COMPARING LEARNED REPRESENTATIONS BETWEEN UNPRUNED AND PRUNED DEEP CONVOLUTIONAL NEURAL NETWORKS

A Thesis

presented to

the Faculty of California Polytechnic State University,

San Luis Obispo

In Partial Fulfillment

of the Requirements for the Degree

Master of Science in Computer Science

by

Parker Mitchell

June 2022

COMMITTEE MEMBERSHIP

TITLE: Comparing Learned Representations Between Unpruned and Pruned Deep Convolutional Neural Networks

AUTHOR: Parker Mitchell

DATE SUBMITTED: June 2022

COMMITTEE CHAIR: John Seng, Ph.D.
Professor of Computer Science

COMMITTEE MEMBER: Maria Pantoja, Ph.D.
Professor of Computer Science

COMMITTEE MEMBER: Stephen Beard, Ph.D.
Professor of Computer Engineering

ABSTRACT

Comparing Learned Representations Between Unpruned and Pruned Deep Convolutional Neural Networks

Parker Mitchell

While deep neural networks have shown impressive performance in computer vision tasks, natural language processing, and other domains, the sizes and inference times of these models can often prevent them from being used on resource-constrained systems. Furthermore, as these networks grow larger in size and complexity, it can become even harder to understand the learned representations of the input data that these networks form through training. These issues of growing network size, increasing complexity and runtime, and ambiguity in the understanding of internal representations serve as guiding points for this work.

In this thesis, we create a neural network that is capable of predicting up to three path waypoints given an input image. This network will be used in conjunction with other networks to help guide an autonomous robotic vehicle. Since this neural network will be deployed to an embedded system, it is important that our network is efficient. As such, we use a network compression technique known as L1 norm pruning to reduce the size of the network and speed up the inference time, while retaining similar loss. Furthermore, we investigate the effects that pruning has on the internal learned representations of models by comparing unpruned and pruned network layers using projection weighted canonical correlation analysis (PWCCA). Our results show that for deep convolutional neural networks (CNN), PWCCA similarity scores between early convolutional layers start low and then gradually increase towards the final layers of the network, with some peaks in the intermediate layers. We also show that for our deep CNN, linear layers at the end of the network also exhibit very high

similarity, serving to guide the dissimilar representations from intermediate convolutional layers to a common representation that yields similar network performance between unpruned and pruned networks.

ACKNOWLEDGMENTS

Thanks to:

- Dr. John Seng, for inspiring this work and supporting me through this research.
- Dr. Maria Pantoja and Dr. Stephen Beard, for teaching me so much and for being on my committee.
- My family, who loved and supported me every step of the way.
- Christopher Siu, for being a great friend and mentor throughout my time at Cal Poly.
- My friends, who were always there for me.

TABLE OF CONTENTS

LIST OF FIGURES

Chapter 1

INTRODUCTION

In recent years, deep neural networks (DNNs) have achieved outstanding performance in a myriad of domains and tasks, including natural language processing, image recognition, entertainment, and more [22, 46]. To create models that perform better, the general trend has been to make such networks deeper and wider by adding more layers and more parameters in each layer, respectively. However, as these networks become more complex, they often take longer to run when making predictions and incur larger storage costs. This increased complexity makes it harder to run such networks on devices that do not have adequate computing power, such as IoT, mobile, and edge devices. Furthermore, as networks become more complex and increase in parameter count, it can be even harder to understand the internal representations that these networks are learning.

As a result of the increasing complexity of these networks, neural network compression techniques have become increasingly important in order to reduce the number of parameters of such networks, in turn reducing their inference time and storage requirements. One such way to compress neural networks is through pruning, where parameters of the network are "pruned", or removed, to reduce overall network size and time to execute, while retaining similar network loss and/or accuracy. Pruning is an important technique that can compress even very large models into smaller and more manageable sizes with faster speeds, allowing them to be distributed to resource-constrained devices. Neural network pruning has been established since the 1980s [15, 31, 18], but has seen even more research in recent years due to ever-increasing network complexity.

However, while pruning techniques have been gaining more attention over the last several years, there is still much research to be done concerning the comparison between pruned and unpruned network representations. Recent studies have primarily focused on deciphering the internal representations within individual networks [48, 4], while some have focused on comparing representations across different networks [37, 30]. The latter cases use a statistical technique known as Canonical Correlation Analysis (CCA) to compare respective layers in each network and compute a measure of the similarity between those layers. In this way, they are able to gain insights about the learning dynamics of deep convolutional neural networks and recurrent neural networks. At a high-level, this technique enables researchers to look into the similarities of the representations that neural networks learn through training. Applying this technique to compare unpruned and pruned networks could offer valuable insights into the effects that pruning has on the learned representations of networks after fine-tuning.

In this thesis, we seek to develop a neural network with multi-output prediction capabilities to predict path waypoint coordinates for an autonomous robotic vehicle. For our network, we use a pretrained lightweight variant of EfficientNet as our base model, and modify the end of the network to perform waypoint coordinate regression [44]. This network outputs one-to-three pairs of (x, y) coordinates indicating the model's predicted waypoint positions. Using this network, we attempt to compress it by pruning it to various degrees in an effort to speed up the network's inference time and reduce storage costs. Ultimately, we want our model to be able to accurately predict these waypoints when given an input image, visualized in Figure 1.1. As this neural network will be guiding a robot's navigation in real-time, it is important that it is fast so that it provides guidance information when the robot needs it. Finally, we further the work of Ansuini et al. to compare and investigate the similarities between the learned representations of unpruned and pruned networks using a variant of CCA

Figure 1.1: Image displaying goal of our neural network, that is, predicting the path waypoints (represented by the blue dots) in an input image.

known as Projection Weighted Canonical Correlation Analysis (PWCCA) [2]. To the best of our knowledge there is no publication that performs an extensive layer-by-layer analysis of pruned and unpruned deep neural networks using PWCCA.

In summary, the main contributions of this thesis are:

- Modification of a state-of-the-art machine learning network to enable it to make high-quality path waypoint predictions when given an input image.
- A performance analysis of a range of compressions of this network via neural network pruning, using Microsoft's Neural Network Intelligence toolkit [29].
- The first layer-by-layer analysis of the unpruned and pruned networks using PWCCA, to give more insight into the representational similarities between these networks.

The remainder of this thesis will give background information regarding deep learning, neural network pruning, other network compression techniques, and autonomous robot navigation in brief in chapter 2. In chapter 3, we will cover how modern machine learning frameworks implement pruning techniques, as well as discuss using CCA and its variants to compare different neural network representations. Chapter 4 will discuss the collection and processing of our waypoint dataset, as well as details about our model implementation. Chapter 5 will outline the structure and techniques of our proposed experiments, with results and analyses being discussed in chapter 6. In chapter 7 will present our conclusions, and in chapter 8 we will end with a discussion of future work.

Chapter 2

BACKGROUND

In recent years, deep neural networks have achieved outstanding performance in a myriad of domains and tasks, ranging from computer vision to natural language processing to entertainment and more. However, while these networks further develop and become more complex to advance the state of the art, their storage footprint and computational complexity also increase greatly. On top of this, as we try to move machine learning models to IoT and edge devices, as well as reduce our carbon footprint, there is an ever growing need to compress these neural networks into manageable and distributable sizes [33]. As a result of such increasing complexity and model size, methods have been developed to compress the size of neural networks and to speed up inference times and shrink models to more manageable sizes. This section seeks to give background about what deep neural networks and convolutional neural networks are, why neural networks need to be compressed, and the main techniques that are being researched and developed to achieve such compression. Additionally, as part of this thesis focuses on building a neural network for an autonomous robot, we briefly discuss autonomous robot navigation and its uses.

2.1 Deep Learning

While neural networks are not new to the world of machine learning (ML) and artificial intelligence (AI), the use of deep neural networks (DNNs), with many intermediate hidden layers, are relatively new due to advances in technology and faster algorithms that allow these networks to learn [23, 39]. Prior to deep learning (DL),

conventional machine learning required that a system be engineered to carefully extract and internally represent relevant features from the raw input data. For images, this would require mapping pixel values into meaningful representations that a machine learning system could work with and learn from. Not only does this require a large amount of specific fine-tuning of the model, but also a great deal of domain-specific knowledge and expertise to be able to define what the relevant features are and how to model them for any given situation. Deep learning has alleviated many of these issues, as one of the most important aspects of deep learning is that it does not require features to be hand engineered. Deep learning enables the discovery and learning of features from the data itself by adjusting its internal learnable parameters, without human intervention.

In supervised deep learning, input data has a known correct label. The deep learning model produces an output of what it thinks is the correct label, and uses a cost or loss function to see how far off its prediction was from the correct label. This loss function is used to adjust the deep learning model's internal parameters, usually referred to as weights, in order to achieve a greater accuracy when predicting solutions. By computing the gradient of the loss function with respect to the learnable weights within the model, this system can determine how much the loss will go up or down if the weight is slightly altered, a method known as gradient descent. The process of going back through the model and updating these weights is called backpropagation. Iteratively feeding data to the model, computing the loss of the model's prediction, and updating the weights accordingly with backpropagation allows one to train a model to achieve high accuracy on a domain specific task.

Deep learning now permeates many different domains and industries due to its unbounded learning potential. However, one of the most prominent uses of deep learning is for computer vision tasks, such as object classification and pattern recognition.

2.1.1 Convolutional Neural Networks

Convolutional neural networks, also called CNNs or ConvNets, are a type of deep learning neural networks that excel in solving complex computer vision tasks [34, 7]. CNNs employ many of the same techniques as other neural networks, but have more image-specific features built into their architecture to allow them to handle these difficult computer vision problems.

The architecture of CNNs often include many different types of layers (as visualized by Figure 2.1(, but typically consist of the following components:

- An **input layer**, to take in the pixel values of an input image.
- A **convolutional layer** that uses low-dimensional filters (also called kernels) that convolve across the input data's height and width, taking the dot product of the filter and input. This produces a 2D activation map of that filter, which is later used to predict class scores for the output.
- A **pooling layer** to downsample results from the convolutional layers along the spatial dimensionality (height and width) of the input.
- One to many **fully-connected layers** to produce class scores from the activation maps generated by the convolutional layers.

Aptly named "convolutional neural networks", CNNs are able to learn the filters in the convolutional layers to identify key patterns within images. Early filters detect basic features like edges and lines, while intermediate filters are able to recognize more complex patterns, like textures and shapes. Towards the end, the final filters are able to detect whole objects, as outlined in Figure 2.2.

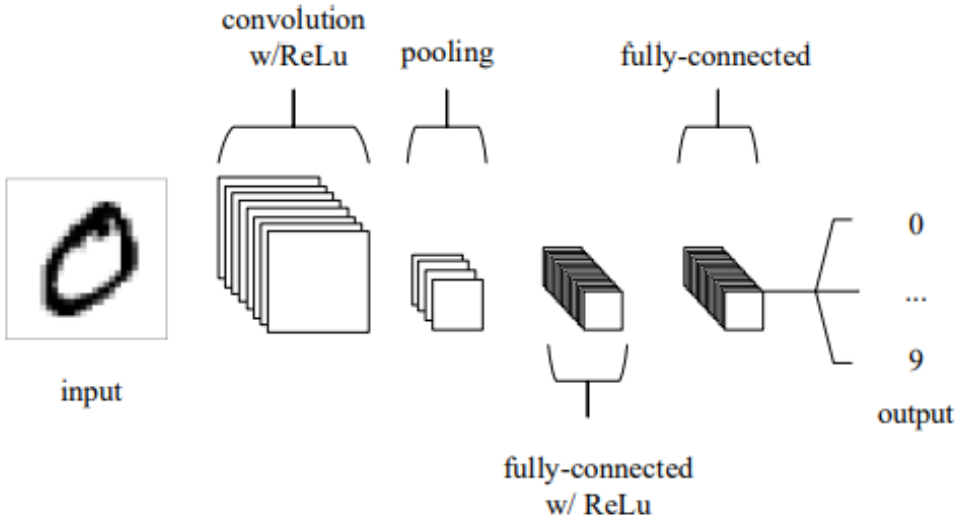


Figure 2.1: Simple CNN Architecture [34]

2.2 Transfer Learning

As neural networks increase in size and complexity, the time and data required to sufficiently train these networks also greatly increases. When more layers, neurons, and weights are added to network architectures, the number of operations are usually increased, and thus require more time to compute. However, these deeper and larger neural networks are often worth the time and effort spent training because of their ever-increasing accuracies and performance. This dilemma is not a new issue to the field, and luckily is partially alleviated through the use of a machine learning technique called transfer learning.

Transfer learning allows us to utilize the knowledge that a neural network model has learned previously to new learning tasks and domains [47]. There are several different approaches to transfer learning, but one of the most successful and widely used at the moment is model-based transfer learning, specifically through the use of pretraining. With pretraining, deep neural networks are first trained on a sufficient amount of

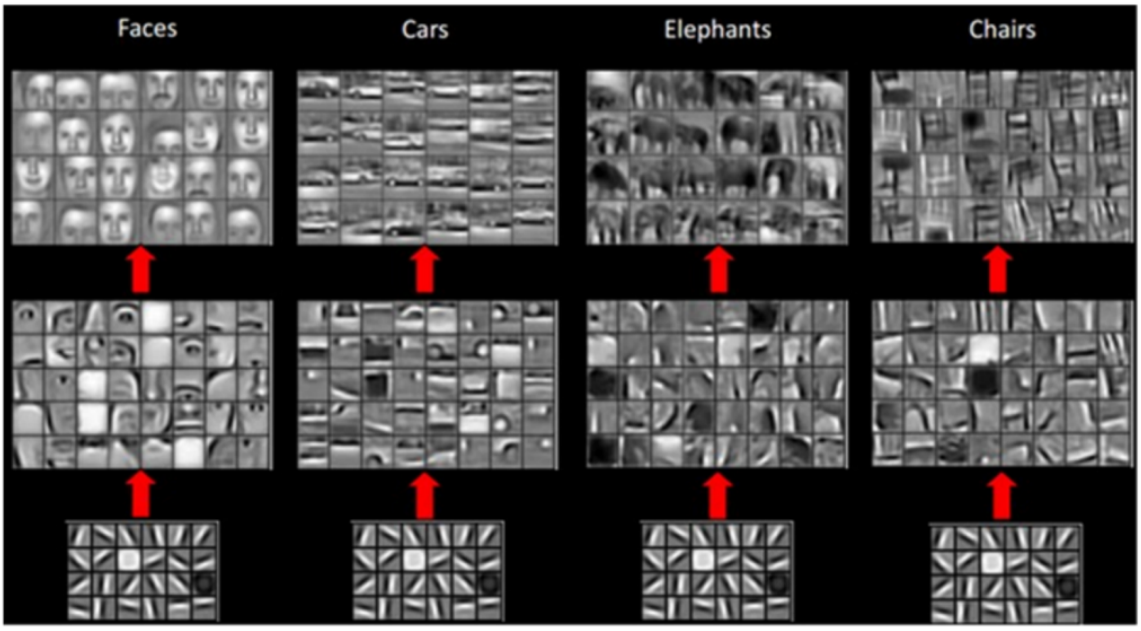


Figure 2.2: Learning CNN Filters - Visualized [41]

source data, which can be different from the eventual target data. After training on the source data, the model is then potentially adjusted or altered, and then retrained, or "fine-tuned", on the new target data. The model alterations and fine-tuning are quite important, and are further discussed below.

