ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA (ICAI)

MÁSTER EN INGENIERÍA DE TELECOMUNICACIÓN

ESTANDARIZACIÓN DE MÉTODOS DE PODADO DE REDES NEURALES

Autor: José Javier González Ortiz

Director: Javier Matanza Domingo

Madrid

Julio 2020

Declaro, bajo mi responsabilidad, que el Proyecto presentado con el título

Estandarización de Métodos de Poda de Redes Neuronales

en la ETS de Ingeniería - ICAI de la Universidad Pontificia Comillas en el

curso académico (2019/20) es de mi autoría, original e inédito y

no ha sido presentado con anterioridad a otros efectos.

El Proyecto no es plagio de otro, ni total ni parcialmente y la información que ha sido

tomada de otros documentos está debidamente referenciada.

Fdo.: José Javier González Ortiz Fecha: ……/ ……/ ……

Autorizada la entrega del proyecto

EL DIRECTOR DEL PROYECTO

Fdo.: Javier Matanza Domingo Fecha: ……/ ……/ ……

Vº Bº del Coordinador de Proyectos

Fdo.: Javier Matanza Domingo Fecha: ……/ ……/ ……

ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA (ICAI)

MÁSTER EN INGENIERÍA DE TELECOMUNICACIÓN

ESTANDARIZACIÓN DE MÉTODOS DE PODA DE REDES NEURONALES

Autor: José Javier González Ortiz

Director: Javier Matanza Domingo

Madrid

Julio 2020

ESTANDARIZACIÓN DE MÉTODOS DE PODA DE REDES NEURONALES

Autor: González Ortiz, Jose Javier

Director: Matanza Domingo, Javier

Entidad Colaboradora: Massachusetts Institute of Technology, Computer Science and Artificial Intelligence Lab

RESUMEN

En este trabajo presentamos un framework que facilita el desarollo y evaluación de métodos de poda para redes neuronales. Debido a la falta de estandarización en la literatura existente sobre los métodos de poda, es díficil comparar y evaluar los métodos desarrollados de una forma rigurosa. Nuestra librería es capaz de realizar experimentos de forma estandarizada, controlando posibles factores de confusión tales como la elección de conjunto de datos, modelo predictivo o las métricas de evaluación. Demostramos la eficacia de la metodología desarrollada implementando varios métodos de poda de redes neuronales. Nuestros resultados indican cómo el uso un proceso de evaluación estandarizdo es necesario para evitar alcanzar conclusiones incorrectas a la hora de comparar métodos de poda.

Palabras clave: Redes Neuronales, Métodos de Poda, Deep Learning, PyTorch

1. Introducción

La inferencia de redes neuronales profundas a menudo requiere grandes cantidades de computación y memoria. Esto aumenta los costes de infraestructura, además de convertir el despliegue de estas redes en entornos con recursos limitados en todo un desafío. Los métodos de *poda*, es decir, la reducción del tamaño de una red mediante la eliminación de parámetros, es un enfoque popular para reducir estos requisitos. Típicamente, los métodos comienzan con una red grande y precisa, y luego tratan de hacerla más pequeña con la menor pérdida de precisión posible.

Muchos de los trabajos publicados utilizan diferentes conjuntos de datos, podan arquitecturas de red muy variadas y no utilizan las mismas métricas para medir el rendimiento, lo que dificulta en gran medida la comparación de técnicas. En este trabajo presentamos una librería de código abierto diseñada para facilitar a los investigadores la comprobación de los métodos de poda de manera que facilite las comparaciones directas. Además, utilizamos esta librería para examinar algunas de las heurísticas de poda existentes. Nuestros resultados empíricos demuestran que el rendimiento relativo de un método de poda puede variar en función de los conjuntos de datos, redes, pesos iniciales y métricas reportadas, lo que confirma la necesidad de una evaluación estandarizada de los métodos de poda.

2. Descripción del sistema

ShrinkBench es una biblioteca de Python diseñada para facilitar la evaluación de los métodos de poda de redes neuronales. ShrinkBench ha sido diseñado con dos objetivos principales: 1) permitir la creación rápida de prototipos de métodos de poda de redes neuronales, y 2) facilitar el uso de métricas estandarizadas, conjuntos de datos, arquitecturas de red y configuraciones de ajustes. El objetivo de ShrinkBench es facilitar a los investigadores e ingenieros la implementación de estrategias de poda y la evaluación de su efectividad en distintos escenarios. La biblioteca proporciona herramientas para realizar experimentos utilizando combinaciones estandarizadas de conjuntos de datos y modelos, controlando factores potencialmente confusos como la afinación de hiperparámetros o pesos iniciales.

Los resultados pueden entonces ser agregados para producir comparaciones de métodos usando varias métricas y ratios de compresión. ShrinkBench trabaja con arquitecturas estándar de PyTorch[PGC+17], simplemente requiriendo al usuario que proporcione las máscaras que indican qué parámetros se deben mantener, y opcionalmente nuevos valores para los pesos restantes. Para facilitar la tarea de computar el conjunto de máscaras de parámetros, ShrinkBench proporciona primitivas para acceder a parámetros, activaciones y gradientes. Dadas las máscaras de parámetros, ShrinkBench entrena y ajusta los modelos asegurando que los parámetros enmascarados no contribuyen a la salida de la red y no se actualizan durante la retropropagación. Usandp estas máscaras, ShrinkBench aplica la poda automáticamente, actualiza la red de acuerdo con un entrenamiento o un ajuste fino, y calcula métricas a través de muchos modelos, conjuntos de datos, semillas aleatorias y ratios de compresión.

3. Resultados

Testeamos ShrinkBench podando más de 800 redes usando las bases descritas y variando los conjuntos de datos, redes, ratios de compresión, pesos iniciales y semillas aleatorias. Al hacerlo, identificamos varios escollos asociados con las prácticas experimentales que son actualmente comunes en la literatura pero que pueden evitarse usando la evaluación de ShrinkBench. Es una práctica común describir la cantidad de poda informando de bien la reducción del número total de parámetros o del número de FLOPs teóricos. Si estos parámetros están suficientemente correlacionados, uno solo es suficiente para caracterizar la eficacia de un método de poda. Nuestros experimentos indican que esto no es necesariamente cierto. Como muestra la Figura 1, los métodos de poda global son más precisos que los métodos por capas para un tamaño de modelo dado, pero los métodos por capas son más precisos para una aceleración teórica dada. Esta discrepancia está relacionada con el hecho de que la poda de las capas con mayores entradas puede resultar en un mayor ahorro en los cálculos. Además, muchos métodos informan de los resultados de sólo un pequeño número de conjuntos de datos, modelos y ratios de compresión. Si el rendimiento relativo de los diferentes métodos fuera constante en todos estos factores, esto no sería problemático. Pero no lo son. La figura 2 muestra la precisión para aumentar los ratios de compresión para CIFAR-VGG [Zag15] y ResNet56 en CIFAR10. Como podemos observar, el método dominante cambia dependiendo de la red y la relación de compresión, lo que demuestra la necesidad de evaluar un método de poda en varias combinaciones de modelos y conjuntos de datos para asegurar que no depende de la arquitectura. De manera similar, esto muestra la necesidad de informar sobre la métrica de varios ratios de compresión para caracterizar la curva de compensación a medida que aumenta la poda.

4. Conclusiones

Con ShrinkBench pretendemos proporcionar un conjunto de herramientas que faciliten la tarea de producir evaluaciones exhaustivas y reproducibles de los enfoques de poda propuestos. Además, nuestros resultados ponen de manifiesto la necesidad de realizar experimentos normalizados al evaluar los métodos de poda de la red neuronal debido a la presencia de posibles factores de confusión.

5. Referencias

[DZW18] Bin Dai, Chen Zhu y David Wipf. “Compressing neural networks using the variational information bottleneck”. En: *arXiv preprint arXiv:1802.10399* (2018).

[FDRC19] Jonathan Frankle, Gintare Karolina Dziugaite, Daniel M Roy y Michael Carbin. “The Lottery Ticket Hypothesis at Scale”. En: *arXiv preprint arXiv:1903.01611* (2019).

[GEH19] Trevor Gale, Erich Elsen y Sara Hooker. *The State of Sparsity in Deep Neural Networks*. 2019. arXiv: `1902.09574 [cs.LG]`.

[HMD15] Song Han, Huizi Mao y William J Dally. “Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding”. En: *arXiv preprint arXiv:1510.00149* (2015).

[HPTD15] Song Han, Jeff Pool, John Tran y William Dally. “Learning both weights and connections for efficient neural network”. En: *Advances in neural information processing systems*. 2015, págs. 1135-1143.

[LAGT19] Namhoon Lee, Thalaiyasingam Ajanthan, Stephen Gould y Philip H. S. Torr. “A Signal Propagation Perspective for Pruning Neural Networks at Initialization”. en. En: *arXiv:1906.06307 [cs, stat]* (jun. de 2019). arXiv: 1906.06307.

[LAT18] Namhoon Lee, Thalaiyasingam Ajanthan y Philip HS Torr. “SNIP: Single-shot network pruning based on connection sensitivity”. En: *arXiv preprint arXiv:1810.02340* (2018).

[LSZ+18] Zhuang Liu, Mingjie Sun, Tinghui Zhou, Gao Huang y Trevor Darrell. “Rethinking the value of network pruning”. En: *arXiv preprint arXiv:1810.05270* (2018).

[PGC+17] Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga y Adam Lerer. “Automatic differentiation in pytorch”. En: (2017).

[YLC+18] Ruichi Yu, Ang Li, Chun-Fu Chen, Jui-Hsin Lai, Vlad I Morariu, Xintong Han, Mingfei Gao, Ching-Yung Lin y Larry S Davis. “Nisp: Pruning networks using neuron importance score propagation”. En: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 2018, págs. 9194-9203.

[Zag15] Sergey Zagoruyko. *92.45 % on CIFAR-10 in Torch*. `https://torch.ch/blog/2015/07/30/cifar.html`. Accessed: 2019-07-22. Jul. de 2015.

STANDARDIZATION OF NEURAL NETWORK PRUNING METHODS

Author: González Ortiz, Jose Javier

Supervisor: Matanza Domingo, Javier

Collaborating Entity: Massachusetts Institute of Technology, Computer Science and Artificial Intelligence Lab

ABSTRACT

Neural network pruning consists of reducing the size of a network by removing parameters. In this work, we introduce ShrinkBench, an open-source library to facilitate standardized evaluation of neural network pruning methods. ShrinkBench simplifies using standardized datasets, pretrained models, and evaluation metrics for implementing pruning methods. In addition to describing the functionality of ShrinkBench, we demonstrate its utility by using it to implement and evaluate several pruning methods. We show that ShrinkBench's comprehensive evaluation can prevent common pitfalls when comparing pruning methods.

Keywords: Neural Networks, Pruning, Deep Learning, Standardized Evaluation, PyTorch

1. Introduction

Deep neural network inference often requires large amounts of computation and memory. This not only increase infrastructure costs, but also makes deploying these networks to resource-constrained environments challenging. *Pruning*, i.e. reducing the size of a network by removing parameters, is a popular approach for reducing these requirements. Typically, methods start with a large and accurate network, and then try to make it smaller with as little loss in accuracy as possible. Unfortunately, unlike in some other areas of machine learning, there is no widely agreed upon set of benchmarks or measurements that facilitate comparison of methods [DZW18; LSZ+18]. Different papers use different datasets, prune different network architectures, and report performance in differing ways, making it almost impossible to know under what conditions one technique is better than another. In this work, we present ShrinkBench, an open-source library designed to make it easier for researchers to test pruning methods in a way that facilitates direct comparisons. In addition to describing ShrinkBench, we use it to examine some existing pruning heuristics. Our empirical findings demonstrate that a pruning method's performance can vary across datasets, networks, initial weights and reported metrics—confirming the need for standardized evaluation of pruning methods.

2. System Overview

ShrinkBench is a Python library designed to ease evaluation of neural network pruning methods. ShrinkBench was designed with two goals in mind: 1) enable rapid prototyping of neural network pruning methods, and 2) facilitate the use of standardized metrics, datasets, network architectures, and finetuning setups. Using ShrinkBench, researchers and engineers can easily implement pruning strategies and evaluate their effectiveness across a wide range of scenarios. The library provides tools for running experiments using standardized dataset-model combinations and controlling for potentially confounding factors such as finetuning hyperparameters or initial weights. Results can then be aggregated to produce method comparisons across several metrics and amounts of compression. ShrinkBench works with off-the-shelf PyTorch[PGC+17] model architectures, simply requiring the user to provide parameter masks that indicate which parameters to keep, and optionally new values for the remaining weights. To facilitate the task of computing the set of parameter masks, ShrinkBench provides primitives for retrieving parameters, activations, and gradients. From the parameter masks, ShrinkBench can train and fine-tune models ensuring that masked parameters do not contribute to the output of the network and are not updated during backpropagation. Given these masks, ShrinkBench will automatically apply the pruning, update the network according to a training or fine-tuning setup, and compute metrics across many models, datasets, random seeds, and compression ratios.

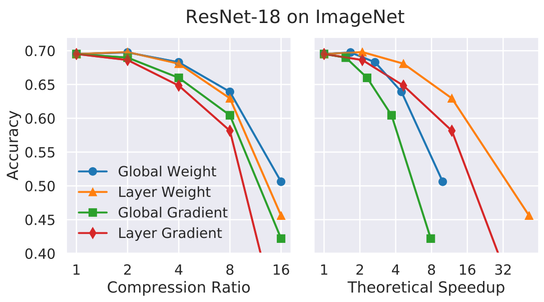

Figure 1: Accuracy for ResNet-18 on ImageNet for several compression ratios and their respective theoretical speedups.

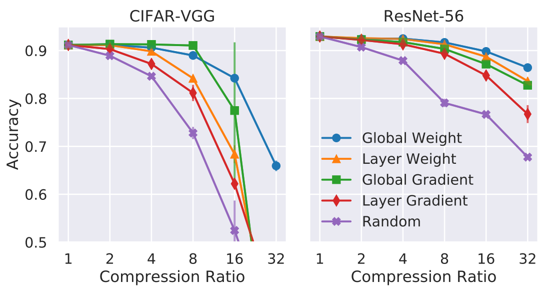

Figure 2: Top 1 Accuracy on CIFAR10 for several compression ratios and two different model architectures: ResNet-56 & VGG.

