Project Report - CNN Pruning

Problem Statement

In this project, we intend to implement two different CNN pruning algorithms, one-shot pruning, and iterative magnitude-based pruning, on a deep neural network model (ResNet-18) optimized for image classification tasks using CIFAR-10 dataset. We will compare the pruning algorithms' influence on *model sparsity, test accuracy, and speedups*. We will use weight pruning, an unstructured pruning algorithm that removes connections/weights from a neural network. We will use global pruning, which removes the lowest x% of connections across the whole model. The two pruning algorithms that we will use are ***One-shot pruning*** and ***Iterative pruning*** with sparsity ratios of 50%, 75% and 90%. We will experiment with different values of hyperparameters like *number of epochs*, *number of iterations* (for Iterative pruning) in combination with the aforementioned sparsity ratios.

Implementation

For this project, we have used the following configuration:

- **Deep neural network model:** ResNet-18 [18 layers deep]

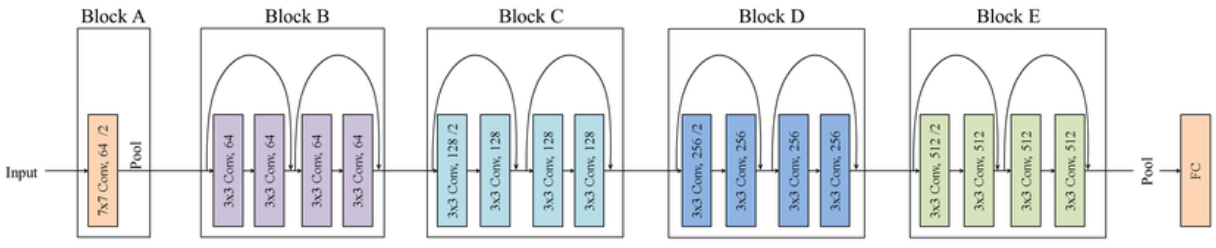


Figure: ResNet-18 architecture. Source: shorturl.at/cruN8

Details of each layer of the ResNet-18 model are as follows:

Layer Name	Output Size	ResNet-18
conv1	$112 \times 112 \times 64$	7×7, 64, stride 2
conv2_x	$56 \times 56 \times 64$	3×3 max pool, stride 2
		$\begin{bmatrix} 3 \times 3, 64 \\ 3 \times 3, 64 \end{bmatrix} \times 2$
conv3_x	$28 \times 28 \times 128$	$\begin{bmatrix} 3 \times 3, 128 \\ 3 \times 3, 128 \end{bmatrix} \times 2$
conv4_x	$14 \times 14 \times 256$	$\begin{bmatrix} 3 \times 3, 256 \\ 3 \times 3, 256 \end{bmatrix} \times 2$
conv5_x	$7 \times 7 \times 512$	$\begin{bmatrix} 3 \times 3, 512 \\ 3 \times 3, 512 \end{bmatrix} \times 2$
average pool	$1 \times 1 \times 512$	7×7 average pool
fully connected	1000	512×1000 fully connections
softmax	1000	

- **Dataset:** CIFAR-10 - The CIFAR-10 dataset consists of 60000 32x32 color images in 10 classes, with 6000 images per class. There are 50000 training images and 10000 test images.

  The dataset is divided into five training batches and one test batch, each with 10000 images. The test batch contains exactly 1000 randomly-selected images from each class. The training batches contain the remaining images in random order, but some training batches may contain more images from one class than another. Between them, the training batches contain exactly 5000 images from each class.

- **Machine learning framework:** PyTorch.

Our implementation of the project can be divided into the following steps:

1. We downloaded a pre-trained ResNet-18 model (base model) from GitHub - huyvnphan/PyTorch_CIFAR10: Pretrained TorchVision models on CIFAR10 dataset (with weights) and model training library functions from PyTorch_CIFAR10/resnet.py at master which will be used later for re-training the pruned version of the base model.

2. Once we have downloaded the appropriate resources from the above mentioned links, we proceed with running inference on the test dataset using this base model in "*cs532_project2_pruning.ipynb*". In the program, we specify to choose cuda enabled GPU resources (if available, otherwise default to CPU) for running the code. We load the test and training datasets using the DataLoader library. This pre-trained model gives us a baseline *accuracy* of 93.07% and *Inference time per image* of 0.0125064 seconds.