2.2.1 Adjusting Model Structure for Transfer Learning

As noted earlier, when using transfer learning with pretraining, the model structure may be altered or adjusted to meet the demands of the new learning task. For instance, if the original learning task was to classify images as either cats or dogs, then the model's output might consist of a two-dimensional vector containing the class probabilities for a cat and a dog, based on the input image. However, the new learning task could be quite different from the original learning task, such as the classification of dolphin images versus whale images. Even further, the new task could change from a classification problem to a regression problem, or to another form. The

work done in this thesis uses transfer learning to adjust a model originally used for image classification to perform a regression task. Transfer learning facilitates this knowledge transfer very well, and provides great flexibility for changing the domain and learning task for neural network models.

2.2.2 Fine-Tuning Pretrained Models

When fine-tuning the new model, there is the option to fix or freeze certain parameters, such as weights and filters, so that during when the model is being retrained those parameters are not adjusted, and remain the same as in the originally-trained model. This is useful if certain parameters are not expected to change very much when fine-tuning on the target data, allas it allows time and resources to be saved by not having to retrain/compute those parameters.

This freezing of parameters can be particularly valuable in convolutional neural networks, as it has been shown that the layers of these networks tend to learn in a general-to-specific fashion. That is, the earlier layers of the network are more general learners, learning low-level notions of edges, shapes, and textures, while the later layers of the network are more specific learners, and are able to identify high-level features that are specific to the input data. Figure 2.3 illustrates these concepts with an example of two models. We have some model A (the top model), that is trained on input A for task A. Then with transfer learning, we can make a new model that leverages model A's learned knowledge named model B, which operates on input B for task B. If model B does not need to relearn the very general things that are picked up in the earlier layers from model A's pretraining, then we can freeze the weights of the first two layers as shown, and only update the later weights with backpropagation when fine-tuning. Additionally, we may change the structure of the network to fit our learning task's needs, as seen on the right-hand side of networks A and B.

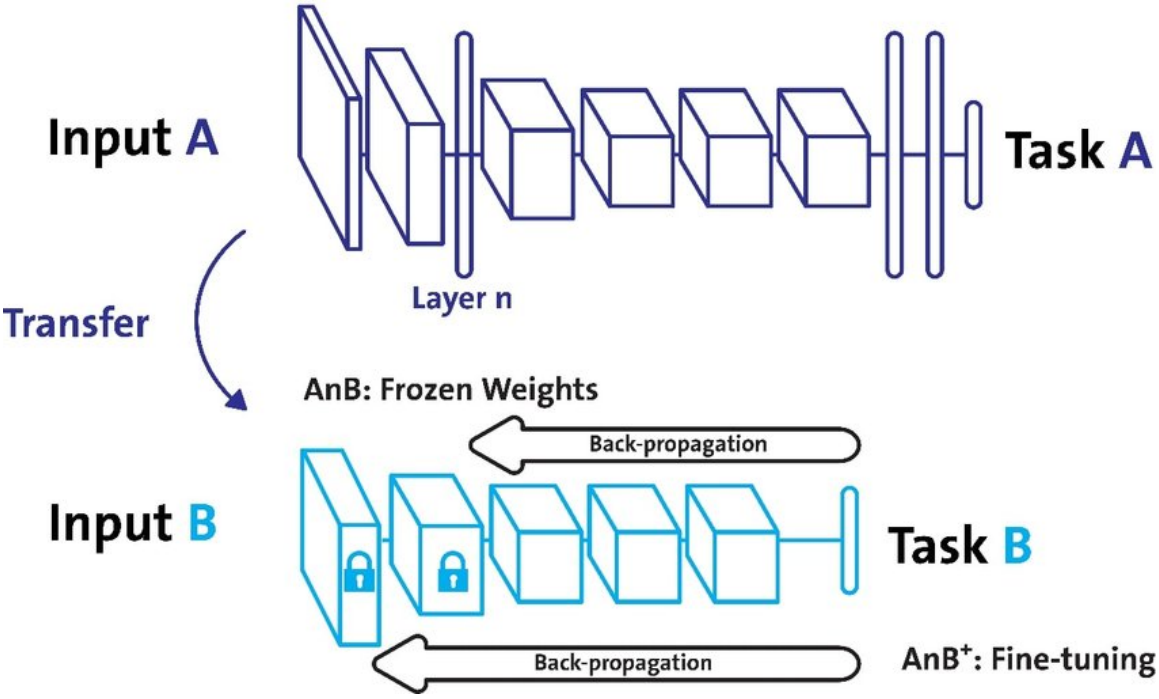


Figure 2.3: Illustration of how transfer learning works [17]

2.3 Neural Network Pruning

Pruning is a neural network compression technique that aims to remove components of the network to reduce its size, while preserving a similar accuracy [5]. Pruning methods for neural networks emerged in the late 1980s [15, 31, 18], with increased efforts in recent years as deep learning has taken off and network complexity has greatly increased.

Across the majority of pruning techniques, there are several key aspects that are common. The first is *scoring* [5]. Scoring, also referred to as ranking, is the way that parameters are evaluated for pruning. Different pruning techniques may use different parameters to judge which parts of the network to prune. Some techniques may score parameters based on their absolute values, activations, gradients, or other parameters. Additionally, parameters may be scored locally (e.g. per layer or block), or globally

across the whole network. The next aspect of pruning is *scheduling*. Scheduling determines the amount to prune within the network at each time step. Some pruning techniques are one-shot, and prune all at once, while others are iterative, and prune across multiple time-steps. Other functions may also be introduced to prune at different rates, dictated by said function. For reference, this thesis focuses on one-shot pruning and its effects. The final aspect that is common across most pruning techniques is *fine-tuning*. This includes methods that continue to train the network after pruning. When retraining, most approaches either use the weights that were present before pruning, or reinitialize the network's weights and then continue to train.

As noted previously, this thesis focuses on and investigates the effects of one-shot pruning. With one-shot pruning, techniques generally adhere to a similar step-by-step approach: 1) train the network to completion, 2) prune unimportant parameters, and 3) retrain the network to fine-tune parameters [13]. Typically, the first and third stages of this process are similar among different pruning implementations. However, the second stage is where the variation within pruning techniques emerges, and will make up most of the work and discussion in this thesis. Traditionally, there are several ways to compress a neural network, but the two most relevant pruning methods include unstructured pruning and structured pruning.

2.3.1 Unstructured Pruning

The first technique is unstructured pruning, sometimes also referred to as magnitude-based pruning (MBP). With this approach, if a weight in the network has a small value, or a value below some predefined threshold value, then the weight is pruned. Hence, this pruning method is unstructured because it does not take into account any relationship between weights that are pruned. In practice, this is done by setting those weights to zero, effectively creating a sparse network. The reasoning behind

this is that if a weight in the network is low, then that often means it has little effect on the output. If the effect is so small, then the weight can be set to zero, effectively removing that weight from the network. This can reduce both memory and computation, as the weight can be removed, and forward passes and updates via backpropagation no longer have to consider that weight.

In practice, there are several different nuances in how unstructured pruning can be implemented. One way is removing all weights in the network that are less than some threshold. This has the potential to create a very sparse network if the threshold is high enough, as there could be many weights that are actually below that. Another way is to prune a percentage of weights. This could be done by either pruning a percentage of the lowest weights in the network, or a percentage of the lowest weights in the network that do not meet some threshold. Finally, all of this pruning can be done in a layer-wise or global fashion. In terms of layer-wise, we would decide on some pruning method listed prior (e.g. percentage under threshold) and prune that layer accordingly. Then, we would move onto the next layer and prune the same way. A global pruning approach would prune the whole network (irrespective of layers), taking all weights into account equally to find the lowest weights or percentage of weights. While choosing between layer-wise and global unstructured pruning is sometimes problem dependent, it has been found that pruning a percentage of the smallest weights globally generally results in the best performance [40]. This prevents the unnecessary pruning of weights in layers just to meet some layer-wise quota, and better prunes small weights that are actually not as important to the inference of the model.

2.3.2 Structured Pruning

The second technique is structured pruning, also commonly referred to as node pruning. Structured pruning is concerned with removing a larger part of a neural network, like a neuron or filter, thus altering the structure of the network. Sometimes when using unstructured pruning of weights, the network can become overly sparse in some areas. In such cases, even though the number of parameters has been reduced, computational costs may not have actually been reduced, as it takes more effort to calculate with such irregular sparsity [24]. While there are libraries for sparse computation that can handle this, structured pruning provides an alternative pruning solution that does not require the use of sparse computational libraries.

Structured pruning is particularly prevalent when pruning convolution neural networks (CNNs). In this context, filters that have little effect on the output accuracy are identified and are removed along with their feature maps, greatly reducing computational costs. This approach does not introduce sparsity into the network, because when a neuron or filter is removed, all ingoing and outgoing connections of that neuron are removed, not just specific individual weights. The difference between unstructured and structured pruning can be visualized in Figure 2.4 [32]. Unstructured pruning is shown on the left-hand side, where we can see how connections that have been pruned are grayed-out. Structured pruning is shown on the right-hand side, where we can see that entire nodes (which could be neurons, filters, or other structural components) have been pruned, along with their corresponding connections.

In a more general sense, node pruning can be employed on deep neural networks by measuring the contribution of a node with respect to some cost or objective function, somewhat akin to backpropagation [3]. By evaluating a node's contribution to this

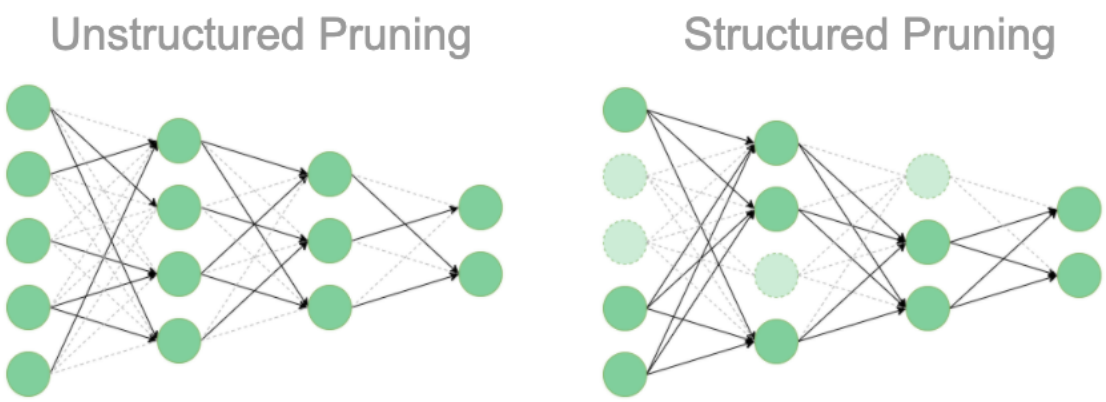


Figure 2.4: Visualizing the difference between unstructured and structured pruning [32]

objective function, we can prune nodes that have small effects on it, and keep nodes that more greatly impact it.

2.4 Alternative Neural Network Compression Techniques

While network pruning remains one of the most common compression techniques, in recent years there has been much research around alternative methods to compress neural networks. These new methods have seen great success, allowing neural networks to run faster and take up less space, and even spawning hybrid compression approaches for better performance. These other notable compression techniques include quantization and knowledge distillation, and will be discussed in the following sections.

2.4.1 Network Quantization

With network quantization, the goal is to compress the model without removing any parameters. The way this is done relies on the way we store the data. Quantization

aims to use a reduced number of bits to represent the values in the network [33]. In most networks trained on the GPU, values are stored in a 32-bit floating point (FP-32) format. Other common types are 16-bit floating point (FP-16), also called "half-precision", and 16-bit integer (INT-16). Quantization approximates the computations that are performed by creating smaller adaptive ranges for the original values to lie in, reducing the total numerical space they exist in and clipping those too far outside those ranges.

Quantization research and implementation centers around reducing memory footprint and accelerating inference by lowering the precision of the network. This is done by scaling down the representations to INT-8, INT-4, INT-2, and even 1 bit representations of the original data. From a memory perspective, if the original network is operating in FP-32, and it is quantized down to INT-8, the result is an immediate 4x reduction in size. If floating point calculations and a bit more precision were still needed then FP-16 could be used, still resulting in a 2x reduction in size compared to the original network. As such, there has been much research around quantization in recent years.

2.4.2 Knowledge Distillation

Knowledge distillation is a compression technique that seeks to learn a smaller "student" model from a larger "teacher" model [12]. This approach is inspired by how human beings learn, where a larger, more knowledgeable teacher model helps a smaller, less knowledgeable model learn what is necessary for their particular task, typically in a more succinct network structure. A simple visualization of knowledge distillation can be seen in Figure 2.5.

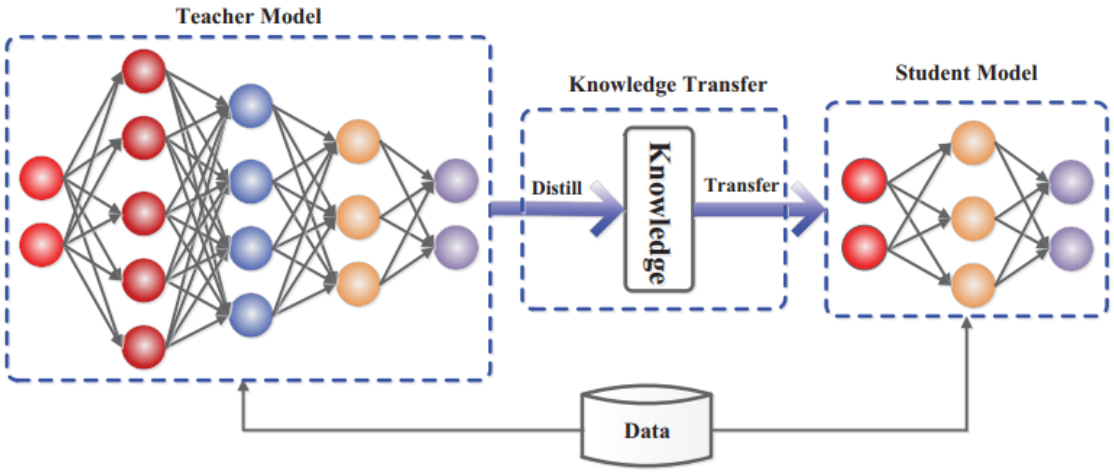


Figure 2.5: Knowledge Distillation Process Flow [12]

In practice, knowledge distillation usually follows a two stage process: 1) train the larger teacher model on a set of training data, and 2) use the teaching model to "extract knowledge in the form of logits or the intermediate features", then used to train the student model during distillation. The actual distillation of knowledge can be done using several different algorithms. A simple yet effective way to transfer knowledge is just to have the student model directly match the knowledge representation contained by the teacher model. In recent practice, this has been done with GANs for adversarial distillation, with multiple teacher models, and even with graphs to learn and model intra-data relationships. These are just a few of many different distillation techniques, but serve to show the breadth of this compression technique.

