Class Conditional Filter Pruning Based on Clustering of The Ranks of Feature Maps

by

Soroush Razavi

A thesis submitted in partial fulfillment of the requirements for the degree of

Master of Science

in

Software Engineering and Intelligent Systems

Department of Electrical and Computer Engineering

University of Alberta

Abstract

The success of deep learning is partly due to the sheer size of modern models. However, such large models strain the capabilities of mobile or resource-constrained devices. Ergo, reducing the resource demands of AI models is essential before AI can be deployed on such devices. One promising solution to this challenge is filter pruning, which aims to streamline models without sacrificing performance. We propose a new approach to filter pruning that extends feature-map ranking to consider the class-by-class rank of each map. These feature vectors are clustered, and automated decision rules select the filters to be pruned. This makes our work one of the few pruning methods that is fully automated and does not require any human labor from end to end. Experiments using VGG and ResNet networks on the CIFAR-10 and ImageNet datasets show that our pruned models are more accurate than the well-known HRank algorithm and perform similarly on the CIFAR-100 dataset. In addition, we see significant reductions in power usage of models post-pruning. Finally, ethical pruning requires that an algorithm does not favor some group of data over others. We have verified that our algorithm follows the ethical pruning approach.

To honour the ones
who walk on the edge of the light to follow their cause.

And to Gaia.

"Not All Those Who Wander Are Lost."

– J. R. R. Tolkien

Acknowledgements

Behind every thesis lies a surprisingly long list of people who deserve a standing ovation. First and foremost, a huge THANK YOU to my brilliant supervisors, Dr. Scott Dick and Dr. Xingyu Li, whose wisdom was only matched by their patience. Not all heroes wear capes—some provide GPU clusters and schedule weekly meetings to guide you through the academic jungle. This project was funded by the Social Sciences and Humanities Research Council under the New Frontiers Research Fund program, and Discovery grants from Dr. Dick and Dr. Li, for which I am incredibly grateful.

Next, a special shout-out to Mahsa Layeghi, whose brilliance and support outshone even the most challenging days, keeping me energized through countless late nights and debugging marathons. Studying abroad is a thrilling adventure, but it can initially create a vacuum, leaving behind friends and family. My lovely family back home always had my back, ensuring I never felt alone on this journey. To my friends here, thank you for being the closest thing to family I could have. Your adherence to the first two rules of Thesis Club—never talking about my thesis—was much appreciated.

A big thanks to the University of Alberta Swimming Club for keeping me afloat, active, and balanced. Thanks to the friendly folks at Amii, who always provided a warm welcome whenever I visited the co-working space for some focused work. Finally, to my supervisory committee, thank you for pushing the boundaries of thought even further.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Recently, there has been an emerging interest in embedding visual AI models within intelligent applications. Examples include augmented or virtual reality [3], [113], the new collaboration between Meta and Ray Ban [45], [76], Google Lens for cameras [24], or Apple's similar Live Text and Visual Look Up [109]. However, training these deep networks, which may be hundreds of layers deep with billions of parameters, demands enormous resources. A cloud server or a large pool of high-end computers is often used to train a state-of-the-art Deep Neural Network. Due to the differences between the hardware capabilities of a cloud server and an edge or mobile device, deploying deep networks in low-power devices is thus challenging [17], [59], [62], [75], [107]. It is not only the training process that causes this challenge; performing inference with a CNN model on an edge or mobile device also requires a considerable number (millions or billions) of floating-point computations. Mobile devices, however, have limited battery capacity, and their computing hardware is power-optimized with limited memory. To make a deep neural network perform well on such devices, we need to substantially reduce the network's size and the total number of Floating-Point Operations (FLOPs) it performs without losing much accuracy. Another critical factor is ensuring the network responds quickly, as the response time is an essential characteristic for users [98].

In response to the above, multiple approaches have been proposed to reduce the number of network parameters and Floating-Point Operations. Network pruning is among the most popular approaches to compressing a network [28].

Network pruning can be divided into two categories: 1) Weight pruning [8], [27], [28], [56], [111], where weights inside a network are pruned individually; and 2) Filter pruning [35], [36], [59], [61]–[63], [66], [115], where candidate filters are eliminated from the network entirely.

Weight pruning techniques produce a sparse network with a minimal drop in prediction performance [27], [28]. However, the location of the remaining weights in the networks is unknown a priori. This is challenging because modern software/hardware stacks rely on non-sparsity in accelerating computations. Such sparse networks would thus be inefficient except on specialized hardware or software [26], [81].

On the other hand, filter pruning approaches, also known as channel pruning or coarse-grained pruning, remove entire filters, and thus the pruned networks remain non-sparse. Hence, they still run efficiently without specialized software or hardware. In this thesis, our focus aligns with the field's consensus on improving filter pruning by having three steps [62], [107]: 1) Training a deep network to convergence; 2) Ranking the filters in the network and pruning selected filters; and 3) Fine-tuning the network after pruning.

Filter ranking is the most vital step in filter pruning. HRank [62] proposed that filters with higher average numerical rank in the feature maps they generate contribute more to the final accuracy of the network. Following this proposal, filters are sorted by rank, and those below a chosen threshold are pruned. The specific threshold varies; in many recent proposals, the thresholds are set individually for each layer, based on manual trial-and-error exploration [17], [59], [62], [107].

In contrast with [17], [59], [62], [107], our model employs an end-to-end approach without manual trial-and-error determination of a pruning threshold. Instead, it uses clustering to identify *groups* of filters that offer the best combination of network pruning and retained accuracy. We show that filters within a layer contribute differentially to accuracy in different classes for CIFAR-10, CIFAR-100, and ImageNet [49], [89]. Expanding on the HRank[62] approach, we propose that filters should be ranked by their aggregated performance on individual classes in the dataset, with filters that perform well across all classes having a more significant impact than those that only perform well on some classes. To evaluate our proposal, we test our pruning method on the VGGNet [95] and ResNet56 [33] architectures applied to CIFAR-10 and CIFAR-100 [49], and ResNet50 on ImageNet [89]. We use the same pre-trained model for each experiment to compare our method against HRank [62]. We have specifically selected HRank as the baseline to compare our method against since it remains one of the best methods for convolutional neural network pruning [22], [34], and most of the current pruning algorithms are also compared to HRank [18], [79], [106].

HRank [62] is the most similar pruning method to our algorithm. The main differences between them are filter selection and how we measure the importance of filters across all classes. We use cluster validity metrics and predetermined criteria to select the filters to be removed. At the same time, HRank determines a pruning rate for each layer and selects filters in order of rank to meet that pruning rate. Thus, to fairly compare our algorithm against HRank, we extract the layer-wise compression rates from our automatic method and set HRank to compress each layer at the same rate. Our proposed algorithm offers superior accuracy compared to the same-sized HRank-pruned network after fine-tuning, indicating that we have selected a superior *group* of filters to keep. Our results showed that we have a higher accuracy for CIFAR-10 with 87. 799% and 87. 67% on VGGNet and ResNet-50, respectively, compared to HRank's 87.527% and 86.878%. In CIFAR-100 experiments, the accuracies for each algorithm were a statistical tie. For the ImageNet experiments, we achieved 60. 13% 83. 51% in the top-1 and top-5 accuracies, respectively, while HRank achieved 59.61% 83.18% (note that these are single runs due to the size of ImageNet).

Combining class conditional methods [79][105] with a clustering algorithm for filter pruning [16][117] is one aspect of the novelty of our method. It is worth noting that we also contribute to a fully automated pruning pipeline. Manually setting the compression rate for pruning is tedious and requires extensive trial-and-error exploration. We have designed a fully automated pipeline where there is no need to set a pruning rate and create an arbitrary number as the threshold. However, one can argue that any form of pruning is trimming "something" from the network and that "something" can be interpreted as a threshold. We will see that there is always an 'implicit' threshold in the examples below, even in methods that do not explicitly define one.

- Iterative Pruning: One can iteratively prune the least important parts of a model [79] [99]. For example, we can repeatedly prune the smallest magnitude weight in neural networks without a pre-defined threshold. This is repeated until some criterion condition is met (like a drop in validation accuracy). Here, one does not explicitly assign a pruning rate, but one prunes some parts of the network based on the criterion.
- Random Pruning: This is less conventional, but one could theoretically prune parts of a model randomly and observe the effects on performance [60]. While not necessarily efficient, it's a close example of pruning without predetermined and explicit thresholds. However, one can argue that after the random values are generated, we still have a threshold for which the pruned network's performance is acceptable.
- Explicit pruning: Similar to HRank, some models specifically select a number (like 70% of filters or weights) and prune that much, or select a specific volume of FLOPs they want to prune and prune that much specifically. This is common in cases where one knows exactly the hardware that the model will be implemented on, and prune can be specialized for that hardware.
- Cluster Based Pruning: Another approach toward pruning is the use of clustering algorithms to select a group of filters to prune, as in [16][117]. Our approach follows this strategy and automates the pruning pipeline rather than requiring manual trial and error. The choice to include or exclude a filter from this group implicitly defines a threshold.

As discussed, CRank falls into the last category, where the algorithm itself automatically sets an implicit threshold. Our method only requires trial and error in determining an optimal number of clusters to divide the filters. This process is also automated, as optimality is determined by maximizing the Silhouette score [87] on all clusterings, which is completed in minutes in one batch. To the best of our knowledge, this is the only work that does filter pruning without manually assigning some form of threshold and is fully automated.

The remainder of this thesis is organized as follows. Essential background and related work is reviewed in Section II. In Section III, we discuss mathematical preliminaries and formulate the pruning problem. Chapter IV presents our proposed method. We present our experimental results, discuss our evaluation methodology in Section V, and close with a summary and discussion of future work in Section VI.

Chapter 2

Background and Literature Review

2.1 Artificial Neural Networks

2.1.1 Early Works

In 1943, McCulloch and Pitts [73] introduced a formal neuron model. The scientific community considers This model a pioneering work in neural networks. Their work laid the foundation for representing the activity of any neuron as a logical statement and how physiological relations among neural activities correspond to relations among propositions. This work paved the way for other scientists interested in describing nets' behavior in logical terms. Six years later, in 1949, Donald Hebb proposed the Hebbian learning paradigm based on observations of biological neural networks. [37]. In his work, he found that information is stored in the weights of synapses, and learning occurs through modifying these synapses. This learning rule is known as Hebbian learning or Hebb's rule. It explains synaptic plasticity, where neurons adapt during the learning process. The theory is often summarized as "Cells that fire together wire together [92]" emphasizing the temporal precedence of firing.

The Rosenblatt perceptron [85] is a linear binary classifier proposed by Frank Rosenblatt in 1958. The perceptron is a feed-forward neural network that maps an input to an output using a weighted sum and an activation function [19]. The perceptron weights are adjusted during training to minimize the error between the output of the perceptron and the desired output. The

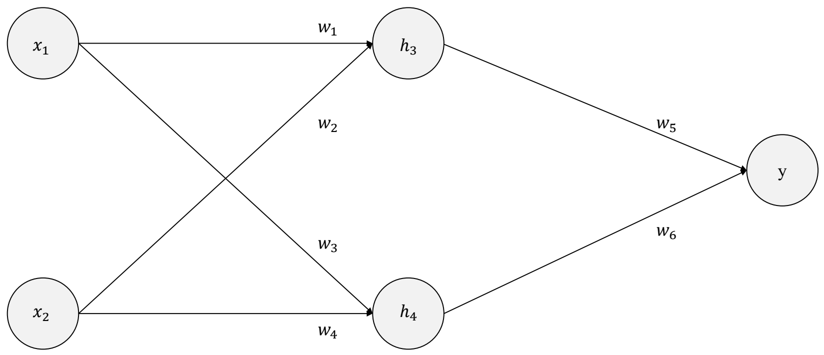

Figure 2.1: Depiction of a simple Multi-Layer Perceptron

perceptron consists of one or more input nodes, a single output node, and a set of weights. The input nodes receive input values, which are multiplied by a weight. The weighted values are then summed and passed through an activation function. The output of the activation function is the output of the perceptron. The Rosenblatt perceptron has some limitations, such as its inability to solve problems that are not linearly separable [94]. However, the perceptron has paved the way for the development of more complex neural networks, such as multi-layer perceptron and convolutional neural networks, which can solve more complicated problems.

2.1.2 Multi-Layer Perceptron

To address the limitations of Rosenblatt's perceptron, Multi-Layer Perceptron was introduced. MLP also follows a feed-forward method that uses backpropagation to reduce its error function. An MLP has at least three distinct layers, namely the Input layer, one or more hidden layer(s), and the output layer. Each layer can have an activation function. This is where a non-linear (or piece-wise linear) function can be employed to make MLP capable of tackling problems that need non-linearity. A simple MLP is shown in Figure 2.1:

The output of each layer is computed as a weighted sum of the inputs from the previous layer, followed by an activation function. The output of the MLP is the output of its last layer. To train an MLP, we need to define a loss function that measures how well the MLP predicts the desired output for a given input. A common choice is the mean squared error (MSE) loss, which is defined as in Equation (2.1):

$$L(\mathbf{y}, \hat{\mathbf{y}}) = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2 \tag{2.1}$$

Here, $\mathbf{y}$ represents the vector of actual outputs, $\hat{\mathbf{y}}$ denotes the vector of predicted outputs, and n stands for the number of samples.

The loss function is minimized using an optimization algorithm such as gradient descent, which adjusts the MLP weights by updating them against the gradient of the loss function concerning the weights. This gradient is computed utilizing the calculus chain rule, enabling error propagation from the output layer back to the input layer in a process known as backpropagation.

Forward Pass

To illustrate backpropagation, we first need to understand how the forward pass works in an MLP. Consider a simple MLP with a single hidden layer and an output layer. The hidden layer contains h neurons, and the output layer contains o. The input layer comprises d features. The following Equations represent the MLP:

$$\mathbf{z}_1 = \mathbf{W}_1\mathbf{x} + \mathbf{b}_1 \tag{2.2}$$

$$\mathbf{a}_1 = f_1(\mathbf{z}_1) \tag{2.3}$$

$$\mathbf{z}_2 = \mathbf{W}_2\mathbf{a}_1 + \mathbf{b}_2 \tag{2.4}$$

$$\hat{\mathbf{y}} = f_2(\mathbf{z}_2) \tag{2.5}$$

Equation (2.2) is for the first (hidden) layer of the MLP. $\mathbf{x}$ is the input vector, $\mathbf{W}_1$ is the weight matrix for the first layer, and $\mathbf{b}_1$ is the bias vector for the first layer. The output of this operation, $\mathbf{z}_1$, is known as the pre-activation output. Each neuron in the hidden layer computes a weighted sum of its inputs plus a bias term.

After calculating the pre-activation output $\mathbf{z}_1$, an activation function $f_1(\cdot)$ such as sigmoid, tanh, ReLU is applied element-wise to $\mathbf{z}_1$, as is shown in Equation (2.3). This yields the post-activation output $\mathbf{a}_1$, which is the final output of the first layer. The activation function introduces non-linearity into the model, which allows the MLP to model complex, non-linear relationships.

Equation (2.4) shows how the post-activation output from the first layer, $\mathbf{a}_1$, serves as the input to the second (output) layer. Similar to the first layer, each neuron in the second layer computes a weighted sum of its inputs plus a bias term. Here, $\mathbf{W}_2$ is the weight matrix for the second layer, and $\mathbf{b}_2$ is the bias vector for the second layer.

The final step in the forward pass is shown in Equation (2.5), which involves applying an activation function $f_2(\cdot)$ to the pre-activation output of the second layer. This results in the predicted output $\hat{\mathbf{y}}$. The activation function used in the output layer depends on the specific task at hand. For binary classification, a sigmoid activation function could be used to get a probability output; for multi-class classification, a softmax function could be used to get a probability distribution over the classes; for a regression task, no activation function (or an identity function) could be used to get a real-valued output.

Computing the Gradient

To compute the gradient of the loss function with respect to each weight and bias, we need to apply the chain rule repeatedly. For example, to compute $\frac{\partial L}{\partial w_{2ij}}$, where w_{2ij} is an element of $\mathbf{W}_2$, we have:

$$\frac{\partial L}{\partial w_{2ij}} = \frac{\partial L}{\partial \hat{\mathbf{y}}} \frac{\partial \hat{\mathbf{y}}}{\partial \mathbf{z}_2} \frac{\partial \mathbf{z}_2}{\partial w_{2ij}} \tag{2.6}$$

Where Equation (2.6) is the application of the chain rule. Here we are finding the derivative of the loss function L with respect to a weight w_{2ij} in $\mathbf{W}_2$. This is done by computing the product of the derivative of the loss function with respect to the predicted output $\hat{\mathbf{y}}$, the derivative of the predicted output with respect to its pre-activation output $\mathbf{z}_2$, and the derivative of the pre-activation output with respect to the weight.

$$\frac{\partial L}{\partial \hat{\mathbf{y}}} = -\frac{2}{n}(\mathbf{y} - \hat{\mathbf{y}}) \tag{2.7}$$

$$\frac{\partial \hat{\mathbf{y}}}{\partial \mathbf{z}_2} = \hat{\mathbf{y}}(1 - \hat{\mathbf{y}}) \tag{2.8}$$

$$\frac{\partial \mathbf{z}_2}{\partial w_{2ij}} = a_{1j} \tag{2.9}$$

$$\frac{\partial L}{\partial w_{2ij}} = -\frac{2}{n}(\mathbf{y} - \hat{\mathbf{y}})\hat{\mathbf{y}}(1 - \hat{\mathbf{y}})a_{1j} \tag{2.10}$$

In the given Equations, eq (2.7) calculates the derivative of the Mean Squared Error (MSE) loss function with respect to the predicted output. Equation (2.8) is the derivative of the predicted output with respect to its pre-activation value, and Equation (2.9) is the derivative of the pre-activation output of the second layer with respect to a weight. This is just the post-activation output of the first layer corresponding to that weight, since $\mathbf{z}_2 = \mathbf{W}_2\mathbf{a}_1 + \mathbf{b}_2$. The derivative of the weighted sum with respect to a specific weight is just the corresponding input. Lastly, Equation (2.10) is the final computed derivative of the loss function with respect to a particular weight obtained by substituting the previous expressions into the first Equation.

The gradient of the loss function with respect to the other weights and biases can be computed similarly using the chain rule. For instance, to calculate $\frac{\partial L}{\partial w_{1ij}}$, where w_{1ij} is an element of $\mathbf{W}_1$, we have:

$$\frac{\partial L}{\partial w_{1ij}} = \frac{\partial L}{\partial \hat{\mathbf{y}}} \frac{\partial \hat{\mathbf{y}}}{\partial \mathbf{z}_2} \frac{\partial \mathbf{z}_2}{\partial \mathbf{a}_1} \frac{\partial \mathbf{a}_1}{\partial \mathbf{z}_1} \frac{\partial \mathbf{z}_1}{\partial w_{1ij}} \tag{2.11}$$

$$\frac{\partial \mathbf{z}_2}{\partial \mathbf{a}_1} = \mathbf{W}_2^T \tag{2.12}$$

$$\frac{\partial \mathbf{a}_1}{\partial \mathbf{z}_1} = \begin{cases} 1 & \text{if } z_{1i} > 0 \\ 0 & \text{otherwise} \end{cases} \tag{2.13}$$

$$\frac{\partial \mathbf{z}_1}{\partial w_{1ij}} = x_j \tag{2.14}$$

$$\frac{\partial L}{\partial w_{1ij}} = -\frac{2}{n}(\mathbf{y} - \hat{\mathbf{y}})\hat{\mathbf{y}}(1 - \hat{\mathbf{y}})(\mathbf{W}_2^T)f_1'(\mathbf{z}_1)x_j \tag{2.15}$$

Once we have determined the gradients of the loss function with respect to all weights and biases, we can update them using gradient descent as follows:

$$w_{kij} = w_{kij} - \eta\frac{\partial L}{\partial w_{kij}} \tag{2.16}$$

$$b_{ki} = b_{ki} - \eta\frac{\partial L}{\partial b_{ki}} \tag{2.17}$$

In these update Equations, η is the learning rate, a hyperparameter that determines how much we adjust the weights in the gradient direction for each step.w_{kij} and b_{ki} represent the elements of the weight matrices and bias vectors, respectively.

The training process of a Multi-Layer Perceptron (MLP) consists of the following steps:

1. Initialize the weights and biases with small random values.
2. For each training example, perform the following:
   (a) Compute the output of the MLP.
   (b) Compute the loss.
   (c) Compute the gradient of the loss at each neuron with respect to the weights and biases.
   (d) Update the weights and biases for each neuron using its loss gradient.
3. Repeat step 2 until a stopping criterion is met. Some examples of this stopping criterion can be a certain number of iterations or until the loss falls below a certain threshold or does not improve significantly anymore.

This whole process is typically repeated over multiple passes through the training dataset, also known as epochs.

It should be noted that the specifics of this process, such as the activation functions used in the hidden and output layers, the number of layers, the number of neurons in each layer, the loss function, and the learning rate, are hyperparameters that vary depending on the specific problem and the dataset at hand.

Moreover, it's also worth mentioning that while the basic principles behind the training of MLPs have been known for several decades, the interest in them has been revitalized in recent years thanks to the advent of deep learning. In a deep learning context, MLPs (or, more generally, artificial neural networks) can have many layers, and sophisticated techniques are used to train them, including advanced optimization methods, regularization techniques to prevent overfitting, and specialized initialization methods, among others. The next section of this thesis will focus on analyzing Convolutional Network (CNN), which has roots in the MLP but uses different linear algebra techniques compared to MLP.

2.2 Convolutional Neural Networks

2.2.1 Brief History of CNNs

A Convolutional Neural Network (CNN) is a type of artificial neural network designed to mimic the way the human brain processes visual data [52]. CNNs are most commonly used in tasks such as image classification, object detection, image recognition, and video analysis. For instance, they are widely employed in facial recognition technologies, medical image analysis, self-driving cars, and even in art and design for style transfer [50].

