DEEP NEURAL NETWORK COMPRESSION

Venkata Sai Kumar Ganesula*, Sai Krishna Sangeetha†, Sricharan Maddena‡, Aishwarya Sripati§

College of Engineering and Computer Science
University of Central Florida, Orlando, Florida – 32816
* venkatasaikumarg@knights.ucf.edu
†saikrishna.sangeetha@Knights.ucf.edu
‡Sricharanmaddena@knights.ucf.edu
§aishwaryasripati@knights.ucf.edu

Abstract - **A considerable increase in computes and parameter storage costs is a result of convolutional neural networks (CNNs') success in various applications. Pruning and compressing the weights of multiple layers are recent attempts to eliminate these overheads while attempting to maintain performance [1]. The proposed novel CNN pruning criterion in this report is motivated by the interpretability of neural networks. The relevance scores of the most pertinent components, such as weights or filters, are automatically determined using explainable AI ideas. We investigate this notion and establish a link between model compression and interpretability research. We demonstrate that, in transfer-learning setups, where networks that have already been trained on sizable model are adapted to specific tasks, our proposed strategy may effectively prune the VGG16 model where CIFAR100 Dataset to evaluate the approach. However, the compressed model clearly outperforms these earlier criteria in the resource-constrained application scenario where the data of the task that is transferred to another model is very scarce, and one chooses to forgo fine-tuning. Using our method, we compressed the model size by 10 times while maintaining minimal drop in accuracy.**

Keywords: VGG16, Structured Pruning, Unstructured Pruning, Sparsity, Regularisation, Global Pruning, Fine tuning.

I. INTRODUCTION

Deep neural networks are excellent at many tasks, but they are often heavy and challenging to implement on edge devices. This can be fixed by compressing the network, more precisely, by lowering the number of parameters it employs in order to lower its requirement for dynamic memory. Additionally, by lowering the necessary number of processes, the network should theoretically operate more quickly. Network pruning is one method that is frequently

used for this. Pruning and tuning involve starting with a large, trained network and alternately eliminating connections based on some saliency metric and fine-tuning the network.

Unstructured and structured pruning techniques make up most pruning techniques. Even though the former has great characterizations, these rely on the future development of specialized hardware capable of utilizing sparse convolutions to speed up inference. We only address structured, unstructured, and fine-tuning in this work since it is more appropriate for operating networks on currently available general-purpose hardware.

From this work we use the important libraries, The general-purpose library Torch-Pruning supports a wide range of neural networks, including Vision Transformers, ResNet, Dense Net, RegNet, Res Next, FCN, Deep Lab, and VGG, among others.

II. RELATED WORK

In the beginning, pruning was created as a comparison between determining the minimal description length for neural networks. Later, it was repurposed in several research a) to enable the deployment of neural networks in resource-constrained circumstances. All these techniques are variations on unstructured pruning. This produces sparse weight matrices, which are challenging to use to increase performance on general-purpose hardware. There is no built-in support for sparsity, not even in many specialized applications like the popular Tensor Processing Unit. No built-in support for sparsity is present. The Nvidia Volta GPU is two orders of magnitude faster than the state-of-the-art sparse neural network accelerator, which only reaches a peak throughput of 2 tera-ops per second.Additionally, many of these methods concentrate largely on huge, fully connected layers, whose utilization has decreased in contemporary network architectures .

In order to expose the advantages of pruning to general purpose devices, more structured methods that all involve the removal of complete channels have been devised. These techniques can readily be used to increase performance on non-specialized hardware while preserving the density of the weight matrices without noticeably impairing their predictive power.

The hyper-parameter overhead introduced by these pruning techniques, however, means that thresholds, pruning rates, and sensitivity parameters frequently need to be hand-tuned. The literature has examined two automation

processes: sensitivity analysis integrated into saliency estimateand guided search across the space of sensitivity parameters.

In contrast to pruning-and-tuning also showed the advantages of training pruned models from scratch (although this was vehemently contested in Gale et al., 2019 for unstructured pruning scenarios). Our work, however, varies in several crucial ways:

- We think about cases when training is not limited by available resources, but inference is. Whereas Liu et al. (2019b) use computational budgets to retrain pruned models, we instead train to convergence. This is a more accurate depiction of the trimming process and a more even playing field, in our opinion.
- In this study, we evaluate many approaches to pruning on high-performance models and compare them to a high-performance pruning strategy. As an alternative to the one-shot pruning used by Liu et al. (2019b), we use iterative pruning, which has been demonstrated to improve performance and offers additional data points for analysis.
- Moreover, we analyse pruning from the viewpoint of inference time and resource efficiency on systems that stand in for common deployment targets. We demonstrate that the through to put of networks on both CPUs and GPUs is negatively impacted by the width reduction generated by structured pruning (cutting the number of channels per layer) and suggest that reduced-depth models are frequently preferable to adopt when inference performance is a goal.

III. PROBLEM STATEMENT

Deep Neural Networks (DNNs) are frequently over-parameterized, which increases the hardware platform's memory and interconnection costs. In order to shrink the size of the model, existing pruning techniques remove extraneous parameters at the end of training. However, because they don't take advantage of inherent network characteristics, they still need the complete interconnectivity to set up the network. We present a unique structural pruning strategy that comprises (1) Reducing the model size, (2) Training the Network for a Specific Dataset, and (3) Reducing the Number of Computing Operations, which is motivated by the discovery that brain networks. By efficiently reducing the model size and the amount of interconnection required before to training, the novel method produces a locally clustered and globally sparse model. We use VGG-16 for CIFAR-10 as an example, reducing the

number of parameters to 9.02% of the baseline model. Our framework focusses on compressing the overparameterized neural network models using different neural network compression techniques like pruning and fine tuning.

