# 머신러닝
# Image Classification Competition
# 10 epoch 수행 보고서

5조 조현우 차민준 최병준 최형우 편수빈 황준우

2023.05.30.

# 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.
   -resize(224,224)
   -Normalize()
3. Cifar-100 데이터셋을 다운로드하고 학습 및 테스트 데이터셋으로 분할합니다. (비율 5:1)
4. 데이터 로더(DataLoader)를 사용하여 학습 데이터셋과 테스트 데이터셋을 배치 단위로 제공합니다. (배치는 16으로 두었습니다.)
5. ResNet-200D 모델을 생성하고, 전이학습을 위해 미리 학습된 가중치를 로드합니다.
6. 모델의 출력 레이어를 변경하여 클래스의 개수에 맞게 설정합니다.
7. 손실 함수(CrossEntropyLoss)와 학습률 스케줄러를 StepLR를 통해 구현합니다. 5 에폭마다 학습률을 0.1배 하였습니다.
8. 지정된 에폭 수에 따라 학습을 진행합니다. 매 에폭마다 학습 데이터를 사용하여 순전파(forward pass), 손실 계산, 역전파(backward pass), 그리고 최적화(optimizer)를 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
```

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

### 2-4-2) 데이터 전처리

```
transform = transforms.Compose([
    transforms.ToTensor(),
    transforms.Resize((224, 224)),
    transforms.Normalize((0.1307,), (0.3081,)),
])
```

데이터 전처리를 위한 변환(Transform)을 정의합니다. `ToTensor()`를 사용하여 이미지를 텐서로 변환하고, `Resize()`를 사용하여 이미지 크기를 224x224로 조정합니다. 정규화(normalization)를 위해 `Normalize()`를 사용하여 입력 데이터를 정규화합니다.

### 2-4-3) Cifar-100 데이터셋 로드 및 분할

```
cifar100 = torchvision.datasets.CIFAR100(root='./data', train=True, download=True, transform=transform)
```

Cifar-100 데이터셋을 다운로드하고, 위에서 정의한 전처리를 적용하여 ‘cifar 100’ 변수에 저장합니다.

```
train_size = int(0.83 * len(cifar100))
test_size = len(cifar100) - train_size
train_dataset, test_dataset = random_split(cifar100, [train_size, test_size])
```

5:1 비율로 train, test 데이터를 나눕니다. 전체 데이터셋의 83%를 학습에 사용하고, 나머지 17%를 테스트에 사용합니다. torch.utils.data.random_split 함수를 사용하여 분할합니다.

**2-4-4) 데이터 로더 생성**

```
trainloader = DataLoader(train_dataset, batch_size=16, shuffle=True)
testloader = DataLoader(test_dataset, batch_size=16, shuffle=False)
```

데이터 로더(DataLoader)를 사용하여 훈련 데이터와 테스트 데이터를 배치 단위로 제공합니다. 각 데이터 로더는 배치 크기를 16로 설정하고, 훈련 데이터는 섞어서 순서를 무작위로 만들도록 shuffle=True로 설정하였습니다.

**2-4-5) ResNet-200D 모델 생성 및 설정**

```
model = resnet200d(pretrained=True)
num_ftrs = model.fc.in_features
model.fc = nn.Linear(num_ftrs, 100)
model = model.to(device)
```

ResNet-200D 모델을 생성하고, 미리 학습된 가중치를 로드합니다. `model.fc`는 모델의 출력 레이어를 변경하여 클래스의 개수에 맞게 설정합니다. 모델의 fully connected layer(model.fc)를 CIFAR-100 데이터셋에 맞게 출력 노드 개수가 100인 선형 레이어로 변경합니다. 이렇게 함으로써 모델은 CIFAR-100의 100개 클래스를 예측할 수 있게 됩니다.

**2-4-6) 손실 함수와 옵티마이저 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)
```

손실함수로 CrossEntropyLoss를 사용하였고 StepLR 스케줄러를 사용하였습니다. 이때 StepLR은 일정한 간격으로 학습률을 감소시키는 스케줄러입니다. 정해진 에폭(epoch)마다 학습률을 감소시키는데, 사용자가 지정한 간격마다 학습률을 주어진 비율로 줄입니다. 이 코드에서는 StepLR을 이용하여 3 epoch마다 0.1배 하여 적용하였습니다.