2.5 Autonomous Robot Navigation

As technology has gotten more performant and electronics have gotten less expensive, the interest and development of autonomous robots has greatly increased. Through the use of machine learning, neural networks are empowering robots to navigate their environments and perform complex tasks, often without any human intervention.

At a high-level, autonomous navigation for robots can be broken down into several distinct steps: 1) have the robot create a map of its environment, 2) have the robot localize, that is, find itself in its environmental map, and 3) have the robot use the map to plan and take actions to achieve its goal [16].

To create such maps and have the data needed to make certain decisions, autonomous robots are typically equipped with several sensors to take in the information around them. Common sensors used for these robots include Lidar, Sonar, Radar, and Cameras [19]. For this thesis, the eventual robot that our neural network will be used on will be collecting and processing data via a front-facing camera. Cameras are well-suited for dynamic environments due to their high sampling rate, sampling 30 frames of image data per second, if not more. In many autonomous robotic navigation setups, including ours, data is passed from the camera to the neural network, where it is transformed and processed to produce outputs that the robot can make decisions based off of. In our case, our neural network will be predicting the waypoint coordinates of each image, and making decisions about which waypoint to drive towards based on its end-goal.

Chapter 3

RELATED WORK

In this chapter, we discuss how most modern deep learning frameworks approach neural network pruning. We then describe recent research surrounding the comparison of neural network representations and the techniques used to do so. Finally, we discuss how these comparison techniques have been used to compare pruned and unpruned neural networks.

3.1 Pruning with Machine Learning Frameworks

Today, there exist a plethora of machine learning frameworks that allow researchers and developers to quickly design and deploy effective neural networks for a variety of tasks. Some of the most popular frameworks at the time of writing this include TensorFlow, Keras, and PyTorch [1, 8, 36]. These frameworks make it relatively simple and intuitive to rapidly prototype machine learning models for research or commercial use, and offer many different options for model compression.

Both TensorFlow (and to an extent Keras, which is built on top of TensorFlow) and PyTorch support model quantization and model pruning to help reduce model size and decrease model latency when performing inference. Focusing on pruning, these frameworks will typically change the pruned weights to zero, or apply a mask over the model that effectively zeroes the weights, creating a sparse model. Once pruned, size reductions can be seen by actually compressing a model with a technique such as gzip. Since there are typically more frequent sequences of zeros in the model's parameters after applying these pruning methods, compression algorithms like gzip

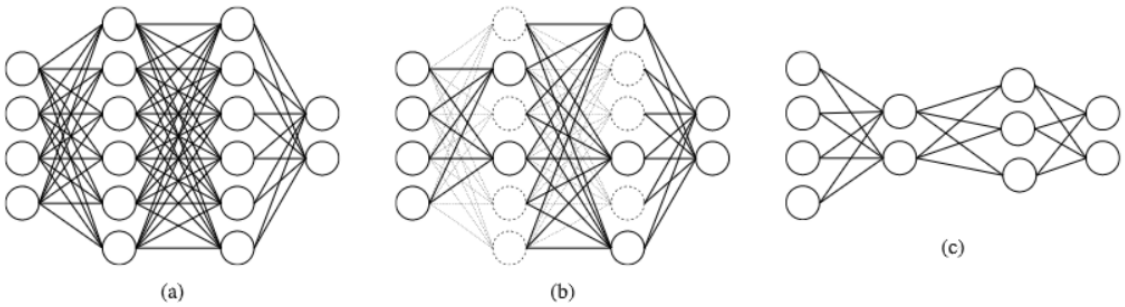


Figure 3.1: The stages of model compression via pruning [6]

often allow for greater compression than when compressing an unpruned/unmasked model.

However, most of these frameworks do not actually reduce the model size or speed up inference times after applying the built-in pruning methods they provide. Instead, they only theoretically reduce the model size and inference time by masking or zeroing out parameters, as mentioned previously. To achieve practical performance gains when using these pruning techniques often requires the use of specialized hardware and/or software accelerators following the pruning to take advantage of the masking or zeroing of these parameters [6]. Figure 3.1 helps outline this by showing most of the stages of model compression using pruning. In (a), we start with a dense model, where the circles represent neurons and the lines represent parameters. In (b), we see the initial stages of pruning, with the pruned neurons represented by the dotted circles. Most machine learning frameworks will only reach stage (b), masking out the pruned neurons in the network but not actually removing neurons or weights. In practice, it is much more desirable to reach stage (c), where the masked parameters are actually removed from the network and/or sparse data structures are used to reduce the model size.

While these frameworks do not explicitly support practical performance gains via pruning, other libraries and toolkits do exist that can actually prune, compress, and

speedup neural network models. One such toolkit is Microsoft's Neural Network Intelligence (NNI) toolkit [29]. NNI enables developers and researchers to perform neural architecture search, automate feature engineering, tune hyperparameters, and most prevalent to this work, it allows them to compress neural network models. NNI supports a myriad of pruning algorithms, ranging from one-shot to iterative and from unstructured (which they call "fine-grained") to structured (which they call "coarse-grained"). NNI also supports real model speedup and size reductions by using the masks generated during model pruning. They do so by computing the model graph for the PyTorch model, and then replacing the pruned layers from the mask with smaller layers for coarse-grained masks, or with sparse kernels for fine-grained masks. A minimal example of NNI pruning and speedup can be seen in Figure 3.2. In this diagram, they start with a sample weight matrix from a neural network and apply pruning to generate a mask for the matrix. After this mask is produced, they perform an element-wise multiplication between the mask and the original weight matrix to find out which cells should be considered after pruning, zeroing out all other cells. This sparse matrix is then passed to NNI's model speedup software to actually change and reduce the dimensions of the new matrix, resulting in a smaller pruned weight matrix.

3.2 Comparing Neural Networks

One field of deep learning research that has seen more interest lately is that of neural network comparison. Researchers have been increasingly more interested in the different learned representations that neural networks form when trained, and how those representations evolve over time and compare to one another [37, 30]. As a result of this interest, several techniques have emerged to evaluate the similarity of these representations between neural networks.

Figure 3.2: Model speedup with NNI [29]

3.2.1 Canonical Correlation Analysis

In 1936, Harold Hotelling published the paper "Relations Between Two Sets of Variates", in which he proposed a technique called Canonical Correlation Analysis (CCA) to measure the associations between two multivariate sets of vectors [14, 42]. As an example, consider variables that are related to exercise and health. We can measure variables related to exercise, such as the magnitude of weight lifted in a weighted squat, how fast someone runs a mile, etc. We can also measure variables that are related to overall health, like heart rate, blood pressure, and more. While these are different types of variables, we may want to further investigate their relationships and see how associated they are. CCA gives us the ability to investigate the relationship between these two sets of variables, and serves as a powerful tool for comparing multivariate vectors.

CCA is motivated by the need to summarize these relationships of sets into a smaller dimensionality, so that the number of statistical values needed to interpret are smaller and manageable. Consider two sets of variables, with the first having p dimensions

and the second having q dimensions. If we wanted to create scatter plots using all combinations of variables from both sets to better visualize potential relations between these sets, we would end up with pq plots. Similarly, if we wanted to compute all correlations between variables from the first and second set, we would have pq correlations. This may be manageable to observe and interpret when pq is small, but when pq grows large these tasks can become very difficult. CCA summarizes these relationships into fewer values while still retaining the most important information about the relationships between the sets.

Mathematically, canonical correlation analysis starts with two sets of variables X and Y, with p and q variables respectively. For convenience, the set X is chosen such that $p \leq q$. Then, for each set X and Y, a set of linear combinations of its variables is created; call them U and V. Within U, each example will be a linear combination of the p X variables, while in V, each example will be a linear combination of the q Y variables. Each member of U is then paired with a member of V, forming canonical variate pairs. As $p \leq q$, there are p canonical variate pairs. CCA then aims to find linear combinations of the variables such that the correlations between the members of each canonical variate pair are maximized. To do so, we calculate the variance for each variable U_i in the set U, as well as the variance for each variable V_j in the set V. Once these values are computed, we are able to compute the canonical correlation for each canonical variate pair. For the i^{th} canonical variate pair, (U_i, V_i), we compute the canonical correlation by calculating the correlation between U_i and V_i. This correlation is what CCA aims to maximize.

To better understand canonical correlation analysis, Mehrabyan gives an algebraic visualization of CCA, see in Figure 3.3 [28]. First, we start with our sets of variables:

$$\begin{aligned} X &= (X_1, X_2, ..., X_p) \\ Y &= (Y_1, Y_2, ..., Y_q) \end{aligned} \tag{3.1}$$

We then create a pair of canonical variates, which are linear combinations of the variables in the two sets:

$$\begin{aligned} CV_{1X} &= a_1X_1 + a_2X_2 + ... + a_pX_p \\ CV_{1Y} &= b_1Y_1 + b_2Y_2 + ... + b_qY_q \end{aligned} \tag{3.2}$$

The coefficients to these variables are weights chosen by CCA such that the correlation between these two canonical variates is maximized. Figure 3.3 helps visualize this by starting on the edges with our variables, then working inwards, creating our canonical variates from our variable sets. Then we further work inwards in the diagram by calculating the correlation between those canonical variates. Once the first canonical correlation is computed, CCA then moves onto the next pair of canonical variates, but with a new constraint. This constraint is that these new variates are perpendicular to the previous variates and that they are uncorrelated to the previous variates. Since we have p variables in the first set and q variables in the second set, we compute $min(p, q)$ canonical correlations to work with, allowing us to compare similarity with CCA now.

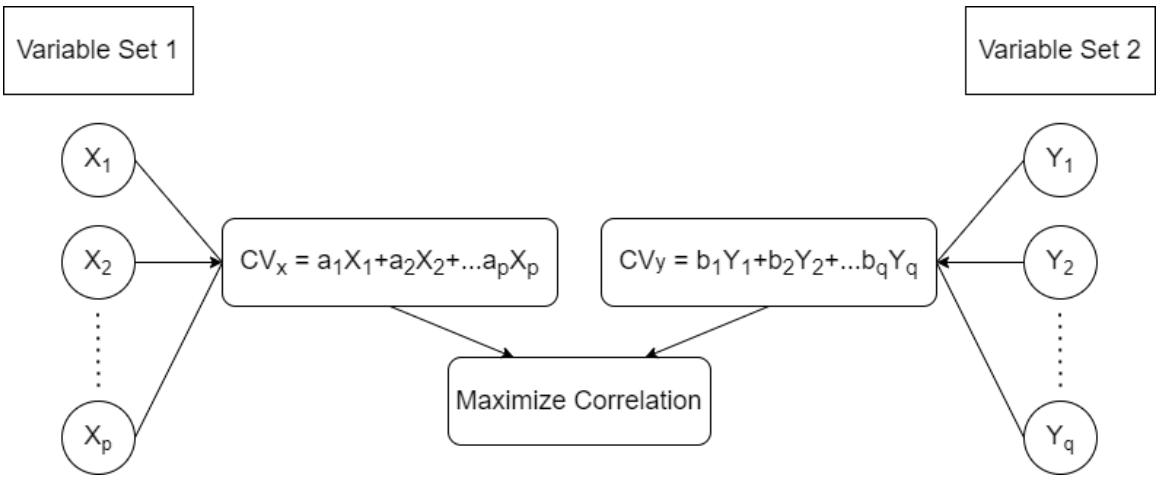


Figure 3.3: Calculating Canonical Variates [28]

3.2.2 Singular Vector Canonical Correlation Analysis

Raghu et al. propose Singular Vector Canonical Correlation Analysis (SVCCA), a new technique for analyzing the deep representations that neural networks learn through training [37]. SVCCA combines Singular Value Decomposition (SVD) and Canonical Correlation Analysis (CCA) to measure the correlations between these learned representations. In order to compare two neural networks, the authors had to decide on the representation of a neuron. They define the representation of a neuron to be a vector of its activations in response to a set of inputs. Further, they define a layer in a network to be the "set of neuron [activation] vectors contained within that layer", where the activation vectors span some subspace. This formalism of the representation of a neuron and a layer allows the authors to apply SVD and CCA to neural network layers and gain powerful insights into their similarities and more.

The reason why SVD is used alongside CCA here is to help reduce noise. Singular Value Decomposition allows us to reduce the dimensionality of a subspace (i.e. a layer) spanned by its vectors (i.e. neuron activations), taking only the most important directions into account. By taking only the most important directions–the directions

that "explain 99% of variance in the subspace"–SVCCA is able to cut down on noise, which is typically composed of those low variance directions. Once SVD finds the most important directions, the authors apply CCA to the new reduced subspaces, finding linear transforming the new subspaces so that they are maximally correlated.

The authors' main findings give insights into the dynamics of how neural networks learn. Their results reinforce the idea that, in general, learning happens bottom-up, meaning that the layers closest to the input seem to "solidify their final representations" earlier than layers later in the network. However, the exception to this is the layers closest to the output, as they found that those layers solidify their final representations early on as well. This insight inspired a new training technique that they call "Freeze Training", wherein lower layers are progressively frozen, only updating the higher layers as time goes on since the lower layers converge to their final representations faster. They showed that this novel technique can decrease the number of floating-point operations when training, making it a more efficient model training approach.

3.2.3 Projection Weighted Canonical Correlation Analysis

Building off of [37], Morcos et al. use CCA to further investigate the learned representations of neural networks [30]. In their work, they find that SVCCA can have trouble when trying to distinguish between the actual signal and noise in the representation of a neural network. Furthermore, in the SVCCA approach, the authors used the mean correlation coefficient of two layers to represent their similarity, implying that all of the correlation vectors are of equal importance when representing a layer. However, Morcos et al. note that deep neural networks have been shown to not rely on the full dimensionality of a layer to form their representations. As a

result of this, SVCCA will typically underestimate the similarity measures between two neural network representations.

To address the potential issues with SVCCA, Morcos et al. propose to replace the mean with a weighted mean, where canonical correlations of greater importance to the neural network's representation are weighted more than less important canonical correlations. They achieve this using a method called projection weighting to determine said weights. First, they compute the CCA vectors for a layer. Formally, they define a layer L_1 to have vectors of its neuron activations in the form of $[z_1, \ldots, z_a]$, and CCA vectors in the form $[h_1, \ldots, h_c]$. They then compute how much each CCA vector contributes to the original output.This is done for each CCA vector, h_i, by taking the sum of the absolute value of the inner product between the CCA vector and each neuron activation z_j, over all j:

$$\tilde{a_i} = \sum_j | < h_i, z_j > | \tag{3.3}$$

Once they have computed how much each CCA vector is accountable for the original output, they normalize those sets to get weights a_i such that $\sum_i a_i = 1$. With these normalized weights, they calculate the projection weighted CCA distances with:

$$d(L_1, L_2) = 1 - \sum_{i=1}^{c} a_i p^{(i)} \tag{3.4}$$

Not only do the authors propose projection weighted canonical correlation analysis (PWCCA) as a novel method for comparing neural network representations, they also use it to generate new insights about the learning properties for generalizing versus memorizing networks, wider versus deeper networks, and recurrent neural networks (RNNs). They found that groups of networks that generalize tend to converge to more

similar representations than groups of networks that memorize. They also discovered that wider networks (i.e. networks that have more neurons in layers) tend to converge to more similar representations than narrower networks (i.e. networks that have less neurons in each layer, but many more layers) do when learning. Finally, they build off of [37]'s work that suggests that residual neural networks and convolutional neural networks learn bottom-up by examining if RNNs convey this same property. Consistent with [37]'s results, the authors find that RNNs also exhibit bottom-up learning convergence properties.

