UNIVERSITY OF OKLAHOMA
GRADUATE COLLEGE

APPLIED HIGH-ORDER SINGULAR VALUE DECOMPOSITION FOR WEIGHT COMPRESSION AND EXPANSION IN DEEP NEURAL NETWORKS

A THESIS
SUBMITTED TO THE GRADUATE FACULTY
in partial fulfillment of the requirements for the
Degree of
MASTER OF SCIENCE

By

AUSTIN GRAHAM
Norman, Oklahoma
2019

APPLIED HIGH-ORDER SINGULAR VALUE DECOMPOSITION FOR WEIGHT COMPRESSION AND EXPANSION IN DEEP NEURAL NETWORKS

A THESIS APPROVED FOR THE
SCHOOL OF COMPUTER SCIENCE

BY

Dr. Christan Grant, Chair

Dr. Dean Hougen

Dr. Amy McGovern

Acknowledgements

Through the writing of this thesis, I have received a great amount of support from colleagues, friends, and family.

First and foremost I express sincere thanks to my advisor Dr. Christan Grant for his constant support, ideas, and unrelenting pressure to do the work of which he knew I was capable. You have provided me with a confidence in both research and practice that I firmly believe industry would not have given me. I would also like to thank my committee for their helpful comments and words of wisdom as I further refined this research and my own understanding of machine learning.

I would like to thank my labmates and friends who motivated me to finish this work, and provided a helping hand in formulating and designing my approach and experiments. In addition, I would also like to thank my mother and father who shared with me all the frustrations and successes of the past two years. I wouldn't be the man or the academic I am today without you both instilling in me a love for learning and sharing knowledge.

Lastly, I would like to thank Bryan Baker. At fourteen years old, I decided to take a computer course because I liked video games. I soon realized that I disliked making video games, but loved to solve hard problems and make machines think for me. If it wasn't for your patient teaching, I might still be a musician. You and I have shared a lot of success and a lot of failure, and I am absolutely grateful for the impact all of those experiences have had on me.

Table of Contents

List Of Figures

List Of Tables

Abstract

Complex deep learning objectives such as object detection and saliency, semantic segmentation, sequence-to-sequence translation, and others have given rise to training processes requiring increasing amounts of time and computational resources. Human-in-the-loop solutions have addressed this problem in several ways; one such pain point is model hyperparameter search. Common methods of parameter search have high time costs and require iterative training of several models. Several algorithms have been proposed to manipulate a neural network's architecture and alleviate this cost. However, these algorithms require tuning of parameters to achieve desired performance and provide little to no intuition as to how such a change may affect overall performance. In this thesis, I present EigenRemove and WeakExpansion for removal and addition of weights providing a human-in-the-loop solution to the architecture search problem in both classical feedforward and convolutional neural network layers. EigenRemove yields results comparable to or better than the more popular Minimum Weight Selection pruning strategy, producing final test accuracies increased by 2-3% at larger compressions on the VGG16 object detection network. WeakExpand is compared with a trivial Zero Weight Expansion approach, where new connections are assigned no weight. WeakExpand is shown to produce final test accuracies in VGG16 comparable to that of Zero Weight Expansion, while providing new trainable weights rather than the dead weights produced by Zero Weight Expansion. Finally, I propose heuristics outlining how a user may use WeakExpand and EigenRemove to have a desired effect based on the current state of their network's training.

Chapter 1

Introduction

In terms of ingenuity and adaptability, the human brain has proven to be amongst the most performant computational systems. Modern computers excel at making logical decisions based on sets of variables; however, acquiring understanding from the context surrounding an objective is not so deterministic. While structure and training procedures in modern deep learning often have biological or psychological inspiration (Glorot and Bengio 2010), the relationship between machine learning and human cognition is weak.

With deep learning continuing to tackle increasingly complex problems, exhaustive search methods that are often used to find a top performing model are becoming painstakingly expensive. Researchers have begun to develop *human-in-the-loop* solutions, algorithms where a user steps in to correct or guide a model during training, to address these issues. Instead of waiting for results of tens or hundreds of models to compare, a user can instead correct errors as they see them. Section 1.1.2 gives more detail on the various forms one may encounter such systems.

In addition to saving computational resources, human-over-the-loop solutions often seek to make algorithms mimic human behavior. Consider the coach of a basketball who needs a player to shoot free throws. He selects a player and quickly finds that the player does not shoot free throws well. He has two options: select another player, or tune the technique of the current player. Were

an exhaustive search approach to be taken in this analogy, the coach would select another player to shoot free throws, and continue to select new players until one is found with enough skill to perform in a game. However, in practice, a basketball coach would be more likely to attempt to correct the technique of the player already trained in the task. It is this behavior that human-*over*-the-loop approaches attempt to emulate.

In the following sections, I outline the motivation for the development of EigenRemove and WeakExpand, two methods for interacting with deep neural networks targeted towards human-in-the-loop-style processing.

1.1 Motivation

Research has yet to agree on a method to explain how a trained network understands its world (Gunning 2017). With no intuition as to where understanding may have gone wrong (or right for that matter), users often resort to a strategy of training several versions of models and selecting the best performer based on some heuristic, called *grid search*. These search processes have high cost in computational power and time. Algorithms automatically selecting hyperparameters have been proposed in several forms; however, interest in the effects of allowing manual human intervention during training has gained ground in the past few years. These human-in-the-loop algorithms typically assume expert users and focus on achieving optimal solutions at process end.

I seek to develop a human-in-the-loop-like approach that facilitates exploration of hyperparameters (architecture in particular) for non-experts, in the style of human-over-the-loop (Graham et al. 2017). This takes focus away from direct optimization and dives more into discovering effects layer sizes have on

final performance. It is undesirable to allow a change in architecture to damage the learning state of the network such that training must be entirely restarted. The training process would begin to resemble a modified grid search, defeating the purpose of human involvement.

In this section, I introduce the training structure of deep learning models to motivate the necessity of human-over-the-loop solutions and discuss the recent work on which my approach is based.

1.1.1 Deep Learning for Classification

Supervised machine learning for classification is the process by which a model learns to classify data points into a set of classifications by means of a set of labeled data points (Kotsiantis et al. 2007). These models can be one of many types including neural networks, support vector machines, and decision trees, each including several variants. Deep learning focuses on the construction of models to process data in its raw form (LeCun et al. 2015). In this thesis, I am focused on neural networks and the image-based convolution networks for their power and popularity in deep learning (Collobert and Weston 2008; Sutskever et al. 2014; Krizhevsky et al. 2012; He et al. 2016).

Supervised neural networks, like most supervised techniques, learn by first feeding several training data points from an external data source to an input layer, propogating the input through the network, calculating an error, adjusting parameters, and testing the new state with a set of unseen data points (Kotsiantis et al. 2007). The data set and choice of *hyperparameters*, the set of predetermined values influencing the behavior of training, are crucial in this learning method; while neural networks have seen wild success in research, deep neural networks are notoriously difficult to train (Glorot and Bengio 2010).

Generally, a network can be either overparameterized or underparameterized. To best illustrate both issues, I will give one such case where either of these problems may occur.

Data instances are accompanied with annotated labels known as *classes*. Depending on the data source and sampling strategy, classes within a training data set may reflect certain distributions. The distribution of classes available in a data set greatly affects the outcome of learning (Lee et al. 2017). Ideally, the training data will consists of an equal distribution of labels between all classes. However, in practical applications this is often not the case. If a data set is largely biased towards one class over another, the corresponding classifier will likely learn to classify the majority well, and the minority poorly (Ganganwar 2012). In this case, the network may appear to be underparameterized. A user would then be motivated to add more parameters (adjust the initial architecture), and restart the training process. While the error rate may decrease, it is very likely the network has become overparameterized and therefore perform poorly on the set of unseen data. Once again, a user may adjust the architecture and repeat the training process.

Algorithm 1 illustrates an iterative pattern of training a neural network.

Algorithm 1 Train Machine Learning Model

```
procedure trainModel(dataSet, parameters)
    model ← initializeModel()
    trainData, testData ← sample(dataSet)
    while testError > threshold & hasNotConverged(model) do
        trainOutput ← model(trainData)
        gradients ← calculateGradient(trainOutput)
        model.adjustWeights(gradients)
        testError ← model(testData)
    end while
    return model
end procedure
```

This process repeats until the performance meets standards of the user constructing the model, whether through manual or automatic testing procedures (Maclaurin et al. 2015). If a single training epoch takes a significant amount time, a metric relative to the objective being learned, these repetitions present a barrier to deploying powerful learning systems. Methods such as warm-start initialization have been proposed to lessen these burdens, for example the one by Tirumala et al. (2016). In addition, the data set may not include the full context necessary to understand the objective effectively either because of missing data or immeasurable key features. In these cases, human domain expertise may be leveraged to improve performance (Awasthi et al. 2014; Graham et al. 2017; Biswas and Jacobs 2014).

For these reasons, researchers in recent years have begun to develop human-in-the-loop systems to boost accuracy by allowing a user to inject domain expertise into the training process (Ankerst et al. 1999; Lad and Parikh 2014; Ueno et al. 2006). However, users are notoriously slow thinkers as compared to machines, and thus arises a tradeoff between time to deployment and accuracy of experimental models as discussed in Graham et al. (2017).

1.1.2 Human Interactivity

The investigation of the tradeoff between time and accuracy present in deep learning has a rich history in research manifesting in either one of two forms. The first focuses on improving accuracy on special cases and the second focuses on shortening training times while sacrificing performance. The first is appropriately called *purely interactive* or just *interactive* and the second *anytime*. Examples of works in either of these two areas is enumerated further in Chapter 2.

It may be useful to think of levels of interactivity as laying on a spectrum as illustrated in Figure 1.1. On the left side sit algorithms including stop-and-wait conditions; training continues until a point of confusion and ceases execution until a user can respond to a query for more information. An example of this is given in Biswas and Jacobs (2014). Learning may also be entirely user driven, where gradients are only calculated once a user has performed an action as in Amershi et al. (2012). Such constructions allow for detailed human feedback either by signaling incorrect classifications or providing reasoning for the classification of an outlier case (Amershi et al. 2012; Biswas and Jacobs 2014; Ankerst et al. 1999; Lad and Parikh 2014).

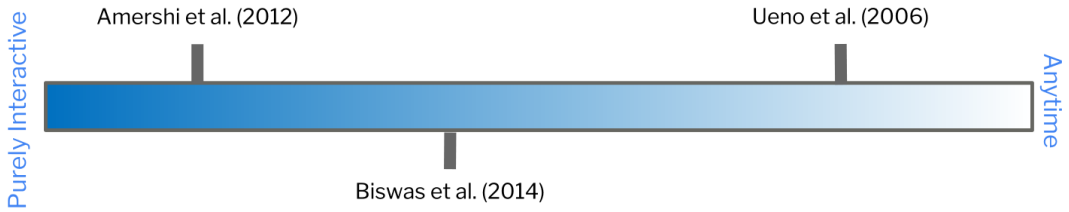


Figure 1.1: The Interactivity Spectrum. As one traverses from the left to the right, the user will lose control over the learning of an algorithm. That is, algorithms at the far left are event-based; computation is only performed on user action. Algorithms at the far right are fully automated; the user has no power to govern computation. Work done in Ueno et al. (2006) would lie towards the right, as the only interaction available is to stop the training process. Amershi et al. (2012) and Biswas and Jacobs (2014) are towards the left; an optimization step is only performed once a human has performed an interaction within their system.

On the right hand side of the spectrum are algorithms in which user interactions are limited to stop/continue signals. These algorithms are aptly named *anytime* due to the underlying assumption that training can be aborted once a usable solution has been reached, even though it may not be optimal. This concept has been applied to machine learning (Ueno et al. 2006), databases (Jermaine et al. 2008), and visualization (Lad and Parikh 2014) with the goal of obtaining a decent decision-making entity quickly.

Additionally, there are algorithms in the middle of the spectrum which balance time and accuracy. The primary goal is to boost the performance of the chosen model while minimizing increases in overall training time (limited increases are expected given high human think times). Such methods of interaction restrict human interactivity to signal actions, ensuring the targeted actions

can be meaningful to the understanding of the model. An example of this in Awasthi et al. (2014) involves pruning decision trees once fitting is complete, or in Crayons by Fails and Olsen Jr (2003) where the user "paints" classes onto images to produce incrementally better classifiers. A proposed phrase for this flow of interaction is *explanatory debugging* as listed in Kulesza et al. (2015).

Purely interactive algorithms, although proven to be effective in increasing performance, dilute the core essence of automation and machine learning. In many cases, they are not autonomous. For long learning processes (upwards of weeks or months), one cannot expect a user to sit in front of computer to monitor progress and answer questions. Anytime algorithms only allow interaction to stop training early, thus increasing the likelihood of deploying only sub-optimal solutions. Graham et al. (2017) propose *interruptble algorithms* defining the middle ground: allow users to interact if they are available, otherwise continue training as prescribed. Such algorithms lie in the middle of the spectrum and are expected to have the benefit of boosting accuracy while minimizing increases in overall run time.

1.2 Problem Statement

This thesis provides another answer to the primary research question: Can neural network architectures be adjusted during training in a manner that maintains performance? This question has been studied in the form of network weight pruning (Molchanov et al. 2016; Han et al. 2015), evolutionary computation (Yao 1999), and constructive training (Frean 1990). EigenRemove and WeakExpand address the first and third methods directly. Because the present approaches are focused on promoting human intervention to make adjustments

rather than automating the architecture search, I have left the exploration of evolving architectures for future work.

1.3 Proposed Approach

I present two primary operations allowing for the mutation of traditional feed-forward and convolutional neural networks by applying High-Order Singular Value Decomposition (HOSVD) in two ways:

1. **Layer Compression**: Removal of redundant information using weight matrix best-k approximation and minimal weight removal, called Eigen-Remove (Section 3.1.2).