3. Once we have the baseline performance metrics available, we then proceed with various pruning experiments. Every pruning experiment consists of the following steps:

    a. Load the base model as per Step 2 and also load the testing and training datasets using the DataLoader library. Also calculate the inference metrics on the base model.
    b. Prune the base model using prune.global_unstructured() function from torch.nn.utils.prune library.
    c. Display the sparsity values in each layer of the pruned model.
    d. Re-train the pruned model and fine-tune the hyperparameters like number of epochs, number of iterations, learning rate, etc. to get the best performance possible.
    e. Evaluate the performance of the fine-tuned version of the pruned model by calculating the accuracy, accuracy drop, inference time, speedup etc. and comparing with the metric value of the unpruned base model as the basis.

One-shot Pruning:

We have implemented the One-shot pruning algorithm as per the following steps:

I. We first imported the required library/package:

```
import torch.nn.utils.prune as prune
```

II. Then, we specify the parameters to be pruned. Since in this experiment we are going to prune weights, we mention the weights of the layers of the ResNet-18 model as parameters to the pruning function.

```
parameters_to_prune = (

    (model.conv1, 'weight'),

    (model.layer1[0].conv1, 'weight'),
    (model.layer1[0].conv2, 'weight'),
    (model.layer1[1].conv1, 'weight'),
    (model.layer1[1].conv2, 'weight'),

    (model.layer2[0].conv1, 'weight'),
    (model.layer2[0].conv2, 'weight'),
    (model.layer2[1].conv1, 'weight'),
    (model.layer2[1].conv2, 'weight'),

    (model.layer3[0].conv1, 'weight'),
    (model.layer3[0].conv2, 'weight'),
    (model.layer3[1].conv1, 'weight'),
    (model.layer3[1].conv2, 'weight'),

    (model.layer4[0].conv1, 'weight'),
    (model.layer4[0].conv2, 'weight'),
    (model.layer4[1].conv1, 'weight'),
    (model.layer4[1].conv2, 'weight'),

    (model.fc, 'weight'),
```

III. We then finally call the global_unstructured() function to prune the model. The **'amount'** attribute specifies the degree of sparsity in the network. For eg: amount = 0.75 corresponds to 75% sparsity. In one-shot pruning, the entire pruning operation is executed all at once and the model is retrained only once.

```
prune.global_unstructured(
    parameters_to_prune,
    pruning_method=prune.L1Unstructured,
    amount=0.75,
)
```

Iterative Pruning:

We have implemented the Iterative pruning algorithm as per the following steps:

I. We first imported the required library/package:

```
import torch.nn.utils.prune as prune
```

II. Then, we specify the parameters to be pruned. Since in this experiment we are going to prune weights, we mention the weights of the layers of the ResNet-18 model as parameters to the pruning function.

```
parameters_to_prune = (

    (model.conv1, 'weight'),

    (model.layer1[0].conv1, 'weight'),
    (model.layer1[0].conv2, 'weight'),
    (model.layer1[1].conv1, 'weight'),
    (model.layer1[1].conv2, 'weight'),

    (model.layer2[0].conv1, 'weight'),
    (model.layer2[0].conv2, 'weight'),
    (model.layer2[1].conv1, 'weight'),
    (model.layer2[1].conv2, 'weight'),

    (model.layer3[0].conv1, 'weight'),
    (model.layer3[0].conv2, 'weight'),
    (model.layer3[1].conv1, 'weight'),
    (model.layer3[1].conv2, 'weight'),

    (model.layer4[0].conv1, 'weight'),
    (model.layer4[0].conv2, 'weight'),
    (model.layer4[1].conv1, 'weight'),
    (model.layer4[1].conv2, 'weight'),

    (model.fc, 'weight'),
```

III. We implemented the Iterative pruning algorithm by repeatedly performing one-shot pruning + retraining cycles until the eventual required sparsity value is achieved. We first specify the number of iterations. Let's say that the number of iterations is 'n' and the required sparsity ratio is 's'. The value of the amount parameter for pruning in each iteration can be evaluated by solving the following equation for x.

