradient flow [9]. This architecture is used since it is the fastest residual network to train for ImageNet-1K classification.
- **ResNet-50** – ResNet-50 is a residual network with 50 layers and skip connections [9]. This architecture is chosen since it has been commonly used for computer vision

applications and achieves higher performance on ImageNet than ResNet-18.

- **ResNet-32** – This variant of ResNet is one variant that is optimized for the CIFAR image classification dataset, where the input image size differs from that used in ResNet-18 and ResNet-50.
- **DenseNet** – The Dense Convolutional Network takes the skip connection idea further [12]. In DenseNets, each layer has a skip connection to every other layer in a feed forward fashion. In this research, DenseNet-BC ($L = 100$, $k = 12$) is used to match the work in [10].
- **MobileNet v2** – MobileNet architectures are designed to efficiently run on mobile devices by replacing convolutional layers with depth-wise separable convolutions. The MobileNet v2 architecture [32] is used, which additionally uses bottlenecks and residual connections. This architecture is chosen since it is computationally efficient and using a fixed identity classifier can further reduce the network's memory requirements.
- **ShuffleNet v2 x0.5** – ShuffleNet architectures use point-wise group convolutions and bottleneck layers to run efficiently on mobile devices. A channel shuffle operation is applied on top of these operations to allow gradients to flow between different channel groups, which improves accuracy. ShuffleNet v2 additionally introduces a channel split operation [25]. In this research, ShuffleNet v2 with half-width (x0.5) is used.

For learned fully connected classifiers, the reference PyTorch implementations from the `torchvision` package are used when available, or implemented as described in the original work when the reference implementation is not available. For fixed Hadamard classifiers, the implementation is based on reference code and the source code provided in [10]. For fixed orthogonal classifiers, the reference implementation with FC is used, but the weights are initialized as a semi-orthogonal matrix and updates for the weight matrix is disabled, which is similar to the implementation in [10]. The fixed identity version simply removes the classifier, and truncates the output to the desired number of dimensions.

4.2 Datasets

Experiments are done on CIFAR-100 to quickly evaluate the performance of fixed identity classifiers. Then, experiments are performed on the ImageNet-1K dataset, demonstrating the robustness of the method on a large dataset with many categories. Additionally, experiments are performed on two smaller datasets to demonstrate the method's ability to perform transfer learning.

These datasets were chosen because they have a large number of classes, making it possible to test the method's capability of performing well, while also saving memory. The following datasets are chosen:

- **ImageNet-1K** – The ImageNet dataset consists of images from 1,000 categories from the internet [31]. Each category consists of 732-1,300 training examples and 50 validation examples, which are used for testing. This is a common large-scale image classification dataset that allows us to test the ability of the fixed identity classifier method to scale up and showcase its parameter savings.
- **CIFAR-100** – The CIFAR-100 dataset contains 100 classes each containing 600 color images of size 32×32 [17]. For each class, there are 500 images for training and 100 for testing.
- **Stanford Cars-196** – The Stanford Cars dataset consists of 196 car classes with

8,144 training and 8,041 testing images [16].

- **Flowers-102** – The Oxford Flowers dataset consists of 102 flower categories, with each class containing 40-258 images [27].

While CIFAR allows for a quick evaluation of different methods, ImageNet tests the ability of the method to scale up to a large number of categories. The Stanford Cars-196 and Flowers-102 datasets are used for evaluating the method's ability to perform fine-grained transfer learning tasks.

4.3 Implementation Details

4.3.1 General Details

PyTorch is used for all experiments. For CIFAR-100, every model on every architecture is trained from scratch. For the ImageNet results using a standard fully connected classification layer, the accuracy from the PyTorch pre-trained models are reported. For other classifiers on ImageNet, the models are trained from scratch. For all other experiments, each model is first initialized with pre-trained ImageNet weights and then fine-tuned on the target dataset.

For training on ImageNet and CIFAR-100, the original setups including methods for data augmentation [9,10,12] are used. For instance, for training ResNet-32 and DenseNet-BC on CIFAR-100, the following data augmentations are performed for training: 4 pixels are padded on each side, then a mirroring is applied at random, followed by cropping to 32×32 randomly. For testing, the original image is used, only normalization is applied. This follows the practice in their respective work. The specific hyperparameters for training the models are given in Table 4.1, both architectures use the stochastic gradient descent

(SGD) optimizer.

Table 4.1: CIFAR-100 training hyperparameters settings for each architecture.

Hyperparameter	ResNet-32	DenseNet-BC
Initial Learning Rate	0.1	0.1
Momentum	0.9	0.9
LR Decay Factor	10	10
LR Decay Epochs	[81, 122]	[150, 225]
Weight Decay	1.0×10^{-4}	1.0×10^{-4}
Batch Size	128	64
Total Epochs	164	300

For training on ImageNet, hyperparameters are given in Table 4.2. Note that MobileNet v2 uses the RMSProp optimizer. The training scheme for ResNet-18 and ShuffleNet V2 can be found in their original work [9, 25]. As for MobileNet V2, the training scheme is partially described in the original work [32], while also making a reference to to [38].

Table 4.2: ImageNet-1K training hyperparameters settings for each architecture.

Hyperparameter	ResNet-18	MobileNet v2	ShuffleNet V2
Optimizer	SGD	RMSProp	SGD
Initial LR	0.1	0.02	0.5
Momentum	0.9	0.9	0.9
LR Decay	$\times 0.1$ on 30, 60 epochs	$\times 0.98$ every epoch	Linear decay to 0
Weight Decay	1.0×10^{-4}	4.0×10^{-5}	4.0×10^{-5}
Batch Size	256	256	1024
Total Epochs	90	100	240

Parameters for the transfer learning experiments on Cars-196 and Flowers-102 are provided in Table 4.3. All networks were trained with stochastic gradient descent and momentum of 0.9 for 40 epochs, with a learning rate decay by a factor of 10 at 30 epochs. Optimal parameters were chosen using a grid search.

Table 4.3: Transfer learning parameter settings for each architecture.

Architecture	Learning Rate	Weight Decay	Batch Size
ResNet-18	0.01	1e-3	64
ResNet-50	0.01	1e-4	64
MobileNet v2	0.01	1e-4	64
ShuffleNet v2 x0.5	0.1	1e-4	64

4.3.2 Adapting ResNet-32 with the Fixed Identity Classifier on CIFAR-100

Despite the that the fixed identity classifier cannot work with architecture and dataset combination that has more classification categories than the dimensions of the feature vector, a slightly modified version of ResNet-32 is used to compare the effects of using a fixed identity classifier with the architecture to classify CIFAR-100.

The key idea is to modify the last convolution layer to output 100 channels instead of 64 channels. In convolutional networks that appeared before residual networks, this is very easy to implement. However, in ResNets, the network consists of major ”layers” (not individual layers), and each layer has several blocks. Within each block, there are several convolution and pooling operations, and in addition, there is a skip connection between the input and output in every block of the network. This means that the input to each block goes through an optional transformation and is added to the output from the final convolution layer in each block, skipping the other operations in between, and then this is used as the final output of the entire block. Therefore, simply modifying the last convolution layer will break the network.

There are several ways to implement the skip connection. Identity, zero-padding, and learned projection. In the case of identity, the input is added as-is, this mode can only be used when the number of input and output channels are the same. A learned projection is a learned one-by-one pointwise convolution filter so that the output can be of a different number of channels. Zero-padding means additional channels are created, but the values are all zero.

The original ResNet research showed that using projections in all skip connections is marginally better than using projections only when doubling the channels. And using identity and projection is slightly better than using identity and zero-padding.

To accommodate the modification of the last channel, several methods were explored in preliminary experiments. The results reported uses the zero-padding method as no method is particularly advantageous while zero-padding introduces the least number of new parameters.

Chapter 5

Results

All variants of the final classifiers are evaluated on multiple architectures and multiple datasets, to compare and demonstrate their ability to perform image classification. The learned classifier is used as the baseline. To see how much accuracy each classifier is sacrificing, the corresponding top-1 classification accuracy is compared against the baseline results.

5.1 Results on CIFAR-100

DenseNet-BC and ResNet-32 are trained to perform classification on CIFAR-100, while different methods are applied to implement its classifier. The results are shown in Table 5.1. For DenseNet-BC, all variants of the classifier are used; for ResNet-32, the fixed identity classifier is not used. This is because the fixed identity classifier is incapable of dealing with the feature extractor in ResNet-32, which outputs 64 channels, for 100 categories classification. However, ResNet-32 is trained with the fixed Hadamard classifier,

Table 5.1: Results on CIFAR with different models and different types of classifiers.

n_k	Architecture	Classifier	Top-1 Accuracy	Performance Gap
100	ResNet-32	Learned	69.46%	N/A
		Fixed Orthogonal	68.61%	-0.85%
		Fixed Hadamard	44.86%	-24.60%
	ResNet-32 w/ 100 ch. output	Learned	70.23%	N/A
		Fixed Identity	69.98%	-0.25%
	DenseNet-BC	Learned	77.61%	N/A
		Fixed Orthogonal	76.68%	-0.93%
		Fixed Hadamard	75.84%	-1.77%
		Fixed Identity	76.90%	**-0.71%**
64	ResNet-32	Learned	73.94%	N/A
		Fixed Orthogonal	73.92%	-0.02%
		Fixed Hadamard	73.97%	+0.03%
		Fixed Identity	74.25%	**+0.31%**

even though it is projected that it will not perform well.

A modified version of ResNet-32 as described in Section 4.3.2 is used to compare both a learned classifier and a fixed identity classifier on the same base architecture for the full CIFAR-100 dataset.