The early investigations into the mechanics of vision by neuroscientists David Hubel and Torsten Wiesel [43] in the 1960s laid foundational insights for computer vision. They conducted experiments on a cat's visual cortex, where the cat's head was immobilized in a rig, and its visual cortex was analyzed using electrodes while various light patterns were displayed, as is shown

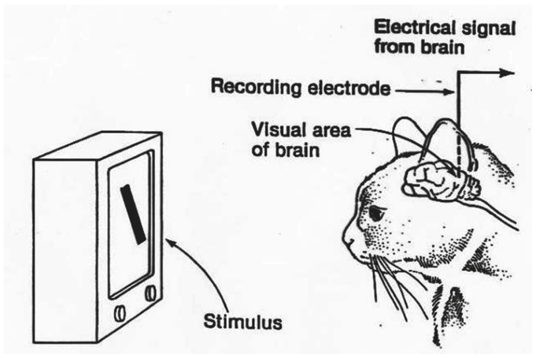

Figure 2.2: Hubel and Wiesel stimulated a cat's visual cortex and saw the response of the brain to those stimulations - image from [10], original source unknown

in Figure 2.2. Their research identified that specific neurons in the visual cortex were responsive to particular light patterns, known as receptive fields, at specific locations and sizes. They discovered that the visual cortex consists of layered retinotopic maps sensitive to simple visual elements like spots, bars, and edges, varying by position, size, and orientation. Further investigations revealed that these receptive fields could be likened to local filters, processing spatial frequency patterns across different bands and orientations. As they probed deeper into the visual cortex, Hubel and Wiesel observed that these basic patterns combined with each other to form more intricate patterns such as corners and crosses, termed "complex" receptive fields [10].

This discovery by Hubel and Wiesel brought them a Nobel prize and encouraged the researchers to expand and explore the idea further by mathematically modeling the receptive field and using convolution with Finite Impulse Response digital filters on them, including Gaussian derivatives and Gabor Functions [15], [86], [118].

The Neocognitron, introduced by Kunihiko Fukushima in 1980, is one of the notable early works on CNNs [21]. It is a hierarchical, multilayered neural network designed for pattern recognition, particularly for robust recognition against variations in position, size, and orientation of patterns. The network comprises multiple layers, including input, convolutional, and pooling (subsampling) layers, leading to a hierarchical feature extraction mechanism. Earlier layers detect simple features like edges in this structure, while deeper

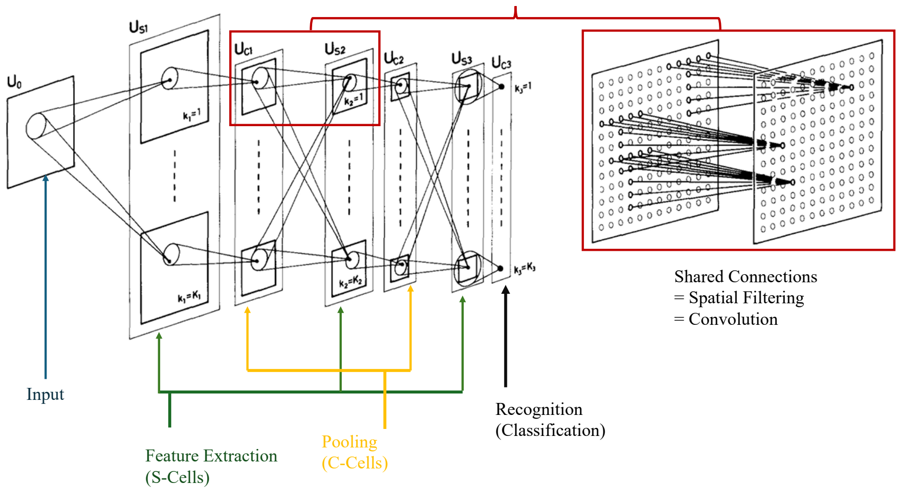

Figure 2.3: Neocognitron Architecture consisting of U_{S_l} and U_{C_l} layers, alongside a deep zoom on the convolution operation between layers - recreated image from visualizations in [21].

layers identify more complex patterns. A critical aspect of the Neocognitron is its use of trainable filters or kernels in the convolutional layers, which are adjusted during the learning process to optimize pattern recognition. This learning mechanism allows the network to automatically and adaptively learn spatial hierarchies of features from input images. While the original paper did not extensively use backpropagation as modern CNNs do, the principles of gradient descent and learning feature representations form the basis of its operation [21].

The Neocognitron's architecture (shown in Figure 2.3) draws inspiration from vertebrates' visual systems, comprising alternating S-cell layers for extracting features and C-cells for handling positional variations. S-cells come from simple cells or lower-order hypercomplex cells, and C-cells represent complex cells or higher-order hypercomplex cells. This setup ensures that features identified at initial levels progressively merge into more comprehensive representations [21], [51]. The Equation below describes the output of an S-cell:

$$u_{s_l}(n,k) = r_l \cdot \phi\left(\frac{\sigma_l + \sum_{k=1}^{K_{c_l-1}} \sum_{v\in A_l} a_l(v,k,K)\cdot u_{c_l-1}(n+v,k)}{\sigma_l + \frac{r_l}{1+r_l}\cdot b(k)\cdot uv_l(n)} - 1\right) \quad (2.18)$$

where

$$\phi[x] = \begin{cases} x & \text{if } x \geq 0 \\ 0 & \text{if } x < 0 \end{cases} \tag{2.19}$$

In Equation 2.18, the notation $u_{s_l}(n, k)$ is the calculation of an exemplary cell such as a S-cell in the l_{th} stage. In the same Equation, n is a 2D indication of the center of the receptive field of the cell's position from the input layer. Here, k being the serial number of the cell-plane is in the range of $1 \leqq k \leqq K_{S_l}$ if it is for S-cell or $1 \leqq k \leqq K_{C_l}$ if it is C-cell. The parameter σ_l is a non-negative constant that shows the saturation level of the input-to-output in the S-cell. $a_l(v, k, K)$ shows the strength of the firing power of the connection coming from $u_{c_l-1}(n+v, k)$ in the exact previous layer. Also, in 2.18, v shows the spatial spread in the input connections from the previous cell to one S-cell, and A_l is the summation of the range of v. Another interesting note about Neocognitron is the activation function. From Equation 2.19 we can understand that this design is also using ReLu [1] which can be also written as Equation 2.20:

$$\phi(x) = \max(0, x) \tag{2.20}$$

It is the first instance where it was used in neural networks as an activation function [11], [96]. So the Neocognitron, which was initially developed for recognizing handwritten Japanese characters, eventually laid the groundwork for subsequent developments in convolutional neural networks (CNNs), influencing models like LeNet-5 by Yann LeCun et al.[53], which further refined and applied these concepts to practical applications such as handwritten digit recognition.

2.2.2 CNN Architecture

Yann LeCun's seminal work on Convolutional Neural Networks (CNNs), particularly the LeNet architecture (now considered LeNet-1) [55], revolutionized the field of digital image processing. Year by year, he and his AT&T team improved their model until three years later, in 1998, they proposed the LeNet-5

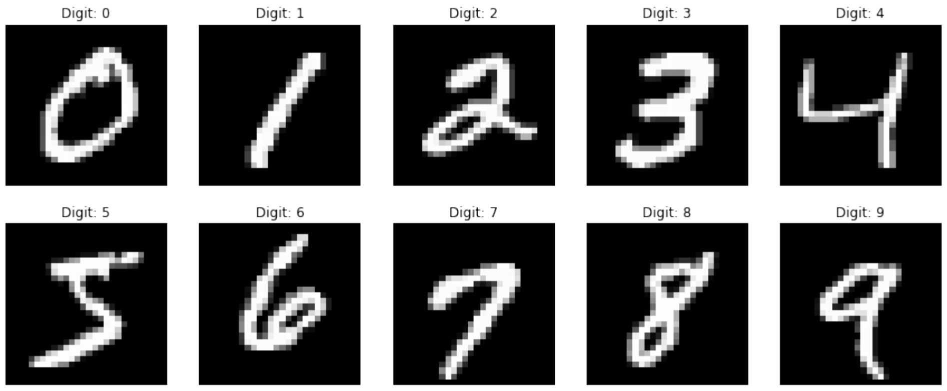

Figure 2.4: Visualization of the MNIST Digits dataset. A sample from every class of data is shown

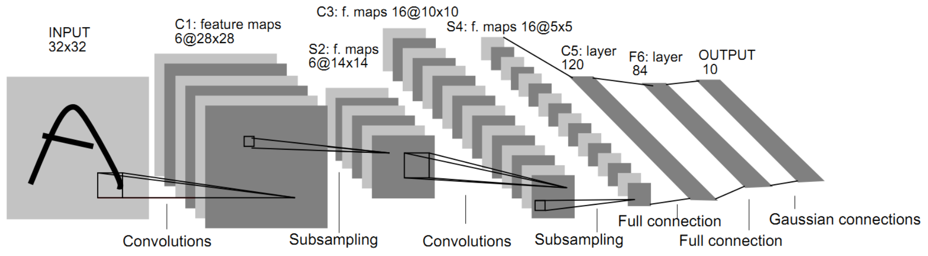

Figure 2.5: Architecture of LeNet-5 showing feature maps and subsampling and convolutional layers - image from [6], a recreation of [53]

[53] architecture that employs convolutional layers to extract hierarchical features from images automatically, had pooling layers and dense layers and utilized non-linear activation functions. The LeNet-5 laid the groundwork for all modern CNN architectures. Similar to Fukushima's work, LeNet was designed explicitly for recognizing handwritten digits in images on a dataset introduced in the same paper called Modified NIST or MNIST for short [53]. A sample of digits in MNIST is shown in Figure 2.4. This dataset consists of 60000 training images and 10000 test images of handwritten digits of 0 to 9. Each image in the dataset is a 28 by 28 pixel, 8-bit grayscale image 2.4).

The LeNet-5 [53] architecture included convolutional layers, subsampling layers, which are now commonly referred to as pooling layers, and fully connected layers, all using various activation functions. This combination of layers has become a standard blueprint for many modern CNNs. Figure 2.5 from the original paper shows the architecture design of LeNet-5.

Next, we consider each of these types of layers.

Convolutional Layer

Convolutional layers use learnable kernels or filters to capture spatial hierarchies in image data by recognizing patterns and features (like edges, textures, and more complex structures in deeper layers). This was a significant advancement over previous neural networks that did not emphasize spatial hierarchies. This task is achieved by employing a discrete convolution operation. The convolution layers in LeNet-5 used a discrete convolution operation [10]:

$$(f * P)(x, y) = \sum_{v=1}^{N} \sum_{u=1}^{N} f(u, v) P(x - u, y - v) \tag{2.21}$$

This operation shows the $2D$ convolution of filter $f(x, y)$, which is $N \times N$ across an image $P(x, y)$ with horizontal axis x and vertical axis y. At each position, it multiplies the filter's weights, $f(u, v)$, by the corresponding image pixels, adds up these products, and places the sum at the position (x, y) in the output image. Image positions outside the valid range are considered zero.

In LeNet-5, there are some adjustments specific to neural networks so that the convolution operation becomes more suitable for image processing in neural networks. Equation 2.21 is adjusted to the following:

$$a(i, j) = f\left(\sum_{u,v}^{N} w(u, v) P(i - u, j - v) + b\right) \tag{2.22}$$

Equation 2.22 shows how a convolutional network operates on a series of 2-D overlapping windows. $p(i, j)$ represents a 2D input layer, $w(u, v)$ is an $N \times N$ receptive field that has been learned, b is a learned bias term, $f(.)$ refers to a nonlinear activation function, and $a(i, j)$ denotes the output layer that results from this process. Equation 2.22 shows the convolution operation for a single neuron, but it can be scaled to benefit from multiple neurons in parallel. This will produce a mapping of features with regard to each pixel, where the number of features is defined by the depth of the feature maps. For example, LeNet-1 had a depth of 4 [10]. The Equation 2.23 shows the convolution operation for a network with D parallel receptive fields:

$$a_d(i,j) = f\left(\sum_{u,v}^{N} w_d(u,v)p(i-u,j-v) + b_d\right) \tag{2.23}$$

A convolutional network that uses this design benefits from the parallel processing of each window using D receptive fields, producing a vector of D feature values $\vec{a}(i,j)$ for each position (i,j) in the image. Equation 2.23 extends the concept from Equation 2.22 by replacing a single learned receptive field, $w(i,j)$, with a vector of D learned receptive fields, $w_d(i,j)$, which results in a vector of D output layers, $a_d(i,j)$. The output of this layer will be a series of feature maps.

Subsampling Operation

Subsampling in CNNs, also known as pooling, reduces the emphasis on precise feature positioning in convolutional layers. CNNs rely on fixed-size kernels or filters to detect image features, but exact feature positioning isn't necessary for classification tasks. By using a pooling operation in a pooling layer, we will force the network not to memorize feature positions so that CNNs become more robust to small translations in the input image if they occur within the receptive field's window size. When pooling regions are contiguous and only features from the same hidden units are pooled, the network achieves translation invariance. This means the same pooled feature remains active even if the image undergoes minor translations.[102]. In addition, pooling reduces spatial resolution by downsampling feature maps. This downsampling effectively increases the diameter of the receptive field, allowing the network to consider larger-scale features in later layers. In a subsampling layer, where each unit has a fixed receptive field, operations are performed within these fields to generate new values for the output. This process helps reduce the dimensionality of the feature maps while retaining important information. There are two main types of pooling: Max pooling and Average Pooling [23].

Max pooling involves substituting a window of features measuring N × N with the highest value present within that window. Figure 2.6 shows the max pooling operation applied to a 4 x 4 matrix. Look at the 2 × 2 green

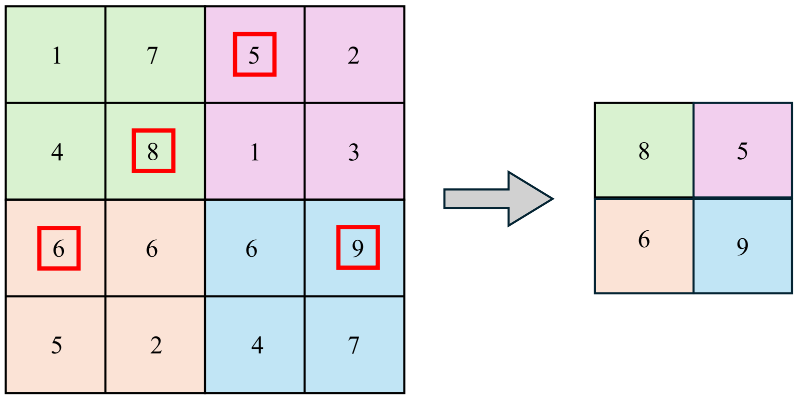

Figure 2.6: Max Pooling Operation

square situated in the top-left corner, which is replaced with the maximum value among the four values contained within the square, which is 8.

In LeNet-5, however, the pooling used was Average Pooling. Average pooling replaces the window of features with the average value rather than the maximum. Max pooling is often preferred over average pooling in CNNs and has shown slightly better performance [10]. Max pooling helps capture the most distinctive features within a region, which can be crucial for effective feature extraction. By retaining only the maximum values, max pooling focuses on preserving the most significant information, enhancing the model's ability to detect relevant patterns. You can see average pooling in action in Figure 2.7.

Fully Connected Layer

Fully connected or dense layers usually come at the end of the network architecture. These layers are responsible for synthesizing the features extracted by earlier convolutional and pooling layers together. By integrating these features, fully connected layers facilitate decision-making processes such as classification or regression. They achieve this by mapping the features onto an output space, where each neuron can represent a class or a value. The inclusion of non-linear activation functions within these layers also allows the network to learn complex and non-linear relationships between the features. At the end

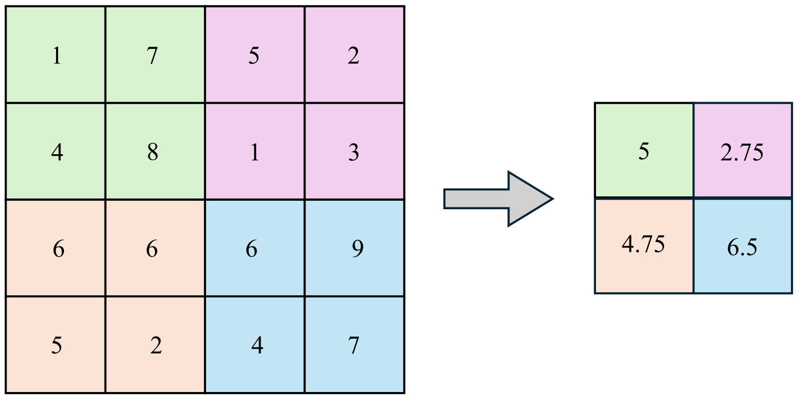

Figure 2.7: Average Pooling Operation

of LeNet-5, we have two fully connected layers. These layers, equipped with a softmax activation function, output the probabilities of the input belonging to each class, highlighting the critical role of fully connected layers in transitioning from feature extraction to classification [23], [53]. In practice, this layer is a normal feed-forward layer where all neurons of one layer are connected to all neurons in the next and previous layers, and the output of each neuron is calculated by Equations 2.2 to 2.5.

Activation Function

We have already introduced activation functions and have gone over why they are fundamental components in deep learning, particularly in the architecture of Convolutional Neural Networks. They help determine the output of neural network nodes and introduce non-linear properties to the network, which are crucial for learning complex patterns in data. Here, we go over some of the most common activation functions. First, the Logistic activation [14] is given by 2.24:

$$\sigma(x) = \frac{1}{1+e^{-x}} \tag{2.24}$$

The ability of the logistic function to compress an infinite input range into a bounded interval of [0, 1] was highly beneficial for early neural models, which often needed to predict probabilities or binary outcomes. The ease of calculating the gradient of Logistic Sigmoid with respect to x is the main reason that it was so desirable since it makes the backpropagation much easier. Its derived form can be written as Equation 2.25:

$$\sigma'(x) = \sigma(x)(1 - \sigma(x)) \tag{2.25}$$

Next, we have the tanh() or Hyperbolic Tangent function, which was used as an activation function in the convolutional layers of LeNet-5. Tanh is one of the hyperbolic functions that have existed since the 1760s [7]. It was introduced by Vincenzo Riccati and Johann Heinrich Lambert and it centers outputs around zero, which can aid in convergence during neural network training. The tanh() function is given by 2.26:

$$\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}} \tag{2.26}$$

Softmax is one of the most common activation functions today, especially for the output layer of a multi-class classification network. It helps to interpret the outputs as probabilities for each class. The softmax function is given as:

$$\text{Softmax}(z_i) = \frac{e^{z_i}}{\sum_j e^{z_j}} \tag{2.27}$$

Last but not least, we have the Rectified Linear Unit "ReLU" [80], which is one of the most popular activation functions today. ReLU was first introduced in the Neocognitron but was largely ignored in favor of sigmoids like the Logisitic or tanh() functions for many years. However, the sheer size of deep neural networks made efficient operations vital, and the simple ReLU is compiled directly to hardware operations, while sigmoids must be software-simulated in most computer architectures. The ReLU activation is given by Equations 2.19 or 2.20, or to keep it consistent with the other functions introduced in this section as:

$$\text{ReLU}(x) = max(0, x) \tag{2.28}$$

ReLU is a linear function for positive values, but it introduces non-linearity through its behavior at zero, where it suddenly transitions to outputting zero. This non-linearity allows complex functions to be modeled by neural networks using ReLU. Also, ReLU helps to mitigate the vanishing gradient problem, which is common with sigmoid or tanh functions. This problem occurs when gradients are passed through many layers during backpropagation and get progressively smaller, effectively halting the network from learning. Since the gradient of ReLU for positive inputs is 1, it does not diminish the gradient during backpropagation, allowing deeper networks to be trained more effectively. However, the problem of vanishing gradient still remains for the negative values in ReLU [14], so another variation of this function was proposed. Leaky Rectified Linear Unit [70] or "LReLU" is an extension to ReLU, with the idea to address the aforementioned vanishing gradient issue. It's range is from $(-\infty, \infty)$ and its formula is:

$$\text{Leaky ReLU}(x) = \begin{cases} x & \text{if } x > 0 \\ \alpha x & \text{if } x \leq 0 \end{cases} \tag{2.29}$$

In Equation 2.29 α is a small constant, typically $\alpha = 0.01$, and it's there so that the activation function can utilize the negative values, even though it reduces their effect by that coefficient. Figure 2.8 shows the plot of these activation functions.

Why Move Toward CNNs

Convolutional Neural Networks (CNNs) and regular Neural Networks (like MLPs) share similarities: both consist of neurons with learnable weights and biases, use a differentiable score function from input to output, and employ a loss function on the last fully connected layer. However, as an example, an image of size 256×256 pixels with RGB channels, when flattened into a vector, results in an input layer with 196,608 neurons. If the next layer has a similar number of neurons, the weight matrix becomes impractically large, leading to intense computational demands and memory requirements.

Also, MLPs treat input data as a flattened vector of features, so the network would lose the spatial hierarchy or local structuring of the data. In the context

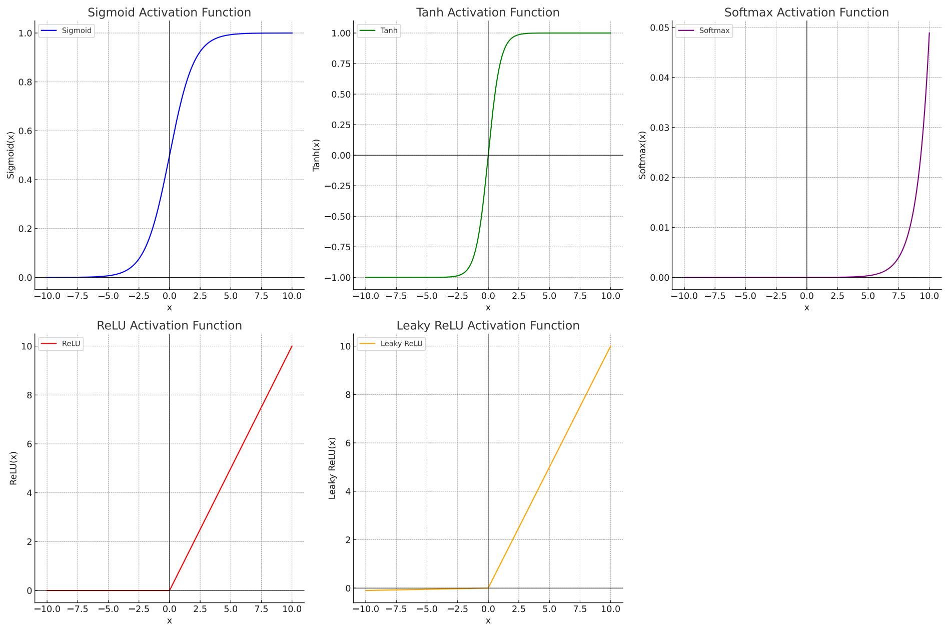

Figure 2.8: Plot of Five Popular Activation Functions

of images, this would mean that MLPs will fail to recognize local patterns such as edges, textures, or shapes. These concepts are crucial for understanding images, which adds to the limitations of MLPs. Compared to MLPs, CNNs are specifically helpful when it comes to image inputs, which leads to their superior performance and scalability compared to regular networks [23], [57].