2. **Layer Expansion**: Addition of new information using a strategy of re-distributing weights, called WeakExpand (Section 3.1.3).

Each is shown to be reasonable to compute during training as per the interruptable index as defined in Graham et al. (2017), and well approximates the activation state of the output layer mutated just after computation. Each approach is designed to accomplish the goal of minimizing the difference in activation states from before and after the operation. By taking this view, the amount by which learned weights are damaged is expected be minimal. Section 3.1 gives a formal statement of this goal and provides the theoretical foundations on which it is built. However, because these approaches are meant to be implemented in a human-in-the-loop system, thus the human aspect of computation must also be addressed.

In the event human behavior could be effectively simulated, a full analysis of the optimization abilities of the present approaches would be feasible [1]. The literature does not have this luxury; thus I aim to answer the research question by providing heuristics as to how both proposed approaches may perform in various scenarios rather than assessing direct effects on model optimization. These heuristics are determined from an analysis on the behavior of both approaches in isolated environments, and in a more practical application.

In an isolated three layer network, it is shown that EigenRemove prunes neurons with higher activation approximation error than the simpler Minimum Weight Selection discussed in Molchanov et al. (2016), but providing comparable final accuracy results that are slightly higher (by 3%). WeakExpand is shown to mimic the behavior of Zero Weight Expansion with the added benefit of producing new trainable parameters. Experimental design is given in Section 4.1.1 and detailed results in Section 4.2.1.

Both approaches are also evaluated in the context of the object detection problem. Total training accuracy and loss spikes are evaluated with VGG16 (Simonyan and Zisserman 2014) using the CIFAR-10 data set. This experimental control was chosen for its prevalence in deep learning research and its architecture consisting of both convolutional and fully connected layers, better enabling discussion on the effects of the operations throughout the architecture. It was found that EigenRemove, while causing larger spikes in loss during training, shows evidence of better generalization as a result of compression. In addition, WeakExpand has the same approximation ability as the trivial Zero Weight Expansion, while producing more trainable parameters.

[1] The construction of an interface to facilitate interaction and a series of usability tests may accomplish this; however these tasks are left for future work.

In addition, time to compute for WeakExpand and EigenRemove was recorded and measured against time for a single epoch and single forward pass of the network. It is shown that WeakExpand and EigenRemove take longer than a single forward pass, but have a significantly smaller time cost for a single epoch. In terms of human interaction with machine learning system, this is acceptable as per Graham et al. (2017). A theoretical discussion is presented in Section 3.1.1. Experimental design is given in Section 4.1.2 and detailed results in Section 4.2.2.

1.4 Organization of Thesis

In Chapter 2, I give a more detailed background of deep learning in neural networks including the feed forward and backpropogation algorithm, choice of activation functions, and convolutional variants. In addition, I will discuss both Singular Value Decomposition and its n-dimensional analog HOSVD.

In Chapter 3, I present a theoretical foundation for the intuition of using matrix decomposition for compression, the formalized activation approximation goal of both EigenRemove and WeakExpand, and the construction of both approaches along with discussions on their effectivenss at approximating outward activations.

In Chapter 4, I present empirical evaluations for the ability of both EigenRemove and WeakExpand to approximate outward activations both in isolated environments, and in the applied object detection network VGG16 accompanying an analysis of computational cost for both algorithms.

Finally in Chapter 5, I present conclusions and future work.

Chapter 2

Background

In this chapter, I present background information on both classical feed forward and convolutional neural networks, as well as operations involved in matrix decomposition.

2.1 Neural Networks

The Feedforward Artificial Neural Network shown in Figure 2.1 is *fully connected*, meaning each neuron holds a weight connected to every other neuron in the layer following. A primary benefit of this structure is to allow for learning of relationships from one weight in a layer to every other weight in the subsequent layer (Schmidhuber 2015).

The Convolutional Neural Network (CNN) (LeCun et al. 1998) depicted in Figure 2.2 grants this translational invariance by applying a series convolution and max-pooling operations across several layers around an input image to account for shift, scale, or distortion. This concept is discussed further in Section 2.1.2.

2.1.1 Classical Feed-Forward Neural Networks

The classical feed-forward neural network serves to classify sets of points with non-linear boundaries (Schmidhuber 2015). These networks consist of an input

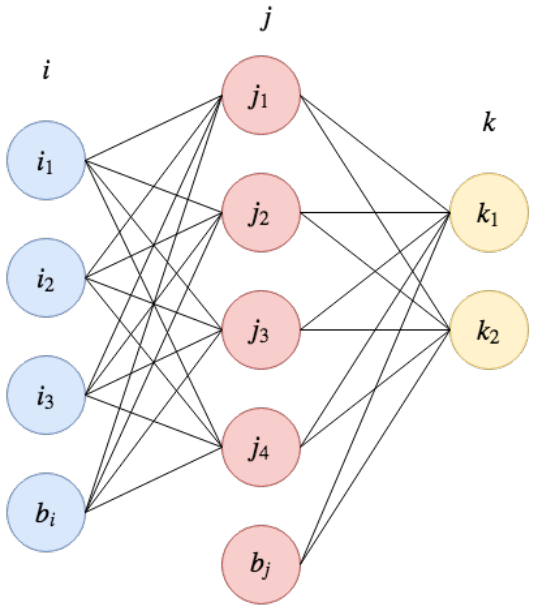


Figure 2.1: A simple feed-forward neural network with an input layer i, hidden layer j, and output layer k. Neurons b_i and b_j denote bias neurons fed into layers j and k respectively. Bias neurons will be further defined in Section 2.1.1

layer accompanied by an additional bias neuron, a set of hidden layers each with their own bias neuron, and an output layer. When constructing a network, several hyperparameters may be specified, including the learning rate, momentum, layer architecture, and activation function at each layer, and others outside of the scope of this thesis.

The output of a particular neuron is described by the dot product of its weight vector and the vector of incoming signals added with the bias signal, normalized by an activation function:

$$f(x) = \phi(\sum_i w_i x_i + b) \tag{2.1}$$

Equation 2.1 describes the output of a single neuron given its input weight set $\vec{w} = \{w_1, w_2, ..., w_{|\vec{w}|}\}$, the output of the neurons in the previous layer $\vec{x} = \{x_1, x_2, ..., x_{|\vec{x}|}\}$, the signal from the bias neuron b, all normalized by an

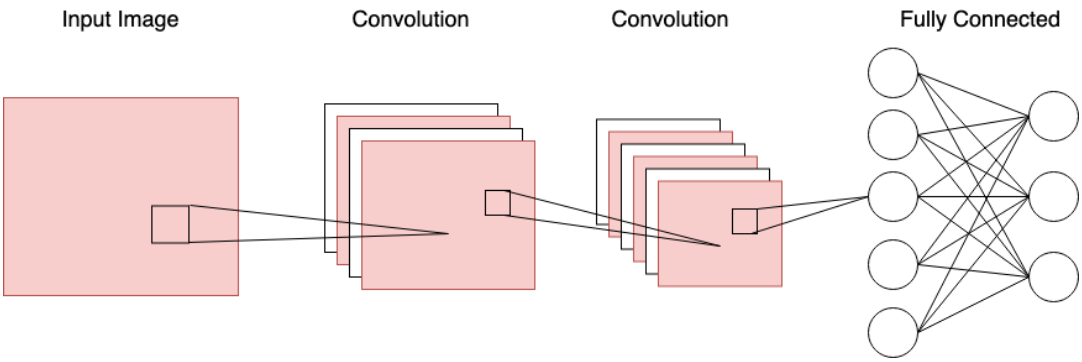


Figure 2.2: A Convolutional Neural Network with two convolution layers, followed by a fully connected layer (LeCun et al. 1998).

activation function $\phi(x)$. Note that this implies $|\vec{w}| = |\vec{x}|$. The term $\sum_i w_i x_i$ expresses a dot product between the weight and input vectors, thus the operation can be rewritten to encapsulate all units in the layer:

$$f_m(x) = \phi(\vec{x}W^T + \vec{b}) \tag{2.2}$$

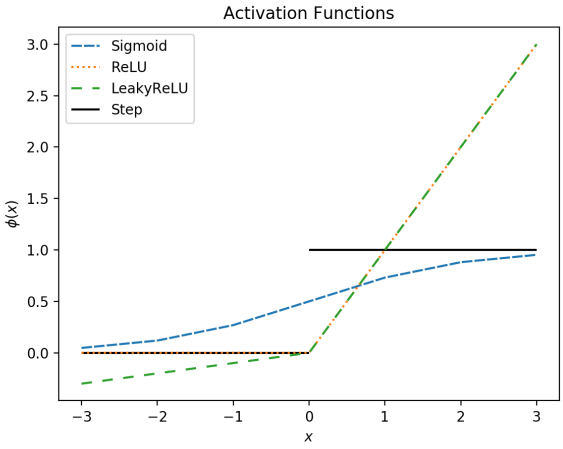
Equation 2.2 expresses the same process as Equation 2.1, but the sum of the weight and input elements is expressed as a matrix multiplication added with a vector of bias signals. While Equation 2.1 produces the output of a single neuron, Equation 2.2 produces a vector of outputs from the entire layer weight matrix W (input weights from all neurons in the layer) multiplied with all input vectors in X. The activation function $\phi(x)$ is performed element-wise. *Deep neural networks*, networks with several hidden layers, may contain millions of parameters; these matrices tend to grow quickly as objective complexity increases. Modern graphical processing units (GPUs) enable efficient matrix operations, making this formulation more desirable (Krizhevsky et al. 2012; Schmidhuber 2015).

2.1.2 Activation Functions

Several activation functions are in use by practitioners, the more popular of which are listed in Figure 2.3. McCulloch and Pitts (1943) formulated the artificial neuron, inspired by the biological structure from which it gets its name. These authors employ the step activation function, where a neuron emits zero when the linear combination of its inputs is below a threshold, and one when above. Rumelhart et al. (1988) discusses the formulation of backpropogation and the required derivative calculation on the activation function to learn parameters and best approximate the target function. The step function has zero derivative, preventing any error gradient from being propogated. Instead, nonlinear functions such as the sigmoid and ReLU produce non-zero gradients for learning weights, while also granting nonlinear classification power to networks (LeCun et al. 2012).

Figure 2.3 shows the shapes of the sigmoid, step, and ReLU functions. The value of the derivative of the sigmoid function approaches zero at the polar ends of the graph. This is evidence of the vanishing gradient problem, where saturated networks produce small gradients (Hochreiter 1998). The ReLU function does not exhibit this problem with positively saturated networks, however, negatively saturated networks will produce zero gradient [1].

[1]The hyperbolic tangent is left out of this discussion due to its similarity in principle to the sigmoid function with respect to vanishing gradient. LeCun et al. (2012) state that the hyperbolic tangent is often preferable to the traditional sigmoid due to its symmetric nature around the origin producing outputs on average close to zero



Activation Functions	
Name	Equation
Sigmoid	$f(x) = \frac{1}{1+e^{-x}}$
Tanh	$f(x) = tanh(x)$
ReLU	$f(x) = \begin{cases} 0 & x < 0 \\ x & x \geq 0 \end{cases}$
LeakyReLU	$f(x) = \begin{cases} \omega x & x < 0 \\ x & x \geq 0 \end{cases}$

Figure 2.3: Popular activation functions and their shapes listed in Glorot and Bengio (2010) and Karlik and Olgac (2011) plotted with the step function. The sigmoid and ReLU functions produce gradients that can be used in learning with backpropogation; however, the derivative of the sigmoid at the outer bounds approaches zero, thus saturated networks will produce very small gradients. ReLU will produce gradients of one with positively saturated weights, zero otherwise. LeakyReLU modifies ReLU to form $max(\omega x, x)$ to produce a usable gradient at negative saturations, where ω is the rate of decay.

2.1.3 Backpropogation

Once input has been fed through the network, it is necessary to quantify the error such that weights can be adjusted accordingly. While many such *loss functions* are in use, the cross entropy loss listed in Equation 2.3 is the primary choice for practitioners because it experiences less gradient disappearance typical in other candidate functions (Nar et al. 2019; Lin et al. 2017).

$$H(q) = -\sum_{c \in C} r_{q,c} \log p(r_{q,c}) \tag{2.3}$$

In the context of neural networks, the cross entropy loss is calculated on a set of output predictions q across a set of class labels C for an input instance in the training set. The result is the product of a known label $r_{q,c}$ for the output vector q and a single class label c multiplied with the log of the probability p of the predicted label for c being correct on q summed across all possible class labels in C.

The cross entropy loss is differentiable, enabling the use of gradient descent for learning weights. This method of weight adjustment forms relationships between weights after random initialization. The adjustment of a single weight can be made via a simple computation (LeCun et al. 1988):

$$w_i = w_i - \eta \frac{\partial E_{tot}}{\partial w_i} \tag{2.4}$$

Equation 2.4 adjusts a weight w_i by the learning rate η and a term expressing the total contribution to the final loss E_{tot} given by w_i. In other words, a single weight is adjusted based on its gradient with respect to the loss for the classification result. The learning rate η normalizes the gradient to avoid over-correction of weights during training (LeCun et al. 1988). The gradient $\frac{\partial E_{tot}}{\partial w_i}$ is calculated with respect to the chosen loss function.

Training networks is aligned with Algorithm 1 (discussed in Section 1.1.1) performed in a batch fashion. Data is partitioned into a *training* and *testing* set, where the training set is used to adjust weights over time, and testing to measure error of the trained network. The training set is further split into *batches*, or groups of examples to be fed at once, with the total gradient being computed on the set as a whole. This process is repeated over a parameterized number of *epochs*: the number of times over which the training data is iterated.