To evaluate all the classifiers on a vanilla version of ResNet-32 and CIFAR-100, a 64 categories subset of CIFAR-100 was used so that the number of categories does not exceed the number of output channels from the feature extractor.

This research was unable to reproduce results for DenseNet-BC using the fixed Hadamard classifier, using their original open-source code. In their original work, they report 77.67% for the test accuracy, only 75.84% was achieved in this work. However, the training setup used in this research is fair to all classifiers, therefore the performance gap still shows that not having a dedicated output layer is slightly superior to using a fixed Hadamard matrix, but not as good as having a learned fully connected classifier.

The results in Table 5.1 indicate that neither the fixed orthogonal classifier nor the fixed Hadamard classifier performs better than the fixed identity classifier, while being more complicated, while exhibiting the same weakness of not capable of working with feature extractors producing less channels than the desired number of classification categories.

In the experiments on the modified ResNet-32, although total parameter count did increase, the comparison between the two classifier methods is still fair. Multiple runs of the experiment was completed for both classifiers, and there was no statistical difference between the classification accuracy from the two methods, although the average for the learned classifier is still higher.

5.2 Results on ImageNet-1K

Moving on to a more challenging dataset, ResNet-18 with different classifiers are trained and evaluated on ImageNet-1K.

Similar to the situation before, due to limitations of the fixed Hadamard classifier and fixed identity classifier, the full 1000 categories are evaluated only on the learned classifier and the fixed orthogonal classifier. Then all classifiers are evaluated on the first 512 categories of ImageNet-1K, so that the Hadamard classifier and the fixed identity classifier can be compared. The results are shown in Table 5.2. The results indicate that while all fixed weights perform worse than learned weights, using a fixed identity matrix, which is equivalent to removing the classifier layer, outperforms both fixed orthogonal classifiers and fixed Hadamard classifiers.

Table 5.2: Results on ResNet-18 with each type of classifier, performing classification on ImageNet-1K and its subset.

n_k	Classifier	Top-1 Accuracy	Performance Gap
1000	Learned	69.76%	N/A
	Fixed Orthogonal	66.48%	-3.27%
512	Learned	77.87%	N/A
	Fixed Orthogonal	77.29%	-0.58%
	Fixed Hadamard	76.33%	-1.53%
	Fixed Identity	77.59%	**-0.28%**

Next, the fixed identity classifiers are used on the mobile architectures, MobileNet v2 and ShuffleNet V2, which are already very compact. Here, only the learned fully connected classifiers are being compared against, as it has been demonstrated that fixed Hadamard classifiers do not perform better. The results can be found in Table 5.3.

Table 5.3: Comparison of classification accuracy of the original ShuffleNet v2 and MobileNet v2 architectures with the fixed identity classifier method applied, trained on the 100 categories subset of ImageNet-1K.

K	Architecture	Classifier	Top-1 Acc.
1000	ShuffleNet V2 x0.5	Learned	60.55%
		Fixed Identity	53.06%
	MobileNet v2	Learned	71.88%
		Fixed Identity	71.03%
100	ShuffleNet V2 x0.5	Learned	72.94%
		Fixed Identity	74.42%

By removing the final layer, the model will see significant parameter savings: on ShuffleNet V2 x0.5 the savings is about 75%, and on MobileNet v2 it's about 37%. It is apparent that there is a non-trivial degradation in performance in the case of ShuffleNet V2. To evaluate whether this is due to the lack of parameters, or the modification to the architecture, the same tests are run on a very small subset of ImageNet, consisting of only 100 categories, and the results are also shown in Table 5.3. In this case, the fixed identity classifier does not perform worse than a learned classifier. Therefore the major performance gap on ImageNet-1K is likely due to the model being too small rather than the difference in the model architecture.

5.3 Scalability of fixed classifiers

Originally, the study of fixed classifiers, especially fixed Hadamard classifiers, was intended to find a classifier that is capable of scaling to more categories without using as many parameters.

It was quickly determined that a fixed Hadamard classifier does not scale past its input channels. Research is conducted on smaller subsets of ImageNet-1K to compare if a fixed Hadamard classifier performs better in any other case. ResNet-18 with different classifiers are trained on subsets of different sizes, the average top-1 accuracy over three runs are reported in Table 5.4.

The results fluctuate somewhat at different subset sizes, although the variance between three runs that only differ in random seeds is not very big. The fluctuations may be due to overfitting and/or the characteristics of specific categories. The relative performance plot is shown in Figure 5.1, and it is obvious that the fixed identity classifier is better than the fixed Hadamard classifier regardless of how the model scales.

Table 5.4: Top-1 accuracy results of different classifiers on smaller subsets of ImageNet-1K, using ResNet-18 as base architecture.

n_k	Learned	Fixed Hadamard	Fixed Identity
8	74.92%	74.00%	76.58%
16	83.21%	81.25%	83.71%
32	83.64%	81.38%	83.50%
64	75.51%	72.23%	75.63%
128	77.98%	73.82%	78.36%
256	79.03%	77.52%	78.54%
512	77.87%	76.33%	77.59%

5.4 Fine-Tuning with More Datasets

One issue with removing the classifier (replacing it with an identity matrix) is that it may harm the model's ability to perform transfer learning. This research demonstrates that the fixed identity classifier can be applied to more datasets and architectures, in transfer learning settings. Results with several architectures on the Stanford Cars-196 and Flowers-102 datasets are shown in Table 5.5, they reflect the average top-1 accuracy of three runs. The results are obtained by fine-tuning a model pretrained on ImageNet.

As the results indicate, the method works on ResNet-18, ResNet-50, MobileNet v2, and ShuffleNet v2 x0.5 on both datasets. It shows that fixed identity classifiers are able to achieve comparable results while using significantly fewer parameters, demonstrating its capabilities in transfer learning and generalization on more datasets.

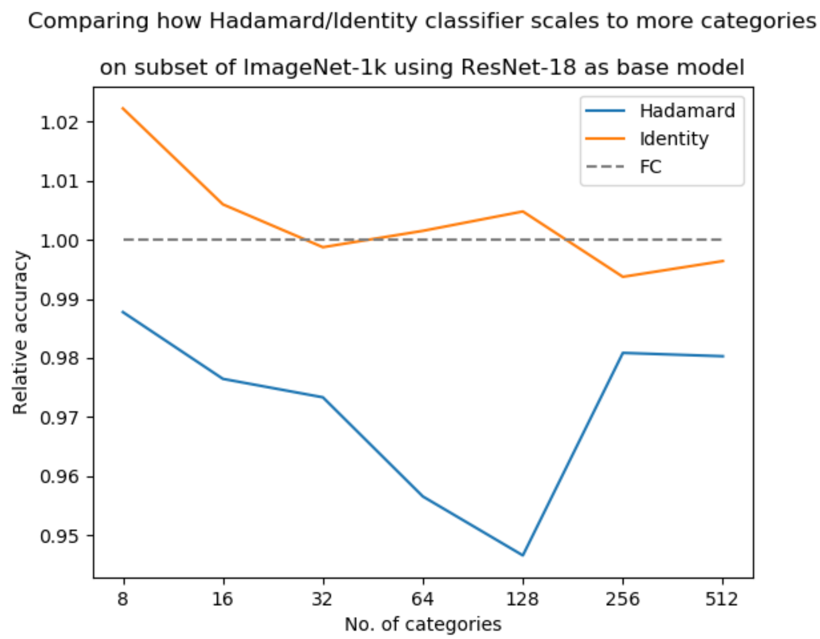


Figure 5.1: Relative performance of fixed Hadamard classifier and fixed identity classifier, against a learned classifier.

5.5 Feature Visualizations with ResNet-50

Visualizations of the final convolutional layer's outputs for ResNet-50 trained on ImageNet-1K are given in Figure 5.2. Unlike CAM, by using a fixed identity classifier, no additional post-processing is required to obtain class-specific visualizations.

Furthermore, despite being trained with only a single label per image, visualizing the final convolutional layer gives class-specific localization from a single forward pass. In Figure 5.3, several example images that were downloaded from the Internet are shown. They consist of multiple ImageNet-1K object categories, demonstrating that this method is able to produce object localization for free. By selecting multiple channels, this method can easily visualize activation maps for multiple categories. This is similar to the result

Table 5.5: Transfer learning performance evaluation of fixed identity classifiers on Cars-196 and Flowers-102 using multiple deep CNN architectures.

	Stanford Cars-196		
	Learned	Fixed Identity	Savings
ResNet-18	88.12%	86.06%	12.92%
ResNet-50	89.90%	90.35%	5.66%
MobileNet v2	87.68%	86.12%	24.26%
ShuffleNet V2 x0.5	77.99%	75.76%	66.65%
	Flowers-102		
	Learned	Fixed Identity	Savings
ResNet-18	93.42%	92.78%	16.83%
ResNet-50	95.06%	94.64%	5.10%
MobileNet v2.	94.24%	93.95%	21.66%
ShuffleNet V2 x0.5	87.75%	86.34%	63.52%

in [28]. However, that model uses multiple fully connected layers, requires using a sliding window method to process the image multiple times, and is trained with a multi-label training objective. In contrast, using fixed identity classifiers is fully convolutional and can handle input images of arbitrary size, and produces localization for all object categories in a single forward pass. This allows controlling the quality of the visualization simply by resizing the input during inference, as shown in Figure 5.4.

5.6 Attempts to improve the method

A few methods were explored to see if the results of the fixed identity classifier can be further improved.