On the other hand, CNNs have filters arranged in 3D volumes (width, height, depth). Each filter consists of multiple smaller matrices called kernels, which hold the network weights. Filters in a layer are only connected to a small region of the previous layer, called the receptive field, reducing the number of connections and parameters. The section below will help you understand these concepts in greater detail.

2.2.3 Visualization of Filters Inside a CNN

A convolutional layer in a CNN takes in the input feature maps denoted by X_i and produces the output feature maps X_{i+1}. The input feature map X_i is a three-dimensional tensor with dimensions n_i (number of input channels) $\times h_i$ (height of input feature maps) $\times w_i$ (width of input feature maps). The transformed output feature maps X_{i+1} also have three dimensions of

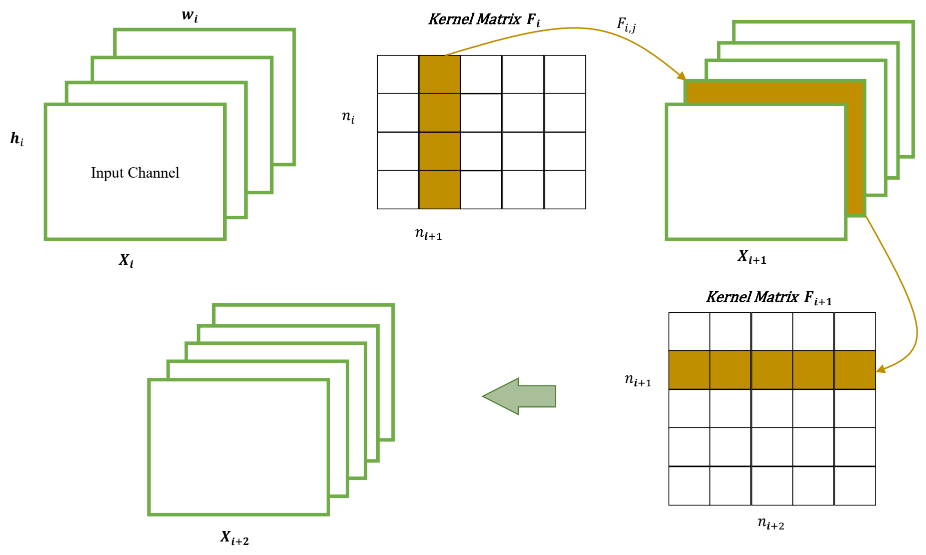

Figure 2.9: Visualization of Convolution Operation and Explaining Filters, Kernels and Feature Maps

$n_{i+1} \times h_{i+1} \times w_{i+1}$, and are the input feature maps in the next layer. This transformation is done by applying n_{i+1} filters $F_{i,j}$ to the n_i input channels. $F_{i,j}$ is a 3D filter consisting of n_i number of 2D $k \times k$ kernels where k is the kernel size. There is a $1:1$ relation between filters and output feature maps; each filter produces one feature map. Combining all filters together, we get the kernel matrix $F_i \in \mathbb{R}^{n_i \times\ n_{i+1} \times\ k\ \times\ k}$. Figure 2.9 visualizes the convolution operation.

As shown in Figure 2.9, kernel matrix F_i is applied on X_i producing output channel X_{i+1}. In this Figure, the second filter in F_i and its subsequent feature map of X_{i+1} are shown in gold. As you can tell from the title of this work, our focus and most of this thesis is about removing filters from a network, which is called filter pruning. In Figure 2.9 The process is repeated with the second kernel matrix F_{i+1}. No filters from F_{i+1} will be pruned, thus neither are any feature maps in X_{i+2}. However, the pruning in F_i has an ongoing impact on the total number of FLOPs as a channel was removed in X_{i+1}. The overall FLOPs reduction can be calculated as per [59].

This reduction in parameters is due to the shared weights of a convolution layer, which simplifies the calculations of the network during the convolution operation that we discussed before. In the convolution layers, the filters pass over the input image, computing dot products at every position. This process significantly reduces the number of free parameters and, at the same time, allows the network to focus on training filters that activate when they see specific types of features. Also, the same filter (weights) is used across the entire input in a convolutional layer, further reducing the model complexity and memory requirements. Finally, the subsampling helps to reduce the dimensionality of each feature map while retaining the most important information. These differences make CNNs more efficient and scalable for image-processing tasks. The architecture of CNNs eventually reduces the entire image into a single vector of class scores, making them highly effective for image classification tasks [10], [23], [57].

The effectiveness of CNNs in image-related tasks was first demonstrated by AlexNet, a CNN model that won the ImageNet Large-Scale Visual Recognition Challenge in 2012 [50]. Since then, a variety of CNN models have significantly improved the performance of image and video-related tasks. In the next section, we are going to take a look at two mainstream CNN architectures that we use extensively in this work, namely VGGNet [95] (with plain structure) and ResNet [33].

2.2.4 Modern Architectures

VGGNet

VGGNet, developed by Karen Simonyan and Andrew Zisserman of the University of Oxford's Visual Geometry Group in 2014, is a pivotal convolutional neural network architecture that underscored the significance of network depth for image recognition tasks. Its design is notable for its simplicity, employing 33 convolutional layers stacked in increasing depth, a significant departure from the shallower networks of the time, such as AlexNet with eight layers.

VGGNet, particularly in its VGG-16 and VGG-19 variants, showcased that deep networks could achieve remarkable improvements in accuracy for image recognition. The architecture requires a fixed input size of 224×224 RGB

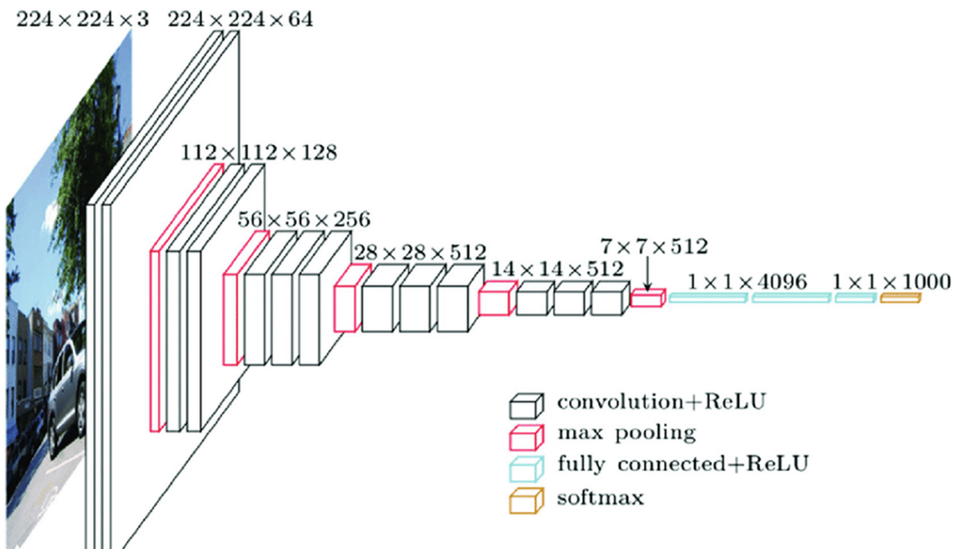

Figure 2.10: VGG-16 Original Architecture - image from [6]

images. It incorporates several key features, including the use of 33 filters with stride and pad of 1, max pooling over a 2×2 pixel window with stride 2, and three fully connected layers at the end, where the first two have a depth of 4096 each. The third is a classification layer whose number of channels depends on the dataset. The terms stride, padding, and depth are common hyperparameters found in CNNS. The depth (or channel) refers to the number of D feature maps a layer produces or receives, as in Figure 2.9. For example, if we have a colored image, it has three channels at layer 0 (the input image): one for the red channel, one for green, and one for blue. Similarly, a grayscale image only has one channel with values between 0 and 255. Stride refers to the step size S between window positions during the convolution operation. The stride is set to 1 by default, meaning the filter moves one pixel at a time. However, for larger windows, it is possible to define larger step sizes, which results in a more compressed output feature map. Zero-padding involves adding a border of zeros around the feature map to preserve the input size, typically denoted as N. This technique ensures that the spatial dimensions of the output feature map remain consistent with the input size, preventing significant reduction after convolution. Both padding and stride play pivotal roles in determining the output dimensions and the receptive fields of neurons in convolutional layers, influencing the network's performance and feature extraction capabilities [10].

The design of VGG-16 and VGG-19 are very similar. The primary distinction between VGG-16 and VGG-19 lies in the number of convolutional layers within each of their blocks. Specifically, VGG-16 is constructed with two convolutional layers in each block, whereas VGG-19 includes three convolutional layers per block. Figures 2.11 and 2.10 show the VGG-16 architecture in detail, which mainly focuses on the size of filters, but Figure 2.10 provides more detail about the size of the feature maps and input and output dimensions. Using the Rectified Linear Unit (ReLU) activation function throughout the network further contributed to its effectiveness [10].

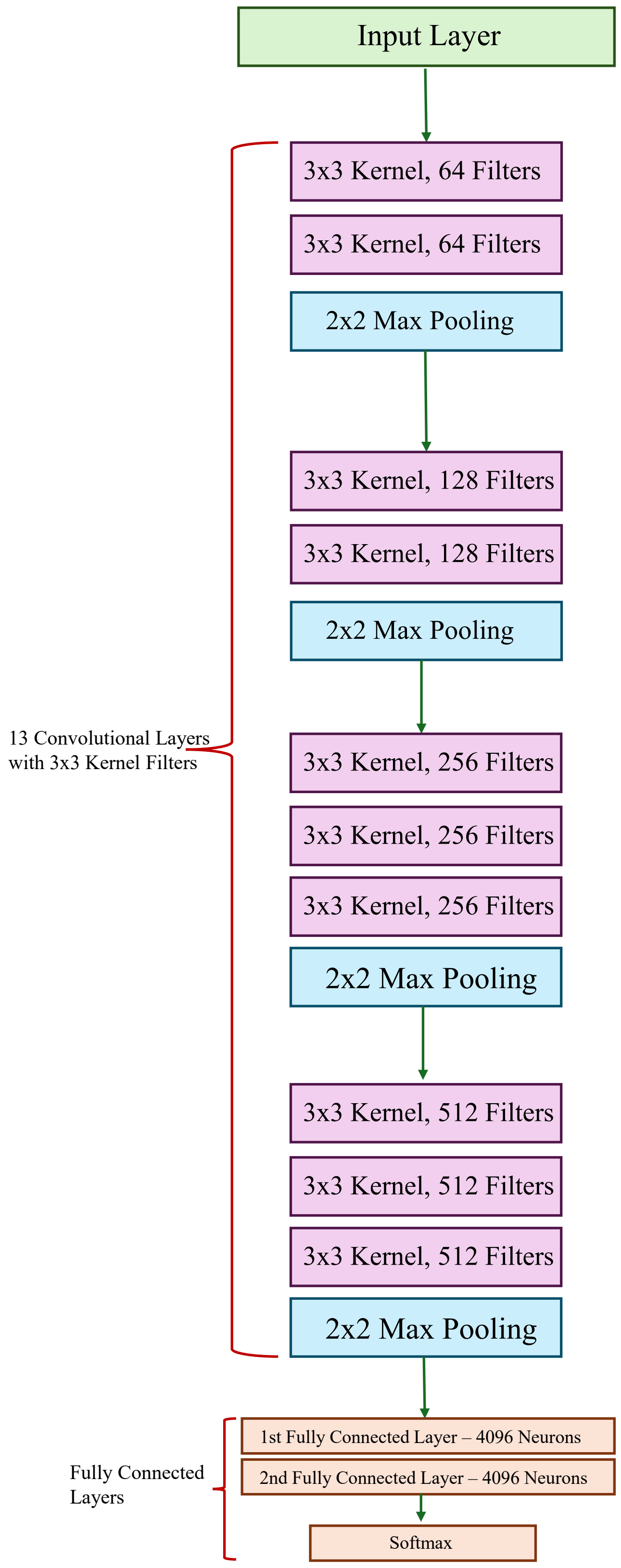

Figure 2.11: VGG-16 Architecture

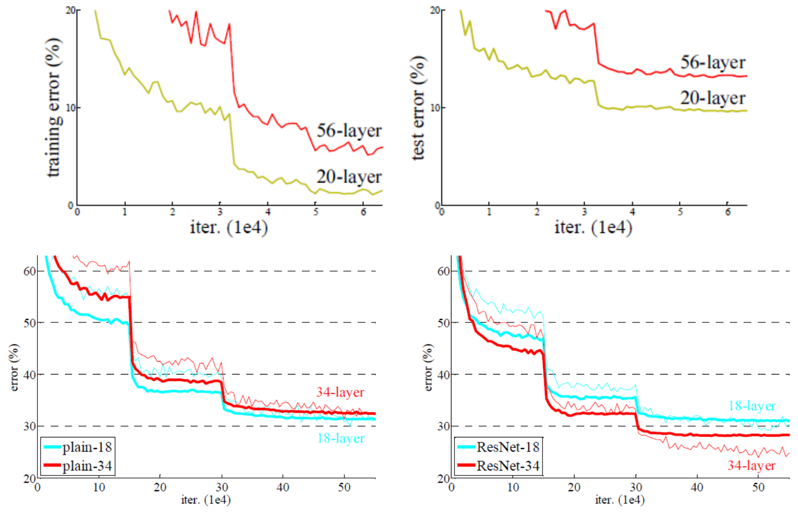

Figure 2.12: The plots on the top compare two models with 56 layers and 20 layers trained on CIFAR-10. The top left shows the training error, and the top right shows the test error. Plots on the bottom show models trained using ImageNet, with the bottom left one showing plain neural networks with 18 and 34 layers. The bottom right image shows ResNet models of the same size. The thin lines show training errors and the bold ones show validation errors. As can be seen, the ResNet model has a lower error. - Image from [33]

ResNet

ResNet, introduced by Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun in their seminal 2015 paper "Deep Residual Learning for Image Recognition [33]" represents a significant advancement in the architecture of convolutional neural networks (CNNs). Since AlexNet, the deep learning community has been moving toward deeper layers, as they have been providing better results. However, [33] showed that a deeper model with 56 layers trained on CIFAR-10 performs worse than a model with only 20 layers. A similar experiment was observed with models trained on the ImageNet dataset, with 34 and 18 layers. Image 2.12 shows those comparisons.

The observed problem was not due to overfitting; the 56-layer model had worse performance in both training and test error. In contrast, if it was performing well on training and poor on the test set, it could have been considered as overfitting. So, the conclusion was that adding more layers can degrade the performance. Also, the problem was not caused by vanishing/exploding gradi-

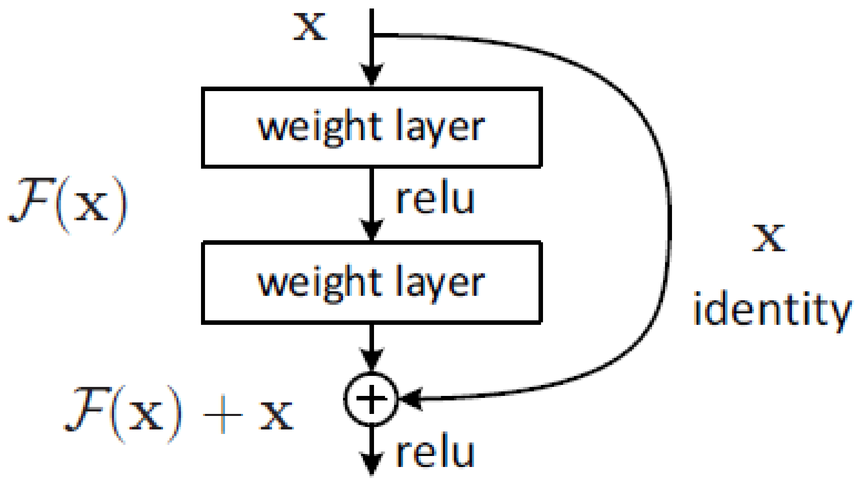

Figure 2.13: Skip Connection Inside ResNet - Image from [33]

ents since batch normalization and some other techniques were used to prevent that. In order to solve the degrading issue, in [33], researchers proposed a skip connection, which is a shortcut from input to output, as is shown in Figure 2.13. After a few weight layers, a skip/shortcut connection adds the input x to the output.

These elements allow the network to learn identity functions when necessary, ensuring that deeper layers can perform at least as well as shallower ones. To achieve this, the later layer in a deep neural network is stimulated to learn the identity function so the output of that layer is equal to its input, preventing them from degradation. In this case, if a layer is not helpful to the overall performance of the model, the regularizations set in place will skip that layer. By doing so, we are sending the network toward learning the residual mappings [33], [47], [101]. By that, we mean the network assuming the layer output is 0 or, as in Figure 2.13, $F(x) = 0$. Here, the assumption is that the output of the initial learned layer is x, and our desired underlying mapping is $H(x)$. So, the learning of residual mapping will become:

$$F(x) = H(x) - x \tag{2.30}$$

Using this architecture, a layer's output is made up of two connections, and if we want to get the gradient of one layer, we can traverse there using two different pathways, one the normal way and the other the skip connection.

$$\frac{\partial L}{\partial x} = \frac{\partial L}{\partial H} \times \frac{\partial H}{\partial x} = \frac{\partial L}{\partial H} \times \left(\frac{\partial F}{\partial x} + 1\right) = \frac{\partial L}{\partial H} \times \frac{\partial F}{\partial x} + \frac{\partial L}{\partial H} \tag{2.31}$$

Equation 2.31 shows the calculation of the gradient of the loss for $H(x) = F(x) + x$ with respect to the input x. As can be seen in Equation 2.31, if the normal connection's gradient vanishes to zero, we would still have another connection (the shortcut), which will help propagate the error.

This architecture supports the training of networks with an unprecedented depth, which allows stacking more layers on top of each other and benefitting from the skip connections. In our work, we use two versions of ResNet: ResNet-50 and a modified version called ResNet-56. The introduction of ResNet significantly impacted deep learning and computer vision; it was the state-of-the-art in image classification tasks on major datasets such as ImageNet on its introduction [50].

2.2.5 Different Approaches Toward Using CNNs

When deploying convolutional neural networks (CNNs) for tasks such as image recognition or classification, there are four primary approaches, each with its own set of benefits and considerations [58]. Training a CNN from scratch involves building and training a new network using a large dataset and significant computational resources. This method allows for customization to specific tasks but requires extensive data and computational power to train a good neural network.

Using a convolutional network as a fixed feature extractor involves taking a pre-trained CNN, removing its final classification layer, and using the rest of the network to process new images and extract features. Here, the pre-trained model will often be a ResNet or VGGNet trained on the ImageNet dataset, which has learned to identify essential features of an image. These features are then fed into a new classifier trained for the task at hand. This approach leverages the pre-trained network's ability to detect generic features that are applicable across different image recognition tasks, offering a balance between efficiency and customization without the need for extensive training data or computation.

Fine-tuning the convolutional network slightly modifies the previous approach by not only replacing the final classification layer but also adjusting the weights of the pre-trained network's layers through additional training. Our method as well as [62] use this method. In this approach, we are still using a pre-trained model, but we allow our model to modify some of the layers and weights in the network and will not use it purely as a feature extractor. This approach is a hybrid between training from scratch and using a pre-trained model.

Lastly, utilizing pre-trained models involves directly applying a pre-trained CNN to a new task with minimal or no modification. This approach is the most straightforward and requires the least amount of effort and resources, making it ideal for tasks closely related to the original training purpose of the model. It allows users to benefit from the powerful feature extraction capabilities of large, complex networks without the need to train them from scratch.

2.3 Model Compression Techniques

Model compression is a very active research area in machine learning that focuses on reducing the size and computational requirements of neural networks without significantly compromising their performance. As the deployment of deep learning models on resource-constrained devices such as mobile phones and edge devices becomes increasingly common, efficient model compression techniques are essential. There are various methods that achieve this goal, including 1) parameter quantization, 2) knowledge distillation, and 3) network pruning. Here in the background section will briefly go over the first two to give an intuition about them, however they fall out of the scope of this thesis since our work focuses on neural network pruning which we will discuss in detail the next chapter.

First, we start with parameter quantization. It is a widely used technique that involves reducing the precision of a network's parameters. Traditionally, neural networks use 32-bit floating-point numbers for weights and biases, but this precision can be excessive for many applications. By quantizing these parameters to lower bit-width representations, such as 16-bit or 8-bit integers, significant reductions in memory usage and computational power can be achieved. Quantization can be applied at various levels, such as Post-training Quantization, which involves quantizing a pre-trained model without additional retraining. For instance, [65] is an example of this approach, which introduced an efficient 8-bit quantization technique that achieves minimal accuracy loss. Then, we have Quantization-Aware Training, where the model is trained with quantization in mind, which usually leads to better performance than post-training quantization. This approach accounts for the quantization error during training; however, it requires more resources to fine-tune the model [93].

Secondly, we have Knowledge distillation, which involves training a smaller ”student” model to mimic the behavior of a larger ”teacher” model. The student model learns from the ground truth labels and the teacher model’s output or the soft targets, which contain more information about the data distribution. This technique was popularized by Geoffrey Hinton, who showed that student models can achieve performance close to teacher models while being significantly smaller [38].

Chapter 3

Preliminary & Problem Formulation

3.1 Pruning Preliminaries

Early research on neural network pruning indicated that networks with too many weights would not generalize effectively, whereas those with too few weights could not represent the data [5], [13], [52]. To maximize generalization, a balance is needed between the training error and the complexity of the network. Hence, researchers in the fields of statistical inference [4], [84], [104] and neural networks [9], [29], [78], [88] proposed jointly optimizing training error and network complexity. Among several complexity measures, Vapnik-Chervonenkis dimensionality [103] and description length[84] were two notable approaches. In Optimal Brain Damage [54], LeCun et al. used the number of non-zero free parameters as the chosen measure of complexity. Today, [54] is regarded as one of the pioneering approaches to weight pruning.

3.1.1 Weight Pruning

The basic idea behind pruning was that we can express total risk $R(W)$, with W being the parameter vector, as:

$$R(W) = \eth_{av}(W) + \lambda \eth_c(W) \tag{3.1}$$

where the $\eth_{av}(W)$ is a performance metric responsible for measuring the performance of the model, i.e., Mean Squared Error. The second variable $\eth_c(W)$

is the complexity penalty and is measured as a function of the weight vector. λ is a regularization parameter, allowing us to balance the contributions of $\eth_{av}(W)$ and $\eth_c(W)$ for a given dataset. This means that if we have, e.g., a very large λ, the model's complexity will be a determining factor for the model risk, while if λ is zero, we do not care about the complexity of the model[94].