2.1.4 Convolutional Networks

Equation 2.2 has an important effect on the relation of weights in successive layers. The output of each weight in one layer is fed into each neuron in the subsequent layer; a single neuron is required for each pixel in the image, leading to large numbers of neurons needed for high definition images. For this reason, LeCun et al. (1998) developed the convolutional neural network LeNet that constructs more complex features from simple features through a series of convolutional layers in combination with pooling layers to provide *spatial invariance*, or the ability to detect a feature anywhere in the input image.

In a convolution layer, a series of kernels given a stride are defined with randomized weights. Each kernel is given dimensions k such that it can be represented by a $k \times k$ matrix. A series of kernels are initialized across the spread of the input image size, skipping a number of pixels defined by the stride. Thus the *feed* of an image is defined by the convolution between a kernel instance and its receptive field (LeCun et al. 1998).

An image can be defined as a series of channels. For example, a greyscale image is in one channel, versus an RGB image in three. With this in mind, a weight matrix for a given convolution layer can be defined as a four dimensional

tensor $W =< x, y, k, k >$, where x is the number of input channels, y the output channels, and k the selected kernel size. For example, we may define an input $m \times n$ RGB image as a three dimensional tensor $I =< 3, m, n >$; the color image has 3 channels, each of size $< m, n >$. To obtain the output activation map, the same process in Equation 2.2 is performed, but rather using the Kronecker product denoted by Equation 2.5 for an $m \times n$ matrix A and $p \times q$ matrix B (Kolda and Bader 2009). The output activation map (before pooling) will have dimensions $< y, m, n >$, having transformed the three input channels into the specified number of output channels.

$$A \otimes B = \begin{bmatrix} a_{11}B & \cdots & a_{1n}B \\ \vdots & \ddots & \vdots \\ a_{m1}B & \cdots & a_{mn}B \end{bmatrix} \tag{2.5}$$

A convolution operation is typically followed by a *max-pooling* operation, where the highest signal entity is kept for the output tensor (Tolias et al. 2015). This primarily functions for generalization, such that the network will not overfit to unimportant features while also maintaining translational invariance. In addition, this significantly reduces the number of parameters in the network and subsequently the amount of computation necessary to train.

2.1.5 Summary

In this section, I briefly described the computation of neural networks, feed forward and convolutional. A primary issue with such networks is the increasingly large number of trainable parameters as networks grow. In both cases these parameters can become sensitive to small details in the input. The number of hyperparameters that are necessary to tune a well-performing network grow

with objective complexity. Such issues have been addressed with applications of Principal Component Analysis, detailed in the next section.

2.2 Principal Component Analysis

In the computational world, an input instance may have thousands of examples with hundreds of features each, rendering manual inspection next to impossible. Principal Component Analysis (PCA), based on Singular Value Decomposition (SVD) solves this problem by using eigenvalue-eigenvector decomposition to find those features that contain the most information, given a set of input instances (Jolliffe 2011).

2.2.1 Singular Value Decomposition

For a given $n \times n$ matrix M, one may wish to obtain information about the high-level structure of M. M can be viewed as a vector space, and therefore we can perform a decomposition to gather knowledge on the geometric structure of the contents of M. Such a decomposition will result in a set of eigenvectors $X = x_1, x_2, ..., x_n$, and a set of eigenvalues $\Lambda = \lambda_1, \lambda_2, ..., \lambda_n$. Each vector in X describes an axis of the space spanned by M, and each value in Λ describes how each axis is transformed in the space described by M. It is known that M must be square for this to be computed (Dickinson and Steiglitz 1982).

Weight matrices in neural networks are rarely square; however SVD exploits the square nature of the product of a matrix and its transpose to obtain a similar geometric structure of M. Considering M is a real, $m \times n$ matrix, it can be decomposed in the following manner (Golub and Reinsch 1971):

$$M = U\Sigma V^T \tag{2.6}$$

The matrix U is an $m \times r$ matrix where each column is a left singular vector of M (equivalent to the the eigenvectors of $M^T M$). Similarly, V is an $n \times r$ matrix where each column is a right singular vector of M (equivalent to the eigenvectors of MM^T). Finally, Σ is a diagonal $r \times r$ matrix where values correspond to the singular values of M in descending order. Each of these values may also be described as the eigenvalues shared by both the eigenvectors of $M^T M$ and MM^T (Golub and Reinsch 1971).

The ordered nature of the singular vectors contained in Σ can be exploited to perform a best-k approximation M' of M. These approximation methods are most popular in collaborative filtering methods (Banweer et al. 2018; Zhang et al. 2005; Yam 1997). This approximation is done by removing all but the top k singular values in the diagonal of Σ, and re-multiplying to produce M'. M' will have the original dimensions of M with a reduced rank.

2.2.2 Dimensionality Reduction

Principal Component Analysis (PCA) uses the rank reducing properties of SVD to cast a vector from the space spanned by M to a smaller space made up of the principal components of M (Jolliffe 2011). As discussed above, SVD produces the matrix U where each column is an eigenvector of $M^T M$. $M^T M$ is recognizable as the covariance matrix of M; thus we can define the principal components to be $PC_M = U\Sigma$, or the result of reconstructing the covariance matrix of M. It can then be interpreted that each singular value in Σ measures the amount by which the corresponding eigenvector in U accounts for the variance in the rows of M. Thus, by removing small singular values, one can reduce the dimensionality of a data set losing minimal information (Pearson 1901). These

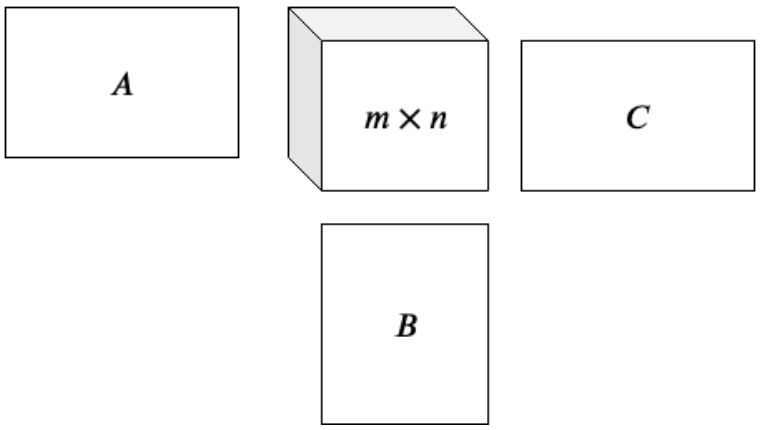


Figure 2.4: High-Order Singular Value Decomposition produces an $m \times n$ core, with factor matrices in each dimension describing the principal components of those modes. Shown here is the three dimensional case

properties make PCA popular for compression of images and streams (Du and Fowler 2007; Huang and Antonelli 2001; Taur and Tao 1996).

2.2.3 High-Order Singular Value Decomposition

High-Order Singular Value Decomposition (HOSVD), shown in Figure 2.4, is the process of applying SVD to multi-dimensional tensors (Kolda and Bader 2009). Performing this decomposition on an n-dimensional tensor χ yields a similar result to that of the two dimensional SVD:

$$\chi = \sigma \times_1 M_1 \times_2 M_2 \times_3 ... \times_N M_N \tag{2.7}$$

The symbol $\times_n$ denotes an n-mode product. The tensor σ is an n-dimensional tensor containing the *core* values of χ. This core provides the levels of interactions along dimensions, analogous to the way singular values define the *importance* of eigenvectors in a matrix transformation (Kolda and Bader 2009). Tensors $M_1, ..., M_N$ are the *factor matrices* of the decomposition, and contain

the principal components in each dimension (Kolda and Bader 2009). In many implementations, it is possible to decompose in one of several modes of the target tensor. Thus, HOSVD in the first two modes is equivalent to the SVD of the same matrix (Kolda and Bader 2009).

2.2.4 Summary

In the above sections, I cover the basics of SVD, PCA, and the multi-dimensional analog HOSVD. Weights in a neural network can be represented as matrices (or tensors); thus it may spark the question of how these processes may be applied for compression. PCA applied to neural networks isn't a new concept; however it is typically applied to network compression post-training to approximate large networks with smaller ones (Chen et al. 2015; Han et al. 2015). Next, I outline approaches EigenRemove and WeakExpand to modify neural network architectures during training.

Chapter 3

EigenRemove and WeakExpand

I will be proposing two methods: EigenRemove for the compression of layers and WeakExpand for the expansion of layers. Both methods make the assumption that these adjustments are made during training, i.e. there is still some training time remaining once the operation has concluded. The bounds on information loss for functions performed post-training are strict, as it is not desirable to damage the accuracy of a fully-trained model. In inter-training methods, this bound is loosened. As discussed in Graham et al. (2017), the expectancy of further training procedures allows for more information loss available for correction in future iterations.

In this chapter, I start with outlining the operations necessary for a layer to be modified and give a formal definition to the goal of approximating outward activation. I then present a proof that verifies the correctness of the approximation and motivates the use of SVD for compression. Finally, I present the formal EigenRemove and WeakExpand algorithms and discuss their complexity inline with the interruptable index.

3.1 Online Modifications

A single modification, as will be discussed in the following sections, is defined to be the tuple (C, γ), where C is the change (either a compression or expansion)

and γ is the amount by which to reduce/expand. These arguments are chosen by the user when they decide a change is necessary. Because users are making decisions that will affect learning outcomes, it is necessary to formulate C in a way that has minimal impact on the current state of learning.

Many human-in-the-loop solutions focus on improving performance. These solutions are restricted to prevent an interaction from impacting final outcomes. I have chosen to take an alternate view: instead of measuring impact of a modification of a layer on final performance, design a modification to best mimic the behavior of that layer in the instant after the modification is complete.

To do this, I take the target layer for modification in isolation with respect to the layer before and after it (forward and backward layers). For example, if layer j in the network in Figure 3.1 were to be compressed or expanded, both weight matrices W_{ij} and W_{jk} must be modified. The inputs to layer i are outside of this isolated scope, will not be modified, and thus the activation pattern in layer i is of no concern. However, the modification of W_{ij} can potentially change the pattern in which the neurons in j activate, and thus there exists a risk in changing the behavior of the latter portion of the network entirely.

In the modification of layer j, a primary objective is to not change the activation pattern of layer k, and therefore minimize the risk of changing the behavior of the network as a whole. Formally, we seek to find the following:

$$Y^* = \arg\min_{\hat{Y}} ||Y - \hat{Y}||_F. \tag{3.1}$$

That is, we want to find weight matrices W'_{ij} and W'_{jk} such that their replacements minimize the Frobenius norm of the difference of the original and altered resulting activation matrices with respect to the input (Molchanov et al. 2016). Y is the activation state of layer k and $\hat{Y}$ the activation state after the

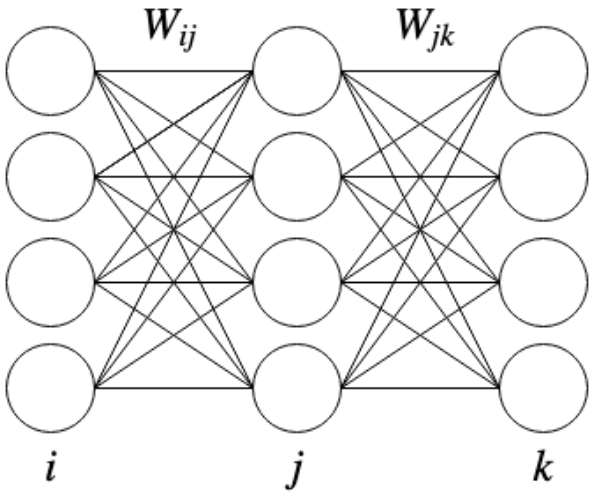


Figure 3.1: A small network with three layers composed of two $4x4$ weight matrices W_{ij} and W_{jk}.

modification is performed. Y^* is then the activation matrix determined from replacement weight matrices W'_{ij} and W'_{jk} that minimizes this difference. If operations are done post training, the classification pattern will be maintained. However, if done during training, backpropogation will adjust the weights as according to the new structure, ideally accounting for issues such as overparameterization or underparameterization.

In order to establish a framework for how either of these modifications will affect user experience, I will also evaluate the approaches in line with the interruptable index as formalized in Graham et al. (2017). The interruptable index $I_{A,C}$, where A denotes an algorithm and C a change to that algorithm, provides a boolean expression evaluating whether a particular change is "reasonable" with respect to the learning algorithm on which is executed. In the following sections, I describe the theoretical foundations for the formulation of

EigenRemove and WeakExpand and theoretically evaluate their performance in the context of neural network weight matrices.

