머신러닝
Image Classification Competition
무한 epoch 수행 보고서

5조 조현우 차민준 최병준 최형우 편수빈 황준우

2023.06.16.

목차

1. 모델, 손실함수 및 스케줄링 기법 소개

1-1. ResNet-200D

딥러닝에서 널리 사용되는 컨볼루션 신경망(CNN) 모델 중 하나입니다. ResNet은 "Residual Network"의 줄임말로, 매우 깊은 신경망을 구성하는 동안 그레디언트 소실 문제를 완화하기 위해 잔차 연결을 도입하였습니다.

ResNet-200D 모델은 ResNet 계열 중에서도 매우 깊은 신경망으로, 총 200개의 층으로 구성되어 있습니다. 이 모델은 얕은 ResNet 모델들과 동일한 잔차 블록 구조를 사용하며, 합성곱 레이어와 배치 정규화 레이어를 포함합니다. 잔차 블록을 반복적으로 쌓아 깊은 네트워크를 구성하고, 마지막에는 평균 풀링과 완전 연결 레이어를 통해 최종 분류를 수행합니다.

해당 모델은 이미지 분류 작업에 있어서 높은 성능을 보여주는 모델로 알려져 있으며, pre-trained된 가중치를 사용하여 더욱 효과적인 학습을 수행할 수 있습니다.

1-2. CrossEntropyLoss

주로 분류 작업에서 사용되는 손실 함수로, 예측값과 실제 타겟값 간의 차이를 계산하여 모델의 오차를 측정하는 데 사용합니다.

일반적으로 CrossEntropyLoss는 다중 클래스 분류 문제에 적용됩니다. 각 클래스에 대해 모델은 확률 분포를 예측하며, 이 확률 분포와 실제 타깃 분포 간의 차이를 계산하여 손실을 구합니다. 해당 손실 함수는 각 클래스에 대한 소프트맥스 함수를 통해 정규화된 예측값과 원-핫 인코딩된 실제 타깃값 사이의 교차 엔트로피를 계산합니다.

CrossEntropyLoss는 기울기 하강법과 같은 최적화 알고리즘을 사용하여 모델의 가중치를 업데이트하는 과정에서 손실 함수로 사용됩니다. 손실 함수의 그래디언트를 계산하고 이를 통해 역전파 알고리즘을 수행합니다. 이러한 과정을 통해 모델의 파라미터가 조정되고, 손실이 최소화되는 방향으로 모델의 학습을 진행합니다.

1-3. StepLR

학습률을 일정한 스텝 간격마다 감소시키는 학습률 스케줄링 기법입니다. 학습률은 경사 하강법 알고리즘에서 각 단계에서 가중치를 업데이트하는 데 사용되는 상수 값으로, 적절한 학습률은 모델의 수렴 속도와 최적 성능에 영향을 미칩니다.

StepLR은 학습률을 이산적으로 감소시키는 방식입니다. 주어진 epoch 또는 반복 횟수가 지나면 미리 정의된 감소 비율을 사용하여 학습률을 줄입니다. 이 스케줄링 기법을 적용하려면, 초기 학습률을 설정하고 특정 epoch 또는 반복 횟수마다 학습률을 감소시키는 스텝 크기와 감소 비율을 정의해야 합니다. 스텝 크기는 학습률을 감소시키는 주기를 결정하고, 감소 비율은 학습률을 얼마나 감소시킬지 결정합니다.

해당 스케줄링 기법은 초기 학습률과 감소 비율을 조정하여 모델의 수렴을 최적화할 수 있습니다. 그러나, 학습률이 너무 높으면 발산할 수 있고, 너무 낮으면 수렴 속도가 느려질 수 있습니다. 따라서 적절한 스텝 크기와 감소 비율을 선택하여 학습률을 조절하는 것이 중요합니다.

1-4. 요약

저희 조는 기본 CNN, DenseNet, EfficientNet, ResNet-50, ResNet-152 등 다양한 모델을 활용하여 세 데이터셋에 대한 분류를 수행해보았으나, 현재를 기점으로 가장 높은 정확도를 도출하는 모델이 일치하여 세 데이터셋 모두 ResNet-200D 모델을 사용하였습니다.

공통적으로 ResNet-200D 모델을 사용하였고, 손실함수로는 CrossEntropyLoss를 사용하였습니다. 스케줄링 기법의 경우 1번 데이터셋과 2번 데이터셋은 StepLR 기법을, 3번 데이터셋의 경우에는 5 에포크까지 학습률을 유지하고, 6 에포크부터 학습률을 줄이는 기법을 사용하여 이미지 분류를 수행하였습니다.

2. Cifar-100

2-1. 개요

ResNet-200D 모델을 활용하여 Cifar-100 데이터셋을 학습하고 평가하였습니다.
Cifar-100 데이터셋은 100개의 클래스를 가진 이미지 분류 데이터셋입니다. 각 클래스에는 600개의 이미지가 있으며, 500개는 학습용, 100개는 테스트용입니다. 100개의 클래스는 20개의 '슈퍼클래스'로 그룹화되며, 작은 사이즈와 다양한 클래스로 인해 빠른 테스트와 비교에 적합하지만, 이미지의 해상도가 낮고 클래스 간 변별력이 적을 수 있습니다.

2-2. 요약

1. 필요한 라이브러리와 모듈을 임포트합니다.
2. 데이터 변환을 합니다.
3. Cifar-100 데이터셋을 다운로드하고 학습 및 테스트 데이터셋으로 분할합니다. (비율 5:1) 이때, 다양한 변환 방식을 사용하여 train 데이터를 증대합니다.
4. 데이터 로더(DataLoader)를 사용하여 학습 데이터셋과 테스트 데이터셋을 배치 단위로 제공합니다.
5. ResNet-200D 모델을 생성하고, 전이학습을 위해 미리 학습된 가중치를 로드합니다.
6. 모델의 출력 레이어를 변경하여 클래스의 개수에 맞게 설정합니다.
7. 손실 함수와 학습률 스케줄러를 StepLR를 통해 구현합니다. 5 에폭마다 학습률을 0.1배 하였습니다.
8. 지정된 에폭 수에 따라 학습을 진행합니다. 매 에폭마다 학습 데이터를 사용하여 순전파, 손실 계산, 역전파, 그리고 최적화를 수행합니다.
9. 학습이 완료된 모델을 사용하여 테스트 데이터셋에 대한 성능을 평가합니다. 정확도를 계산하여 출력합니다.

코드 실행 시, Cifar-100 데이터셋을 다운로드하고 모델을 학습하며, 각 에폭(epoch)마다 Test Accurarcy를 출력합니다. 학습이 완료되면 테스트 데이터셋을 사용하여 모델의 최종 정확도를 계산하여 출력합니다.

2-3. 코드 설명

2-3-1) 필요한 라이브러리 및 모듈 임포트

```
import torch
import torchvision
import torchvision.transforms as transforms
import torch.optim as optim
import torch.nn as nn
import time
from torch.utils.data import DataLoader, random_split
from timm.models.resnet import resnet200d
```

torch : PyTorch 라이브러리
torchvision : PyTorch의 컴퓨터 비전 모델과 데이터셋 등을 포함한 패키지
transforms : 데이터 전처리를 위한 변환 함수들이 포함된 모듈
optim : 최적화 알고리즘을 제공하는 모듈
nn : 신경망 모델을 구성하는 데 필요한 클래스들이 포함된 모듈
time : 시간 측정을 위한 모듈
DataLoader, random_split : 데이터를 불러오고 분할하는 데 필요한 클래스들

2-4-2) 장치 설정

```
# Device 설정
device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
```

GPU를 사용할 수 있는 경우 "cuda:0"을, 사용할 수 없는 경우 "cpu"를 장치로 설정

2-4-3) 데이터 변환

```
# 데이터 변환
transform = transforms.Compose([
    transforms.Resize((224, 224)),
    transforms.ToTensor(),
    transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5)),
])
```

입력 데이터에 대한 변환 방식을 정의. 여기에서는 이미지의 크기를 224x224로 조정하고, 텐서로 변환한 뒤 정규화를 수행.