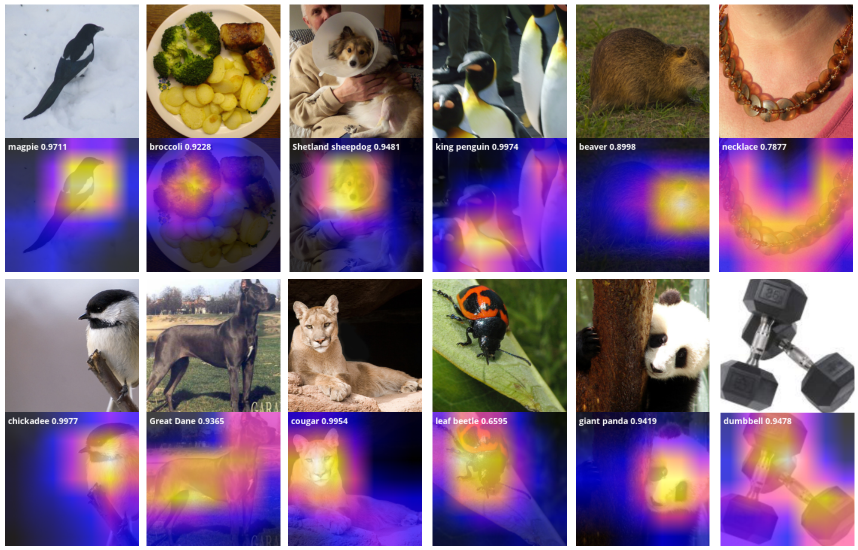


Figure 5.2: Visualizations using fixed identity classifier with the ResNet-50 architecture fine-tuned on ImageNet-1K. Maximally activated classes are visualized for each object. Normalized scores and class labels are shown in the top-left corner of each visualization.

5.6.1 Orthogonal initialization and regularization

In this attempt, the network was either initialized using (semi-)orthogonal matrices (and broadcast into tensors in some cases), or applied soft orthogonality regularization or double soft orthogonality regularization as described in [2], using weight of 0.025. ResNet-18 is used as the base architecture, and the networks are trained on 100 category subset of ImageNet-1K. Results are given in Table 5.6. Unless otherwise mentioned, the parameters are initialized using uniform He initialization described in [8].

From the results, it is clear that neither orthogonal initialization or orthogonal regularization can further improve the performance of fixed identity classifiers.

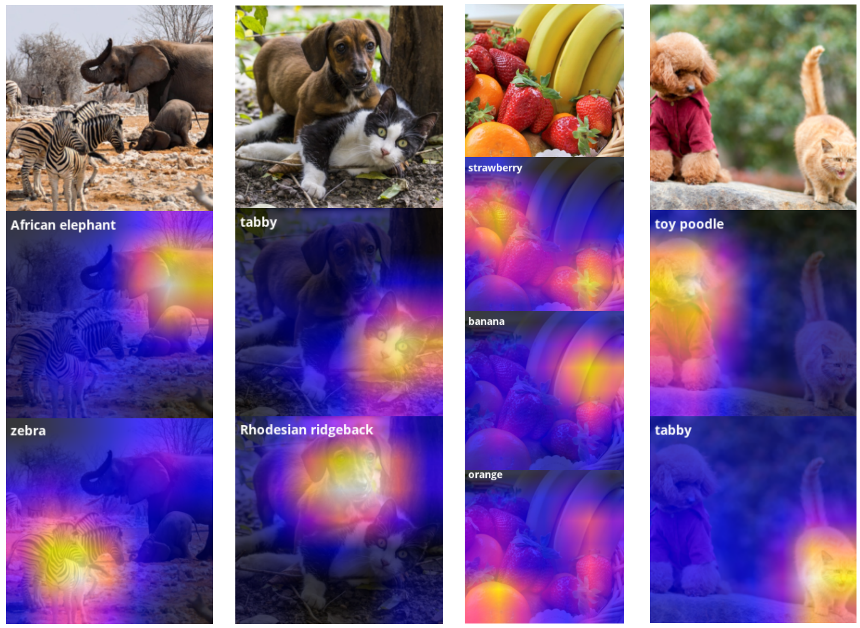


Figure 5.3: Feature map visualizations for images contain multiple categories from ImageNet-1K.

5.6.2 Alternative pooling methods

Power-average pooling and soft attention pooling are also explored.

When using global power-average pooling in this setting, there is one parameter p, and the pooling is given as

$$g(\mathbf{f}) = \sqrt[p]{\sum_{f \in \mathbf{f}} f^p} \ , \tag{5.1}$$

where each element is power-averaged per channel. In the case of $p = 1$, it is equivalent to sum pooling, which is proportional to average pooling; in the case of $p = \infty$ it is equivalent to max pooling.

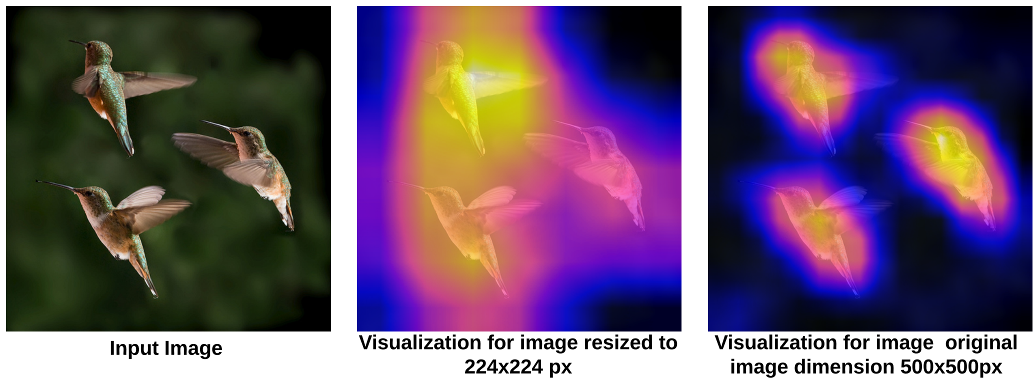


Figure 5.4: CNN visualization for original image size (500px $\times$ 500px) and resized (224px $\times$ 224px)

Table 5.6: Top-1 accuracy results of different orthogonal initialization and regularization configurations, using ResNet-18 as base on ImageNet-1K 100 category subset.

Orthogonal Initialization	Orthogonal Regularization	Top-1 Accuracy
None	None	**81.20%**
Final Conv. Layer	None	80.58%
All Conv. Layers	None	77.44%
None	Final Conv. Layer	80.24%
None	All Conv. Layers	78.80%
Final Conv. Layer	Final Conv. Layer	80.76%

In soft attention pooling, an additional module is created, it has two fully connected layers with $\tanh(\cdot)$ as non-linear activation, and the number of hidden units is variable. It takes the flattened feature map as input, and the output goes through $\text{softmax}(\cdot)$ activation before being used weights for summing the feature maps.

Models based on the ResNet-18 architecture were trained for ImageNet-1K 100 category subset. Neither offers a significant boost to accuracy when using fixed identity classifiers. For the sake of brevity, the detailed results are omitted. Furthermore, due to modifying the pooling operation, this makes free feature visualization unobtainable.

5.7 Summary

The fixed identity classifier was evaluated in multiple configurations, and compared against other fixed classifiers. In general, all fixed classifiers perform worse than a learned fully connected classifier. However, among the fixed classifiers, the fixed identity classifier performs best overall, while being the most simple method.

Chapter 6

Discussions

This research is primarily driven by the work in fixed classifiers, which claims to be more efficient while maintaining performance [10]. In this research, fixed classifiers are put to the test, against learned classifiers, and also against the fixed identity classifier, which is equivalent to removing the fully connected classifier layer. This is an unorthodox approach, because traditionally CNN architectures have a feature extractor followed by a classifier. In the method proposed here, the classifier is removed, and classification scores are directly obtained from the last convolutional layer. Comparing to conventional networks, this is equivalent to removing the classifier; compared to fixed classifiers, this is equivalent to using an identity matrix as the fixed weights, which is a matrix that contains as little information as possible. This method can serve as a proxy to evaluate both learned fully connected classifiers, as well as fixed classifiers with specifically designed weights.

6.1 Benefits over other fixed classifiers

In all experiments that involve both fixed identity and fixed Hadamard classifiers conducted in this research, the fixed identity classifier outperforms the fixed Hadamard classifier. This answers the question, whether fixed classifiers help the model learn anything. The results presented in this research show that specially designed weight matrices do not help the model learn to better classify. These designed fixed weights does force the feature extractor to produce features with specific characteristics, as presented in [29]. While also being capable of producing a classification score comparable to learned classifiers, it is actually worse than using a simple identity matrix.

While other Hadamard matrices exist (other than those constructed using the Sylvester's method), Hoffer *et al.* does not use them in their source code. Also, they do not explain the rationale for why Hadamard matrices are beneficial, other than the fact that it does not require updating and is more efficient in terms of computation costs. Compute efficiency will be discussed later.

6.2 Parameter efficiency

On the large-scale ImageNet dataset and smaller CIFAR-100 dataset, along with two even smaller transfer learning datasets, the fixed identity classifier demonstrates it only suffers a relatively small sacrifice in accuracy, compared against learned classifiers, while saving a lot of parameters. Furthermore, on mobile architectures such as MobileNets and ShuffleNets that already reduce the total number of parameters required by a model, using a fixed identity classifier can reduce these memory requirements even further (e.g., 39% reduction for MobileNet v2 and 75% reduction for ShuffleNet V2, both on ImageNet) with only a small degradation in performance, thus improving the parameter efficiency of models.

There is a greater degradation of ImageNet-1K classification performance when using mobile architectures in conjunction with this method. In these scenarios, a significant amount of parameters are removed from the model, and in the case of ShuffleNet V2 x0.5, around 75% parameters are removed, leaving the model with only 0.3M parameters, compared to 1.3M parameters of the vanilla model. Results on ImageNet-100 showed that there is no performance degradation, which implies that the performance gap on ImageNet-1K is due to the model being too small to capture the statistics of the dataset. This suggests that while the final classifier layer uses a lot of parameters, it does not contribute much to the classification accuracy.