3. Results

We tested ShrinkBench by pruning over 800 networks using the described baselines and varying datasets, networks, compression ratios, initial weights, and random seeds. In doing so, we identified various pitfalls associated with experimental practices that are currently common in the literature but that can be avoided by using ShrinkBench's evaluation. It is a common practice to describe the amount of pruning by reporting either the reduction in the total number of parameters or in the number of theoretical FLOPs. If these metrics are sufficiently correlated, reporting only one is sufficient to characterize the effectiveness of a pruning method. Our experiments indicate that this not necessarily true. As Figure 1 shows, Global pruning methods are more accurate than Layerwise methods for a given model size, but Layerwise methods are more accurate for a given theoretical speedup. This discrepancy is related to the fact that pruning layers with larger inputs can result in higher computational savings. Moreover, many methods report results on only a small number of datasets, models, and compression ratios. If the relative performance of different methods were constant across these factors, this would not be problematic. But they are not. Figure 2 shows accuracy for increasing compression ratios for CIFAR-VGG [Zag15] and ResNet56 on CIFAR10. As we can observe, the dominant method changes depending on the network and compression ratio, showing the need to evaluate a pruning method on various model and dataset combinations to ensure it is not architecture dependent. Similarly, this shows the need to report metrics for several compression rations to characterize the trade-off curve as pruning increases.

4. Conclusions

With ShrinkBench we aim to provide a toolkit that facilitates the task of producing comprehensive and reproducible evaluations of proposed pruning approaches. Furthermore, our results evidence the need for standardized experiments when evaluating neural network pruning methods because of the presence of potential confounding factors.

5. References

[DZW18] Bin Dai, Chen Zhu, and David Wipf. "Compressing neural networks using the variational information bottleneck". In: *arXiv preprint arXiv:1802.10399* (2018).

[FDRC19] Jonathan Frankle, Gintare Karolina Dziugaite, Daniel M Roy, and Michael Carbin. "The Lottery Ticket Hypothesis at Scale". In: *arXiv preprint arXiv:1903.01611* (2019).

[GEH19] Trevor Gale, Erich Elsen, and Sara Hooker. *The State of Sparsity in Deep Neural Networks*. 2019. arXiv: `1902.09574 [cs.LG]`.

[HMD15] Song Han, Huizi Mao, and William J Dally. "Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding". In: *arXiv preprint arXiv:1510.00149* (2015).

[HPTD15] Song Han, Jeff Pool, John Tran, and William Dally. "Learning both weights and connections for efficient neural network". In: *Advances in neural information processing systems*. 2015, pp. 1135–1143.

[LAGT19] Namhoon Lee, Thalaiyasingam Ajanthan, Stephen Gould, and Philip H. S. Torr. "A Signal Propagation Perspective for Pruning Neural Networks at Initialization". en. In: *arXiv:1906.06307 [cs, stat]* (June 2019). arXiv: 1906.06307.

[LAT18] Namhoon Lee, Thalaiyasingam Ajanthan, and Philip HS Torr. "SNIP: Single-shot network pruning based on connection sensitivity". In: *arXiv preprint arXiv:1810.02340* (2018).

[LSZ+18] Zhuang Liu, Mingjie Sun, Tinghui Zhou, Gao Huang, and Trevor Darrell. "Rethinking the value of network pruning". In: *arXiv preprint arXiv:1810.05270* (2018).

[PGC+17] Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. "Automatic differentiation in pytorch". In: (2017).

[YLC+18] Ruichi Yu, Ang Li, Chun-Fu Chen, Jui-Hsin Lai, Vlad I Morariu, Xintong Han, Mingfei Gao, Ching-Yung Lin, and Larry S Davis. "Nisp: Pruning networks using neuron importance score propagation". In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 2018, pp. 9194–9203.

[Zag15] Sergey Zagoruyko. *92.45% on CIFAR-10 in Torch*. `https://torch.ch/blog/2015/07/30/cifar.html`. Accessed: 2019-07-22. July 2015.

“The purpose of a storyteller is not to tell you how to think, but to give you questions to think upon.”

— *Brandon Sanderson*

“In life, there are no solutions, only trade-offs.”

— *C.G.P. Grey*

“The brick walls are there for a reason. The brick walls are not there to keep us out; the brick walls are there to give us a chance to show how badly we want something. The brick walls are there to stop the people who don’t want it badly enough. They are there to stop the other people!”

— *Randy Pausch*

A mis padres, por su apoyo inasequible al desaliento

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Deep Neural Networks have played a major role in the progress the machine learning field and industry have experienced over the past decade. In order to achieve competitive performance these networks require environments amounts of computation and memory Huang et al., 2018. These constraints not only make neural network infrastructure costs to be quite sizable but also limit the adoption of these models to more constrained environments such as mobile platforms or embedded devices Han et al., 2015; Sze et al., 2017; Yang et al., 2017.

Neural network pruning is one of the techniques practitioners have devised for reducing the high computation and memory requirements these networks entail. Usually, one starts with a large and accurate network that has been pretrained on a dataset of interest and the goal is to produce a smaller network without dropping the accuracy of the model in a significant manner. Conceptually, neural network pruning has existed since the 1980s (Janowsky, 1989; Karnin, 1990; Mozer and Smolensky, 1989a, 1989b) but it has seen an explosion of interest in the recent years because of the widespread use of deep neural networks.

1.1 Overview

Deep neural network inference often requires large amounts of computation and memory. *Pruning*, i.e. reducing the size of a network by removing parameters, is a popular approach for reducing these requirements. Typically, methods start with a large and accurate network, and then try to make it smaller with as little loss in accuracy as possible. Unfortunately, unlike in some other areas of machine learning, there is no widely agreed upon set of benchmarks or measurements that facilitate comparison of methods Dai et al., 2018; Liu et al., 2019. Different papers use different datasets, prune different network architectures, and report performance in differing ways, making it almost impossible to know under what conditions one technique is better than another.

In this work, we present ShrinkBench, an open-source library designed to make it easier for researchers to test pruning methods in a way that facilitates direct comparisons. ShrinkBench simplifies using standardized datasets, pretrained models, and evaluation metrics for implementing pruning methods. In addition to describing the functionality of ShrinkBench, we demonstrate its utility by using it to implement and evaluate several pruning methods. Our results evidence the need for standardized evaluation given that a pruning method's performance can vary across datasets, networks, initial weights and reported metrics. In addition to describing ShrinkBench, we use it to examine some existing pruning heuristics.

1.2 Motivation

The motivation of this work stems from the intersection of the exponentially growing interest in deploying neural network models in a efficient and scalable way, and the current state of the neural network pruning literature which lacks consistent findings and recommendations. In order for pruning techniques to continue improving upon each other there needs to be clear benchmarks and ways for researchers to compare their results in a fair manner. Many machine learning subfields have converged on specific datasets and metrics for results to be comparable. In some scenarios machine learning competitions have been held to foster the development of novel and more accurate approaches, such as the *ImageNet Large Scale Visual Recognition Challenge* Russakovsky et al., 2015. Moreover, the experimental setting and guidelines from the ImageNet competition have been quite valuable for other fields of machine learning and computer vision research who have used them to great benefit.

Unfortunately, there seems to be a significant fragmentation in the neural network pruning literature on the choice of datasets, models, metrics and hyperparameter settings which make understanding the current state-of-the-art a fundamentally difficult task. Since the interest in neural network predictive models seems to be only increasing, we deem important to 1) perform a systematic analysis of the neural network pruning literature to distill any consistent findings and detect potential pitfalls and 2) develop a framework that facilitates the development and evaluation of neural network pruning methods in a rigorous and standardized way.

1.3 Contributions

The main contributions of this work are:

- We identify what are some the current consistent findings and existing limitations in the state-of-the-art of neural neural pruning methods.

- We develop ShrinkBench, an open-source evaluation framework for neural network pruning methods that enables researchers and practitioners to efficiently perform standardized evaluation of their methods.

- In addition to describing ShrinkBench, we use it to examine some existing pruning heuristics. Our empirical findings show that a pruning method's performance can vary across datasets, networks, initial weights and reported metrics—confirming the need for standardized evaluation of pruning methods.

1.4 Structure

This thesis has the following structure: Chapter 2 provides some background knowledge to contextualize this work; Chapter 3 provides a literature metanalysis that motivates the need for standardized evaluation of pruning methods; Chapter 4 throughly describes ShrinkBench's implementation and design choices; Chapter 5 document the experimental setup followed for all the obtained results, including the baseline pruning methods implemented with ShrinkBench; Chapter 6 includes results from pruning over 800 neural networks and presents potential pitfalls caused by lurking confounding factors in the neural network evaluation pipeline; Chapter 7 closes by discussing potential avenues of future work.

Chapter 2

Background

Before proceeding further, we will first introduce some background on neural network pruning and a high-level overview of how existing pruning methods typically work. We will also detail how evaluation of pruning is usually performed.

2.1 Definitions

We define a neural network *architecture* as a function family $f(x; \cdot)$. The architecture consists of the configuration of the network's parameters and the sets of operations it uses to produce outputs from inputs, including the arrangement of parameters into convolutions, activation functions, pooling, batch normalization, etc. Example architectures include AlexNet and ResNet-18. We define a neural network *model* as a particular parameterization of an architecture, i.e., $f(x; W)$ for specific parameters W. Neural network *pruning* entails taking as input a model $f(x; W)$ and producing a new model $f(x; M \odot W')$. In practice, rather than using an explicit mask, pruned parameters of W are fixed to zero or removed entirely.

2.2 High-Level Algorithm

There are many methods of producing a pruned model $f(x; M \odot W')$ from an initially untrained model $f(x; W_0)$, where W_0 is sampled from an initialization distribution $\mathcal{D}$. Nearly all neural network pruning strategies derive from Algorithm 2.1 (Han et al., 2015). In this algorithm, the network is first trained to convergence. Afterwards, each parameter or structural element in the network is issued a score, and the network is pruned based on these scores. Pruning reduces the accuracy of the network, so it is trained further (known as *fine-tuning*) to recover. The process of pruning and fine-tuning is often iterated several times, gradually reducing the network's size.

Many papers propose slight variations of this algorithm. For example, some papers prune periodically during training (Gale et al., 2019) or even at initialization (Lee et al., 2018). Others modify the network to explicitly include additional parameters that encourage sparsity and serve as a basis for scoring the network after training (D. Molchanov et al., 2017).

Algorithm 2.1 Pruning and Fine-Tuning

```
Require: $N$: the number of iterations of pruning
Require: $X$: the dataset on which to train and fine-tune
1: $W \leftarrow initialize()$
2: $W \leftarrow trainToConvergence(f(X;W))$
3: $M \leftarrow 1^{|W|}$
4: for $i$ in $1$ to $N$ do
5:     $M \leftarrow prune(M, score(W))$
6:     $W \leftarrow fineTune(f(X;M \odot W))$
7: return $M, W$
```

2.3 Differences Betweeen Pruning Methods

Within the framework of Algorithm 2.1, pruning methods vary primarily in their choices regarding sparsity structure, scoring, scheduling, and fine-tuning.

- **Structure.** Some methods prune individual parameters (*unstructured pruning*). Other methods consider parameters in groups (*structured pruning*), removing entire neurons, filters, or channels to exploit hardware and software optimized for dense computation (Y. He et al., 2017; Li et al., 2016).

- **Scoring.** It is common to score parameters based on their absolute values, trained importance coefficients, or contributions to network activations or gradients as well as other measures of saliency. Some pruning methods compare scores locally, pruning a fraction of the parameters with the lowest scores within each structural subcomponent of the network (e.g., layers) (Han et al., 2015). Others consider scores globally, comparing scores to one another irrespective of the part of the network in which the parameter resides (Frankle & Carbin, 2019; Lee et al., 2018).

- **Scheduling.** Pruning methods differ in the amount of the network to prune at each step. Some methods prune all desired weights at once in a single step (Liu et al., 2019). Others prune a fixed fraction of the network iteratively over several steps (Han et al., 2015) or vary the rate of pruning according to a more complex function (Gale et al., 2019).

- **Fine-tuning.** For methods that involve fine-tuning, it is most common to continue to train the network using the trained weights from before pruning. Alternative proposals include rewinding the network to an earlier state (Frankle et al., 2019a) and reinitializing the network entirely (Liu et al., 2019). Liu et al. (2019) suggest that, when performing structured pruning, reinitializing the network weights before fine-tuning has no detrimental affect on the accuracy of the eventual network.

2.4 Evaluating Pruning

Pruning can accomplish many different goals, including reducing the storage footprint of the neural network, the computational cost of inference, the energy requirements of inference, etc. Each of these goals favors different design choices and requires different evaluation metrics. For example, when reducing the storage footprint of the network, all parameters can be treated equally, meaning one should evaluate the overall compression ratio achieved by pruning. However, when reducing the computational cost of inference, different parameters may have different impacts. For instance, in convolutional layers, filters applied to spatially larger inputs are associated with more computation than those applied to smaller inputs. Thus, given two convolutional filters pruned by the same amount, the one with a larger input size will lead to higher computational savings.

Regardless of the goal, pruning imposes a trade-off between model efficiency and quality, with pruning increasing the former while (typically) decreasing the latter. This means that a pruning method is best characterized not by a single model it has pruned, but by a family of models corresponding to different points on the efficiency-quality curve.

To quantify efficiency, most papers report at least one of two metrics. The first is the number of multiply-adds (often referred to as FLOPs) required to perform inference with the pruned network. The second is the fraction of parameters pruned. To measure quality, nearly all papers report changes in Top-1 or Top-5 image classification accuracy.

As others have noted (Figurnov et al., 2016; Han et al., 2015; Y. He et al., 2018; Kim et al., 2015; Lebedev et al., 2014; Louizos et al., 2017; Luo et al., 2017; Wen et al., 2016; Yang et al., 2017), these metrics are far from perfect. Parameter and FLOP counts are a loose proxy for real-world latency, throughout, memory usage, and power consumption. In particular, despite many papers assuming otherwise, the number parameters has little bearing on memory of activations. Similarly, image classification is only one of the countless tasks to which neural networks have been applied. However, because the overwhelming majority of published papers focus on these metrics, our analysis necessarily does as well.

To evaluate computational costs, many papers measure the number of multiply-adds (often referred to as FLOPs) required to perform inference with the pruned network. In practice, nearly all pruning papers measure the compression ratio or FLOP reduction (or equivalently the associated speedup) achieved by pruning, so we focus on these metrics throughout the paper. However, we acknowledge (as do others (Han et al., 2015; Kim et al., 2015; Louizos et al., 2017; Yang et al., 2017)) that these metrics may not capture the performance considerations of running neural network inference on modern hardware. A crucial aspect of the pruning problem is that it usually requires optimizing a trade-off between efficiency and quality. That is, achieving a particular efficiency threshold may require removing parameters such that performance on the network's task declines. This means that, when comparing pruning methods, it is important to consider the entire pruning/quality trade-off curve for different amounts of pruning.