3.1.1 Theoretical Foundations

I first establish the foundation that drives the theoretical discussions of EigenRemove and WeakExpand. The relation $||A||_F = \sqrt{\sum_i \sigma_i^2}$ serves as a portal from the Frobenius norm into the SVD of a matrix A, enabling discussions of how manipulating the singular values can affect the minimization expressed in Equation 3.1. The proof I give here for the sake of completeness:

Theorem 1. *Given a square matrix A, the following relationship between the Frobenius norm and singular values holds: $||A||_F = \sqrt{\sum_i \sigma_i^2}$.*

Proof. The squared Frobenius norm can be defined as the sum of the squared elements of an $n \times m$ matrix A as follows:

$$||A||_F^2 = \sum_{i=1}^{n} \sum_{j=1}^{m} a_{ij}^2$$

I will also define matrix multiplication $AB = C$ such that each element $c_{ij} \in C$ is the dot product of the ith row of A and the jth column of B.

$$c_{ij} = \boldsymbol{a}_i \cdot \boldsymbol{b}_j = \sum_{k=1}^{n} a_{ik} b_{kj}$$

The product of $A^T A$ can then be computed:

$$A^T A = \begin{bmatrix} \sum_{y=1}^{n} a_{y1}^2 & \cdots & \cdots & \cdots \\ \cdots & \sum_{y=1}^{n} a_{y2}^2 & \cdots & \cdots \\ \cdots & \cdots & \ddots & \cdots \\ \cdots & \cdots & \cdots & \sum_{y=1}^{n} a_{yn}^2 \end{bmatrix}$$

The trace of of A^TA, $tr(A^TA)$ is then:

$$tr(A^TA) = \sum_{x=1}^{n}\sum_{y=1}^{n} a_{xy}^2 = ||A||_F^2$$

Given the SVD of $A = U\Sigma V^T$:

$$tr(A^TA) = tr(U\Sigma V^T V\Sigma^T U^T)$$

Matrices U and V are orthonormal, thus $U^TU = V^TV = I$:

$$tr(U\Sigma V^T V\Sigma^T U^T) = tr(U\Sigma\Sigma^T U^T)$$

Grouping these matrices into U and $\Sigma\Sigma^T U^T$, the property $tr(AB) = tr(BA)$ can be leveraged to rearrange the expression:

$$tr(U\Sigma\Sigma^T U^T) = tr(\Sigma\Sigma^T U^T U) = tr(\Sigma\Sigma^T) = tr(\Sigma^T\Sigma)$$

Σ is a diagonal matrix, thus multiplying it by its transpose will square all items along the diagonal:

$$tr(\Sigma^T\Sigma) = \sum_i \sigma_i^2 = ||A||_F^2$$

□

Theorem 1 also serves as the foundation for rank reduction. The matrix Σ is diagonal, and contains the singular values of A in descending order. Each of these singular values is interpreted as describing how much of the variance in the row or column vectors of A is explained. Therefore, singular values close to zero serve little purpose in describing the structure of the matrix and can be set to 0. This will remove the left singular vectors in U and right singular vectors in V. U, Σ, and V^T can be multiplied to obtain a rank reduced estimation of A, A'. Equation 3.1 can be expressed in more general terms:

$$A^* = \arg\min_{A'} ||A - A'||_F \tag{3.2}$$

The goal of rank reduction is to find the matrix A^* estimating A with minimum norm difference. A threshold δ is defined such that $\sigma_k := 0 \quad \forall \sigma_k < \delta$. An obvious choice of δ to achieve an optimal A^* is 0; this will reconstruct the original A perfectly having removed no singular values. However, the rank has not been reduced. It becomes clear that higher deltas construct less optimal A^*s, although the reconstructed matrix will satisfy the reduced rank (Xue et al. 2013).

In the next two sections, I describe EigenRemove and WeakExpand, and discuss the effectiveness of minimization of Equation 3.1 in terms of Theorem 1 and how rank reduction minimized under Equation 3.2 plays a roll to improve weight removal selection.

3.1.2 EigenRemove

Much interest has been generated in approximating deep models with a models of fewer parameters. Recent work has focused on using traditional compression techniques to achieve this goal (Han et al. 2015; Chen et al. 2015). These methods optimize compression of fully trained models, thus they must maintain strict bounds on information loss once compression is done. EigenRemove is motivated by compression techniques of images using PCA (Du and Fowler 2007; Huang and Antonelli 2001; Taur and Tao 1996; Molchanov et al. 2016), wherein information loss is more tolerable. In fact, matrix-decomposition-based methods have been introduced to neural networks in several cases, primarily for the use of speedup (Jaderberg et al. 2014; Xue et al. 2013). For example, Xue et al. (2013) leverages the serial multiplicative nature of singular value decomposition to expand a single layer into a lower rank approximation of two layers. That is, a layer i is expanded into $i_1 = U$ and $i_2 = \Sigma V^T$.

3.1.2.1 Method

The concept of "neural network pruning" is used to incrementally remove weights from layers and allow training to correct for any error introduced (Molchanov et al. 2016). Several methods have been proposed for this task, many of which require holding extra memory to track activation, gradients, or masks. The method of minimum weight selection calculates the L_2 norm of a neuron's vectorized outgoing weights, and removes neurons with the lowest value (Molchanov et al. 2016). Zhu and Gupta (2017) discusses the effectiveness of pruning networks by forcing sparsity on weight matrices in this manner. The authors found that larger, sparse networks produce more performant models. However, these sparse weight matrices do not accomplish the primary goal of EigenRemove: to reduce the total number of weights present in a network. In fact, many of these approaches require extra memory overhead (Molchanov et al. 2016; Zhu and Gupta 2017; Han et al. 2015).

EigenRemove uses the approach of Psichogios and Ungar (1994) to improve upon Molchanov et al. (2016) without incurring any extra memory overhead. Psichogios and Ungar (1994) uses SVD to reduce the size of networks based on redundant information determined by the singular values of the weight matrix. The columns of the U matrix are the eigenvectors of W^TW, and the columns of the V matrix the eigenvectors of WW^T. The matrix Σ is the eigenvalues shared by these eigenspaces. The removal of small eigenvalues followed by the reconstruction of the matrix gives a low-rank approximation of the original matrix as discussed in the previous section. Sainath et al. (2013) also shows lesser rank weight matrices are more generalized; the rank of a matrix is related to the number of independent rows or columns, thus it can be interpreted that a lesser rank weight matrix models a greater proportion of neurons that consistently

contribute classification as opposed to a higher rank weight matrix, which is likely to contain more redundant weights.

Algorithm 2 EigenRemove Algorithm

```
1: procedure EigenRemove(W, rankRatio, newSize)
2:     core, factors ← hosvdDecompose(W, rankRatio)
3:     reduced ← reconstruct(core, factors)              ▷ Rank reduce W
4:     norms ← frobNorm(reduced)          ▷ Frobenius Norm along rows of W
5:     selected ← topK(norms)                    ▷ Select k = newSize weights
6:     return W[selected]                          ▷ Return high norm weights
7: end procedure
```

The minimum weight selection is performed as per Molchanov et al. (2016) and shown in Figure 3.2. Rather than selecting based on the L_2 norm of the original weight matrix, EigenRemove selects on the best-k approximated version of W, *reduced*, to produce W'. W' is assigned weights not selected for removal by minimum weight selection. Implementation for this algorithm can be found in Chapter 6, and a summarized version in Algorithm 2.

The previous section discusses how rank reduction with a proper heuristic choice of δ minimizes overall difference between activation matrices both before and after the rank reduction is performed. In EigenRemove, I have added the extra step of minimum weight selection, thus it is necessary to discuss its effect on the minimization step. The trivial selection for Y^* is similar to the selection of A^* as discussed in the previous section. $\delta = 0$ in rank reduction and removing no weights will yield an optimal Y^*. This result is trivial, as the weight matrix will remain the same. Thus a similar heuristic decision need to be made to determine how many weights are necessary to remove to obtain an appropriate Y^*.

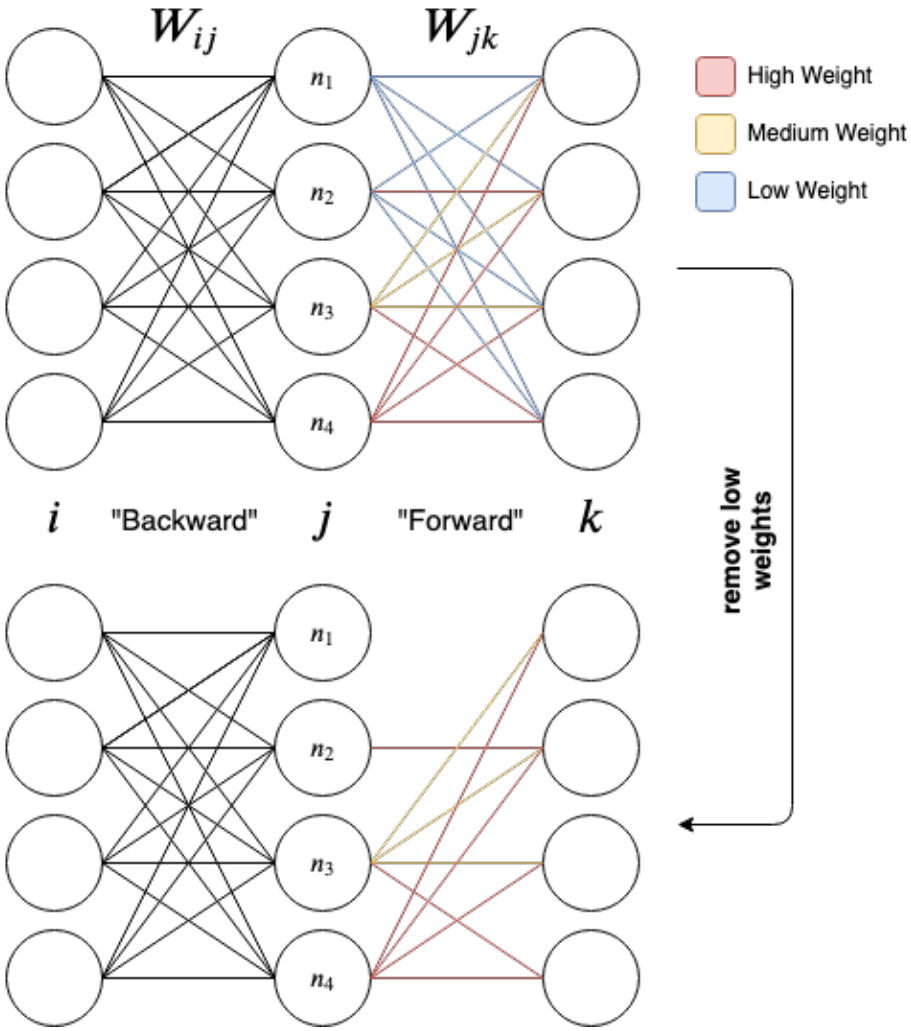


Figure 3.2: A modified version of Figure 3.1, where the output weights of layer j are labeled either red, yellow, or blue, corresponding to weights that are high, medium, or low respectively. On closer inspection, one can estimate that neurons n_1 and n_2 have the lowest average weight, and thus will be removed by minimum weight selection.

Sainath et al. (2013) show how low rank approximations damage summarization when rank reducing by increasing percentages of the original parameter set; however the authors (along with those in Xue et al. (2013)) do not discuss a rule by which to select a δ. Various values are explored and compared with studies in Sainath et al. (2013) in Chapter 4. A similar problem is encountered when selecting the number of weights to remove and is also discussed in Chapter 4. Is is worth noting that this method of selection may encounter trouble in regularized networks; these issues are saved for future work.

3.1.3 WeakExpand

WeakExpand addresses the opposite objective to EigenRemove: it may be desirable to expand a layer in such a way that the activation pattern of the forward layer is approximated, while guaranteeing the weights are assigned values meaningful to gradient calculation (a new weight assigned a value of 0 will not be trainable). Because the original weights are not being modified in this case, altering the transformation state of the original weights is of less concern. Detailed code can be found in Chapter 6, with an overview in Algorithm 3.

3.1.3.1 Method

Similar to EigenRemove, when adding a new node, connections to both the forward and backward layers must be established. In determining values for the new weights, it is of value to recall that a single feed is modeled in a summation, as in Equation 2.1. In order to achieve an optimal Y^*, a "division of power" strategy is introduced. The primary idea is to distribute strong neurons amongst new neurons and therefore break single, complex features into several simpler

Algorithm 3 WeakExpand Algorithm

```
1: procedure WeakExpand(W, newSize)
2:     weightAvgs ← means(W)                    ▷ Means along rows of W
3:     idxs ← argsort(weightAvgs)               ▷ Sort descending
4:     extend ← newSize / len(idxs)
5:     idxs ← repeat(idxs, extend)              ▷ Repeat idxs to newSize
6:     ratios ← 1/count(idxs)                   ▷ Ratios are 1/count of individual value
7:     return W[idxs] * ratios                  ▷ Repeat weights, multiply by ratio
8: end procedure
```

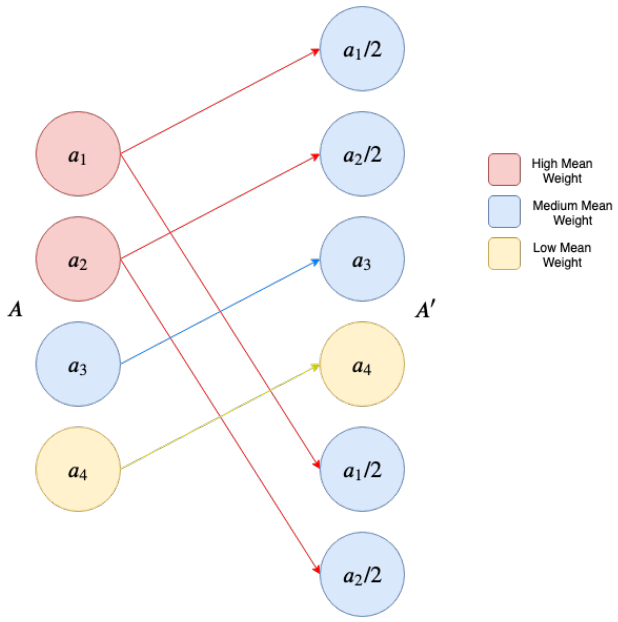


Figure 3.3: The layer A is expanded by two neurons into A'. WeakExpand distributes the two highest signal neurons (red) into four medium signal neurons (blue), each with a mean weight of half of its parent. Since the expansion only required two new neurons, the neurons a_3 and a_4 were not touched. The neuron placement is in line with how WeakExpand will create the new neurons, by appending new neurons to the end of the layer.

features. A neuron is considered stronger than another if it has a higher mean of output weights. Code examples can be found in Chapter 6.