2-4-4) Cifar-100 데이터셋 로드

```
# CIFAR-100 데이터셋 로드
cifar100 = torchvision.datasets.CIFAR100(root='./data', train=True, download=True, transform=transform)
testset = torchvision.datasets.CIFAR100(root='./data', train=False,
                                        download=True, transform=transform)
```

torchvision의 datasets.CIFAR100 클래스를 사용하여 CIFAR-100 데이터셋을 로드. root는 데이터셋을 저장할 디렉토리를 나타내며, train은 훈련 데이터셋 여부를 나타냄. 테스트 데이터셋도 동일한 방식으로 로드.

2-4-5) 데이터 증대 설정

```
def augmentation_random_rotation(degrees):
    return transforms.RandomRotation(degrees=degrees)

def augmentation_color_jitter(brightness=0, contrast=0, saturation=0, hue=0):
    return transforms.ColorJitter(brightness=brightness, contrast=contrast, saturation=saturation, hue=hue)
```

augmentation_random_rotation, augmentation_color_jitter 함수를 정의하여 데이터 증대에 사용할 변환 방식들을 구성

```
# 데이터 증대 설정
augmentation_crop = transforms.Compose([
    transforms.Resize((224, 224)),
    transforms.RandomCrop((224, 224)),
    transforms.GaussianBlur(kernel_size=3),
    transforms.ToTensor(),
    transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5)),
])
```

```
augmentation_all = transforms.Compose([
    transforms.Resize((224, 224)),
    transforms.RandomCrop((224, 224)),
    transforms.RandomHorizontalFlip(),
    transforms.GaussianBlur(kernel_size=3),
    augmentation_random_rotation(degrees=30),
    augmentation_color_jitter(brightness=0.4, contrast=0.4, saturation=0.4, hue=0.1),
    transforms.RandomVerticalFlip(),
    transforms.ToTensor(),
    transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5)),
])
```

augmentation_crop, augmentation_all은 데이터를 증대하기 위해 여러 변환 방식들을 조합

+) augmentation_all에서 사용한 기법들

1) transforms.Resize((224, 224))
: 이미지의 크기를 224x224로 조정. 일반적으로 이미지 크기를 통일하여 모델에 입력하기 위해 사용.

2) transforms.RandomCrop((224, 224))
: 이미지를 무작위로 자르거나 크롭. 이를 통해 이미지 내의 다양한 부분을 학습.

3) transforms.RandomHorizontalFlip()
: 이미지를 무작위로 수평으로 뒤집음. 좌우 대칭된 이미지가 증가하여 데이터의 다양성을 높이는 효과를 가질 수 있음.

4) transforms.GaussianBlur(kernel_size=3)
: 가우시안 블러를 적용하여 이미지를 흐리게 만듬. 이는 이미지 내의 노이즈를 감소시키고 모델의 일반화 성능을 향상시킬 수 있음.

5) augmentation_random_rotation(degrees=30)
: 무작위로 이미지를 회전시킴. degrees 매개변수로 지정한 각도 범위 내에서 랜덤하게 회전이 적용. 이미지를 다양한 각도로 변형하여 모델이 회전에 불변한 특성을 학습할 수 있음.

6) augmentation_color_jitter(brightness=0.4, contrast=0.4, saturation=0.4, hue=0.1)
: 이미지의 밝기, 대비, 채도, 색상을 랜덤하게 조정함. 이는 조명 조건이나 색깔 변화에 더 강건한 모델을 학습할 수 있게 함.

7) transforms.RandomVerticalFlip()
: 이미지를 무작위로 수직으로 뒤집음. 상하 대칭된 이미지가 증가하여 데이터의 다양성을 높이는 효과를 가질 수 있음.

8) transforms.ToTensor()
: 이미지를 텐서 형태로 변환. 모델의 입력으로 사용하기 위해 이미지를 숫자로 표현하는 텐서 형태로 변환.

9) transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))
: 이미지를 정규화. 입력 이미지의 픽셀 값을 정규화하여 모델의 학습을 안정화시키고 수렴 속도를 향상시킬 수 있음. 이 때 사용된 평균과 표준편차는 CIFAR-100 데이터셋을 기반으로 계산된 값.

2-4-6) 데이터 분할

```
# 데이터를 train, test로 나누기
train_size = int(0.83 * len(cifar100))
test_size = len(cifar100) - train_size
train_dataset, test_dataset = random_split(cifar100, [train_size, test_size])
```

random_split 함수를 사용하여 전체 데이터셋을 훈련 데이터셋과 테스트 데이터셋으로 나눔. 비율은 5:1을 맞춤

2-4-7) 훈련 데이터 증대

```
# 훈련 데이터에 증대된 데이터 추가
#augmented_dataset1 = train_dataset
augmented_dataset2 = train_dataset
augmented_dataset3 = train_dataset
#augmented_dataset4 = train_dataset
#augmented_dataset5 = train_dataset

augmented_dataset2.dataset.transfrom = augmentation_crop
train_dataset += augmented_dataset2

augmented_dataset3.dataset.transfrom = augmentation_all
train_dataset += augmented_dataset3

#augmented_dataset4.dataset.transfrom = augmentation_flip
#train_dataset += augmented_dataset4

#augmented_dataset5.dataset.transfrom = augmentation_rotation
#train_dataset += augmented_dataset5
```

augmented_dataset2, augmented_dataset3를 생성하여 원래 훈련 데이터셋에 추가.
augmented_dataset2는 augmentation_crop을 사용하여 데이터를 증대.
augmented_dataset3는 augmentation_all을 사용하여 데이터를 증대.

2-4-8) 데이터 로더 생성

```
# 데이터 로더 생성
trainloader = DataLoader(train_dataset, batch_size=16, shuffle=True, num_workers=2)
testloader = DataLoader(test_dataset, batch_size=16, shuffle=False, num_workers=2)
```

DataLoader 클래스를 사용하여 훈련 데이터셋과 테스트 데이터셋에 대한 데이터 로더를 생성.
배치 크기는 16이며, 데이터를 섞고 2개의 워커를 사용하여 데이터를 로드.

2-4-9) 모델 생성

```
# 모델 생성
model = resnet200d(pretrained=True)
num_ftrs = model.fc.in_features
model.fc = nn.Linear(num_ftrs, 100)

model = model.to(device)
```

timm.models.resnet 모듈에서 resnet200d 모델을 가져옴. 이 모델은 ResNet-200 아키텍처의 사전 훈련된 가중치를 사용. 모델의 fully connected 레이어를 CIFAR-100 클래스 수에 맞게 변경.

2-4-10) 손실 함수, optimizer 설정

```
# 손실 함수, Optimizer 설정
criterion = nn.CrossEntropyLoss()
optimizer = optim.SGD(model.parameters(), lr=0.01, momentum=0.9)
scheduler = optim.lr_scheduler.StepLR(optimizer, step_size=5, gamma=0.1)
```

nn.CrossEntropyLoss를 사용하여 분류 작업에 대한 손실 함수를 정의.
optim.SGD를 사용하여 SGD (Stochastic Gradient Descent) 옵티마이저를 설정.
optim.lr_scheduler.StepLR을 사용하여 학습률을 업데이트할 스케줄러를 설정.

2-4-11) 학습 및 평가

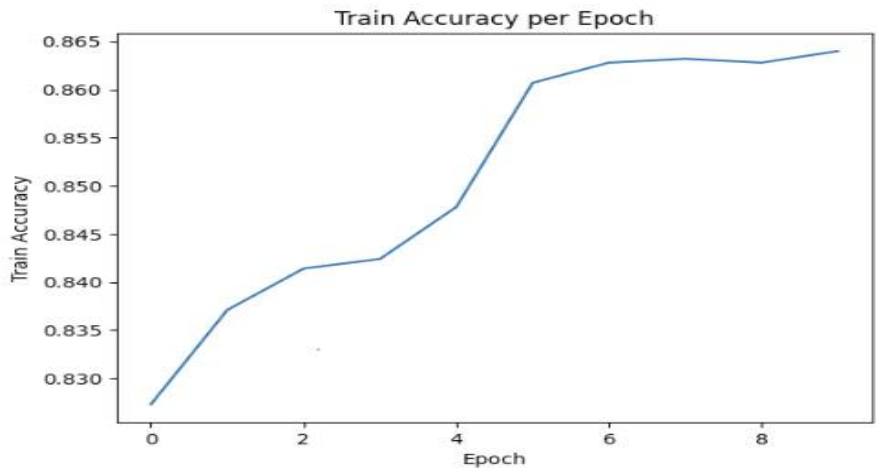
1) start_time을 기준으로 학습 시작 시간을 저장.
2) trainloader를 사용하여 에폭(epoch)마다 훈련을 수행.
3) 에폭 내에서 배치 단위로 데이터를 가져와 모델을 학습시키고, 손실을 계산하여 모델을 업데이트.
4) 매 100번째 배치마다 손실을 출력.
5) 에폭이 끝난 후 테스트 데이터셋을 사용하여 모델의 정확도를 평가. 학습률을 업데이트.