One measure of $\eth_c(W)$ in the equation 3.1 was proposed in [39], where the complexity penalty term was defined as the squared norm of W or the weight vector:

$$\eth_c(\mathbf{w}) = ||\mathbf{w}||^2 = \sum_{i \in \eth_{total}} w_i^2 \tag{3.2}$$

In equation 3.2 $\eth_{total}$ represents all the synaptic weights of the network. The weights of the network are grouped into two categories. The first is the weights with significant influence on the network's performance, and the second is the weights that have little or no impact on performance (called excess weights) [94]. If there were no complexity regularization, these excess weights would lead to poor generalization, so the complexity regularization sets the values of the excess weights to zero to improve generalization [44].

In contrast, in [54], the authors show that the importance of a weight is not only due to its magnitude; the "saliency" of a weight also depends on its location in the network. Thus, their objective was to prune weights that had the lowest saliency while also adjusting the surviving weights to compensate for the removed one (as saliency is usually not zero). For this, the network's loss function is modeled as the first two terms of a Taylor series approximation [31]:

$$\Delta\eth_{av} = \eth(\mathbf{w} + \Delta\mathbf{w}) - \eth(\mathbf{w}) = \frac{1}{2}\Delta\mathbf{w}^T\mathbf{H}\Delta\mathbf{w} \tag{3.3}$$

The Optimal Brain Surgeon (OBS) algorithm eliminates one of the synaptic weights to reduce the incremental increase in the cost function $\Delta\eth_{av}$ as described in Equation . Let $w_i(n)$ represent this specific synaptic weight. The removal of this weight can be represented by the condition:

$$\mathbf{1}_i^T \Delta\mathbf{w} + w_i = 0 \tag{3.4}$$

Where $\mathbf{1}_i$ is a unit vector with all elements set to zero, except for the i-th element, which is set to one. The goal of OBS is to minimize the quadratic form $\frac{1}{2}\Delta\mathbf{w}^T\mathbf{H}\Delta\mathbf{w}$ with respect to the incremental change in the weight vector, $\Delta\mathbf{w}$, subject to the constraint that $\mathbf{1}_i^T\Delta\mathbf{w} + w_i = 0$, and then minimize the result with respect to the index i [94]. To solve this constrained optimization problem, we form the Lagrangian.

$$S = \frac{1}{2}\Delta\mathbf{w}^T\mathbf{H}\Delta\mathbf{w} - \lambda(\mathbf{1}_i^T\Delta\mathbf{w} + w_i) \tag{3.5}$$

where λ represents the Lagrange multiplier, $\mathbf{H}$ is the Hessian matrix and $\Delta\mathbf{w}$ is the change in weight vector . By differentiating the Lagrangian S with respect to $\Delta\mathbf{w}$, and utilizing matrix inversion, the optimal change in the weight vector $\mathbf{w}$ is determined as:

$$\Delta\mathbf{w} = -\frac{w_i}{[\mathbf{H}^{-1}]_{i,i}}\mathbf{H}^{-1}\mathbf{1}_i \tag{3.6}$$

Equation 3.6 shows how to adjust the weights in the network to minimize the cost function while eliminating the specific weight w_i. The optimal value of the Lagrangian S for the element w_i which is also the saliency of the given weight can then be expressed as:

$$S_i = \frac{w_i^2}{2[\mathbf{H}^{-1}]_{i,i}} \tag{3.7}$$

We thus select the network weight with the lowest value of S_i for pruning.

More recently, [72] identifies a subset of diverse neurons in a network and blends the remaining neurons into the selected ones. [28] applies the $\ell1$-norm to prune weights, followed by fine-tuning the network. [2] prunes a network layer-wise via convex optimization. All these methods suffer from the same sparsity problem discussed in the introduction: the sparse weight matrices are unsuitable for the acceleration techniques in modern hardware and software [26].

3.1.2 Filter Pruning

In filter pruning, the saliency of entire filters instead of individual weights is determined, and entire low-saliency filters are deleted. This leaves the network in a suitable state for modern acceleration techniques. Similar to what [28] had done in weight pruning, Li et al. [59] also use $\ell 1$-norm and sort filters based on this criterion. Then, they prune low-saliency filters and their corresponding feature maps and the kernels in the next layer associated with those feature maps (see Figure 2.9 in the previous chapter).

HRank

In HRank [62], Lin et al. use a different measure of saliency and again prune both filters and the subsequent kernels. In 2020, Lin et al. [62] empirically showed that a single filter usually generates feature maps with the same average rank, regardless of how many image batches the CNN has received. In addition, they showed that feature maps with lower average rank contain less information about the dataset and, therefore, are less impactful to the final accuracy of the network.

However, in our investigations, we found substantial redundancy among the filters that HRank preserves, and there was no similarity metric to differentiate between the filters based on the feature maps. Also, in HRank, the number of filters to be pruned in each layer is determined by manual trial-and-error, which is contrary to our goal[62]. Since HRank's approach toward pruning is closest to what we have done, we will explore it in more detail during our problem formulation in Section 3.2.

3.1.3 Class-Conditional Pruning

Class conditional pruning is a technique in neural networks that tailors the process of pruning to specific classes in a classification task [79][105]. This technique enhances model efficiency and accuracy for targeted categories. In [79], researchers proposed a solution for the problem of formalizing and quantifying the discriminative capability of filters using the total variation (TV) distance between the class-conditional distributions of the filter outputs. Given that the samples (X, u) are drawn from a distribution $\mathcal{D}$, where X represents the data point and u is the associated class label, they denote the class-conditional distribution for class α as $\mathcal{D}^\alpha$. Here, X belongs to $\mathbb{R}^N$, indicating that the input data point has N dimensions. Additionally, u is an integer class label that lies within the set $[N_{\text{classes}}] \subseteq \mathbb{N}$. Based on that and the assumption that Q_1 and Q_2 are two probability measures supported on a d-dimensional real space $\mathbb{R}^d$, [79] calculates the TV distance as:

$$\mathrm{TV}(Q_1, Q_2) = \sup_{A \subseteq \mathbb{R}^d} |Q_1(A) - Q_2(A)| \tag{3.8}$$

Given the lack of a closed-form solution for the Total Variation distance in equation 3.8, when Q_1 and Q_2 are Gaussian distributions, [79] tries the Hellinger distance as an alternative. This assumption is similar to before with Q_1 and Q_2 as two probability distributions over $\mathbb{R}^d$. However, q_1 and q_2 are defined as their respective density functions. They defined the squared Hellinger distance as [79]:

$$\mathrm{HELLD}^2(Q_1, Q_2) = \frac{1}{2} \int_{\mathbb{R}^d} \left(\sqrt{q_1(x)} - \sqrt{q_2(x)} \right)^2 dx \tag{3.9}$$

and the bounded TV Distance is given by [79]:

$$\mathrm{HELLD}^2(Q_1, Q_2) \leq \mathrm{TV}(Q_1, Q_2) \leq \sqrt{2 \cdot \mathrm{HELLD}(Q_1, Q_2)} \tag{3.10}$$

and if Q_1 and Q_2 follow Gaussian distributions, such that $Q_1 \sim \mathcal{N}(\mu_1, \sigma_1^2 I)$ and $Q_2 \sim \mathcal{N}(\mu_2, \sigma_2^2 I)$, then the squared Hellinger distance can be calculated as [79]:

$$\mathrm{HELLD}^2(Q_1, Q_2) = 1 - \left(\frac{2\sigma_1\sigma_2}{\sigma_1^2 + \sigma_2^2} \right)^{d/2} e^{-\Delta} \tag{3.11}$$

where

$$\Delta = \frac{\|\mu_1 - \mu_2\|^2}{\sigma_1^2 + \sigma_2^2} \tag{3.12}$$

3.1.4 Fair Pruning

While pruning reduces the size of the network with minimal reduction in accuracy, it is prone to amplifying the existing model biases toward underrepresented groups [40]. Such biases exist where there are class imbalances such as classifying chest X-rays [91] or reviewing resumes [12] where the intelligent systems were shown to be biased against women (due to under-representation in the training data). Pruning may damage the model's performance by not prioritizing parameters that preferentially impact the minority groups [77]. Fairness is a measure of how well a model treats different groups of samples, such as different genders, races, or ages. This "fairness" is measured via different approaches in the literature [77], [100], [110], [114], but a general consensus is a form of measuring class accuracies before and after pruning. [100] focuses on datasets D composed of n data points (x_i, a_i, y_i), with $i \in [n]$, sampled independently from an unknown distribution Π. Here, $x_i \in \mathcal{X}$ represents a feature vector, $a_i \in \mathcal{A}$ with $\mathcal{A} = [m]$ (where m is a finite number) denotes a demographic group attribute, and $y_i \in \mathcal{Y}$ signifies a class label. For example, in a face recognition task, x_i might describe an individual's headshot, a_i could indicate the person's gender or ethnicity, and y_i represents the individual's identity. The objective is to learn a function $f_\theta : \mathcal{X} \to \mathcal{Y}$, where θ is a k-dimensional real-valued parameter vector that minimizes the empirical risk:

$$\hat{\theta} = \arg\min_{\theta} J(\theta; D) = \frac{1}{n}\sum_{i=1}^{n} \ell(f_\theta(x_i), y_i) \tag{3.13}$$

Essentially, [100] tries to identify the parameter vector θ that optimizes the predictor f_θ for the given dataset, minimizing the average loss across all data points. The measurement of fairness then comes from a term called "excessive loss", which is basically the difference between the risk function of two models - the pruned and unpruned risk function over a group $a \in \mathcal{A}$:

$$R(a) = J(\bar{\theta}; D_a) - J(\hat{\theta}; D_a) \tag{3.14}$$

Here, the samples (x_i, a_i, y_i) are a subset of D, denoted by D_a whose group membership $a_i = a$. So, the excessive loss in [100] is a change in predictive accuracy after pruning with differential impacts on different groups in the training dataset. In a perfectly fair pruning system, the maximum excessive loss for a model should be zero, or $\xi(D) = 0$ in equation 3.15.

$$\xi(D) = \max_{a,a' \in \mathcal{A}} |R(a) - R(a')| \tag{3.15}$$

In [110], researchers propose the FairPrune method, which tries to achieve model fairness through pruning. The idea in this paper is not only to reduce the model size but also to remove the existing model biases after pruning. In their works, they were using two public dermatology datasets, Fitzpatrick 17k [25] and ISIC 2019 [48].

The Fitzpatrick 17k dataset is a dermatology dataset containing 16,577 clinical images, characterized by its diverse representation of skin types according to the Fitzpatrick skin type classification ranging from Type I, pale white skin, to Type VI, dark brown or black skin. Each image is annotated with both a skin type and a specific diagnosis, covering a broad spectrum of skin conditions from common ailments like acne, eczema, and psoriasis to more rare disorders. ISIC 2019 is also a dermatology dataset consisting of 25,331 dermoscopic images of skin lesions, annotated with one of by eight different categories. Each image is accompanied by clinical metadata such as patient age, sex, and lesion location. An important feature of these datasets was that the images represent a broad spectrum of skin types. Researchers used them to address a critical gap in medical image datasets that predominantly feature lighter skin tones, aiming to improve the performance and fairness of machine learning models in dermatology. [110] followed the idea that different parameters inside a machine learning model have different importance for various groups' accuracy. Therefore they evaluate the significance of each parameter by analyzing the second derivative of the parameters of a pre-trained model, assessing their impact on model accuracy for different groups.

[77] also uses Fitzpatrick 17k and CelebA [67] datasets. The CelebA (CelebFaces Attributes) dataset is a large-scale face attributes dataset with more than 200,000 celebrity images, each annotated with 40 attribute labels. The

images in CelebA cover large pose variations and background clutter, making it suitable for a variety of tasks such as face recognition, face attribute recognition, and generative modeling. While CelebA is mainly used in advancing facial analysis technologies, in [77] looked at it as a binary problem with a face being blond and non-blond blond, where 14.05% of the data were blond people. In all these examples, the main focus of the research was to design a fairness-first pruning method, meaning that the model's fairness was a measure of how much to prune and how to prune. While we are following a similar approach by comparing the accuracy of each class before and after pruning, we only do this as a final test. Unlike the mentioned works, we did not aim to remove the existing model biases. We checked the model's final status to ensure that after pruning, we have not made existing biases worse.

3.1.5 Using Clustering Methods in Pruning Algorithm

Clustering algorithms are also being used in pruning techniques [16][117]. [16] introduces Cluster Pruning (CUP), which prunes similar filters by clustering them based on features derived from both incoming and outgoing weight connections. This is a three-step process where new features are created based on weights and biases in the first step and are concatenated to a single value.

$$\tilde{F}_{i,:}^{(l)} = \text{concat}(\quad \tilde{W}_{i,:}^{(l)} \bar{B}_{i}^{(l)} \quad , \quad \tilde{W}_{i,:}^{(l+1)} \quad) \tag{3.16}$$

In equation 3.16 we have dense layers l and $l+1$, $\tilde{W}_{i,:}^{(l)} \bar{B}_{i}^{(l)}$ are the incoming features and $\tilde{W}_{i,:}^{(l+1)}$ represents the outgoing features. This equation represents parameterization of the l^{th} dense layer such that $\tilde{W}^{(l)} \in \mathbb{R}^{m \times n}$ are the weights and $\bar{B}^{(l)} \in \mathbb{R}^{m}$ are the bias, assuming that the neural network layer has $n; m$ and p filters respectively. However, t[16] formed the feature set for the convolutional layers as:

$$\tilde{F}_{i,:}^{(l)} = \text{concat}(g(\tilde{W}_{:,i,:,:}^{(l)}), b_{i}^{(l)},\ \tilde{W}_{i,:,:,:}^{(l+1)}) \tag{3.17}$$

$$\text{where } g(\tilde{X}_{:,:,:}) = [\|X_{1,:,:}\|_F, \ldots, \|X_{C,:,:}\|_F] \tag{3.18}$$

where $\tilde{W}^{(l)} \in \mathbb{R}^{n \times m \times k_h \times k_w}$ shows the weight parameterization of the l^{th} convolutional layer and the bias vector is represented by $\bar{B}^{(l)} \in \mathbb{R}^m$. In this notation, n is the number of input channels, m is the number of filters in layer l, and k_h, k_w represent the filter's height and width, respectively. In equation 3.18, the $g : \mathbb{R}^c \times \mathbb{R}^d \times \mathbb{R}^e \times \mathbb{R}^c$ is the channel-wise calculation of the Frobenius norm of $\tilde{X}$, any arbitary 3D tensor.

In the second step, using agglomerative clustering [108] and the feature vector $\tilde{F}_{i,:}^{(l)}$, a dendrogram is created. A dendrogram is a type of tree diagram that represents the arrangement of the clusters produced by hierarchical clustering [32]. It illustrates the process of cluster formation by successively merging or splitting clusters, with each node representing a cluster and each edge representing the dissimilarity or distance between clusters. The height of each node corresponds to the distance at which clusters are merged, providing a visual summary of the clustering process and the relative distances between clusters. Then, using a hyperparameter t, they select the threshold height for selecting clusters from the dendrogram. The lower this t is set, the more clusters will be selected. Finally, CUP selects a representative filter for each cluster and removes the other filters.

3.2 Problem Formulation

3.2.1 Measuring Filter Importance

Let C^i be the i-th convolutional layer of a pre-trained CNN. Feature maps are the output of the kernel matrix $O^i = \{ o_1^i, o_2^i, \ldots, o_{n_i}^i \} \in R^{n_i \times g \times h_i \times w_i}$, where the j-th feature map $o_j^i \in R^{g \times h_i \times w_i}$ is generated by $F_{i,j}$. Here, g, h_i, and w_i are the size of input images, height, and width of the feature map, respectively. We divide the filters in F_{C^i} in two groups: 1) a subset of filters to be kept, denoted as $I_{C^i} = \{ F_{I_1^i}^i, F_{I_2^i}^i, \ldots, F_{I_{n_{i1}}^i}^i \}$; and 2) the remaining filters that will be pruned, denoted as $U_{C^i} = \{ F_{U_1^i}^i, F_{U_2^i}^i, \ldots, F_{U_{n_{i2}}^i}^i \}$. Their union (all filters in C^i) will be denoted as W_{C^i}. I_j^i is the j-th filter in I_{C^i}, and U_j^i is the j-th filter in U_{C^i}, in the i-th convolutional layer. Also, n_{i1} and n_{i2} are the number of filters in I_{C^i} and U_{C^i}, respectively. Plainly, the following statements are true: $I_{C^i} \cap U_{C^i} = \emptyset, I_{C^i} \cup U_{C^i} = W_{C^i}$ and $n_{i1} + n_{i2} = n_i$

3.2.2 Filter Categorization

We consider all filters to initially be in I_{C^i}; we seek to remove the less important filters from it. We treat this as an optimization problem:

$$\min_{\delta_{ij}} \sum_{i=1}^{K} \sum_{j=1}^{n_i} \delta_{ij} L\left(w_{j^i}\right), \quad s.t. \sum_{j=1}^{n_i} \delta_{ij} = n_{i2}, \tag{3.19}$$

In the above equation δ_{ij} indicates which set $F_{i,j}$ belongs to; if $F_{i,j}$ is grouped in U_{C^i} then $\delta_{ij} = 1$, and $\delta_{ij} = 0$ if it is assigned to I_{C^i}. $L(\cdot)$ is a measurement used to define if a filter is important to the CNN or not, so by minimizing Eq. 3.19 the goal of removing the n_{i2} least important filters from C^i will be met. Next, consider that per [116], every feature map has a different role in the network. Instead of directly applying $L(\cdot)$ on the filters, we apply it on the feature maps since they reflect both filter properties and the input images. Thus, we reformulate Eq. 3.19 to:

$$\min_{\delta_{ij}} \sum_{i=1}^{K} \sum_{j=1}^{n_i} \delta_{ij} E_{I \sim P(I)} \left[\hat{L}\left(o_j^i(I)\right)\right], s.t. \sum_{j=1}^{n_i} \delta_{ij} = n_{i2},, \tag{3.20}$$

In Eq. 3.20, I is a sample input image from the distribution $P(I)$, $o_j^i(I)$ is the generated feature maps from the filter $F_{i,j}$ and $\hat{L}(\cdot)$ is an estimate of the information content of $o_j^i(I)$. Since $\hat{L}(\cdot)$ is proportional to $L(\cdot)$ in Eq. 3.19, we interpret lower information content to mean the corresponding filter is also less important.

3.2.3 Information Richness of Feature Maps

HRank adopts the rank of feature maps as the value of $\hat{L}(\cdot)$. The information measurement is defined as:

$$\hat{L}\left(o_j^i(I)\right) = Rank\left(o_j^i(I)\right), \tag{3.21}$$

In Eq. 3.21, $Rank(\cdot)$ is the rank of a feature map, given I as the input image. Then, on the $o_j^i(I)$, a Singular Value Decomposition (SVD) is conducted:

$$o_j^i(I) = \sum_{i=1}^{r} \sigma_i \, u_i v_i^T = \sum_{i=1}^{r'} \sigma_i u_i v_i^T + \sum_{i=r'+1}^{r} \sigma_i u_i v_i^T \tag{3.22}$$

Where r is equal to $Rank(o_j^i(I,:,:))$, r' is less than r, and σ_i , u_i and v_i are the top-i singular values, left singular vector and right singular vector of $o_j^i(I,:,:)$, respectively. From the equation, it is clear that a feature map with rank r is decomposable into: 1) a lower rank feature map with rank r' 2) some extra information not present in r'. $\sum_{i=1}^{r'} \sigma_i u_i v_i^T$ is a representation of the lower rank feature map and $\sum_{i=r'+1}^{r} \sigma_i u_i v_i^T$ is the additional information. Thus, higher-ranked feature maps contain more information than lower-ranked ones. Therefore, the rank of a filter is determined from the rank of its feature map.

Chapter 4

Methodology of CRank

4.1 Class Conditional Ranking System

Pruning models based on the feature maps was explored in [41], [62]. One of the main experiments performed in HRank [62] compares the average generated ranking of a filter, calculated as the average ranking generated by SVD applied on the corresponding feature maps. They found the ranking doesn't change when the size of input batches is increased from 1 to 50, and they mathematically prove that filters with lower ranks contain less information. In this paper, we utilize both of these findings. We choose a very small batch size for ranking generation, and we measure the information content of each filter as in 3.21 and 3.22. However, we also show that the average generated rank does not fully describe the information content of filters. Our experiments show that, within a CNN layer, many filters perform well for some image classes but not others, implying different impacts on network accuracy even when average generated rankings are similar.

To measure these impacts, we calculate the rank of the feature maps by selecting a random set of input images from each class, building a Class-based Ranking (CRank), which is a feature matrix, to represent the information content of a filter. This process is depicted in 4.1. We first start by feeding batches of images of the same class to the network. Consider a single convolutional layer L as is shown in figure 4.1. In that layer, there are J filters that, upon receiving images, would produce a feature map. We calculate the average rank of feature maps produced by each filter per each image class, which will form

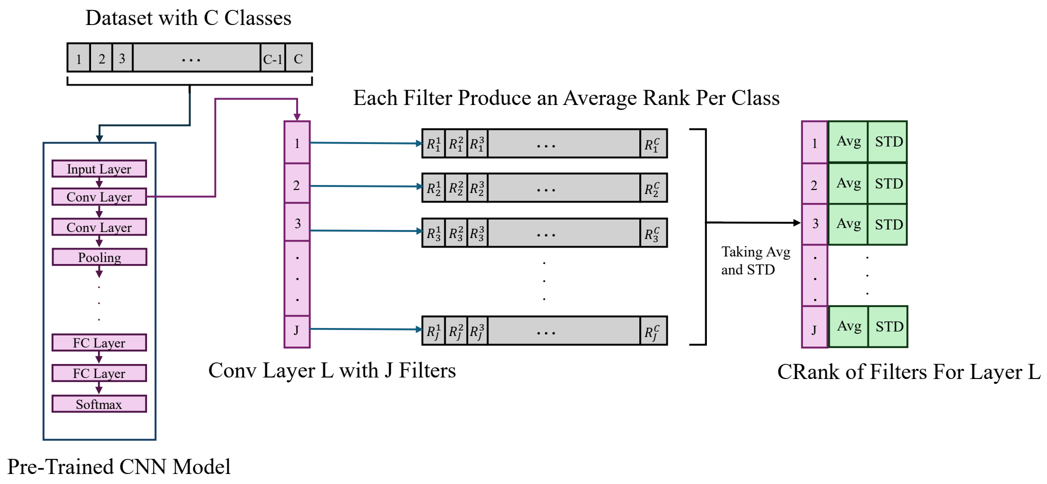

Figure 4.1: CRank Framework

a vector from R_1^1 to R_1^C for filter 1 and R_J^1 to R_J^C for the last filter. This results in a matrix of values that can be visualized, such as figure 4.2, where we can see the performance of each filter with regard to the image classes it received. Finally, a min-max normalization is applied to the CRank features of each layer.