In order to best discuss WeakExpand, I will also give a brief analysis of the trivial approach as discussed in the previous chapter appropriately named Zero Weight Expansion. This approach inserts new neurons with zero input and output weights. Thus, the weight set $\vec{w}$ becomes $\vec{w}' = \{w_1, w_2, ..., w_n, 0, ..., 0\}$. The input of the next layer from a zeroed neuron n_k with weight set $\vec{w}_{n_k} = \{0, 0, ..., 0\}$ can be expressed as $y = \sum_i 0 \cdot x_i$. It becomes obvious the output of this new neuron will be zero, and thus not affect the activation state of the latter layer. While this is an optimal Y^* in accordance with Equation 3.1, these new weights are untrainable and thus do not accomplish the final goal.

WeakExpand (depicted in Figure 3.3) instead distributes the current weights across the newly formed connections. When adding new parameters, neuron weights are distributed in order of decreasing mean output weight. To achieve an optimal Y^*, the weights are split in fractions equal to the number of times the neuron is duplicated. In Figure 3.3, the expanded layer A' has two more neurons the former A, thus the two highly signaling neurons a_1 and a_2 are each split in half once. Were this new size to be eight, or double the original, each neuron would be split in half and replicated. At triple the size, each weight set is split in thirds, and so on.

Consider the dot product expressing the feed value for a given neuron: $y = \sum_{\vec{w}} w_i x_i$ over a weight set $\vec{w}$ and input vector $\vec{x}$. Each weight $w_i \in \vec{w}$ can be decomposed into a sum of components of that weight: $w_i = \sum_{w_i} \omega_j$. y can then be expressed as $y = \sum_{\vec{W}} \sum_{w_i} \omega_j x_i$. Thus, splitting a weight in half to be included in the feed to the same neuron should have no affect on the output value of that neuron, thus achieving an optimal Y^*.

3.1.4 Complexity Analysis

Graham et al. (2017) present the interruptibility index $I_{A,U}$ to provide insight into how a particular change C to attribute U made during algorithm A may affect overall runtime of training. The primary idea is only execute changes that take significantly less time than the driving algorithm to minimize the additional time to train. The index $I_{A,U}$ expresses this idea in the form of the ratio $O(A)/O(C)$, or the hardness of C compared to the hardness of A. If this ratio is greater than one, then A is harder than C and is deemed *doable*. Otherwise, C is harder than A and the resulting extension in runtime is *wasteful*.

It's clear $I_{A,U}$ can be considered a heuristic at best. Graham et al. (2017) lacks formal definitions for A, U, and C. Thus, to analyze how the present approaches may affect the perception of runtime for a user, I give more formal definitions to these variables for a more complete analysis. Experiments validating this analysis are given in Section 4.2.2. While the values for U and C will become clear, the choice of A is less obvious. There are two primary options: either we consider the hardness against a single forward pass, or over an entire epoch. These approaches are centered in human-in-the-loop style algorithms, thus human think time must be taken into account. As will be shown in Section 4.2.2, a single forward pass is quick and thus if a change is double the time, a user is less likely to notice the increase. An entire epoch takes several minutes, thus a change taking double the time will be very noticeable to a user. A primary motivation for this work is to perform these actions in such a way that a user will experience minimal perceived delay, thus considering these changes on the scale of an epoch better accomplishes the overall goal of this work.

In the case of EigenRemove, U is the architecture of the network and C is the EigenRemove process. When computing the rank reduced $n \times m$ weight

matrix in EigenRemove, the TensorLy library (Kossaifi et al. 2019) provides partial computations based on the target rank, and thus this complexity is lessened to $O(r^2m)$ where r is the target computation rank and $r < n < m$. This complexity is compared with the complexity of doing a series of matrix multiplications: in an l-layer network, there are $l-1$ matrix multiplications of complexity approximately $O(n^{2.87})$ for an $n \times n$ square matrix (considering the Strassen matrix multiplication algorithm on a single node (Coppersmith and Winograd 1990)). This is of total complexity $O(ln^{2.87})$. If an epoch involves γ forward passes, the complexity becomes: $O(\gamma ln^{2.87})$. For the heuristic $I_{A,U} = O(A)/O(C) > 1$ to hold true, γ and l need to be large; in a more formal sense, $\gamma * l > m$ for this change to be reasonable. This is a borderline result, thus it would be advisable to use EigenRemove with very large networks with large epochs.

WeakExpand is a simpler case. It involves an estimated $O(n \log n)$ sort, in addition to an $O(n)$ replication process. This is in total $O(n \log n)$, thus $I_{A,U} = O(n^{2.87})/O(n \log n) > 1$ and this change is reasonable to perform during training. This a pronounced results, and one would expect to see WeakExpand take significantly less time to compute than a full epoch.

3.2 Summary

In this chapter, I presented the EigenRemove and WeakExpand algorithms for the modification of neural network architectures. The use of HOSVD in the removal process as well as dimensional blindness in the expansion process make both algorithms applicable to feedforward and convolutional layers in CNNs.

Next, I present experiments that validate these methods both in an isolated and applied environment.

Chapter 4

Experiments and Results

In this chapter, I present experiment design and results validating the claims made in Chapter 3.

4.1 Experiment Design

Both approaches are verified in two ways: in isolation to evaluate performance in approximating outward activations, and in application with the object detection network VGG16. Experiments are written using PyTorch in Python 3.7, and run on a GeForce GTX 1060 GPU and Intel i7 CPU with 64GB of RAM.

4.1.1 Isolation Experiments

The primary goal of the isolation experiments is to evaluate the immediate effect each operation has on outward activation, and thus measure the algorithm's effectiveness in estimating Y^*. For each algorithm, a three layer network was constructed with uniformly randomized weights (the default behavior of PyTorch); the input layer consisted of 2048 neurons, the hidden with an initial 2048, and the output with 2048. The hyperbolic tangent activation function is chosen for its prevalence in practice (LeCun et al. 2012). These values are chosen to best evaluate both approaches with layer sizes akin to those found in large networks more commonly used in practice.

A target operation C is chosen as either EigenRemove or WeakExpand. The hidden layer size changes with respect to C, either increasing the number of neurons linearly by 204 or decreasing linearly by 204 (growing/shrinking by approximately 10% of the original size). Each iteration consisted of feeding the original architecture 2048 input vectors from a uniform distribution, executing C on the hidden layer, then feeding the input again. The mean and variance of the squared Frobenius norm of the distance between the activation states of the output both before and after C is used to measure error. The common significance measure of 0.05 was chosen. The focus of these experiments is measuring the activation behavior of an output layer rather than the direct classification performance, therefore uniformly random inputs suffice to hold constant through each iteration.

4.1.2 Application Experiments

It is necessary to verify if this same behavior is present in an applied environment. To do this validation, EigenRemove and WeakExpand are applied while training VGG16 (Simonyan and Zisserman 2014) on the CIFAR-10 object detection data set. VGG16 and other VGG-style networks are a popular benchmarking network that has been used to test various compression techniques including those by Alippi et al. (2018) and Qassim et al. (2018). Final test accuracy is used to measure impact on performance, and the factor by which the current loss increases post-operation is used to measure immediate impact on training. For both experiments, VGG16 is trained fully five times with average final accuracy and median loss increase reported. To evaluate statistical significance, data sets are tested for normality followed by an appropriate hypothesis test

for difference between results of proposed approaches and alternate approaches. The common significance threshold of 0.05 is used throughout these tests.

The VGG16 network contains eleven total convolution layers and four total fully connected layers. Convolution layers increase in channels from three (input is RGB) to a total of 512; each layer has a 3×3 kernel with a stride and padding of one. Fully connected layers downsample from 2048 to 10 output neurons with ReLU activation and LogSoftmax output. BatchNorm is applied at each layer. In each experiment, the network was fully trained over fifty epochs five separate times, each at a different compression/expansion size. At each iteration, a layer was either compressed or expanded by factors of three quarters, two thirds, one half, one third, and one fourth (i.e. the layer was compressed to three fourths its original size, or expanded to one and three quarters its original size). A random sequence of layers to modify was generated and held constant across all experiments in order to discount behaviors caused by a change in one layer significantly affecting another, and thus causing unnecessary variance during training. Modifications follow this order, generated randomly: c10, c7, c2, f2, c9, c5, c6, c4, c3, f1; where c represents and convolution layer, f represents a fully connected layer, and the number following is the particular layer in the architectural sequence.

In addition, I also measure total epoch time against the time for a single operation, either EigenRemove or WeakExpand, to validate the timing analysis discussed in Section 3.1.4.

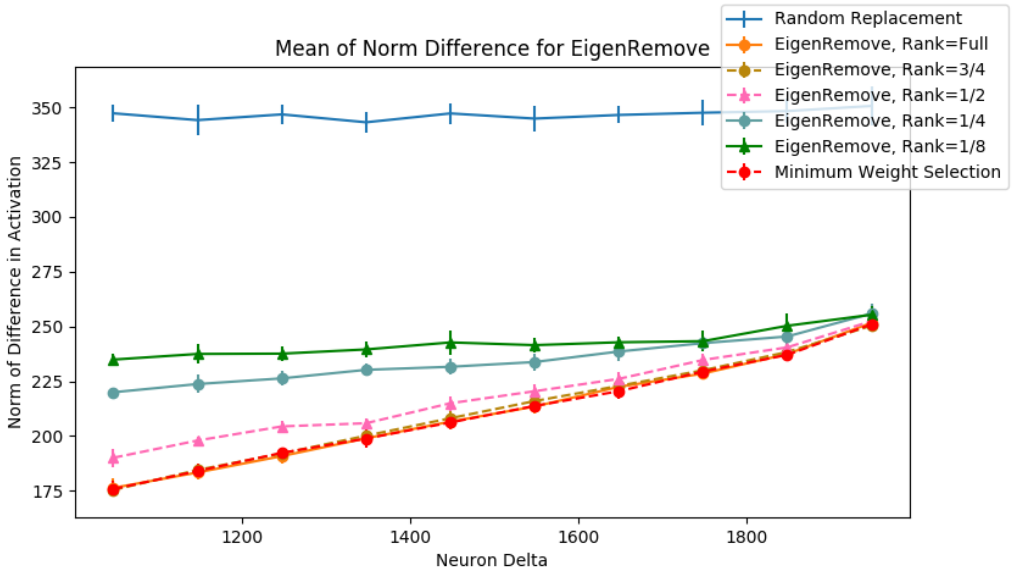


Figure 4.1: Mean and variance of the norm difference across ten iterations for each size of the EigenRemove algorithm. Minimum weight selection and EigenRemove at full rank have the same performance, with each level of rank reduction causing slightly more error as discussed in Section 3.1.2. However, one will notice the flattening of the curve as rank is furthered reduced. This submits a conclusion that EigenRemove at higher degrees of rank reduction have a more stable effect on latter activation states of the network.

4.2 Results and Discussion

4.2.1 Isolation Experiments

Figure 4.1 shows the results for the EigenRemove isolation experiments. Since EigenRemove is sensitive to the rank by which the target weight matrix is reduced, the experiment was conducted over a series of five rank reduction ratios: the first keeping full rank, the second keeping three fourths of the rank, the third keeping half of the rank, the fourth keeping one fourth of the rank, and the fifth keeping one fifth of the rank. These are compared to minimum weight selection as outlined in Molchanov et al. (2016) in addition to total replacement of both forward and backward weight matrices with random weights sampled from a uniform distribution.

A trade-off becomes immediately clear: when compressing only a few neurons, full or nearly full rank approximations maintain the majority of information while lesser rank approximations are more prone to error. However, error increases quickly in fully and nearly full rank approximations as the amount of of compression increases. Conversely, lower rank approximations have a lesser slope, and therefore are more stable across compression amounts. This result leads to the formulation of the following heuristic to choose a δ, the amount by which to rank reduce with EigenRemove: Choose a high δ (> 0.5) when it is expected that alterations will be infrequent and contained. Choose a low δ (< 0.5) when it is expected that alterations will be frequent and drastic.

A high δ corresponds to how much of the rank should be preserved (i.e. $\delta = 0.75$ means to preserve 75% of the original rank). If a user is fine tuning an almost trained model with smaller, incremental compressions, there is less likelihood that a change will drastically damage learning as more information is

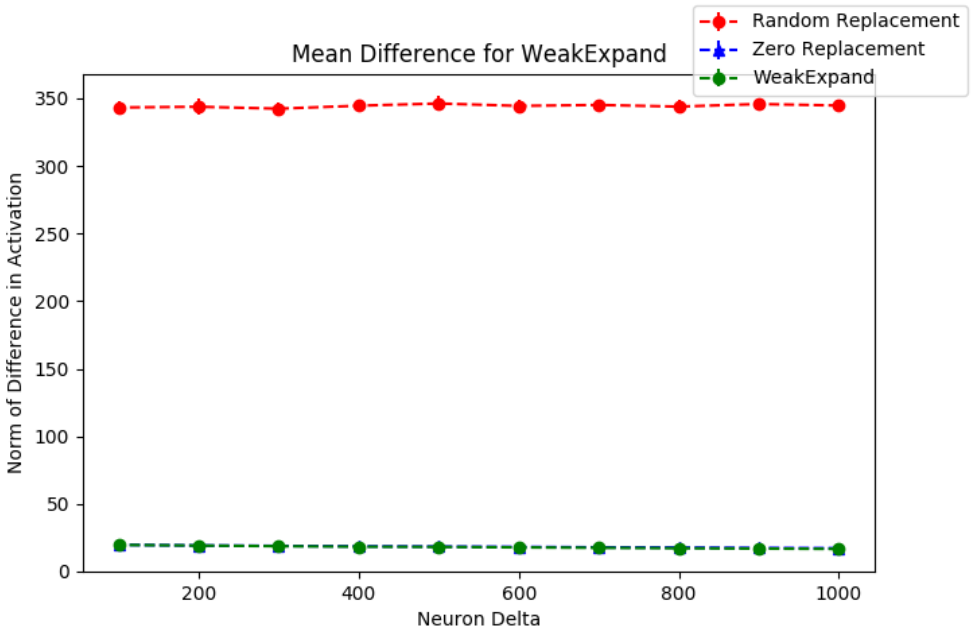


Figure 4.2: The mean and variational difference of the Frobenius norm between activations pre- and post-operation. This result confirms the behavior outlined in Chapter 3.2: both WeakExpand and Zero Weight Expansion do not alter the input signals to the next layer, thus the difference between activations will be zero.

preserved. However, if a user is towards the beginning of the training process, a higher error is more acceptable.