**2-4-7) 학습 및 평가**

```
for epoch in range(10)
```

10 에폭에 걸쳐 학습을 수행합니다. 매 에폭마다 훈련 데이터를 사용하여 순전파, 손실 계산, 역전파, 그리고 옵티마이저를 통한 가중치 업데이트를 수행합니다. 이때 test accuracy를 매번 출

력합니다.

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.

## 2-4. 결과

```
Epoch 10 시작
Epoch [10, 100] loss: 0.007
Epoch [10, 200] loss: 0.007
Epoch [10, 300] loss: 0.007
Epoch [10, 400] loss: 0.010
Epoch [10, 500] loss: 0.007
Epoch [10, 600] loss: 0.006
Epoch [10, 700] loss: 0.011
Epoch [10, 800] loss: 0.006
Epoch [10, 900] loss: 0.009
Epoch [10, 1000] loss: 0.007
Epoch [10, 1100] loss: 0.009
Epoch [10, 1200] loss: 0.008
Epoch [10, 1300] loss: 0.006
Epoch [10, 1400] loss: 0.010
Epoch [10, 1500] loss: 0.008
Epoch [10, 1600] loss: 0.009
Epoch [10, 1700] loss: 0.006
Epoch [10, 1800] loss: 0.006
Epoch [10, 1900] loss: 0.008
Epoch [10, 2000] loss: 0.008
Epoch [10, 2100] loss: 0.007
Epoch [10, 2200] loss: 0.005
Epoch [10, 2300] loss: 0.008
Epoch [10, 2400] loss: 0.006
Epoch [10, 2500] loss: 0.010
테스트 데이터셋 정확도: 0.8576
학습 완료. 소요 시간: 6038.09초
테스트 데이터셋 정확도: 0.8576
```

학습이 완료된 후, 테스트 데이터셋에 대한 성능을 평가하였습니다. 정확도 0.8576의 결과를 얻었습니다.

## 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

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

# 데이터 전처리
transform = transforms.Compose([
    transforms.ToTensor(),
    transforms.Resize((224, 224)),
    transforms.Normalize((0.1307,), (0.3081,)),
])

cifar100 = torchvision.datasets.CIFAR100(root='./data', train=True, download=True,
transform=transform)
train_size = int(0.83 * len(cifar100))
test_size = len(cifar100) - train_size
train_dataset, test_dataset = random_split(cifar100, [train_size, test_size])

trainloader = DataLoader(train_dataset, batch_size=16, shuffle=True)
testloader = DataLoader(test_dataset, batch_size=16, shuffle=False)

# 모델 생성
model = resnet200d(pretrained=True)
num_ftrs = model.fc.in_features
model.fc = nn.Linear(num_ftrs, 100)  # CIFAR-100은 100개의 클래스를 가지므로 출력
노드를 100으로 설정

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"Training instances: {len(trainloader.dataset)}")
print(f"Validation instances: {len(testloader.dataset)}")

