CSCI 3485 Lab 4

Rafael Almeida

October 22, 2024

1 Introduction & Methodology

This lab reports on the training time and accuracy of four pre-trained models on the full CIFAR10 dataset. The pre-trained models chosen were ResNet18, ResNet34, VGG11, and VGG13.

For transfer learning, only the models' final linear layer was updated to have its $out\_features$ equal to 10 (the number of classes in CIFAR10) instead of 1000 (the original number of classes). Nothing else was changed in any of the models. Moreover, the following are the parameters for training all the models:

- Epochs: 5
- Learning Rate: 0.001
- Optimizer: Adam
- Loss Function: Cross Entropy Loss
- Data Transformations: only the transformations provided by each model's definition in PyTorch were used
- Batch Size: 32
- Refer to the source code below for any more information

The results include the total number of parameters and the number of trainable parameters, the accuracy of the models ($\frac{correct\_classifications}{total\_data\_points}$), the training time (the execution time of the $Model.train$ method), and the predicted classes of two airplane, frog, and horse images arbitrarily chosen from Google Images.

2 Results

2.1 Main Statistics

Table 1: ResNet18

Total Parameters	11,181,642
Trainable Parameters	5,130
Training Time (s)	620.77
Accuracy	0.7724

Table 2: ResNet34

Total Parameters	21,289,802
Trainable Parameters	5,130
Training Time (s)	700.85
Accuracy	0.7881

Table 3: VGG11

Total Parameters	128,807,306
Trainable Parameters	40,970
Training Time (s)	819.61
Accuracy	0.8212

Table 4: VGG13

Total Parameters	128,991,818
Trainable Parameters	40,970
Training Time (s)	976.45
Accuracy	0.8031

2.2 Google Images

Table 5: Classification Results

Image	ResNet18	ResNet34	VGG11	VGG13
Airplane 1	airplane	airplane	airplane	airplane
Airplane 2	airplane	airplane	airplane	airplane
Frog 1	dog	frog	frog	frog
Frog 2	frog	frog	frog	frog
Horse 1	bird	frog	deer	horse
Horse 2	dog	deer	deer	deer

Figure 1: Airplane 1

Figure 2: Airplane 2

Figure 3: Frog 1

Figure 4: Frog 2

Figure 5: Horse 1

Figure 6: Horse 2

3 Discussion & Conclusion

ResNet performed considerably worse than VGG, even though it is usually considered more performant than the VGG architectures because of its increased depth. This depth comes with better capabilities in complex tasks. However, the structure of this experiment may have been particularly disadvantageous to the ResNet models because their final linear layer has only 5,130 parameters, which were the only ones that were trained. Despite being shallower, the VGG models may have performed better because their final linear layer is much more significant with 40,970 trained parameters. This vast difference in the number of parameters trained in this experiment may cause the VGG architectures to perform better.

Interestingly, when comparing the models to each other, from ResNet18 to 34, there was some improvement, while there was a worsening performance from VGG11 to 13. This may have occurred because ResNet34 has nearly double the number of parameters to 18, which gives it an advantage. However, from VGG11 to 13, there is not a massive change in the number of parameters, but rather, there is an increased depth in the convolutional layers. It then seems reasonable to conclude that this increased depth in convolutions only becomes beneficial with slightly more epochs. It could be that the increased complexity of VGG13 could not be captured in the 40,970 parameters trained because more training time is required for all the deep features learned by each convolution to show up at the final layer trained.

The models were good at classifying the airplane and decent at classifying the frogs. Maybe the frog images I chose excited the cuteness neurons associated with dogs! Furthermore, the models were terrible at classifying the horses, most of the time getting it confused with a deer. This is probably because the images are in very natural environments and not in a farm environment enclosed by fences, for example. The other noticeable thing is that horses and deer have very similar structures. Probably more training time would have been required for the trained parameters to learn to differentiate between those nuances.

