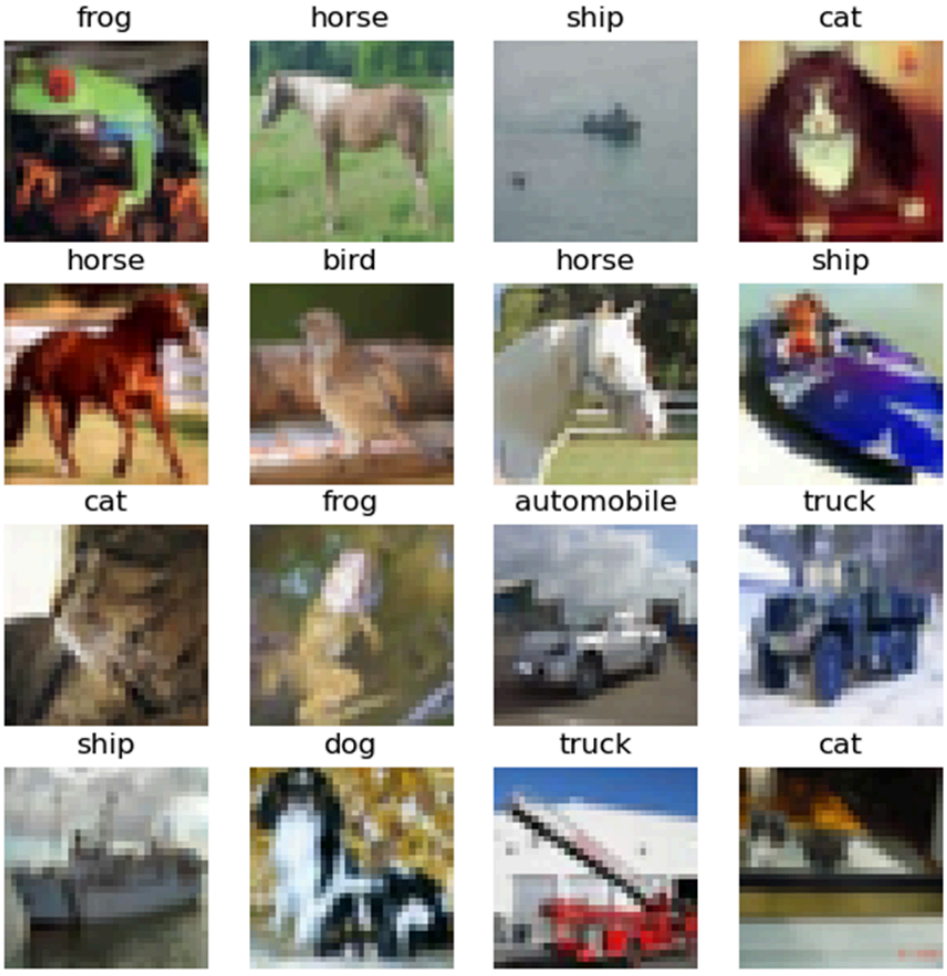
CSI 4140/5140 — Project2 Report

Group 9
Mahdi Moghaddami - G00869650

1. Introduction

In Project 1, we implemented a deep neural network (DNN) to classify images from the famous CIFAR-10 dataset. The CIFAR-10 dataset (Canadian Institute For Advanced Research) is a collection of images commonly used to train machine learning and computer vision algorithms. It is one of the most widely used datasets for machine learning research. The CIFAR-10 dataset contains 60,000 32x32 color images in 10 different classes. The 10 classes represent airplanes, cars, birds, cats, deer, dogs, frogs, horses, ships, and trucks. There are 6,000 images of each class. Here are some example images and their labels from the dataset:



In this project, we are trying to compress the DNN implemented in Project 1 using at least one of the three compression methods we studied in class: Pruning, Knowledge Distillation (KD), or Quantization. Model compression is important for several reasons:

- **Reduced Memory Footprint:** Compressed models require less memory, making them suitable for deployment on devices with limited storage, such as mobile phones and embedded systems.
- **Faster Inference:** Smaller models typically have fewer parameters and operations, leading to faster inference times. This is crucial for real-time applications like autonomous driving and augmented reality.
- **Lower Power Consumption:** Compressed models consume less power, which is beneficial for battery-powered devices and helps in reducing operational costs in large-scale deployments.
- **Efficient Deployment:** Smaller models are easier to deploy and update across a wide range of devices and platforms, improving scalability and maintainability.
- **Bandwidth Savings:** Transmitting smaller models over networks requires less bandwidth, which is advantageous for cloud-based applications and edge computing.
- **Cost Reduction:** Reduced computational and storage requirements can lead to cost savings in both hardware and cloud infrastructure.