3.3 Comparing Unpruned and Pruned Neural Networks

The research described in the previous sections outline several techniques used to compare neural network representations. In these works, the authors measure similarity on different networks in different contexts. Some of their experiments are performed on the same network at different points during the training process compared to the final network to gain insights about the dynamics of learning. Other experiments compare two networks with the same architecture but different parameter initializations. However, there has been little research concerning the application of these similarity indexes to compare pruned and unpruned neural networks.

To our knowledge, the only other work focusing on comparing the similarities of pruned and unpruned neural networks using one of these techniques was carried out by Ansuini et al. [2]. In their work, the authors compare each hidden layer between a pruned and unpruned network by applying SVCCA to the activation vectors from their respective layers. In their experiments, they prune networks using an iterative pruning method called iterative magnitude pruning (IMP) [10], which works by incrementally pruning away the parameters with the lowest magnitudes over several

iterations, until some target sparity is achieved. They employ this technique on minimal convolutional neural network architecture inspired by VGGNet core, where they stack blocks composed of 2 to 4 convolutional layers, followed by a max-pooling layer, eventually leading to a fully-connected layer before the output. While they perform their experiments using increasing amounts of these convolutional blocks, the deepest network they test with contains no more than 14 layers.

In their results they find that pruned and unpruned convolutional neural networks exhibit high similarity in earlier layers while the intermediate layers exhibit much lower similarity. However, they also saw that the fully-connected layers at the end had high similarity. They feel that this suggests that fully-connected layers help to pivot the pruned models to establish somewhat similar performance and representations as the unpruned models, even though the intermediate layers of the pruned models conveyed different learned representations when compared to the unpruned models.

Chapter 4

IMPLEMENTATION

In this chapter, we discuss the collection and processing of our waypoint dataset, the implementation of our waypoint neural network, and the pruning of said network. Furthermore, we discuss the parameters used within our model, and detail how we trained the network.

4.1 Data Collection and Processing

As this thesis regards one research component of a larger project, the data used within this thesis is specific to the aforementioned project's domain. The project at hand is waypoint prediction for an autonomous Star Wars robot. This entails creating a neural network to predict where the robot should be headed. More specifically, given an input image (a frame from the robot's video feed), our neural network will output up to 3 (x, y) coordinate pairs representing the waypoint predictions for that frame. At the moment, the environment in which the robot will be driving is the concrete pathways in the quad between Cal Poly's Bonderson Center and Building 192-Engineering IV. This environment is shown in Figure 4.1, where the driveable space is outlined in green, and in Figure 4.2, where the driveable space is the gray concrete paths between the buildings. As such, the data used to train this waypoint prediction network consists of images of those concrete pathways, as shown in Figure 4.3. Since we are predicting waypoints on an image, we want our model to be able to find the (x, y) coordinates of a waypoint on an image. Thus, this is a regression

Figure 4.1: Driveable Environment - Outlined in Green [11]

problem, where our model seeks to improve when predicting these (x, y) waypoints based on ground-truth labels.

4.1.1 Data Collection

When starting this thesis, there was no prior waypoint data that could be leveraged for the Engineering IV/Bonderson quad. As such, a significant amount of time was spent collecting, processing, and labeling waypoint data that could be used to train our model. This was a laborious process, and warranted the creation of several scripts and tools to help.

Figure 4.2: Driveable Environment - Satellite Outline with Terrain [11]

The first step in the data collection process was to record video footage of the Engineering IV/Bonderson quad. For the sake of our model, we only concerned ourselves with the concrete pathways, as shown in Figure 4.4. As such, we collected video footage walking around those pathways from different angles to ensure we were collecting representative data. From there, we wrote a Python script to grab one frame from the video every second, resize that frame to an image that is 640 pixels wide by 360 pixels tall to match the input dimensions of our model, and save the resized frame as a JPEG image. Each frame is saved to a local directory to then be labeled, which will be discussed in the next section.

4.1.2 Data Labeling and Partitioning

In order for our model to be able to predict accurate path waypoints given an image, our inputs must be clearly labeled. In this context, a label for a waypoint is an (x,

Figure 4.3: An Input Image to the Neural Network

y) coordinate on the image representing the pixel location of that waypoint. It is imperative that these labels are as consistent as possible, meaning that two similar images have similarly labeled waypoints. Since waypoint prediction is a regression problem, there is no exact correct output, that is, there is no perfect (x, y) coordinate that represents a waypoint. As such, we need to do our best to label waypoints as consistently and accurately as we can. By doing so, our model should be able to learn what features in the input images make up a waypoint by seeing multiple consistent examples, enabling it to make predictions and improve upon its mistakes. Through this continuous process, our model will be able to generalize better when handling new images.

To ensure that waypoints on images were consistently labeled, we proposed a small set of labeling rules to more objectively guide the labeling process. The proposed rules are as follows:

Figure 4.4: Waypoint Pathways

- Waypoints must be within the top 75% of the image (stated in another way, waypoints must be above the bottom 25% of the image)
- If there is a fork within the middle 50% of the image, mark all individual waypoints
- If there are more than 3 waypoints in the image, place a single waypoint between the farthest waypoints

While these rules exist to help objectify the labeling process, some images may not always perfectly align with such structure. As such, liberties were taken in the labeling process to handle these oddities, while still making a conscious effort to label them as consistently as the rest.

In order to efficiently label the frames collected from the data collection stage, we created a tool in Python called EZLabeler. EZLabeler works by taking in a path to a directory of unlabeled images, then allows the user to click on different parts

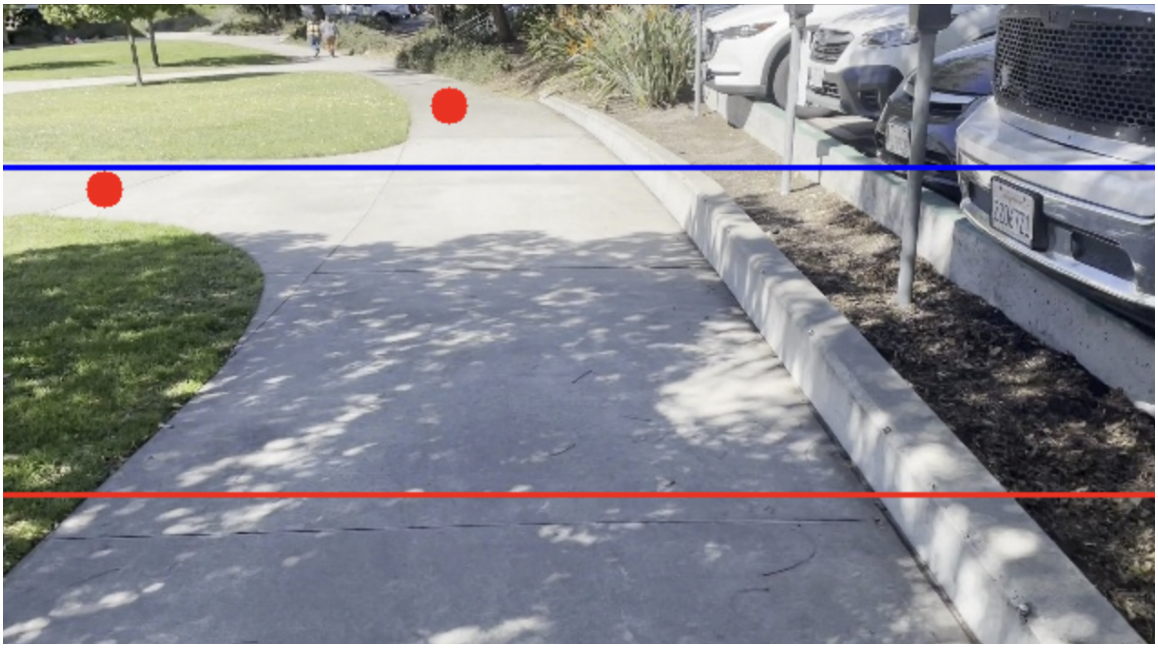


Figure 4.5: EZLabeler User Interface

of the image, capturing the (x, y) cursor position and marking that as a waypoint label. The user is able to select between 1-3 waypoints. Once the waypoints are selected and the user is done labeling an image, EZLabeler encodes the waypoint coordinates into the filename of the image for later processing. For example, if an image was originally named "frame1.jpeg", and a user labeled two waypoints on the image at (123, 225) and (439, 304), then the waypoints would be encoded into the new filename as "frame1_123_225_439_304_0_0.jpeg". The values are all separated by underscores, and nil waypoints are signified with zeros. Figure 4.5 shows an example of what EZLabeler looks like to the user. As you can see, the waypoints are indicated by red dots drawn onto the image. Additionally, there are two horizontal lines drawn onto the image to assist with the labeling rules mentioned earlier, where the top 25% of the image is above the blue line and the bottom 25% of the image is below the red line.

In total, we collected and labeled approximately 4,500 images to train the model with, randomly partitioned into a 80% / 10% / 10% split for training, validation, and testing purposes, respectively.

4.2 Models

When designing our model, we wanted to ensure that it was efficient even prior to pruning. It was important that our model be able to perform fast inference, as it would be running on an embedded system in the future. As such, the EfficientNet [44] architecture looked very promising, and became the base architecture that was decided upon.

4.2.1 EffficientNet

EfficientNet is a deep convolutional neural network architecture (also referred to as a ConvNet or CNN) that emerged in 2020 from a team at Google Brain as a result of the need for a more accurate and performant CNN architecture. The authors note how many CNNs are developed with a fixed resource budget in mind, and then later scaled up to be more performant as necessary. When scaling a model up, previous popular techniques would usually only scale the model up by its depth (i.e. adding more layers) or width (i.e. adding more neurons to each layer), and somewhat less commonly by image resolution. However, the authors of EfficientNet found that by uniformly scaling all dimensions of the model–depth, width, and image resolution–the effectiveness of scaling up ConvNets greatly increased, allowing EfficientNet to achieve "state-of-the-art 84.3% top-1 accuracy on ImageNet" upon its inception. Not only did EfficientNet achieve state-of-the-art, but it did so while being 6.1x faster and having 8.4x fewer parameters.

Due to EfficientNet's impressive speed and performance, it was a clear choice to serve as the base architecture for our model.

4.2.2 Transfer Learning with EfficientNet

The authors of EfficientNet released a whole family of EfficientNet model architectures and weights for open-source use, with the most accurate and largest variant being EfficientNet-B7. While EfficientNet-B7 was still 8.4x smaller than the previous best state of the art model (GPipe), the model was still too large for our needs, with over 66 million parameters in total. As such, we chose a pretrained lightweight variant of EfficientNet aptly named "EfficientNet-lite0", bringing the parameter count down to around 3.6 million parameters [45].

Since our network's task is to perform waypoint regression on images, we altered the base network architecture to fit our needs. The original EfficientNet architecture was trained to perform image classification on ImageNet, classifying images into 1000 different classes. As such, the base architecture's output layer is a fully-connected layer that outputs 1000 class probabilities. To fit our needs, we altered the final layer of the network to output 6 values, representing the normalized pixel coordinates of 3 waypoint (x, y) pairs. So, the every 2 outputs are the normalized x and y values for their corresponding waypoints. In this way, our model no longer performs image classification, but rather regression to predict the waypoint coordinates for a given input image. Before the output layer, we also added two fully-connected layers, each with 200 neurons, so that our model could gain better insights from the activations passed from the preceding convolutional layers.

It's worth noting that we experimented with slightly different model architectures before deciding on the one described above. At first, we evaluated the model per-

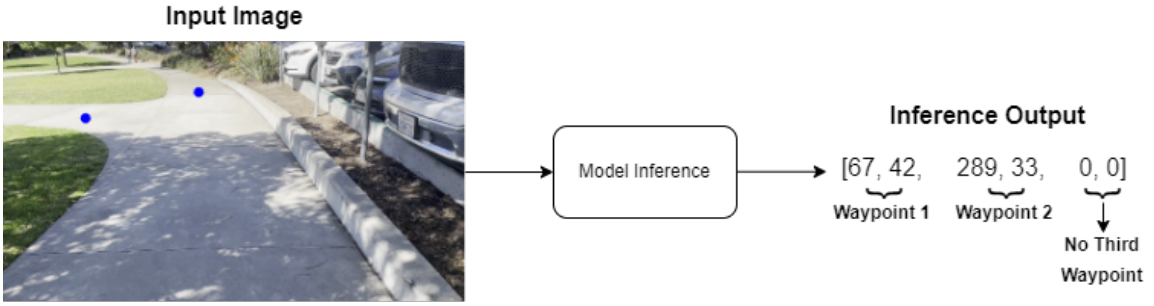


Figure 4.6: Example of model input and output. The model takes in an image, and predicts where the waypoints in the image are. The output is a tensor containing three pairs of predicted waypoint coordinates.

formance without any fully-connected layers before the 6 output nodes. We also investigated the performance when using only 1 and 2 fully-connected layers with 50 neurons, 500 neurons, and 1000 neurons each. In these cases, we found that model performance was not as high as when using 2 fully-connected layers with 200 neurons each leading into the 6 output nodes. Furthermore, we tried using a small variant of EfficientNet version 2, the next generation of EfficientNet models. In this case, model performance was close to our final model's performance, but was not better, even while using roughly 4x the number of parameters as our model. As such, we finalized our architecture to be the one described above, using 2 fully-connected layers with 200 neurons each leading into the 6 output nodes. Figure 4.6 conveys the model inference pipeline, showing the inputs and outputs. As seen in this figure, the input is an image and the output is a tensor consisting of three pairs of predicted waypoint coordinates. If any of these pairs are zero, then that means there is one less waypoint. For example, the output tensor in Figure 4.6 is [67, 42, 289, 33, 0, 0], meaning that the first waypoint is at $X_1 = 67, Y_1 = 42$, the second waypoint is at $X_2 = 289, Y_2 = 33$, and since the third pair consists of two zeros, then there is no third waypoint.

4.3 Model Training

We trained our model in PyTorch using roughly 4,500 hand-gathered and labeled images. As modern neural networks often require large amounts of data to train on, we used data augmentation to increase the size of our dataset, and help our model generalize better. Data augmentation is a common machine learning where various image transformations are applied to the original dataset to create an altered image, with similar qualities as the original. Some of the transformations we applied include random brightness contrast, gaussian noise, motion blur, random shadows, perspective transformations, and random dropout of image sections. Figure 4.7 displays examples of some of these transformations, along with the pre-augmented image for comparison. These augmentations can create different, and sometimes slightly noisy images, which when applied to our neural network training can help the model see a wider variety of images and generalize better.