Chapter 3

State of the Art

Many neural network pruning methods have been described in the recent years given the exponential growth of deep learning research. In this chapter we analyze the current state of the literature for neural network pruning methods, identifying what are some the current consistent findings and existing limitations in the literature.[1]

3.1 Effectivity of Pruning

Pruning, i.e. reducing the size of a network by removing parameters, is a popular approach for reducing the requirements of training and performing inference on large deep neural networks. Typically, methods start with a large and accurate network, and then try to make it smaller with as little loss in accuracy as possible. Unfortunately, unlike in some other areas of machine learning, there is no widely agreed upon set of benchmarks or measurements that facilitate comparison of methods (Dai et al., 2018; Liu et al., 2019).

One of the clearest findings about pruning is that it works. More precisely, there are various methods that can significantly compress models with little or no loss of accuracy. In fact, for small amounts of compression, pruning can sometimes *increase* accuracy (Han et al., 2015; Suzuki et al., 2018). This basic finding has been replicated in a large fraction of the papers in our corpus. Along the same lines, it has been repeatedly shown that, at least for large amounts of pruning, many pruning methods outperform random pruning (Frankle et al., 2019a; Gale et al., 2019; Y. He et al., 2017; Mariet & Sra, 2015; Suau et al., 2018; Yu et al., 2018). Interestingly, this does not always hold for small amounts of pruning (Morcos et al., 2019). Similarly, pruning all layers uniformly tends to perform worse than intelligently allocating parameters to different layers (Gale et al., 2019; Han et al., 2015; Li et al., 2016; Luo et al., 2017; P. Molchanov et al., 2016) or pruning globally (Frankle & Carbin, 2019; Lee et al., 2018). Lastly, when holding the number of fine-tuning iterations constant, many methods produce pruned models that outperform retraining from scratch with the same sparsity pattern (Frankle & Carbin, 2019; Y. He et al., 2017; Louizos et al., 2017; Luo et al., 2017; Yu et al., 2018; X. Zhang et al., 2015) (at least with a large enough amount of pruning (Suau et al., 2018)). Retraining from scratch in this context

[1] *Parts of the analysis in this chapter are from a related piece of work (Blalock, D., Ortiz, J. J. G., Frankle, J., & Guttag, J. (2020). What is the State of Neural Network Pruning? Third Conference on Machine Learning and Systems). We thank our collaborators for the great insights presented here.*

means training a fresh, randomly-initialized model with all weights clamped to zero throughout training, except those that are nonzero in the pruned model.

Another consistent finding is that sparse models tend to outperform dense ones for a fixed number of parameters. Lee et al. (2019) show that increasing the nominal size of ResNet-20 on CIFAR-10 while sparsifying to hold the number of parameters constant decreases the error rate. Kalchbrenner et al. (2018) obtain a similar result for audio synthesis, as do Gray et al. (2017) for a variety of additional tasks across various domains. Perhaps most compelling of all are the many results, including in Figure 3.1 (Blalock et al., 2020), showing that pruned models can obtain higher accuracies than the original models from which they are derived. This demonstrates that sparse models can not only outperform dense counterparts with the same number of parameters, but sometimes dense models with even more parameters.

While there do appear to be a few general and consistent findings in the pruning literature, by far the clearest takeaway is that pruning papers rarely make direct and controlled comparisons to existing methods. This lack of comparisons stems largely from a lack of experimental standardization and the resulting fragmentation in reported results. This fragmentation makes it difficult for even the most committed authors to compare to more than a few existing methods.

3.2 Pruning vs Architecture Changes

One current unknown about pruning is how effective it tends to be relative to simply using a more efficient architecture. These options are not mutually exclusive, but it may be useful in guiding one's research or development efforts to know which choice is likely to have the larger impact. Along similar lines, it is unclear how pruned models from different architectures compare to one another---i.e., to what extent does pruning offer similar benefits across architectures?

Figure 3.1 (Blalock et al., 2020) suggests several conclusions. First, it reinforces the conclusion that pruning can improve the time or space vs accuracy tradeoff of a given architecture, sometimes even increasing the accuracy. Second, it suggests that pruning generally does not help as much as switching to a better architecture. Finally, it suggests that pruning is more effective for architectures that are less efficient to begin with. Finally, it suggests that pruning is more effective at reducing the size of models (left column) than their number of FLOPs (right column). This is unsurprising,

While there do appear to be a few general and consistent findings in the pruning literature (see the previous section), by far the clearest takeaway is that pruning papers rarely make direct and controlled comparisons to existing methods. This lack of comparisons stems largely from a lack of experimental standardization and the resulting fragmentation in reported results. This fragmentation makes it difficult for even the most committed authors to compare to more than a few existing methods.

3.3 Metrics Fragmentation

As depicted in Figure 3.2 (Blalock et al., 2020), papers report a wide variety of metrics and operating points, making it difficult to compare results. Each column in this figure is one (dataset,

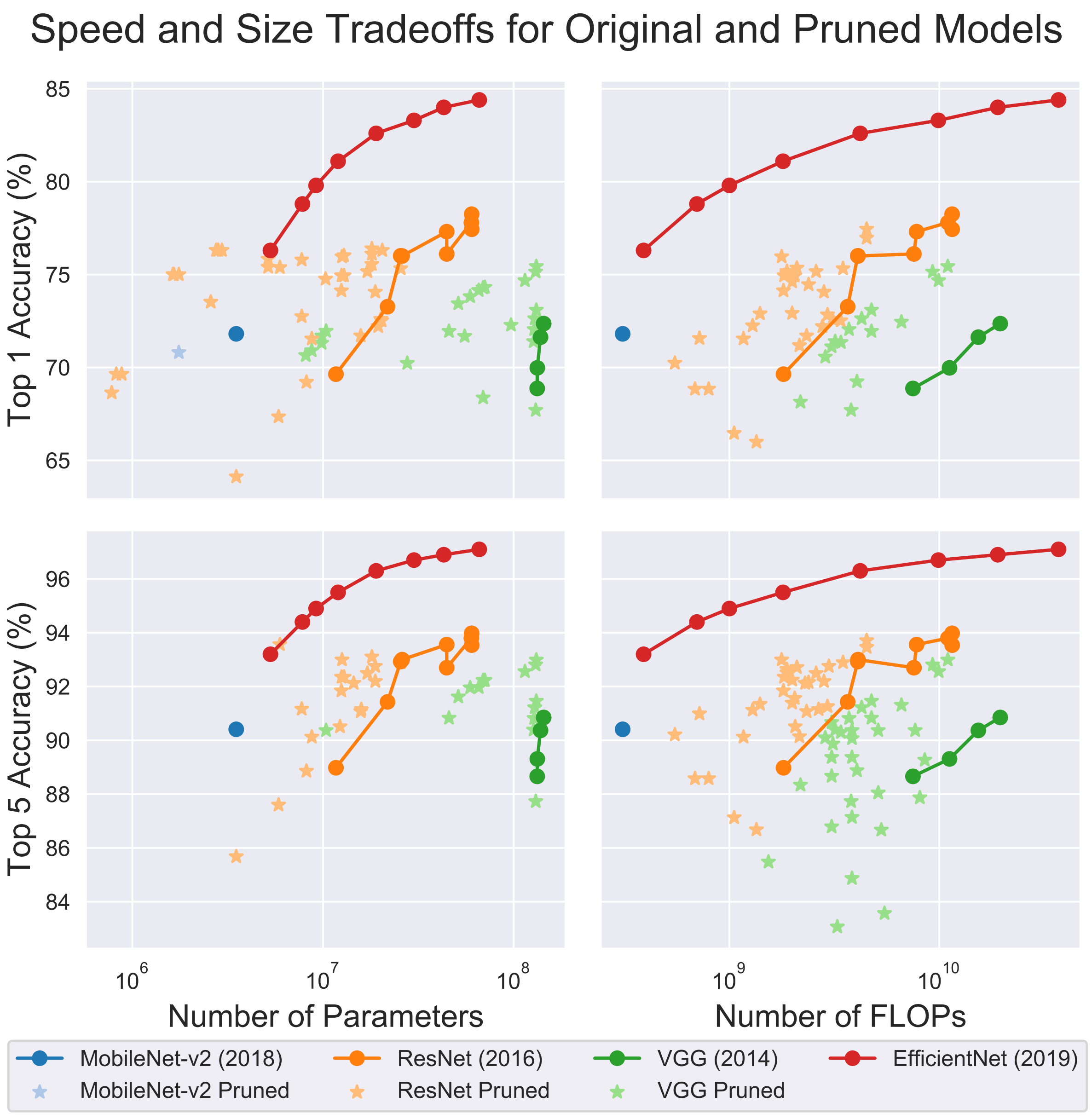

Figure 3.1: Size and speed vs accuracy tradeoffs for different pruning methods and families of architectures. Pruned models sometimes outperform the original architecture, but rarely outperform a better architecture. (Blalock et al., 2020).

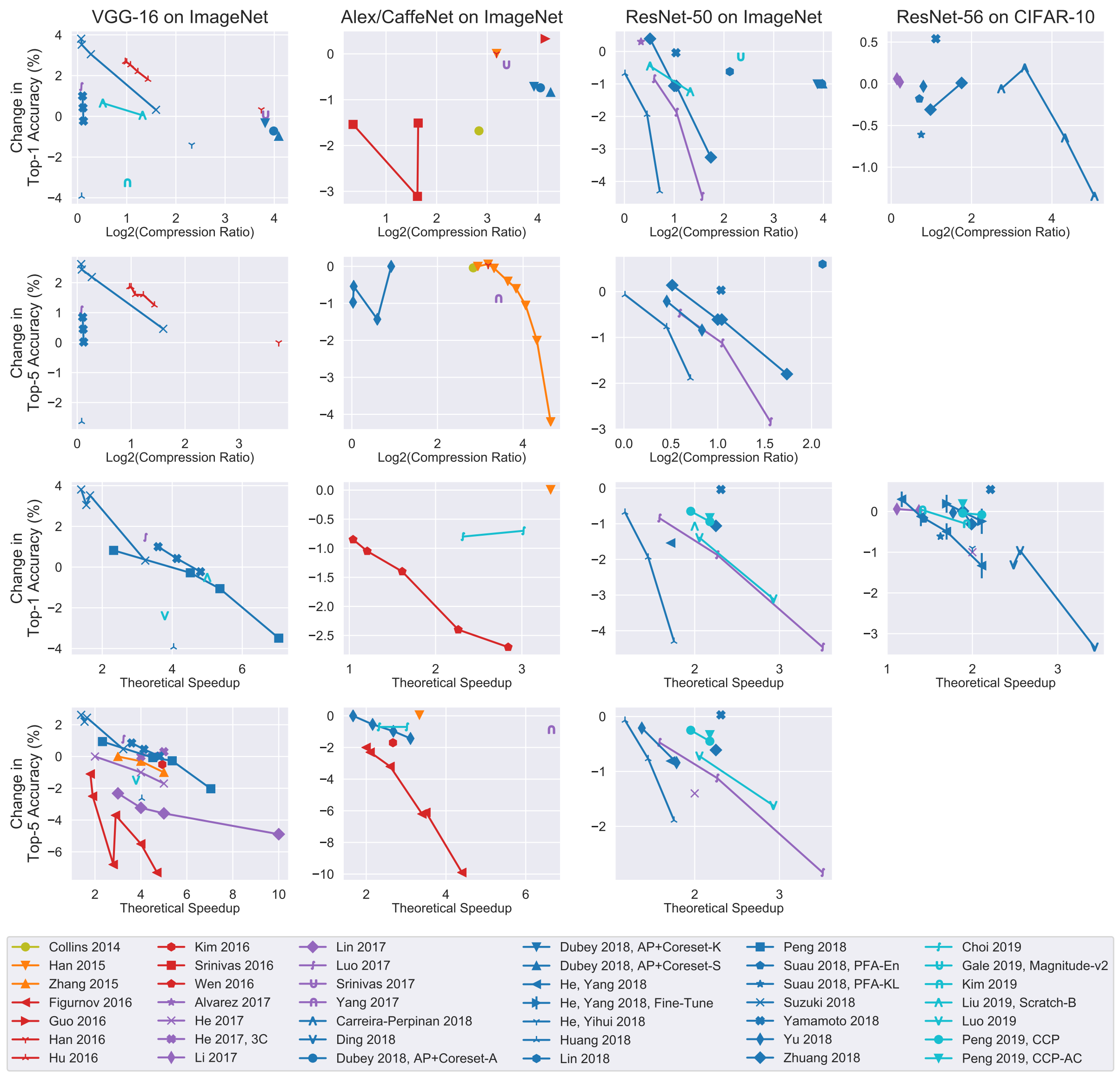

Figure 3.2: Fragmentation of results. Shown are all self-reported results on the most common (dataset, architecture) combinations. Each column is one combination, each row shares an accuracy metric (y-axis), and pairs of rows share a compression metric (x-axis). Up and to the right is always better. Standard deviations are shown for He 2018 on CIFAR-10, which is the only result that provides any measure of central tendency. (Blalock et al., 2020).

architecture) combination taken from the four most common combinations, excluding results on MNIST. Each row is one pair of metrics. Each curve is the efficiency vs accuracy trade-off obtained by one method. Methods are color-coded by year.

It is hard to identify any consistent trends in these plots, aside from the existence of a trade-off between efficiency and accuracy. A given method is only present in a small subset of plots. Methods from later years do not consistently outperform methods from earlier years. Methods within a plot are often incomparable because they report results at different points on the x-axis. Even when methods are nearby on the x-axis, it is not clear whether one meaningfully outperforms another since neither reports a standard deviation or other measure of central tendency. Finally, most papers in our corpus do not report any results with any of these common configurations.

3.4 Confounding Variables

Even when comparisons include the same datasets, models, metrics, and operating points, other confounding variables still make meaningful comparisons difficult. Some variables of particular interest include:

- Accuracy and efficiency of the initial model
- Data augmentation and preprocessing
- Random variations in initialization, training, and fine-tuning. This includes choice of optimizer, hyperparameters, and learning rate schedule.
- Pruning and fine-tuning schedule
- Deep learning library. Different libraries are known to yield different accuracies for the same architecture and dataset (Nola, 2016; Northcutt, 2019) and may have subtly different behaviors (Vryniotis, 2018).
- Subtle differences in code and environment that may not be easily attributable to any of the above variations (Crall, 2018; Jogeshwar, 2017; “Keras Exported Model shows very low accuracy in Tensorflow Serving,” 2017).

In general, it is not clear that any paper can succeed in accounting for all of these confounders unless that paper has both used the same code as the methods to which it compares and reports enough measurements to average out random variations. This is exceptionally rare, with Gale et al. (2019) and Liu et al. (2019) being arguably the only examples. Moreover, neither of these papers introduce novel pruning methods *per se* but are instead inquiries into the efficacy of existing methods.

There are at least two more empirical reasons to believe that confounding variables can have a significant impact. First, as one can observe in Figure 3.2, methods often introduce changes in accuracy of much less than 1% at reported operating points. This means that, even if confounders have only a tiny impact on accuracy, they can still have a large impact on which method appears better.