2. Background and Related Works

Deep learning models often have millions or even billions of parameters, which can make them computationally expensive and memory-intensive. Model compression methods aim to reduce the size and complexity of these models while maintaining or minimally sacrificing performance. Two widely used techniques for model compression are pruning and knowledge distillation.

2.1 Pruning

Pruning involves the systematic removal of unnecessary parameters (weights, neurons, or layers) from a model. This method is based on the observation that many deep learning models are overparameterized, meaning they have more parameters than are necessary for good performance.

Types of Pruning:

1. Weight Pruning:
    - Removes individual weights in the model that have minimal impact on the final output (e.g., those with small magnitude).
    - Techniques:
        - Magnitude-based pruning: Prunes weights with the smallest absolute values.
        - Threshold-based pruning: Removes weights below a certain threshold.

2. Structured Pruning:
    - Removes entire structures such as neurons, filters, or layers.
    - Often results in more hardware-efficient models as it aligns with matrix operations.
    - Techniques:

    - Filter pruning: Removes convolutional filters.
    - Neuron pruning: Removes neurons in fully connected layers.

3. Dynamic/Iterative Pruning:
    - Pruning is applied gradually during or after training, allowing the model to adjust and retrain to compensate for the loss of parameters.

2.1 Knowledge Distillation

Knowledge distillation is a technique where a smaller, simpler model (the "student") learns from a larger, more complex model (the "teacher"). Instead of training the student model directly on the dataset, it is trained to mimic the teacher's behavior by approximating its output.

- Soft Targets:
    - The student model learns from the soft probabilities (output logits) produced by the teacher model rather than hard labels.
    - These soft targets provide more information about class relationships, which aids learning.

- Loss Function:
    - Combines two components:
        - Distillation loss: Minimizes the difference between the student's and teacher's output distributions.
        - Standard classification loss: Ensures the student still performs well on the original task.

- Temperature Scaling:
    - Softens the teacher's output distribution using a temperature parameter T. Higher temperatures make the output distribution smoother, emphasizing knowledge transfer.

3. Methodology

I implemented a combination of KD and pruning for this project. I train a smaller student model using the original model from Project 1 as the teacher. Then, I use pruning to compress the student model and compress the model even further.

3.1 Knowledge Distillation

The implementation of this part can be found in file `kd.ipynb`. I train two different student models: `SmallStudentModel` and `StudentModel`. Both models can be found at `model.py` in the submission folder. Let's take a look at the number of parameters for each model:

	Small Student	Student	Teacher
# Parameters	89,834	156,074	553,514

The architecture of the models is identical. However, I changed the number of output channels, and the size of the linear layers to make smaller iterations of the teacher model.
While training the student models, we use the output of the teacher model too. This is the key aspect of KD and is implemented via having a cost function that takes into account the difference between the output logits of the student and the teacher. This is how the cost function is calculated for the student model:
$cost = \alpha * KLdiv + (1 - \alpha)NLLL$
Where KLdiv: Kullback-Leibler divergence Loss, NLLL: Negative Log-Likelihood Loss, alpha: Hyperparameter in range [0, 1].
KL divergence loss is calculated using the outputs of both the student and the teacher models. The NLL loss however comes only from the student model's outputs. Here's the code implementation in detail:

```
def distillation_loss(
    y_student: torch.Tensor,
    y_teacher: torch.Tensor,
    y_true: torch.Tensor,
    alpha: float = 0.5,
    temperature: float = 2.0,
) -> torch.Tensor:
    distillation_loss = nn.KLDivLoss(reduction="batchmean")(
        nn.functional.log_softmax(y_student / temperature, dim=1),
        nn.functional.softmax(y_teacher / temperature, dim=1),
    ) * (temperature**2)
    student_loss = criterion(y_student, y_true)
    return alpha * distillation_loss + (1 - alpha) * student_loss
```

`Adam` optimizer and `CosineAnnealingLR` scheduler were used in training the student models:

```
optimizer = torch.optim.Adam(student_model.parameters(), lr=1e-3,
weight_decay=5e-4)
scheduler = torch.optim.lr_scheduler.CosineAnnealingLR(optimizer,
epochs)
```

All models were trained for 60 epochs on an NVIDIA RTX 4070 Ti GPU. The student model is saved into the models folder after training.