$$(1-x)^n = s$$

For eg: the value of x for each iteration for s = 0.9 and n = 3 is 0.0345.

```
iterations = 3
for itr in range(iterations):
  print("Iteration - %d \n"%itr)
  prune.global_unstructured(
    parameters_to_prune,
    pruning_method=prune.L1Unstructured,
    amount=0.0345,
  )
  criterion = nn.CrossEntropyLoss()
  optimizer = torch.optim.SGD(model.parameters(), lr=learning_rate, momentum=0.7)
  scheduler = torch.optim.lr_scheduler.MultiStepLR(optimizer, milestones=[150, 200], gamma=0.1)

  for mod in modules:
      prune.remove(mod, 'weight')

  for epoch in range(num_epochs):
    start = time.time()

    for i, (inputs, targets) in enumerate(train_loader):
        inputs = inputs.to(device)
        targets = targets.to(device)
        outputs = model(inputs)
        loss = criterion(outputs, targets)
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()
        end = time.time()

        if (i+1) % 100 == 0:
          print('Batch Index : %d Loss : %.4f Time : %.3f seconds ' % (
              i+1,
              loss.item(),
              end - start))
```

Model re-training:

We have implemented the model re-training procedure by using CrossEntropyLoss() function as the loss function and Stochastic Gradient Descent (SGD) as the gradient descent algorithm. We proceed by specifying the number of epochs (repeats) for which the entire training dataset will be used to train/retrain the model. The training loop also implements backward propagation at every training instance step.

```
    # criterion = nn.MSELoss()
    criterion = nn.CrossEntropyLoss()
    optimizer = torch.optim.SGD(model.parameters(), lr=learning_rate, momentum=0.7)
    scheduler = torch.optim.lr_scheduler.MultiStepLR(optimizer, milestones=[150, 200], gamma=0.1)

    for mod in modules:
        prune.remove(mod, 'weight')

    # def update_lr(optimizer, lr):
    #     for param_group in optimizer.param_groups:
    #         param_group['lr'] = lr


    # total_step = len(train_loader)
    # curr_lr = learning_rate
    for epoch in range(num_epochs):
      # losses = []
      scheduler.step()
      start = time.time()

      for i, (inputs, targets) in enumerate(train_loader):
          inputs = inputs.to(device)
          targets = targets.to(device)

          outputs = model(inputs)
          loss = criterion(outputs, targets)

          optimizer.zero_grad()
          loss.backward()
          optimizer.step()
```

Experimental Results

This section reports the evaluation methodology and significant results achieved. It includes both the sparsity and the test accuracy metrics.

The results of the experiment can be summarized in the following tables and charts:

a) As part of this project, we have implemented the following 12 test cases:

Algorithm	Sparsity	Num Epochs	Iterations
Oneshot	0.5	3	NA
Oneshot	0.5	5	NA
Oneshot	0.75	3	NA
Oneshot	0.75	5	NA
Oneshot	0.9	3	NA
Oneshot	0.9	5	NA
Iterative	0.5	3	3
Iterative	0.5	5	5
Iterative	0.75	3	3
Iterative	0.75	5	5
Iterative	0.9	3	3
Iterative	0.9	5	5

- We have implemented each of the pruning algorithms with 2 different values of the number of epochs to get better performance metrics.
- The learning rate for each of the experiments was finally chosen as 0.0005. We arrived at this value by trying out different values and finally choosing the most suitable value to balance the trade-off between accuracy and model training time.