Another notable pattern is the apparent convergence of EigenRemove and Minimum Weight Selection at higher levels of compression. When enough weights are to be removed, the majority of information will be lost and thus the reduced rank weight matrix will begin to yield results similar to minimum weight selection (approaching a point of diminishing returns). This result is similar to that of the study in Sainath et al. (2013).

Figure 4.2 shows the results for the WeakExpand isolation experiments. The behavior discussed in Chapter 3.2 is clearly confirmed: The p-value calculated

was exactly 0, and therefore WeakExpand provides the same minimization as Zero Weight Expansion.

4.2.2 Application Experiments

To verify the explored behavior of EigenRemove, the five-sequence training strategy was run with both EigenRemove and minimum weight selection on the generated sequence of layers. To have more pronounced results, EigenRemove was performed with a 1/8 rank reduction ($\delta = 1/8$). Table 4.1 and Figure 4.3 shows the accuracy results of these experiments and Table 4.2 shows resulting factors of loss increase post-operation. The raw data can be found in the InterruptNet GitHub repository.

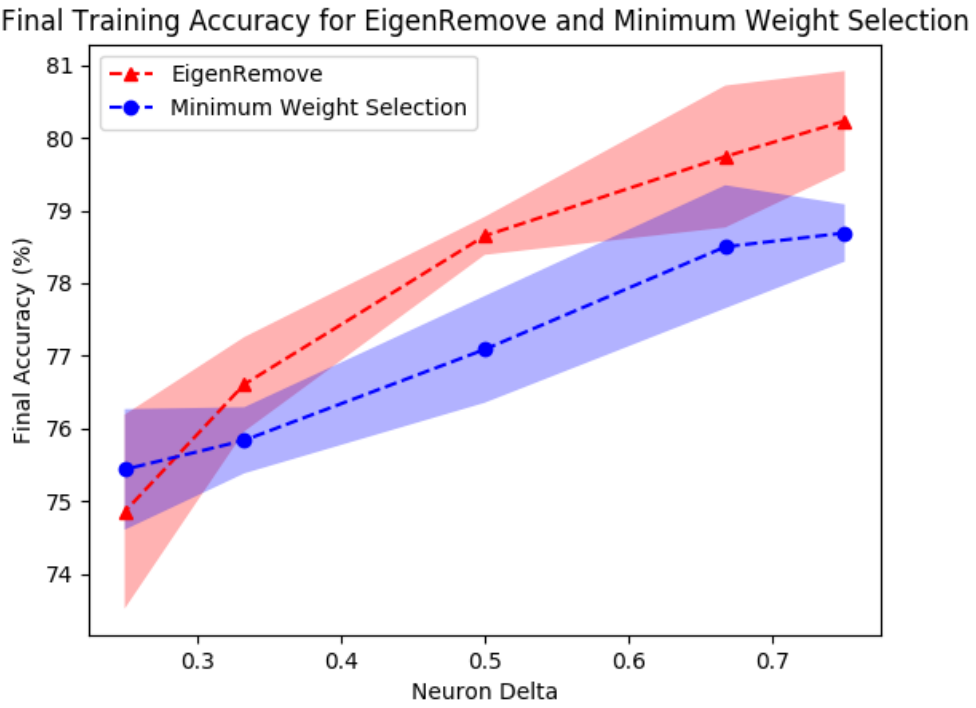


Figure 4.3: Final training accuracy between EigenRemove and Minimum Weight Selection on VGG16.

Final Training Accuracy					
Compression	1/4	1/3	1/2	2/3	3/4
EigenRemove (%)	74.86	76.61	78.65	79.74	80.23
Min Weight Select (%)	75.44	75.84	77.09	78.5	78.69

Table 4.1: The final average accuracy obtained after five iterations of training VGG16 over fifty epochs and applying compressions across layers in a randomly generated order. From left to right, more information is preserved after the compression, therefore it is less likely for error to be introduced.

The variance over the median loss increases using EigenRemove is 0.003, while the variance over the median loss for Minimum Weight Selection is 0.029. The behavior in the isolation experiments is mirrored here: Minimum Weight Selection causes overall less disruption in the loss. However, recall at large compression amounts all approaches converge; this is also shown in Table 4.1 and Figure 4.4: at the 1/4 and 1/3 ratios, the accuracy score is very close.

Figure 4.5 shows a histogram of the final accuracies obtained from training VGG16 while applying EigenRemove and Minimum Weight Selection with the given pattern. These values are not normal, thus significance testing is done with the non-parametric Mann-Whitney U test. The test yields a p-value of 2.229×10^{-06}, therefore the null hypothesis can be rejected and the two samples are significantly different.

Figure 4.6 shows a histogram of the loss increases obtained from training VGG16 while applying EigenRemove and Mininum Weight Selection with the given pattern. These values are approximately normal, therefore the Student's t-test can be applied. The test yields a p-value of .00002, therefore the null hypothesis can be rejected and the two samples are significantly different.

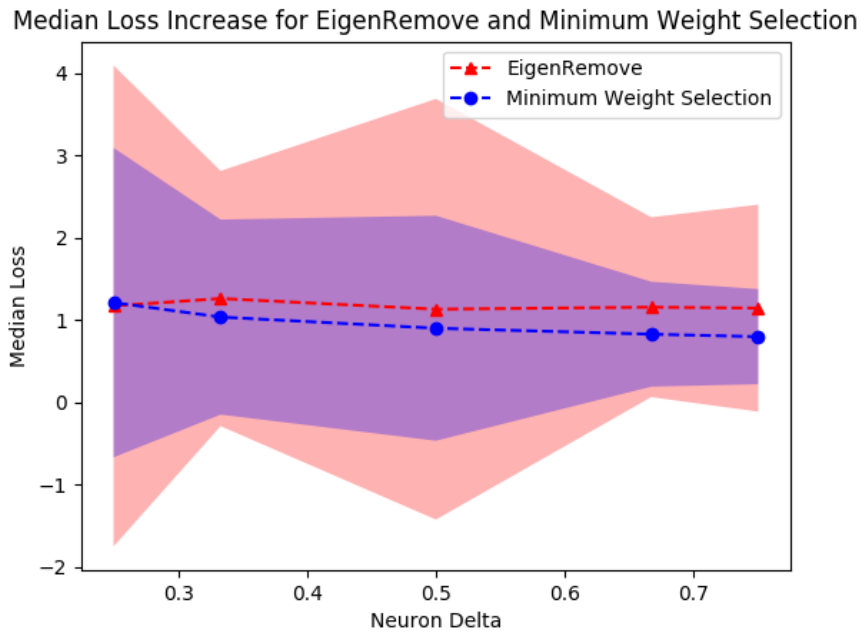


Figure 4.4: Graphic of the results in Table 4.2. The stability discussed in Section 3.1.1 can be seen here; As the neuron delta increases, EigenRemove remains more stable than Minimum Weight Selection, with more variability at each neuron delta.

Median Loss Increase					
Compression	1/4	1/3	1/2	2/3	3/4
EigenRemove	1.173	1.263	1.133	1.158	1.146
Min Weight Select	1.214	1.038	0.902	0.829	0.799

Table 4.2: The median factor by which the total training loss increased after an operation is performed over all layers in the generated list. As in the accuracy table above, loss decreases from left to right because more information is preserved towards the right of the table (less neurons are removed).

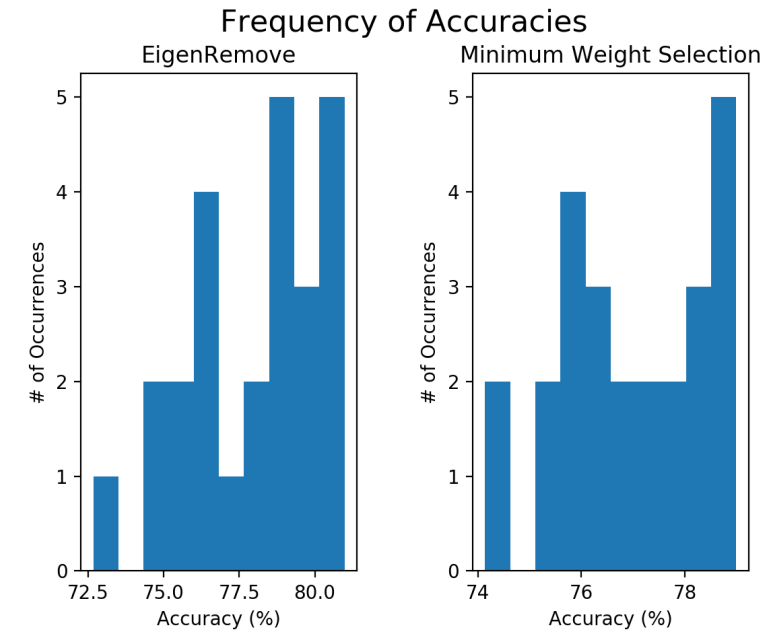


Figure 4.5: Histogram of final accuracies obtained from training VGG16 while applying EigenRemove and Minmum Weight Selection.

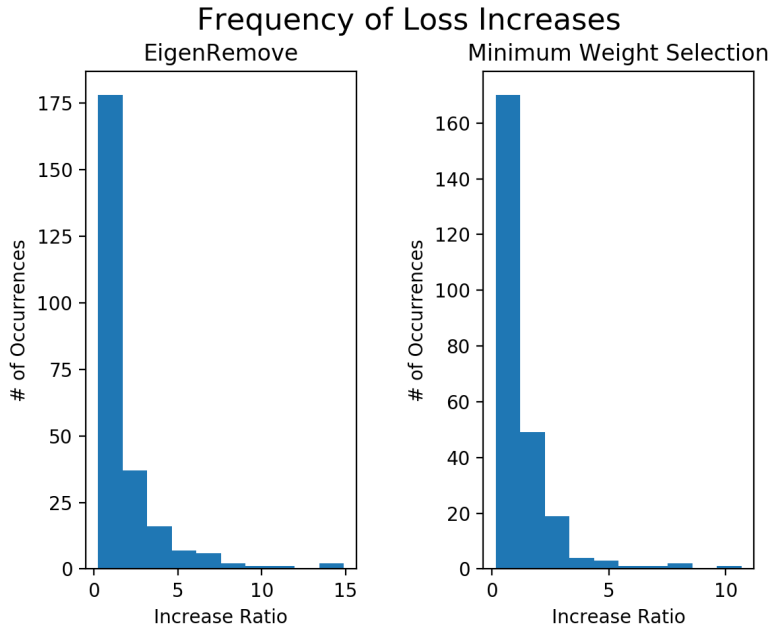


Figure 4.6: Histogram of loss increases obtained from training VGG16 while applying EigenRemove and Minmum Weight Selection.

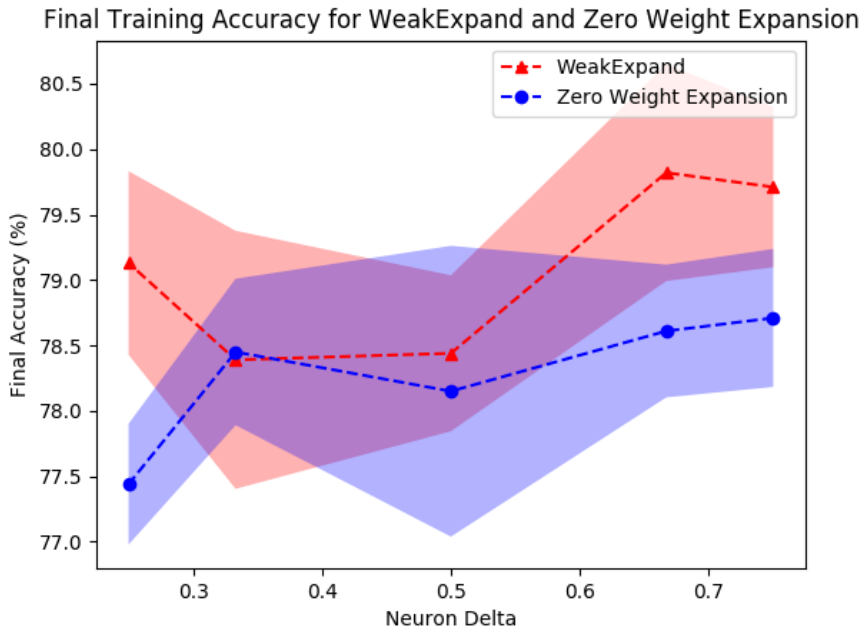


Figure 4.7: Final training accuracy between WeakExpand and Zero Weight Expansion on VGG16.

The same verification process was run for WeakExpand and Zero Weight Expansion. Each expansion ratio shows the percentage increase in neurons; for example a 100 neuron layer increased by 1/4 will result in a layer of 125 neurons. As Zero Weight Expansion does not add any values that can be trained, one should expect overall accuracy to be consistent across all expansion amounts. This behavior holds true in Table 4.3 and Figure 4.7.