3.2 Pruning

After applying KD and training a smaller student model, I used pruning to compress the student model even further. The implementation for this part of the project can be found in `pruning.ipynb`. The main part of the code in use is highlighted below:

```
def fine_grained_prune(tensor: torch.Tensor, sparsity: float) ->
torch.Tensor:
    sparsity = min(max(0.0, sparsity), 1.0)
    if sparsity == 1.0:
        tensor.zero_()
        return torch.zeros_like(tensor)
    elif sparsity == 0.0:
        return torch.ones_like(tensor)
    num_elements = tensor.numel()
    num_zeros = round(num_elements * sparsity)
    importance = tensor.abs()
    threshold = importance.view(-1).kthvalue(num_zeros).values
    mask = torch.gt(importance, threshold)
    tensor.mul_(mask)
    return mask


class FineGrainedPruner:
    def __init__(self, model, sparsity_dict):
        self.masks = FineGrainedPruner.prune(model, sparsity_dict)

    @torch.no_grad()
    def apply(self, model):
```

```
        for name, param in model.named_parameters():
            if name in self.masks:
                param *= self.masks[name]


    @staticmethod
    @torch.no_grad()
    def prune(model, sparsity_dict):
        masks = dict()
        for name, param in model.named_parameters():
            if param.dim() > 1:
                if isinstance(sparsity_dict, dict):
                    masks[name] = fine_grained_prune(param,
sparsity_dict[name])
                else:
                    assert sparsity_dict < 1 and sparsity_dict >= 0
                    if sparsity_dict > 0:
                        masks[name] = fine_grained_prune(param,
sparsity_dict)
        return masks
```

This class is designed to prune the weights of a neural network model to achieve sparsity, which can help in reducing the model size and potentially improve inference speed. The degree of sparsity is a hyperparameter. We'll see how this hyperparameter affects the results in the ablation study part in Section 5 of this report.

After pruning the loaded student model, I fine-tune the model for 10 epochs. Fine-tuning regains some of the lost accuracy points of the model and helps the model achieve higher accuracy on the test set. It is important to keep in mind that the pruning mask must be applied at the end of each fine-tuning epoch to keep the model sparse. This is done using this line in the training loop:

```
pruner.apply(loaded_model)
```

The model is saved into the models folder after fine-tuning.

4. Evaluation

To evaluate the compressed model, I used the accuracy metric on the test dataset. Since my original model from Project 1 achieved a test accuracy of 88.22%, I was trying to keep the test accuracy of the compressed model as close to that as possible while trying to have a compressed model be as small as possible.
Here's the result of KD on the two different student models:

	Small Student	**Student**	**Teacher**
Test Accuracy	77.98%	78.90%	80.22%

After getting this result, I was convinced I should implement pruning only on `StudentModel`. It seemed to be small enough but accurate. Pruning the selected student model results in the final compressed model. Here are the evaluation metrics:

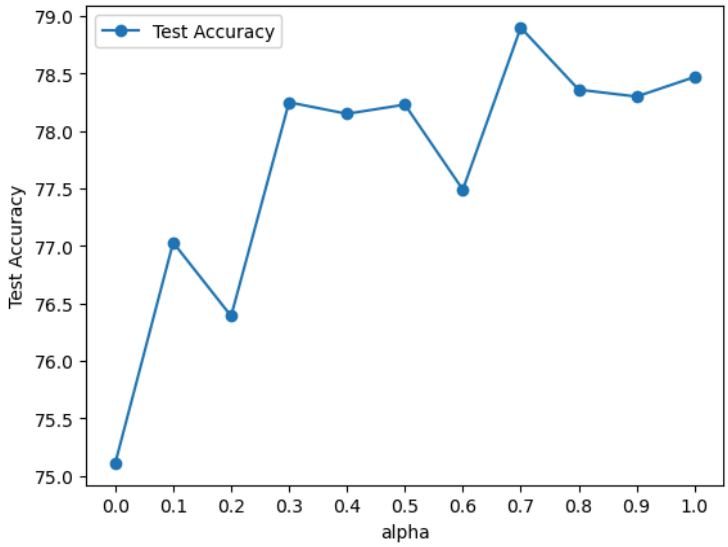
	Original	**Compressed (KD + Pruning)**
# Parameters	553,514	39,206 (93% smaller)
Model Size	2.11 MB	0.15 MB (14.12X smaller)
FLOPs	6,662,144	5,868,032
Training Accuracy	90.87%	85.28%
Test Accuracy	80.22%	77.26% (lost ~3 points)

5. Ablation study

I analyzed the effects of three different hyperparameters on the test accuracy of the compressed model.