Other definitions:

- *Accuracy drop* = Accuracy of base model - Accuracy of the pruned model
- *Speedup* = Inference time per image for base model / Inference time per image for pruned model

b) **Base model stats**:

The base model of ResNet-18 was then evaluated on the test dataset and the following results were obtained:

Accuracy	Inference time per image
93.07%	0.0125064

c) **One-shot pruning**:

Once we had the base model stats available, we then proceeded with the first pruning algorithm - Oneshot pruning. The results of Oneshot pruning experiment are as follows:

Sparsity	Test Accuracy for 3 epochs	Test Accuracy for 5 epochs	Accuracy Drop for 3 epochs	Accuracy Drop for 5 epochs
50%	92.93%	93.05%	0.14%	0.02%
75%	93%	93.24%	0.07%	-0.17%
90%	93%	93.09%	0.07%	-0.02%

Sparsity	Inference time per image for 3 epochs	Inference time per image for 5 epochs	Speedup for 3 epochs	Speedup for 5 epochs
50%	0.0126116	0.0127628	0.991601	0.9799165
75%	0.0129951	0.0126724	0.962394	0.9869027
90%	0.0123727	0.0124647	1.010811	1.003344

Observations:

i) For 3 epochs, we can observe that the test accuracies are slightly lower than the Base model accuracy of 93.07%.
ii) For 5 epochs, the pruned model after re-training learns the most important weights thoroughly and the resultant model slightly outperforms the base model by a small magnitude.
iii) For Sparsity ratios 50% and 75%, we do not observe significant speedup and for 90% sparsity we observe a speedup by a small margin. The possible reasons for not observing any significant speedup could be because we are not using *torch.sparse* library in our implementation which is optimized to work with sparse tensors. [Source]

d) **Iterative pruning:**

Once we have completed the Oneshot pruning experiment, we then proceed with the second pruning algorithm - Iterative pruning. The results of Iterative pruning experiment are as follows:

Sparsity	Test Accuracy for 3 epochs and 3 iterations	Test Accuracy for 5 epochs and 5 iterations	Accuracy Drop for 3 epochs and 3 iterations	Accuracy Drop for 5 epochs and 5 iterations
50%	93.17%	92.89%	-0.10%	0.18%
75%	93.22%	93.13%	-0.15%	-0.06%
90%	93.01%	93.13%	0.06%	-0.06%

Sparsity	Inference time per image for 3 epochs and 3 iterations	Inference time per image for 5 epochs and 5 iterations	Speed up for 3 epochs and 3 iterations	Speedup for 5 epochs and 5 iterations
50%	0.012549	0.0126961	0.996605	0.985058
75%	0.0125771	0.012793	0.994379	0.977597
90%	0.0125455	0.0127079	0.996883	0.984144

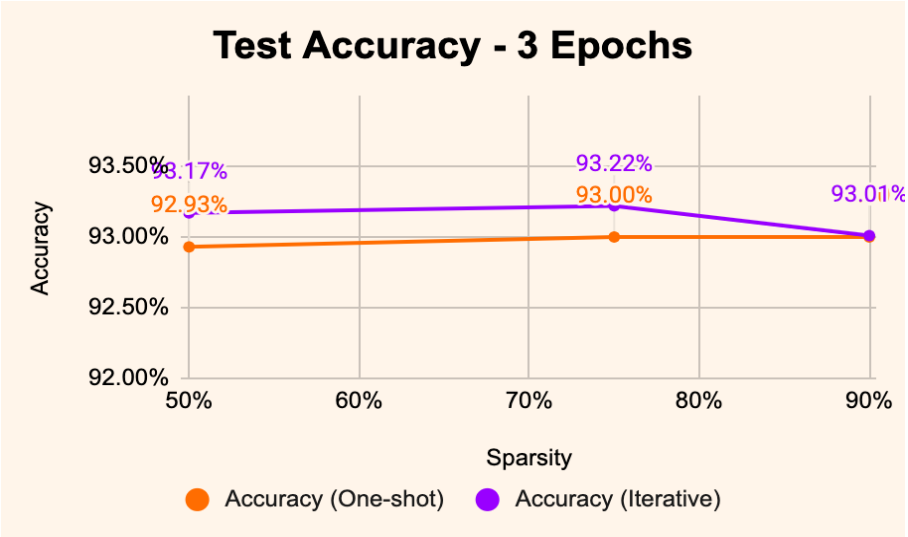
Observations:

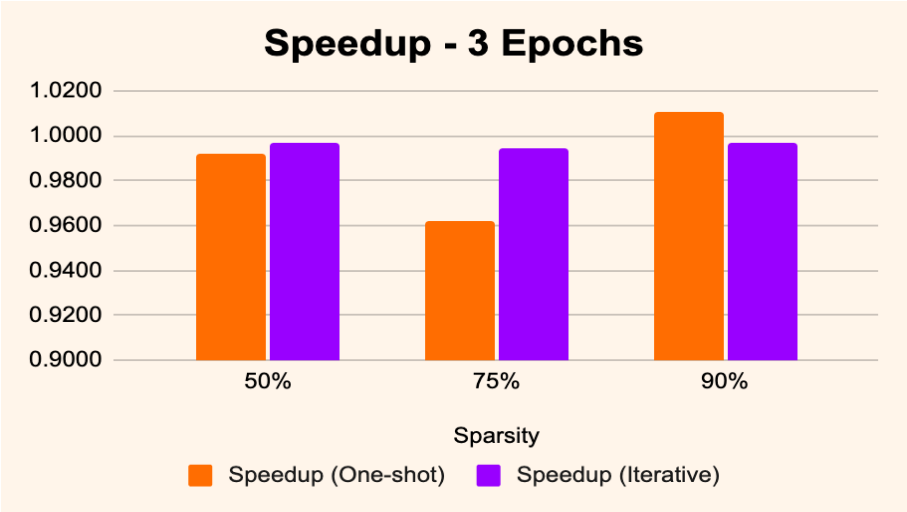
i) For 3 epochs, we can observe that the test accuracies are slightly higher than the Base model accuracy of 93.07% except for 90% sparsity.

ii) While the iterative pruning method results in higher accuracy than in One-shot algorithm, we also observed that relatively better speedup was achieved in the Oneshot algorithm.

iii) Iterative pruning has higher accuracy because the algorithm greedily prunes the least significant weights and retrains the model in each iteration which eventually acts like a progressively learning pruning algorithm. Every higher iteration tells us more about the importance of each weight than the weights in the previous iteration. Iterative pruning takes advantage of this phenomenon and makes the pruned model more accurate.