Second, as shown in Figure 3.3, existing results demonstrate that different training and fine-tuning settings can yield nearly as much variability as different methods. Specifically, consider 1) the variability introduced by different fine-tuning methods for unstructured magnitude-based pruning (Figure 3.3 top) and 2) the variability introduced by entirely different pruning methods (Figure 3.3 bottom). The variability between fine-tuning methods is nearly as large as the variability between pruning methods.

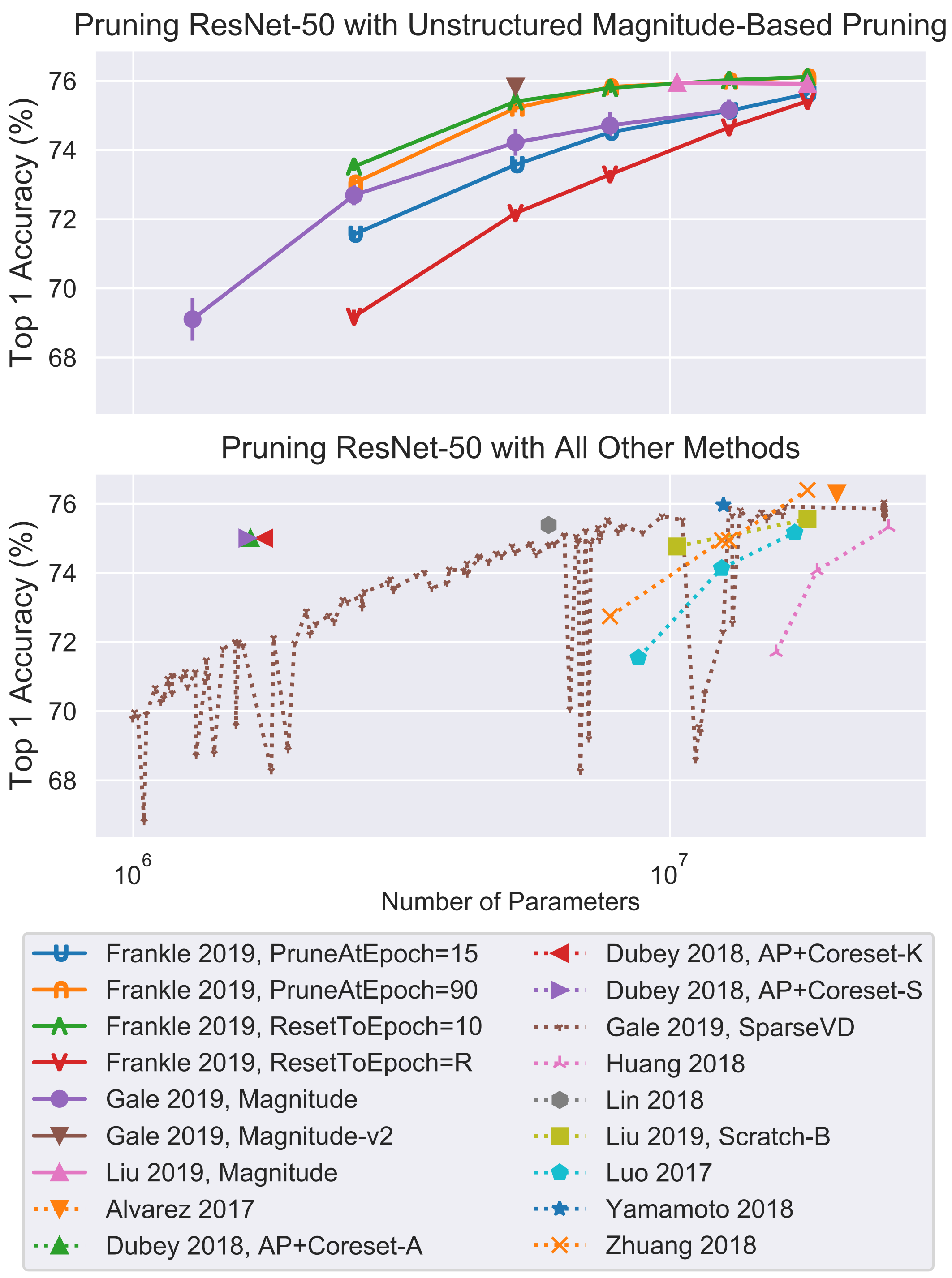

Figure 3.3: Pruning ResNet-50 on ImageNet. Methods in the upper plot all prune weights with the smallest magnitudes, but differ in implementation, pruning schedule, and fine-tuning. The variation caused by these variables is similar to the variation across different pruning methods, whose results are shown in the lower plot. All results are taken from the original papers. (Blalock et al., 2020).

Chapter 4

Standardizing Neural Network Pruning

As we have seen in the previous section, the task of correctly evaluating neural network pruning methods is a challenging one. A main barrier towards comparison is the fact that in most pruning methods, the main contribution is the pruning algorithm but one needs to build an entire machine learning pipeline in order to obtain results from using the pruning algorithm in practice.

In this chapter we describe ShrinkBench, an open-source library aimed at facilitating the development and evaluation of neural network pruning methods in a standardized and reproducible way. We will first describe how we implemented ShrinkBench, highlighting the design choices taken and describing how we believe those decisions should help the community adhering to best practices.

4.1 System Overview

ShrinkBench is a Python library designed to ease evaluation of neural network pruning methods. ShrinkBench was designed with two goals in mind: 1) enable rapid prototyping of neural network pruning methods, and 2) facilitate the use of standardized metrics, datasets, network architectures, and finetuning setups.

Using ShrinkBench, researchers can easily implement pruning strategies and evaluate their effectiveness across a wide range of scenarios (datasets, models and compression ratios). The library provides tools for running experiments using standardized dataset-model combinations and controlling for potentially confounding factors such as finetuning hyperparameters or initial weights. Results can then be aggregated to produce method comparisons like the ones presented in the results chapter. ShrinkBench works with off-the-shelf PyTorch (Paszke et al., 2017) model architectures, simply requiring the user to provide parameter masks that indicate which parameters to keep, and optionally new values for the remaining weights.

To facilitate the task of computing the set of parameter masks, ShrinkBench provides primitives for retrieving parameters, activations, and gradients. Since only the parameter masks are required, ShrinkBench supports arbitrary scoring functions, allocation of parameters across layers, and sparsity structures. From the parameter masks, ShrinkBench can train and fine-tune models ensuring that masked parameters do not contribute to the output of the network and are not updated during backpropagation. Given these masks, ShrinkBench will automatically apply the pruning, update the network according to a training or fine-tuning setup, and compute metrics across many models, datasets, random seeds, and compression ratios. Since the existing literature on pruning has mostly focused on computer vision tasks, ShrinkBench provides a set of vision-related dataset-model combinations with pretrained weights.

The entire source code and revision history of ShrinkBench is publicly available at https://github.com/JJGO/shrinkbench and can be used under the MIT License. The pretrained models for the networks trained in the CIFAR10 dataset can be found under https://github.com/JJGO/shrinkbench-models.

4.2 Pipeline Design

In order to design ShrinkBench we went back and analyzed how traditional neural network pruning baselines worked to identify what were the potential confounding factors that need to be abstracted away.

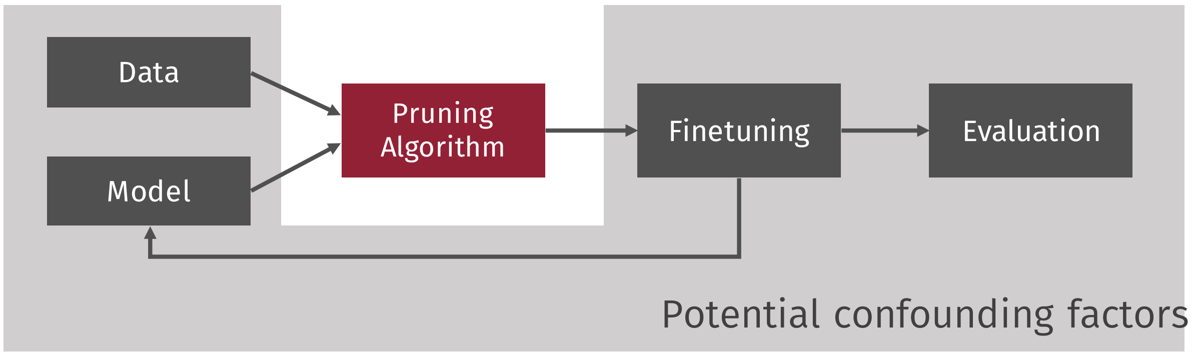

Figure 4.1: Typical diagram for a neural network pruning pipeline. Starting from a pretrained network on a set of *data*, a pruning algorithm is applied to the *model* that only preserves a sparse subset of the weights. After the weight pruning is done, the network is trained on the target dataset for a few epochs to restore some of the lost accuracy (*finetuning*). Finally, the network is evaluated on the heldout test set with a set of metrics which include both performance and efficiency.

As figure 4.1 depicts, a traditional pipeline for evaluating a pruning algorithm. Starting from a dataset-architecture combination, usually encoded as a pretrained model on the data, a pruning algorithm is applied to reduce the number of weights in the network. The pruned model is now trained for several epochs, to recover some of the lost accuracy introduced by the removal of weights. Here, training is done over the pruned topology, ensuring that the pruned weights remained pruned (i.e. at zero). This inner loop of pruning and finetuning can be repeated several times when weights are pruned in a iterative fashion instead of pruning all of them upfront.

However, usually the novel contribution in this entire pipeline is the choice of pruning algorithm [1] whilst the rest of the moving parts is a potential source of confusion that might become a barrier to further comparison. In ShrinkBench we prevent all the potential confounding factors from interacting with the design of the pruning algorithm as depicted in Figure 4.2. ShrinkBench provides standardized model and dataset combinations of associated complexities,

[1]Some methods do implement custom finetuning procedures, however these can be viewed as a iterative stateful pruning algorithm and thus can fit into the depicted pipeline.

preventing the pitfall of using a overparametrized network for a target dataset. The end user can also extend ShrinkBench by implementing custom dataset and model combinations, however ShrinkBench's design ensures that these additions do not affect the behavior of the rest of the pruning pipeline. ShrinkBench also provides tested evaluation metrics for measuring sparsity in a rigorous way as well as a variety of auxiliary utilities that facilitate the development of neural network pruning methods. Both of these will be described later in this chapter with corresponding examples.

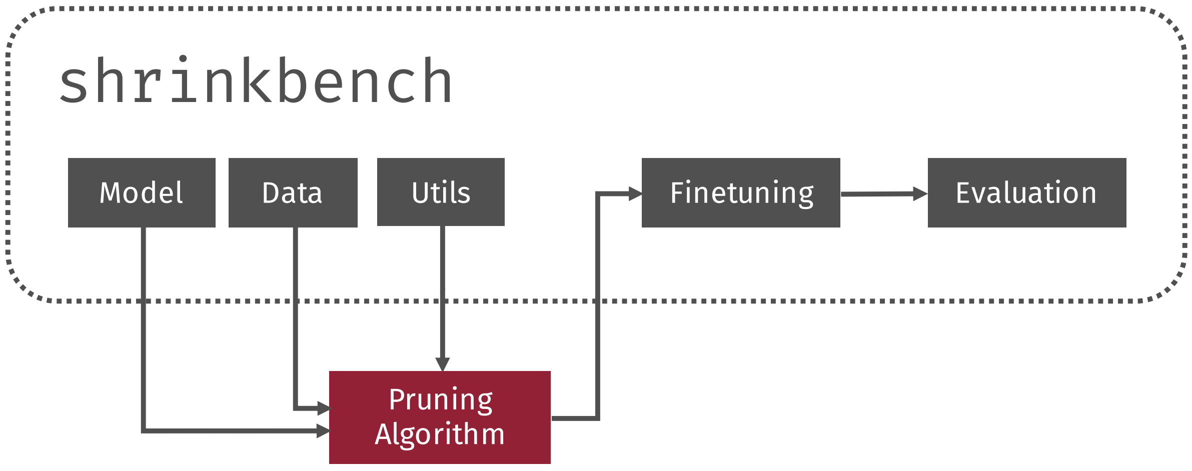

Figure 4.2: In our proposed framework, the API to the pruning algorithm is exposed in a way that facilitates the overall evalutaion of the methodology while ensuring that evaluation experiments are performed in a controlled and reproducible way.

4.3 Pruning through binary masks

As we discussed in the background chapter, a given pretrained neural network can be described as a parametric function $f(x;W)$ where W is the set of all parameters or weights of the network. The act of pruning produces a new model with the same architecture $f(x;W' \odot M)$ where $M \in \{0,1\}^{|W'|}$ is a binary mask that fixes some of the parameters to 0 and $\odot$ is the tensor elementwise product operator. In ShrinkBench, we make use of these binary masks not only as a useful abstraction to specify arbitrary pruning methods but also as a low level detail to ensure the correctness of the pruning procedure.

In practice, once a network is pruned, the mask abstraction is not needed anymore since the weights are either set to zero or removed from the specification of the architecture. However, when finetuning steps are required over a pruned model, steps need to taken so that the network stays pruned after every single optimizer iteration, namely after every forward and backward pass over a minibatch of data. In practice, layers are defined over dense tensor weights so unless structured pruning is used, in order to make use of the efficient hardware primitives that implement the layers, one needs to take steps to prevent gradients from reviving pruned weights.

There are different ways of achieve this. A simple naive way is to keep the masks in memory and perform the elementwise mask multiplication after every minibatch. This will zero out the pruned weights, undoing any gradient updates performed by the optimizer. However, because of the nature of Python interpreter and how GPU tensors work, this can be time consuming to do since one needs to iterate over each weight tensor and

In ShrinkBench we decided to explicitly implement the mask elementwise product multiplication as part of the architecture of the network. Despite this requiring more GPU memory to store the masks, implementing masking this way achieves a higher resource utilization and throughput since the masking operations can be computed directly in the GPU. To achieve this, we effectively modify the computation graph of the network replacing each weight with the elementwise product of said weight and its associated mask.