IV. IMPORTANT DEFINATIONS

A. VGG Model

There are a few key distinctions between the ConvNet design of Simonyan and Zisserman and those of topperforming entries in the ILSVRC-2012 and ILSVRC-2013 (Zeiler & Fergus; Sermanet et al. contests. the model suggested by Simonyan and Zisserman and our own proposed model. To begin, instead of using large receptive fields in the first convolutional layers (for example, 11 x 11 with a stride of 4 [3] or (7 x 7) with a stride of , Simonyan and Zisserman use very small receptive field size throughout the entire net (3 x 3), convolved with input at every Without any spatial pooling between them, the effective receptive field size of a pair of (3 x 3) convolutional layers are (5 x 5). The effective receptive field size of three such layers would be (7 x 7). In light of this, one could wonder what advantage Simonyan and Zisserman got by using a series of (3 x 3) convolutional layers instead of a single (7 x 7) layer. In the first place, Simonyan and Zisserman employed three non-linear rectification layers instead of one, leading to a more nuanced conclusion.

The number of parameters was also reduced, which brings us to our second point. Let's pretend the input and output of a 33-convolution stack both have C channels. The stack is therefore parameterized by 3(3 2C 2) =27C2 weights, and a single (7 × 7) conv. layer would require 7 2C 2= 49C2 parameters, an increase of 81%. It's possible to read this as forcing a regularization on the (7 7) convolutional filters, which leads to a decomposition through the (3 3) filters (with non-linearity inserted in between) By not altering the receptive fields of the convolutional layers, (1 x 1) convolutional layers boost the nonlinearity of the decision function. Although the (1 x 1) convolution is analogous to a linear projection onto the same-sized space, in the model of Simonyan and Zisserman, the rectification function adds an additional non-linearity. It's worth noting that (1 x 1) convolutional layers were employed in "Network in Network" design lately by Lin et al. The convolution filters utilized by Ciresan et al. were very tiny, but the depth of their networks was far smaller than that of Simonyan and Zisserman, and they did not conduct any experiments on the massive ILSVRC dataset.

Interesting findings were obtained by Goodfellow et al. who utilized deep ConvNets with 11 weight layers to detect street numbers, demonstrating a correlation between network depth and accuracy. Like Simonyan and Zisserman's

[work, Google Net is built on extremely deep ConvNets (22 weight layers) and tiny convolution filters, however it was developed separately. Szegedy et al.'s Google Net had a more sophisticated network topology than Simonyan and Zisserman's, and its early layers downscaled spatial resolution to minimize computation. Nonetheless, in terms of accuracy for a single network, the model developed by Simonyan and Zisserman is superior to that of GoogLeNet.

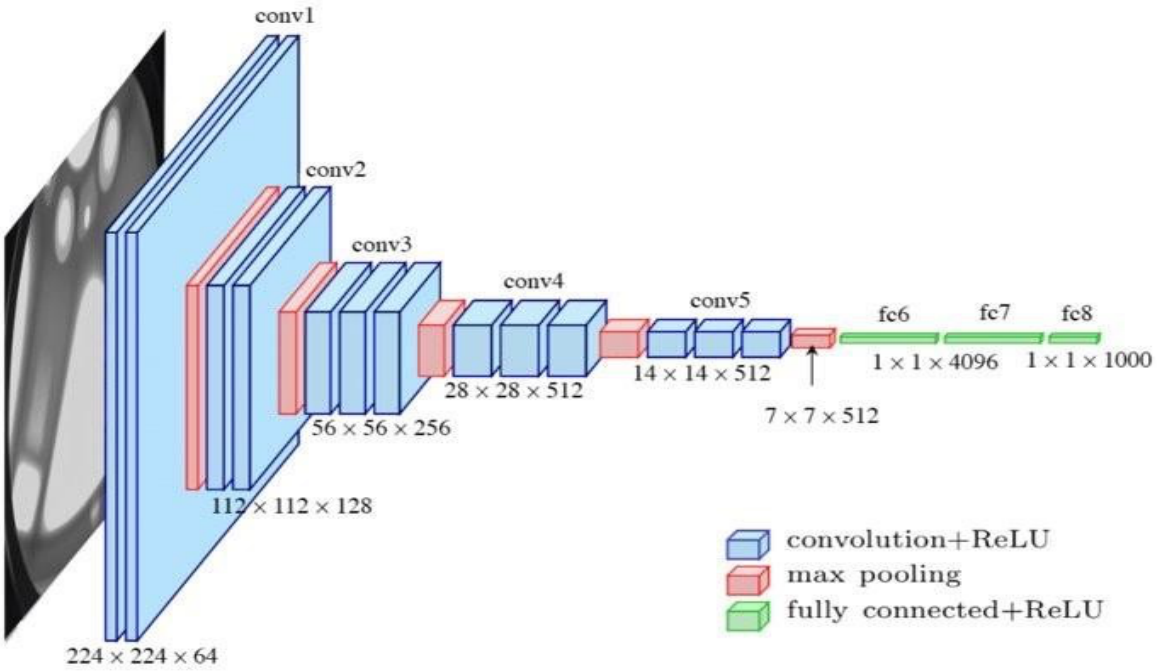

Figure 1. VGG-16 Architecture from [14].