Additional insights (Comparison of One-shot vs Iterative**):**

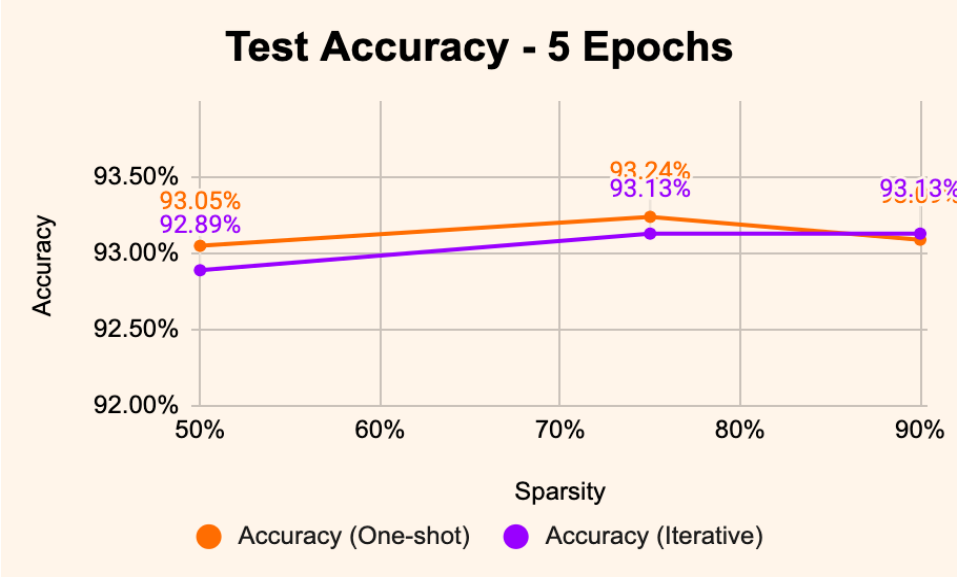


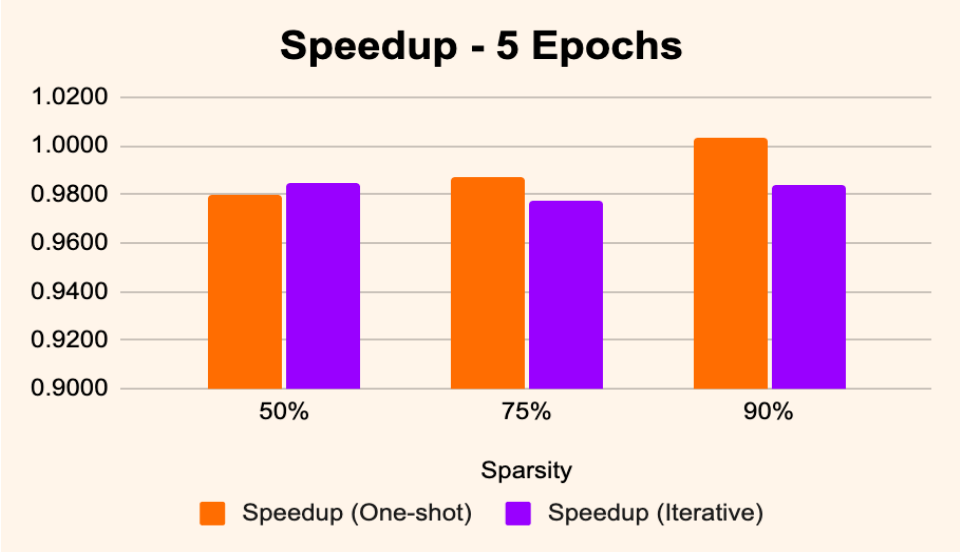


Observations:

i) For fewer epochs (in our case, 3), Iterative pruning results in higher accuracy.

ii) As the sparsity of the model increases, the accuracies of Iterative and One-shot pruning converge for both 3 and 5 epochs.





iii) For 5 epochs, as the sparsity ratio increases, the speedup of One-shot pruning algorithm starts to outperform Iterative pruning speedup. At 90% sparsity, the One-shot pruned model outperformed the base model in inference time for both 3 and 5 epochs.

iv) For both 3 and 5 epochs & among the 3 sparsity ratios, the accuracy of the model is maximum at 75% and lower on the other 2 ends. This could be because when we pruned the model by only 50%, we still retained the less important weights before re-training the model. This could have retained the inaccuracy causing weights in the model. On the other hand, the accuracy slightly reduces at 90% pruning because we could have possibly pruned out more

weights than required which could cause some important weights of the model to be lost resulting in lower accuracy.

v) *Typical sparsity in each layer in ResNet-18 after running the unstructured global pruning algorithms* - We can observe that the sparsity is unevenly distributed with higher sparsity values in the later layers of the neural network.

```
Sparsity in conv1: 11.98%
Sparsity in layer1->basicblock0->conv1: 17.56%
Sparsity in layer1->basicblock0->conv2: 2.66%
Sparsity in layer1->basicblock1->conv1: 2.82%
Sparsity in layer1->basicblock1->conv2: 3.07%
Sparsity in layer2->basicblock0->conv1: 2.64%
Sparsity in layer2->basicblock0->conv2: 2.87%
Sparsity in layer2->basicblock1->conv1: 3.57%
Sparsity in layer2->basicblock1->conv2: 4.58%
Sparsity in layer3->basicblock0->conv1: 4.82%
Sparsity in layer3->basicblock0->conv2: 7.62%
Sparsity in layer3->basicblock1->conv1: 12.82%
Sparsity in layer3->basicblock1->conv2: 20.57%
Sparsity in layer4->basicblock0->conv1: 36.38%
Sparsity in layer4->basicblock0->conv2: 52.13%
Sparsity in layer4->basicblock1->conv1: 82.45%
Sparsity in layer4->basicblock1->conv2: 68.15%
Sparsity in fc: 0.04%
```

Summary and Takeaway

From this project, we learnt the following things:

a) Pruning is done to reduce the memory and processing requirements of a NN model without significantly affecting the performance of the model. Having said that, we must also state that the pruning functionality provided by PyTorch seems to be at an experimental stage and cannot significantly improve model performance. Theoretically, it allows you to prune the model and re-train resulting in sometimes marginal increase in performance, but practically we need to make use of more efficient libraries on pruned models (like *torch.sparse*) for actual realization of performance gains in inference times.