This means while fixed identity classifiers are not a drop-in replacement in some cases, the conventional approach of having a fully connected classifier is not very efficient in terms of the model size.

Furthermore, one could additionally make use of network pruning [1,7,11,20,21,24,36] to explicitly reduce parameters even further. Another option is to use network quantization to store parameters at a lower precision to save disk space and improve computational efficiency [5,14,15]. Also, it is possible to specifically promote sparsity in the final classifier using *L1* regularization, using a learned final classifier.

While a lot of parameters can be saved in the final classifier, many models are very deep and wide, consisting of tens and even hundreds of millions of parameters. To these non-mobile models, the parameters in the fully connected final classifier can be negligible. It is debatable whether compressing the final FC layer is very useful in these scenarios. Despite this, as models get more complicated and are used for classification of datasets with an even larger number of categories, an alternative to a fully connected classifiers

Table 6.1: Compute cost for different components in different architecture in FLOPs, and percentage of compute the final fully connected classifier accounts for.

Architecture	Total	First Conv.	Final Classifier (FC)	% FC
ResNet-50	4.12G	118M	2.05M	0.05%
ResNet-18	1.82G	118M	512K	0.03%
MobileNet v2	320M	10.8M	1.28M	0.40%
ShuffleNet V2 x0.5	43.6M	8.13M	1.02M	2.35%

may be helpful.

6.3 Compute efficiency

One argument for using a fixed Hadamard classifier is that: 1) by not updating the weights during training 2) by using only +1 and -1 in the weights which simplifies calculation to use only inversions and summing , can significantly reduce computation costs. By using fixed identity classifiers proposed in this research, the cost for computation is even lower.

However, by looking at a bigger picture, when taking into consideration the entire network, saving a single matrix-vector multiplication is negligible. Table 6.1 shows the compute requirements for different architectures in terms of floating-point operations.

As the numbers indicate, except for in ShuffleNet V2 x0.5, the final fully connected classifier layer uses more than 1% of the total compute, the FC layer uses a negligible amount of computation. And even in the case of ShuffleNet V2 x 0.5 which FC accounts for 75% of total parameters, the compute is only 2.35%.

Furthermore, while both Hadamard [10] and Binarized Neural Networks [5] argue for special hardware designs that can further improve efficiency, it is hard to imagine a hardware that implements accelerated convolutions and general matrix multiplication (GEMM, level 3 BLAS) but does not have generalized matrix-vector multiplication (level 2 BLAS). Even if that is the case, it is not difficult to perform a single matrix-vector multiplication using the existing GEMM hardware.

6.4 Summary

While the fixed identity classifier yields comparable performance to a standard classifier when trained on ImageNet for all architectures tested, and completely outperforms other fixed classifiers in many ways, it still has a lot of limitations and pitfalls. It is incapable of handling more classes than the number of channels of output categories, which is the same for fixed Hadamard classifiers. It does reduce the computation requirements, but the effect is not very significant in the grand scheme of things in a deep CNN for image classification.

Despite the caveats of fixed identity classifiers, the results indicate that the final output layer does not need to be a learned fully connected layer. The final output layer in deep neural network architectures for image classification contains a lot of redundancy and can be greatly compressed for more efficiency. The results can be insightful for future efficient architecture design and/or efficient neural architecture search, enabling models to more easily scale to handle even larger datasets.

Chapter 7

Conclusion

In this work, the performance and efficiency of various fixed classifier methods are evaluated and compared against each other, and conventional learned classifiers. This work proposed the elimination of the fully connected classifier, and evaluated its performance on several modern CNN base architectures. By using global average pooling to compute classification predictions directly from the final convolutional layer, it is possible to achieve comparable performance to several CNNs that use a fully connected layer, while greatly reducing the total number of parameters required by the model, proving that specially designed fixed classifiers are not as effective as simply removing the final layer from networks, both in terms of parameter efficiency and classification accuracy. Research also showed that this approach is able to work on multiple datasets and neural network architectures.

It is also demonstrated that using a fixed identity classifier is not only simpler, but also helpful in the visualization of the neural network features. It can generate visualizations similar to class activation maps, while requiring no additional post-processing.

This work also explored several methods that attempt to close the gap between this fixed identity classifier method and learned fully connected classifiers. It was demonstrated that all these patchwork are of no avail.

Finally, this work demonstrated that the final classifier in general is not very efficient in terms of parameter size, and does not contribute very much to classification accuracy. While it was discussed that neither of the alternative methods offers significant improvements in terms of computational efficiency, this work still suggests future neural architecture designs should use output layers more efficient than fully connected layers, in terms of parameter count.

Appendices

Appendix A

Source Code (Selection)

A.1 Model Architectures

`./models_implementation/resnet_cifar_altskipconn.py`
This implements more alternatives of the skip connection in the last block for ResNet-32.

```
import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.nn.init as init
import random
from textwrap import dedent
import math
from models_implementation.clsf_utils import __fixed_eye, __no_bias, \
    generate_hadamard, generate_orthoplex, generate_cube_ordered,
    ↪ generate_cube_random


__all__ = []


```

```python
def _weights_init(m):
    if isinstance(m, nn.Linear) or (isinstance(m, nn.Conv2d) and not isinstance(m, FixedConv2d)):
        init.kaiming_normal_(m.weight)


class FixedConv2d(nn.Conv2d):
    pass  # just a hack to change signature


class LambdaLayer(nn.Module):
    def __init__(self, lambd):
        super(LambdaLayer, self).__init__()
        self.lambd = lambd

    def forward(self, x):
        return self.lambd(x)


class BasicBlock(nn.Module):
    expansion = 1

    def __init__(self, in_planes, planes, stride=1, option='A'):
        super(BasicBlock, self).__init__()
        self.conv1 = nn.Conv2d(in_planes, planes, kernel_size=3, stride=stride, padding=1, bias=False)
        self.bn1 = nn.BatchNorm2d(planes)
        self.conv2 = nn.Conv2d(planes, planes, kernel_size=3, stride=1, padding=1, bias=False)
        self.bn2 = nn.BatchNorm2d(planes)

        self.shortcut = nn.Sequential()
        if stride != 1 or in_planes != planes:
            if option == 'A':
                """
                For CIFAR10 ResNet paper uses option A.
                """
```

```
                self.shortcut = LambdaLayer(lambda x:
                                            F.pad(x[:, :, ::2, ::2], (0, 0, 0, 0, planes//4, planes//4), "constant", 0))
            elif option == 'B':
                self.shortcut = nn.Sequential(
                     nn.Conv2d(in_planes, self.expansion * planes, kernel_size=1, stride=stride, bias=False),
                     nn.BatchNorm2d(self.expansion * planes)
                )

    def forward(self, x):
        out = F.relu(self.bn1(self.conv1(x)))
        out = self.bn2(self.conv2(out))
        out += self.shortcut(x)
        out = F.relu(out)
        return out


class ClsfBlock(nn.Module):
    expansion = 1

    def __init__(self, in_planes, planes, stride=1, option='A', num_classes=100):
        super(ClsfBlock, self).__init__()
        self.conv1 = nn.Conv2d(in_planes, planes, kernel_size=3, stride=stride, padding=1, bias=False)
        self.bn1 = nn.BatchNorm2d(planes)
        self.conv2 = nn.Conv2d(planes, num_classes, kernel_size=3, stride=1, padding=1, bias=False)
        self.bn2 = nn.BatchNorm2d(num_classes)
        self.shortcut = nn.Sequential()
        if stride != 1 or in_planes != num_classes:
            if option == 'A':
                """
                For CIFAR10 ResNet paper uses option A.
```

```
            """
            pad_size = int((num_classes - in_planes)//2)
            self.shortcut = LambdaLayer(lambda x:
                                        F.pad(x, [0, 0, 0, 0,
                                        ↪  pad_size, pad_size],
                                        ↪  "constant", 0))
        elif option == 'B':
            self.shortcut = nn.Sequential(
                 nn.Conv2d(in_planes, num_classes, kernel_size=1,
                 ↪  stride=stride, bias=False),
                 nn.BatchNorm2d(num_classes)
            )
        elif option == 'C':  # Hadamard and scaling
            h = generate_hadamard(in_planes, num_classes)
            h = h.view(num_classes, in_planes, 1, 1)
            conv = FixedConv2d(in_planes, num_classes, kernel_size=1,
            ↪  stride=stride, bias=False)
            conv.weight.data = h.float()
            conv.weight.requires_grad_(False)

            init_scale = 1. / math.sqrt(num_classes)
            self.scale = nn.Parameter(torch.tensor(init_scale))
            self.shortcut = nn.Sequential(
                conv,
                LambdaLayer(lambda x: - self.scale * x),
                nn.BatchNorm2d(num_classes)
            )
        elif option == 'D':  # Fixed Orthoplex
            w = torch.tensor(generate_orthoplex(in_planes,
            ↪  num_classes))
            w = w.view(num_classes, in_planes, 1, 1)
            conv = FixedConv2d(in_planes, num_classes, kernel_size=1,
            ↪  stride=stride, bias=False)
            conv.weight.data = w.float()
            conv.weight.requires_grad_(False)
            self.shortcut = nn.Sequential(
                conv,
```

```
                nn.BatchNorm2d(num_classes)
            )
        elif option == 'E':  # shuffled fixed Orthoplex, using all channels at least once (64x +1 then 36x -1)
            w = torch.zeros(num_classes, in_planes)
            for row in range(num_classes):
                col = row % in_planes
                w[row, col] = 1 if row < in_planes else -1
            w = w .view(num_classes, in_planes, 1, 1)
            conv = FixedConv2d(in_planes, num_classes, kernel_size=1, stride=stride, bias=False)
            conv.weight.data = w.float()
            conv.weight.requires_grad_(False)
            self.shortcut = nn.Sequential(
                conv,
                nn.BatchNorm2d(num_classes)
            )
        elif option == 'F':  # d-cube ordered
            w = generate_cube_ordered(64, 100)
            w = w.view(num_classes, in_planes, 1, 1)
            conv = FixedConv2d(in_planes, num_classes, kernel_size=1, stride=stride, bias=False)
            conv.weight.data = w.float()
            conv.weight.requires_grad_(False)
            self.shortcut = nn.Sequential(
                conv,
                nn.BatchNorm2d(num_classes)
            )
        elif option == 'G':  # d-cube random
            w = generate_cube_random(64, 100)
            w = w.view(num_classes, in_planes, 1, 1)
            conv = FixedConv2d(in_planes, num_classes, kernel_size=1, stride=stride, bias=False)
            conv.weight.data = w.float()
            conv.weight.requires_grad_(False)
            self.shortcut = nn.Sequential(
                conv,
```

```
                    nn.BatchNorm2d(num_classes)
                )
            elif option == 'H':  # d-cube some better ordering that I can think of
                raise NotImplementedError  # don't know how to do it yet
            else:
                raise NotImplementedError

    def forward(self, x):
        out = F.relu(self.bn1(self.conv1(x)))
        out = self.bn2(self.conv2(out))
        out += self.shortcut(x)
        return out


class ResNet_alt(nn.Module):
    def __init__(self, block, num_blocks, num_classes=100, option='A'):
        super(ResNet_alt, self).__init__()
        self.clsf_expansion_option = option
        self.in_planes = 16
        self.conv1 = nn.Conv2d(3, 16, kernel_size=3, stride=1, padding=1, bias=False)
        self.bn1 = nn.BatchNorm2d(16)
        self.layer1 = self._make_layer(block, 16, num_blocks[0], stride=1)
        self.layer2 = self._make_layer(block, 32, num_blocks[1], stride=2)
        self.layer3 = self._make_layer(block, 64, num_blocks[2], stride=2, num_classes=num_classes)
        self.fc = nn.Linear(100, num_classes)
        self.apply(_weights_init)

    def _make_layer(self, block, planes, num_blocks, stride, num_classes=None):
        strides = [stride] + [1]*(num_blocks-1)
        layers = []
        for idx, stride in enumerate(strides):
```

```
            if (num_classes is not None) and (idx == len(strides) - 1):
                layers.append(ClsfBlock(self.in_planes, planes, stride, self.clsf_expansion_option, num_classes))
            else:
                layers.append(block(self.in_planes, planes, stride))
                self.in_planes = planes * block.expansion

        return nn.Sequential(*layers)

    def forward(self, x):
        out = F.relu(self.bn1(self.conv1(x)))
        out = self.layer1(out)
        out = self.layer2(out)
        out = self.layer3(out)
        out = F.avg_pool2d(out, out.size()[3])
        out = out.view(out.size(0), -1)
        out = self.fc(out)
        return out


for option in ['a', 'b', 'c', 'd', 'e', 'f', 'g']:
    code = f"""\
    def rn32_cf100_ex{option}():
        model = ResNet_alt(BasicBlock, [5, 5, 5], option='{option.upper()}')
        return model

    def rn32_cf100_ex{option}_fixed_eye():
        model = rn32_cf100_ex{option}()
        model = __fixed_eye(model)
        return model

    def rn32_cf100_ex{option}_no_bias():
        model = rn32_cf100_ex{option}()
        model = __no_bias(model)
        return model