Configuration:

The various VGG architectures were summarized in the table below. Evidently, there are two distinct VGG-16 variants (C and D). The only real difference between the two is that, except for a few convolution layers, the latter employs (3, 3) filter size convolution (1, 1). They have 138 million and 134 million parameters, respectively**.**

ConvNet Configuration					
A	A-LRN	B	C	D	E
11 weight layers	11 weight layers	13 weight layers	16 weight layers	16 weight layers	19 weight layers
input (224 × 224 RGB image)					
conv3-64	conv3-64 **LRN**	conv3-64 **conv3-64**	conv3-64 conv3-64	conv3-64 conv3-64	conv3-64 conv3-64
maxpool					
conv3-128	conv3-128	conv3-128 **conv3-128**	conv3-128 conv3-128	conv3-128 conv3-128	conv3-128 conv3-128
maxpool					
conv3-256 conv3-256	conv3-256 conv3-256	conv3-256 conv3-256	conv3-256 conv3-256 **conv1-256**	conv3-256 conv3-256 **conv3-256**	conv3-256 conv3-256 conv3-256 **conv3-256**
maxpool					
conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512 **conv1-512**	conv3-512 conv3-512 **conv3-512**	conv3-512 conv3-512 conv3-512 **conv3-512**
maxpool					
conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512 **conv1-512**	conv3-512 conv3-512 **conv3-512**	conv3-512 conv3-512 conv3-512 **conv3-512**
maxpool					
FC-4096					
FC-4096					
FC-1000					
soft-max					

Figure 2. Six Different Architectures of VGG Net

B. Sparsity

Attempts to mitigate deep learning's impact on the environment and computing resources by selectively pruning neural networks have gained traction in recent years. Sparse networks, like their biological analogues, may generalize just as well as the original dense networks, if not better. Sparsity may lessen the burden on mobile devices' memory resources and cut down on the time it takes to train ever-larger networks. We present a comprehensive introduction to sparsification for both inference and training, as well as a summary of previous work on sparsity in deep learning. Smaller dense models, factored representations of matrices, low precision values, parameter sharing among neurons,

etc. may all help reduce the size of a model. However, the major emphasis of this blog is on using sparsity to do the same thing.

The idea of sparsity is based on the intuition that not all traits are equally important in every situation. There would be less room for error in the model and hence less overfitting if feature vectors were represented in (sparse) vector space. Also, it aids interpretability, albeit maybe not in big models with many parameters. Also, it aids in the principle of least storage, which ensures that only those essences that are necessary to keep are kept , it can be represented as below,

$$sparsity = \frac{(num\ of\ zeros)}{(number\ of\ elements)}$$

Sparsity is the fraction of a tensor's elements that are zero. Strictly speaking, a tensor is said to be sparse if "most" of its components are 0. There is no hard and fast definition of "most," but you can tell a sparse tensor by its look. The l_0 - “norm” function measures how many zero-elements are in a tensor x:

$$||x||_0 = |x_1|^0 + |x_2|^0 + \cdots . . + |x_n|^0$$

It can be represented as an element contributes either a value of 1 or 0 to l_0. Anything but an exact zero contributes a value of 1.

C. Pruning modular networks

Pruning a neural network is a technique for reducing the size of a trained model by eliminating some of the weights from the network's nodes. Pruning refers to the practice of removing dead or diseased parts of a plant. Pruning refers to the process of deleting extraneous neurons or weights in machine learning.

Accuracy, size, and calculation time are all factors that should be considered when determining how effective a pruning procedure is. To evaluate the model's effectiveness, precision is essential. The size of a model is measured in bytes. FLOPs (Floating point operations) may be used as a statistic for computing time. This is independent of the platform the model is being run on, making it a more reliable metric than inference time.

When it comes to pruning, you'll have to choose between how well your model performs and how efficiently it runs. You may use extensive pruning to get a smaller, more efficient network at the expense of some accuracy. On the other hand, you may do little tree trimming and end up with a massive, high-performing network that is also very costly to maintain. For many neural networks uses, this trade-off is essential to weigh. Structured Pruning and Unstructured Pruning are the two primary categories of Pruning.

1. Unstructured Pruning

Magnitude pruning is another name for unstructured pruning. Parameters or weights of lesser size are set to zero as part of unstructured pruning. A dense network may be pruned in an unstructured fashion to produce a sparse one. The sparse network has a parameter matrix (or weight matrix) of the same size as the original network. Parameter matrices of sparse networks tend to have more gaps, or zeros.

In Unstructured Pruning, nodes in a network are deleted by individually reducing their weight connections to 0. As a result, pruning causes the network to include multiplications by 0, which may be transformed into no-ops during the prediction phase. Therefore, programs like the Neural Magic Inference Engine may get a lot more work done by using trimmed networks. The model files may be compressed and saved on disk in a far more space-efficient manner.

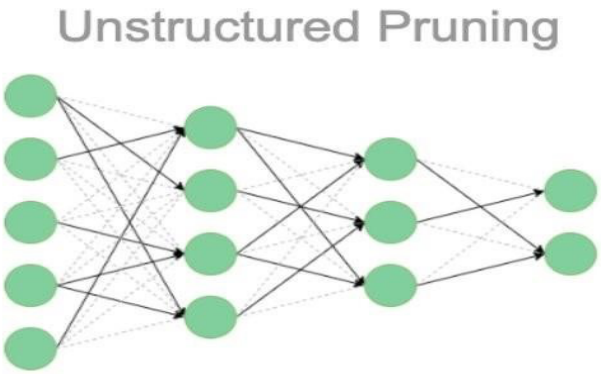

Figure 3. Unstructured Pruning

2. Global Pruning

As previously indicated, layer-wise pruning is the only method currently used in studies of robust pruning because global pruning may result in disjointed subnetworks in high pruning regimes (see Figure 1). Adopting the loss term L (which evaluates the prediction accuracy) in Equation is an easy but efficient technique to get around this restriction (4). The pruned model's loss L will considerably rise in the absence of an active path. In order to decrease L during the model pruning phase, the pruned network keeps an active path through the retained parameters = b, enabling global pruning while simultaneously guaranteeing a connected pruned network.

3. Structured Pruning

In structured pruning, a bigger portion of the network, such a layer or a channel, is removed selectively. Groupbased weights are reduced using a process known as "structured pruning" (remove entire neurons, filters, or channels of convolution neural networks). One common technique for shrinking a trained neural network is called "structured pruning," which involves eliminating channel connections and then fine-tuning them in turn to reduce the network's breadth. However, systematic pruning's effectiveness has been mostly overlooked.