WeakExpand adds more *trainable* parameters, and thus the overall accuracy is slightly higher. However, because weights are split based on the number of repetitions, a large increase will force many weights close to zero. Thus, one may encounter a "dead layer", where all weights are close to or exactly zero and feed will produce little to no output signals. There is evidence of this behavior in Table 4.3 and Figure 4.8, showing accuracy decreases at very large expansions.

Final Training Accuracy					
Expansion	1/4	1/3	1/2	2/3	3/4
WeakExpand (%)	79.13	78.39	78.44	79.82	79.71
Zero Weight Expansion (%)	77.44	78.45	78.15	78.61	78.71

Table 4.3: The final accuracy obtained after training VGG16 over fifty epochs and applying expansions across layers in a randomly generated order.

Because WeakExpand has the same minimization properties as Zero Weight Expansion, one should expect the loss increases to mirror that of Zero Weight Expansion. This is the behavior demonstrated in Table 4.2.

Median Loss Increase					
Expansion	1/4	1/3	1/2	2/3	3/4
WeakExpand	0.847	0.822	0.781	0.723	0.792
Zero Weight Expansion	0.803	0.862	0.714	0.773	0.789

Table 4.4: The median factor by which the total training loss increased after an operation is done over all operations in the generated list.

Figure 4.9 shows the average time in milliseconds for EigenRemove to execute. As was discussed in Section 3.1.4, EigenRemove has the possibility to take longer than a single epoch, if only by a few milliseconds. Thus, it can be concluded that while EigenRemove is reasonable, one should perform this operation sparingly.

Figure 4.10 shows the average time in milliseconds for WeakExpand to execute. As was discussed in Section 3.1.4, it is clear that WeakExpand takes significantly less time than a single epoch, by tens of milliseconds. Thus, it can

Figure 4.8: The median loss increase between WeakExpand and Zero Weight Expansion is very similar as is expected. It is worth nothing these methods do not address regularization, and thus adding trainable parameters with a regularized architecture like VGG16 can cause unwanted increases in loss.

be concluded that WeakExpand can be performed with little concern as a user of a system noticing the extra time.

Figure 4.12 shows a histogram of the loss increases obtained after applying WeakExpand and Zero Weight Expansion during training of VGG16. This data is not normal, thus the non-parametric Mann-Whitney U test. The test yields a p-value of 0.0869, therefore the null hypothesis is accepted and the two samples can be concluded to loosely be from the same distribution. This is expected, as theoretical analysis shows the two approaches to show nearly the same behavior.

Figure 4.11 shows a histogram of the accuracies obtained after applying WeakExpand and Zero Weight Expansion during training of VGG16. This

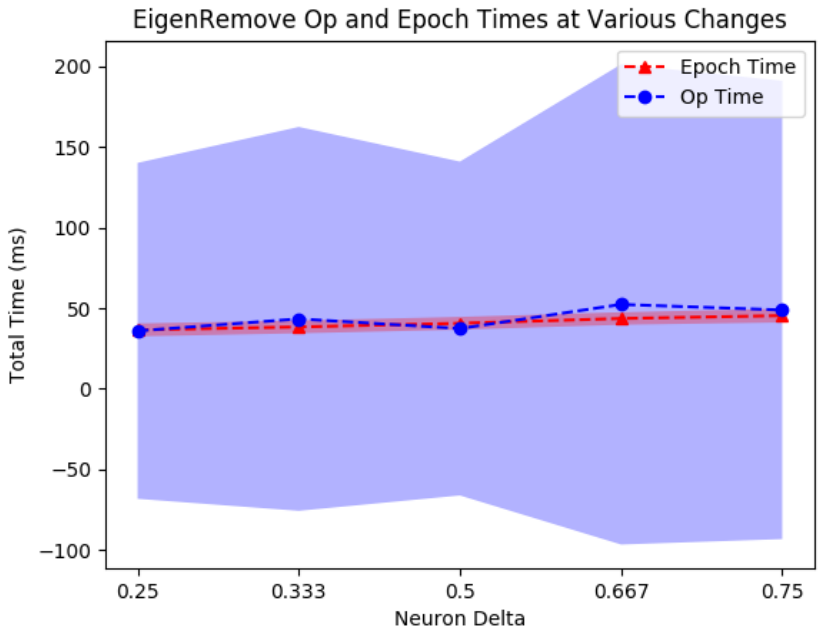


Figure 4.9: Average time for EigenRemove and single epochs across all compression amounts to execute. Note the high variability in EigenRemove; the base SVD operation in HOSVD is initialized or random, and can therefore have large effects on time for operation.

data is approximately normal, thus the Student's t-test is used for significance testing. The test yields a p-value of 0.0025, therefore the null hypothesis can be rejected and the two samples can be concluded to be significantly different.

4.2.3 Summary

In this section, I present experiments measuring EigenRemove and WeakExpand against the respective alternatives Minimum Weight Selection and Zero Weight Expansion. Experiments isolating single layers show that EigenRemove with high degrees of rank reduction (δ) has less variability in error over amount of compression, although producing more error than that of Minimum Weight

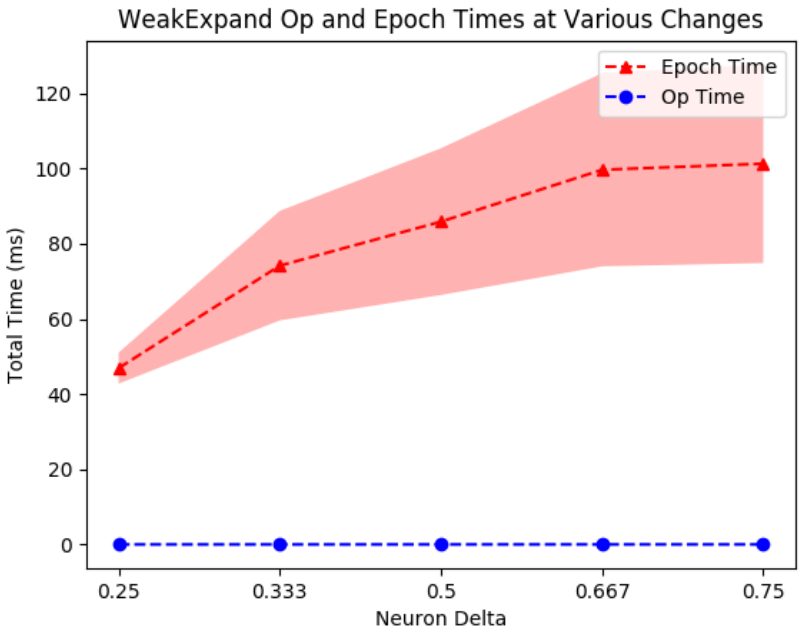


Figure 4.10: Average time for WeakExpand and single epochs across all expansion ratios to execute. Notice the op time appears to have zero error; as discussed in Section 3.1.4, the op time for WeakExpand is significantly less than the total epoch time, thus the error is close to zero relative to the total epoch error.

Selection. This has lead to a heuristic dictating EigenRemove with a high δ may be used when changes are expected to be frequent and minimal, while a low δ is more effective for infrequent and drastic changes. This behavior, and the resulting heuristic were confirmed by applying a series of operations on layers of VGG16 during training of the CIFAR-10 dataset.

WeakExpand was confirmed to match Zero Weight Expansion in activation approximation by distributing high weights across newly created neurons. The benefit of these new trainable parameters is shown by overall higher accuracy of VGG16 after a series of expansions over the generated series of layers. However,

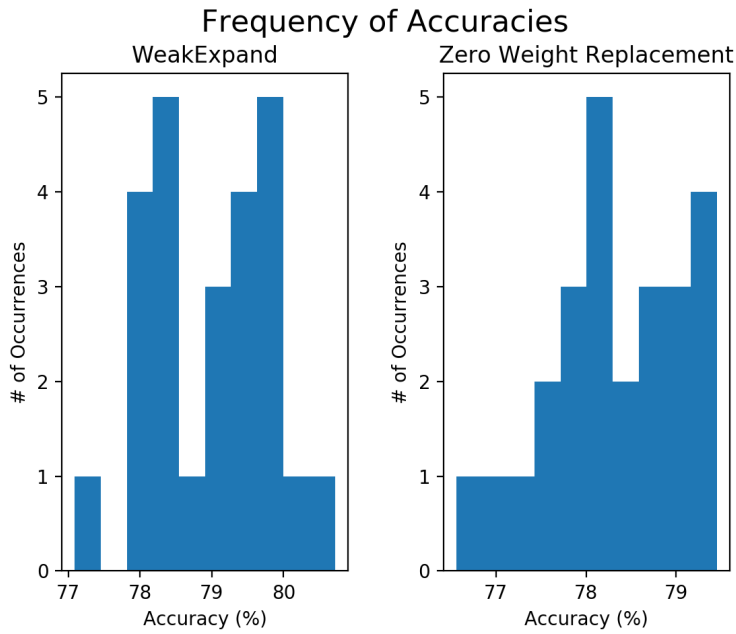


Figure 4.11: Histogram of accuracies obtained from training VGG16 while applying WeakExpand and Zero Weight Expansion.

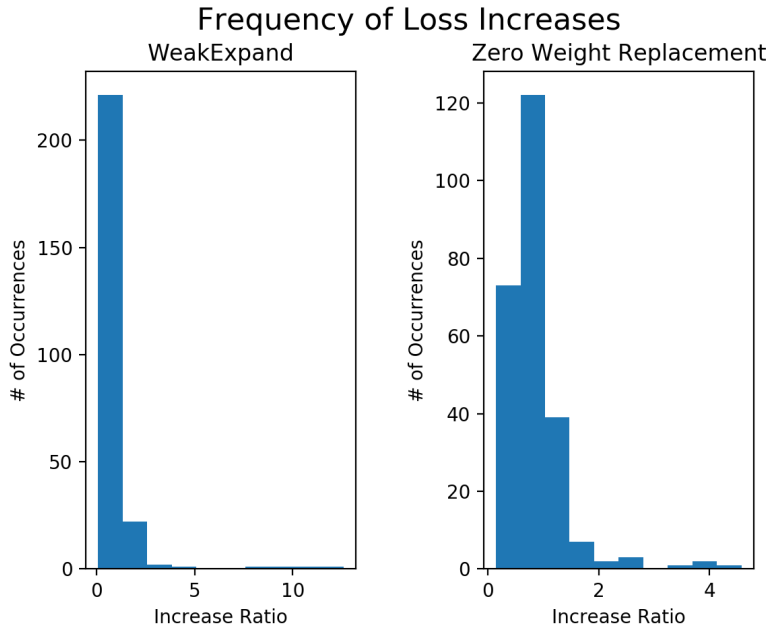


Figure 4.12: Histogram of loss increases obtained from training VGG16 while applying WeakExpand and Zero Weight Expansion.

a result of an expansion to a high degree will result in all weights being forced close to zero.

Chapter 5

Conclusions and Future Work

Modern deep learning models have become more difficult to train due to the more cognitive objectives they target such as object saliency or sequence-to-sequence translation. Patterns may exist in the underlying problem a data set models that cannot be learned from the data immediately available. Researchers have begun deploying human-in-the-loop solutions to alleviate such issues, some entirely user driven (Amershi et al. 2012) and others limited to knowing when a solution is "good enough" (Awasthi et al. 2014). A primary pain point in line with this problem is hyperparameter search present in the majority of learning solutions, particularly in the architecture choices of neural networks.

In this thesis, I have presented partial solutions to this problem in the form of two algorithms: EigenRemove for the removal of weights from network layers and WeakExpand for the addition of weights to network layers. EigenRemove leverages the geometric structure of a weight matrix to determine which weights contribute the least to the information held in the matrix as a whole. This process is executed on both the forward and backward matrices for a particular layer, with the row space eigenvectors of the forward matrix making the final decision as to which neurons to remove. It was found that EigenRemove may cause higher spikes in loss as compared to Minimum Weight Selection, but will better aid in generalization in further training of the network.

WeakExpand takes highly signaling neurons and distributes them into the newly created weights based on amount by which a user wishes to expand a given layer. This process was compared to the trivial Zero Weight Expansion strategy that provably minimizes total norm difference in activation. It was shown that WeakExpand has comparable performance to Zero Weight Expansion. However, large expansions will distribute weights potentially several times, forcing weights dangerously close to zero.

These two operations are a stepping stone to the realization of fully interactive neural network training. Operations that remain to be explored include the collapse of two layers into one and the stretching of one layer into two. The latter has been explored in network speedup (Jaderberg et al. 2014). However, this approach does not fully separate the layer into two independent layers, rather it performs SVD on the layer weights and splits by eigenspaces, feeding input to the first and reconstructing the original matrix with the second. With the addition of these two operations and the amount of research available in applications of matrix decomposition in deep learning, I believe a training procedure similar to Psichogios and Ungar (1994) can be more efficiently applied to deep networks. Users can potentially adjust architecture online with visualizations providing information as to the generalization of the layer itself using the resultant singular values generated from the weight matrix decomposition.

These operations in conjunction with a visualization have a strong potential to save users money and time in training deep models. Rather than construct complicated distributed training systems to efficiently perform an architecture grid search, they can simply open a visualization and tweak the architecture to better performance. This method of training can not only lead to more generalized models, but more intuitive AI.

Chapter 6

Appendix A

The code in Listing 6.1 gives a more detailed description of the implementations of EigenRemove for both classical neural network weight matrices and convolutional neural network weight matrices. Since classical weight matrices are two dimensions and can be transposed, this implementation can be written blind to whether matrix is the forward or backward weight matrix to the target layer. However, transpose is not defined for the four dimensional convolutional weight matrices, therefore the reduced matrix estimation must be reshaped to flatten all kernel filters into a single dimension along the output channel so means can be effectively calculated. Full code can be found at Code for experiments can be found at https://github.com/austinpgraham/InterruptNet.