2-4. 결과

```
import matplotlib.pyplot as plt

train_acc_list = [0.8273,0.8371,0.8414,0.8424,0.8478,0.8607,0.8628,0.8632,0.8628,0.8640]

# Plotting the train accuracy values
plt.plot(train_acc_list)
plt.xlabel('Epoch')
plt.ylabel('Train Accuracy')
plt.title('Train Accuracy per Epoch')
plt.show()
```



train_acc_list로 각 에폭별로 기록된 훈련 정확도를 입력하였고 각 훈력 정확도를 추적하여 그래프로 시각화함. 그래프를 통해 에폭이 증가함에 따라 훈련 정확도가 어떻게 변화하는지 확인할 수 있었음

최종결과 : 0.8640

```
테스트 데이터셋 정확도: 0.8640
학습 완료. 소요 시간: 13607.11초
테스트 데이터셋 정확도: 0.8640
```

2-5. 전체 코드

```
import torch
import torchvision
import torchvision.transforms as transforms
import torch.optim as optim
import torch.nn as nn
import time
from torch.utils.data import DataLoader, random_split
from timm.models.resnet import resnet200d

# 랜덤 시드 설정
torch.manual_seed(42)

# Device 설정
device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")

# 데이터 변환
transform = transforms.Compose([
    transforms.Resize((224, 224)),
    transforms.ToTensor(),
    transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5)),
])

# CIFAR-100 데이터셋 로드
cifar100 = torchvision.datasets.CIFAR100(root='./data', train=True, download=True, transform=transform)
testset = torchvision.datasets.CIFAR100(root='./data', train=False,
                                        download=True, transform=transform)



def augmentation_random_rotation(degrees):
    return transforms.RandomRotation(degrees=degrees)

def augmentation_color_jitter(brightness=0, contrast=0, saturation=0, hue=0):
    return transforms.ColorJitter(brightness=brightness, contrast=contrast, saturation=saturation, hue=hue)
```

```
# 데이터 증대 설정
augmentation_crop = transforms.Compose([
    transforms.Resize((224, 224)),
    transforms.RandomCrop((224, 224)),
    transforms.GaussianBlur(kernel_size=3),
    transforms.ToTensor(),
    transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5)),
])

augmentation_rotation = transforms.Compose([
    transforms.Resize((224, 224)),
    transforms.RandomCrop((224, 224)),
    augmentation_random_rotation(degrees=30),
    augmentation_color_jitter(brightness=0.4, contrast=0.4, saturation=0.4, hue=0.1),
    transforms.ToTensor(),
    transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5)),
])


augmentation_flip = transforms.Compose([
    transforms.Resize((224, 224)),
    transforms.RandomCrop((224, 224)),
    transforms.RandomHorizontalFlip(),
    transforms.RandomVerticalFlip(),
    transforms.ToTensor(),
    transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5)),
])

augmentation_all = transforms.Compose([
    transforms.Resize((224, 224)),
    transforms.RandomCrop((224, 224)),
    transforms.RandomHorizontalFlip(),
    transforms.GaussianBlur(kernel_size=3),
    augmentation_random_rotation(degrees=30),
    augmentation_color_jitter(brightness=0.4, contrast=0.4, saturation=0.4, hue=0.1),
    transforms.RandomVerticalFlip(),
    transforms.ToTensor(),
    transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5)),
])
```

```
# 데이터를 train, test로 나누기
train_size = int(0.83 * len(cifar100))
test_size = len(cifar100) - train_size
train_dataset, test_dataset = random_split(cifar100, [train_size, test_size])



# 훈련 데이터에 증대된 데이터 추가
#augmented_dataset1 = train_dataset
augmented_dataset2 = train_dataset
augmented_dataset3 = train_dataset
#augmented_dataset4 = train_dataset
#augmented_dataset5 = train_dataset

augmented_dataset2.dataset.transfrom = augmentation_crop
train_dataset += augmented_dataset2

augmented_dataset3.dataset.transfrom = augmentation_all
train_dataset += augmented_dataset3

#augmented_dataset4.dataset.transfrom = augmentation_flip
#train_dataset += augmented_dataset4

#augmented_dataset5.dataset.transfrom = augmentation_rotation
#train_dataset += augmented_dataset5


# 데이터 로더 생성
trainloader = DataLoader(train_dataset, batch_size=16, shuffle=True, num_workers=2)
testloader = DataLoader(test_dataset, batch_size=16, shuffle=False, num_workers=2)

# 모델 생성
model = resnet200d(pretrained=True)
num_ftrs = model.fc.in_features
model.fc = nn.Linear(num_ftrs, 100)

model = model.to(device)

# 손실 함수, Optimizer 설정
criterion = nn.CrossEntropyLoss()
```

```
optimizer = optim.SGD(model.parameters(), lr=0.01, momentum=0.9)
scheduler = optim.lr_scheduler.StepLR(optimizer, step_size=5, gamma=0.1)

# 학습 및 평가
start_time = time.time()

print(f"학습 데이터셋 크기: {len(trainloader.dataset)}")
print(f"테스트 데이터셋 크기: {len(testloader.dataset)}")

for epoch in range(10):  # 10 에폭으로 학습
    print(f"Epoch {epoch + 1} 시작")
    running_loss = 0.0

    for i, data in enumerate(trainloader):
        inputs, labels = data[0].to(device), data[1].to(device)

        optimizer.zero_grad()

        outputs = model(inputs)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()

        running_loss += loss.item()
        if (i + 1) % 100 == 0:
            print(f"Epoch [{epoch + 1}, {i + 1}] loss: {running_loss / 100:.3f}")
            running_loss = 0.0

    # 테스트 데이터셋에서 평가
    correct = 0
    total = 0

    with torch.no_grad():
        for data in testloader:
            images, labels = data[0].to(device), data[1].to(device)
            outputs = model(images)
            _, predicted = torch.max(outputs.data, 1)
            total += labels.size(0)
            correct += (predicted == labels).sum().item()

    test_acc = correct / total
```

```
        print(f"테스트 데이터셋 정확도: {test_acc:.4f}")

        # 학습률 업데이트
        scheduler.step()

print(f"학습 완료. 소요 시간: {time.time() - start_time:.2f}초")

# 테스트 데이터셋에서 평가
correct = 0
total = 0

with torch.no_grad():
    for data in testloader:
        images, labels = data[0].to(device), data[1].to(device)
        outputs = model(images)
        _, predicted = torch.max(outputs.data, 1)
        total += labels.size(0)
        correct += (predicted == labels).sum().item()

test_acc = correct / total
print(f"테스트 데이터셋 정확도: {test_acc:.4f}")

import matplotlib.pyplot as plt

train_acc_list =
[0.8273,0.8371,0.8414,0.8424,0.8478,0.8607,0.8628,0.8632,0.8628,0.8640]

# Plotting the train accuracy values
plt.plot(train_acc_list)
plt.xlabel('Epoch')
plt.ylabel('Train Accuracy')
plt.title('Train Accuracy per Epoch')
plt.show()
```

3. Standford Dogs

3-1. 개요

ResNet-200D 모델을 활용하여 Stanford Dogs 데이터셋을 학습하고 평가하였습니다.
Stanford Dogs 데이터셋은 120개의 강아지 종을 대표하는 20,580개의 이미지로 구성되어 있습니다. 각 이미지는 특정 강아지 종을 표현하며, 강아지의 위치를 나타내는 바운딩 박스 주석이 포함되어 있습니다. 해당 데이터셋은 ImageNet에서 파생되었으며, 세부적인 카테고리 내에서의 분류 문제를 해결하는데 유용합니다.