Res Nets and Dense Nets generated by means of systematic pruning and tuning, both of which reveal two noteworthy facts: Taking the architecture of a pruned network and retraining it from scratch yields considerably better performance, and I reduced networks routinely beat pruned networks. We can prune once to create a family of novel, scalable network topologies, and then train them from scratch since they are so simple to approximate. Finally, we demonstrate that reduced networks are substantially quicker than pruned networks when doing inference on hardware.

Structured pruning involves the removal of channels or filters in their whole, together with several weight connections between them. Therefore, the input and output forms of layers and weight matrices are altered as a result of structured pruning. As a result, practically every computer is now capable of efficiently running structurally trimmed networks at higher speeds .Structured pruning necessitates layer-to-layer channel consistency, which must be preserved at all costs. When the model includes skip connections or concatenations, the layer dependence becomes considerably more difficult.

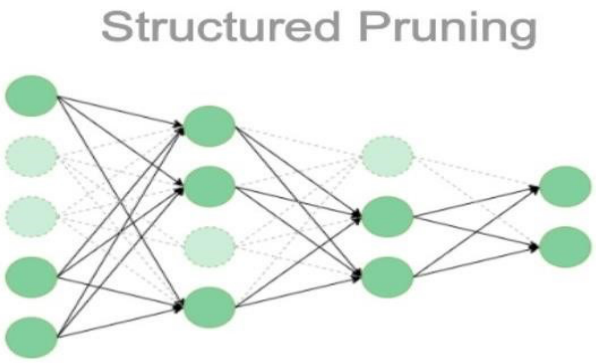

Figure 4. Structured Pruning

When a section of a network's architecture is taken out, the corresponding parameter or weight matrices get smaller.

Weight matrix size = Number of output connections × Number of incoming connections

- Prune entire (currently unpruned) channels in a tensor based on their lowest Ln-norm.
- **Ln Norm:** also called the **General form of norm.** The norm is fundamentally a function that maps a vector to a single non-negative number. • If n = 2 then it is **Euclidean** norm
- If n = Infinity, then it is **Max** norm

$$L_n\, norm\colon\ ||e||_n\ =\ [\sum |e_i|^n]^{\frac{1}{n}}$$

D. Regularization

The loss function is modified by include an L1 or L2 regularization term that punishes non-zero parameters; this is used in unstructured pruning. Non-zero parameters will be forced to zero by regularization during training.

1. ***l1-norm based pruning***

We follow the procedure described by Li et al (2016). Taxi fare or the Manhattan distance is another name for this. The L1 Norm of a space is the sum of all the vectors' lengths. The absolute difference between two vectors may be measured by adding up the differences between their individual components; this is the most intuitive method for determining how far apart two vectors are.

The norm in question assigns equal importance to each of the vector's components. Hey, demonstrate that, in comparison to random or largest-sum pruning, deleting channel activations whose associated filter weights have the least absolute sum is more effective. In this thought experiment, we will use a weight tensor of Ni K filters. One of

No possible activation channels is generated as a result of each filter being applied to one of Ni possible input activation channels. Each of these channels' relevance metrics, denoted by the symbol c, is just the aggregate of those of the individual filters. Next, we rank each model channel by c and remove the channels with the fewest t norms (where t is a temperature parameter that determines the number of channels removed after each pruning step). Since Li et al. (2016) found that t = 1 and repeated pruning produced the least error growth, we follow this strategy. This technique will thereafter be known as 1-norm pruning.

E. Fine Tuning

Strong learnt representations may be transferred to other domains with the help of fine-tuning. In most cases, we fine-tune elaborate network architectures that have already been pre-trained on massive picture datasets comprising millions of photos. As an example, we may tweak Alex Net that was pre-trained using ImageNet's collection of 1.2 million photos (61 million parameters). This is how we scale down these intricate frameworks for use in niche applications like remote sensing imagery. While the original network was pre-trained on a large dataset of natural pictures, the specialized domain may only include a subset of those images. This points to the network design being over-parameterized and inefficient in terms of memory and power consumption in comparison to the more limited new domain, where a much more lightweight network would suffice for decent performance. Applications with memory and power restrictions, such as mobile devices or robots, may benefit from a less resource-intensive network with equivalent classification accuracy.

Network pruning is an easy method for obtaining a more lightweight network from a fine-tuned network. Nonetheless, this technique has disadvantages:

Since modern pruning algorithms are heavily parameterized, it is sometimes impossible to manually search for effective pruning hyperparameters for deep networks, resulting in coarse pruning strategies.

- The activities of fine-tuning and pruning are undertaken separately.
- The criteria for pruning are specified once and cannot be modified once training has commenced.
- Because cutting-edge pruning techniques are heavily parameterized, it is often impossible to manually search for effective pruning hyperparameters for deep networks, resulting in coarse pruning strategies.

Tuning is a kind of transfer learning. The term "fine-tuning" refers to the act of adjusting the settings of a model that has previously been trained for one specific job so that it can execute a second identical task.

Depending on the nature of the data we use to train our model, the process of building and verifying it may be quite a daunting endeavour in and of itself. For this reason, the fine-tuning strategy stands out as a promising option. We can take use of the model's pre-learned knowledge and apply it to our unique work if we can identify a trained model that already executes one task effectively and the two tasks are related in at least some distant aspect.
It's best to use the learning rate to train the weights at the conclusion of the training t epoch.

1. Stochastic Gradient Descent Optimizer

One of the popular algorithms to perform neural network optimization is Gradient Descent. Gradient descent is a method for minimizing an objective function say, $\mathcal{J}(\theta)$ parameterized by a model's parameters $\theta \in \mathbb{R}^d$ by updating the parameters in the opposite direction of the gradient of the objective function $\nabla_\theta \mathcal{J}(\theta)$ with respect to the parameters. The learning rate (η), determines the size of the steps we require to reach a local minimum.