The code in Listing 6.2 gives a more detailed description of the implementations of WeakExpand for both classical neural network weight matrices and convolutional neural network weight matrices. The four dimensional structure of convolutional weight matrices must be taken into account here as well when determining to which dimension a ratio must be applied. If this is not done, ratios will span inter-channel and damage the current learned state of the network.

Listing 6.1: The EigenRemove algorithm. The function EigenRemoveANN performs the process on two dimensional weight matrices for classical neural networks; the function EigenRemoveCNN performs the process on four dimensional weight tensors for convolutional neural networks

```
def EigenRemoveANN(W, size, rank):
    # Perform Tucker Decomposition with smaller rank
    core, factors = tucker_decompose(W, rank=rank)
    # Reconstruct rank-reduced estimtion
    reduced = reconstruct_matrix(core, factors)
    # Minimum weight selection by mean output
    means = mean(reduced)
    idxs = topk(means, size)
    # Select neurons to keep
    return reduced[idxs]
def EigenRemoveCNN(W, size, rank):
    # It is only desirable to perform
    # the decomposition in the modes
    # mapping the channels, as to
    # operate on the means of the kernel
    # filter weights
    core, factors = partial_tucker_decompose(data)
    # Reconstruct rank-reduced tensor
    reduced = reconstruct_tensor(core, factors)
    # Map to processable shape
    reduced = map_shape(reduced)
    # Minimum weight selection by mean of kernel filters
    means = mean(reduced)
    idxs = topk(means, size)
    # Select filters to keep
    return reduced[idxs]
```

Listing 6.2: The WeakExpand algorithm. The function WeakExpandANN performs the process on two dimensional weight matrices for classical neural networks; the function WeakExpandCNN performs the process on four dimensional weight tensors for convolutional neural networks

```
def WeakExpandANN(W, size):
    # Get the means of output weights. For the
    # forward layer this should be along the column
    # axis, and backward along row axis.
    weight_avgs = mean(W, dimension=x)
    # Sort them, keeping the order where the indices
    # appear in the sorted list.
    idxs = weight_avgs.sort(descending=True)
    # Calculate the number of neurons to add
    extend_amount = ceiling(size / len(idxs))
    # Repeat each index by number of appearances
    ratios = idxs.count().repeat(extend_amount)
    # Return the adjusted weights
    return W[idxs] / ratios
def WeakExpandCNN(W, size):
    # The process is similar to that of a
    # classical ANN. However, the ratios must
    # be expanded into four dimensions to
    # operate on kernel filters.
    weight_avgs = mean(W, dimension=x)
    idxs = weight_avgs.sort(descending=True)
    ratios = idxs.count()
    # Extend into four dimensions
    ratios = ratios.unsqueeze(-1).unsqueeze(-1)
    ratios = ratios.repeat(extend_amount)
    return W[idxs] / ratios
```

Reference List

Alippi, C., S. Disabato, and M. Roveri, 2018: Moving convolutional neural networks to embedded systems: the alexnet and vgg-16 case. *Proceedings of the 17th ACM/IEEE International Conference on Information Processing in Sensor Networks*, IEEE Press, 212–223.

Amershi, S., J. Fogarty, and D. Weld, 2012: Regroup: Interactive machine learning for on-demand group creation in social networks. *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, ACM, 21–30.

Ankerst, M., C. Elsen, M. Ester, and H.-P. Kriegel, 1999: Visual classification: an interactive approach to decision tree construction. *Proceedings of the fifth ACM SIGKDD international conference on Knowledge discovery and data mining*, ACM, 392–396.

Awasthi, P., M.-F. Balcan, and K. Voevodski, 2014: Local algorithms for interactive clustering. *ICML*, 550–558.

Banweer, K., A. Graham, J. Ripberger, N. Cesare, E. Nsoesie, and C. Grant, 2018: Multi-stage collaborative filtering for tweet geolocation. *Proceedings of the 2nd ACM SIGSPATIAL Workshop on Recommendations for Location-based Services and Social Networks*, ACM, 4.

Biswas, A. and D. Jacobs, 2014: Active image clustering with pairwise constraints from humans. *International Journal of Computer Vision*, **108**, 133–147.

Chen, W., J. Wilson, S. Tyree, K. Weinberger, and Y. Chen, 2015: Compressing neural networks with the hashing trick. *International Conference on Machine Learning*, 2285–2294.

Collobert, R. and J. Weston, 2008: A unified architecture for natural language processing: Deep neural networks with multitask learning. *Proceedings of the 25th international conference on Machine learning*, ACM, 160–167.

Coppersmith, D. and S. Winograd, 1990: Matrix multiplication via arithmetic progressions. *Journal of symbolic computation*, **9**, 251–280.

Dickinson, B. and K. Steiglitz, 1982: Eigenvectors and functions of the discrete fourier transform. *IEEE Transactions on Acoustics, Speech, and Signal Processing*, **30**, 25–31.

Du, Q. and J. E. Fowler, 2007: Hyperspectral image compression using jpeg2000 and principal component analysis. *IEEE Geoscience and Remote sensing letters*, **4**, 201–205.

Fails, J. A. and D. R. Olsen Jr, 2003: Interactive machine learning. *Proceedings of the 8th international conference on Intelligent user interfaces*, ACM, 39–45.

Frean, M., 1990: The upstart algorithm: A method for constructing and training feedforward neural networks. *Neural computation*, **2**, 198–209.

Ganganwar, V., 2012: An overview of classification algorithms for imbalanced datasets. *International Journal of Emerging Technology and Advanced Engineering*, **2**, 42–47.

Glorot, X. and Y. Bengio, 2010: Understanding the difficulty of training deep feedforward neural networks. *Proceedings of the thirteenth international conference on artificial intelligence and statistics*, 249–256.

Golub, G. H. and C. Reinsch, 1971: Singular value decomposition and least squares solutions. *Linear Algebra*, Springer, 134–151.

Graham, A., Y. Liang, L. Gruenwald, and C. Grant, 2017: formalizing interruptible algorithms for human over-the-loop analytics. *Big Data (Big Data), 2017 IEEE International Conference on*, IEEE, 4378–4383.

Gunning, D., 2017: Explainable artificial intelligence (xai). *Defense Advanced Research Projects Agency (DARPA), nd Web*.

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

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

Hochreiter, S., 1998: The vanishing gradient problem during learning recurrent neural nets and problem solutions. *International Journal of Uncertainty, Fuzziness and Knowledge-Based Systems*, **6**, 107–116.

Huang, H.-L. and P. Antonelli, 2001: Application of principal component analysis to high-resolution infrared measurement compression and retrieval. *Journal of Applied Meteorology*, **40**, 365–388.

Jaderberg, M., A. Vedaldi, and A. Zisserman, 2014: Speeding up convolutional neural networks with low rank expansions. *arXiv preprint arXiv:1405.3866*.

Jermaine, C., S. Arumugam, A. Pol, and A. Dobra, 2008: Scalable approximate query processing with the dbo engine. *ACM Transactions on Database Systems (TODS)*, **33**, 23.

Jolliffe, I., 2011: *Principal component analysis*. Springer.

Karlik, B. and A. V. Olgac, 2011: Performance analysis of various activation functions in generalized mlp architectures of neural networks. *International Journal of Artificial Intelligence and Expert Systems*, **1**, 111–122.

Kolda, T. G. and B. W. Bader, 2009: Tensor decompositions and applications. *SIAM review*, **51**, 455–500.

Kossaifi, J., Y. Panagakis, A. Anandkumar, and M. Pantic, 2019: Tensorly: Tensor learning in python. *Journal of Machine Learning Research*, **20**, 1–6. URL `http://jmlr.org/papers/v20/18-277.html`

Kotsiantis, S. B., I. Zaharakis, and P. Pintelas, 2007: Supervised machine learning: A review of classification techniques. *Emerging artificial intelligence applications in computer engineering*, **160**, 3–24.

Krizhevsky, A., I. Sutskever, and G. E. Hinton, 2012: Imagenet classification with deep convolutional neural networks. *Advances in neural information processing systems*, 1097–1105.

Kulesza, T., M. Burnett, W.-K. Wong, and S. Stumpf, 2015: Principles of explanatory debugging to personalize interactive machine learning. *Proceedings of the 20th international conference on intelligent user interfaces*, ACM, 126–137.

Lad, S. and D. Parikh, 2014: Interactively guiding semi-supervised clustering via attribute-based explanations. *European Conference on Computer Vision*, Springer, 333–349.

LeCun, Y., Y. Bengio, and G. Hinton, 2015: Deep learning. *nature*, **521**, 436.

LeCun, Y., L. Bottou, Y. Bengio, P. Haffner, et al., 1998: Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, **86**, 2278–2324.

LeCun, Y., D. Touresky, G. Hinton, and T. Sejnowski, 1988: A theoretical framework for back-propagation. *Proceedings of the 1988 connectionist models summer school*, CMU, Pittsburgh, Pa: Morgan Kaufmann, volume 1, 21–28.

LeCun, Y. A., L. Bottou, G. B. Orr, and K.-R. Müller, 2012: Efficient backprop. *Neural networks: Tricks of the trade*, Springer, 9–48.

Lee, J., Y. Bahri, R. Novak, S. S. Schoenholz, J. Pennington, and J. Sohl-Dickstein, 2017: Deep neural networks as gaussian processes. *arXiv preprint arXiv:1711.00165*.

Lin, T.-Y., P. Goyal, R. Girshick, K. He, and P. Dollár, 2017: Focal loss for dense object detection. *Proceedings of the IEEE international conference on computer vision*, 2980–2988.

Maclaurin, D., D. Duvenaud, and R. Adams, 2015: Gradient-based hyperparameter optimization through reversible learning. *International Conference on Machine Learning*, 2113–2122.

McCulloch, W. S. and W. Pitts, 1943: A logical calculus of the ideas immanent in nervous activity. *The bulletin of mathematical biophysics*, **5**, 115–133.

Molchanov, P., S. Tyree, T. Karras, T. Aila, and J. Kautz, 2016: Pruning convolutional neural networks for resource efficient inference. *arXiv preprint arXiv:1611.06440*.

Nar, K., O. Ocal, S. S. Sastry, and K. Ramchandran, 2019: Cross-entropy loss and low-rank features have responsibility for adversarial examples. *arXiv preprint arXiv:1901.08360*.

Pearson, K., 1901: Liii. on lines and planes of closest fit to systems of points in space. *The London, Edinburgh, and Dublin Philosophical Magazine and Journal of Science*, **2**, 559–572.

Psichogios, D. C. and L. H. Ungar, 1994: Svd-net: An algorithm that automatically selects network structure. *IEEE Transactions on Neural Networks*, **5**, 513–515.

Qassim, H., A. Verma, and D. Feinzimer, 2018: Compressed residual-vgg16 cnn model for big data places image recognition. *2018 IEEE 8th Annual Computing and Communication Workshop and Conference (CCWC)*, IEEE, 169–175.

Rumelhart, D. E., G. E. Hinton, R. J. Williams, et al., 1988: Learning representations by back-propagating errors. *Cognitive modeling*, **5**, 1.

Sainath, T. N., B. Kingsbury, V. Sindhwani, E. Arisoy, and B. Ramabhadran, 2013: Low-rank matrix factorization for deep neural network training with high-dimensional output targets. *2013 IEEE international conference on acoustics, speech and signal processing*, IEEE, 6655–6659.

Schmidhuber, J., 2015: Deep learning in neural networks: An overview. *Neural networks*, **61**, 85–117.

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

Sutskever, I., O. Vinyals, and Q. V. Le, 2014: Sequence to sequence learning with neural networks. *Advances in neural information processing systems*, 3104–3112.

Taur, J.-S. and C.-W. Tao, 1996: Medical image compression using principal component analysis. *Proceedings of 3rd IEEE International Conference on Image Processing*, IEEE, volume 2, 903–906.

Tirumala, S. S., S. Ali, and C. P. Ramesh, 2016: Evolving deep neural networks: A new prospect. *2016 12th International Conference on Natural Computation, Fuzzy Systems and Knowledge Discovery (ICNC-FSKD)*, IEEE, 69–74.

Tolias, G., R. Sicre, and H. Jégou, 2015: Particular object retrieval with integral max-pooling of cnn activations. *arXiv preprint arXiv:1511.05879*.

Ueno, K., X. Xi, E. Keogh, and D.-J. Lee, 2006: Anytime classification using the nearest neighbor algorithm with applications to stream mining. *Data Mining, 2006. ICDM'06. Sixth International Conference on*, IEEE, 623–632.

Xue, J., J. Li, and Y. Gong, 2013: Restructuring of deep neural network acoustic models with singular value decomposition. *Interspeech*, 2365–2369.

Yam, Y., 1997: Fuzzy approximation via grid point sampling and singular value decomposition. *IEEE Transactions on Systems, Man, and Cybernetics, Part B (Cybernetics)*, **27**, 933–951.

Yao, X., 1999: Evolving artificial neural networks. *Proceedings of the IEEE*, **87**, 1423–1447.

Zhang, S., W. Wang, J. Ford, F. Makedon, and J. Pearlman, 2005: Using singular value decomposition approximation for collaborative filtering. *Seventh IEEE International Conference on E-Commerce Technology (CEC'05)*, IEEE, 257–264.

Zhu, M. and S. Gupta, 2017: To prune, or not to prune: exploring the efficacy of pruning for model compression. *arXiv preprint arXiv:1710.01878*.