3-2. 요약

1. 필요한 라이브러리와 모듈을 임포트합니다.
2. Stanford dog 데이터셋을 다운로드하고 학습 및 테스트 데이터셋으로 분할하고 증대합니다.
3. 데이터 로더를 사용하여 학습 데이터셋과 테스트 데이터셋을 배치 단위로 제공합니다.
4. ResNet-200D 모델을 생성하고, 전이학습을 위해 미리 학습된 가중치를 로드합니다.
5. ResNet-200D 모델을 응용하여 SpinalNet_ResNet 모델을 생성합니다..
6. 손실 함수와 최적화 기법, 학습률 스케줄러를 구현합니다.
7. 지정된 에폭 수에 따라 학습을 진행합니다. 매 에폭마다 학습 데이터를 사용하여 순전파, 손실 계산, 역전파, 그리고 최적화를 수행합니다.
8. 학습이 완료된 모델을 사용하여 테스트 데이터셋에 대한 성능을 평가합니다. 정확도를 계산하여 출력합니다.

코드 실행 시, Stanford dog 데이터셋을 다운로드하고 모델을 학습하며, 각 에폭(epoch)마다 Train Accuracy와 Valid Accurarcy를 출력합니다. 최종적으로 학습이 완료되면 테스트 데이터셋을 사용하여 모델의 최종 정확도를 계산하여 출력하였습니다.

3-3. 코드 설명

3-3-1) 필요한 라이브러리 및 모듈 임포트

필요한 라이브러리와 모듈을 임포트합니다. torchvision, torch.optim, torch.nn 등의 라이브러리를 사용하였습니다. ResNet-200D 모델을 사용하기 위해 `resnet200d`를 임포트합니다.

3-3-2) 데이터 전처리

```
transform = transforms.Compose([
    transforms.Resize((224, 224)),
    transforms.ToTensor(),
    transforms.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225])
])
```

데이터 전처리를 위한 변환입니다. `ToTensor()`를 사용하여 이미지를 텐서로 변환하고, `Resize()`를 사용하여 원본 이미지 크기를 224x224로 조정합니다. 또한, 정규화를 위해 `Normalize()`를 사용하여 입력 데이터를 정규화하였습니다.

3-3-3) Stanford dog 데이터셋 로드 및 분할

```
extracted_dir = os.path.join(download_dir, "Images")
original_dataset = torchvision.datasets.ImageFolder(extracted_dir, transform=transform)
```

Stanford dog 데이터셋을 다운로드하고, 위에서 정의한 전처리를 적용하여 'original_dataset' 변수에 저장하였습니다.

```
train_dataset, test_dataset = torch.utils.data.random_split(dataset, [int(0.8 * len(dataset)), int(0.2 * len(dataset))])
```

random_split 함수를 사용하여 원본 데이터셋을 4:1 비율로 train, test 데이터로 나눕니다.

3-3-4) 데이터 증대

```
augmentation_crop = transforms.Compose([
    transforms.Resize((256, 256)),
    transforms.RandomCrop((224, 224)),
    transforms.ToTensor(),
    transforms.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225])
])
```

데이터 증대에 사용할 변환을 정의하였습니다. 전체 변환은 총 5가지로, 공통적으로 이미지 크기를 224x224로 조정하고 텐서로 변환한 뒤, 정규화를 수행하였습니다. 각 변환의 특징은 다음과 같습니다.

- augmentation_crop: 이미지를 256x256으로 크롭
- augmentation_rotation: 이미지를 임의의 각도로 회전
- augmentation_jitter: 이미지의 색상을 무작위로 변화
- augmentation_flip: 이미지를 무작위로 수평 및 수직으로 뒤집음
- augmentation_blur: 이미지에 가우시안 블러를 적용

3-3-5) 데이터셋 확장

```
augmented_dataset1 = train_dataset
augmented_dataset1.dataset.transfrom = augmentation_blur
train_dataset += augmented_dataset1
```

위에서 정의한 변환 방법을 사용하여 훈련 데이터셋을 증대시킵니다. 원본 훈련 데이터셋과 같은 데이터셋 형태를 유지하면서 증대된 데이터셋을 생성하고, 변환을 적용하여 훈련 데이터에 추가하였습니다. 증대를 할때 loss가 너무 빨리 닳는 점을 해결하기 위해서 회전, crop, rotation등 하나 하나 증대해서 더해주는 방식에서 crop+rotation , crop + blur ,여러가지를 모두 증대하는 방식 등으로 데이터 셋을 조금 더 복잡하게 만들었습니다.

3-3-6) 데이터 로더 생성

```
train_loader = torch.utils.data.DataLoader(train_dataset, batch_size=16, shuffle=True, num_workers=2)
test_loader = torch.utils.data.DataLoader(test_dataset, batch_size=16, shuffle=False, num_workers=2)
```

데이터 로더를 사용하여 훈련 데이터셋과 테스트 데이터셋에 대한 데이터 로더를 생성합니다. 각 데이터 로더는 배치 크기를 16로 설정하고, 훈련 데이터는 섞어서 순서를 무작위로 사용하도록 shuffle=True로 설정하였습니다.

3-3-7) SpinalNet_ResNet 모델 생성

```
model = resnet200d(pretrained=True)
```

ResNet-200D 모델을 생성하고, 미리 학습된 가중치를 로드하였습니다.

```
class SpinalNet_ResNet(nn.Module)
def forward(self, x)
```

ResNet-200d 모델을 기반으로 한 SpinalNet_ResNet 모델을 생성하였습니다. SpinalNet 아키텍처를 사용하여 ResNet-200d 모델을 확장하였으며, 해당 클래스는 네 개의 Spinal 레이어와 마지막 분류 레이어로 구성되어 있습니다. forward는 입력을 네 개의 Spinal 레이어를 통과시키고, 각 레이어의 출력을 연결하여 최종 특성을 생성하는 메서드입니다. SpinalNet_ResNet에서는 이 메서드를 마지막 분류 레이어에 통과시켜 예측값을 반환합니다.

```
model.fc = SpinalNet_ResNet()
model = model.to(device)
```

ResNet-200d 모델을 기반으로 하여 주어진 데이터셋에 더욱 적합한 SpinalNet_ResNet 모델을 생성하였으며, 해당 모델을 사용하여 Stanford Dogs 데이터셋을 분류하였습니다.

3-3-8) 손실 함수와 옵티마이저 설정

```
criterion = nn.CrossEntropyLoss()
optimizer = optim.AdamW(model.parameters(), lr=0.00001, weight_decay=1e-5)
scheduler = lr_scheduler.OneCycleLR(optimizer, max_lr=0.0001, epochs=25, steps_per_epoch=len(train_loader))
```

손실함수로 다중 클래스 분류 작업에 사용되는 CrossEntropyLoss를, 최적화기로는 AdamW를 사용하였습니다. 학습률은 0.00001이고 가중치 감쇠는 1e-5로 설정하였습니다. 학습률 스케줄러로는 lr_scheduler.OneCycleLR을 사용하였습니다. 해당 스케줄러는 학습률을 한 사이클 동안 선형적으로 증가하고 감소시킵니다. 최대 학습률은 0.0001로 설정하였고, 에폭 수와 train_loader의 배치 수에 기반하여 스케줄링됩니다.

개선점)

기존의 SGD 옵티마이저 대신 AdamW를 선택했습니다. AdamW는 Adam의 변형 버전으로, 가중치 감쇠(weight decay)를 적용하는 방식이 다릅니다. loss가 너무 빨리 닳는 이전의 단점을 해결하기 위해서 사용하였습니다. 이때 optim.AdamW()를 사용하여 AdamW 옵티마이저를 초기화하였습니다. model.parameters()는 모델의 학습 가능한 매개변수를 전달하는 역할을 합니다.