In terms of model training, we will start by discussing our model's hyperparameters. For our optimizer, we used PyTorch's AdamW optimizer, an adaptive gradient algorithm based on the popular Adam optimizer that decouples the optimizer's weight decay regularization from L2 regularization, improving Adam's generalization performance [26]. We used this optimizer to train our model over 150 epochs, using a learning rate of 0.001 and a batch size of 16. We use L1 loss for our loss function, measuring the mean absolute error between the normalized waypoint coordinate labels and the network's predicted waypoint coordinates. Additionally, we used a cosine annealing learning rate scheduler, which starts off with a high learning rate, rapidly lowers it to some minimum value, and then is rapidly increased again, maintaining this cycle until training is completed.

Figure 4.7: Pre and post augmented training images. Top left is pre-augmented, and top right is its corresponding augmented image. Bottom left is pre-augmented, and bottom right is its corresponding augmented image. Transformations such as random section dropout and random shadows can be observed.

Chapter 5

EXPERIMENTAL DESIGN

In this chapter, we discuss the design and structure of our experiments in order to determine our model's performance. Furthermore we talk about the design of our pruning experiments to compress our model and decrease inference time. Finally, we discuss our approach to comparing unpruned and pruned network representational similarity using projection weighted canonical correlation analysis (PWCCA).

5.1 Experiments and Metrics to Evaluate Model Performance

The primary metric we use to evaluate our model's performance is L1 loss. The L1 loss measures the mean absolute error between two values, where a lower number represents less error and is more indicative of a better model. This function is used during training as our loss function, and serves as a valuable metric to determine how the trained model performs on the test dataset. We use L1 loss as a measure of performance because we are performing waypoint coordinate regression, thus we cannot measure the accuracy of our model as there are no objective classes categories that we can predict. In addition to L1 loss, we use several other metrics to measure our model's performance once it is trained. These include waypoint count accuracy, model inference time, and total model size.

The waypoint count accuracy is a measure of the percentage of times that the number of waypoints from our model prediction match the number of labeled waypoints. For example, if an image is labeled to have two waypoints, but our model predicted three waypoints, we would not count this as a match. However, if the model did correctly

predict that there were two waypoints, then we would consider this a match and have it contribute to the total waypoint count accuracy. To calculate the total waypoint count accuracy, we sum the number of correct matches and divide by the total number of test images to compute a total percentage.

Other important metrics to measure are the model inference time and total size of the model. Since we are trying to optimize our model to run on a resource-constrained device, we want to take note of how fast the model can perform inference on images, and how large the model is. To measure inference time, we pass a batch of 16 data points through the model and average the total inference time across 300 repeated runs. To measure model size, we measure the size that the model file takes up on disk.

To get these measurements, we first train our model for 150 epochs on our training and validation datasets of 3,658 and 457 images respectively. To measure L1 loss, we pass our testing dataset of 458 images through our model and compute the loss between the labeled waypoint coordinates and our model's predicted waypoint coordinates using PyTorch's L1 loss function. Similarly for waypoint count accuracy, we compare the number of labeled waypoints in each test set image and compare that to the number of waypoints predicted by our model to compute the total waypoint count accuracy. For inference time, the process is as described previously, averaging the inference times of the trained model over 300 runs on a batch size of 16 data points.

5.2 Pruning Neural Networks and Measuring Performance

Using the training procedure outlined in Section 4.3, we train a neural network using the training and validation dataset for 100 epochs, generating a base model to serve as the foundation for our pruning. Once we have the base model, we use Microsoft's

Neural Network Intelligence (NNI) toolkit to perform our model pruning and speedup [29]. NNI offers a variety of different pruners ranging from structured to unstructured, but the vast majority of the pruners are structured pruners.

As EfficientNet is a deep convolutional neural network, we decided to use the L1 norm pruner for our network pruning experiments. In vector terms, the L1 norm is the sum of the absolute value of a vector's components. That is, if we had a vector of the form $\mathbf{v} = (3, 2)$, then $\mathbf{v}$'s L1 norm would be $|3| + |2| = 5$. The L1 norm is also sometimes referred to as the Manhattan distance. In the context of pruning, the L1 norm pruner was introduced by Li et al. as an effective technique for pruning filters in convolutional neural networks [24]. This pruner calculates the relative importance of a filter in each layer by computing the sum of its absolute weights, which is equivalent to its L1 norm. This also gives a measure of the average magnitude of its kernel weights. The authors note that this is useful, as "filters with smaller kernel weights tend to produce feature maps with weak activations" making them prime candidates for pruning. NNI's implementation of this pruner works on both convolutional and linear layers, using the L1 norm of the weight by rows in the linear layers to serve as the pruning metric.

Using the L1 norm pruner, we pruned the base model saved at different pruning sparsities to create seven pruned models. The sparsities we pruned at were in 10% increments, ranging from 10% to 70%. A higher percentage represents a more aggressive round of pruning. After creating the pruning mask using NNI's pruner, we used NNI's model speedup library to compress the model structure based on the mask. After pruning and speeding up, we fine-tune each model by following the same training regimen from Section 4.3, but only training for 50 epochs this time. Once the pruning and fine-tuning is finished, we resume training our base model for an additional 50 epochs so that all models have been trained for 150 total epochs.

To measure the performance of our pruned models, we use the same measurements as described above in Section 5.1. That is, we measure the L1 loss, waypoint count accuracy, pruned inference time, and the pruned model size. By comparing these measurements to the measurements of the unpruned model, we can see how higher pruning sparisties may impact model performance metrics. Additionally, this comparison allows us to investigate the tradeoffs between a more compressed model versus potentially higher loss due to a reduction in parameters.

5.3 Experiments to Compare Unpruned and Pruned Neural Networks Using PWCCA

Following the release of two papers discussing the comparison of learning dynamics and representational similarities between different models using canonical correlation analysis (CCA) and projection weighted canonical correlation analysis (PWCCA), the authors published their implementations of these methods, as described in Section 3.2, under Google's open-source GitHub [37, 30]. Before using these libraries though, we had to do some upfront processing to get model activations and ensure they were in the correct format so that we could compute the PWCCA values.

To gather model activations, we first registered forward hooks on all of the layers that we were interested in comparing. Forward hooks are simply a tool in PyTorch that allow a command to be executed whenever a forward or backward pass is executed on the model. Once we registered these hooks, we performed standard inference on the model over 1,500 randomly sampled training set data points to gather our activations. Note that this sampling is seeded, so each run of PWCCA between the unpruned and pruned networks will see the same data. We then stored each set of activations along with the layer name which they came from for further processing.

In this thesis, we are only interested in comparing the linear and convolutional layers between unpruned and pruned networks. As such, we only collect the activations from those layer types. Once we have gathered those activations, we need them to have shape (*number of neurons, number of datapoints*). Naturally, the activations retrieved from linear layers will have this shape. However, the activations gathered from convolutional layers will be of shape (*number of datapoints, number of channels, height, width*). Since parameters are shared across the spatial dimensions (height and width), we can flatten these dimensions into the number of datapoints, allowing us to compare the similarities of two convolutional layers by comparing across their channels. To do so, we reshape the activations into the form (*number of datapoints x height x width, number of channels*). From here, we can use the aforementioned libraries to compute the mean PWCCA between layers. Once the activations have been reshaped, we compare each pruned model against our unpruned model by calculating the mean PWCCA score for every convolutional and linear layer over the activations dataset. For visualization purposes, we then plot those values in layer-wise order based on the model structure.

5.4 Test and Development Environment

All development, experiments, and testing were conducted on a workstation with a AMD Ryzen Threadripper 3990X 64-Core Processor CPU, running with a 3.5GHz clock speed. The GPU used on this workstation was an NVIDIA RTX A6000, with 48GB of GDDR6 memory.

Chapter 6

RESULTS AND ANALYSIS

This chapter discusses the results of the experiments laid out in chapter 5. We start by covering the results of our unpruned model performance. Next, we investigate the results of pruning our base model and measuring the performance of those pruned networks. After that, we discuss the results of comparing unpruned and pruned networks layer-by-layer using projection weighted canonical correlation analysis (PWCCA). Finally, we conclude with a discussion around some of the limitations regarding our implementation and experimental design.

6.1 Unpruned Model Performance

Based on the experiments laid out in chapter 5, we measured the performance of our model based on its L1 loss, waypoint count accuracy, inference time, and model size. Starting with arguably the most important metric, our unpruned model (also referred to as our "base" model) exhibited an L1 loss 0.0637 over our testing dataset, averaged across 20 runs with an input batch size of 16 data points at a time. Another way of interpreting this value is to say that across our entire test set, our model's predicted coordinate waypoints were within 7% of the actual coordinate waypoints. Figure 6.1 shows several example predictions, where the red dots indicate the actual labeled waypoints and the blue dots show the model's predicted waypoints. In the first image, the model's predictions are quite close to the labeled waypoints. However, sometimes the model falls short and does not always correctly predict the waypoint, as shown by the prediction in the second image.

Figure 6.1: Example waypoint coordinate predictions. Red dots represent the actual labeled waypoints, while blue dots represent the model predictions.

For the waypoint count accuracy, our model achieves a 60.26% accuracy on the testing dataset, meaning that roughly 60% of the time the number of waypoints predicted by the model match the number of actual waypoints in the image. As noted in chapter 5, this metric only considers the number of waypoints in the image, and takes no regard for the position of these waypoints (which is covered by the L1 loss metric). This metric can be useful in a broad range of applications where there is a need to know the count of waypoints in an image, and so a higher accuracy bodes well for a model.

The unpruned model's inference time for a batch size of 16 data points was measured to be 22.48 milliseconds, averaged across 300 runs. In regards to the model's size, our base model has a total model size of 15MB, with a total of 3,668,614 parameters.

After training our base model, we pruned the convolutional and linear layers of the model at varying percentages to create 7 additional models, ranging in 10% increments from 10% pruned to 70% pruned. After pruning, we trained each of them again for 50 epochs to fine-tune them. As noted previously, a higher percentage indicates more pruning, creating a more sparse model. After pruning, we measured their performance using the same metrics used to evaluate the base model.

Starting with L1 loss, we expected the loss to increase as the sparsity increased. Our reasoning behind this was that if a model has less parameters, its ability to learn complex features diminishes. This proved true, demonstrated by the loss plot for each sparsity shown in Figure 6.2. In this plot, we can see that the average loss for the 10%, 20%, and 30% pruned models is actually quite similar to the unpruned model's average loss, and then the loss steadily increases at a greater rate for the subsequent models. One hypothesis for this can be explained due to the over-parameterization of our initial model. As we are leveraging a pretrained model with transfer learning, the model initially has many parameters that may not be relevant to our data, making it over-parameterized for our use-cases. As such, these parameters could have had such little impact on the output, that pruning them did not affect the total loss very much at all. However, once the pruning became more aggressive around the 40% sparsity mark, more important parameters might have started to be pruned away, therefore increasing the average L1 loss of subsequent models.

Next, we evaluate the pruned models based on their waypoint count accuracy. We expected the waypoint count accuracy to decrease as model sparsity increased, but what we found through our experiments differed slightly. As shown in Figure 6.3, the waypoint count accuracy actually increased as model sparsity increased for a time, peaking at an accuracy of 71.83% for the 20% pruned model. After 20% pruning, the models still achieve a relatively high waypoint count accuracy around at the 30% pruning mark, with a steady accuracy dropoff after that. The final 70% pruned model exhibits a waypoint count accuracy of 49.78%. Interestingly, we also see that the 70% pruned model's waypoint count accuracy was about 2% higher than the 60% pruned model's waypoint count accuracy, which was 47.60%.

Moving onto model inference time, we see a fairly linear decay as expected, shown in Figure 6.4. As model sparsity increases, mean inference time over a batch size of 16

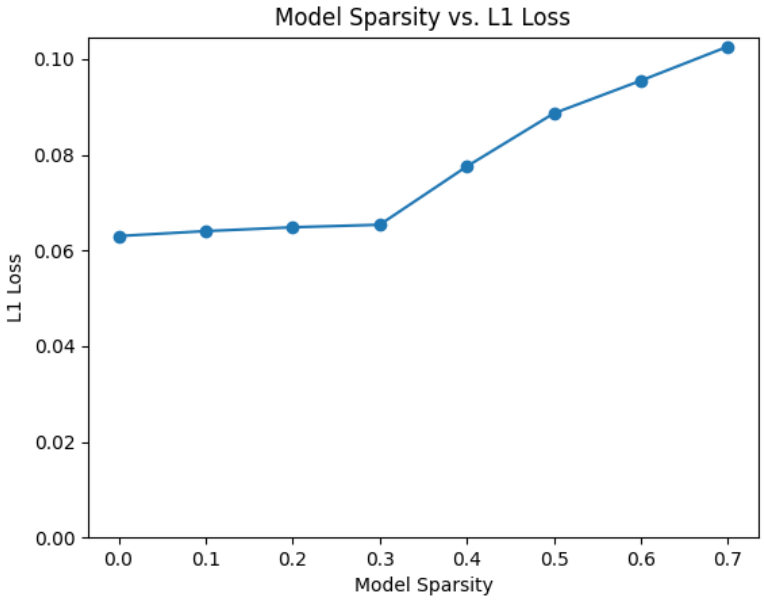


Figure 6.2: Plot of model sparsity on the x-axis versus L1 loss value on the y-axis. A sparsity value of 0 represents the unpruned model, while a higher sparsity represents a model that has been pruned more aggressively.

data points seems to linearly drop, starting at 22.48 milliseconds for the unpruned model and ending at 3.51 milliseconds for the 70% pruned model. Intuitively this makes sense, as we are pruning the linear and convolutional layers of each model linearly in 10% increments, so we expect a roughly linear decay in model inference time. Numerically, each additional 10% pruning that was applied to the model reduced the inference time by about 3 milliseconds. Taking into account figures 6.2 and 6.4, we can see that a model that has had 30% of its convolutional and linear layer parameters pruned away can achieve a very similar loss while performing inference at 12.74 milliseconds, almost 10 milliseconds faster than the base model. Models pruned past 30% can achieve even faster inference, but at the cost of a slightly greater loss.