Stochastic gradient descent (SGD) performs a parameter update for each training example $x^i$and label y^i.

$$\theta = \theta - \eta \cdot \nabla_\theta \mathcal{J}(\theta;\ x^i;\ y^i)$$

SGD performs one update at a time for large datasets which helps in reducing redundancy hence, it is faster and can also be used to learn online.

2. SGD Optimizer Algorithm

1. **input**: γ (lr), θ_0(params), $f(\theta)$ (objective), λ (weight decay), μ (momentum), τ (dampening), *nesterov, maximize*
2. **for** t =1 **to**…**do**
    a. $g_t \leftarrow \nabla_\theta f_t(\theta_t - 1)$
    b. **if** λ not equals 0
        i. $g_t \leftarrow g_t + \lambda\theta_{t-1}$
    c. **if** μ not equals 0
        i. **if** $t > 1$

1. $b_t \leftarrow \mu b_{t-1} + (1-\tau) g_t$

ii. **else**

1. $b_t \leftarrow g_t$

iii. **if** *nesterov*

1. $g_t \leftarrow g_t + \mu b_t$

iv. **else**

1. $g_t \leftarrow b_t$

d. **if** *maximize*

i. $\theta_t \leftarrow \theta_{t-1} + \gamma\, g_t$

e. **else**

i. $\theta_t \leftarrow \theta_{t-1} - \gamma\, g_t$

3. **return** θ_t

F. Data set

Utilizing the CIFAR 100 dataset With the exception of the fact that there are 100 classes with a total of 600 pictures in each, this dataset is the same as the CIFAR-10. There are 500 training photographs and 100 assessment photos for each lesson. Twenty super classes make up the CIFAR-100's 100 classes. Each image includes a label that reads "fine" or "coarse," designating the class to which it belongs (the superclass to which it belongs).

V. Overview of proposed approach/system

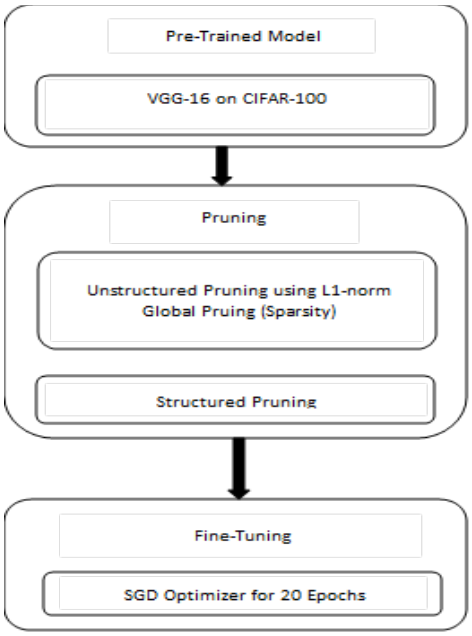

Figure 5. Proposed Architecture

VI. TECHNICAL DETAILS OF PROPOSED APPROACHES/SYSTEMS

A. Torch Pruning

The general-purpose library Torch-Pruning for structural network pruning supports a wide range of neural networks, including Vision Transformers, ResNet, DenseNet, RegNet, ResNext, FCN, DeepLab, and VGG, among others.

By eliminating redundancies, pruning is a popular method for lowering the high computational cost of neural networks. Networks are pruned using current approaches on a case-by-case basis, which involves building unique code for each model. However, it is exceedingly challenging to apply conventional pruning methods to contemporary neural networks because of the increasingly complex network architectures. Layer dependencies, where multiple layers are related and must be pruned simultaneously to ensure the accuracy of networks, are one of the most significant pruning difficulties. With the help of this project's capacity to recognize and manage layer relationships, we can prune complex networks without exerting too much manual labor.

Your model will be forwarded using fictitious inputs by Torch-Pruning, which will also trace the computational graph. jit. We'll create a dependency graph to track the relationship linking between layers. By propagating your

pruning actions across the whole graph, torch-pruning will gather all impacted layers and then deliver a Pruning Plan for pruning.

B. TorchVision

A component of the PyTorch project is this library. An open-source machine learning framework is PyTorch. Features listed in this documentation are categorized according to their release status:

- Stable: These features will be maintained for the foreseeable future, and there shouldn't typically be any significant performance restrictions or documentation gaps. Additionally, we plan to keep things backwards compatible.
- Beta: Features are designated as Beta because there is still room for improvement in terms of performance, operator coverage, or API, depending on user feedback. We commit to bringing Beta features all the way to the Stable classification. However, we do not guarantee backwards compatibility.
- Prototype: These features are still at the prototype stage and are not usually included in binary distributions like PyPI or Conda, unless they are hidden behind run-time options.

Popular datasets, model architectures, and typical image modifications for computer vision are all included in the TorchVision package.

C. PyTorch

A machine learning framework that is open-source and speeds up the transition from research prototype to deployment. PyTorch is a Python package that provides high-level features:

- Tensor computation (like NumPy) with strong GPU acceleration
- Deep neural networks built on a tape-based auto-grad system
- A complete workflow from Python to iOS and Android deployment is supported by PyTorch.

PyTorch API is expanded to include typical pre-processing and integration tasks required for implementing machine learning in mobile applications. ***D. Implementation:***

a. **Input:** Pre-trained model.
b. sparsity ratio sp %;
c. Dataset batches n1;

d. Weight prune ratios sp [] %

e. Unstructured Global Pruning

    i. for module name, module in pruning_model.named_modules()

        1. Remove Parameters

f. Return pruned model

g. Structured Pruning

    i. For each layer in prunable models

        1. if isinstance(nn.Conv2d):

            a. prune conv

        2. elif isinstance(nn.Linear):

            a. prune linear

    ii. Dependency Graph plan iii. return pruned model

h. Output: Pruned model

i. Fine Tuning

    i. For each epoch

        1. Fit(pruned model)

VII. EXPERIMENTS AND RESULTS

To demonstrate the viability of our suggested strategy, we perform experiments on VGG-16 architecture over the large-scale dataset CIFAR-100. Our test results show that the suggested method produces progressive and effective model compression.