```

```
    def rn32_cf100_ex{option}_fixed_eye_no_bias():
        model = rn32_cf100_ex{option}()
        model = __no_bias(model)
        model = __fixed_eye(model)
        return model
    """
    exec(dedent(code))
    __all__ += [f'rn32_cf100_ex{option}', f'rn32_cf100_ex{option}_fixed_eye',
                f'rn32_cf100_ex{option}_no_bias', f'rn32_cf100_ex{option}_fixed_eye_no_bias']
```

`./models_implementation/resnet_orthogonal.py`

This implements various modifications based on ResNet-18.

```python
from collections import OrderedDict
import inspect

import torch.nn as nn
from torchvision.models import resnet18, ResNet

from .resnet_imagenet import Bias
from .hadamard_3rdpty import HadamardProj


class LambdaLayer(nn.Module):
    """Module/Layer that encapsulates a single function for PyTorch

    This is to make it easier to a lambda in an nn.Sequential() container.
    """

    def __init__(self, lm):
        """

        Args:
            lm (Callable): function to use/call when the module is called.
        """
        super().__init__()
        self.lm = lm
        # this is because I want to see whatever the anonymous function is
        # but I do not know how to parse python syntax or want to learn to write a parser now
        self.src = inspect.getsourcelines(self.lm)
        if len(self.src[0]) == 1:
            module_code_str: str = self.src[0][0]
            lam_start_pos = module_code_str.find("lambda")
```

```python
            # the case where def f(x): ... is a one liner
            if lam_start_pos == -1 and module_code_str[:4] == 'def ':
                xtr_repr = module_code_str.strip('\r\n')
            else:
                xtr_repr = module_code_str[lam_start_pos:]  # finds the start of "lambda..."
                xtr_repr = xtr_repr.strip(')\r\n')  # removes trailing parenthesis and newlines
            self.xtr_repr = xtr_repr
        else:
            self.xtr_repr = '(not lambda)'

    def forward(self, *input):
        return self.lm(*input)

    def extra_repr(self) -> str:
        return self.xtr_repr


class SoftAttentionPooling(nn.Module):
    def __init__(self, in_channels: int, middle_channels: int):
        super().__init__()
        self.in_channels = in_channels
        self.middle_channels = middle_channels

        self.attention = nn.Sequential(
            nn.Conv1d(in_channels, middle_channels, kernel_size=1),
            nn.Tanh(),
            nn.Conv1d(middle_channels, 1, kernel_size=1)
        )

    def forward(self, x):
        n, c = x.size(0), x.size(1)
        x = x.view(n, c, -1)
        summarized = self.attention(x).view(n, -1)
        att = nn.functional.softmax(summarized, 1)
        x = (x * att.unsqueeze(1)).sum(2)
```

```
        return x


class RepackagedResNet18(nn.Module):
    def __init__(self, pretrained: bool):
        super().__init__()
        orig_resnet: ResNet = resnet18(pretrained=pretrained)
        self.features = nn.Sequential(
            OrderedDict([
                ('conv1', orig_resnet.conv1),
                ('bn1', orig_resnet.bn1),
                ('relu1', orig_resnet.relu),
                ('maxpool1', orig_resnet.maxpool),
                ('layer1', orig_resnet.layer1),
                ('layer2', orig_resnet.layer2),
                ('layer3', orig_resnet.layer3),
                ('layer4', orig_resnet.layer4)
            ]))
        self.classifier = nn.Sequential(
            nn.AdaptiveAvgPool2d((1, 1)),
            nn.Flatten(),
            orig_resnet.fc
        )

    def forward(self, x):
        x = self.features(x)
        y = self.classifier(x)
        return y


def rn18_3x3clsf():
    model = RepackagedResNet18(pretrained=False)
    model.classifier = nn.Sequential(
        OrderedDict([
        ('conv', nn.Conv2d(512, 1000, kernel_size=3)),
        ('pool', nn.AdaptiveAvgPool2d((1, 1))),
        ('flatten', nn.Flatten())
```

```
    ]))
    return model


# note on this one
# it is semi-orthogonal
def rn18_orthogonal_fc():
    model = RepackagedResNet18(pretrained=False)
    fc = nn.Linear(512, 1000, bias=True)
    nn.init.orthogonal_(fc.weight.data)
    fc.weight.requires_grad_(False)
    model.classifier[2] = fc
    return model


# this is truly orthogonal
def rn18_512_orthogonal_fc():
    model = RepackagedResNet18(pretrained=False)
    fc = nn.Linear(512, 512, bias=True)
    nn.init.orthogonal_(fc.weight.data)
    fc.weight.requires_grad_(False)
    model.classifier[2] = fc
    return model


# this is semi-orthogonal but the other way around
def rn18_256_orthogonal_fc():
    model = RepackagedResNet18(pretrained=False)
    fc = nn.Linear(512, 256, bias=True)
    nn.init.orthogonal_(fc.weight.data)
    fc.weight.requires_grad_(False)
    model.classifier[2] = fc
    return model


def rn18_512_scratch():
    model = RepackagedResNet18(pretrained=False)
```

```python
    model.classifier[2] = nn.Linear(512, 512)
    return model

def rn18_512_id_scratch():
    model = RepackagedResNet18(pretrained=False)
    model.classifier[2] = nn.Identity()
    return model

def rn18_512_id_bias_scratch():
    model = rn18_512_id_scratch()
    model.classifier.add_module('bias', Bias(512, 512))
    return model

def rn18_512_hadamard_scratch():
    model = RepackagedResNet18(pretrained=False)
    model.classifier[2] = HadamardProj(512, 512)
    return model


def rn18_l4_1a_nc_1k():
    model = RepackagedResNet18(pretrained=True)
    model.classifier = nn.Sequential(
        nn.AdaptiveAvgPool2d((1, 1)),
        nn.Flatten()
    )
    model.features.layer4[1].bn2 = nn.Sequential()
    model.features.layer4[1].conv2 = nn.Conv2d(512, 1000, kernel_size=(3,
    ↪  3), stride=(1, 1), padding=(1, 1))
    pad_size = int((1000 - 512) / 2)
    pad_param = (0, 0, 0, 0, pad_size, pad_size)
    model.features.layer4[1].downsample = LambdaLayer(lambda x:
    ↪  nn.functional.pad(x, pad_param, 'constant', 0))
    return model


def rn18_l4_1a_orthogonal_nc_1k():
    model = rn18_l4_1a_nc_1k()
```

```
    nn.init.orthogonal_(model.features.layer4[1].conv2.weight)
    return model


def rn18_l4_1a_all_conv_orthogonal_nc_1k():
    model = rn18_l4_1a_nc_1k()
    for m in model.modules():
        if type(m) is nn.Conv2d:
            nn.init.orthogonal_(m.weight)
    return model


def rn18_l4_1a_relu_before_pool_nc_1k():
    model = rn18_l4_1a_nc_1k()
    model.classifier = nn.Sequential(
        nn.ReLU(inplace=False),
        nn.AdaptiveAvgPool2d((1, 1)),
        nn.Flatten()
    )
    return model

def rn18_l4_1a_nc_1k_scratch():
    model = RepackagedResNet18(pretrained=False)
    model.classifier = nn.Sequential(
        nn.AdaptiveAvgPool2d((1, 1)),
        nn.Flatten()
    )
    model.features.layer4[1].bn2 = nn.Sequential()
    model.features.layer4[1].conv2 = nn.Conv2d(512, 1000, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    pad_size = int((1000 - 512) / 2)
    pad_param = (0, 0, 0, 0, pad_size, pad_size)
    model.features.layer4[1].downsample = LambdaLayer(lambda x: nn.functional.pad(x, pad_param, 'constant', 0))
    return model

def rn18_l4_1a_maxpool_nc_1k():
```

```python
    model = rn18_l4_1a_nc_1k_scratch()
    model.classifier = nn.Sequential(
        nn.AdaptiveMaxPool2d((1, 1)),
        nn.Flatten()
    )
    return model


def rn18_l4_1a_lppool_p2():
    model = rn18_l4_1a_nc_1k_scratch()
    model.classifier = nn.Sequential(
        nn.LPPool2d(2, kernel_size=7),
        LambdaLayer(lambda x: x / 7),
        nn.Flatten()
    )
    return model

def rn18_l4_1a_lppool_p1_5():
    model = rn18_l4_1a_nc_1k_scratch()
    model.classifier = nn.Sequential(
        nn.LPPool2d(1.5, kernel_size=7),
        LambdaLayer(lambda x: x / 13.390518),
        nn.Flatten()
    )
    return model

def rn18_l4_1a_lppool(p: float):
    n = 49 ** (1/p)
    model = rn18_l4_1a_nc_1k_scratch()
    model.classifier = nn.Sequential(
        nn.LPPool2d(p, kernel_size=7),
        LambdaLayer(lambda x: x / n),
        nn.Flatten()
    )
    return model

def rn18_l4_1a_lppool_p0_5():
```

```
    return rn18_l4_1a_lppool(0.5)

def rn18_l4_1a_lppool_p1_0():
    return rn18_l4_1a_lppool(1.0)

def rn18_l4_1a_lppool_p4_0():
    return rn18_l4_1a_lppool(4.0)


def rn18_l4_1a_soft_attention_pool(units: int):
    model = rn18_l4_1a_nc_1k_scratch()
    model.classifier = nn.Sequential(
        SoftAttentionPooling(1000, units)
    )
    return model

def rn18_l4_1a_soft_attention_pool_32():
    return rn18_l4_1a_soft_attention_pool(32)

def rn18_l4_1a_soft_attention_pool_64():
    return rn18_l4_1a_soft_attention_pool(64)

def rn18_l4_1a_soft_attention_pool_128():
    return rn18_l4_1a_soft_attention_pool(128)

def rn18_l4_1a_soft_attention_pool_256():
    return rn18_l4_1a_soft_attention_pool(256)

def rn18_l4_1a_soft_attention_pool_512():
    return rn18_l4_1a_soft_attention_pool(512)

def rn18_l4_1a_soft_attention_pool_1024():
    return rn18_l4_1a_soft_attention_pool(1024)
```