Measuring the unpruned and pruned model sizes, we also see a fairly linear decay of size as model sparsity increases, as shown in Figure 6.5. Again, this makes sense as we are pruning each successive model at a linear rate. Figure 6.6 reinforces this by

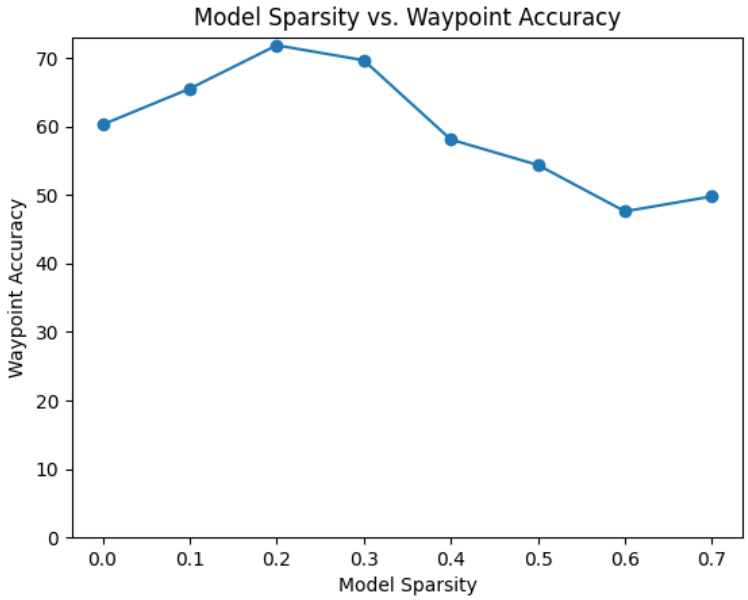


Figure 6.3: Plot of model sparsity on the x-axis versus waypoint count accuracy on the y-axis.

following the same trend, showing how the total parameter count decreases linearly as model sparsity increases.

6.2 Comparing Unpruned and Pruned Networks Using PWCCA

Once we trained our base model and produced our pruned models, we set out to investigate the similarities between the learned representations of our unpruned and pruned networks. As outlined in section 5.3, we first collected all of the convolutional and linear layer activations across 1,500 test images for each model. Then, for each pruned model, we computed the PWCCA scores for each convolution and linear layer between the unpruned and pruned model, as seen in Figure 6.7. For reference, these figures plot the layer type (either convolutional or linear, denoted by "C" and "L" respectively) in order through the network on the x-axis, and the mean PWCCA score on the y-axis.

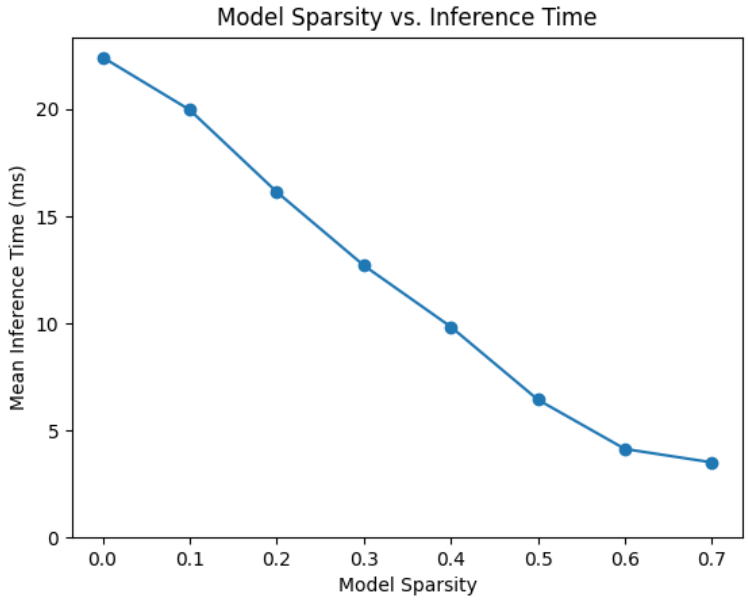


Figure 6.4: Plot of model sparsity on the x-axis versus mean inference time in milliseconds on the y-axis. Inference time is averaged across 300 runs over a 16 data point batch size.

From our results, we recognize several distinct trends. First, we are able to notice a general decline in the peaks and valleys of PWCCA scores as pruning increases. Looking at the PWCCA plots for the 10% and 20% pruned models, we notice very large spikes in PWCCA similarity in the convolutional layers. These spikes seem to primarily occur in the network's pointwise linear projection (PWL) layers, which occur at the end of Inverted Residual (IR) blocks. Inverted residual blocks were proposed by Sandler et al. in 2019 as a novel layer module [38]. Inverted residual blocks work by expanding the block input into a high-dimensional representation, performing a lightweight depth-wise separable convolution on that representation, and then using a linear convolution (also known as a pointwise convolution, or pointwise linear projection) to project the features back to a low-dimensional representation. We hypothesize that the expansion and compression/projection of these features due to the inverted residual block is guiding the representations of the both the unpruned and pruned activations to be very similar, but a further study into this would be

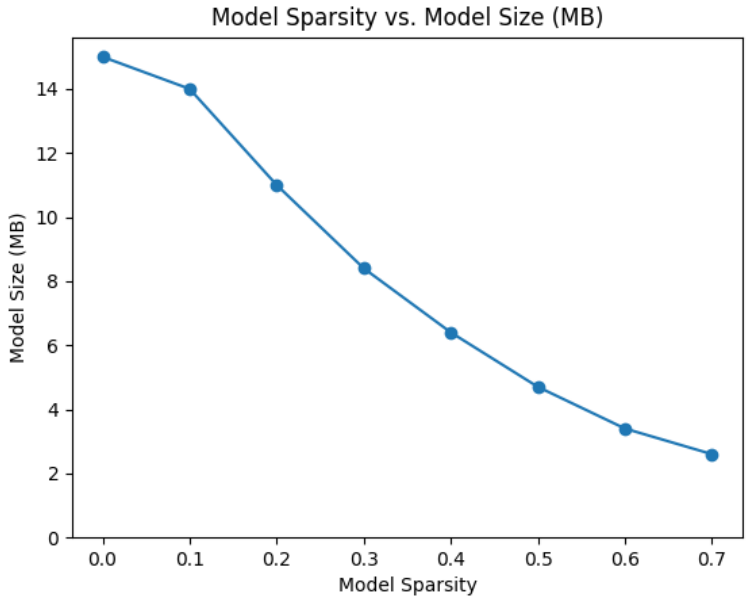


Figure 6.5: Plot of model sparsity on the x-axis versus model size on the y-axis.

needed to fully understand the relationship between them. Additionally, we see that as the model is pruned more and more, the PWCCA similarities for these PWL layers decrease.

The second trend we notice is that in all cases the activations from the unpruned and pruned networks show a general progressive trend upwards in similarity as we progress along the layers. This is best visualized in networks that are more heavily pruned, where we can see a very subtle increase in similarity as we move along roughly the first 75% of the layers, and then a more notable increase in the last quarter of the layers, spiking with very high mean PWCCA scores at the final linear layers.

In general, we can see how pruning networks produce different learned layer representations when compared to the unpruned network they were based off of, even when fine-tuned on the same dataset. This occurs even when pruning rates are small, as seen for the 10% pruned model in Figure 6.7. This reinforces the findings of Ansuini

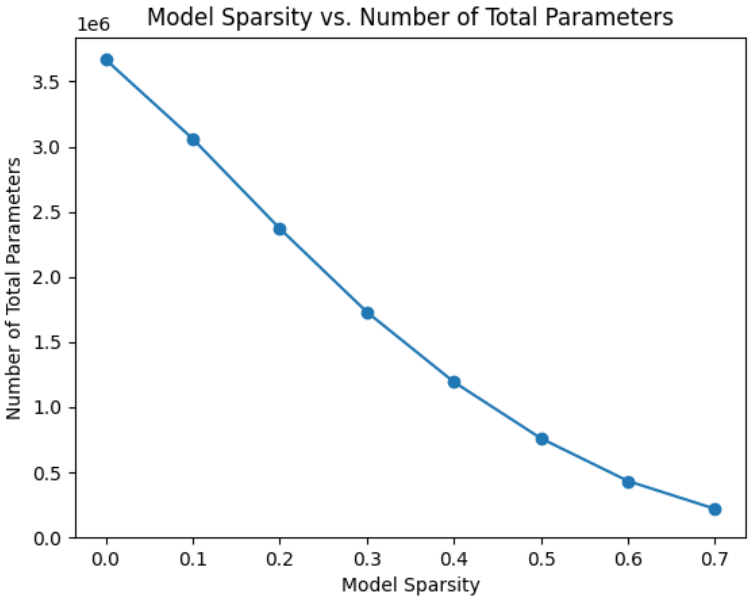
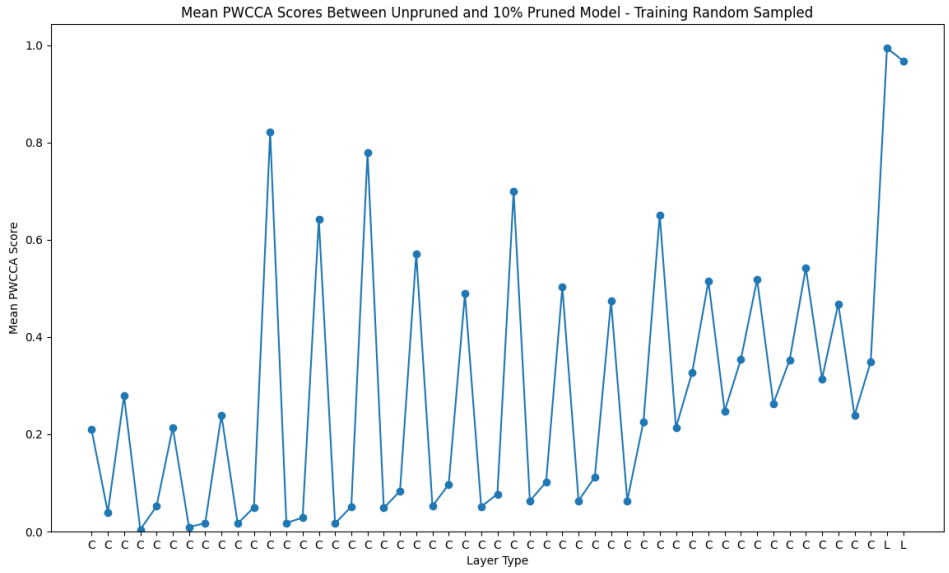
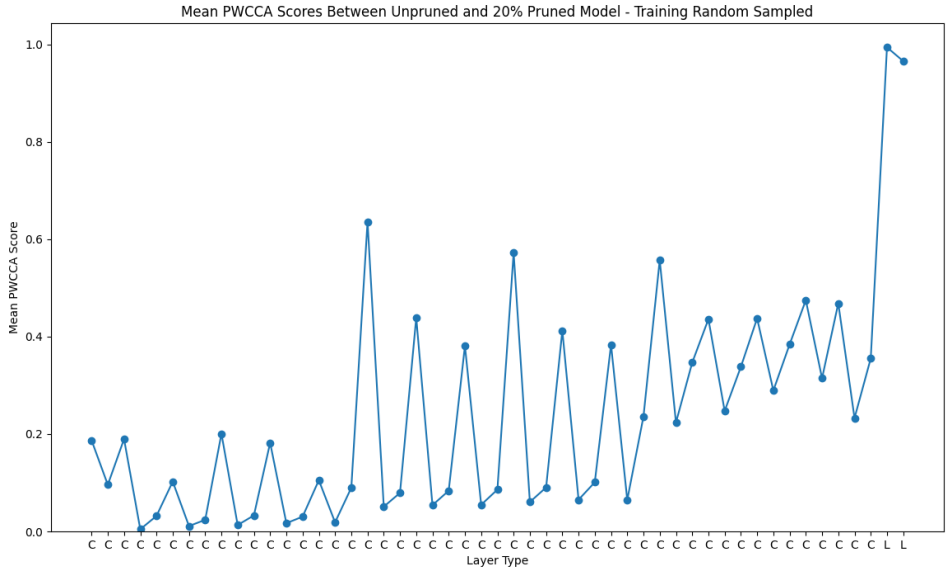


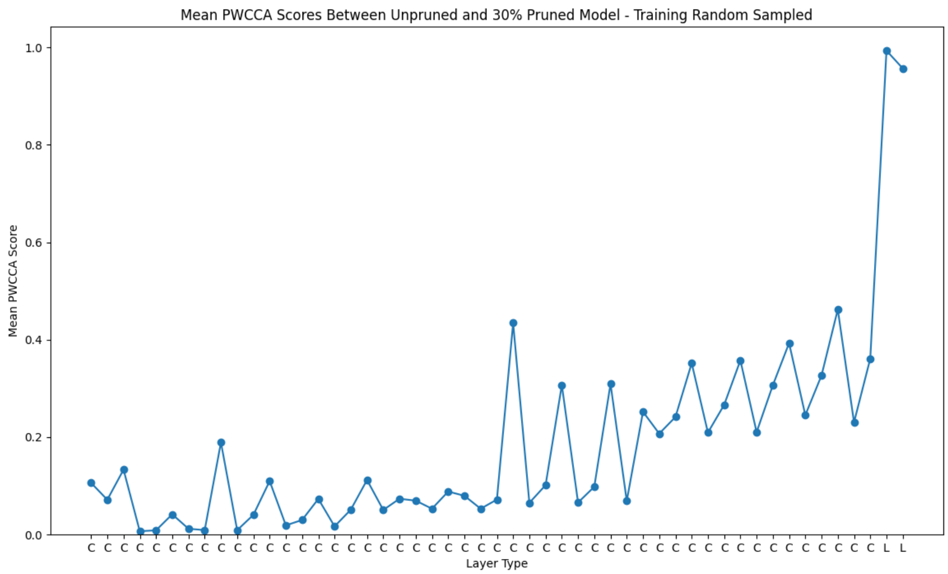
Figure 6.6: Plot of model sparsity on the x-axis versus total number of model parameters on the y-axis.