for epoch in range(10):  # 10 에폭만큼 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}")
```

# 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 데이터 셋은 ImageNet에서 파생되었으며, 세부적인 카테고리 내에서의 분류 문제를 해결하는데 유용합니다.

## 3-2. 요약

1. 필요한 라이브러리와 모듈을 임포트합니다.
2. 데이터 변환을 합니다.
    -resize(224,224)
    -Normalize()
3. Stanford dog 데이터셋을 다운로드하고 학습 및 테스트 데이터셋으로 분할합니다. (이때 비율을 4:1로 둡니다)
4. 데이터 로더(DataLoader)를 사용하여 학습 데이터셋과 테스트 데이터셋을 배치 단위로 제공합니다. (배치는 16으로 두었습니다.)
5. ResNet-200D 모델을 생성하고, 전이학습을 위해 미리 학습된 가중치를 로드합니다.
6. 모델의 출력 레이어를 변경하여 클래스의 개수에 맞게 설정합니다.
7. 손실 함수(CrossEntropyLoss)와 학습률 스케줄러를 StepLR를 통해 구현합니다. 3에폭마다 학습률을 0.1배 하여 구현하였습니다
8. 지정된 에폭 수에 따라 학습을 진행합니다. 매 에폭마다 학습 데이터를 사용하여 순전파(forward pass), 손실 계산, 역전파(backward pass), 그리고 최적화(optimizer)를 수행합니다.
9. 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 Test Accurarcy를 출력합니다. 학습이 완료되면 테스트 데이터셋을 사용하여 모델의 최종 정확도를 계산하여 출력합니다.

## 3-3. 코드 설명

### 3-3-1) 필요한 라이브러리 및 모듈 임포트

```
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 timm.models.resnet import resnet200d
from sklearn.metrics import accuracy_score
```

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 변환(Transform)을 정의합니다. `ToTensor()`를 사용하여 이미지를 텐서로 변환하고, `Resize()`를 사용하여 이미지 크기를 224x224로 조정합니다. 정규화(normalization)를 위해 `Normalize()`를 사용하여 입력 데이터를 정규화합니다.

### 3-3-3) Stanford dog 데이터셋 로드 및 분할

```
extracted_dir = "./data/stanford_dogs/Images"
dataset = torchvision.datasets.ImageFolder(extracted_dir, transform=transform)
```

Stanford dog 데이터셋을 다운로드하고, 위에서 정의한 전처리(transform)를 적용하여 'dataset' 변수에 저장합니다.

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

4:1 비율로 train, test 데이터를 나눕니다.

### 3-3-4) 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 로더(DataLoader)를 사용하여 훈련 데이터와 테스트 데이터를 배치 단위로 제공합니다. 각 데이터 로더는 배치 크기를 16로 설정하고, 훈련 데이터는 섞어서 순서를 무작위로 만들도록 shuffle=True로 설정하였습니다.

### 3-3-5) ResNet-200D 모델 생성 및 설정

model = resnet200d(pretrained=True)
model.fc = nn.Linear(num_ftrs, num_classes)
model = model.to(device)

ResNet-200D 모델을 생성하고, 미리 학습된 가중치를 로드합니다. `model.fc`는 모델의 출력 레이어를 변경하여 클래스의 개수에 맞게 설정합니다. 또한 출력 레이어의 노드 개수를 120개로 설정합니다. 모델을 GPU로 이동시킵니다.

### 3-3-6) 손실 함수와 옵티마이저 설정

optimizer = optim.SGD(model.parameters(), lr=0.01)
scheduler = torch.optim.lr_scheduler.StepLR(optimizer, step_size=3, gamma=0.1)

손실함수로 CrossEntropyLoss를 사용하였고 StepLR 스케줄러를 사용하였습니다. 이때 StepLR은 일정한 간격으로 학습률을 감소시키는 스케줄러입니다. 정해진 에폭(epoch)마다 학습률을 감소시키는데, 사용자가 지정한 간격마다 학습률을 주어진 비율로 줄입니다. 이 코드에서는 StepLR을 이용하여 3 epoch마다 0.1배 하여 적용하였습니다.

### 3-3-7) 학습 및 평가

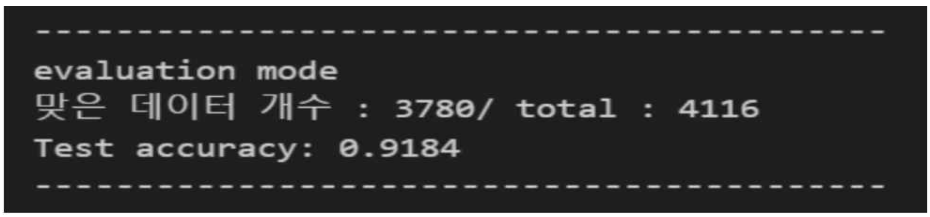
for epoch in range(num_epochs)

10 에폭에 걸쳐 학습을 수행합니다. 매 에폭마다 훈련 데이터를 사용하여 순전파(forward pass), 손실 계산, 역전파(backward pass), 그리고 옵티마이저를 통한 가중치 업데이트를 수행합니다. 이때 train accuracy와 test accuracy를 매번 출력합니다.

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.

## 3-4. 결과




학습이 완료된 후, 테스트 데이터셋에 대한 성능을 평가하였습니다. 정확도 0.9184의 결과를 얻었습니다.

## 3-5. 전체 코드