./models_implementation/clsf_utils.py
This implements several utility functions for modifying architectures.

```
import math
import random

import torch


def generate_hadamard(in_features, out_features):
    from scipy.linalg import hadamard
    n = math.ceil(math.log2(max(in_features, out_features)))
    h = hadamard(2**n)
    return torch.tensor(h[:out_features, :in_features])


def generate_orthoplex(in_features, out_features):
    t = torch.zeros(out_features, in_features)
    for row in range(out_features):
        col = row // 2
        t[row, col] = (-1) ** row
    return t


def generate_cube_ordered(in_features, out_features):
    t = torch.ones(out_features, in_features)
    for row in range(out_features):
        binary_coded = f'{{0:0{in_features}b}}'
        binary_coded = binary_coded.format(row)
        for col, val in enumerate(binary_coded):
            t[row, col] = (-1)**int(val)
    return t / math.sqrt(in_features)


def generate_cube_random(in_features, out_features):
    t = torch.ones(out_features, in_features)
    # FIXME: This causes ValueError: Maximum allowed size exceeded
```

```
    rnd_vector_numbers = set()
    while len(rnd_vector_numbers) < out_features:
        rnd_vector_numbers.add(random.randint(0, 2**in_features - 1))
    rnd_vector_numbers = list(rnd_vector_numbers)
    for row in range(out_features):
        binary_coded = f'{{0:0{in_features}b}}'
        binary_coded = binary_coded.format(rnd_vector_numbers[row])
        for col, val in enumerate(binary_coded):
            t[row, col] = (-1)**int(val)
    return t / math.sqrt(in_features)


def __fixed_eye(model):
    torch.nn.init.eye_(model.fc.weight.data)
    model.fc.weight.requires_grad_(False)
    return model


def __no_bias(model):
    model.fc = torch.nn.Linear(model.fc.in_features,
    ↪  model.fc.out_features, bias=False)
    return model
```

A.2 Main Script

./main.py
This script sets up and runs experiments. It is invoked by other scripts to automatically run experiments in batches.

```
#!/usr/bin/env python3

import argparse
import os
import random
import shutil
import time
import warnings
import sys

import numpy as np

import torch
import torch.nn as nn
import torch.nn.parallel
import torch.backends.cudnn as cudnn
import torch.distributed as dist
import torch.optim
import torch.multiprocessing as mp
import torch.utils.data
import torch.utils.data.distributed
import torchvision.transforms as transforms
import torchvision.datasets as datasets

import models
import datasets
import optimizers
model_names = sorted(name for name in models.__dict__
    if name.islower() and not name.startswith("__")
    and callable(models.__dict__[name]))

```

```
dataset_names = sorted(name for name in datasets.__dict__
                       if name.islower() and not name.startswith('__')
                       and callable(datasets.__dict__[name]))

optimizer_names = sorted(name for name in optimizers.__dict__
                         if name.islower() and not name.startswith('__')
                         and callable(optimizers.__dict__[name]))


parser = argparse.ArgumentParser(description='PyTorch Training')
parser.add_argument('-a', '--arch', metavar='ARCH', required=True,
                    choices=model_names,
                    help=f"model architecture: {'/'.join(model_names)}")
parser.add_argument('-d', '--dataset', metavar='DATASET', required=True,
                    choices=dataset_names,
                    help=f"dataset to use: {'/'.join(dataset_names)}")
parser.add_argument('--optimizer', metavar='OPTIM', required=True,
                    choices=optimizer_names,
                    help=f"optimizer/lr_scheduler to use: {'/'.join(optimizer_names)}")
parser.add_argument('-j', '--workers', default=4, type=int, metavar='N',
                    help='number of data loading workers (default: 4)')
parser.add_argument('--epochs', default=90, type=int, metavar='N',
                    help='number of total epochs to run')
parser.add_argument('--start-epoch', default=0, type=int, metavar='N',
                    help='manual epoch number (useful on restarts)')
parser.add_argument('-b', '--batch-size', default=256, type=int,
                    metavar='N',
                    help='mini-batch size (default: 256), this is the total '
                         'batch size of all GPUs on the current node when '
                         'using Data Parallel or Distributed Data Parallel')

parser.add_argument('-p', '--print-freq', default=10, type=int,
                    metavar='N', help='print frequency (default: 10)')
```

```
parser.add_argument('--resume', default='', type=str, metavar='PATH',
                    help='path to latest checkpoint (default: none)')
parser.add_argument('-e', '--evaluate', dest='evaluate',
↪   action='store_true',
                    help='evaluate model on validation set')
parser.add_argument('--pretrained', dest='pretrained',
↪   action='store_true',
                    help='use pre-trained model')
parser.add_argument('--world-size', default=-1, type=int,
                    help='number of nodes for distributed training')
parser.add_argument('--rank', default=-1, type=int,
                    help='node rank for distributed training')
parser.add_argument('--dist-url', default='tcp://224.66.41.62:23456',
↪   type=str,
                    help='url used to set up distributed training')
parser.add_argument('--dist-backend', default='nccl', type=str,
                    help='distributed backend')
parser.add_argument('--seed', default=None, type=int,
                    help='seed for initializing training. ')
parser.add_argument('--gpu', default=None, type=int,
                    help='GPU id to use.')
parser.add_argument('--multiprocessing-distributed', action='store_true',
                    help='Use multi-processing distributed training to
                    ↪   launch '
                         'N processes per node, which has N GPUs. This is
                         ↪   the '
                         'fastest way to use PyTorch for either single
                         ↪   node or '
                         'multi node data parallel training')

best_acc1 = 0


def set_all_rng_seed(seed: int):
    random.seed(seed)
    np.random.seed(seed)