The example image in figure 4.2 shows the heatmap of the CRank matrix for all filters in the first convolutional layer of VGGNet trained on CIFAR-10. Each row is one filter, and each column shows the average generated rank of filters as per Eq. 3.21 for various classes.

Then, we take those values' average and standard deviation for all the image batches. These new features are stored in the CRank Matrix for layer L. Upon repeating this process for all the layers, we will have the CRank tensor for all the filters in our pre-trained networks, which can later be used to select filters. This process is done for all filters, so at the end of this feature engineering phase, each filter in our neural network will have a number of ranks (our new features). Note that the number of features is equal to the number of classes of the dataset. In order to show the difference between our method and HRank, we have visualized two CRank feature matrices of filters 18 and 27 on top of the heatmap. As shown in this figure, our class conditional

Algorithm 1 Rank Generation

```
Require: $I(c) \leftarrow$ Per class batch of images
         $model \leftarrow$ A pre-trained model
Ensure: $CRank \leftarrow$ Class Conditional Ranks of Filters

for each $I(c)$ do
    for conv. layer $l$ do
        for filter $j$ do
            $CRank_j^l \leftarrow$Calculate the Filter's Rank
            $CRank_j^l \leftarrow$Calculate Average and Standard Deviation of Each Filter for all classes
        end for
    end for
end for
```

ranking system yields a more precise definition of the information carried by each filter compared to HRank, which only assigns a single value to the filter.

Our class conditional ranking system is summarized in algorithm 1 where we select a fixed batch size of sample images from one class called $I(c)$, where I represents the image and (c) denotes the class. Then, for each convolutional layer in our model, we calculate the numerical matrix rank produced by each filter for all the input images. $CRank_j^l$ shows the rank of all batches of image classes $I(c)$ for filter j in layer l of the convolution layers. Then, we aggregate those values in $CRank$. A sample of CRank can be seen in figure 4.2. Next, we will apply a k-means clustering algorithm to the filters of each convolutional layer using newly created CRank features for AVG and STD.

4.1.1 K-Means Clustering

Clustering is one of the fundamental approaches to pattern recognition. The aim is to partition a finite set of data into K clusters (subgroups) such that data points in the same cluster are very similar to each other and also highly different from the data points in other clusters. There are many clustering algorithms in the literature; one of the simplest and most popular is K-means clustering [19], [71]. In K-means, K is a predetermined value representing the number of distinct, non-overlapping clusters, and each data point can only belong to one group. The algorithm optimizes a cost function given by:

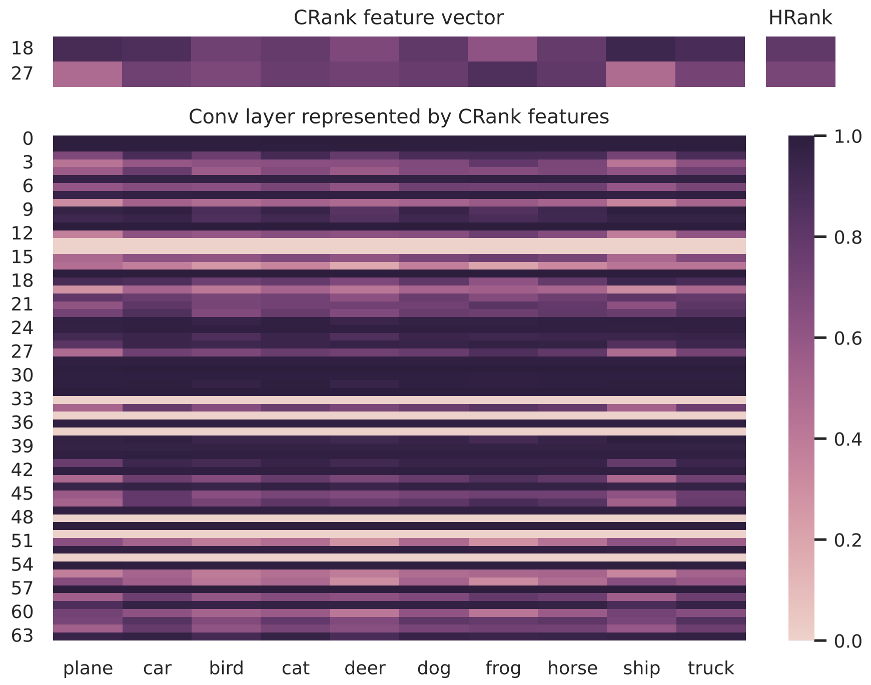

Figure 4.2: Heatmap of filter's ranking based on different classes for the first conv. layer of VGGNet for CIFAR-10.

$$J = \sum_{i=1}^{n} \sum_{k=1}^{K} r_{ik} \|\mathbf{x}_i - \mu_k\|^2 \tag{4.1}$$

where n is the number of data points, K is the number of clusters, $\mathbf{x}_i$ is the i-th data point, μ_k is the centroid of the k-th cluster, r_{ik} is a binary indicator showing whether $\mathbf{x}_i$ is assigned to cluster k, and $\|\mathbf{x}_i - \mu_k\|^2$ represents the squared Euclidean distance. The algorithm minimizes this function through an iterative process that alternates between assigning each data point to the nearest centroid and updating the centroids to the mean of the points assigned to them. It continues until the centroids stabilize or a stopping criterion is met. Algorithm 2 summarizes the above.

One important question remains: how to select the value of K, i.e., the number of clusters. We will look at how to answer this question next.

Algorithm 2 K-means Clustering

```
Require: Number of clusters $K$, dataset $\mathbf{X}$ containing $n$ data points
Ensure: Clusters that minimize the within-cluster sum of squares
Initialize $K$ centroids $\mu_1, \mu_2, \ldots, \mu_K$ randomly from the dataset $\mathbf{X}$
repeat
  Assignment Step:
  for each data point $\mathbf{x}_i$ in dataset $\mathbf{X}$ do
    $c_i \leftarrow \arg\min_k \|\mathbf{x}_i - \mu_k\|^2$
  end for
  Update Step:
  for $k = 1$ to $K$ do
    $\mu_k \leftarrow \frac{1}{|\{i:c_i=k\}|} \sum_{\{i:c_i=k\}} \mathbf{x}_i$
  end for
until centroids $\mu_k$ do not change significantly or a stopping criterion is met
```

4.1.2 Validating The Number of Clusters Via Silhouette Score

There are various metrics available to determine the optimal value for K. One such metric is the Silhouette score, which assesses how similar an object is to its own cluster compared to other clusters [87]. The Silhouette score provides insight into how well each object fits within its cluster compared to other clusters, effectively measuring the quality of the clustering clusters[87]. Specifically, for a dataset divided into K clusters, where C_i represents a specific cluster and an element i belongs to C_i, the average intra-cluster distance, $a(i)$, can be calculated as follows:

$$a(i) = \frac{1}{|C_i| - 1} \sum_{\substack{j \in C_i \\ i \neq j}} d(i, j) \tag{4.2}$$

Here, $|C_i|$ denotes the number of points in cluster i, and $d(i, j)$ represents the distance between points i and j in cluster C_i. Essentially, $a(i)$ measures how well i fits into its cluster, with lower values indicating better cohesion. Another critical aspect is separation, which evaluates how dissimilar point i is from the nearest cluster C_k that it does not belong to. This is quantified by $b(i)$, the minimum average distance from i to all points in a different cluster:

$$b(i) = \min_{k \neq i} \left(\frac{1}{|C_k|} \sum_{j \in C_k} d(i,j) \right) \tag{4.3}$$

So $b(i)$ is the minimum distance from i to all the data points in a cluster that i is not a member of. The cluster with this min value is the next best-fit cluster for i and is called the neighboring cluster of i. We can now computer the Silhouette score of datapoint i as:

$$s(i) = \frac{b(i) - a(i)}{\max\{a(i), b(i)\}}, \quad \text{if } |C_i| > 1 \tag{4.4}$$

where $s(i) = 0$ if $|C_i| = 1$. The score $s(i)$ ranges from -1 to $+1$, where a score closer to 1 indicates a strong affiliation of the object to its cluster, delineated by both tight intra-cluster cohesion and clear inter-cluster separation. A score near 0 suggests the object is positioned on the boundary between two clusters, and a negative score indicates potential misclassification of the object into its current cluster, highlighting closer proximity to another cluster [46]. Through analysis of silhouette scores for various K values ($3 < K \leq n$, where n is the total number of data points, the optimal K can be automatically selected based on the highest $s(i)$ value.

4.2 The Four CRank Regions

The core idea of our pruning algorithm is to use clustering to group filters together, using the features in *CRank*. We can then examine the clusters and select which ones will be included in I_{C^i} and which are relegated to U_{C^i}. The ideal cluster in I_{C^i} would be a group of filters with the maximum possible numerical matrix rank in all classes that would result in $AVG = Max \& STD = 0$. While such a cluster is unlikely to exist, our selection criteria will essentially be how closely a cluster approximates that ideal. Notice that this approach also follows the fair pruning model; by seeking clusters that perform well on all classes rather than just total performance, we minimize differential impacts from pruning on different groups in our data.

We perform a cluster analysis of the *CRank* matrix, using the K-means

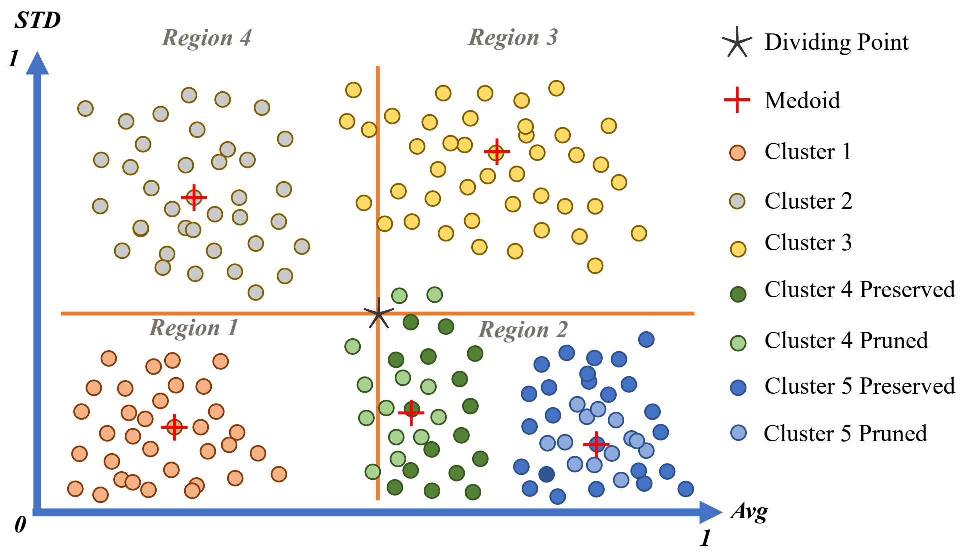

Figure 4.3: Visualization of filters clustering based on overall AVG and STD rankings.

algorithm, with the selection of k guided by the Silhouette score [87] discussed above. We use the Scikit-learn [83] implementation of K-Means, which offers several different distance functions (see Table 3.1). After some experimentation, we found that the other distance metrics made no discernable difference in our results, so we proceeded with the primary Eudclean distance in our clustering.

identifier	**class name**	**args**	**distance function**
"euclidean"	EuclideanDistance	•	$\sqrt{\sum((x-y)^2)}$
"manhattan"	ManhattanDistance	•	$\sum(\|x-y\|)$
"chebyshev"	ChebyshevDistance	•	$\max(\|x-y\|)$
"minkowski"	MinkowskiDistance	p	$\sum(\|x-y\|^p)^{1/p}$
"wminkowski"	WMinkowskiDistance	p, w	$\sum(\|w*(x-y)\|^p)^{1/p}$
"seuclidean"	SEuclideanDistance	V	$\sqrt{\sum((x-y)^2/V)}$
"mahalanobis"	MahalanobisDistance	V or VI	$\sqrt{((x-y)'V^{-1}(x-y))}$

Table 4.1: Metrics intended for real-valued vector spaces

Figure 4.3 shows an example of *CRank* clusters, sorted by AVG and STD, which we divide into four regions. Clusters with High Average and Low Standard Deviation (Region 2) are those that perform well for all classes; these are the filters that we keep in the first round of pruning.

Algorithm 3 Finding Unimportant Filters

```
Require: $CRank^l_j$ ←Calculated the Filter's Rank
Ensure: $U_{C^i}$ ← A selection of filters to be pruned

for conv. layer $l$ do
    Dividing point ← $mean(CRanks^l_{avg}, CRanks^l_{std})$
    $N_l$ ← number of filters in layer $l$
    for $K$ from 4 to $N_l$ do
        Perform $K$-means on $CRank^l$
        Calculate silhouette score of $S(K)$
    end for
    select best $K$ using $argmax\ S(K)$
    for each medoid of $K$-means do
        If medoid is not in Region 2 then
            Insert cluster's filters to the unimportant filter set
        else
            for each remaining cluster $Cl$ do
                for each filter $j$ in the cluster do
                    $sil_{CL_j}$ ← Calculate silhouette sample
                end for
                $avg_{sil_{Cl}}$ ← average silhouette of $Cl$
                if $sil_{CL_j} > avg_{sil_{Cl}}$ then
                    Add filter to Unimportant filter set
                end if
            end for
        end else
    end for
end for
Add remaining filters into the Important filter set
```

4.3 Automatic Compression

CRank is designed to determine optimal pruning without manual trial-and-error, unlike [36], [42], [62], [64], [68]. [90] applies automatic compression, but still requires manual trial-and-error exploration of a hyperparameter controlling the compression. As above, we cluster the filters based on their CRank feature vectors and eliminate the clusters outside of the high-AVG, low-STD region. However, some redundancy likely persists in the remaining filters. Thus, we next remove the filters whose individual silhouette score [87] is greater than the average for the cluster. The idea behind this approach is that a high sil-

Algorithm 4 Fine-Tuning

```
/* **Fine-Tuning** */
for epoch from 1 to 30 do
    Pruning unimportant filters and *model* fine-tuning
endfor
```

houette score means these filters are densely packed in the cluster and highly similar to each other and the cluster medoid. After this step, the remaining filters are all high-performing and should have little remaining redundancy. For example, in figure 4.3, cluster 5 is generally the highest-quality cluster. However, the light blue filters are pruned due to their similarity to the medoid. Our experiments show that removing these filters increases the pruning rate without losing significant accuracy. Our full proposed technique is presented in Algorithm 3.

Fine Tuning and Pruning

The final step of our method involves removing the filters in U_C. This is done during the fine-tuning step, during which we will update the model weights. If the filter is in U_C, we set the weights and biases of all neurons in the filter to 0, as in algorithm 4 shows this procedure. The fine-tuning and pruning happens at the same time in this step

Chapter 5

Experiments and Results

5.1 Evaluation

In this section, we describe our experimental methodology and results. We compare our CRank method against HRank using the same compression rates in each layer in order to fairly evaluate the quality of filters being retained by each method. We also compare the accuracy, number of parameters, and total FLOPs of our reduced networks with the same-compression HRank, the published results on HRank, and published results for other pruning methods. To put the fairness of our method to the test, we calculate the class-based accuracy of models before and after pruning. We compute the Pearson and Spearman correlation coefficients [20] [112] to see if the pruned model's class-based accuracies are following a different pattern than the original pre-trained model. We also tested the energy efficiency of CRank by comparing the power consumption of pruned and unpruned models.

5.1.1 Experimental Setup

Datasets

In order to evaluate the performance of our method, we have chosen three well-known benchmark datasets. The CIFAR-10 dataset [49] contains 60,000 32×32 pixel color images in 10 classes, with 6,000 images per class. The images are split into 50,000 training images and 10,000 testing images. The CIFAR-100 dataset [49] is similar to CIFAR-10 with the exception that it consists ofs 100 classes with 600 images in each class divided into 500 training samples and 100 test samples. The ILSVRC subset of the ImageNet dataset [89] contains over 1.2 million high-resolution images belonging to 1,000 classes. The images in ImageNet are much more varied and complex than those in CIFAR datasets, making it a more challenging and computationally demanding image classification problem.

Our empirical tests showed that input batch size does not affect the average generated rank for feature maps, which confirms the observations in [62]. Thus, we chose 50 randomly selected images per class for the CIFAR datasets and ten randomly selected samples per class for the ImageNet dataset in order to produce the average rankings.

Baseline Architectures

As we explained in the introduction, due to the similarity of our technique to HRank and since it is one of the leading pruning methods, we selected HRank as our baseline comparison point. We tested CRank and HRank pruning on two mainstream state-of-the-art CNN architectures: VGGNet [95] (with plain structure) and ResNet [33]. These two models are commonly used in evaluating pruning techniques [36], [42], [62], [64], [68]. Our experiments on CIFAR-10 covered both architectures, while on CIFAR-100, we only used VGGNet, and for the ImageNet dataset, we only used ResNet architecture due to the computational demands of the experiments. In addition to our comparison with Hrank, we also compare CRank with other pruning techniques in the literature.

Evaluation Protocols

To evaluate CRank and HRank pruning, we have followed a commonly accepted protocol: we calculate the number of remaining parameters and require Floating-Point Operations per second (FLOPs) after pruning. In addition, after fine-tuning, the top-1 accuracy of the pruned model is calculated to give a sense of the task-specific abilities of the pruned model. When reporting the results for ImageNet, the top-5 accuracy is also presented (as this is frequently done in the literature). The Pruning Rate (PR) for FLOPs and Parameters are indicated to show the models' compression capabilities. In our experiments, we also care about how similar our approach is to [62]. Therefore, we compare the remaining filters in the I_{C^i} list between the two methods. The results on the class-conditional accuracies and power consumption of each model are then brought into their respective subsections after going through the accuracy and flops.

Configurations

Our CRank and HRank pruning are performed in Pytorch [82]. For fine-tuning, we have used Stochastic Gradient Descent (SGD) [97] with an initial learning rate of 10^{-2} as our optimizer. Momentum is set to 0.9, and weight decay is set to 5×10^{-4}. Our CIFAR-10 and CIFAR-100 experiments were run on an NVIDIA TITAN Xp GPU, whereas the ImageNet experiments were run on a dual NVIDIA Tesla A100 system. The batch size for CIFAR datasets is set to 12,8, and for ImageNe,t, it is set to 750. Our fine-tuning runs for 30 epochs. During this process, we reduce the learning rate by an annealing schedule. This is highly similar to the fine-tuning procedures in [62]. As CRank pruning is designed to automatically find the"best" pruning rate, while HRank accepts the pruning rate as an input parameter, a head-to-head comparison can only be accomplished by running CRank first and then setting the HRank pruning rates of each layer to match those selected by the CRank algorithm.

5.1.2 Experiments Analysis

In this section, we will present and compare our accuracy and Pruning Rate (PR) with other state-of-the-art methods. We compare our filter pruning approach with various methods such as adaptive importance methods of GAL [64], Zhao et al. [115], SSS [42], and a more head-to-head (apple-to-apple) comparison with the state of the art pruning method of HRank [62]. We focus on comparison with HRank as CRank shares a similar conceptual framework to HRank, and there are fewer confounding factors when we attempt to judge the quality of the pruning process itself.

Table 5.1: CRank v HRank for 10 experiments on CIFAR-10

Arch.	Method	Mean	STD
VGGNet	CRank	87.799	0.2110
	HRank	87.527	0.1966
ResNet50	CRank	87.67	0.1192
	HRank	86.878	0.2306

Similarity of Filters Within Layers

A key aspect of our study involves comparing our approach with HRank, particularly in terms of accuracy, compression rate, and the selection of filters retained by each methodology. The pre-trained models used in the experiments are identical in each set of experiments (HRank v.. CRank), so the filters had identical features. This similarity of input features prompted our investigation into the distinctiveness of our method in comparison to HRank. A straightforward approach to understanding this difference was to examine the specific filters each method preserves. For instance, in the VGG16 architecture's first convolutional layer, there are 64 filters. Based on CRank's automatic methodology tested on CIFAR-10, our objective is to prune 89% of this layer, leaving only seven filters intact. A pertinent question arises: when assigning an 89% pruning rate to VGG16's first layer using the HRank methodology, how many of the remaining filters would be similar to those retained by our CRank method? Given the fact that pruning is done on the same pre-trained model, thefilters" weights are identical at the rank generation and filter selection steps. So we can actually answer this question when analyzing the results for each dataset in their respective subsections in this work.

5.1.3 Results of CIFAR-10 Experiments

We first examine the head-to-head comparison with HRank in VGGNet and ResNet trained on the CIFAR-10 dataset. Our experiments were repeated ten times. Table 5.1 shows the mean and STD of the result of 10 runs of CRank and HRank on CIFAR-10 on both VGGNet and ResNet architectures.

On the VGGNet experiments, the maximum accuracy of CRank was 88.08%

Table 5.2: Similarity of Filters Preserved in VGG16 Architecture between CRank and HRank using CIFAR10 Dataset

	Compress Rate	Total Filters	Preserved	Difference	Similar
Conv 1	0.890625	64	7	7	0
Conv 2	0.8125	64	12	12	0
Conv 3	0.796875	128	26	25	1
Conv 4	0.835937	128	21	18	3
Conv 5	0.828125	256	44	37	7
Conv 6	0.851562	256	38	30	8
Conv 7	0.867187	256	34	31	3
Conv 8	0.882812	512	60	49	11
Conv 9	0.783203	512	111	68	43
Conv 10	0.771484	512	117	63	54
Conv 11	0.80468	512	100	52	48
Conv 12	0.845703	512	79	75	4
Conv 13	0.845703	512	79	75	4

and the minimum was 87.54%, whereasHRank'ss maximum was 87.78% and its minimum was 87.20%. On the ResNet architectureCRank'ss maximum accuracy was 87.87% and the minimum was 87.51%, whereasHRank'ss maximum and minimum accuracy were 87.32% and 86.39% respectively.

We examine the statistical significance of our results using a two-tailed t-test and the usual significance of $\alpha = 0.05$. For VGGNet, the two-tailed p value is equal to 0.0080, indicating that the difference between CRank and HRank is statistically significant. For the ResNet experiments, the two-tailed p value is less than 0.0001, indicating that the difference between CRank and HRank is again statistically significant.