```
pip install timm

import os
import urllib.request
import tarfile
import random
from sklearn.model_selection import train_test_split

# 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")

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
import os

# 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
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])
])

# 추출된 데이터 경로 설정
extracted_dir = "./data/stanford_dogs/Images"

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

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

# 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))

from torch.optim.lr_scheduler import CosineAnnealingLR

# 사전 훈련된 ResNet-200d 모델
model = resnet200d(pretrained=True)

# Replace the last fully connected layer with a new layer for the desired number of classes
model.fc = nn.Linear(model.fc.in_features, num_classes)

# 모델을 장치로 이동(사용 가능한 경우 GPU)
model = model.to(device)

# 손실 함수
criterion = nn.CrossEntropyLoss()

# 훈련 에포크 설정
num_epochs = 10

# 옵티마이저 초기화
optimizer = optim.SGD(model.parameters(), lr=0.01)

# StepLR 적용위해 스케줄러 초기화 (step_size는 학습률을 감소시킬 에폭 주기, gamma는 학습률을 감소시키는 비율)
scheduler = torch.optim.lr_scheduler.StepLR(optimizer, step_size=3, gamma=0.1)

# 훈련
for epoch in range(num_epochs):
```

```
# train 위해 변수 초기화
correct = 0
total = 0

# train 모드
model.train()

# Iterate over the training data
for images, labels in train_loader:
    # 기기로 데이터 이동
    images = images.to(device)
    labels = labels.to(device)

    optimizer.zero_grad()

    # 순방향 패스
    outputs = model(images)
    loss = criterion(outputs, labels)

    # 역전파 및 최적화
    loss.backward()
    optimizer.step()

    # train 정확도 계산
    _, predicted = torch.max(outputs.data, 1)
    total += labels.size(0)
    correct += (predicted == labels).sum().item()

# train 정확도 계산
train_accuracy = correct / total

# 평가 모드
model.eval()

# test 위해 변수 초기화
correct = 0
total = 0

# Disable gradient computation to save memory and computation
with torch.no_grad():
```

```
            # Iterate over the test data
            for images, labels in test_loader:
                # 기기로 데이터 이동
                images = images.to(device)
                labels = labels.to(device)

                # 순방향 패스
                outputs = model(images)

                # 정확도 계산
                _, predicted = torch.max(outputs.data, 1)
                total += labels.size(0)
                correct += (predicted == labels).sum().item()

        # 정확도 계산
        test_accuracy = correct / total

        # train, test 정확도 출력
        print(f"Epoch [{epoch+1}/{num_epochs}], Train Accuracy: {train_accuracy:.4f}, Test
Accuracy: {test_accuracy:.4f}")

        #학습률 업데이트
        scheduler.step()

# 모델 평가
print('----------------------------------------')
print('evaluation mode')
correct = 0
total = 0
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 변환(Transform)을 정의합니다.
3. Caltech-256 데이터셋을 다운로드하고 학습 및 테스트 데이터셋으로 분할합니다.
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 함수(CrossEntropyLoss)와 최적화기(SGD)를 설정합니다.
8. 지정된 에폭 수에 따라 학습을 진행합니다. 매 에폭마다 학습 데이터를 사용하여 순전파(forward pass), 손실 계산, 역전파(backward pass), 그리고 최적화(optimizer)를 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

```
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 filtering import HandleInconsistentImageSize, collate_wrapper
from timm.models.resnet import resnet200d
```

필요한 라이브러리와 모듈을 임포트합니다. PyTorch, torchvision, torch.optim, torch.nn 등의 라이브러리를 사용하였습니다. 또한, 이미지 크기가 일관되지 않는 경우를 처리하기 위해 `HandleInconsistentImageSize` 클래스와 데이터 로더에서 사용할 `collate_wrapper` 함수를 임포트하였습니다. 그리고 ResNet-200D 모델을 사용하기 위해 `resnet200d`를 임포트합니다.