A. VGG-16 Model on CIFAR-100 before pruning

We made use of the available built-in VGG16 model from the torch vision library which has 130+ million parameters. The model is saved to the device, and we measured the size of the model **527.802** MB. The model is initially trained on CIFAR100 dataset using Cross Entropy Loss with SGD optimizer (learning rate = 0.003 and momentum = 0.9). The model has been printed which consists of a kernel size of (3,3), stride (1,1) and 1,1 padding for each layer. The batch size has been set to 64 with 'out-classes' being 100.

On the pre-trained model, we achieved a Test Accuracy of 74.44 and Test Loss of 1.055. Size of the pre-trained

VGG-16 model is 527.802 MB.

VGG((features): Sequential (
(0) : Conv2d(3, 64, kernel_size= (3, 3), stride=(1, 1), padding=(1, 1))
(1) : ReLU (inplace=True)
(2) : Conv2d(64, 64, kernel_size= (3, 3), stride=(1, 1), padding=(1, 1))
(3) : ReLU (inplace=True)
(4) : MaxPoo12d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
(5) : Conv2d(64, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
(6) : ReLU(inplace=True)
(7) : Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
(8) : ReLU (inplace=True)
(9) : MaxPoo12d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
(10) : Conv2d(128, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
(11) : ReLU (inplace=True)
(12) : Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
(13) : ReLU (inplace=True)
(14) : Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
(15) : ReLU (inplace=True)
(16) : MaxPoo12d(kernel_size=2, stride=2, padding=0, dilation=1, ceil mode=False)
(17) : Conv2d (256, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
(18) : ReLU(inplace=True)
(19) : Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
(20) : ReLU (inplace=True)
(21) : Conv2d(512, 512, kernel_size= (3, 3), stride=(1, 1), padding=(1, 1))
(22) : ReLU (inplace=True)
(23) : MaxPoo12d(kernel_size=2, stride=2, padding-0, dilation=1, ceil _mode-False)
(24) : Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
(25) : ReLU (inplace=True)
(26) : Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
(27) : ReLU(inplace=True)
(28) : Conv2d(512, 512, kernel_size= (3, 3), stride=(1, 1), padding=(1, 1))
(29) : ReLU(inplace=True)
(30) : MaxPoo12d (kernel_size-2, stride=2, padding=0, dilation=1, ceil mode-False)
)
(avgpool): AdaptiveAvgPoo12d(output size= (7, 7)) (classifier): Sequential ((0)
: Linear (in _features=25088, out features=4096, bias=True)
(1) : RELU (inplace=True)
(2) : Dropout (p=0.5, inplace=False)
(3) : Linear (in_ features=4096, out_ features=4096, bias=True)
(4) : ReLU(inplace=True)
(5) : Dropout (p=0.5, inplace-False)
(6) : Linear (in features=4096, out features=100, bias=True)
)

B. Calculating sparsity for each layer

We know that Unstructured Pruning helps us in removing redundant or unimportant connections. So, we have introduced a sparsity of 0.8. We can observe that sparsity for each layer has seen a increasing trend as the model

contains more parameters in consecutive layer, and the last layer which is responsible for predicting and contains less parameters has reduced sparsity (See Figure 6).

Layer	Number of Zeros	Number of Elements	Sparsity
Layer 0	56	1728	0.032407407407407406
Layer 1	6081	36864	0.16495768229166666
Layer 2	12449	73728	0.1688503689236111
Layer 3	29101	147456	0.1973537868923611
Layer 4	68043	294912	0.23072306315104166
Layer 5	164093	589824	0.2782067192925347
Layer 6	160281	589824	0.2717437744140625
Layer 7	373458	1179648	0.3165842692057292
Layer 8	929979	2359296	0.3941764831542969
Layer 9	934632	2359296	0.396148681640625
Layer 10	897492	2359296	0.3804066975911458
Layer 11	905980	2359296	0.3840043809678819
Layer 12	995514	2359296	0.42195383707682294
Layer 13	93590467	102760448	0.9107635167180276
Layer 14	10156255	16777216	0.6053599715232849
Layer 15	1451421	4096000	0.354350830078125

Table 1. Results for Sparsity vs Each Layer

C. Plotting Sparsity vs Each Layer

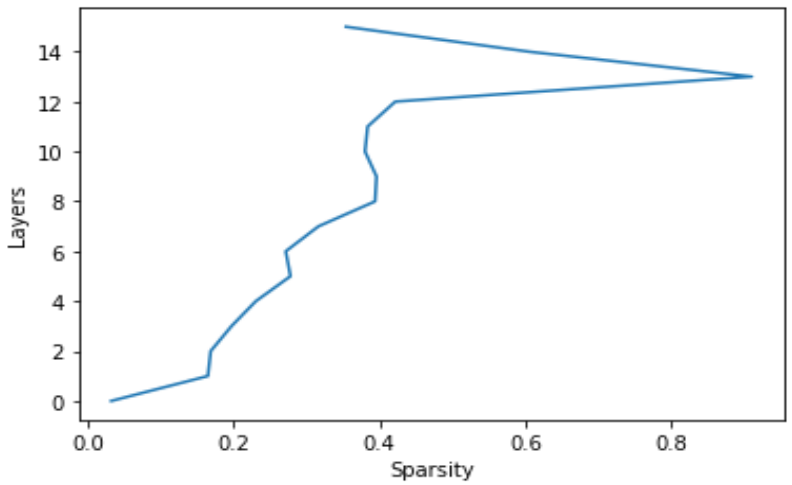

Figure 6. Sparsity vs Each Layer

D. VGG16 Model after Structured Pruning

Post the pruning we can see the number of parameters has been reduced for example in the seventh convolution layer initially contains (256, 256). But after pruning we can see the parameters have been decreased to (185, 187). After pruning we got a new model which is yet to be trained and tested. The model obtained has been tested without fine-tuning we got an accuracy of 7.31% and a loss of 4.49%.