b) Re-training pruned models does generally increase the model accuracy by retaining the most important weights from the non-pruned model and building a new one. *"In effect, this training process learns the network connectivity in addition to the weights - much as in the mammalian brain, where synapses are created in the first few months of a child's development, followed by gradual pruning of little-used connections, falling to typical adult values."* - (Source - [2])

c) Parameters like sparsity, learning_rate and number of epochs are hyperparameters that enable fine-tuning of the model. As we have observed before, the optimal value of sparsity is somewhere in between too much and too little sparsity. An interesting takeaway from this experiment is that a significant number of weights in a large neural network are of no use and their removal does not significantly affect the eventual performance of the model.

Team Contribution

Team Members:

- Prasann Desai
- Akhil Reddy Anthireddy
- Muneer Ahmed

All of us worked on the code and the final report. Then we divided the work, by each one executing both One-shot pruning and Iterative pruning for a particular sparsity ratio. Prasann

ran both the models for 50% sparsity, Akhil for 75% sparsity and Muneer for 90% sparsity. Then we collectively worked on the report and the testing script. Prasann worked on the Implementation, Experimental results sections in the report, Akhil worked on the Artifact Evaluation section, collaborated for Experimental results section and created test scripts for both colab and local machine and Muneer worked on the Problem statement, Summary and Takeaway, References sections.

Artifact Evaluation

How to Run the Test Cases - We have written a test script py file for running on local and a notebook(.pynb) to run on colab. Either of the methods can be used to run the test script.

1) **Method - 1** (Running py file on the local)

    - Clone the repository and go inside the root directory.
    - Required libraries for running the code - numpy, torch, torchvision
    - Please install all the above libraries using pip3 eg. - “pip3 install torch”
    - Please install any other dependencies required and not mentioned above.
    - Run the test_script.py file with the command “sudo python3 testing_script.py”
    - Each model must take approximately a minute to run and print the accuracy and inference time

2) **Method -2 (**Running the notebook on colab)

    - Clone the repository to a folder named “cs532_project2” in your local
    - Upload the “cs532_project2” folder to google drive
    - The folder “cs532_project2” must contain the “src” folder and other .pt model files
    - Open the “src” folder and open the testing_script.ipynb file in google colab and run each cell
    - Each model must take approximately a minute to run and print the accuracy and inference time

PyTorch files:
We have named the pytorch files by including the sparsity percentage, type of algorithm(oneshot or iterative) and the number of epochs in the end, in the following manner:

resnet18_prune_’’***Sparsity***’perc_***’oneshot/iterative’*** epochs_iterations(only for iterative)

Here we have considered 3 epochs and 5 epochs only
Thus we have a total of 12 pytorch files:

1. Resnet18_prune_50perc_iterative3_3
2. Resnet18_prune_50perc_iterative3_5
3. resnet18_prune_50perc_oneshot3
4. resnet18_prune_50perc_oneshot5
5. Resnet18_prune_75perc_iterative3_3
6. Resnet18_prune_75perc_iterative3_5
7. resnet18_prune_75perc_oneshot3
8. resnet18_prune_75perc_oneshot5
9. Resnet18_prune_90perc_iterative3_3
10. Resnet18_prune_90perc_iterative3_5
11. resnet18_prune_90perc_oneshot3
12. resnet18_prune_90perc_oneshot5

References

1. https://github.com/huyvnphan/PyTorch_CIFAR10/tree/master/cifar10_models
2. Song Han, John Tran, Jeff Pool, William J. Dally. "Learning both Weights and Connections for Efficient Neural Networks".
3. Jason Brownlee. "PyTorch Tutorial: How to Develop Deep Learning Models with Python".
4. Michela Paganini. "Pruning Tutorial - PyTorch"
5. ResNet with 18 layers depicted in five blocks. | Download Scientific Diagram
6. ResNet-18 Architecture. | Download Table
7. CIFAR-10 and CIFAR-100 datasets
8. https://discuss.pytorch.org/t/purpose-of-pytorch-pruning/147457