To show the difference between our filter selection procedure compared to HRank, we analyzed the retained and pruned filters in both methods, as can be seen in Table 5.2, with about 82.76% overall compression rate among all the 4224 filters in this network, only 186 of them were similar which means only

Table 5.3: Comparing CRank with literature on CIFAR-10

Model	FLOPs (PR)	Parameters (PR)	Acc %
	VGG16 Architecture		
VGGNet (Base)	313.73M (0.0%)	14.98M(0.0%)	93.96
L1 [59]	206.00M(34.3%)	5.40M(64.0%)	93.4
SSS [42]	183.13M(41.6%)	3.93M(73.8%)	93.02
Zhao et al [115]	190.00M(39.1%)	3.92M(73.3%)	93.18
Gal - 1 [64]	171.89M(45.2%)	2.67M(82.2%)	90.73
HRank [62]	63.18M(79.86)	4.90M(67.28%)	87.52
CRank (Ours)	63.18M(79.86)	4.90M(67.28%)	87.79
	ResNet56 Architecture		
ResNet56 (Base)	125.49M (0.0%)	0.85M(0.0%)	93.26
L1 [59]	90.90M(27.6%)	0.73M(14.1%)	93.06
NISP [74]	81.00M(35.5%)	0.49M(42.4%)	93.01
FilterSketch [61]	32.47M(74.4%)	0.24M(71.8%)	91.2
He et al. [36]	62.00M(50.6%)	-	90.8
HRank [62]	32.52(74.1%)	0.27M(68.1%)	90.72
Gal - 0.8 [64]	49.99(60.2%)	0.29M(42.4%)	90.36
HRank [62]	23.87M(80.97%)	0.14M(83.53%)	86.87
CRank (Ours)	23.87M(80.97%)	0.14M(83.53%)	87.67

about 4% similarity between the two methods. A similar table for the ResNet model could be constructed as well, but due to the large column size, the table is not shown here. Nevertheless, for ResNet-56 on the CIFAR-10 dataset,t we again found a similarity of about 4%. This low similarity shows that while both in HRank and CRank, the rank of output feature maps are used as a metric to rank the filter values, our class conditional method combined with clustering techniques makes the method very different than HRank. Next, we will compare the accuracies of our method with HRank and the literature to see, aside from being different, how effective our method is.

Table 5.3 compares CRank with other notable filter pruning results. On VGGNet, CRank pruning reduces the computational burden of the network in FLOPs by 79.86%, which is considerably more than any other method. The CRank also reduces the number of parameters in the network by 67.28%

Table 5.4: CRank vs. HRank on ImageNet (ResNet56)

Arch.	Method	Top-1 Acc %	Top-5 Acc %	FLOPs PR	Param PR
Freezing Last Block	CRank	60.13%	83.51%	78.72%	63.13%
	HRank	59.61%	83.18%		
Pruning All Layers	CRank	47.88%	74.57%	83.37%	78.39%
	HRank	47.53%	73.95%		

reduction. In the Resnet56 Architecture, CRank achieves the highest pruning rate compared to all other methods, with 80.97% and 83.53% pruning rates for FLOPs and Parameters, respectively. Note that we include the published result for HRank pruning on ResNet56 in [62]; this is the entry for HRank in the 4-th row from the bottom.

5.1.4 Results of ImageNet Experiments

Due to the computationally demanding nature of pruning tasks, we were not able to have ten repetitions of our experiments on ImageNet, so we instead used a single-split method to test the model. Table 5.4 compares CRank and HRank in a head-to-head comparison. There are two sets of results, as we found that, for the 1,000-class Imagenet problem, CRank's automatic pruning removes too many filters in the last layer of the CNN. Rather than create special pruning rules for this dataset, we examine simply not pruning ”freezing””) this layer, and compare that against allowing the automatic pruning to proceed unchanged. In both cases, the top-1 and top-5 accuracy after CRank pruning are greater than that of HRank pruning. Analysis of the retained filters shows that CRank and HRank only retain 2.5% of filters in common. Thus, we again see that the difference in accuracy between these approaches comes down to a more effective selection of filters to retain.

A comparison between CRank and other filter pruning methods for the ImageNet architecture is presented in Table 5.5. CRank reduces the required FLOPs more than any other current method, even when freezing pruning in the last layer. The trade-off is that other methods have superior top-1

Table 5.5: Comparing CRank with literature on ImageNet

Model	FLOPs (PR)	Parameters (PR)	Top-1 Acc %	Top-5 Acc %
ResNet50 [68]	4.09B(0.0%)	25.50M(0.0%)	76.15	92.87
SSS [42]	2.82B(31.1%)	18.60M(27.1%)	74.18	91.91
He et al [36]	2.73B(33.3%)	-	72.3	90.8
HRank [62]	0.98B(76.0%)	8.27(67.6%)	68.1	89.58
FilterSketch [61]	0.93B(77.3%)	7.18M(71.8%)	69.43	89.23
Gal-1 [64]	1.11B(72.9%)	10.21M(59.9%)	69.31	89.12
ThiNet-50 [68]	1.10B(73.1%)	8.66M(66.03)	68.42	88.3
CRank (Ours)	0.87B(78.72%)	9.4M(63.13%)	60.13	83.51
HRank [62]	0.87B(78.72%)	9.4M(63.13%)	59.61	83.18
CRank (Ours)	0.68B(83.37%)	5.51M(78.39%)	47.88	74.57
HRank [62]	0.68B(83.37%)	5.51M(78.39%)	47.53	73.95

and top-5 accuracy when they permit more computational effort. SSS, for instance, achieves a top-5 accuracy of 91.91%, but the network requires 2.82 GFLOPs of compute.CRank'ss top-5 accuracy is 83.51% but requires only 0.87 GFLOPs of computing. This raises the question of how CRank would perform under different choices of our automatic pruning criteria (e.g., removing fewer high-performing but redundant filters in the second pruning step above). We discuss this point in our Future Work section below. As for the similarity of CRank and HRank, due to the number of layers in ResNet50, we again did not duplicate the presentation of table 5.2. However, we found only about 2.5% similarity between CRank and HRank pruning for this network.

5.1.5 Results of CIFAR-100 Experiments

We compare HRank and CRank pruning of a VGGNet that we trained to achieve an accuracy of 69.57% Table 5.6 shows the mean and STD of the result of 10 runs of CRank and HRank on CIFAR-100 on the VGG-16 architecture.

On the CIFAR-100 experiments, the maximum accuracy of CRank was 59.44%, and the minimum was 58.48%, whereas HRank's maximum was 59.65% and its minimum was 58.83%. We examine the statistical significance of our

Table 5.6: CRank vs. HRank for 10 experiments on CIFAR-100

Arch.	Method	Mean	STD
VGGNet	CRank	59.04	0.3050
	HRank	59.30	0.2902

Table 5.7: Similarity of Filters Preserved in VGG16 Architecture between CRank and HRank using CIFAR100 Dataset

	Compress Rate	Total Filters	Preserved	Difference	Similar
Conv 1	0.828125	64	11	9	2
Conv 2	0.78125	64	14	10	4
Conv 3	0.8359375	128	21	18	3
Conv 4	0.765625	128	30	29	1
Conv 5	0.84765625	256	39	27	12
Conv 6	0.88671875	256	29	23	6
Conv 7	0.80078125	256	51	40	11
Conv 8	0.953125	512	24	20	4
Conv 9	0.90625	512	48	37	11
Conv 10	0.775390625	512	115	102	13
Conv 11	0.921875	512	40	39	1
Conv 12	0.943359375	512	29	28	1
Conv 13	0.943359375	512	29	28	1

results using a two-tailed t-test and the usual importance of $\alpha = 0.05$. We found that the two-tailed p value equals 0.967, indicating that the difference between CRank and HRank is not statistically significant. This means that in CIFAR-100 experiments, the comparison between our CRank method and HRank ended up in a draw.

Table 5.7 is similar to table 5.2, where a head-to-head similarity comparison between HRank and CRank is made. This table analyzes how similar the filters retained by HRank and CRank are. In this example, the similarities were even less than the CIFAR-10 tests with the same architecture, with only about 1.72% similarity between the two models.

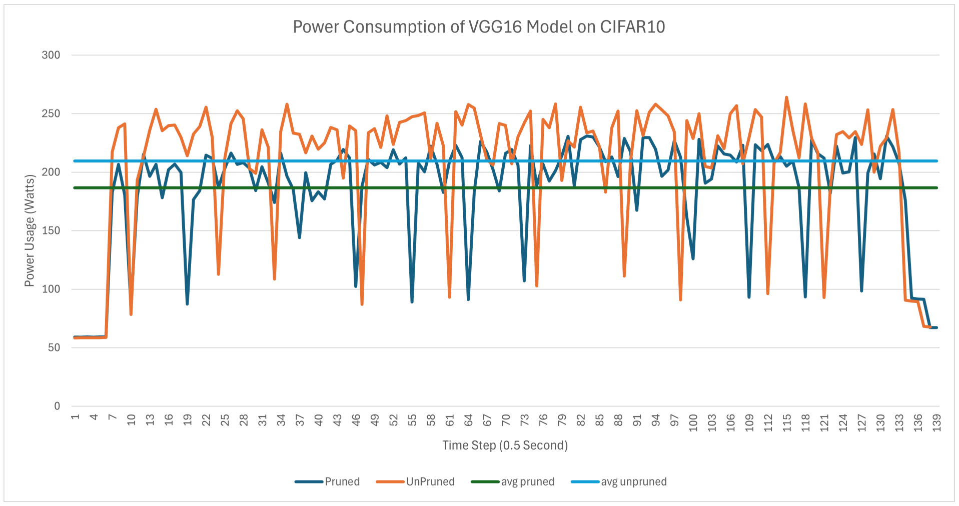

Figure 5.1: VGG16 on CIFAR-10

Power Consumption

As discussed in the introduction, a primary motive behind our work is to extend the ability of edge devices that have limited resources. When it comes to edge and portable devices, one of the most important resources is the battery. So as another set of experiments, we next compare the power consumption of our method to see if there is any reduction of power usage between the pruned and unpruned models. To run these tests, we are using the NVIDIA-SMI interface and log the power consumption of the device the models are running on 5 seconds before and after an inference. The maximum capacity of the NVIDIA TITAN Xp GPU, the hardware that we experimented with, was 250 Watts, with the ability to overclock.

As you can see from the figures, in all cases, the pruned model had a lower power consumption. Especially in ImageNet, the unpruned model had many jumps to nearly 300 Watts, overclocking our GPU. Meanwhile, the pruned model was more stable and rarely exceeded 250 Watts. Also, in the VGG experiments, both on CIFAR-10 and CIFAR-100, the pruned model had a noticeably lower average power consumption than the unpruned model. This average difference was less noticeable in ResNet on CIFAR-10, but we still see an overall higher wattage for the unpruned model. This confirms the effectiveness of pruning in reducing the model's power consumption. It is worth

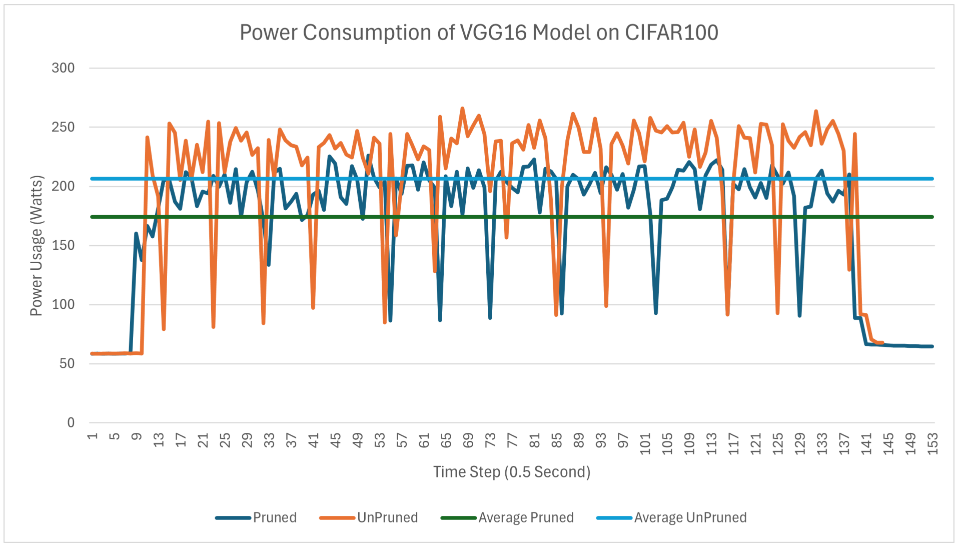

Figure 5.2: VGG16 on CIFAR-100

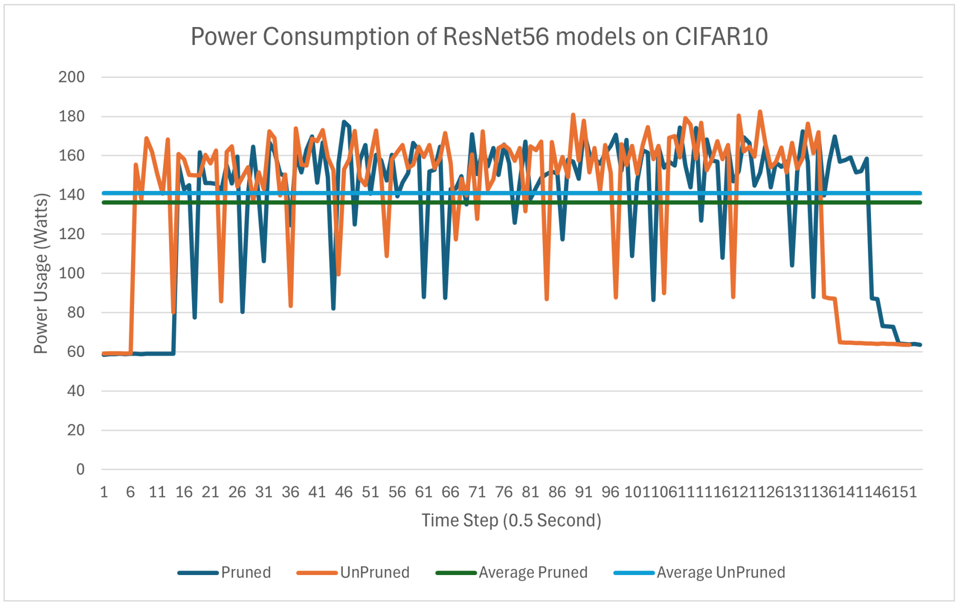

Figure 5.3: ResNet-56 on CIFAR-10

mentioning that to run these tests; we used desktop-based hardware connected to 120V AC power; the results for power-constrained hardware running on a battery are important questions for future work.

Fair Pruning

1) Pearson and Spearman Correlation Tests: To test the fairness of our method, we compare the accuracy of the VGG16 models on each class before

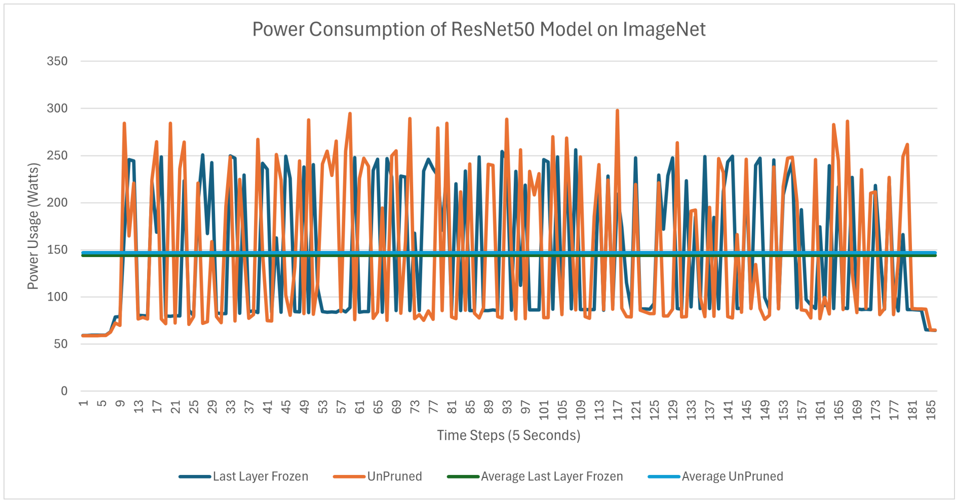

Figure 5.4: ResNet-50 on ImageNet

and after pruning on CIFAR-10. In the conducted analysis, we computed the Pearson and Spearman correlation coefficients to assess the relationship between the accuracies of UnPruned and Pruned models [20], [112] when the order of classes is constant. The Pearson correlation coefficient was found to be 0.6602 with a p-value of 0.0378, indicating a moderate positive correlation and statistical significance. Concurrently, the Spearman correlation coefficient was higher at 0.7439, with a p-value of 0.0136, suggesting a stronger positive correlation and also statistically significant. These results imply a significant, positive association between the accuracies of the two models, where classes that perform well or poorly in one model tend to exhibit similar performance in the other.

Building upon this experiment, we also tested the ResNet-56 architecture on CIFAR-10. The Pearson correlation coefficient was found to be 0.9763, with a p-value of approximately 2.75×10^{-7}. Similarly, the Spearman correlation coefficient was 0.9658, accompanied by a p-value of approximately 1.40×10^{-6}. There is thus little difference between the relative performances of the pruned vs. unpruned models.

Further correlation tests were conducted after extending this analysis to the CIFAR100 dataset using the VGG16 architecture. Here, both the Pearson and Spearman correlation coefficients indicated very strong positive correlations between the accuracies of UnPruned and Pruned models. Specifically, the Pearson coefficient was 0.9185 with a p-value of approximately 2.84×10^{-41}, and the Spearman coefficient was 0.9239 with a p-value of approximately 1.11×10^{-42}. These results underscore a stronger consistency in model performance across the CIFAR100 dataset compared to the CIFAR10 dataset. The high correlation coefficients and the extremely low p-values reflect a robust, statistically significant relationship, suggesting that the pruning process retains the relative performance across different classes very effectively. This consistency in model behavior for CIFAR100, evident from both linear and rank-order correlations, highlights the effectiveness of the pruning method in maintaining class-wise accuracy while reducing model complexity.

As the final test of class-conditional accuracies, we are going to look at the Pearson and Spearman tests for ResNet-50 on ImageNet in two different scenarios: 1) Unpruned model Vs. Fully Pruned Model and 2) Unpruned model Vs. The model is pruned while freezing the last layer, as discussed in section IV. The results of our analysis demonstrated significant correlations in both scenarios.

For the Unpruned versus Pruned model comparison, the Pearson Correlation Coefficient was 0.7715 with a p-value of approximately 1.98×10^{-198}, indicating a strong positive correlation. Similarly, the Spearman Correlation Coefficient was slightly higher at 0.7959, with a p-value of 5.99×10^{-220}. In the Unpruned versus Frozen Data comparison, the correlation coefficients were notably higher, reinforcing the effectiveness of our approach. The Pearson Correlation Coefficient reached 0.8897, while the Spearman Correlation Coefficient was 0.8978; in both cases, we had an extremely low p-value, which was truncated by Python to simply 0.

2) Equalized Opportunity and Equalized Odds Tests: In addition to Pearson and Spearman Correlation tests, we employ multi-class equalized opportunity (Eopp) and equalized odds (Eodd) [30] to evaluate the fairness of our model. These metrics help in assessing the model's fairness by examining the differences in prediction rates between the two groups. Eopp0 and Eopp1 focus on the fairness of negative and positive predictions, respectively,

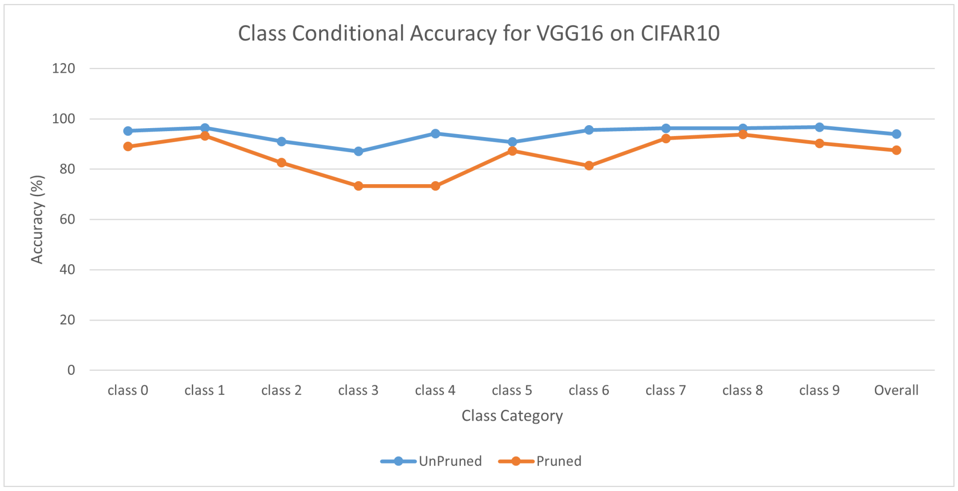

Figure 5.5: VGG16 on CIFAR-10

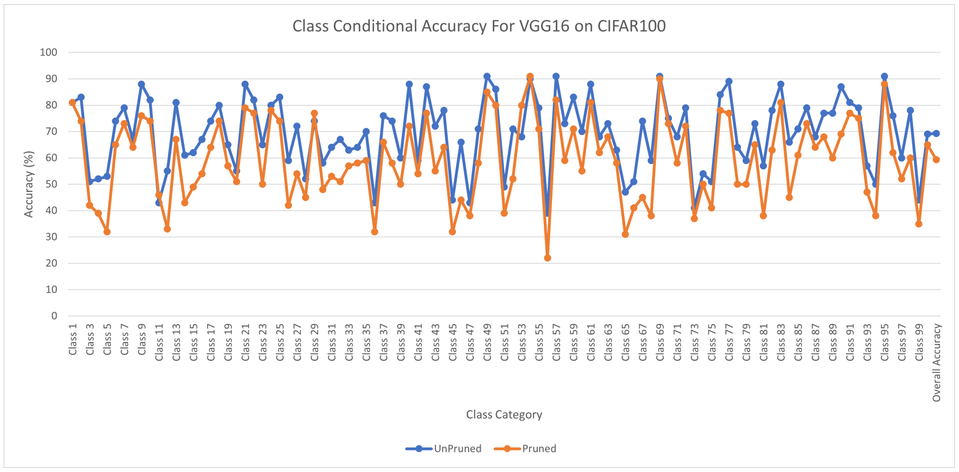

Figure 5.6: VGG16 on CIFAR-100

while Eodd provides a comprehensive measure of fairness by considering both positive and negative predictions across different groups.