또한 문제점이였던 너무 빠른 수렴을 막기 위해서 lr=0.00001로 AdamW 옵티마이저의 학습률을 설정했습니다. weight_decay=1e-5로 AdamW 옵티마이저의 가중치 감쇠를 설정했습니다. 가중치 감쇠는 과적합을 방지하기 위해 가중치 갱신 시에 손실 함수에 대한 정규화 항으로 추가되는 역할을 합니다.

또한 validation loss가 너무 빨리 주는것을 방지하기 위해서lr_scheduler.OneCycleLR()을 사용하여 스케줄러를 초기화했습니다. OneCycleLR은 주어진 epoch 수와 학습 단계 수에 따라 학습률을 동적으로 조절하는 방식입니다. max_lr=0.0001로 학습률의 최대값을 설정하여 너무빨리

loss가 줄어들어 과적합이 나는 것을 방지했으며, epochs=30으로 전체 학습 과정을 30번 반복하도록 설정했습니다.

이렇게 변경된 부분은 AdamW 옵티마이저를 사용하고 있으며, 가중치 감쇠와 학습률 스케줄링을 적용하여 모델의 학습을 조절하고 특히 validation loss의 빠른 수렴 문제를 개선하는 데 목적이 있습니다.

3-3-9) 학습 및 가중치 저장

```
for epoch in range(epochs)
```

무한 에폭으로 학습이 가능하지만, 시간 관계상 30 에폭에 걸쳐 학습을 수행해 보았습니다. 학습 중에는 각 미니배치마다 순전파, 역전파, 옵티마이저를 실행합니다. 출력과 레이블 간의 손실을 계산하고, 역전파를 수행하여 모델의 매개변수를 업데이트합니다. 이때 매 에폭마다 train accuracy와 valid accuracy를 출력합니다.

3-3-10) 모델 평가

테스트 데이터셋을 사용하여 모델의 성능을 평가하였습니다. test_loader를 순회하면서 각 배치의 입력을 모델에 전달하고, 모델의 출력을 예측값으로 변환하여 정확도를 계산합니다. 최종적으로 전체 테스트 데이터셋에 대한 맞은 개수와 총 개수를 출력하고, 모델의 테스트 정확도를 출력하였습니다.

+) 최고 가중치 추가하는 부분

```
if valid_acc > max_accuracy:
    max_accuracy = valid_acc
    max_weight = model.state_dict()
    # 최고 가중치 저장
    torch.save(max_weight, "max_weight.pth")
```

에폭을 수행하면서, 최고 정확도를 갖는 경우에 모델 가중치를 저장합니다. 가장 높은 정확도를 갖는 모델 가중치는 "max_weight.pth" 파일에 저장하였습니다.

```
weight = torch.load("max_weights.pth")
model.load_state_dict(weight)
```

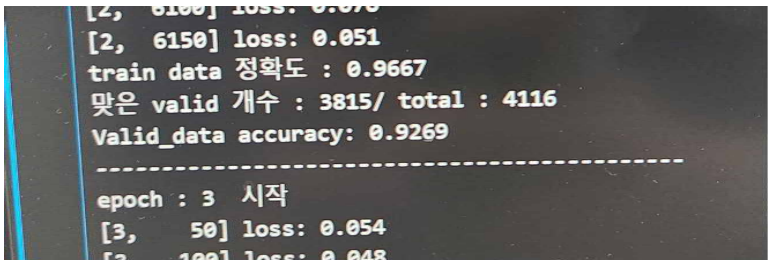
저장된 가중치 파일인 "max_weights.pth"에서 모델의 가중치를 로드합니다. torch.load() 함수를 사용하여 파일로부터 가중치를 읽어옵니다. 읽어온 가중치는 weight 변수에 저장하고, model.load_state_dict() 함수를 사용하여 모델의 상태 사전에 로드한 가중치를 설정하였습니다. 해당 코드로 모델의 가중치를 최적 가중치로 갱신하였습니다.

3-4. 결과

3-4-1) 최종 정확도 결과

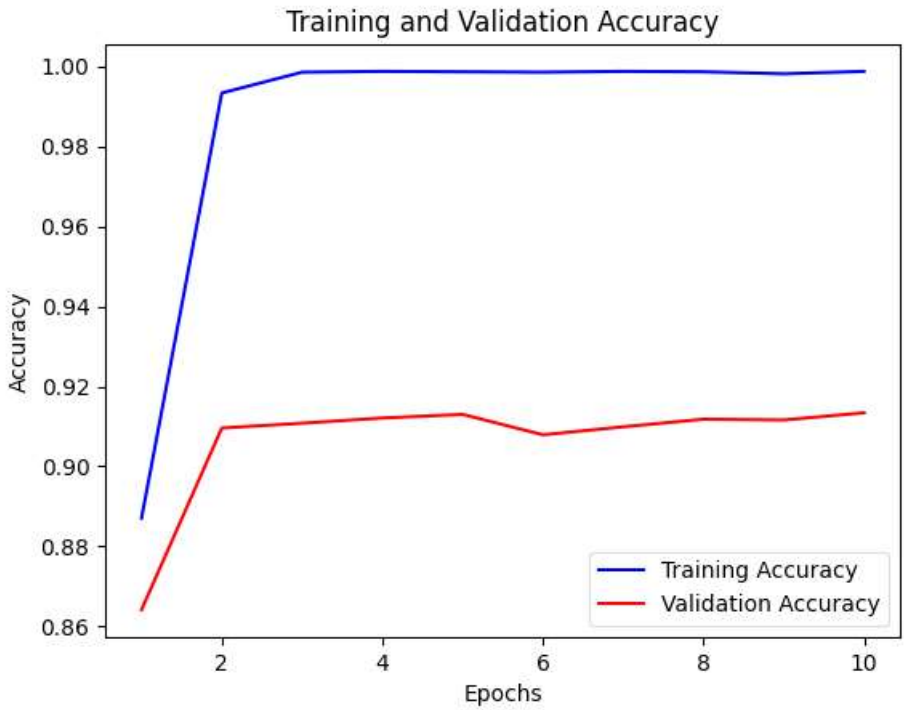
```
----------------------------------------
evaluation mode
맞은 데이터 개수 : 3753/ total : 4116
Test accuracy: 0.9118
----------------------------------------
```

학습이 완료된 후, 테스트 데이터셋에 대한 성능을 평가하였습니다. 정확도 0.9118의 결과를 얻었습니다.



앞서 모델의 최고 가중치를 적용하는 부분을 추가하였지만, 그래프를 그리는 과정에서 오류가 생겨 30 에포크를 전부 수행하지 못해 최종 정확도를 도출하지 못하였습니다. 하지만, 2 에포크만에 위의 정확도를 능가하는 모습을 보였으며 해당 코드로 다시 성능을 평가할 경우에 이전보다 더 높은 성능을 보일 것으로 예상됩니다.