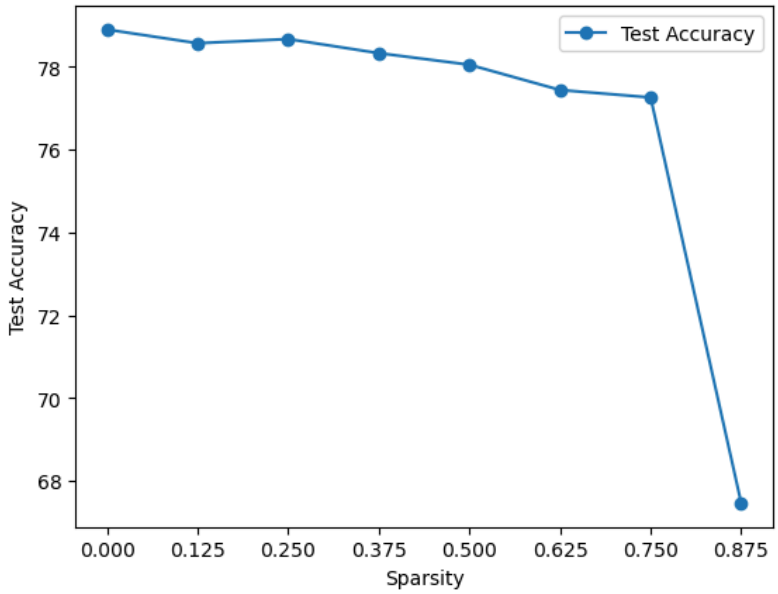
5.1 KD alpha

As explained before, the hyperparameter α controls how much help the student model is getting from the teacher model while calculating its cost function. When $\alpha = 0$ the KL divergence loss doesn't matter at all and the student model does not pay any attention to the outputs of the teacher mode. When $\alpha = 1$ the student model only cares about what the teacher has to say and does not take its own loss into account. As expected, a good value for this hyperparameter is somewhere in between. Here's a plot showing different α values and the corresponding test accuracies for the student model:



5.2 Pruning sparsity

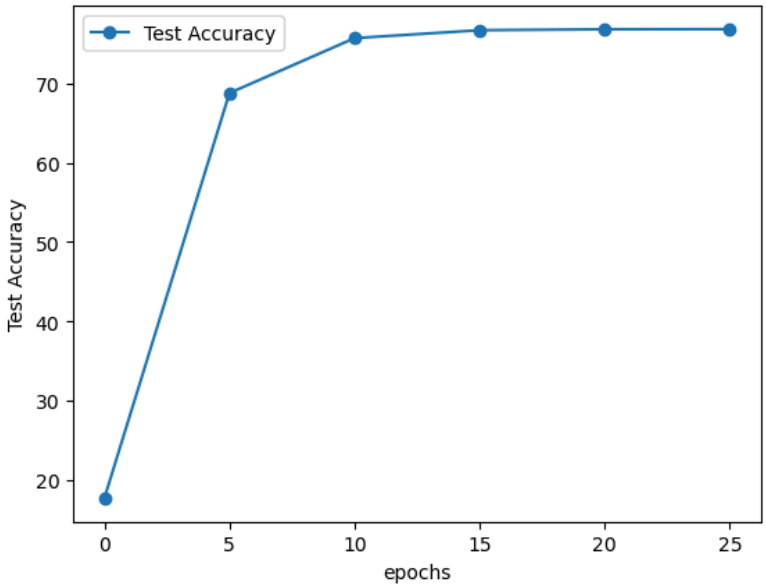
Sparsity is the hyperparameter controlling what percentage of weights will be pruned. If sparsity = 0 it means that no pruning will take place. If sparsity = 1 it means that all the weights will be pruned. Obviously, the higher the sparsity the smaller the model. I wanted to have sparsity as high as possible. I finally decided on a sparsity value of 0.75 (75% of the weights are pruned).



It's evident that sparsity values higher than 0.75 lower the test accuracy too much.

5.3 Fine-tuning epochs

The original model was trained for 60 epochs. Here I was trying to determine how many epochs the fine-tuning after pruning should be.



If epochs = 0 no fine-tuning will take place. If epochs = 20, the pruned model will iterate over the training set 20 times and adjust the weights that have not been pruned.

We can see that after 10 epochs, we are not gaining much accuracy. This is why I decided to keep fine-tuning after pruning limited to 10 epochs.

6. Individual contributions

This was an individual effort since I did not have a teammate for this project. Every part of the project is done by me and all contributions are mine.