We show the correctness of this approach by proving that for masked weights the gradients will be zero while for the unpruned weights will remain as if the masks were not there. Given a weight tensor involved in some operation $y = f(x, w)$ where x is the input to the layer, y is the output and f is the function the layer implements, let us replace w by $w' = w \odot m$ where $m \in \{0,1\}^{|w|}$ is the weight mask. Now, when computing the gradients with respect to the weights, the mask will ensure no updates reach the masked weights. We can see this by deriving the gradient of the loss $\mathcal{L} \triangleq \mathcal{L}(f(x;W), y)$ with respect to the weights using the chain rule. Originally, the updates are as follows

$$\frac{\partial \mathcal{L}}{\partial w} = \frac{\partial \mathcal{L}}{\partial y}\frac{\partial y}{\partial w} \tag{4.1}$$

After substituting the masks we get

$$\begin{aligned}\frac{\partial \mathcal{L}}{\partial w} &= \frac{\partial \mathcal{L}}{\partial y}\frac{\partial y}{\partial w} \\ &= \frac{\partial \mathcal{L}}{\partial y}\left(\frac{\partial y}{\partial w'}\frac{\partial w'}{\partial w}\right) \\ &= \frac{\partial \mathcal{L}}{\partial y}\left(\frac{\partial y}{\partial w'}m\right) \\ &= \frac{\partial \mathcal{L}}{\partial y}\left(\frac{\partial y}{\partial w}m\right) \\ &= m\frac{\partial \mathcal{L}}{\partial y}\frac{\partial y}{\partial w}\end{aligned} \tag{4.2}$$

Which is the same expression except that is multiplied by the value of $m \in \{0,1\}$ so the gradient will be the expected value when $m = 1$ and zero otherwise. This depicted in Figure 4.3 where the same analysis is shown over the equivalent computation graph.

Technically, PyTorch does not have static computation graph since it is centered around a dynamic computation graph paradigm. In order to edit it we replace each parametric module (i.e. layers) with modified modules that perform the same operation but with the premultiplication by the binary mask.

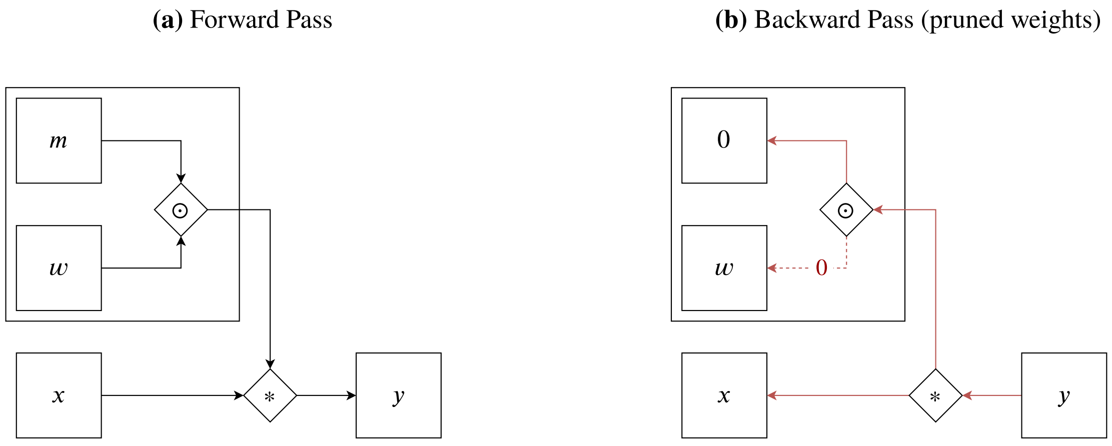

Figure 4.3: By replacing w with $w' = w \odot m$ in the computational graph we can achieve a persistent pruning that prevails after forward and backwards passes of the network.

4.4 Evaluation Metrics

As we saw in the literature metanalysis, standardization of metrics is also required so that results can be compared in a meaningful way. To this end, ShrinkBench provides a set of standardized metrics to measure the amount of pruning in a network as well as related metrics to approximate the potential performance gains to be achieved by that amount of pruning. The correctness and behavior of these metrics are critical in order to ensure that comparisons between methods.

ShrinkBench includes the following primitives:

- **Model Size** - Returns the total number of parameters in the network and the number of unpruned parameters (i.e. with nonzero value).
- **Compression** - Ratio between total model size and unpruned model size
- **FLOPs** - Number of floating point operations per second. We also report the amount of FLOPs involving unpruned parameters.
- **Speedup** - ratio between the original FLOPs and FLOPs involving unpruned parameters.
- **Memory Usage** - Amount of memory required to store the model along with its activations. As the previous metrics, it also returns the amount of memory taken by unpruned parameters. Importantly, this considers activations where all connected weights are pruned to be zero. Otherwise false positives might be reported because of activation functions outputting zero such as ReLUs.

As we will see in the results section while some of these metrics are vaguely correlated since model with more parameters lead to higher FLOPs and higher memory usage this is not a strict relationship. Namely, a larger model can have fewer FLOPs and memory footprint that a smaller network because the latter metrics also depend on the size of the input and output tensors. Thus we encourage authors to use all metrics where applicable to better understand the performance trade-offs of pruning methods.

A known limitatation of ShrinkBench is that it does not attempt to get actual hardware measurements apart from wallclock time, i.e. the amount of time taken for each step. Getting accurate hardware measurements for many of these metrics is a challenging task and it will vary depending on the choice of hardware backend: CPU, GPU, TPU, FPGA, &c. Moreover, some metrics such as latency or energy consumption are fundamentally hard to standardize since they will vary greatly depending on the platform. Thus, while we encourage authors to report hardware metrics is those when those are relevant to the bottom line of a pruning method, ShrinkBench assumes ideal hardware performance and reports all its metrics in a hardware agnostic way. As we will see in the results chapter, even with a hardware abstraction in place, standardization of all the above layers is required to prevent confounding factors leading us to incorrect conclusions.

4.5 Sample Implementations

In this section we show how ShrinkBench can be used to implement a couple simple magnitude pruning methods.

Magnitude pruning methods use the absolute value of the weights as a proxy for their importance and have proved to be good baseline that performs competitive in many tasks (Frankle et al., 2019a; Gale et al., 2019; Han et al., 2016; Han et al., 2015). For instance, to achieve a compression ratio of 4, we would compute the absolute value of the set of parameters we care about, sort them and keep the 25% with highest value, discarding the rest. More generally, to achieve a compression of x, one computes the $1/x$ quantile of the distribution of importances (here the absolute weights), and keeps the values above that quantile.

However, magnitude pruning just indicates that we are using the absolute value function as our scoring function, we still need to decide what groupings we are going to choose when computing the relative weight importances. In practice, two common choices are:

- **Globally** - All the weights from the network are treated equally, and ranked in a global fashion. This approach can lead to larger amounts of pruning on over-parametrized layers but it ignores the relative scale of weights within each layer which can affect the selectivity of the scoring function.
- **Layerwise** - Instead of concatenating all the weights of the network, layerwise approaches treat each layer of the neural network independently. This way, the scale of the weights in each layer is accounted for in the pruning step which can lead to better accuracy. Nevertheless, underparametrized layers can be quite impacted from this choice since the same amount of pruning will be enforced in every single layer.

In the Listing 4.1, we show how to implement both of these approaches using the API provided by ShrinkBench. In the implementation we can observe that the user is tasked with just computing the tasks from the set of parameters. Since the parameters of PyTorch models are contained in a nested hierarchy of modules, ShrinkBench offers convenient accessors for those with `self.params()` and `self.module_params()`.

Moreover, even though our scoring function is quite trivial to specify (in our example is just NumPy's absolute value `np.abs`), we still need to broadcast its application to all the parameters to obtain the required weight masks. To this end, ShrinkBench provides auxiliary functions such as:

```python
import numpy as np
from shrinkbench.pruning import VisionPruning, LayerPruning
from shrinkbench.utils import (map_importances,
                               flatten_importances,
                               fraction_threshold,
                               fraction_mask,
                               importance_masks)

class GlobalMagWeight(VisionPruning):

    def model_masks(self):
        importances = map_importances(np.abs, self.params())
        flat_importances = flatten_importances(importances)
        threshold = fraction_threshold(flat_importances, self.fraction)
        masks = importance_masks(importances, threshold)
        return masks


class LayerMagWeight(LayerPruning, VisionPruning):

    def layer_masks(self, module):
        params = self.module_params(module)
        importances = {param: np.abs(value) for param, value in params.items()}
        masks = {param: fraction_mask(importances[param], self.fraction)
                 for param, value in params.items() if value is not None}
        return masks
```

Listing 4.1: ShrinkBench's implementation of Global and Layerwise Weight Magnitude pruning. In this examples, we can observe how both pruning strategies are specified concisely and in a fairly descriptive fashion. From the model parameters the user just needs to return the associated binary masks. To simplify the tasks, there are helper functions to map arbitrary scoring functions in a global or layerwise manner.

- `map_importances` - Applies a scoring function in a elementwise manner to each one of the weights in the provided argument
- `flatten_importances` - Flattens a potentially nested hierarchy of importances to a flat array for easier handling.
- `fraction_threshold` - Computes the threshold so that the specified fraction of the weights are larger than said threshold. Very similar to a quantile function.
- `importance_masks` - Computes binary masks from a set of importances and a associated threshold value to use as a cut-off.
- `fraction_mask` - Compound function of `importance_masks` and `fraction_threshold`

By using these util functions, the task of specifying a pruning method is further simplified while still keeping. Moreover, researchers are free to implement their own versions of these functions and modify them as they see fit. These are provided for convenience and to show how decomposition of the steps of developing a pruning algorithm can lead to simple implementation and code reuse.

4.6 Recommendations for Evaluating a Pruning Method

In this section we provide a set of general recommendations and guidelines for evaluation pruning methods. Whilst some are enforced in ShrinkBench's design, it might be in researchers' best interest to write their own implementation since ShrinkBench cannot account for all possible scenarios.

For any pruning technique proposed, check if:

- It is contextualized with respect to magnitude pruning, recently-published pruning techniques, and pruning techniques proposed prior to the 2010s.
- The pruning algorithm, constituent subroutines (e.g., score, pruning, and fine-tuning functions), and hyperparameters are presented in enough detail for a reader to reimplement and match the results in the paper.
- All claims about the technique are appropriately restricted to only the experiments presented (e.g., CIFAR-10, ResNets, image classification tasks, etc.).
- There is a link to downloadable source code.

For all experiments, check if you include:

- A detailed description of the architecture with hyperparameters in enough detail to for a reader to reimplement it and train it to the same performance reported in the paper.
- If the architecture is not novel: a citation for the architecture/hyperparameters and a description of any differences in architecture, hyperparameters, or performance in this paper.
- A detailed description of the dataset hyperparameters (e.g., batch size and augmentation regime) in enough detail for a reader to reimplement it.
- A description of the library and hardware used.

For all results, check if:

- Data is presented across a range of compression ratios, including extreme compression ratios at which the accuracy of the pruned network declines substantially.
- Data specifies the raw accuracy of the network at each point.
- Data includes multiple runs with separate initializations and random seeds.
- Data includes clearly defined error bars and a measure of central tendency (e.g., mean) and variation (e.g., standard deviation).
- Data includes FLOP-counts if the paper makes arguments about efficiency and performance due to pruning.

For all pruning results presented, check if there is a comparison to:

- A random pruning baseline.
  - A global random pruning baseline.
  - A random pruning baseline with the same layerwise pruning proportions as the proposed technique.
- A magnitude pruning baseline.
  - A global or uniform layerwise proportion magnitude pruning baseline.
  - A magnitude pruning baseline with the same layerwise pruning proportions as the proposed technique.
- Other relevant state-of-the-art techniques, including:
  - A description of how the comparisons were produced (data taken from paper, reimplementation, or reuse of code from the paper) and any differences or uncertainties between this setting and the setting used in the main experiments.

Chapter 5

Experimental Setup

In this chapter we detail the experimental setup used to carry out the results presented in the results chapter along with the motivation for these experiments. Here we describe the pruning baselines we implemented into ShrinkBench and the set of experiments we used them in along with detailed accounts of hyperparameter and pipelines choices to ensure the reproducibility of the experiments.

5.1 Motivation

So far we have analyzed the current state of the neural network pruning literature and described ShrinkBench, our proposed framework that attempts to remediate most of the identified barriers to comparison. However, it remains unclear whether the need for standardization satisfies merely a data aggregation and analysis issue or whether there are fundamental issues when running pruning experiments without using a common ground. Namely, we need to identify whether the potential confounding factors do actually become sources of confusion if they are not accounted for. Even though proving that they are not confounding factors is a fundamentally hard problem since it needs to be shown for every possible experiment, showing that they are reduces to finding a counterexample.

To test this, we implement a few simple yet common pruning baselines and use them to prune popular models on a set of vision classification tasks. While vision classification tasks are not a fully representative sample of all the deep learning models being used, as we saw the existing pruning literature focused on these tasks.

5.2 Baseline Pruning Methods

In order to test our hypothesis we need to implement a series of pruning methods so that we can compare them and analyze the effect of the potential confounding factors of the pipeline. We denominate them as baselines since they represent unsophisticated yet performant approaches to neural network pruning and thus will showcase the behavior that we expect from an actual pruning method. As we delineated in the recommendations section in the previous chapter we encourage authors to include at least a random and magnitude pruning baselines in their experiments to better understand the relative performance of the proposed methods.

Magnitude-based approaches are common baselines in the literature and have been shown to be competitive with more complex methods (Frankle et al., 2019a; Gale et al., 2019; Han et al., 2016; Han et al., 2015). Gradient-based methods are less common, but are simple to implement and have recently gained popularity (Lee et al., 2019; Lee et al., 2018; Yu et al., 2018). Note that these baselines are not reproductions of any of these methods, but merely inspired by their pruning heuristics. Namely, when implementing these we aimed at simplicity while maintaining the choice of heuristics for the scoring function.

We used ShrinkBench to implement several existing pruning heuristics, both as examples of how to use our library and as baselines that new methods can compare to. An important consideration is that when designing this baseline we made a couple of choices specific to vision classification networks:

- Pruned layers are just dense layers (i.e. `nn.Linear`) and convolutional layers (`nn.Conv2d`). All considered networks in the torchvision package (Marcel & Rodriguez, 2010), only contain parameters in linear, convolutional and batch normalization layers. Moreover, pruning batch normalization layers is not recommended since it will effectively prune entire neurons instead of weights which can lead to many pathways in the network to be lost. Since these parameters are a minority, even if left unpruned, they will not have a very large impact unless we reach very high compression ratios.

- The very last dense layer is not pruned in any model. This is a common rule applied in most pruning methods (Frankle et al., 2019a; Gale et al., 2019) motivated by the fact that overly pruning the last layer can have a very significant impact in the performance of the network. Namely, pruning the layer leading to the softmax operator excessively can make some of the output neurons disconnected or weakly connected leading to some classes not being outputted during inference.

- When aiming for a compression level, all parameters in a network are counted, including parameters not considered for the pruning task (in our experiments these are batch normalization layer parameters).

5.2.1 Random Pruning

This strategy prunes each weight independently with probability equal to the fraction of the network to be pruned. Random pruning is just a straw man approach that is useful for debugging purposes since any approach should be better than randomly pruning weights.

5.2.2 Global Magnitude Pruning

This strategy prunes the weights with the lowest absolute value anywhere in the model. Parameters are ranked using their absolute value. Let W be the set of all parameters in the network, then we compute the quantile over the absolute values and threshold smaller values than the quantile.