Obtained Model Size is – 46.334 MB

VGG((features): Sequential (
(0) : Conv2d(3, 62, kernel_size= (3, 3), stride=(1, 1), padding=(1, 1))
(1) : ReLU(inplace=True)
(2) : Conv2d(62, 54, kernel_size= (3, 3), stride=(1, 1), padding=(1, 1))
(3) : ReLU (inplace=True)
(4) : MaxPoo12d(kernel_size=2, stride=2, padding 0, dilation=1, ceil mode=False)
(5) : Conv2d(54, 107, kernel_size (3, 3), stride=(1, 1), padding-(1, 1))
(6) : ReLU (inplace=True)
(7) : Conv2d(107, 103, kernel_size= (3, 3), stride=(1, 1), padding (1, 1))
(8) : ReLU (inplace=True)
(9) : MaxPoo12d(kernel_size=2, stride=2, padding=0, dilation=1, ceil mode=False)
(10) : Conv2d (103, 197, kernel _size= (3, 3), stride=(1, 1), padding=(1, 1)) (11) : ReLU (inplace=True)
(12) : Conv2d (197, 185, kernel_size=(3, 3), stride=(1, 1), padding (1, 1))
(13) : ReLU (inplace=True)
(14) : Conv2d (185, 187, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
(15) : ReLU(inplace=True)
(16) : MaxPoo12d(kernel_size=2, stride=2, padding=0, dilation=1, ceil _mode=False)
(17) : Conv2d (187, 350, kernel_size-(3, 3), stride=(1, 1), padding=(1, 1))
(18) : ReLU (inplace=True)
(19) : Conv2d(350, 311, kernel_size= (3, 3), stride=(1, 1), padding=(1, 1))
(20) : ReLU(inplace=True)
(21) : Conv2d(311, 310, kernel_size-(3, 3), stride=(1, 1), padding=(1, 1))
(22) : ReLU(inplace=True)
(23) : MaxPoo12d(kernel_size=2, stride=2, padding 0, dilation=1, ceil _mode=False)
(24) : Conv2d(310, 318, kernel_size= (3, 3), stride=(1, 1), padding=(1, 1))
(25) : ReLU (inplace=True)
(26) : Conv2d(318, 316, kernel_size-(3, 3), stride=(1, 1), padding=(1, 1))
(27) : ReLU (inplace=True)
(28) : Conv2d(316, 296, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
(29) : ReLU(inplace=True)
(30) : MaxPoo12d(kernel_size-2, stride-2, padding 0, dilation=1, ceil mode-False)
)
(avgpool): AdaptiveAvgPoo12d (output_ size=(7, 7)) (classifier): Sequential (
(0) : Linear (in features=14504, out features=366, bias=True)
(1) : ReLU (inplace=True)
(2) : Dropout (p=0.5, inplace=False)
(3) : Linear (in features=366, out_features=1617, bias=True)

(4) : ReLU (inplace=True)
(5) : Dropout (p=0.5, inplace=False)
(6) : Linear (in_features=1617, out_features=100, bias=True)
)

E. Accuracy of the Pruned VGG16 Model Before Fine Tuning

Test Accuracy: 7.31 Test Loss: 4.481076611075432

F. Fine Tuning with SGD Optimizer

During the first epoch of fine tuning using the SGD optimizer, where momentum is 0.9 and $learning\ rate(\eta)$ is roughly 0.001, train loss is 1.0395, train accuracy is 69.8120, and test loss is 1.3146 while test accuracy is 62.5000. With the same momentum and $learning\ rate(\eta)$, the train loss is 0.3366, the train accuracy has grown to 89.7160, and the test loss is 1.1417, while the test accuracy has increased to 71.2300 after 10 epochs.

TABLE I

LEARNING RATE η = 0.001, MOMENTUM = 0.9, EPOCHS = 10

Epoch	**Train Loss**	**Train Accuracy**	**Test Loss**	**Test Accuracy**
Epoch 1	1.0395	69.8120	1.3146	62.5000
Epoch 2	0.7662	77.2240	1.1516	67.3600
Epoch 3	0.6293	81.0560	1.1135	69.4900
Epoch 4	0.5646	83.0220	1.0826	69.9800
Epoch 5	0.5177	84.2300	1.0877	70.8000
Epoch 6	0.4690	85.9500	1.0963	71.1000
Epoch 7	0.4099	87.6200	1.0931	71.6300
Epoch 8	0.4026	87.5080	1.1391	71.0200
Epoch 9	0.3752	88.4620	1.1494	71.6300
Epoch 10	0.3366	89.7160	1.1417	71.2300

Table I. Results for Train Loss, Train Accuracy, Test Loss, Test Accuracy with each Epoch (10 Epochs)

After 10 epochs, we modified the $learning\ rate(\eta)$ to 0.0001 for epoch 11, and with the same momentum, we obtained results with train loss of 0.2359 and train accuracy of 92.5780, and test loss of 1.1505 and test accuracy of 73.2100. After 4 epochs Test accuracy has increased somewhat with the same $learning\ rate(\eta)$ and momentum to 73.6000, while train accuracy has also increased slightly to 93.8360.