4 Code

GitHub Repository

4.1 main.py

```python
"""
Defines the experiments to be run.
"""

from datetime import datetime
from time import time

from torch import cuda, save
from torch.backends import mps
from torchsummary import summary
from torchvision.models import (
    ResNet18_Weights,
    ResNet34_Weights,
    VGG11_Weights,
    VGG13_Weights,
    resnet18,
    resnet34,
    vgg11,
    vgg13,
)

from data import append_to_file, classify_images
from model import Model

if cuda.is_available():
    device = "cuda"
elif mps.is_available():
    device = "mps"
else:
    device = "cpu"

print(f"Using device: {device}")


def time_it(f):
    """
    Times the execution of a function.
    """

    def wrapper(*args, **kwargs):
        start = time()
        result = f(*args, **kwargs)
        return (time() - start, result)
```

```python

    return wrapper


MODELS = {
    "resnet18": resnet18,
    "resnet34": resnet34,
    "vgg11": vgg11,
    "vgg13": vgg13,
}

WEIGHTS = {
    "resnet18": ResNet18_Weights,
    "resnet34": ResNet34_Weights,
    "vgg11": VGG11_Weights,
    "vgg13": VGG13_Weights,
}

for name in MODELS:
    print("my name", name)

    model = Model(
        MODELS[name],
        WEIGHTS[name].IMAGENET1K_V1,
        classes=10,
        batch_size=32,
        data_size=-1,
    )
    training_time, info = time_it(model.train)(
        device=device, max_epochs=5, min_delta=0, lr=1e-3
    )

    # save(model.model, f"./models/{name}.pth")

    accuracy = model.test(device=device)

    file_name = "results.txt"
    append_to_file(file_name, datetime.now())
    append_to_file(file_name, f"{name}")
    append_to_file(file_name, f"param count: {model.get_parameter_count()}")
    append_to_file(file_name, f"time: {training_time}")
    append_to_file(file_name, f"info: {info}")
    append_to_file(file_name, f"accuracy: {accuracy}")

    classify_images(
        model.model,
        WEIGHTS[name].IMAGENET1K_V1.transforms(),
        device,
        "./images",
```

```
    )
```

4.2 model.py

```
"""
Defines a helper Model class to manage the setup, training, and testing
    of all models.
"""

import torch.nn as nn
from torch import no_grad, tensor
from torch.optim import Adam

from data import get_data_loaders


class Model:
    def __init__(
        self,
        model,
        weights,
        classes: int,
        batch_size: int = 32,
        data_size: int = -1,
    ) -> None:
        self.model = None
        self.train_loader = None
        self.test_loader = None

        self.weights = weights
        self.classes = classes
        self.batch_size = batch_size
        self.data_size = data_size
        self.set_model_and_data(model)

    def set_model_and_data(self, model) -> None:
        """Sets the last layer of the model and the data with the
    correction transformation"""

        self.model = model(weights=self.weights)

        # freeze all parameters
        for param in self. model.parameters():
            param.requires_grad = False

        # change last layer
        if hasattr(self.model, "classifier"):
            in_features = self.model.classifier[-1].in_features
```

```
            self.model.classifier[-1] = nn.Linear(in_features, self.classes)

        elif hasattr(self.model, "fc"):
            in_features = self.model.fc.in_features
            self.model.fc = nn.Linear(in_features, self.classes)

        else:
            raise ValueError("Model architecture not supported")

        self.train_loader, self.test_loader = get_data_loaders(
            transforms=[self.weights.transforms()],
            batch_size=self.batch_size,
            size=self.data_size,
        )

    def train(
        self,
        device: str,
        max_epochs: int = None,
        lr: float = 1e-3,
        min_delta: float = 1e-2,
    ) -> dict:
        """
        Trains the model following the given parameters.
        """

        self. model.to(device)
        self. model.train()

        optimizer = Adam(self.model.parameters(), lr=lr)
        loss_fn = nn.CrossEntropyLoss()

        # info to return about the training
        # could include the loss/accuracy per epoch for example
        epochs = 0

        best_loss = float("inf")
        while max_epochs is None or epochs < max_epochs:
            epoch_loss = 0
            # training on each batch
            for x, y in self.train_loader:
                x, y = x.to(device), y.to(device)

                # avoids error:
                # RuntimeError: element 0 of tensors does not require grad and does not have a grad_fn

                optimizer.zero_grad()
                outputs = self.model(x)
```

```
                loss = loss_fn(outputs, y)
                loss.backward()
                optimizer.step()

                epoch_loss += loss.item()

            epochs += 1
            avg_loss = epoch_loss / len(self.train_loader)

            # stop if the loss has not improved enough
            if abs(best_loss - avg_loss) < min_delta:
                break

            best_loss = min(best_loss, avg_loss)

        return {"epochs": epochs}

    @no_grad()
    def test(self, device: str) -> float:
        """
        Tests the accuracy of the model.
        """

        self. model.to(device)
        self. model.eval()
        # maintain the calculation in the gpu until returning
        correct = tensor(0, device=device)
        total = 0

        for x, y in self.test_loader:
            x, y = x.to(device), y.to(device)
            predicted = self.model(x)
            correct += (predicted.argmax(dim=1) == y).sum()
            total += y.size(0)

        accuracy = (correct / total).item()
        return accuracy

    def get_parameter_count(self) -> int:
        """
        Custom function that returns the number of total and trainable parameters in a model like torchvision's summary function
        """
        total_params = sum(p.numel() for p in self.model.parameters())
        trainable_params = sum(
            p.numel() for p in self.model.parameters() if p.requires_grad
        )
        return total_params, trainable_params
```