Equalized Opportunity (Eopp) consists of two components: Eopp0 and Eopp1. Eopp0 measures the difference in True Negative Rate (TNR) between the two groups, while Eopp1 measures the difference in True Positive Rate (TPR) between the two groups. The True Positive Rate (TPR_k^c) and True Negative Rate (TNR_k^c) for class k and group c are calculated as follows:

$$TPR_k^c = \frac{TP_k^c}{TP_k^c + FN_k^c} \tag{5.1}$$

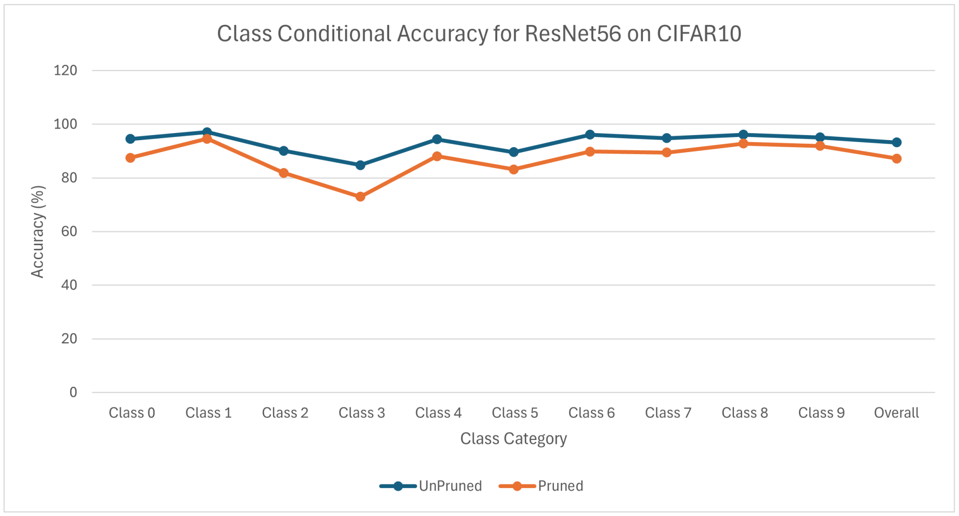

Figure 5.7: ResNet on CIFAR-10

$$TNR_k^c = \frac{TN_k^c}{TN_k^c + FP_k^c} \tag{5.2}$$

Where TP_k^c, FN_k^c, TN_k^c, and FP_k^c represent the true positive, false negative, true negative, and false positive counts for class k and group c. Eopp0 and Eopp1 are then computed using the following equations:

$$EOpp0 = \sum_{k=1}^{K} |TNR_k^1 - TNR_k^0| \tag{5.3}$$

$$EOpp1 = \sum_{k=1}^{K} |TPR_k^1 - TPR_k^0| \tag{5.4}$$

Equalized Odds (Eodd) evaluates the combined differences between the two groups' True Positive Rate and False Positive Rate (FPR). The False Positive Rate (FPR_k^c) for class k and group c is calculated as follows:

$$FPR_k^c = \frac{FP_k^c}{TN_k^c + FP_k^c} \tag{5.5}$$

The Eodd metric is computed by summing the absolute differences of TPR and FPR between the groups for each class:

$$EOdd = \sum_{k=1}^{K} |TPR_k^1 - TPR_k^0 + FPR_k^1 - FPR_k^0| \tag{5.6}$$

Table 5.8: Comparison of TPR, TNR and FPR - CIFAR10 on VGG16

Class	TPR UnPruned	TNR UnPruned	FPR UnPruned	TPR Pruned	TNR Pruned	FPR Pruned
airplane	0.952	0.992222	0.007778	0.890	0.984556	0.015444
automobile	0.964	0.997000	0.003000	0.933	0.993111	0.006889
bird	0.910	0.991778	0.008222	0.826	0.982667	0.017333
cat	0.871	0.986222	0.013778	0.733	0.975556	0.024444
deer	0.942	0.993667	0.006333	0.873	0.986222	0.013778
dog	0.908	0.988666	0.011333	0.814	0.979666	0.02033
frog	0.956	0.99522	0.004777	0.922	0.987889	0.01211
horse	0.963	0.99688	0.00311	0.922	0.99033	0.009666
ship	0.963	0.9958	0.004111	0.9388	0.99055	0.009444
truck	0.967	0.995333	0.004666	0.903	0.9911	0.009

So, to test the fairness of our method, we need to calculate the EOpp0, EOpp1, and EOdd. First, let's look at the True Positive Rate, True Negative Rate, and False Positive Rates for the VGG16 architecture on CIFAR10 with ten classes. The details of these values for both pruned and unpruned models can be seen in table 5.8.

So based on equations 5.3, 5.4, 5.6 and the values from table 5.8, the EOpp0, EOpp1 and EOdd for the VGG16 model on CIFAR-10 dataset will be 0.0713, 0.6401, 0.7101. The small value of $EOpp0 = 0.071$ indicates that the True Negative Rates between the pruned and unpruned models are very close, suggesting minimal disparity in negative predictions. The $EOpp1 = 0.6401$ is higher than EOpp0, suggesting some disparity in positive predictions. However, given the range that EOpp0 and EOpp1 could have, which is between 0 and $2k$, with k being the number of classes, we can confirm that the pruning is still fair toward all classes. The same goes for EOdd, with $EOdd = 0.7101$ and the range of $[0, 4k]$ for 0 when both the TPR and FPR for the pruned and unpruned models are identical for all classes. The maximum value of $4k$ occurs when the TPR and FPR for one group are 0 and for the other group are 1 for each class, resulting in an absolute difference of 2 for each class. So, in the case of pruning the VGG16 model trained on the CIFAR-10 dataset, these values suggest that our pruning process is fair, with some minor disparity in the True Positive Rates.

The same calculations can be done for our other experiments. For example, in the CIFAR-100 dataset on a VGG16 backbone, we formed a similar table to 5.8 but did not present it here due to a large number of rows. The results of

Table 5.9: Comparison of TPR, TNR and FPR - CIFAR10 on ResNet56

Class	TPR UnPruned	TNR UnPruned	FPR UnPruned	TPR Pruned	TNR Pruned	FPR Pruned
airplane	0.945	0.991111	0.008889	0.875	0.987222	0.012778
automobile	0.971	0.996111	0.003889	0.946	0.992556	0.007444
bird	0.901	0.993000	0.007000	0.819	0.979222	0.020778
cat	0.848	0.986000	0.014000	0.730	0.977889	0.022111
deer	0.944	0.991556	0.008444	0.881	0.983000	0.017000
dog	0.908	0.988666	0.011333	0.814	0.979666	0.02033
frog	0.956	0.99522	0.004777	0.922	0.987889	0.01211
horse	0.963	0.99688	0.00311	0.922	0.99033	0.009666
ship	0.963	0.9958	0.004111	0.9388	0.99055	0.009444
truck	0.967	0.995333	0.004666	0.903	0.9911	0.009

the calculations are as follows: $EOpp0 = 0.11191, EOpp1 = 10.3398, EOdd = 10.4519$. The threshold of fairness differs case by case, but based on the calculations in [30] as well the trade-off between fairness, the Fair Threshold for EOPP is $EOpp < 20$ which is equal to 10% of the range, anywhere from $20 < EOpp < 50$ or 25% of the range is Moderate Fairness, and anything above $50 < EOpp$ or above 50% of the range is unfair. The range percentage for EOdd remains the same: $EOdd :< 40$ is fair (10%of the range), $EOdd :< 40$ is moderately fair (10%of the range), and EOdd ¿ 100 is unfair. Based on these, we see that our pruning method for VGG16 on CIFAR-100 is also within the fair threshold.

Next, we move to the ResNet architecture and test the Equalized Opportunity and Equalized Odds for CIFAR-10 using ResNet-56. Table 5.10 shows the details of TPR, TNR, and FPR for each class. Based on equations 5.3, 5.4, 5.6 we obtain $EOpp0 As we explained earlier, = 0.06698, EOpp1 = 0.60298, EOdd = 0.6699$, which all stay within the Fair Threshold.

Finally, the same fairness evaluation is applied to the ImageNet dataset with 1000 classes. For this dataset, the calculated values are as follows: $EOpp0 = 0.1773$, $EOpp1 = 161.46$, and $EOdd = 161.6373$. Given the possible ranges for these metrics ($EOpp0 : 0{-}2000$, $EOpp1 : 0{-}2000$, $EOdd : 0{-}4000$), the pruning process for the ImageNet dataset with 1000 classes is again within the Fair range. The low values for $EOpp0$, $EOpp1$, and $EOdd$ indicate that the pruning does not introduce significant unfairness, maintaining acceptable levels of disparity in the model's performance across different groups. Table 5.10 shows the summary of these experiments. In these experiments, all values

Table 5.10: Table of All EOpp and EOdd Values

Model	Dataset	Number of Classes	EOpp0	EOpp1	EOdd	Is Fair
VGG16	CIFAR-10	10	0.0713	0.6401	0.7101	Yes
VGG16	CIFAR-100	100	0.11191	0.3398	10.4519	Yes
ResNet56	CIFAR-10	10	0.06698	0.60298	0.6699	Yes
ResNet50	ImageNet	1000	0.1773	161.46	161.6373	Yes

were within 10% of the possible range, which indicates our pruning method was fair toward all classes.

Chapter 6

Conclusion & Future Works

6.1 Thesis Summary

In summary, we have developed a method to rank the importance of filters within a convolutional layer. CRank expands the concept of ranking feature maps by considering the class-by-class performance of each filter. CRank also automatically finds the pruning rate in each layer rather than requiring the pruning rates to be input parameters, which results in a fully automated pruning pipeline. In a head-to-head comparison with the state-of-the-art methods, such as the HRank pruning algorithm, our CRank techniques retain filters that perform significantly better than HRank in three cases and are tied in a fourth. Our method is empirically fair as it does not discriminate against any specific classes and is shown to be more efficient in power consumption than the unpruned model.

6.2 Future Works

This work aimed to have a measuring system to identify the most critical filters within a convolutional network. This idea can be further expanded beyond CNN models and classification tasks. Nowadays, transformer architectures are replacing many RNNs, and CNN techniques and pruning in transformers is an active research topic [69]. We want to expand this work toward transformer models and see how what we have learned here can be modified to prune attention layers. Also, identifying critical filters has obvious uses in explainable

AI, which we intend to explore.

References

[1] A. F. Agarap, “Deep learning using rectified linear units (relu),” *arXiv preprint arXiv:1803.08375*, 2018.

[2] A. Aghasi, A. Abdi, N. Nguyen, and J. Romberg, “Net-trim: Convex pruning of deep neural networks with performance guarantee,” *arXiv preprint arXiv:1611.05162*, 2016.

[3] A. Ahmadyan, T. Hou, J. Wei, L. Zhang, A. Ablavatski, and M. Grundmann, “Instant 3d object tracking with applications in augmented reality,” *arXiv preprint arXiv:2006.13194*, 2020.

[4] H. Akaike, “Use of statistical models for time series analysis,” in *ICASSP’86. IEEE International Conference on Acoustics, Speech, and Signal Processing*, IEEE, vol. 11, 1986, pp. 3147–3155.

[5] E. Baum and D. Haussler, “What size net gives valid generalization?” *Advances in neural information processing systems*, vol. 1, 1988.

[6] T. Bezdan and N. B. Džakula, “Convolutional neural network layers and architectures,” in *International scientific conference on information technology and data related research*, Singidunum University Belgrade, Serbia, 2019, pp. 445–451.

[7] R. E. Bradley, L. A. D’Antonio, and C. E. Sandifer, “Euler at 300: An appreciation,” *(No Title)*, 2007.

[8] M. A. Carreira-Perpinán and Y. Idelbayev, ““learning-compression” algorithms for neural net pruning,” in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pp. 8532–8541.

[9] Y. Chauvin, “A back-propagation algorithm with optimal use of hidden units,” *Advances in neural information processing systems*, vol. 1, 1988.

[10] M. Chetouani, V. Dignum, P. Lukowicz, and C. Sierra, *Human-centered artificial intelligence: Advanced lectures.* Springer Nature, 2023, vol. 13500.

[11] W. contributors, *Rectifier (neural networks)*, Wikipedia, The Free Encyclopedia, Accessed: 2023-05-03, 2023. [Online]. Available: `https://en.wikipedia.org/wiki/Rectifier_(neural_networks)`.

[12] J. Dastin, “Amazon scraps secret ai recruiting tool that showed bias against women,” in *Ethics of data and analytics*, Auerbach Publications, 2022, pp. 296–299.

[13] J. Denker, D. Schwartz, B. Wittner, *et al.*, "Large automatic learning, rule extraction, and generalization," *Complex systems*, vol. 1, no. 5, pp. 877–922, 1987.

[14] S. R. Dubey, S. K. Singh, and B. B. Chaudhuri, "Activation functions in deep learning: A comprehensive survey and benchmark," *Neurocomputing*, vol. 503, pp. 92–108, 2022.

[15] R. O. Duda and P. E. Hart, *Pattern recognition and scene analysis*, 1973.

[16] R. Duggal, C. Xiao, R. Vuduc, D. H. Chau, and J. Sun, "Cup: Cluster pruning for compressing deep neural networks," in *2021 IEEE International Conference on Big Data (Big Data)*, IEEE, 2021, pp. 5102–5106.

[17] L. Enderich, F. Timm, and W. Burgard, "Holistic filter pruning for efficient deep neural networks," in *Proceedings of the IEEE/CVF Winter Conference on Applications of Computer Vision*, pp. 2596–2605.

[18] G. Fang, X. Ma, M. Song, M. B. Mi, and X. Wang, "Depgraph: Towards any structural pruning," pp. 16 091–16 101, 2023.

[19] E. Forgy, "Cluster analysis of multivariate data : Efficiency versus interpretability of classifications," *Biometrics*, vol. 21, pp. 768–769, 1965.

[20] D. Freedman, R. Pisani, and R. Purves, "Statistics (international student edition)," *Pisani, R. Purves, 4th edn. WW Norton & Company, New York*, 2007.

[21] K. Fukushima, "Neocognitron: A self-organizing neural network model for a mechanism of pattern recognition unaffected by shift in position," *Biological cybernetics*, vol. 36, no. 4, pp. 193–202, 1980.

[22] D. Ghimire, D. Kil, and S.-h. Kim, "A survey on efficient convolutional neural networks and hardware acceleration," *Electronics*, vol. 11, no. 6, p. 945, 2022.

[23] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. MIT Press, 2016, `http://www.deeplearningbook.org`.

[24] Google, *A mobile object detector and text recognizer*, `https://lens.google/`, Blog, 2021.

[25] M. Groh, C. Harris, L. Soenksen, *et al.*, "Evaluating deep neural networks trained on clinical images in dermatology with the fitzpatrick 17k dataset," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2021, pp. 1820–1828.

[26] S. Han, X. Liu, H. Mao, *et al.*, "Eie: Efficient inference engine on compressed deep neural network," *ACM SIGARCH Computer Architecture News*, vol. 44, no. 3, pp. 243–254, 2016, ISSN: 0163-5964.

[27] S. Han, H. Mao, and W. J. Dally, "Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding," *arXiv preprint arXiv:1510.00149*, 2015.

[28] S. Han, J. Pool, J. Tran, and W. J. Dally, "Learning both weights and connections for efficient neural networks," *arXiv preprint arXiv:1506.02626*, 2015.

[29] S. Hanson and L. Pratt, "Comparing biases for minimal network construction with back-propagation," *Advances in neural information processing systems*, vol. 1, 1988.

[30] M. Hardt, E. Price, and N. Srebro, "Equality of opportunity in supervised learning," *Advances in neural information processing systems*, vol. 29, 2016.

[31] B. Hassibi and D. G. Stork, *Second order derivatives for network pruning: Optimal brain surgeon.* Morgan Kaufmann, 1993, ISBN: 1558602747.

[32] T. Hastie, R. Tibshirani, J. H. Friedman, and J. H. Friedman, *The elements of statistical learning: data mining, inference, and prediction.* Springer, 2009, vol. 2.

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

[34] Y. He and L. Xiao, "Structured pruning for deep convolutional neural networks: A survey," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 46, no. 5, pp. 2900–2919, 2024. DOI: 10.1109/TPAMI.2023.3334614.

[35] Y. He, J. Lin, Z. Liu, H. Wang, L.-J. Li, and S. Han, "Amc: Automl for model compression and acceleration on mobile devices," in *Proceedings of the European conference on computer vision (ECCV)*, pp. 784–800.

[36] Y. He, X. Zhang, and J. Sun, "Channel pruning for accelerating very deep neural networks," in *Proceedings of the IEEE international conference on computer vision*, pp. 1389–1397.

[37] D. O. Hebb, *The organization of behavior: A neuropsychological theory.* Psychology press, 2005.

[38] G. Hinton, O. Vinyals, and J. Dean, "Distilling the knowledge in a neural network," *arXiv preprint arXiv:1503.02531*, 2015.

[39] G. E. Hinton and D. Van Camp, "Keeping the neural networks simple by minimizing the description length of the weights," in *Proceedings of the sixth annual conference on Computational learning theory*, 1993, pp. 5–13.

[40] S. Hooker, N. Moorosi, G. Clark, S. Bengio, and E. Denton, "Characterising bias in compressed models," *arXiv preprint arXiv:2010.03058*, 2020.

[41] H. Hu, R. Peng, Y.-W. Tai, and C.-K. Tang, "Network trimming: A data-driven neuron pruning approach towards efficient deep architectures," *arXiv preprint arXiv:1607.03250*, 2016.

[42] Z. Huang and N. Wang, "Data-driven sparse structure selection for deep neural networks," in *Proceedings of the European conference on computer vision (ECCV)*, pp. 304–320.

[43] D. H. Hubel and T. N. Wiesel, "Receptive fields and functional architecture of monkey striate cortex," *The Journal of physiology*, vol. 195, no. 1, pp. 215–243, 1968.

[44] D. R. Hush and B. G. Horne, "Progress in supervised neural networks," *IEEE signal processing magazine*, vol. 10, no. 1, pp. 8–39, 1993.

[45] M. Z. Iqbal and A. G. Campbell, "Adopting smart glasses responsibly: Potential benefits, ethical, and privacy concerns with ray-ban stories," *AI and Ethics*, vol. 3, no. 1, pp. 325–327, 2023.

[46] L. Kaufman and P. J. Rousseeuw, *Finding groups in data: an introduction to cluster analysis.* John Wiley & Sons, 2009.

[47] C. T. Kien, *Skip connection and explanation of resnet*, `https://chautuankien.medium.com/skip-connection-and-explanation-of-resnet-afabe792346c`, Accessed: 2024-05-06, 2023.

[48] N. M. Kinyanjui, T. Odonga, C. Cintas, *et al.*, "Fairness of classifiers across skin tones in dermatology," in *International Conference on Medical Image Computing and Computer-Assisted Intervention*, Springer, 2020, pp. 320–329.

[49] A. Krizhevsky and G. Hinton, "Learning multiple layers of features from tiny images," 2009.

[50] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," *Advances in neural information processing systems*, vol. 25, pp. 1097–1105, 2012.

[51] B. Kumar, "Convolutional neural networks: A brief history of their evolution," *Medium*, 2021. [Online]. Available: `https://medium.com/appyhigh-technology-blog/convolutional-neural-networks-a-brief-history-of-their-evolution-ee3405568597`.

[52] Y. LeCun *et al.*, "Generalization and network design strategies," *Connectionism in perspective*, vol. 19, no. 143-155, p. 18, 1989.

[53] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, 1998.

[54] Y. LeCun, J. S. Denker, and S. A. Solla, "Optimal brain damage," in *Advances in neural information processing systems*, pp. 598–605.

[55] Y. LeCun, L. Jackel, L. Bottou, *et al.*, "Comparison of learning algorithms for handwritten digit recognition," in *International conference on artificial neural networks*, Perth, Australia, vol. 60, 1995, pp. 53–60.

[56] N. Lee, T. Ajanthan, and P. H. Torr, "Snip: Single-shot network pruning based on connection sensitivity," *arXiv preprint arXiv:1810.02340*, 2018.

[57] F.-F. Li, J. Johnson, and S. Yeung, *Convolutional neural networks for visual recognition*, CS231n course, Accessed: 2024-03-23, 2019. [Online]. Available: `https://cs231n.github.io/convolutional-networks/`.

[58] F.-F. Li, J. Johnson, and S. Yeung, *Transfer learning: Convolutional neural networks for visual recognition*, CS231n course, Accessed: 2024-03-24, 2019. [Online]. Available: `https://cs231n.github.io/transfer-learning/`.

[59] H. Li, A. Kadav, I. Durdanovic, H. Samet, and H. P. Graf, "Pruning filters for efficient convnets," *arXiv preprint arXiv:1608.08710*, 2016.

[60] Y. Li, K. Adamczewski, W. Li, S. Gu, R. Timofte, and L. Van Gool, "Revisiting random channel pruning for neural network compression," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2022, pp. 191–201.

[61] M. Lin, L. Cao, S. Li, *et al.*, "Filter sketch for network pruning," *IEEE Transactions on Neural Networks and Learning Systems*, vol. 33, no. 12, pp. 7091–7100, 2021, ISSN: 2162-237X.

[62] M. Lin, R. Ji, Y. Wang, *et al.*, "Hrank: Filter pruning using high-rank feature map," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2020, pp. 1529–1538.

[63] M. Lin, R. Ji, Y. Zhang, B. Zhang, Y. Wu, and Y. Tian, "Channel pruning via automatic structure search," *arXiv preprint arXiv:2001.08565*, 2020.

[64] S. Lin, R. Ji, C. Yan, *et al.*, "Towards optimal structured cnn pruning via generative adversarial learning," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pp. 2790–2799.

[65] F. Liu, W. Zhao, Z. He, *et al.*, "Improving neural network efficiency via post-training quantization with adaptive floating-point," in *Proceedings of the IEEE/CVF International Conference on Computer Vision (ICCV)*, Oct. 2021, pp. 5281–5290.

[66] Z. Liu, J. Li, Z. Shen, G. Huang, S. Yan, and C. Zhang, "Learning efficient convolutional networks through network slimming," in *Proceedings of the IEEE international conference on computer vision*, pp. 2736–2744.

[67] Z. Liu, P. Luo, X. Wang, and X. Tang, "Deep learning face attributes in the wild," in *Proceedings of the IEEE international conference on computer vision*, 2015, pp. 3730–3738.

[68] J.-H. Luo, J. Wu, and W. Lin, "Thinet: A filter level pruning method for deep neural network compression," in *Proceedings of the IEEE international conference on computer vision*, pp. 5058–5066.

[69] S. Ma, H. Wang, L. Ma, *et al.*, "The era of 1-bit llms: All large language models are in 1.58 bits," *arXiv preprint arXiv:2402.17764*, 2024.