TABLE II

LEARNING RATE η = 0.0001, MOMENTUM = 0.9, EPOCHS = 10

Epoch	Train Loss	Train Accuracy	Test Loss	Test Accuracy
Epoch:11	0.2359	92.5780	1.1505	73.2100
Epoch:12	0.2218	93.0740	1.1604	73.5300

Epoch:13	0.2177	93.1920	1.1712	73.4200
Epoch:14	0.2043	93.5640	1.1693	73.3700
Epoch:15	0.1951	93.8360	1.1893	73.6000

Table II. Results for Train Loss, Train Accuracy, Test Loss, Test Accuracy with each Epoch (5 Epochs)

Following 15 epochs, we changed the learning rate (η) to 0.000001 for epoch 16. Using the same momentum, we got results with a train loss of 0.1950 and a train accuracy of 93.9640, as well as a test loss of 1.1796 and a test accuracy of 73.5600. After 4 epochs, the accuracy of the test has improved to 73.8000 at the 20th epoch with the same learning rate and momentum, and the accuracy of the train has also improved somewhat to 94.0920.

TABLE III

LEARNING RATE η = 0.000001, MOMENTUM = 0.9, EPOCHS = 10

Epoch	**Train Loss**	**Train Accuracy**	**Test Loss**	**Test Accuracy**
Epoch 16	0.1950	93.9640	1.1796	73.5600
Epoch 17	0.1899	94.0400	1.1847	73.6600
Epoch 18	0.1888	94.0840	1.1866	73.6900
Epoch 19	0.1898	94.0160	1.1897	73.6900
Epoch 20	0.1881	94.0920	1.1907	73.8000

Table III. Results for Train Loss, Train Accuracy, Test Loss, Test Accuracy with each Epoch (5 Epochs)

G. Accuracy and Loss after Fine Tuning

Test Accuracy: 74.15 Test Loss: 1.185583380756864

VIII. CONCLUSION

We have proposed 'Deep Neural Network Compression' that compresses neural networks with minimal accuracy drop. Our works focusses on pruning a neural network by removing unwanted connections. Based on our experiments we can see that the initial size of a model is **527.802** MB and that has been reduced to **46.334 MB**, which is a compression of **91.2%,** by introducing sparsity followed by Unstructured, Structured Pruning and Fine-tuning. We achieved all of this with a very minimal drop in accuracy of 0.40%. Future works include, implementing iterative pruning and see whether any more compression is possible without losing accuracy. We can also try implementing either dynamic or post training quantization as well on the obtained model.

REFERENCES

[1] Seul-KiYeom, PhilippSeegerer, SebastianLapuschkin, AlexanderBinder, SimonWiedemann, Klaus-RobertMüller, WojciechSamek, "Explaining Deep Neural Networks and Beyond: A Review of Methods and Applications", Proceedings of the IEEE 109 (3) (2021) 1–32, doi: doi:10.1109/JPROC.2021.3060483.

[2] Benoit Jacob, Skirmantas Kligys, Bo Chen, Menglong Zhu, Matthew Tang, Andrew Howard, Hartwig Adam,

Dmitry Kalenichenko, “Quantization and Training of Neural Networks for Efficient Integer-Arithmetic-Only Inference”, arXiv:1712.05877 [cs.LG], 15 Dec 2017.
[3] Song Han, Huizi Mao, William J. Dally, “Deep Compression: Compressing Deep Neural Networks with Pruning, Trained Quantization and Huffman Coding”, arXiv:1510.00149v5 [cs.CV], 5 Feb 2016.
[4] Chen, Liyang & Chen, Yongquan & Xi, Juntong & Le, Xinyi, “Knowledge from the original network: restore a better pruned network with knowledge distillation”, Complex & Intelligent Systems. 10.1007/s40747-02000248-y, December 2020.
[5] Prakosa, S.W., Leu, JS. & Chen, ZH, “Improving the accuracy of pruned network using knowledge distillation” PatternAnalApplic24,819–830,2021.
[6] N. Aghli and E. Ribeiro, "Combining Weight Pruning and Knowledge Distillation For CNN Compression," 2021 IEEE/CVF Conference on Computer Vision and Pattern Recognition Workshops (CVPRW), 2021.
[7] Sebastian Ruder,” An overview of gradient descent optimization algorithms∗”, arXiv:1609.04747v2 [cs.LG] 15 Jun 2017
[8] Jinhyuk Park, Albert No, “Prune Your Model Before Distill It”, arXiv:2109.14960v2 [cs.LG], September 2021.
[9] Xie, H., Jiang, W., Luo, H. et al.,” Model compression via pruning and knowledge distillation for person reidentification” J Ambient Intell Human Comput 12, 2149–2161 (2021).
[10] Jangho Kim, Simyung Chang, Nojun Kwak, “PQK: Model Compression via Pruning, Quantization, and Knowledge Distillation”, arXiv:2106.14681 [cs.LG], 25 Jun 2021.
[11] Jangho Kim, Yash Bhalgat, Jinwon Lee, Chirag Patel, Nojun Kwak, “QKD: Quantization-aware Knowledge Distillation”,arXiv:1911.12491v1[cs.CV]28Nov2019.
[12] Zhuang Liu, Jianguo Li, Zhiqiang Shen, Gao Huang, Shoumeng Yan, Changshui Zhang, “Learning Efficient Convolutional Networks through Network Slimming”, arXiv:1708.06519v1 [cs.CV] 22 Aug 2017
[13] Yu Cheng, Duo Wang, Pan Zhou, and Tao Zhang, “Model Compression and Acceleration for Deep Neural Networks”, IEEEXPLORE- arnumber=8253600, January 2018.
[14] Simonyan, K., Zisserman, A.: Very deep convolutional networks for large-scale image recognition. CoRR abs/1409.1556 (2014), http://arxiv.org/abs/1409. 1556
[15] https://pytorch.org/
[16] https://www.tensorflow.org/
GitHub Code Link- Deep_Neural_Network_Compression.ipynb