3-4-2) train, valid 정확도 그래프



3-5. 전체 코드

```
%pip install timm

import os
import urllib.request
import tarfile
import random
from sklearn.model_selection import train_test_split
import torch
import torch.nn as nn
import torch.optim as optim
import torchvision
import torchvision.transforms as transforms
from torchvision.models import resnet50
from torchvision.models import resnet152
from timm.models.resnet import resnet200d
from sklearn.metrics import accuracy_score
from torch.optim.lr_scheduler import CosineAnnealingLR
from torch.optim.lr_scheduler import ReduceLROnPlateau
import torch
import torchvision
import torchvision.transforms as transforms
import torch.optim as optim
import torch.nn as nn
import time
from torch. utils.data import Dataset,DataLoader, random_split
from timm.models.resnet import resnet200d
from torchvision import models
from torch.optim import lr_scheduler


# 데이터셋 다운로드할 폴더 경로 지정
download_dir = "./data/stanford_dogs"

# 폴더 없으면 생성
if not os.path.exists(download_dir):
    os.makedirs(download_dir)

# 데이터셋 다운로드
url = "http://vision.stanford.edu/aditya86/ImageNetDogs/images.tar"
filename = os.path.join(download_dir, "images.tar")
```

```
urllib.request.urlretrieve(url, filename)

# 데이터셋 추출
tar = tarfile.open(filename, "r")
tar.extractall(download_dir)
tar.close()

# 압축 파일 제거
os.remove(filename)

# 추출한 데이터셋의 경로
extracted_dir = os.path.join(download_dir, "Images")

# 랜덤 시드 설정
torch.manual_seed(42)

# 디바이스 설정(사용 가능한 경우 GPU, 그렇지 않은 경우 CPU)
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")

# Stanford Dogs 데이터 세트의 클래스 수 설정
num_classes = 120

# 데이터 변환

# 데이터 변환
transform = transforms.Compose([
    transforms.Resize((224, 224)),
    transforms.ToTensor(),
    transforms.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225])
])

# 데이터셋 형성
original_dataset = torchvision.datasets.ImageFolder(extracted_dir, transform=transform)


def augmentation_random_rotation(degrees):
    return transforms.RandomRotation(degrees=degrees)

def augmentation_color_jitter(brightness=0, contrast=0, saturation=0, hue=0):
    return transforms.ColorJitter(brightness=brightness, contrast=contrast,
saturation=saturation, hue=hue)
```

```
# 데이터 증대 설정
augmentation_crop = transforms.Compose([
    transforms.Resize((256, 256)),
    transforms.RandomCrop((224, 224)),
    transforms.ToTensor(),
    transforms.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225])
])

augmentation_rotation = transforms.Compose([
    transforms.Resize((224, 224)),
    augmentation_random_rotation(degrees=30),
    transforms.ToTensor(),
    transforms.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225])
])

augmentation_jitter = transforms.Compose([
    transforms.Resize((224, 224)),
    augmentation_color_jitter(brightness=0.4, contrast=0.4, saturation=0.4, hue=0.1),
    transforms.ToTensor(),
    transforms.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225])
])

augmentation_flip = transforms.Compose([
    transforms.Resize((224, 224)),
    transforms.RandomHorizontalFlip(),
    transforms.RandomVerticalFlip(),
    transforms.ToTensor(),
    transforms.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225])
])

augmentation_blur = transforms.Compose([
    transforms.Resize((224, 224)),
    transforms.GaussianBlur(kernel_size=3),
    transforms.ToTensor(),
    transforms.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225])
])
```

```
# 4:1 비율로 train, test 데이터 나누기
train_dataset, test_dataset = torch.utils.data.random_split(original_dataset, [int(0.8 * len(original_dataset)), int(0.2 * len(original_dataset))])

# 훈련 데이터에 증대된 데이터 추가
augmented_dataset1 = train_dataset
augmented_dataset2 = train_dataset
augmented_dataset3 = train_dataset
augmented_dataset4 = train_dataset
augmented_dataset5 = train_dataset

augmented_dataset1.dataset.transfrom = augmentation_blur
train_dataset += augmented_dataset1

augmented_dataset2.dataset.transfrom = augmentation_crop
train_dataset += augmented_dataset2

augmented_dataset3.dataset.transfrom = augmentation_jitter
train_dataset += augmented_dataset3


augmented_dataset4.dataset.transfrom = augmentation_flip
train_dataset += augmented_dataset4

augmented_dataset5.dataset.transfrom = augmentation_rotation
train_dataset += augmented_dataset5

# 데이터 로더 생성
train_loader = torch.utils.data.DataLoader(train_dataset, batch_size=16, shuffle=True, num_workers=2)
test_loader = torch.utils.data.DataLoader(test_dataset, batch_size=16, shuffle=False, num_workers=2)

# 각 set의 sample 수 출력
print("Train set size:", len(train_dataset))
print("Test set size:", len(test_dataset))

# 사전 훈련된 ResNet-200d 모델
model = resnet200d(pretrained=True)
num_ftrs = model.fc.in_features
```

```
half_in_size = round(num_ftrs/2)
layer_width = 1024
Num_class=120

class SpinalNet_ResNet(nn.Module):
    def __init__(self):
        super(SpinalNet_ResNet, self).__init__()

        self.fc_spinal_layer1 = nn.Sequential(
            nn.Dropout(p = 0.5),
            nn.Linear(half_in_size, layer_width),
            nn.BatchNorm1d(layer_width),
            nn.ReLU(inplace=True),)
        self.fc_spinal_layer2 = nn.Sequential(
            nn.Dropout(p = 0.5),
            nn.Linear(half_in_size+layer_width, layer_width),
            nn.BatchNorm1d(layer_width),
            nn.ReLU(inplace=True),)
        self.fc_spinal_layer3 = nn.Sequential(
            nn.Dropout(p = 0.5),
            nn.Linear(half_in_size+layer_width, layer_width),
            nn.BatchNorm1d(layer_width),
            nn.ReLU(inplace=True),)
        self.fc_spinal_layer4 = nn.Sequential(
            nn.Dropout(p = 0.5),
            nn.Linear(half_in_size+layer_width, layer_width),
            nn.BatchNorm1d(layer_width),
            nn.ReLU(inplace=True),)
        self.fc_out = nn.Sequential(
            nn.Dropout(p = 0.5),
            nn.Linear(layer_width*4, Num_class),)

    def forward(self, x):
        x1 = self.fc_spinal_layer1(x[:, 0:half_in_size])
        x2 = self.fc_spinal_layer2(torch.cat([ x[:,half_in_size:2*half_in_size], x1],
dim=1))
        x3 = self.fc_spinal_layer3(torch.cat([ x[:,0:half_in_size], x2], dim=1))
        x4 = self.fc_spinal_layer4(torch.cat([ x[:,half_in_size:2*half_in_size], x3],
dim=1))
```

```
        x = torch.cat([x1, x2], dim=1)
        x = torch.cat([x, x3], dim=1)
        x = torch.cat([x, x4], dim=1)


        x = self.fc_out(x)
        return x


model.fc = SpinalNet_ResNet()

model = model.to(device)

criterion = nn.CrossEntropyLoss()

# 학습률 조정
optimizer = optim.Adam(model.parameters(), lr=0.00001)

# 추가적인 최적화 기법 적용
optimizer = optim.AdamW(model.parameters(), lr=0.00001, weight_decay=1e-5)
scheduler = lr_scheduler.OneCycleLR(optimizer, max_lr=0.0001, epochs=25,
steps_per_epoch=len(train_loader))

epochs = 30
max_accuracy = 0.0
max_weight = None

for epoch in range(epochs):  # 30 에폭만큼 학습
    print("-------------------------------------------")
    print(f"epoch : {epoch+1}  시작")

    running_loss = 0.0

    for i, data in enumerate(train_loader):
        # 입력 데이터 가져오기 및 GPU에 로드
        inputs, labels = data[0].to(device), data[1].to(device)
        # 매개변수 경사도 초기화
        optimizer.zero_grad()

        # 순전파, 역전파, Optimizer 실행
        outputs = model(inputs)
```

```
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()

        # 통계 출력
        running_loss += loss.item()
        if i % 50 == 49:    # 50 미니배치마다 출력
            print('[%d, %5d] loss: %.3f' %(epoch + 1, i + 1, running_loss / 100))
            running_loss = 0.0

model.eval()
val_loss=0
total,correct=0,0

with torch.no_grad():

    train_total, train_correct = 0, 0

    for inputs, labels in train_loader:
        inputs, labels = inputs.to(device), labels.to(device)
        outputs = model(inputs)
        _, predicted = torch.max(outputs.data, 1)
        train_total += labels.size(0)
        train_correct += (predicted == labels).sum().item()

    for inputs, labels in test_loader:
        inputs, labels = inputs.to(device), labels.to(device)
        outputs = model(inputs)
        loss = criterion(outputs, labels)
        val_loss += loss.item()
        _, predicted = torch.max(outputs.data, 1)
        total += labels.size(0)
        correct += (predicted == labels).sum().item()

train_acc = train_correct / train_total
print(f"train data 정확도 : {train_acc:.4f}")

valid_acc = correct / total
print(f'맞은 valid 개수 : {correct}/ total : {total}')
print(f'Valid_data accuracy: {valid_acc:.4f}')
val_loss /= len(test_dataset)
```

```
    # 최고 정확도에서 가중치 갱신
    if valid_acc > max_accuracy:
        max_accuracy = valid_acc
        max_weight = model.state_dict()

        # 최고 가중치 저장
        torch.save(max_weight, "max_weight.pth")

# 모델 평가
print('-----------------------------------------')
print('evaluation mode')
correct = 0
total = 0

eight = torch.load("max_weights.pth")
odel.load_state_dict(weight)
model.eval()

with torch.no_grad():
    for inputs, labels in test_loader:
        inputs, labels = inputs.to(device), labels.to(device)
        outputs = model(inputs)
        _, predicted = torch.max(outputs.data, 1)
        total += labels.size(0)
        correct += (predicted == labels).sum().item()
test_acc = correct / total

print(f'맞은 데이터 개수 : {correct}/ total : {total}')
print(f'Test accuracy: {test_acc:.4f}')
print('-----------------------------------------')
```