[70] A. L. Maas, A. Y. Hannun, A. Y. Ng, *et al.*, "Rectifier nonlinearities improve neural network acoustic models," in *Proc. icml*, Atlanta, GA, vol. 30, 2013, p. 3.

[71] J. MacQueen, "Some methods for classification and analysis of multivariate observations," in *Proceedings of the fifth Berkeley symposium on mathematical statistics and probability*, vol. 1, Oakland, CA, USA, pp. 281–297.

[72] Z. Mariet and S. Sra, "Diversity networks: Neural network compression using determinantal point processes," *arXiv preprint arXiv:1511.05077*, 2015.

[73] W. S. McCulloch and W. Pitts, "A logical calculus of the ideas immanent in nervous activity," *The bulletin of mathematical biophysics*, vol. 5, pp. 115–133, 1943.

[74] W. McKinney, "Data structures for statistical computing in python," in *Proceedings of the 9th Python in Science Conference*, vol. 445, Austin, TX, pp. 51–56.

[75] F. Meng, H. Cheng, K. Li, *et al.*, "Pruning filter in filter," *arXiv preprint arXiv:2009.14410*, 2020.

[76] Meta, *New ray-ban meta smart glasses*, `https://about.fb.com/news/2023/09/new-ray-ban-meta-smart-glasses/`, Accessed: January 29, 2024, Sep. 2023.

[77] R. Meyer and A. Wong, "A fair loss function for network pruning," *arXiv preprint arXiv:2211.10285*, 2022.

[78] M. C. Mozer and P. Smolensky, "Skeletonization: A technique for trimming the fat from a network via relevance assessment," *Advances in neural information processing systems*, vol. 1, 1988.

[79] C. Murti, T. Narshana, and C. Bhattacharyya, "Tvsprune-pruning non-discriminative filters via total variation separability of intermediate representations without fine tuning," in *The Eleventh International Conference on Learning Representations*, 2022.

[80] V. Nair and G. E. Hinton, "Rectified linear units improve restricted boltzmann machines," in *Proceedings of the 27th international conference on machine learning (ICML-10)*, 2010, pp. 807–814.

[81] J. Park, S. Li, W. Wen, *et al.*, "Faster cnns with direct sparse convolutions and guided pruning," *arXiv preprint arXiv:1608.01409*, 2016.

[82] A. Paszke, S. Gross, F. Massa, *et al.*, "Pytorch: An imperative style, high-performance deep learning library," *Advances in neural information processing systems*, vol. 32, 2019.

[83] F. Pedregosa, G. Varoquaux, A. Gramfort, *et al.*, "Scikit-learn: Machine learning in Python," *Journal of Machine Learning Research*, vol. 12, pp. 2825–2830, 2011.

[84] J. Rissanen, *Stochastic complexity in statistical inquiry.* World scientific, 1998, vol. 15.

[85] F. Rosenblatt, "The perceptron: A probabilistic model for information storage and organization in the brain.," *Psychological review*, vol. 65, no. 6, p. 386, 1958.

[86] A. Rosenfeld, "Picture processing by computer," *ACM Computing Surveys (CSUR)*, vol. 1, no. 3, pp. 147–176, 1969.

[87] P. J. Rousseeuw, "Silhouettes: A graphical aid to the interpretation and validation of cluster analysis," *Journal of computational and applied mathematics*, vol. 20, pp. 53–65, 1987, ISSN: 0377-0427.

[88] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, "Learning representations by back-propagating errors," *nature*, vol. 323, no. 6088, pp. 533–536, 1986.

[89] O. Russakovsky, J. Deng, H. Su, *et al.*, "Imagenet large scale visual recognition challenge," *International journal of computer vision*, vol. 115, no. 3, pp. 211–252, 2015, ISSN: 1573-1405.

[90] A. Sankaran, O. Mastropietro, E. Saboori, *et al.*, "Deeplite neutrino: An end-to-end framework for constrained deep learning model optimization," *arXiv preprint arXiv:2101.04073*, 2021.

[91] L. Seyyed-Kalantari, G. Liu, M. McDermott, I. Y. Chen, and M. Ghassemi, "Chexclusion: Fairness gaps in deep chest x-ray classifiers," in *BIOCOMPUTING 2021: proceedings of the Pacific symposium*, World Scientific, 2020, pp. 232–243.

[92] C. J. Shatz, "The developing brain," *Scientific American*, vol. 267, no. 3, pp. 60–67, 1992.

[93] M. Shen, F. Liang, R. Gong, *et al.*, "Once quantization-aware training: High performance extremely low-bit architecture search," in *Proceedings of the IEEE/CVF International Conference on Computer Vision*, 2021, pp. 5340–5349.

[94] H. Simon, *Neural networks and learning machines*, 2009.

[95] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," *arXiv preprint arXiv:1409.1556*, 2014.

[96] C. to StackExchange, *When was the relu function first used in a neural network?* StackExchange, Accessed: 2023-05-03, 2019. [Online]. Available: `https://stats.stackexchange.com/questions/447674/when-was-the-relu-function-first-used-in-a-neural-network`.

[97] I. Sutskever, J. Martens, G. Dahl, and G. Hinton, "On the importance of initialization and momentum in deep learning," in *International conference on machine learning*, PMLR, 2013, pp. 1139–1147.

[98] C. Szegedy, V. Vanhoucke, S. Ioffe, J. Shlens, and Z. Wojna, "Rethinking the inception architecture for computer vision," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 2818–2826.

[99] H. Tanaka, D. Kunin, D. L. Yamins, and S. Ganguli, "Pruning neural networks without any data by iteratively conserving synaptic flow," *Advances in neural information processing systems*, vol. 33, pp. 6377–6389, 2020.

[100] C. Tran, F. Fioretto, J.-E. Kim, and R. Naidu, "Pruning has a disparate impact on model accuracy," *Advances in neural information processing systems*, vol. 35, pp. 17 652–17 664, 2022.

[101] S.-H. Tsang, *Review: Resnet - winner of ilsvrc 2015 image classification, localization, detection*, `https://towardsdatascience.com/review-resnet-winner-of-ilsvrc-2015-image-classification-localization-detection-e39402bfa5d8`, Accessed: 2024-05-06, 2023.

[102] S. University, *Pooling in convolutional neural networks*, Accessed: 2024-05-18, 2024. [Online]. Available: `http://deeplearning.stanford.edu/tutorial/supervised/Pooling/#:~:text=If%20one%20chooses%20the%20pooling,image%20undergoes%20(small)%20translations.`.

[103] V. N. Vapnik and A. Y. Chervonenkis, "On the uniform convergence of relative frequencies of events to their probabilities," *Measures of complexity: festschrift for alexey chervonenkis*, pp. 11–30, 2015.

[104] V. N. Vapnik, "Inductive principles of the search for empirical dependences (methods based on weak convergence of probability measures)," in *COLT'89: Proceedings of the second annual workshop on computational learning theory*, 1989, pp. 3–21.

[105] D. M. Vo, A. Sugimoto, and H. Nakayama, "Ppcd-gan: Progressive pruning and class-aware distillation for large-scale conditional gans compression," in *Proceedings of the IEEE/CVF Winter Conference on Applications of Computer Vision*, 2022, pp. 2436–2444.

[106] Z. Wang, C. Li, and X. Wang, "Convolutional neural network pruning with structural redundancy reduction," pp. 14 913–14 922, 2021.

[107] Z. Wang, X. Liu, L. Huang, *et al.*, "Model pruning based on quantified similarity of feature maps," *arXiv preprint arXiv:2105.06052*, 2021.

[108] D. S. Wilks, *Statistical methods in the atmospheric sciences.* Academic press, 2011, vol. 100.

[109] M. Wilson, *Your iphone will soon get apple's answer to google lens*, `https://www.techradar.com/news/your-iphone-will-soon-get-apples-answer-to-google-lens`, Blog, 2021.

[110] Y. Wu, D. Zeng, X. Xu, Y. Shi, and J. Hu, "Fairprune: Achieving fairness through pruning for dermatological disease diagnosis," in *International Conference on Medical Image Computing and Computer-Assisted Intervention*, Springer, 2022, pp. 743–753.

[111] H. Yang, Y. Zhu, and J. Liu, "Energy-constrained compression for deep neural networks via weighted sparse projection and layer input masking," *arXiv preprint arXiv:1806.04321*, 2018.

[112] J. H. Zar, "Spearman rank correlation," *Encyclopedia of Biostatistics*, vol. 7, 2005.

[113] F. Zhang, V. Bazarevsky, A. Vakunov, *et al.*, "Mediapipe hands: On-device real-time hand tracking," *arXiv preprint arXiv:2006.10214*, 2020.

[114] L. Zhang, Z. Wang, X. Dong, *et al.*, "Towards fairness-aware adversarial network pruning," in *Proceedings of the IEEE/CVF International Conference on Computer Vision*, 2023, pp. 5168–5177.

[115] C. Zhao, B. Ni, J. Zhang, Q. Zhao, W. Zhang, and Q. Tian, "Variational convolutional neural network pruning," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pp. 2780–2789.

[116] B. Zhou, Y. Sun, D. Bau, and A. Torralba, "Revisiting the importance of individual units in cnns via ablation," *arXiv preprint arXiv:1806.02891*, 2018.

[117] Z. Zhou, W. Zhou, H. Li, and R. Hong, "Online filter clustering and pruning for efficient convnets," in *2018 25th IEEE International Conference on Image Processing (ICIP)*, IEEE, 2018, pp. 11–15.

[118] S. W. Zucker and R. A. Hummel, *Receptive fields and the reconstruction of visual information.* New York University, Courant Institute of Mathematical Sciences, Computer ..., 1985.

Appendix A

Code Sections

A.1 Code Snippets

A.1.1 Controlling the data

The code snippet below is the basic building block to load the data and shows how we control the data so that later in the calculations, we can access a certain class of data. In particular, it is the first step from algorithm 1, which is denoted by

$$\textbf{Require:} I(c) \leftarrow \textbf{Per class batch of images} \tag{A.1}$$

This snippet is for CIFAR-10 and shows how we control the images of only one class to be fed to the network during the ranking generation phase and then move to another class, which will give us the class conditional approach that we want.

```
def get_same_index(target, label):
    label_indices = []

    for i in range(len(target)):
        if target[i] == label:
            label_indices.append(i)

    return label_indices

for run in range(0, 10):

    # Set params
    print("run is ", run)
    label_class = run  # runs for 1000 classes

    print('==> Preparing data..')

    pin_memory = True

    transform_train = transforms.Compose([
        transforms.RandomCrop(32, padding=4),
        transforms.RandomHorizontalFlip(),
        transforms.ToTensor(),
        transforms.Normalize((0.4914, 0.4822, 0.4465),
        // (0.2023, 0.1994, 0.2010)),
    ])
    transform_test = transforms.Compose([
        transforms.ToTensor(),
        transforms.Normalize((0.4914, 0.4822, 0.4465),
        //(0.2023, 0.1994, 0.2010)),
    ])

    trainset = CIFAR10(root=args.data_dir, train=True,
    // download=True, transform=transform_train)

    train_indices = get_same_index(trainset.targets, label_class)

    trainloader = DataLoader(
        trainset,
        batch_size=128,
        num_workers=2,
        pin_memory=pin_memory,
        sampler=torch.utils.data.sampler.
        //SubsetRandomSampler(train_indices)
    )

    testset = CIFAR10(root=args.data_dir,
    // train=False, download=True, transform=transform_test)
    testloader = DataLoader(
        testset,
        batch_size=128,
        shuffle=False,
        num_workers=2,
        pin_memory=True,
    )

    net = eval(args.arch)(compress_rate=compress_rate)
    net = net.to(device)
```
Code Snippet A.1: Controlling the data

A.1.2 Calculating The Rank

The code below focuses on only a part of the code that focuses on calculating the rank of feature maps. Its mathematical expression was introduced in section 3.2.3 with equation 3.20. This code is shown in algorithm 1. When the data is fed to the model through a forward hook, we capture the feature maps of the desired convolutional layer. Then, we start our calculations and get the matrix rank for each feature map. Finally, we take an average of those feature maps and assign it to the filter.

```python
def get_feature_hook(self, input, output):
    global feature_result
    global entropy
    global total
    a = output.shape[0]
    b = output.shape[1]
    c = torch.tensor([torch.matrix_rank(output[i, j, :, :]).item()
    // for i in range(a) for j in range(b)])

    c = c.view(a, -1).float()
    c = c.sum(0)
    feature_result = feature_result * total + c
    total = total + a
    feature_result = feature_result / total

def test():
    global best_acc
    net.eval()
    test_loss = 0
    correct = 0
    total = 0
    limit = 5

    with torch.no_grad():
        for batch_idx, (inputs, targets) in enumerate(trainloader):
            # use the first 6 batches to estimate the rank.
            if batch_idx >= limit:
                break
            inputs, targets = inputs.to(device), targets.to(device)
            outputs = net(inputs)
            loss = criterion(outputs, targets)

            test_loss += loss.item()
            _, predicted = outputs.max(1)
            total += targets.size(0)
            correct += predicted.eq(targets).sum().item()

            progress_bar(batch_idx, limit, 'Loss: %.3f |
            //Acc: %.3f%% (%d/%d)'
                         % (test_loss / (batch_idx + 1),
                         //100. * correct / total, correct, total))

if args.arch == 'vgg_16_bn':
    if len(args.gpu) > 1:
        relucfg = net.module.relucfg
    else:
        relucfg = net.relucfg

    for i, cov_id in enumerate(relucfg):
        cov_layer = net.features[cov_id]
        handler = cov_layer.register_forward_hook(get_feature_hook
    )
        test()
        handler.remove()

        if not os.path.isdir('./rank_conv/' + args.arch):
            os.mkdir('./rank_conv/' + args.arch)
        np.save('./rank_conv/' + args.arch + '/rank_conv%d'
        // % (i + 1) + '_Class_' + str(label_class) + '.npy',
                feature_result.numpy())

        feature_result = torch.tensor(0.)
        total = torch.tensor(0.)
```
Code Snippet A.2: Python Snippet For Calculating The Rank

A.1.3 Finding Unimportant Filters

Both of the code snippets below are Python implementations of algorithm 3, which gives us the CRank matrix. The first divides the data points into 4 different regions based on their ranks' average and standard. The reasoning on why we do this has been explained extensively in section 4.2 The 4 CRank Regions. The implementation is easy, and each point is compared with the average STD and average AVG to see which region it belongs to.

```
def region_maker(pd_datapoints_2d, x_divider, y_divider):
    regions = [[] for x in range(int(4))]

    filter_num_in_region = [[] for x in range(int(4))]
    for i, points in enumerate(pd_datapoints_2d.values):

        if points[0] <= x_divider and points[1] <= y_divider:
            regions[0].append(points)
            filter_num_in_region[0].append(i)

        elif points[0] > x_divider and points[1] < y_divider:
            regions[1].append(points)
            filter_num_in_region[1].append(i)

        elif points[0] > x_divider and points[1] > y_divider:
            regions[2].append(points)
            filter_num_in_region[2].append(i)

        elif points[0] < x_divider and points[1] > y_divider:
            regions[3].append(points)
            filter_num_in_region[3].append(i)
    return regions, filter_num_in_region
```

Code Snippet A.3: Region Maker Function

This second part of the code snippet focuses on the automatic compression setting, where in order to avoid setting the compression rates manually, we use clustering metrics and select a number of filters to be preserved. More specifically, in algorithm 3, the portion that this code covers is from

$$\textbf{for conv. layer l do} \tag{A.2}$$

until the end. Famous libraries such as Pandas and SciKit-Learn have been utilized to do this task. The details of the rationale behind this snippet can be found in section 4.2 and 4.3.

```
for LAYER_NUMBER in range(1, 56):
    # Dict of ranks gives me the avg rank of each filter.
    dict_of_ranks = Sor_dictionary()
    for i in range(num_class):
        dict_of_ranks.add(i, np.load(path_to_ranks + prefix +
        // str(LAYER_NUMBER) +'_Class_'+str(i)+'.npy') )

    dict_of_n_features = n_feature_per_filter()


    # Making Dict of n features to DataFrame (n == 10 or n ==
   1000)
    # Then turn it into pandas dataframe
    list_of_features = np.zeros((len(dict_of_n_features),
   num_class))
    for i in range(len(dict_of_n_features)):
        for x in range(num_class):
            list_of_features[i][x] = dict_of_n_features[i][x]
    list_of_features = np.array(list_of_features)
    pd_datapoints_nd = pd.DataFrame(list_of_features)
    # (n == 10 or n == 1000)

    # Normalizing Data `Column wise`
    # Using Sklearn MinMaxSacaler method
    scaler = preprocessing.MinMaxScaler()
    for i in range(len(pd_datapoints_nd.columns)):
        pd_datapoints_nd[i] =
        // scaler.fit_transform(pd_datapoints_nd[i]
        // .values.reshape(-1,1))

    # Use Pickle file for best K.
    # It was made in google collab. Set the Random State of KMeans
    to
    // 6 so the results do not vary a lot.
    # The Following Decides the number of K's and I am
    // reading them from a Pickle file
    with open(complete_pickle_filepath, 'rb') as handle:
        Best_K_in_layers = pickle.load(handle)

    # N-D Kmeans with 6 as random state, as it should be
    Km = KMeans(n_clusters = Best_K_in_layers[LAYER_NUMBER],
    // random_state=6, max_iter=10000).fit(pd_datapoints_nd)
    df_master = pd.DataFrame(pd_datapoints_nd.copy())
    df_master['data_index'] = pd_datapoints_nd.index.values
    df_master['cluster'] = Km.labels_


    # Making Distance Matrix to find n-D Medoids
    # (idea: CAN be replaced by max sil score in each cluster)
    list_of_medoids = []
    for i in range(Best_K_in_layers[LAYER_NUMBER]):
        temp_df = pd.DataFrame(df_master[df_master.cluster == i ])
        df_distance_matrix =
        // pd.DataFrame(distance_matrix(temp_df.values[:, :
    num_class],
        //temp_df.values[:, :num_class]),
        // index = temp_df.index, columns = temp_df.index)
        //list_of_medoids.append(np.array
        // (df_distance_matrix.index)
        // [np.argmin(df_distance_matrix.sum(axis=0))])


    # a flag to identify the medoids
    df_master['medoid'] = 0
    df_master.loc[list_of_medoids, 'medoid'] = 1

    # Switching to 2-D Space to find Star-Point and Visualize
    pd_datapoints_2d = pd.DataFrame(columns=['avgs', 'stds'])
    for i in range(len(pd_datapoints_nd.index)):
        pd_datapoints_2d.loc[i] = pd_datapoints_nd.loc[i].mean(),
        // pd_datapoints_nd.loc[i].std()

    # Normalize the 2D Space
    pd_datapoints_2d['avgs'] =
    // scaler.fit_transform(pd_datapoints_2d['avgs'].
    // values.reshape(-1,1)).reshape(pd_datapoints_2d.shape[0],)

    pd_datapoints_2d['stds'] =
    //scaler.fit_transform(pd_datapoints_2d['stds'].
    // values.reshape(-1,1)).reshape(pd_datapoints_2d.shape[0],)

    # adding more info to df_master
    df_master['avgs'] = pd_datapoints_2d['avgs']
    df_master['stds'] = pd_datapoints_2d['stds']

    new_x_divider = df_master['avgs'].mean()
    new_y_divider = df_master['stds'].mean()

    # making a list of filters with below AVG silhouette sample
    # values and adding those vals as data to master df
    silhouette_sample_values =
    // silhouette_samples(df_master.iloc[:,:num_class],
    // df_master.cluster)
    df_master['sil_samp'] = silhouette_sample_values

    # adding the mean silhouette of each cluster
    // as an info to each filter
    # designing cluster based "cluster mean sil"
    df_master['cluster_mean_sil'] = None


    # Critical part of the code, assign regions to each filter
    regions, filter_no_in_each_region =
    // region_maker(pd_datapoints_2d,
    // new_x_divider, new_y_divider)

    # Critical
    df_master['region'] = None
    for i in range(4):
        df_master.iloc[filter_no_in_each_region[i],[-1]] = i

    # To find medoids in region 1
    medoids_in_region_1 = np.array(df_master.query("medoid == 1 &
    // region == 1")['cluster'])


    # Getting the cluster assossiated with medoids in region one.
    # In the new temporary table, we will get the
    // mean on all sil samps,
    # Then will assign them to their respective
    // datapoints in the master
    for clus in medoids_in_region_1:
        temp_df = df_master.query('cluster == @clus &
        // region == 1')

        temp_mean = temp_df['sil_samp'].mean()

        df_master.loc[temp_df.index.values,
        // ['cluster_mean_sil']] = temp_mean



    # Comparing each datapoint sil samp to their clusters mean sil
    # If a datapoint has None as its cluster mean,
    // it means that datapoint is not in region 1,
    // which we will not keep it
    # TypeError is for comparinf None to float,
    // which is not important
    df_master['below_avg'] = None
    for i in df_master['data_index']:
        try:
            df_master.loc[i, 'below_avg'] =
            // df_master.loc[i, 'sil_samp'] <=
            // df_master.loc[i, 'cluster_mean_sil']
        except TypeError:
            pass

    final_staying_filters = []

    staying_datapoints =
    // df_master.query('below_avg == 1')['data_index'].tolist()
    staying_medoids = medoids_in_region_1.tolist()
    for elm in staying_datapoints:
        final_staying_filters.append(elm)

    for elm in staying_medoids:
        final_staying_filters.append(elm)

    final_staying_filters.sort()

    final_staying_filters = np.unique(final_staying_filters)
    dict_of_filters_to_keep.add( LAYER_NUMBER,
    final_staying_filters )

```

Code Snippet A.4: Automatic Filter Selection

A.1.4 Fine Tuning and Pruning

The code snippet below shows a portion of the fine-tuning step. The fine-tuning step is very similar to training, with the difference that we will set some of the weights to 0 and not allow them to update during fine-tuning. In the code, the dict of masks holds the number of filters that we will keep, and the rest will be pruned. This code is the implementation of 4 from the section 4.3.

```
for index, item in enumerate(params):

    if index == cov_id * param_per_cov:
        break

    if index == (cov_id -1) * param_per_cov:
        f, c, w, h = item.size()

        zeros = torch.zeros(f, 1, 1, 1).to(self.device)

        for i in range(int(f)):
            if i in dict_of_masks[cov_id]:
                zeros[i, 0, 0, 0] = 1.

            else:
                pass

        self.mask[index] = zeros  # covolutional weight
        item.data = item.data * self.mask[index]
```

Code Snippet A.5: Fine Tuning and Pruning