To keep a proportion $p \in \{0, 1\}$ of the parameters. Compute the threshold $T(p)$ as the $(1-p)^{\text{th}}$

```python
def random_mask(tensor, fraction):
    idx = np.random.uniform(0, 1, size=tensor.shape) > fraction
    mask = np.ones_like(tensor)
    mask[idx] = 0.0
    return mask

class RandomPruning(VisionPruning):

    def model_masks(self):
        params = self.params()
        masks = map_importances(lambda x: random_mask(x, self.fraction), params)
        return masks
```

Listing 5.1: Implementation for Random Pruning. ShrinkBench's design allows for this approach to be implemented in a succinct manner.

quantile of the set $I_W = \{|w|, w \in W\}$ of importances and then set the masks as follows.

$$m(w) = \begin{cases} 1 & w > T(p) \\ 0 & w \leq T(p) \end{cases} \tag{5.1}$$

ShrinkBench's implementation can be found in Listing 4.1.

5.2.3 Layerwise Magnitude Pruning

For each layer prunes the weights with the lowest absolute value. The implementation can be found in Listing 4.1. The same pruning fraction is applied to each layer independently of how many parameters it has.

5.2.4 Global Gradient Magnitude Pruning

Prunes the weights with the lowest absolute value of (weight $\times$ gradient), evaluated on a batch of inputs. Let $S = \{x_i, y_i : i = 1 \ldots n\}$ be a set of input, output pairs to the network. We compute a Montecarlo approximation of the gradients with respect to the weights using these inputs:

$$\tilde{\nabla}_w = \frac{1}{n} \sum_{i=1}^{n} \frac{\partial L(f(x_i; W), y_i)}{\partial w} \tag{5.2}$$

Then the set of importances I_W is defined as

$$I_W = \left\{ |w \cdot \tilde{\nabla}_w| : w \in W \right\} \tag{5.3}$$

And then the masks are computed analogously to the Global Magnitude Pruning method.

```python
class GlobalMagGrad(GradientMixin, VisionPruning):

    def model_masks(self):
        params = self.params()
        grads = self.param_gradients()
        importances = {mod:
                       {p: np.abs(params[mod][p]*grads[mod][p])
                        for p in mod_params}
                       for mod, mod_params in params.items()}
        flat_importances = flatten_importances(importances)
        threshold = fraction_threshold(flat_importances, self.fraction)
        masks = importance_masks(importances, threshold)
        return masks


class LayerMagGrad(GradientMixin, LayerPruning, VisionPruning):

    def layer_masks(self, module):
        params = self.module_params(module)
        grads = self.module_param_gradients(module)
        importances = {param: np.abs(value*grads[param]) for param, value in
        ↪   params.items()}
        masks = {param: fraction_mask(importances[param], self.fraction)
                 for param, value in params.items() if value is not None}
        return masks
```

Listing 5.2: Implementations for *Global Gradient Magnitude Pruning* and *Layerwise Gradient Magnitude Pruning*. In these implementations we can see how the user can access sample parameter gradients to use them in the computation of the weight masks.

ShrinkBench's implementation of Global Gradient Magnitude Pruning is shown in Listing 5.2.

5.2.5 Layerwise Magnitude Pruning

For each layer, prunes the weights the lowest absolute value of (weight $\times$ gradient), evaluated on a batch of inputs. Uses analogous importances to the ones described for Global Gradient Magnitude Pruning but only considers one layer at a time. The implementation can be found in Listing 5.2. The same pruning fraction is applied to each layer independently of how many parameters it has.

5.3 Techniques

To perform the described analysis several tools and techniques were be employed. The analysis can be broken down into two parts: performing the experiments that prune neural networks using different heuristics and then analyzing processing the results.

To implement the neural network pruning techniques the PyTorch framework (Paszke et al.,

2019) were used, one of the most popular open source neural network libraries. The PyTorch library is throughly documented and provided a solid foundation upon which to build the framework for the described experiments. First, it provided a flexible interface that allows us to edit the architecture and behavior of neural networks. Moreover, by using PyTorch we were also able to leverage torchvision (Marcel & Rodriguez, 2010), a package written in PyTorch that implements popular datasets, transformations and model architectures for computer vision tasks.

To perform the analysis shown in the Results chapter, we employed a wide variety of Python packages for scientific computing, including:P

- NumPy (Oliphant, 2006; Van Der Walt et al., 2011) - NumPy is a library that adds support in Python for large, multi-dimensional arrays and matrices, along with a large collection of high-level mathematical functions to operate on these arrays. NumPy was employed in other to implement the pruning heuristics that output binary masks that encode what neural network weights to keep as part of the model.
- Pandas (McKinney, 2010; pandas development team, 2020) - Data analysis and manipulation tool. It was employed to aggregate the results from the different experiments and post process them in efficient and reproducible manner.
- Matplotlib (Hunter, 2007) - As a last step in our pipeline, we are interested in producing plots that report the drop in accuracy of the studied model as we increase the amount of pruning. Therefore, we employ matplotlib, the most popular plotting library for Python.

All the listed software is licensed as Free and Open Source Software (FOSS) so it is readily available. Since a core goal of this project is to produce a framework that enables researchers to perform standardized evaluation of neural network pruning methods, it is crucial that we can distribute this framework in FOSS way as well.

Chapter 6

Results

This chapter details the experimental results that we obtained when using ShrinkBench to prune over 800 networks with varying datasets, networks, compression ratios, initial weights and random initializations. During this process we identified various pitfalls linked with common experimental practices in the literature that can be prevented by using ShrinkBench.

We highlight several noteworthy results below and comment their implications. Additional experimental results are included in Appendix A for completeness.

6.1 Experiment Details

We swept a series of geometrically spaced values of compression for all the experiments. To simplify the analysis we chose increasing powers of two, including the value 1 with the original unpruned network to act as a reference for original performance of the network. We computed the pruned fraction so that the actual compression ratio would be as close as possible to the desired ratio and made sure that parameters not considered for pruning were used in the computation of the compression values. For all networks, compression ratios and FLOP counts were stored at the beginning and end of the finetuning step.

We evaluated the following network-dataset combinations on all the described baseline pruning methods.

- CIFAR10 dataset with the following networks (Zagoruyko, 2015):
  - CIFAR-VGG
  - ResNet-20
  - ResNet-56
  - ResNet-110
- ImageNet dataset with ResNet-18 (Deng et al., 2009; K. He et al., 2016)

After applying the corresponding pruning technique, networks were finetuned until convergence. Pruning was performed from the pretrained weights and fixed from there forwards.

Unless otherwise noted, the reported values are always for the held out validation set and no hyperparameter was performed. Early stopping is implemented during finetuning. Thus if the validation accuracy repeatedly decreases after some point we stop the finetuning process to prevent overfitting. The hyperparameter choices for the experiments are displayed in Table 6.1.

Hyperparameter	**CIFAR10**	**ImageNet**
Batch size	64	256
Epochs	30	20
Optimizer	Adam	SGD
Nesterov Momentum	-	0.9
Initial Learning Rate	3×10^{-4}	10^{-3}
Learning rate schedule	Fixed	Fixed

Table 6.1: Hyperparmeter details for the pruning experiments

For reproducibility purposes, ShrinkBench fixes random seeds for all the dependencies (PyTorch, NumPy, Python). Three independent runs using different random seeds were performed for every CIFAR10 experiment. This was not done for the ImageNet networks because of how time consuming and computationally expensive training ImageNet networks is. For the CIFAR10 experiments, we found some variance across methods that relied on randomness, such as random pruning or gradient based methods that use a sampled minibatch to compute the gradients with respect to the weights.

6.2 Metrics are not Interchangeable

As we discussed previously, it is a common practice to report either reduction in the number of parameters or in the number of FLOPs. If these metrics are extremely correlated, reporting only one is sufficient to characterize the efficacy of a pruning method. We found after computing these metrics for the same model under many different settings that reporting one metric is not sufficient.

While these metrics are correlated, the correlation is different for each pruning method. Thus, the relative performance of different methods can vary significantly under different metrics. Figure 6.1 showcases and example of this scenario by displaying the same 4 pruned ResNet-18 networks on ImageNet using the previously described baselines. The four pruned methods are plotted both in terms of compression ratio and theoretical speedup.

As Figure 6.1 shows, Global pruning methods are more accurate than Layerwise methods for a given model size, but Layerwise methods are more accurate for a given theoretical speedup. This discrepancy is related to the fact that pruning layers with larger inputs can result in higher computational savings. This might appear counterintuive since in the presented results, networks with more parameters (i.e. lower compression) perform fewer operations (i.e. higher speedup). However, this can be easily explained by the fact that in convolutional neural networks the amount of operations in a convolutional layer depends linearly both on the size of the convolutional filter and the size of the input. Thus, since layerwise methods prune uniformly all layers to achieve the same sparsity, this leads to higher reductions of parameters in layers with large inputs (early layers in the network).

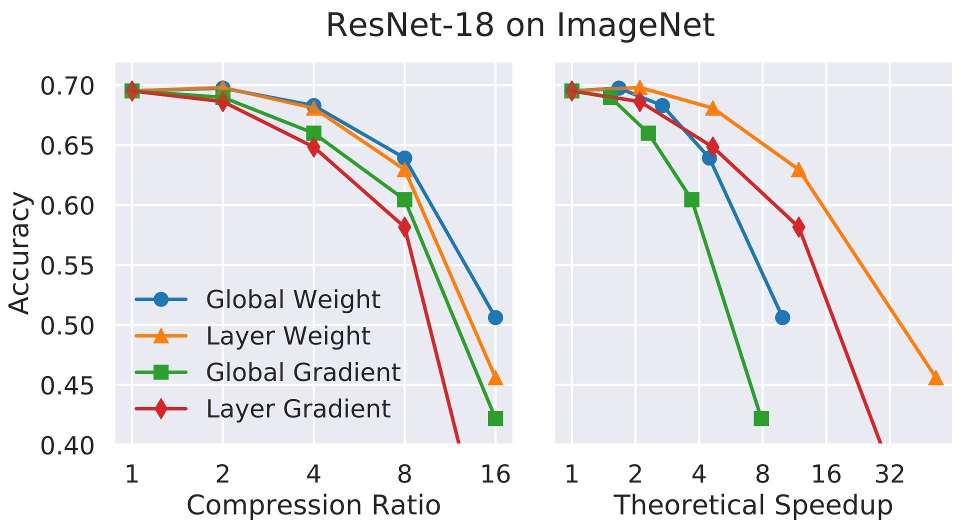

Figure 6.1: Top 1 Accuracy for ResNet-18 on ImageNet for several compression ratios and their corresponding theoretical speedups. Global methods give higher accuracy than Layerwise ones for a fixed model size, but the reverse is true for a fixed theoretical speedup.

6.3 Results Vary Across Models, Datasets, and Compression

Many methods report results on only a small number of datasets, models, amounts of pruning, and random seeds. If the relative performance of different methods tends to be constant across all of these variables, this may not be problematic. However, our results suggest that this performance is not constant. Figure 6.2 shows the accuracy for various compression ratios for CIFAR-VGG (Zagoruyko, 2015) and ResNet-56 on CIFAR-10. In general, Global methods are more accurate than Layerwise methods and Magnitude-based methods are more accurate than Gradient-based methods, with random performing worst of all.

However, if one were to look only at CIFAR-VGG for compression ratios smaller than 10, one could conclude that Global Gradient outperforms all other methods. Similarly, while Global Gradient consistently outperforms Layerwise Magnitude on CIFAR-VGG, the opposite holds on ResNet-56 (i.e., the orange and green lines switch places). This is the opposite of the conclusion one would reach using ResNet-56, where that method is always outperformed by Global and Layerwise Magnitude Pruning. This shows the need to evaluate on a pruning method on various model and dataset combinations to ensure it is not architecture dependent. Similarly, this shows the need to report metrics for several compression rations to characterize the trade-off curve as pruning increases.

Moreover, we found that for some settings close to the drop-off point (such as Global Gradient, compression 16), different random seeds yielded significantly different results (0.88 vs 0.61 accuracy) due to the randomness in minibatch selection. This is illustrated by the large vertical error bar in the left subplot.

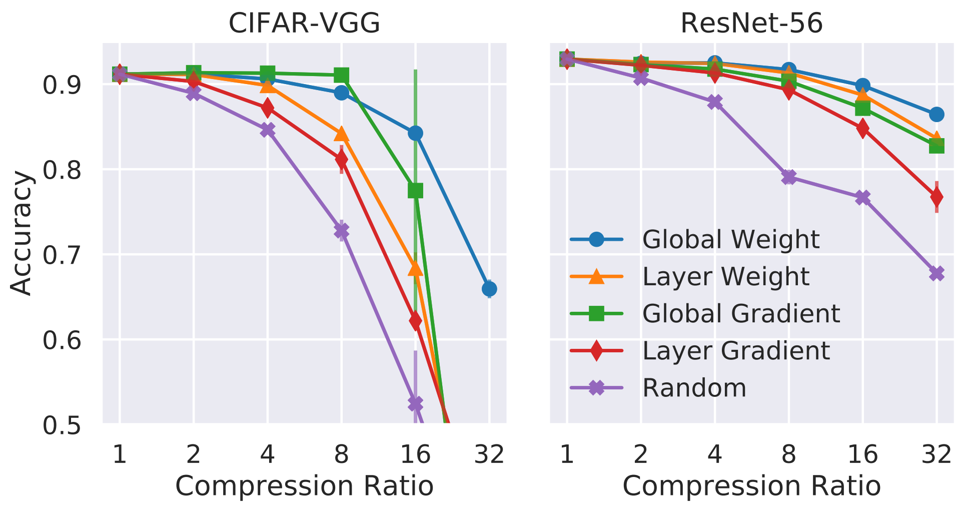

Figure 6.2: Top 1 Accuracy on CIFAR-10 for several compression ratios. Global Gradient performs better than Global Magnitude for CIFAR-VGG on low compression ratios, but worse otherwise. Global Gradient is consistently better than Layerwise Magnitude on CIFAR-VGG, but consistently worse on ResNet-56. One standard deviation bars across three runs are included for all compression ratios greater than one.

6.4 Using the Same Initial Model is Essential

As mentioned previously, many methods are evaluated using different initial models with the same architecture. To assess whether beginning with a different model can skew the results, we created two different models and evaluated Global vs Layerwise Magnitude pruning on each with all other variables held constant.

To obtain the models, we trained two ResNet-56 networks using Adam until convergence with $\eta = 10^{-3}$ and $\eta = 10^{-4}$. We'll refer to these pretrained weights as Weights A and Weights B, respectively. As shown on the left side of Figure 6.3, the different methods appear better on different models. With Weights A, the methods yield similar absolute accuracies. With Weights B, however, the Global method is more accurate at higher compression ratios.