```

```python
    # see PyTorch Notes
    # https://pytorch.org/docs/stable/notes/randomness.html
    torch.backends.cudnn.deterministic = True
    torch.backends.cudnn.benchmark = False
    torch.manual_seed(seed)


def get_all_rng_states():
    r = {
        'pytorch': torch.get_rng_state(),
        'pytorch_cuda': torch.cuda.get_rng_state_all(),
        'numpy': np.random.get_state(),
        'python': random.getstate()
    }
    return r

def set_all_rng_states(state: dict):
    random.setstate(state['python'])
    np.random.set_state(state['numpy'])
    torch.set_rng_state(state['pytorch'])
    if 'pytorch_cuda' in state:
        torch.cuda.set_rng_state_all(state['pytorch_cuda'])


def main():
    args = parser.parse_args()

    if args.seed is not None:
        set_all_rng_seed(args.seed)
        warnings.warn('You have chosen to seed training. '
                      'This will turn on the CUDNN deterministic setting, '
                      'which can slow down your training considerably! '
                      'You may see unexpected behavior when restarting '
                      'from checkpoints.')

    if args.gpu is not None:
```

```
        warnings.warn('You have chosen a specific GPU. This will completely '
                      'disable data parallelism.')

    if args.dist_url == "env://" and args.world_size == -1:
        args.world_size = int(os.environ["WORLD_SIZE"])

    args.distributed = args.world_size > 1 or args.multiprocessing_distributed

    ngpus_per_node = torch.cuda.device_count()
    if args.multiprocessing_distributed:
        # Since we have ngpus_per_node processes per node, the total world_size
        # needs to be adjusted accordingly
        args.world_size = ngpus_per_node * args.world_size
        # Use torch.multiprocessing.spawn to launch distributed processes: the
        # main_worker process function
        mp.spawn(main_worker, nprocs=ngpus_per_node, args=(ngpus_per_node, args))
    else:
        # Simply call main_worker function
        main_worker(args.gpu, ngpus_per_node, args)


def main_worker(gpu, ngpus_per_node, args):
    global best_acc1
    args.gpu = gpu

    if args.gpu is not None:
        print("Use GPU: {} for training".format(args.gpu))

    if args.distributed:
        if args.dist_url == "env://" and args.rank == -1:
            args.rank = int(os.environ["RANK"])
        if args.multiprocessing_distributed:
```

```python
            # For multiprocessing distributed training, rank needs to be the
            # global rank among all the processes
            args.rank = args.rank * ngpus_per_node + gpu
        dist.init_process_group(backend=args.dist_backend, init_method=args.dist_url,
                                world_size=args.world_size, rank=args.rank)
    # create model
    if args.pretrained:
        print("=> using pre-trained model '{}'".format(args.arch))
        model = models.__dict__[args.arch](pretrained=True)
    else:
        print("=> creating model '{}'".format(args.arch))
        model = models.__dict__[args.arch]()

    if args.distributed:
        # For multiprocessing distributed, DistributedDataParallel constructor
        # should always set the single device scope, otherwise,
        # DistributedDataParallel will use all available devices.
        if args.gpu is not None:
            torch.cuda.set_device(args.gpu)
            model.cuda(args.gpu)
            # When using a single GPU per process and per
            # DistributedDataParallel, we need to divide the batch size
            # ourselves based on the total number of GPUs we have
            args.batch_size = int(args.batch_size / ngpus_per_node)
            args.workers = int(args.workers / ngpus_per_node)
            model = torch.nn.parallel.DistributedDataParallel(model, device_ids=[args.gpu])
        else:
            model.cuda()
            # DistributedDataParallel will divide and allocate batch_size to all
            # available GPUs if device_ids are not set
            model = torch.nn.parallel.DistributedDataParallel(model)
```

```
    elif args.gpu is not None:
        if args.gpu != -1:
            torch.cuda.set_device(args.gpu)
            model = model.cuda(args.gpu)
    else:
        # DataParallel will divide and allocate batch_size to all
        ↪  available GPUs
        if args.arch.startswith('alexnet') or
        ↪  args.arch.startswith('vgg'):
            model.features = torch.nn.DataParallel(model.features)
            model.cuda()
        else:
            model = torch.nn.DataParallel(model).cuda()

    # define loss function (criterion) and optimizer
    criterion = nn.CrossEntropyLoss().cuda(args.gpu)

    optimizer, scheduler = optimizers.__dict__[args.optimizer](model)

    # optionally resume from a checkpoint
    if args.resume:
        if os.path.isfile(args.resume):
            print("=> loading checkpoint '{}'".format(args.resume))
            checkpoint = torch.load(args.resume)
            args.start_epoch = checkpoint['epoch']
            best_acc1 = checkpoint['best_acc1']
            if args.gpu is not None and args.gpu != -1:
                # best_acc1 may be from a checkpoint from a different
                ↪  GPU
                best_acc1 = best_acc1.to(args.gpu)
            model.load_state_dict(checkpoint['state_dict'])
            optimizer.load_state_dict(checkpoint['optimizer'])
            scheduler.load_state_dict(checkpoint['scheduler'])
            set_all_rng_seed(args.seed)
            set_all_rng_states(checkpoint['rng_state'])
            print("=> loaded checkpoint '{}' (epoch {})"
                  .format(args.resume, checkpoint['epoch']))
```

```
        else:
            print("=> no checkpoint found at '{}'".format(args.resume))

    cudnn.benchmark = True

    # Data loading code
    train_dataset, val_dataset = datasets.__dict__[args.dataset]()

    if args.distributed:
        train_sampler =
         ↪  torch.utils.data.distributed.DistributedSampler(train_dataset)
    else:
        train_sampler = None

    train_loader = torch.utils.data.DataLoader(
        train_dataset, batch_size=args.batch_size, shuffle=(train_sampler
         ↪  is None),
        num_workers=args.workers, pin_memory=True, sampler=train_sampler)

    val_loader = torch.utils.data.DataLoader(
        val_dataset,
        batch_size=args.batch_size, shuffle=False,
        num_workers=args.workers, pin_memory=True)

    if args.evaluate:
        validate(val_loader, model, criterion, args)
        return

    for epoch in range(args.start_epoch, args.epochs):
        if args.distributed:
            train_sampler.set_epoch(epoch)

        # train for one epoch
        train(train_loader, model, criterion, optimizer, epoch, args)

        # evaluate on validation set
        acc1 = validate(val_loader, model, criterion, args)
```

```

        scheduler.step()

        # remember best acc@1 and save checkpoint
        is_best = acc1 > best_acc1
        best_acc1 = max(acc1, best_acc1)

        if not args.multiprocessing_distributed or
        ↪  (args.multiprocessing_distributed
                and args.rank % ngpus_per_node == 0):
            save_checkpoint({
                'epoch': epoch + 1,
                'arch': args.arch,
                'state_dict': model.state_dict(),
                'best_acc1': best_acc1,
                'optimizer': optimizer.state_dict(),
                'scheduler': scheduler.state_dict(),
                'rng_state': get_all_rng_states()
            }, is_best, f'checkpoint.pth')


def train(train_loader, model, criterion, optimizer, epoch, args):
    batch_time = AverageMeter('Time', ':6.3f')
    data_time = AverageMeter('Data', ':6.3f')
    losses = AverageMeter('Loss', ':.4e')
    top1 = AverageMeter('Acc@1', ':6.2f')
    top5 = AverageMeter('Acc@5', ':6.2f')
    progress = ProgressMeter(len(train_loader), batch_time, data_time,
    ↪  losses, top1,
                             top5, prefix="Epoch: [{}]".format(epoch))

    # switch to train mode
    model.train()

    end = time.time()
    for i, (input, target) in enumerate(train_loader):
        # measure data loading time
```

```
        data_time.update(time.time() - end)

        if args.gpu is not None and args.gpu != -1:
            input = input.cuda(args.gpu, non_blocking=True)
        if not args.gpu == -1:
            target = target.cuda(args.gpu, non_blocking=True)

        # compute output
        output = model(input)
        loss = criterion(output, target)

        # measure accuracy and record loss
        acc1, acc5 = accuracy(output, target, topk=(1, 5))
        losses.update(loss.item(), input.size(0))
        top1.update(acc1[0], input.size(0))
        top5.update(acc5[0], input.size(0))

        # compute gradient and do SGD step
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

        # measure elapsed time
        batch_time.update(time.time() - end)
        end = time.time()

        if i % args.print_freq == 0:
            progress.print(i)


def validate(val_loader, model, criterion, args):
    batch_time = AverageMeter('Time', ':6.3f')
    losses = AverageMeter('Loss', ':.4e')
    top1 = AverageMeter('Acc@1', ':6.2f')
    top5 = AverageMeter('Acc@5', ':6.2f')
    progress = ProgressMeter(len(val_loader), batch_time, losses, top1,
    ↪  top5,
```

```
                                 prefix='Test: ')

    # switch to evaluate mode
    model.eval()

    with torch.no_grad():
        end = time.time()
        for i, (input, target) in enumerate(val_loader):
            if args.gpu is not None and args.gpu != -1:
                input = input.cuda(args.gpu, non_blocking=True)
            if args.gpu != -1:
                target = target.cuda(args.gpu, non_blocking=True)

            # compute output
            output = model(input)
            loss = criterion(output, target)

            # measure accuracy and record loss
            acc1, acc5 = accuracy(output, target, topk=(1, 5))
            losses.update(loss.item(), input.size(0))
            top1.update(acc1[0], input.size(0))
            top5.update(acc5[0], input.size(0))

            # measure elapsed time
            batch_time.update(time.time() - end)
            end = time.time()

            if i % args.print_freq == 0:
                progress.print(i)

        # TODO: this should also be done with the ProgressMeter
        print(' * Acc@1 {top1.avg:.3f} Acc@5 {top5.avg:.3f}'
              .format(top1=top1, top5=top5))

    return top1.avg
```

```python
def save_checkpoint(state, is_best, filename='checkpoint.pth.tar'):
    torch.save(state, filename)
    if is_best:
        shutil.copyfile(filename, 'model_best.pth')


class AverageMeter(object):
    """Computes and stores the average and current value"""
    def __init__(self, name, fmt=':f'):
        self.name = name
        self.fmt = fmt
        self.reset()

    def reset(self):
        self.val = 0
        self.avg = 0
        self.sum = 0
        self.count = 0

    def update(self, val, n=1):
        self.val = val
        self.sum += val * n
        self.count += n
        self.avg = self.sum / self.count

    def __str__(self):
        fmtstr = '{name} {val' + self.fmt + '} ({avg' + self.fmt + '})'
        return fmtstr.format(**self.__dict__)


class ProgressMeter(object):
    def __init__(self, num_batches, *meters, prefix=""):
        self.batch_fmtstr = self._get_batch_fmtstr(num_batches)
        self.meters = meters
        self.prefix = prefix

    def print(self, batch):
```

```python
        entries = [self.prefix + self.batch_fmtstr.format(batch)]
        entries += [str(meter) for meter in self.meters]
        print('\t'.join(entries))

    def _get_batch_fmtstr(self, num_batches):
        num_digits = len(str(num_batches // 1))
        fmt = '{:' + str(num_digits) + 'd}'
        return '[' + fmt + '/' + fmt.format(num_batches) + ']'


def accuracy(output, target, topk=(1,)):
    """Computes the accuracy over the k top predictions for the specified
    ↪ values of k"""
    with torch.no_grad():
        maxk = max(topk)
        batch_size = target.size(0)

        _, pred = output.topk(maxk, 1, True, True)
        pred = pred.t()
        correct = pred.eq(target.view(1, -1).expand_as(pred))

        res = []
        for k in topk:
            correct_k = correct[:k].view(-1).float().sum(0, keepdim=True)
            res.append(correct_k.mul_(100.0 / batch_size))
        return res


if __name__ == '__main__':
    main()
```