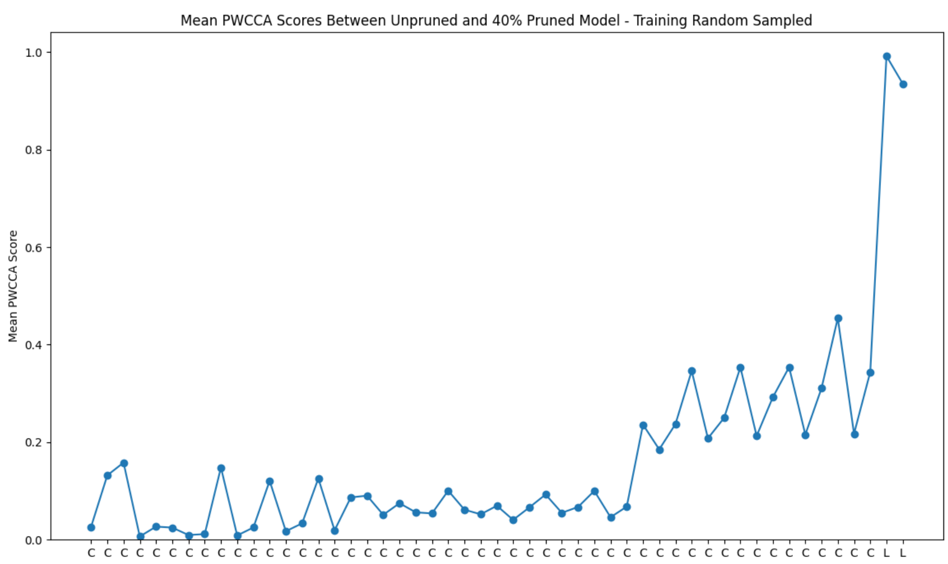
et al., where they find this to be the case when comparing networks using SVCCA, even for much smaller models [2]. This suggests that the effects of pruning cause pruned networks to learn a different representation of the dataset when compared to the representation learned by their unpruned counterpart. We also notice that in all plots, the linear layers at the end exhibit very high mean PWCCA similarity scores. This is also consistent with the findings of Ansuini et al., where they suggest that as the intermediate layers approach the final linear layers, the network is seemingly forced to guide the earlier representations from the convolutional layers to a common representation for the linear layers. In this sense, they refer to the final linear layers as "pivot points" that create common representations of the input data, leading to similar outputs that attribute to the similar accuracies between the unpruned and pruned networks. This concept of linear layers acting as pivots is a powerful notion, as it allows for different representations of the data in earlier layers between networks,

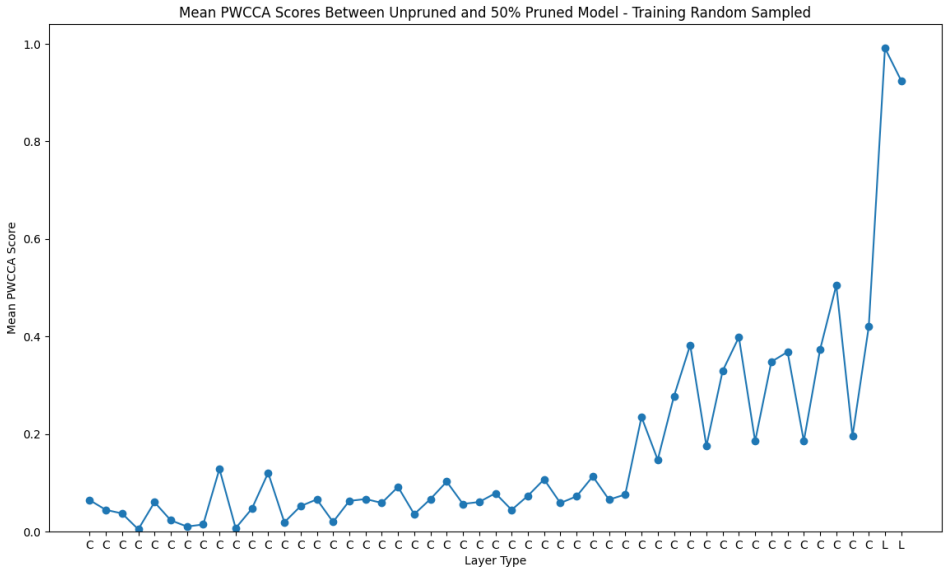
so long as the linear layers can guide the representations to be similar towards the end of the network.

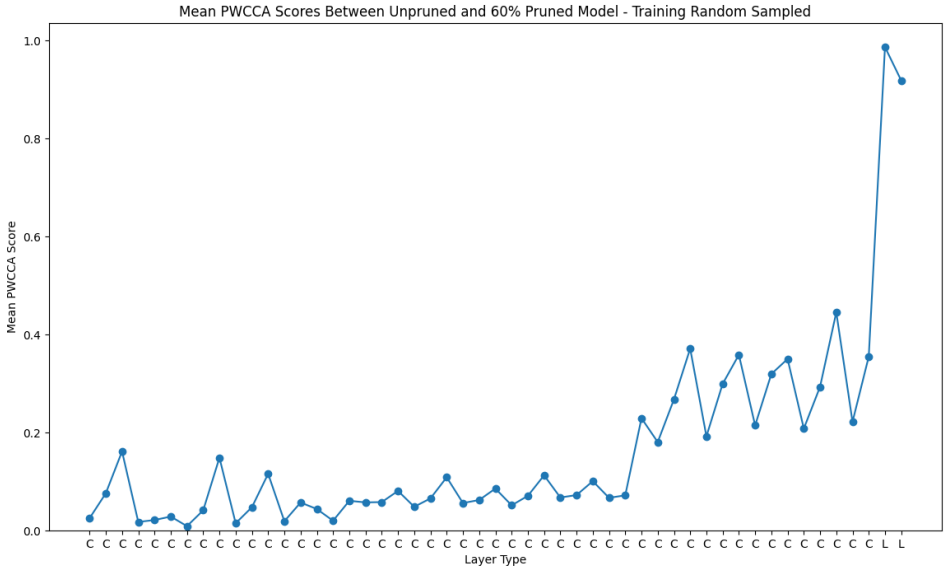





Mean PWCCA Scores Between Unpruned and 30% Pruned Model - Training Random Sampled
Mean PWCCA Score
Layer Type
0.0
0.2
0.4
0.6
0.8
1.0
C C C C C C C C C C C C C C C C C C C C C C C C C C C C C C C C C C C C C C C C C C C C C C C C C C L L


Mean PWCCA Scores Between Unpruned and 40% Pruned Model - Training Random Sampled
Mean PWCCA Score
Layer Type
0.0
0.2
0.4
0.6
0.8
1.0
C C C C C C C C C C C C C C C C C C C C C C C C C C C C C C C C C C C C C C C C C C C C C C C C C C L L


Mean PWCCA Scores Between Unpruned and 50% Pruned Model - Training Random Sampled
Mean PWCCA Score
Layer Type


Mean PWCCA Scores Between Unpruned and 60% Pruned Model - Training Random Sampled
Mean PWCCA Score
Layer Type

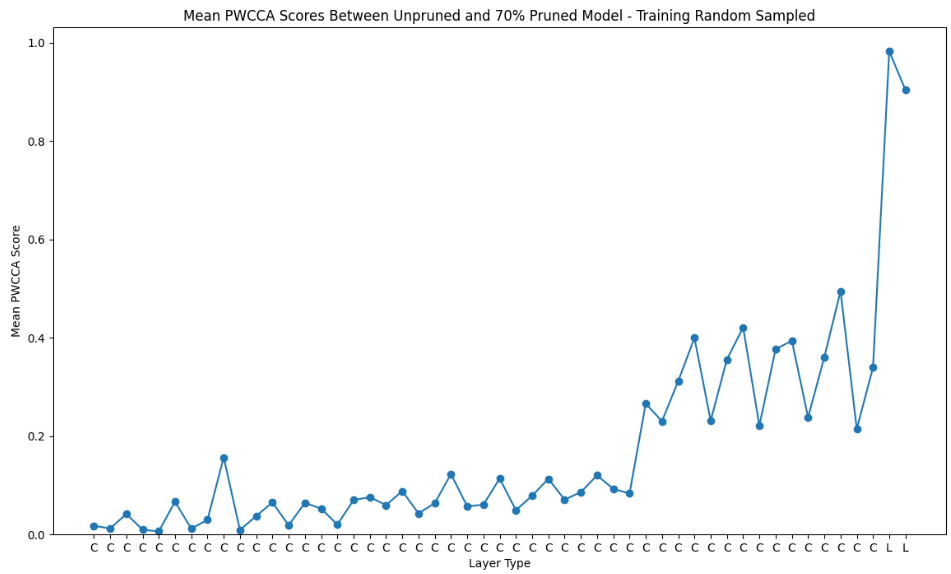


Figure 6.7: Plots of layer type on the x-axis and mean PWCCA score on the y-axis. A layer type of "C" denotes a convolutional layer, while a layer type of "L" denotes a linear layer. Layers on the x-axis are listed in order of their position in the neural network.

6.3 Limitations

Throughout the development of our neural network and research into the similarities between unpruned and pruned networks, there were a few limitations that will be discussed regarding the dataset used to train the model and the pruning techniques that were used to produce our pruned models. These limitations will be discussed in this section.

6.3.1 Investigating Loss and Dataset Limitations

When investigating the loss of our model, we first started by looking at images from the testing dataset that our model performed very well on, and images that our model performed poorly on. To do so, we manually looked at several images with the lowest

Figure 6.8: Examples of test set images with low L1 loss from our unpruned network. Red dots indicate the actual labeled waypoints, while blue dots indicate the predicted waypoints from our model.

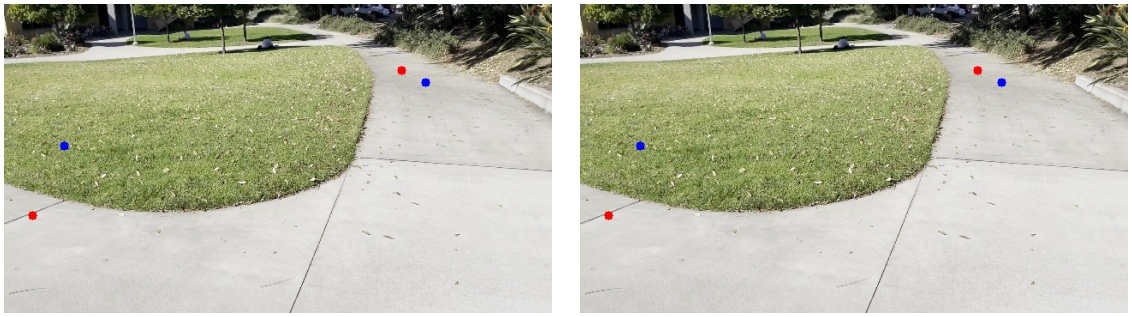


Figure 6.9: Examples of test set images high low L1 loss from our unpruned network. Red dots indicate the actual labeled waypoints, while blue dots indicate the predicted waypoints from our model.

losses and several images with the highest loss across our test dataset. Figure 6.8 shows examples of images with the lowest loss, with the red dots representing the actual labeled waypoints and blue dots representing predicted waypoints from our model. From our observations, images with the lowest loss were typically images that only had one labeled waypoint, where it was very clear and objective to see where the waypoint was. Figure 6.9 shows examples of images with the highest loss, which usually consisted of two to three-waypoint images.

Based on some of the above observations, and since the input images could contain one, two, or three waypoints, we also wanted to investigate how our base model was performing on each type of input image. To do this, we partitioned the test dataset by waypoint count, where all images with one labeled waypoint coordinate would be

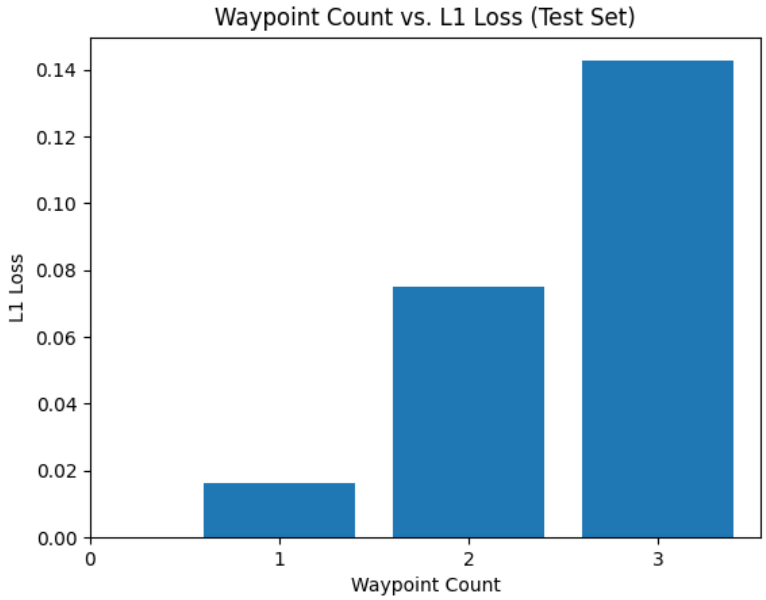


Figure 6.10: Bar chart of average L1 loss for each waypoint dataset formed from the test dataset. The waypoint count dataset is on the x-axis, and the average L1 loss for each dataset is on the y-axis.

one dataset, and so on for images with two and three waypoint counts. Then, we computed the average L1 loss over each waypoint-count dataset. Using these values, we plotted a bar chart of the waypoint count versus the average L1 loss over all images with that waypoint count from the test dataset, shown by Figure 6.10. What we found was that our model performed worse when faced with input images that had more labeled waypoints. While starting with relatively low loss on one-waypoint images, the loss gradually increased when evaluated over the two-waypoint images, and increased even more on the three-waypoint images. This is likely due to the difficulty of the regression problem increasing. With only one waypoint the model must only predict a single location, whereas finding multiple waypoints in an image accurately can be much more difficult.

In addition to the difficulty of the problem increasing as the number of waypoints needing to be predicted increases, the distribution of the dataset likely partially at-

tributed to the higher loss on multi-waypoint images. The test dataset consists of 199 one-waypoint images, 174 two-waypoint images, and 85 three-waypoint images. Additionally, the training dataset consists of 1543 one-waypoint images, 1512 two-waypoint images, and 603 three-waypoint images, and the validation dataset consists of 194 one-waypoint images, 180 two-waypoint images, and 83 three-waypoint images. From these counts, we can see that there are less three-waypoint images in the datasets than there are one and two waypoint images. This could partially explain why the loss for the three-waypoint images is higher. However, it is also worth noting that there are an almost equal number of one and two-waypoint images, yet the loss for two-waypoint images is still higher than the loss for one-waypoint images. This potentially reinforces the idea that as the model must predict more waypoints, the difficulty of the regression problem increases, resulting in increased loss.

To better see the loss for one, two, and three-waypoint images, and to reinforce the above observations, we also plotted histograms for each of the waypoint datasets, as seen in Figure 6.11. Here, we plotted the L1 loss in 0.05 increments on the x-axis, and the number of images with those loss values on the y-axis. Here we can also see that the vast majority of one-waypoint images exhibit very low L1 loss, most being under 0.05. The average loss for all the one-waypoint images was 0.016. Similarly, many of the two-waypoint images also have loss below 0.05, but contain several more higher loss images, with a max image loss of 0.369, and an average loss of 0.075. Finally, we see that the three-waypoint images seem to have more images in the higher loss bins, with a max image loss of 0.351 and an average loss of 0.142. We believe that having more three-waypoint images would be beneficial and would help decrease the overall loss of our model on three-waypoint images.