We also found that the common practice of examining changes in accuracy is insufficient to correct for initial model as a confounder. Even when reporting changes, one pruning method can artificially appear better than another by virtue of beginning with a different model. We see this on the right side of Figure 6.3, where Layerwise Magnitude with Weights B appears to outperform Global Magnitude with Weights A, even though the former never outperforms the latter when initial model is held constant.

All these results evidence the need for standardized experiments when evaluating neural network pruning methods. With ShrinkBench we aim to provide a toolkit that facilitates the task of producing comprehensive and reproducible evaluations of proposed pruning approaches.

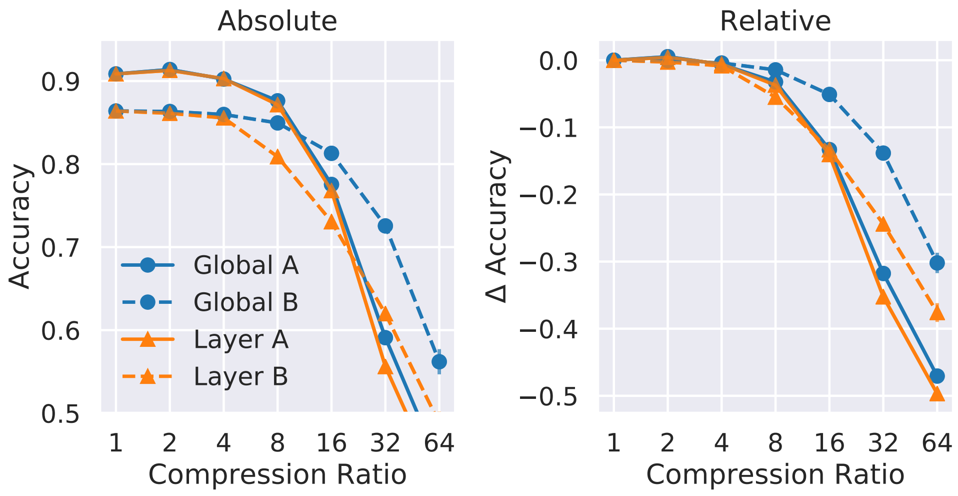

Figure 6.3: Global and Layerwise Magnitude Pruning on two different ResNet-56 models. Even with all other variables held constant, different initial models yield different tradeoff curves. This may cause one method to erroneously appear better than another. Controlling for initial accuracy does not fix this.

Chapter 7

Conclusions and Future Work

We wrap up by providing future avenues of work in this area and provide some concluding remarks that highlight the contributions and developments that this work encompasses.

7.1 Future Work

There are multiple avenues for extending the work presented here. In its current state ShrinkBench cannot evaluate any type of neural network experiment. We have identified the following limitations:

- Some structured pruning methods implicitly have higher amount of effective pruning than what ShrinkBench would report. This happens when there are compounding effects between the pruning between layers. For instance, if one prunes all the weights in the same channel in a convolutional layer then the output will always be zero for that entire channel. Thus any nonzero weights in any consecutive layer are effectively pruned as well since they will always take zeros as input.

  However, this compounding effect is difficult to account for since it requires knowledge of the computational graph of the network a priori. Because of PyTorch's uses a dynamic computational graph design paradigm, it is very hard to efficiently and correctly perform an automated analysis of an arbitrary computational graph. A potential approach would be to use to compile the model to a static graph representation like ONNX and use that information to get the effective sparsity. Another potential way of dealing with this would be to design a set of inputs so that the gradient information from those inputs would help uncover these dependencies.

- In its current state, ShrinkBench assumes that pruning is an operation that is performed once training has completed, and does not provide support for incorporating that operation while the network trains. Evaluating approaches that prune during training is a fundamentally harder task since it requires standardizing the training procedure itself since there is no pretrained state to start from. However, the recent interest of these type of approaches (Frankle et al., 2019a, 2019b) proves that there is interest in this type of functionality.

- As we indicated during ShrinkBench's description, it only reports hardware agnostic metrics to facilitate the comparison between methods. However, in many scenarios the main motivation behind pruning is improving some hardware dependent metric for a network such a energy consumption, latency or throughput. Future work could analyze ways to standardize this task drawing insights from existing approaches like the MLPerf benchmark.

Moreover, ShrinkBench could also be easily extended to run other type of experiments:

- Given the rise of deep learning models for NLP applications, ShrinkBench could be extended so that it Modern NLP models are implemented using Recurrent Neural Networks (RNN), oftentimes in the form of Long Short Term Memory Networks (LSTM) along with Transformers architectures that rely on attention based building blocks. Despite some of the required layers not being explicitly implemented in ShrinkBench, they could be added by employing the same approach used for the dense and convolutional layers. More generally, any parametric layer can be replaced by a layer that performs a elementwise multiplication by a prespecified mask. Using this design, most of the current design of ShrinkBench can be reused for other tasks.
- ShrinkBench can be easily extended to support other datasets. Since it uses the standard PyTorch Dataset and DataLoader interface, relevant datasets can be incorporated.
- Similarly, ShrinkBench can be extended to other network architectures and models. Whereas architectures are arguably as easy to add as datasets are, the issue would be to ensure pre-trained models have been trained using a correct hyperparameter choice. For ImageNet models ShrinkBench uses torchvision's model architectures and publicly available pre-trained models. For CIFAR10, it provides its own models under https://github.com/JJGO/shrinkbench-models. Moreover, any provided models should be representative of the types of models employed for the task of interest, otherwise it wouldn't be beneficial to include in the framework.

Lastly, neural network pruning is just one of the many sub-fields that have seen an explosion of interest over the past few years given the rise of deep learning in general. However, the same evaluation issues that we have identified and done our best to remedy are present to some extent in other machine learning domains. For instance, the gradient-compression literature seems to suffer from some of the same systematic problem that we listed in the pruning literature. Gradient compression is a task common is distributed training scenarios, where gradients are compressed (sometimes in a lossy fashion) to minimize the network requirements for communicating the gradient updates. Given the similarity between gradient compression and pruning, developing a similar standardized evaluation framework with similar principles would of benefit to the literature and community.

7.2 Conclusion

Given the enormous interest in neural network pruning over the past decade, it seems natural to ask simple questions about the relative efficacy of different pruning techniques. Although a few basic findings are shared across the literature, missing baselines and inconsistent experimental settings make it impossible to assess the state of the art or confidently compare the dozens of techniques proposed in recent years. After carefully studying the literature and enumerating numerous areas of incomparability and confusion, we suggest concrete remedies in the form of a list of best practices and an open-source library —ShrinkBench —to help future research endeavors to produce the kinds of results that will harmonize the literature and make our motivating questions easier to answer. Furthermore, ShrinkBench results on various pruning techniques evidence the need for standardized experiments when evaluating neural network pruning methods.

Bibliography

Crall, J. (2018). Accuracy of resnet50 is much higher than reported! [Accessed: 2019-07-22]. (Cit. on p. 13).

Dai, B., Zhu, C., & Wipf, D. (2018). Compressing neural networks using the variational information bottleneck. *arXiv preprint arXiv:1802.10399* (cit. on pp. 1, 9).

Deng, J., Dong, W., Socher, R., Li, L.-J., Li, K., & Fei-Fei, L. (2009). Imagenet: A large-scale hierarchical image database, In *2009 ieee conference on computer vision and pattern recognition*. Ieee. (Cit. on p. 31).

Figurnov, M., Ibraimova, A., Vetrov, D. P., & Kohli, P. (2016). Perforatedcnns: Acceleration through elimination of redundant convolutions, In *Advances in neural information processing systems*. (Cit. on p. 7).

Frankle, J., & Carbin, M. (2019). The lottery ticket hypothesis: Finding sparse, trainable neural networks, In *7th international conference on learning representations, ICLR 2019, new orleans, la, usa, may 6-9, 2019*, OpenReview.net. https://openreview.net/forum?id=rJl-b3RcF7. (Cit. on pp. 6, 9, 47)

Frankle, J., Dziugaite, G. K., Roy, D. M., & Carbin, M. (2019a). The lottery ticket hypothesis at scale. *arXiv preprint arXiv:1903.01611* (cit. on pp. 6, 9, 20, 26, 37).

Frankle, J., Dziugaite, G. K., Roy, D. M., & Carbin, M. (2019b). Stabilizing the Lottery Ticket Hypothesis [arXiv: 1903.01611]. *arXiv:1903.01611 [cs, stat]*. Retrieved July 19, 2019, from http://arxiv.org/abs/1903.01611 (cit. on p. 37)

Gale, T., Elsen, E., & Hooker, S. (2019). The state of sparsity in deep neural networks. (Cit. on pp. 5, 6, 9, 13, 20, 26).

Gray, S., Radford, A., & Kingma, D. P. (2017). Gpu kernels for block-sparse weights. *arXiv preprint arXiv:1711.09224* (cit. on p. 10).

Han, S., Mao, H., & Dally, W. J. (2016). Deep compression: Compressing deep neural network with pruning, trained quantization and huffman coding (Y. Bengio & Y. LeCun, Eds.). In Y. Bengio & Y. LeCun (Eds.), *4th international conference on learning representations, ICLR 2016, san juan, puerto rico, may 2-4, 2016, conference track proceedings*. http://arxiv.org/abs/1510.00149. (Cit. on pp. 20, 26, 47)

Han, S., Pool, J., Tran, J., & Dally, W. (2015). Learning both weights and connections for efficient neural network, In *Advances in neural information processing systems*. (Cit. on pp. 1, 5–7, 9, 20, 26).

He, K., Zhang, X., Ren, S., & Sun, J. (2016). Deep residual learning for image recognition, In *Proceedings of the ieee conference on computer vision and pattern recognition*. (Cit. on p. 31).

He, Y., Lin, J., Liu, Z., Wang, H., Li, L.-J., & Han, S. (2018). Amc: Automl for model compression and acceleration on mobile devices, In *Proceedings of the european conference on computer vision (eccv)*. (Cit. on p. 7).

He, Y., Zhang, X., & Sun, J. (2017). Channel pruning for accelerating very deep neural networks, In *Proceedings of the ieee international conference on computer vision*. (Cit. on pp. 6, 9).

Huang, Y., Cheng, Y., Chen, D., Lee, H., Ngiam, J., Le, Q. V., & Chen, Z. (2018). Gpipe: Efficient training of giant neural networks using pipeline parallelism. *arXiv preprint arXiv:1811.06965* (cit. on p. 1).

Hunter, J. D. (2007). Matplotlib: A 2d graphics environment. *Computing in Science & Engineering*, *9*(3), 90–95 (cit. on p. 29).

Janowsky, S. A. (1989). Pruning versus clipping in neural networks. *Physical Review A*, *39*(12), 6600–6603 (cit. on p. 1).

Jogeshwar, A. (2017). Validating resnet50 [Accessed: 2019-07-22]. (Cit. on p. 13).

Kalchbrenner, N., Elsen, E., Simonyan, K., Noury, S., Casagrande, N., Lockhart, E., Stimberg, F., Oord, A. v. d., Dieleman, S., & Kavukcuoglu, K. (2018). Efficient neural audio synthesis. *arXiv preprint arXiv:1802.08435* (cit. on p. 10).

Karnin, E. D. (1990). A simple procedure for pruning back-propagation trained neural networks. *IEEE transactions on neural networks*, *1*(2), 239–242 (cit. on p. 1).

Keras exported model shows very low accuracy in tensorflow serving [Accessed: 2019-07-22]. (2017). (Cit. on p. 13).

Kim, Y.-D., Park, E., Yoo, S., Choi, T., Yang, L., & Shin, D. (2015). Compression of deep convolutional neural networks for fast and low power mobile applications. *arXiv preprint arXiv:1511.06530* (cit. on p. 7).

Lebedev, V., Ganin, Y., Rakhuba, M., Oseledets, I., & Lempitsky, V. (2014). Speeding-up convolutional neural networks using fine-tuned cp-decomposition. *arXiv preprint arXiv:1412.6553* (cit. on p. 7).

Lee, N., Ajanthan, T., Gould, S., & Torr, P. H. S. (2019). A Signal Propagation Perspective for Pruning Neural Networks at Initialization [arXiv: 1906.06307]. *arXiv:1906.06307 [cs, stat]*. Retrieved July 19, 2019, from http://arxiv.org/abs/1906.06307 (cit. on pp. 10, 26)

Lee, N., Ajanthan, T., & Torr, P. H. (2018). Snip: Single-shot network pruning based on connection sensitivity. *arXiv preprint arXiv:1810.02340* (cit. on pp. 5, 6, 9, 26).

Li, H., Kadav, A., Durdanovic, I., Samet, H., & Graf, H. P. (2016). Pruning filters for efficient convnets. *arXiv preprint arXiv:1608.08710* (cit. on pp. 6, 9).

Liu, Z., Sun, M., Zhou, T., Huang, G., & Darrell, T. (2019). Rethinking the value of network pruning, In *7th international conference on learning representations, ICLR 2019, new orleans, la, usa, may 6-9, 2019*, OpenReview.net. https://openreview.net/forum?id=rJlnB3C5Ym. (Cit. on pp. 1, 6, 9, 13)

Louizos, C., Ullrich, K., & Welling, M. (2017). Bayesian compression for deep learning, In *Advances in neural information processing systems*. (Cit. on pp. 7, 9).

Luo, J.-H., Wu, J., & Lin, W. (2017). Thinet: A filter level pruning method for deep neural network compression, In *Proceedings of the ieee international conference on computer vision*. (Cit. on pp. 7, 9).

Marcel, S., & Rodriguez, Y. (2010). Torchvision the machine-vision package of torch, In *Proceedings of the 18th acm international conference on multimedia*. (Cit. on pp. 26, 29).

Mariet, Z., & Sra, S. (2015). Diversity networks: Neural network compression using determinantal point processes. *arXiv preprint arXiv:1511.05077* (cit. on p. 9).

McKinney, W. (2010). Data Structures for Statistical Computing in Python (S. van der Walt & J. Millman, Eds.). In S. van der Walt & J. Millman (Eds.), *Proceedings of the 9th Python in Science Conference*. (Cit. on p. 29).

Molchanov, D., Ashukha, A., & Vetrov, D. (2017). Variational dropout sparsifies deep neural networks, In *Proceedings of the 34th international conference on machine learning-volume 70*. JMLR. org. (Cit. on p. 5).

Molchanov, P., Tyree, S., Karras, T., Aila, T., & Kautz, J. (2016). Pruning convolutional neural networks for resource efficient inference. *arXiv preprint arXiv:1611.06440* (cit. on p. 9).

Morcos, A. S., Yu, H., Paganini, M., & Tian, Y. (2019). One ticket to win them all: Generalizing lottery ticket initializations across datasets and optimizers [arXiv: 1906.02773]. *arXiv:1906.02773 [cs, stat]*. Retrieved July 19, 2019, from http://arxiv.org/abs/1906.02773 (cit. on p. 9)

Mozer, M. C., & Smolensky, P. (1989a). Skeletonization: A technique for trimming the fat from a network via relevance assessment, In *Advances in neural information processing systems*. (Cit. on p. 1).