4. Caltech-256

4-1. 개요

ResNet-200D 모델을 활용하여 Caltech-256 데이터셋을 학습하고 평가하였습니다.
Caltech-256 데이터셋은 256개의 다양한 카테고리를 가진 이미지 분류 데이터셋입니다. 257번째 클래스는 'clutter' 또는 '기타' 클래스로, 특정 카테고리에 속하지 않는 이미지들이 포함됩니다. 클래스에는 여러 종류의 동물, 차량, 가구, 사람, 식물, 음식, 건물 등이 포함되어 있으며, Caltech-101 데이터셋의 확장판이기도 한 Caltech-256 데이터셋은 객체의 크기, 모양, 각도, 위치 등 다양한 변형을 보여줍니다. 해당 데이터셋에는 여전히 일부 클래스 간 불균형이 존재하며, 이미지의 해상도가 낮다는 단점이 있습니다.

4-2. 요약

1. 필요한 라이브러리와 모듈을 임포트합니다.
2. 데이터 전처리를 위한 변환을 정의합니다.
3. Caltech-256 데이터셋을 다운로드하고 학습 및 테스트 데이터셋으로 분할합니다.
4. 데이터 로더를 사용하여 학습 데이터셋과 테스트 데이터셋을 배치 단위로 제공합니다.
5. ResNet-200D 모델을 생성하고, 전이학습을 위해 미리 학습된 가중치를 로드합니다.
6. 모델의 출력 레이어를 변경하여 클래스의 개수에 맞게 설정합니다.
7. 손실 함수와 최적화기(SGD)를 설정합니다.
8. 지정된 에폭 수에 따라 학습을 진행합니다. 매 에폭마다 학습 데이터를 사용하여 순전파, 손실 계산, 역전파, 그리고 최적화를 수행합니다.
9. 학습이 완료된 모델을 사용하여 테스트 데이터셋에 대한 성능을 평가합니다. 정확도를 계산하여 출력합니다.

코드 실행 시, Caltech-256 데이터셋을 다운로드하고 모델을 학습하며, 각 에폭(epoch)마다 학습 손실(loss)을 출력합니다. 학습이 완료되면 테스트 데이터셋을 사용하여 모델의 정확도를 계산하여 출력합니다.

4-3. 코드 설명

4-3-1) 필요한 라이브러리 및 모듈 임포트

학습에 필요한 PyTorch 라이브러리와 관련 모듈을 임포트합니다. 다음과 같은 라이브러리가 사용됩니다:

- `torch`: PyTorch의 기본 라이브러리

- `torch.nn`: 신경망 모델과 관련된 모듈
- `torch.optim`: 옵티마이저를 포함한 최적화 알고리즘
- `torch.utils.data`: 데이터셋 및 데이터로더 관련 모듈
- `torchvision.datasets`: 이미지 데이터셋을 제공하는 모듈
- `torchvision.transforms`: 이미지 전처리를 위한 변환 함수
- `torchvision.models`: 사전 학습된 모델 아키텍처를 포함한 모듈

4-3-2) 학습 설정

학습에 사용되는 설정 값들을 정의합니다. 다음과 같은 값들이 설정됩니다:

- `device`: 학습을 실행할 디바이스를 설정합니다. GPU를 사용 가능한 경우 "cuda", 그렇지 않은 경우 "cpu"로 설정됩니다.
- `batch_size`: 미니배치의 크기를 설정합니다.
- `num_epochs`: 전체 학습 과정에서 반복할 에폭(epoch) 수를 설정합니다.
- `learning_rate`: 학습률을 설정합니다.

4-3-3) 데이터 전처리 및 데이터 로더 설정

학습에 사용될 데이터셋의 전처리 및 데이터로더 설정을 수행합니다. 다음과 같은 전처리 과정이 포함됩니다:

- 이미지 크기 조정: `transforms.Resize`를 사용하여 이미지 크기를 (224, 224)로 조정
- 이미지를 텐서로 변환: `transforms.ToTensor`를 사용하여 이미지를 PyTorch 텐서로 변환합
- 이미지 정규화: `transforms.Normalize`를 사용하여 이미지를 정규화. 평균과 표준편차 값을 사용하여 정규화를 수행

4-3-4) 데이터 로더 생성

```
trainloader = DataLoader(train_dataset, batch_size=32, shuffle=True, collate_fn=collate_wrapper)
testloader = DataLoader(test_dataset, batch_size=32, shuffle=False, collate_fn=collate_wrapper)
```

데이터 로더(DataLoader)를 사용하여 훈련 데이터와 테스트 데이터를 배치 단위로 제공합니다. 각 데이터 로더는 배치 크기를 32로 설정하고, 훈련 데이터는 섞어서 순서를 무작위로 만들도록 shuffle=True로 설정하였습니다.

4-3-5) ResNet-200D 모델 생성 및 설정

```
model = resnet200d(pretrained=True)
```

num_ftrs = model.fc.in_features
model.fc = nn.Linear(num_ftrs, 256+1)
model = model.to(device)

ResNet-200D 모델을 생성하고, 미리 학습된 가중치를 로드합니다. `model.fc`는 모델의 출력 레이어를 변경하여 클래스의 개수에 맞게 설정합니다. Caltech-256 데이터셋은 256개의 클래스와 clutter 클래스를 포함한 총 257개의 클래스를 가지므로, 출력 레이어의 노드 개수를 256+1로 설정합니다. 모델을 GPU로 이동시킵니다.

4-3-6) 손실 함수와 옵티마이저 설정

criterion = nn.CrossEntropyLoss()
optimizer = optim.SGD(model.parameters(), lr=0.01, momentum=0.9)

손실 함수로는 CrossEntropyLoss를 사용하고, 옵티마이저로는 확률적 경사 하강법(SGD)을 사용합니다. 1 에폭에서 5 에폭까지는 0.01의 learning rate를 사용하였고, 6 에폭부터는 0.001의 learning rate를 사용하였습니다. 모멘텀은 0.9로 설정합니다.

4-3-7) 학습 및 평가

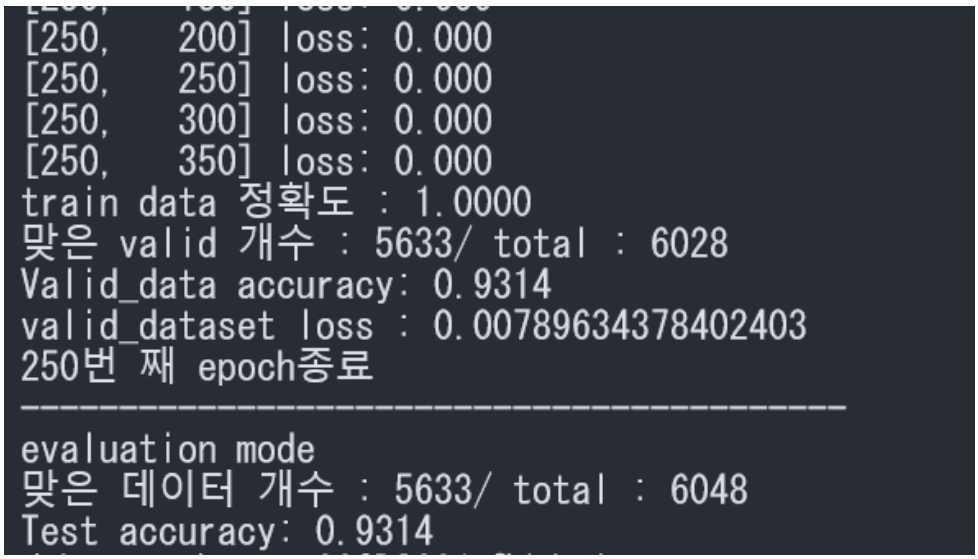
for epoch in range(10)

10 에폭에 걸쳐 학습을 수행합니다. 매 에폭마다 훈련 데이터를 사용하여 순전파(forward pass), 손실 계산, 역전파(backward pass), 그리고 옵티마이저를 통한 가중치 업데이트를 수행합니다.

with torch.no_grad()

학습이 완료된 후, 테스트 데이터셋을 사용하여 모델의 성능을 평가합니다. 평가 시에는 기울기 계산이 필요하지 않으므로 `torch.no_grad()`로 감싼 상태에서 평가를 수행합니다. 예측값과 실제 레이블을 비교하여 정확도를 계산합니다.