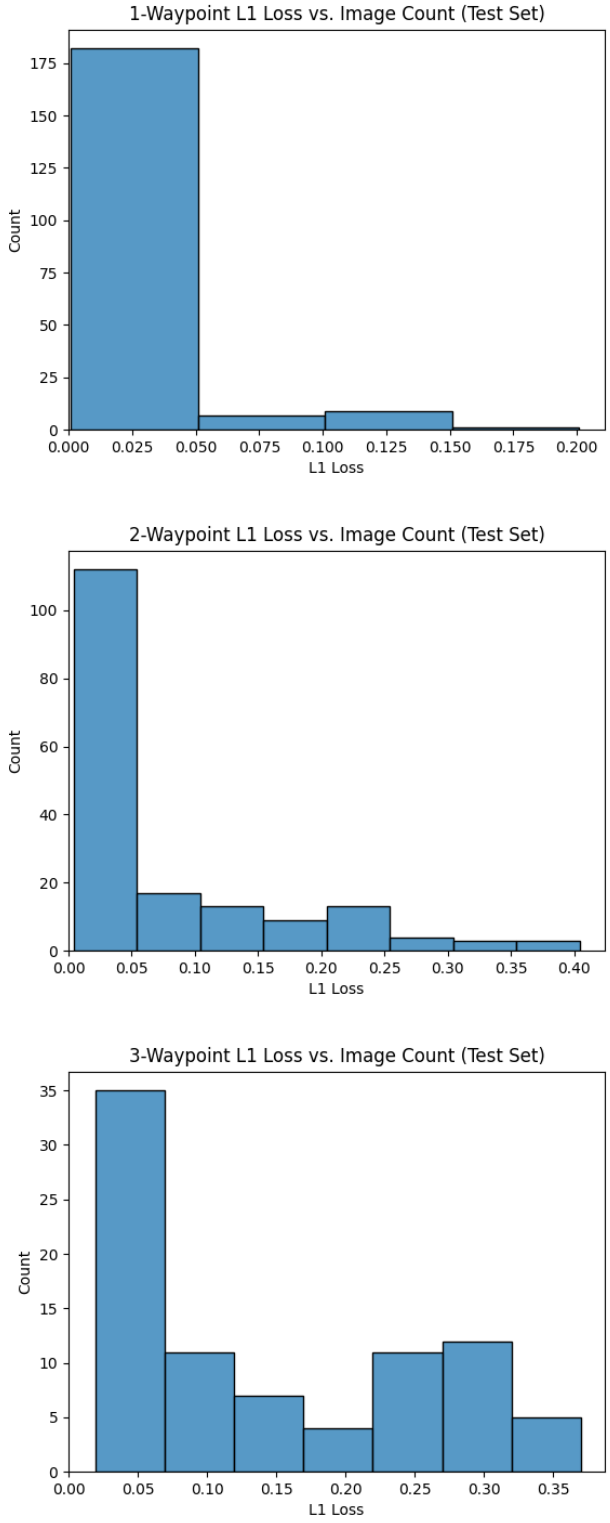


Figure 6.11: Histograms of L1 loss over each waypoint dataset on the x-axis and count of images with that loss on the y-axis.

Chapter 7

CONCLUSIONS

In this thesis, we developed a neural network to predict the waypoint coordinates of different paths given an input image, to later be used to help guide an autonomous robotic vehicle. To ensure that this network was small enough and fast enough to run on an embedded system, we used L1 Norm pruning to reduce the size and inference time of the base network. Upon pruning our network at numerous different levels of sparsity, we investigated the similarity of the learned representations of unpruned and pruned deep neural networks using a statistical technique known as projection weighted canonical correlation analysis (PWCCA).

For our base neural network, we created a network based off of a lightweight variant of EfficientNet that was effective at predicting up to three distinct path waypoints given an input image, achieving an average L1 loss value of 0.0637 [44]. This model had a total model size of 15MB and took 22.48 milliseconds on average to perform inference on a batch size of 16 data points. After pruning this model, we were able to create numerous compressed models, with one of the most promising performing the same inference 10 milliseconds faster and taking up one half of the size, while still retaining near-identical loss. Furthermore, several of the pruned models operate faster with fewer parameters, at the cost of a slight increase in loss.

Once our base network and unpruned networks were produced, we sought to further the work of Ansuini et al. by investigating the similarity between pruned and unpruned networks [2]. We did so by performing an in-depth layer-by-layer analysis between our unpruned network and each pruned network, computing the mean

PWCCA similarity score for each convolutional and linear layer. Our results reinforce some of the findings that Ansuini et al. showed for small convolutional networks using SVCCA; however in our case we showed that similar findings hold true for deep neural networks when using PWCCA. Namely, these findings are that the linear layers at the end of the networks exhibit high similarity in the activations that these layers produce over a target dataset. This suggests that for both shallow and deep convolutional neural networks, linear layers may act as a "pivot" that guide the different representations from prior convolutional layers to a more similar representation towards the end of the network, producing comparable network outputs and performance. From our experiments, we also observe two new trends. The first is that the PWCCA similarities between the layer activations of unpruned and pruned networks are quite high for pointwise linear projection (PWL) layers, which occur at the end of inverted residual blocks [38], but diminish as the network pruning becomes more aggressive. Further study into this relationship between inverted residual block representations and network pruning could be useful to better understand why this is the close. The second trend we observed was that in all cases, there was a progressive trend upwards in PWCCA similarity between unpruned and pruned networks as we progressed along the layers, suggesting that as these networks get closer to their final layers, their representations are guided to become more similar, potentially explaining how unpruned and pruned networks are able to achieve similar performance when all is said and done.

Chapter 8

FUTURE WORK

While ultimately we found our results and observations quite successful and insightful, there are some aspects of this work that could be improved and expanded upon, given ample time and resources. This chapter will discuss some of these potential improvements and future efforts that can be done to further the work done in this thesis.

8.1 Dataset Growth

As noted in section 6.4.1, we observed that it was harder for our model to predict as accurately in the presence of images with more waypoints, suggesting that prediction on more waypoints may have increased the difficulty of the problem. However, we also had fewer examples of three-waypoint images within our dataset, which we believe partially contributed to the higher loss for three-waypoint images with our model. Further, we noted that even though our dataset contained an almost equal number of examples for one and two-waypoint images, the average loss over two-waypoint images was higher than that for one-waypoint images, potentially reinforcing this hypothesis of problem difficulty. To help mitigate this issue, the most immediate step would be to have more examples of three-waypoint images. While there are less naturally occurring examples of three-waypoint paths in the geographical area of interest, future additions to this dataset could more heavily focus on these areas to collect more of these image types. Additionally, it could be worth investigating if scaling the number of images with respect to the waypoint count would help the model achieve lower

loss on those images. By this, we mean doing something along the lines of collecting X number of one-waypoint images, 2X number of two-waypoints images, and so on. However, we must be careful with this technique, as it might cause the network to overfit to the data with more waypoints, as there would be more examples of them within the dataset.

8.2 Investigating the Role of Certain Layers within Similarity

During our in-depth layer-by-layer analysis between our unpruned network and each pruned network, we noticed several interesting results that could incur future research. The first was the high PWCCA similarity spikes between the unpruned and pruned network activations from pointwise linear projection (PWL) layers that occur at the end of inverted residual blocks [38]. While these spikes in similarity decrease as the pruned networks become more sparse, it could be worth investigating why these spikes are occurring regardless, and the potential relationship between the expansion and compression/projection of features as a result of the inverted residual block and a more similar representation between pruned and unpruned networks. The other observation we noticed was that linear layers towards the end of deep convolutional neural networks act as pivots that guide the dissimilar representations produced from earlier convolutional layers towards a more similar representation that results in similar model results and performance. This reinforced similar findings of linear layers acting as pivots for shallow CNNs using SVCCA by Ansuini et al., though we showed that it also held true for deep CNNs using PWCCA [2]. However, it could also be worth investigating if the presence of linear layers earlier in a deep CNN also produce similar representations of the data, or if this only holds true for linear layers at the end of the network.

8.3 Comparing Deep Neural Networks with other Similarity Metrics

While our results unpruned and pruned neural networks using PWCCA as a measure of similarity, other similarity metrics for network layers exist. As discussed in section 3.2, normal canonical correlation analysis (CCA) and singular vector canonical correlation analysis (SVCCA) serve as other metrics that, to the best of our knowledge, have not been used to investigate the similarities between unpruned and pruned network layers. Another similarity metric that shows promise is centered kernel alignment (CKA), proposed by Kornblith et al. in 2019 [20]. Further work could use one or all of these techniques to reinforce the results found through PWCCA, and potentially offer new insights about these similarities that PWCCA may not address.

BIBLIOGRAPHY

[1] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org.

[2] A. Ansuini, E. Medvet, F. A. Pellegrino, and M. Zullich. On the similarity between hidden layers of pruned and unpruned convolutional neural networks. In *ICPRAM*, 2020.

[3] M. Augasta and T. Kathirvalavakumar. Pruning algorithms of neural networks — a comparative study. *Central European Journal of Computer Science*, 3:105–115, 09 2013.

[4] D. Bau, B. Zhou, A. Khosla, A. Oliva, and A. Torralba. Network dissection: Quantifying interpretability of deep visual representations. *CoRR*, abs/1704.05796, 2017.

[5] D. W. Blalock, J. J. G. Ortiz, J. Frankle, and J. V. Guttag. What is the state of neural network pruning? *CoRR*, abs/2003.03033, 2020.

[6] A. Bragagnolo and C. A. Barbano. Simplify: A python library for optimizing pruned neural networks. *SoftwareX*, 17:100907, 2022.

[7] N. Chandra. Node classification on relational graphs using deep-rgcns. Feb 2021.

[8] F. Chollet et al. Keras, 2015.

[9] E. Dogan, H. F. Ugurdag, and H. Unlu. Deep compression for pytorch model deployment on microcontrollers. *CoRR*, abs/2103.15972, 2021.

[10] J. Frankle and M. Carbin. The lottery ticket hypothesis: Training pruned neural networks. *CoRR*, abs/1803.03635, 2018.

[11] Google. Monterey bay, Accessed Feb. 11, 2022. [Online].

[12] J. Gou, B. Yu, S. J. Maybank, and D. Tao. Knowledge distillation: A survey. *CoRR*, abs/2006.05525, 2020.

[13] S. Han, J. Pool, J. Tran, and W. J. Dally. Learning both weights and connections for efficient neural networks. *CoRR*, abs/1506.02626, 2015.

[14] H. Hotelling. Relations between two sets of variates. *Biometrika*, 28(3/4):321, Dec 1936.

[15] S. A. Janowsky. Pruning versus clipping in neural networks. *Phys. Rev. A*, 39:6600–6603, Jun 1989.

[16] G. Kahn, P. Abbeel, and S. Levine. BADGR: an autonomous self-supervised learning-based navigation system. *CoRR*, abs/2002.05700, 2020.

[17] V. Kamath, V. S, and V. Manjunath. Transferred fusion learning using skipped networks. 11 2020.

[18] E. Karnin. A simple procedure for pruning back-propagation trained neural networks. *IEEE Transactions on Neural Networks*, 1(2):239–242, 1990.

[19] J. Kerfs. Models for pedestrian trajectory prediction and navigation in dynamic environments. 05 2017.

[20] S. Kornblith, M. Norouzi, H. Lee, and G. E. Hinton. Similarity of neural network representations revisited. *CoRR*, abs/1905.00414, 2019.

[21] P. Koutsovasilis and M. Beitelschmidt. Comparison of model reduction techniques for large mechanical systems. *Multibody System Dynamics*, 20(2):111–128, 2008.

[22] A. Krizhevsky, I. Sutskever, and G. E. Hinton. Imagenet classification with deep convolutional neural networks. In F. Pereira, C. Burges, L. Bottou, and K. Weinberger, editors, *Advances in Neural Information Processing Systems*, volume 25. Curran Associates, Inc., 2012.

[23] Y. LeCun, Y. Bengio, and G. Hinton. Deep learning. *Nature*, 521:436–44, 05 2015.

[24] H. Li, A. Kadav, I. Durdanovic, H. Samet, and H. P. Graf. Pruning filters for efficient convnets. *CoRR*, abs/1608.08710, 2016.

[25] W. Liu, Z. Wang, X. Liu, N. Zeng, Y. Liu, and F. E. Alsaadi. A survey of deep neural network architectures and their applications. *Neurocomputing*, 234:11–26, 2017.

[26] I. Loshchilov and F. Hutter. Fixing weight decay regularization in adam. *CoRR*, abs/1711.05101, 2017.

[27] A. Marchisio, M. A. Hanif, M. Martina, and M. Shafique. Prunet: Class-blind pruning method for deep neural networks. In *2018 International Joint Conference on Neural Networks (IJCNN)*, pages 1–8, 2018.

[28] L. Mehrabyan. Understanding how schools work with canonical correlation analysis, Mar 2020.

[29] Microsoft. Neural Network Intelligence, 1 2021.

[30] A. S. Morcos, M. Raghu, and S. Bengio. Insights on representational similarity in neural networks with canonical correlation, 2018.

[31] M. C. MOZER and P. SMOLENSKY. Using relevance to reduce network size automatically. *Connection Science*, 1(1):3–16, 1989.

[32] ODSCCommunity. What is pruning in machine learning?, 2020.

[33] J. O'Neill. An overview of neural network compression. *CoRR*, abs/2006.03669, 2020.

[34] K. O'Shea and R. Nash. An introduction to convolutional neural networks. *CoRR*, abs/1511.08458, 2015.

[35] M. Paganini and J. Forde. Streamlining tensor and network pruning in pytorch. *arXiv preprint arXiv:2004.13770*, 2020.

[36] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Kopf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala. Pytorch: An imperative style, high-performance deep learning library. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, editors, *Advances in Neural Information Processing Systems 32*, pages 8024–8035. Curran Associates, Inc., 2019.

[37] M. Raghu, J. Gilmer, J. Yosinski, and J. Sohl-Dickstein. Svcca: Singular vector canonical correlation analysis for deep learning dynamics and interpretability, 2017.

[38] M. Sandler, A. G. Howard, M. Zhu, A. Zhmoginov, and L. Chen. Inverted residuals and linear bottlenecks: Mobile networks for classification, detection and segmentation. *CoRR*, abs/1801.04381, 2018.

[39] I. H. Sarker. Deep learning: A comprehensive overview on techniques, taxonomy, applications and research directions. *SN Computer Science*, 2(6):420, Nov 2021.

[40] A. See, M. Luong, and C. D. Manning. Compression of neural machine translation models via pruning. *CoRR*, abs/1606.09274, 2016.

[41] C. Siegel, J. Daily, and A. Vishnu. Adaptive neuron apoptosis for accelerating deep learning on large scale systems. *CoRR*, abs/1610.00790, 2016.

[42] P. S. Statistics. Lesson 13: Canonical correlation analysis.

[43] C. M. J. Tan and M. Motani. DropNet: Reducing neural network complexity via iterative pruning. In H. D. III and A. Singh, editors, *Proceedings of the 37th International Conference on Machine Learning*, volume 119 of *Proceedings of Machine Learning Research*, pages 9356–9366. PMLR, 13–18 Jul 2020.

[44] M. Tan and Q. V. Le. Efficientnet: Rethinking model scaling for convolutional neural networks. *CoRR*, abs/1905.11946, 2019.

[45] R. Wightman. Pytorch image models. https://github.com/rwightman/pytorch-image-models, 2019.

[46] Y. Wu, M. Schuster, Z. Chen, Q. V. Le, M. Norouzi, W. Macherey, M. Krikun, Y. Cao, Q. Gao, K. Macherey, J. Klingner, A. Shah, M. Johnson, X. Liu, Łukasz Kaiser, S. Gouws, Y. Kato, T. Kudo, H. Kazawa, K. Stevens, G. Kurian, N. Patil, W. Wang, C. Young, J. Smith, J. Riesa, A. Rudnick, O. Vinyals, G. Corrado, M. Hughes, and J. Dean. Google's neural machine translation system: Bridging the gap between human and machine translation. *CoRR*, abs/1609.08144, 2016.

[47] Q. Yang, Y. Zhang, W. Dai, and S. J. Pan. *Transfer Learning*. Cambridge University Press, 2020.

[48] M. D. Zeiler and R. Fergus. Visualizing and understanding convolutional networks. *CoRR*, abs/1311.2901, 2013.