#### 4-3-2) 데이터 전처리

```
transform = transforms.Compose([
    transforms.ToTensor(),
    transforms.Resize((224, 224)),
    transforms.Normalize((0.1307,), (0.3081,)),
])
```

데이터 전처리를 위한 변환(Transform)을 정의합니다. `ToTensor()`를 사용하여 이미지를 텐서로 변환하고, `Resize()`를 사용하여 이미지 크기를 224x224로 조정합니다. 정규화(normalization)를 위해 `Normalize()`를 사용하여 입력 데이터를 정규화합니다.

#### 4-3-3) Caltech-256 데이터셋 로드 및 분할

```
caltech_full = torchvision.datasets.Caltech256(root='./data', transform=transform, download=True)
```

Caltech-256 데이터셋을 다운로드하고, 위에서 정의한 전처리(transform)를 적용하여 `caltech_full` 변수에 저장합니다.

```
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])
```

전체 데이터셋을 훈련 데이터와 테스트 데이터로 분할합니다. 전체 데이터의 80%를 훈련 데이터로 사용하고, 나머지 20%를 테스트 데이터로 사용합니다.

#### 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

```
[10,   600] loss: 0.004
[10,   700] loss: 0.003
10번 째 epoch종료
-----------------------------------------------
Finished Training. Took 2733.5449180603027 seconds
-----------------------------------------
evaluation mode
맞은 데이터 개수 : 5565/ total : 6048
Test accuracy: 0.9201
```

학습이 완료된 후, 테스트 데이터셋에 대한 성능을 평가하였습니다. 정확도 0,9201의 결과를 얻었습니다.

## 4-5. 전체 코드

```
pip install timm

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 filtering import HandleInconsistentImageSize,collate_wrapper
from timm.models.resnet import resnet200d

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

# 데이터 전처리
transform =  transforms.Compose([
                            transforms.ToTensor(),
                            transforms.Resize((224, 224)),
                            transforms.Normalize((0.1307,), (0.3081,)),
                        ])
```

```
caltech_full= torchvision.datasets.Caltech256(root='./data', transform=transform,download=True)

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


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])


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)

print(f"Training instances: {len(trainloader.dataset)}")
print(f"Validation instances : {len(testloader.dataset)}")



# 모델 생성
model = resnet200d(pretrained=True)
num_ftrs = model.fc.in_features
print(num_ftrs)
model.fc = nn.Linear(num_ftrs, 256+1)  # 256개 클래스와 clutter 클래스를 포함한 257개의 출력노드를 가지는 레이어 추가
model = model.to(device)

# 손실 함수, Optimizer 설정
criterion = nn.CrossEntropyLoss()
optimizer = optim.SGD(model.parameters(), lr=0.01, momentum=0.9)


print('-------------------------------------')
print(f"{len(trainloader.dataset)}개의 caltech-256 데이터에 대한 학습을 시작합니다")
print('-------------------------------------')
```

```
for epoch in range(10):  # 10 에폭만큼 학습
    print("-------------------------------------------")
    print(f"epoch : {epoch+1}  시작")

    #6에폭 부터는 learning rate = 0.001
    if epoch==5:
        optimizer=optim.SGD(model.parameters(),lr=0.001,momentum=0.9)

    running_loss = 0.0
    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.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 % 100 == 99:    # 100 미니배치마다 출력
            print('[%d, %5d] loss: %.3f' %
                  (epoch + 1, i + 1, running_loss / 100))
            running_loss = 0.0


    print(f"{epoch+1}번 째 epoch종료")
    print('-------------------------------------------------')

print('Finished Training. Took {} seconds'.format(time.time() - start_time))

# 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}')
```