4-4. 결과



학습이 완료된 후, 테스트 데이터셋에 대한 성능을 평가하였습니다. 정확도 0.9314의 결과를 얻었습니다.

4-5. 전체 코드

```
import torch
import torchvision
import torchvision.transforms as transforms
import torch.optim as optim
import torch.nn as nn
import time
from torch. utils.data import Dataset,DataLoader, random_split
from filtering import HandleInconsistentImageSize,collate_wrapper # 그레이 스케일 이미지 제거 함수 따로 설정
from timm.models.resnet import resnet200d
from torchvision import models
from torch.optim import lr_scheduler

#data augmentation을 원활하게 하기 위해 따로 dataset 클래스를 정의

class MyDataset(Dataset):
    def __init__(self,subset,transform=None):
        self.subset=subset
        self.transform=transform

    def __getitem__(self,index):
        x,y=self.subset[index]
        if self.transform:
            x=self.transform(x)
        return x,y

    def __len__(self):
        return len(self.subset)

# Device 설정
device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")


# 데이터 전처리 설정
# train dataset에 data augmentation을 적용할 것입니다.
# 매 에폭마다 train dataset에 transform을 가하여 augmentation을 할 것입니다.
train_transforms = transforms.Compose([
    transforms.Resize((240,240)),
    transforms.RandomRotation(15,),
```

```
    transforms.RandomCrop(224),
    transforms.RandomHorizontalFlip(),
    transforms.ToTensor(),
    transforms.Normalize((0.48,), (0.225,))])

test_transforms = transforms.Compose([
    transforms.Resize((224, 224)),
    transforms.ToTensor(),
    transforms.Normalize((0.48,), (0.225,))])

caltech_full= torchvision.datasets.Caltech256(root='./caltech',
transform=None,download=True)

print('caltech-256 dataset download completed')
print(f'{len(caltech_full)}개의 데이터셋이 다운로드 되었음')
print('------------------------------------------------------------')

#train data와 test data분할
train_size = int(0.8 * len(caltech_full))
test_size = len(caltech_full) - train_size
train_dataset, test_dataset = random_split(caltech_full, [train_size, test_size])

test_set=MyDataset(test_dataset,test_transforms)

batch_size=64

#valid data도 만들기
testloader = DataLoader(test_set, batch_size=batch_size,
shuffle=False,collate_fn=collate_wrapper)
val_loader=DataLoader(test_set,batch_size=batch_size,shuffle=False,collate_fn=collate_wr
apper)

# 모델 생성 및 전이학습
model = resnet200d(pretrained=True)
num_ftrs = model.fc.in_features

half_in_size = round(num_ftrs/2)
layer_width = 1024
Num_class=257

class SpinalNet_ResNet(nn.Module):
```

```
    def __init__(self):
        super(SpinalNet_ResNet, self).__init__()

        self.fc_spinal_layer1 = nn.Sequential(
            nn.Dropout(p = 0.5),
            nn.Linear(half_in_size, layer_width),
            nn.BatchNorm1d(layer_width),
            nn.ReLU(inplace=True),)
        self.fc_spinal_layer2 = nn.Sequential(
            nn.Dropout(p = 0.5),
            nn.Linear(half_in_size+layer_width, layer_width),
            nn.BatchNorm1d(layer_width),
            nn.ReLU(inplace=True),)
        self.fc_spinal_layer3 = nn.Sequential(
            nn.Dropout(p = 0.5),
            nn.Linear(half_in_size+layer_width, layer_width),
            nn.BatchNorm1d(layer_width),
            nn.ReLU(inplace=True),)
        self.fc_spinal_layer4 = nn.Sequential(
            nn.Dropout(p = 0.5),
            nn.Linear(half_in_size+layer_width, layer_width),
            nn.BatchNorm1d(layer_width),
            nn.ReLU(inplace=True),)
        self.fc_out = nn.Sequential(
            nn.Dropout(p = 0.5),
            nn.Linear(layer_width*4, Num_class),)

    def forward(self, x):
        x1 = self.fc_spinal_layer1(x[:, 0:half_in_size])
        x2 = self.fc_spinal_layer2(torch.cat([ x[:,half_in_size:2*half_in_size], x1], dim=1))
        x3 = self.fc_spinal_layer3(torch.cat([ x[:,0:half_in_size], x2], dim=1))
        x4 = self.fc_spinal_layer4(torch.cat([ x[:,half_in_size:2*half_in_size], x3], dim=1))


        x = torch.cat([x1, x2], dim=1)
        x = torch.cat([x, x3], dim=1)
        x = torch.cat([x, x4], dim=1)
```

```
        x = self.fc_out(x)
        return x


model.fc = SpinalNet_ResNet()

model = model.to(device) #모델의 fully connected layer에 spinalnet 사용

for epoch in range(250):  # 250 에폭만큼 학습
    print("-------------------------------------------")
    print(f"epoch : {epoch+1}  시작")


    running_loss = 0.0


    train_set=MyDataset(train_dataset,train_transforms)
    trainloader = DataLoader(train_set, batch_size=batch_size,
shuffle=True,collate_fn=collate_wrapper)


    for i, data in enumerate(trainloader):
        # 입력 데이터 가져오기 및 GPU에 로드
        inputs, labels = data[0].to(device), data[1].to(device)
        # 매개변수 경사도 초기화
        optimizer_sgd.zero_grad()

        # 순전파, 역전파, Optimizer 실행
        outputs = model(inputs)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer_sgd.step()

        # 통계 출력
        running_loss += loss.item()
        if i % 50 == 49:    # 50 미니배치마다 출력
            print('[%d, %5d] loss: %.3f' %(epoch + 1, i + 1, running_loss / 100))
            running_loss = 0.0

    model.eval()
    val_loss=0
```

```
    total,correct=0,0

    with torch.no_grad():

        train_total, train_correct = 0, 0

        for inputs, labels in trainloader:
            inputs, labels = inputs.to(device), labels.to(device)
            outputs = model(inputs)
            _, predicted = torch.max(outputs.data, 1)
            train_total += labels.size(0)
            train_correct += (predicted == labels).sum().item()

        for inputs, labels in val_loader:
            inputs, labels = inputs.to(device), labels.to(device)
            outputs = model(inputs)
            loss = criterion(outputs, labels)
            val_loss += loss.item()
            _, predicted = torch.max(outputs.data, 1)
            total += labels.size(0)
            correct += (predicted == labels).sum().item()

    train_acc = train_correct / train_total
    print(f"train data 정확도 : {train_acc:.4f}")

    valid_acc = correct / total
    print(f'맞은 valid 개수 : {correct}/ total : {total}')
    print(f'Valid_data accuracy: {valid_acc:.4f}')
    val_loss /= len(test_dataset

# Evaluate the model on the test set
print('----------------------------------------')
print('evaluation mode')
correct = 0
total = 0
with torch.no_grad():
    for inputs, labels in testloader:
        inputs, labels = inputs.to(device), labels.to(device)
        outputs = model(inputs)
        _, predicted = torch.max(outputs.data, 1)
        total += labels.size(0)
```

```
            correct += (predicted == labels).sum().item()
test_acc = correct / total

print(f'맞은 데이터 개수 : {correct}/ total : {total}')
print(f'Test accuracy: {test_acc:.4f}'))
```