4.3 data.py

```python
"""
Manages the data for the experiments.
"""

from os import listdir

from torch import Tensor, no_grad
from torch.utils.data import DataLoader, Dataset, Subset
from torchvision.datasets import CIFAR10
from torchvision.io import ImageReadMode, read_image
from torchvision.transforms import Compose


class CIFAR10Dataset(Dataset):
    def __init__(
        self,
        root: str,
        train: bool = True,
        transform: list = [],
        size: int = -1,
    ) -> None:
        self.dataset = CIFAR10(
            root=root, train=train, download=True, transform=Compose(transform)
        )

        if size != -1:
            indices = range(size)
            self.dataset = Subset(self.dataset, indices)

    def __getitem__(self, i) -> tuple[Tensor, Tensor]:
        image, label = self.dataset[i]
        return image, label

    def __len__(self) -> int:
        return len(self.dataset)


def get_data_loaders(
    root: str = "./data/CIFAR10",
    transforms: list = [],
    batch_size: int = 32,
    size: int = -1,
) -> tuple[DataLoader]:
    test_dataset = CIFAR10Dataset(
        root=root, train=False, transform=transforms, size=size
    )
    train_dataset = CIFAR10Dataset(
```

```python
        root=root, train=True, transform=transforms, size=size
    )

    train_loader = DataLoader(
        train_dataset, batch_size=batch_size, shuffle=True
    )
    test_loader = DataLoader(
        test_dataset, batch_size=batch_size, shuffle=False
    )

    return train_loader, test_loader


def append_to_file(filename: str, data: any) -> None:
    with open(filename, "a") as file:
        file.write(str(data) + "\n")


def classify_images(model, transforms, device, path: str):
    """
    Classifies images in a directory using a trained model.
    """
    classes = [
        "airplane",
        "automobile",
        "bird",
        "cat",
        "deer",
        "dog",
        "frog",
        "horse",
        "ship",
        "truck",
    ]

    model.eval()
    model.to(device)

    for image_name in listdir(path):
        image_path = path + "/" + image_name
        # Read image in RGB mode
        image = read_image(image_path, mode=ImageReadMode.RGB)
        image = transforms(image)
        image = image.unsqueeze(0)
        image = image.to(device)

        with no_grad():
            output = model(image)

        append_to_file(
```

```
            "results.txt", f"{image_path} {classes[output.argmax()]}"
        )
```