Appendix B

List of third-party source code referenced and used

- The PyTorch Framework and the `torchvision` package, including example codes, at `https://pytorch.org`

- Fixed Hadamard classifier [10], at `https://github.com/eladhoffer/fix_your_classifier`

- Classification on CIFAR-10/100 and ImageNet with PyTorch at `https://github.com/bearpaw/pytorch-classification`

Bibliography

[1] Jose M Alvarez and Mathieu Salzmann. Learning the number of neurons in deep networks. In *NeurIPS*, pages 2270–2278, 2016.

[2] Nitin Bansal, Xiaohan Chen, and Zhangyang Wang. Can we gain more from orthogonality regularizations in training deep networks? In *NeurIPS*, pages 4261–4271, 2018.

[3] Aditya Chattopadhay, Anirban Sarkar, Prantik Howlader, and Vineeth N Balasubramanian. Grad-cam++: Generalized gradient-based visual explanations for deep convolutional networks. In *WACV*, pages 839–847. IEEE, 2018.

[4] François Chollet. Xception: Deep learning with depthwise separable convolutions. In *CVPR*, pages 1251–1258, 2017.

[5] Matthieu Courbariaux, Itay Hubara, Daniel Soudry, Ran El-Yaniv, and Yoshua Bengio. Binarized neural networks: Training deep neural networks with weights and activations constrained to+ 1 or-1. *arXiv preprint arXiv:1602.02830*, 2016.

[6] Alexey Dosovitskiy and Thomas Brox. Inverting visual representations with convolutional networks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 4829–4837, 2016.

[7] Song Han, Jeff Pool, John Tran, and William Dally. Learning both weights and connections for efficient neural network. In *NeurIPS*, pages 1135–1143, 2015.

[8] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. In *Proceedings of the 2015 IEEE International Conference on Computer Vision (ICCV)*, ICCV '15, page 1026–1034, USA, 2015. IEEE Computer Society.

[9] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *CVPR*, 2016.

[10] Elad Hoffer, Itay Hubara, and Daniel Soudry. Fix your classifier: the marginal value of training the last weight layer. In *ICLR*, 2018.

[11] Hengyuan Hu, Rui Peng, Yu-Wing Tai, and Chi-Keung Tang. Network trimming: A data-driven neuron pruning approach towards efficient deep architectures. *arXiv preprint arXiv:1607.03250*, 2016.

[12] Gao Huang, Zhuang Liu, Laurens Van Der Maaten, and Kilian Q Weinberger. Densely connected convolutional networks. In *CVPR*, pages 4700–4708, 2017.

[13] Forrest N Iandola, Song Han, Matthew W Moskewicz, Khalid Ashraf, William J Dally, and Kurt Keutzer. Squeezenet: Alexnet-level accuracy with 50x fewer parameters and¡ 0.5 mb model size. *arXiv preprint arXiv:1602.07360*, 2016.

[14] Benoit Jacob, Skirmantas Kligys, Bo Chen, Menglong Zhu, Matthew Tang, Andrew Howard, Hartwig Adam, and Dmitry Kalenichenko. Quantization and training of neural networks for efficient integer-arithmetic-only inference. In *CVPR*, pages 2704–2713, 2018.

[15] Minje Kim and Paris Smaragdis. Bitwise neural networks. *arXiv preprint arXiv:1601.06071*, 2016.

[16] Jonathan Krause, Michael Stark, Jia Deng, and Li Fei-Fei. 3d object representations for fine-grained categorization. In *4th International IEEE Workshop on 3D Representation and Recognition (3dRR-13)*, Sydney, Australia, 2013.

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

[18] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In *NeurIPS*, pages 1097–1105, 2012.

[19] Yann LeCun, Léon Bottou, Yoshua Bengio, Patrick Haffner, et al. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

[20] Yann LeCun, John S Denker, and Sara A Solla. Optimal brain damage. In *NeurIPS*, pages 598–605, 1990.

[21] Hao Li, Asim Kadav, Igor Durdanovic, Hanan Samet, and Hans Peter Graf. Pruning filters for efficient convnets. In *ICLR*, 2017.

[22] Min Lin, Qiang Chen, and Shuicheng Yan. Network in network. *arXiv preprint arXiv:1312.4400*, 2013.

[23] Y. Lin, F. Lv, S. Zhu, M. Yang, T. Cour, K. Yu, L. Cao, and T. Huang. Large-scale image classification: Fast feature extraction and svm training. In *CVPR*, pages 1689–1696, 2011.

[24] Christos Louizos, Max Welling, and Diederik P Kingma. Learning sparse neural networks through l_0 regularization. In *ICLR*, 2018.

[25] Ningning Ma, Xiangyu Zhang, Hai-Tao Zheng, and Jian Sun. Shufflenet v2: Practical guidelines for efficient cnn architecture design. In *ECCV*, pages 116–131, 2018.

[26] Aravindh Mahendran and Andrea Vedaldi. Understanding deep image representations by inverting them. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 5188–5196, 2015.

[27] M-E. Nilsback and A. Zisserman. Automated flower classification over a large number of classes. In *Proceedings of the Indian Conference on Computer Vision, Graphics and Image Processing*, Dec 2008.

[28] Maxime Oquab, Léon Bottou, Ivan Laptev, and Josef Sivic. Is object localization for free?-weakly-supervised learning with convolutional neural networks. In *CVPR*, pages 685–694, 2015.

[29] Federico Pernici, Matteo Bruni, Claudio Baecchi, and Alberto Del Bimbo. Fix your features: Stationary and maximally discriminative embeddings using regular polytope (fixed classifier) networks. *arXiv preprint arXiv:1902.10441*, 2019.

[30] Florent Perronnin, Yan Liu, Jorge Sánchez, and Herve Poirier. Large-scale image retrieval with compressed fisher vectors. pages 3384–3391, 06 2010.

[31] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, Alexander C. Berg, and Li Fei-Fei. ImageNet Large Scale Visual Recognition Challenge. *IJCV*, 115(3):211–252, 2015.

[32] Mark Sandler, Andrew Howard, Menglong Zhu, Andrey Zhmoginov, and Liang-Chieh Chen. Mobilenetv2: Inverted residuals and linear bottlenecks. In *CVPR*, pages 4510–4520, 2018.

[33] Ramprasaath R Selvaraju, Michael Cogswell, Abhishek Das, Ramakrishna Vedantam, Devi Parikh, and Dhruv Batra. Grad-cam: Visual explanations from deep networks via gradient-based localization. In *ICCV*, pages 618–626, 2017.

[34] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. 2014.

[35] Jost Tobias Springenberg, Alexey Dosovitskiy, Thomas Brox, and Martin Riedmiller. Striving for simplicity: The all convolutional net. *arXiv preprint arXiv:1412.6806*, 2014.

[36] Suraj Srinivas and R Venkatesh Babu. Data-free parameter pruning for deep neural networks. *arXiv preprint arXiv:1507.06149*, 2015.

[37] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. Going deeper with convolutions. In *CVPR*, pages 1–9, 2015.

[38] Christian Szegedy, Vincent Vanhoucke, Sergey Ioffe, Jon Shlens, and Zbigniew Wojna. Rethinking the inception architecture for computer vision. In *CVPR*, pages 2818–2826, 2016.

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

[40] Bolei Zhou, Aditya Khosla, Agata Lapedriza, Aude Oliva, and Antonio Torralba. Object detectors emerge in deep scene cnns. *arXiv preprint arXiv:1412.6856*, 2014.

[41] Bolei Zhou, Aditya Khosla, Agata Lapedriza, Aude Oliva, and Antonio Torralba. Learning deep features for discriminative localization. In *CVPR*, pages 2921–2929, 2016.