Mozer, M. C., & Smolensky, P. (1989b). Using Relevance to Reduce Network Size Automatically. *Connection Science*, *1*(1), 3–16 (cit. on p. 1).

Nola, D. (2016). Keras doesn't reproduce caffe example code accuracy [Accessed: 2019-07-22]. (Cit. on p. 13).

Northcutt, C. (2019). Towards reproducibility: Benchmarking keras and pytorch [Accessed: 2019-07-22]. (Cit. on p. 13).

Nvidia tesla p100 datasheet [Accessed: 2020-07-15]. (n.d.). (Cit. on p. 47).

Oliphant, T. E. (2006). *A guide to numpy* (Vol. 1). Trelgol Publishing USA. (Cit. on p. 29).

pandas development team, T. (2020). *Pandas-dev/pandas: Pandas* (Version latest). Zenodo. (Cit. on p. 29).

Paszke, A., Gross, S., Chintala, S., Chanan, G., Yang, E., DeVito, Z., Lin, Z., Desmaison, A., Antiga, L., & Lerer, A. (2017). Automatic differentiation in pytorch (cit. on p. 15).

Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., Killeen, T., Lin, Z., Gimelshein, N., Antiga, L., Desmaison, A., Kopf, A., Yang, E., DeVito, Z., Raison, M., Tejani, A., Chilamkurthy, S., Steiner, B., Fang, L., … Chintala, S. (2019). Pytorch: An imperative style, high-performance deep learning library. In H. Wallach, H. Larochelle, A.

Beygelzimer, F. d'Alché-Buc, E. Fox, & R. Garnett (Eds.), *Advances in neural information processing systems 32* (pp. 8024–8035). Curran Associates, Inc. http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf. (Cit. on p. 28)

Russakovsky, O., Deng, J., Su, H., Krause, J., Satheesh, S., Ma, S., Huang, Z., Karpathy, A., Khosla, A., Bernstein, M., et al. (2015). Imagenet large scale visual recognition challenge. *International journal of computer vision*, *115*(3), 211–252 (cit. on p. 2).

Suau, X., Zappella, L., & Apostoloff, N. (2018). Network compression using correlation analysis of layer responses (cit. on p. 9).

Suzuki, T., Abe, H., Murata, T., Horiuchi, S., Ito, K., Wachi, T., Hirai, S., Yukishima, M., & Nishimura, T. (2018). Spectral-pruning: Compressing deep neural network via spectral analysis. *arXiv preprint arXiv:1808.08558* (cit. on p. 9).

Sze, V., Chen, Y.-H., Yang, T.-J., & Emer, J. (2017). Efficient processing of deep neural networks: A tutorial and survey. *arXiv preprint arXiv:1703.09039* (cit. on p. 1).

Van Der Walt, S., Colbert, S. C., & Varoquaux, G. (2011). The numpy array: A structure for efficient numerical computation. *Computing in Science & Engineering*, *13*(2), 22 (cit. on p. 29).

Vryniotis, V. (2018). Change bn layer to use moving mean/var if frozen [Accessed: 2019-07-22]. (Cit. on p. 13).

Wen, W., Wu, C., Wang, Y., Chen, Y., & Li, H. (2016). Learning structured sparsity in deep neural networks, In *Advances in neural information processing systems*. (Cit. on p. 7).

Yang, T.-J., Chen, Y.-H., & Sze, V. (2017). Designing energy-efficient convolutional neural networks using energy-aware pruning, In *Proceedings of the ieee conference on computer vision and pattern recognition*. (Cit. on pp. 1, 7).

Yu, R., Li, A., Chen, C.-F., Lai, J.-H., Morariu, V. I., Han, X., Gao, M., Lin, C.-Y., & Davis, L. S. (2018). Nisp: Pruning networks using neuron importance score propagation, In *Proceedings of the ieee conference on computer vision and pattern recognition*. (Cit. on pp. 9, 26).

Zagoruyko, S. (2015). 92.45% on cifar-10 in torch [Accessed: 2019-07-22]. (Cit. on pp. 31, 33).

Zhang, S., Yao, L., & Sun, A. (2017). Deep learning based recommender system: A survey and new perspectives. *arXiv preprint arXiv:1707.07435* (cit. on p. 47).

Zhang, X., Zou, J., He, K., & Sun, J. (2015). Accelerating very deep convolutional networks for classification and detection. *IEEE transactions on pattern analysis and machine intelligence*, *38*(10), 1943–1955 (cit. on p. 9).

Appendix A

Additional Results

HERE we include the entire set of results obtained with ShrinkBench. For CIFAR10, results are included for CIFAR-VGG, ResNet-20, ResNet-56 and ResNet-110. Standard deviations across three different random runs are plotted as error bars. For ImageNet, results are reported for ResNet-18.

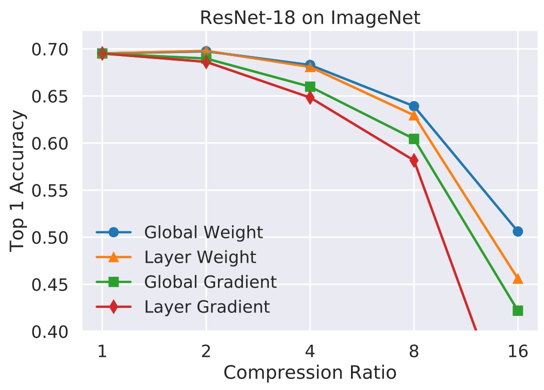

Figure A.1: Accuracy as compression varies for ResNet-18 on ImageNet

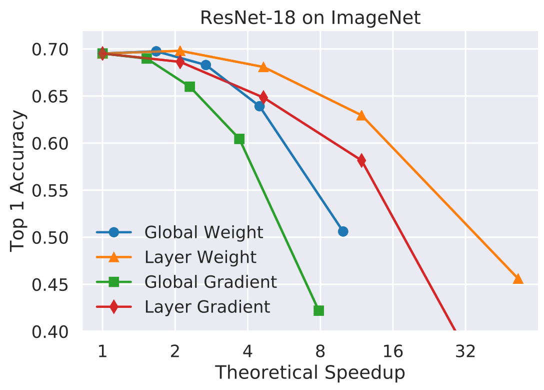

Figure A.2: Accuracy as speedup varies for ResNet-18 on ImageNet

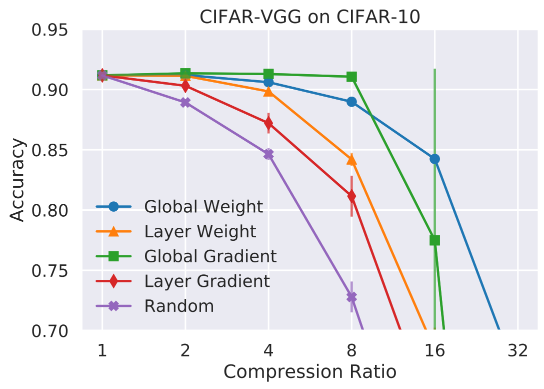

Figure A.3: Accuracy as compression varies for CIFAR-VGG on CIFAR-10

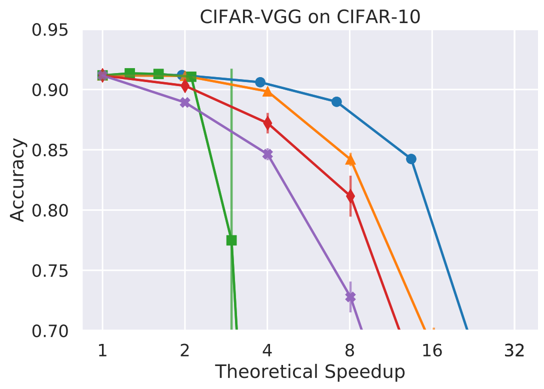

Figure A.4: Accuracy as speedup varies for CIFAR-VGG on CIFAR-10

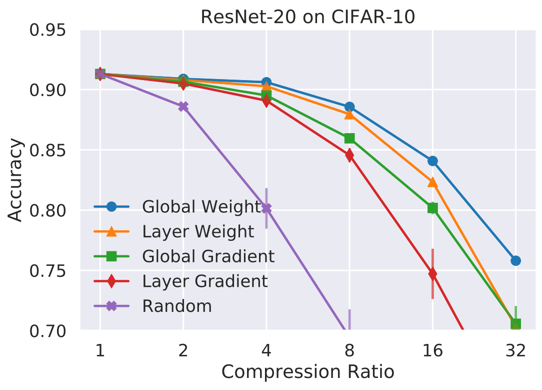

Figure A.5: Accuracy as compression varies for ResNet-20 on CIFAR-10

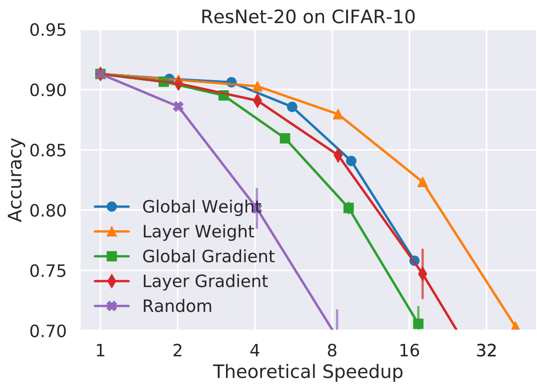

Figure A.6: Accuracy as speedup varies for ResNet-20 on CIFAR-10

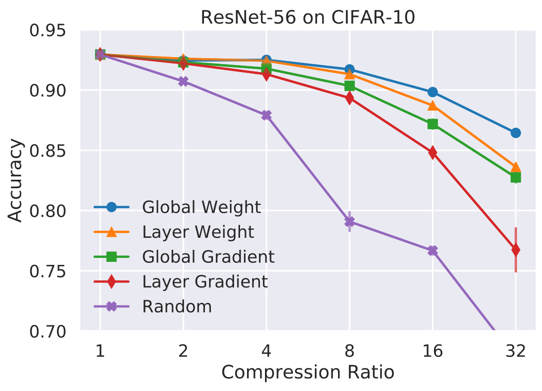

Figure A.7: Accuracy as compression varies for ResNet-56 on CIFAR-10

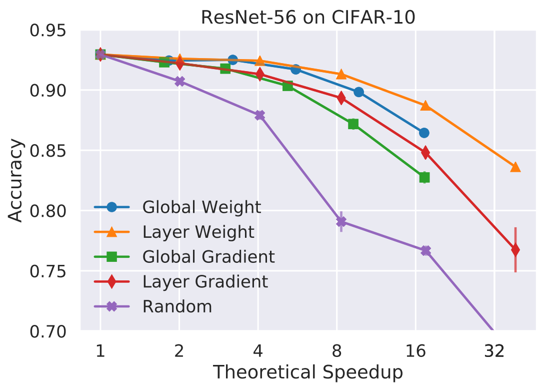

Figure A.8: Accuracy as speedup varies for ResNet-56 on CIFAR-10

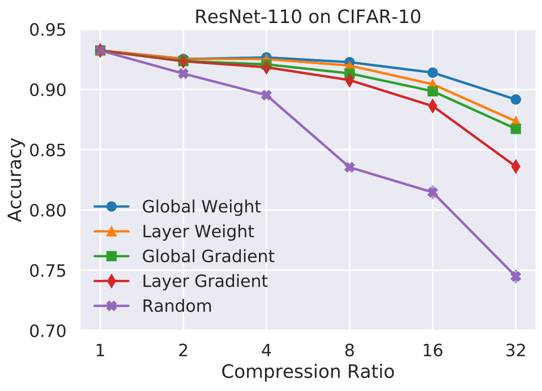

Figure A.9: Accuracy as compression varies for ResNet-110 on CIFAR-10

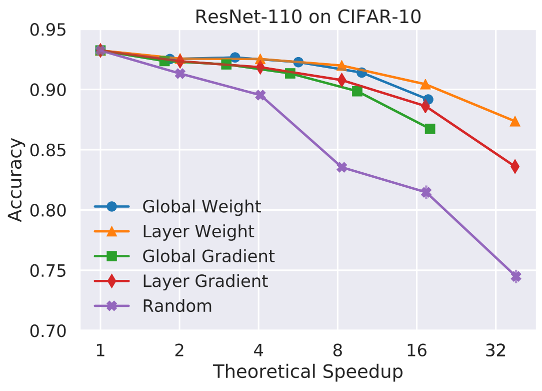

Figure A.10: Accuracy as speedup varies for ResNet-110 on CIFAR-10

Appendix B

Sustainable Development Goals

A significant fraction of the deployed machine learning models in practice, are neural network models ((Frankle & Carbin, 2019; Han et al., 2016; S. Zhang et al., 2017). Because of the large amount of parameters these networks have and the number of stochastic optimization iterations required to train them to convergence, neural networks are very computationally expensive to train. In practice, this leads to Graphical Processing Units (GPUs) to be used as hardware accelerators for all the tensor operations. This leads to a significant energy footprint associated with the training of neural network models. For instance a Nvidia P100 consumes approximately 250 W ("Nvidia Tesla P100 DataSheet," n.d.) during heavy deep learning workloads. Moreover, neural network models often need to be cross validated against different model and hyperparameter choices to maximize performance which leads to a combinatorial growth of training runs.

Neural network pruning techniques offer a way of taking a large and accurate pretrained network and decreasing its complexity whilst maintaining most of its original performance. Most pruning heuristics can be materialized in reduced training and inference times which reduce the overall energy consumption. However, because of the lack of standardized evaluation it is unclear which pruning methods could lead to smaller and more efficient models that retain accuracy.

This work is aligned from United Nations' Sustainable Development Goals. The Sustainable Development Goals (SDGs) are a collection of 17 global goals designed to be a "blueprint to achieve a better and more sustainable future for all". First, we believe that by enhancing the way neural network pruning research is carried out, we can enable a more efficient and energy responsible deployment of these models. Pruned models not only have a smaller instantaneous energy footprint, but in most cases it takes less time to finetune and to perform inference. Ensuring efficient and scalable energy consumption is a core part of the *Responsible Consumption and Production* goal and we think that there a lot of room for improvement in the energy efficiency in current neural network models.

Moreover, neural network pruning methods not only produce models that run more efficiently and with less energy consumption but they also enable the deploymnet of neural networks in platforms where it wasn't feasible before. Mobile systems and embedded devices are some examples of the kind of platforms that require neural network technology to be more energy efficient to be practical. Thus we believe that by providing a way for scientist and engineers to better evaluate and compare pruning methods, we can increase the availability and access to these technologies, a key part of the *Industry, Innovation, and Infrastructure* goal.