CNN architecture

2317038

허지원

개요

CNN architecture은 Pytorch를 사용하여 CIFAR-10 데이터 세트에 대한 이미지 분류 작업을 수행하는 프로젝트이다. 학습된 CNN(VGG16 또는 ResNet50) 모델을 CIFAR-10 데이터 세트에 대해 불러오고, 이를 CPU(또는 GPU)에서 설정한 epoch동안 학습한 뒤, 텍스트 정확도를 측정한다.

VGG(Visual Geometry Group)

: 2014년 Simonyan과 Zisserman이 제안한 CNN Architecture로, 단순하면서도 일관된 구조로 깊은 신경망의 성능을 향상시키는 딥러닝 모델이다.

전체 네트워크에서 3x3 크기의 작은 합성곱 필터만 사용하여 여러 번 연속해서 쌓음으로써, 더 많은 비선형성을 적용하고 파라미터를 줄인다.

- 장점
  - 단순성과 확장성: 일관된 3x3 filter 사용과 반복적인 구조로 모델 구현과 이해가 쉽고, 다양한 변형에 용이하다.
  - 전이 학습 활용: 이미지넷에 학습이 되어있는 VGGNet을 가져와서 다른 테스크에 적용하는 방식으로 전이 학습이 가능하다.

ResNet(Residual Network)

: 2015년 He 등이 제안한 딥러닝 신경망 구조로, Residual Connection(잔차 연결)을 활용하여 매우 깊은 네트워크도 효과적으로 학습할 수 있도록 설계된 모델이다.

각 블록의 입력을 출력에 더하는 구조(Residual Connection)을 도입하여 입력 대비 변화량(잔차)만 학습하도록 하여, 기존의 딥러닝 네트워크에서의 degradation problem을 해결하였다.

- 핵심 원리: 잔차 학습(Residual Learning)

$$F(x) = H(x) - x$$

  - $F(x)$: 잔차(입력 대비 변화량)
  - $H(x)$: 원하는 함수

- ResNet의 특징
  - Residual Block을 쌓아 전체 네트워크를 구성
  - ResNet50에서는 1x1→3x3→1x1의 BottleNeck 구조로 이루어짐

➢ 깊은 네트워크에서도 기울기 소실 없이 안정적인 학습이 가능함

▫ BottleNeck 구조
: 1x1 합성곱(차원 축소) → 3x3 합성곱(특징 추출) → 1x1 합성곱(차원 복원)의 구조로 연산 효율성을 높이고, 파라미터 수를 줄이면서도 깊은 네트워크를 만들 수 있게 한다.

코드 설명

1. vgg16_full.py

1) VGG 클래스: VGG16 모델의 전체 구조를 정의한 클래스

```python
import torch.nn as nn
import math

###### VGG16 #############
class VGG(nn.Module):  2개의 사용 위치
    def __init__(self, features):
        super(VGG, self).__init__()
        self.features = features
        self.classifier = nn.Sequential(
            nn.Dropout(),
            nn.Linear( in_features: 512,  out_features: 512),
            nn.BatchNorm1d(512),
            nn.ReLU(True),
            nn.Dropout(),
            nn.Linear( in_features: 512,  out_features: 10),
        )
         # Initialize weights
        for m in self.modules():
            if isinstance(m, nn.Conv2d):
                n = m.kernel_size[0] * m.kernel_size[1] * m.out_channels
                m.weight.data.normal_( mean: 0, math.sqrt(2. / n))
                m.bias.data.zero_()
```

A. 생성자(__init__ 메서드)
   i. features: convolution 및 pooling 계층으로 구성된 피처 추출 부분
   ii. classifier: FC layer 기반의 분류기
   iii. 가중치 초기화: 모든 합성곱 계층(nn.Conv2d)에 대해 평균을 0, 표준편차를 $\sqrt{2/n}$으로 초기화를 적용하고, bias를 0으로 초기화

```python
    def forward(self, x):
        x = self.features(x)
        x = x.view(x.size(0), -1)
        x = self.classifier(x)
        return x
```

B. forward 함수: 입력 x를 features로 전달하여 피처 맵을 추출하고, batch 차원을 유지하며 1차원 벡터로 변환하여 classifier에 전달

2) make_layer 함수

```python
def make_layers(cfg, batch_norm=False):  1개의 사용 위치
    layers = []
    in_channels = 3
    for v in cfg:
        if v == 'M':
            layers += [nn.MaxPool2d(kernel_size=2, stride=2)]
        else:
            conv2d = nn.Conv2d(in_channels, v, kernel_size=3, padding=1)
            if batch_norm:
                layers += [conv2d, nn.BatchNorm2d(v), nn.ReLU(inplace=True)]
            else:
                layers += [conv2d, nn.ReLU(inplace=True)]
            in_channels = v
    return nn.Sequential(*layers)
```

A. 함수 파라미터
   i. cfg: 각 계층의 출력 채널 수와 M(MaxPooling)을 순서대로 지정하는 리스트
   ii. batch_norm: 배치 정규화(BatchNorm) 계층 추가 여부를 결정
B. 초기 설정: layers(계층들을 저장할 리스트), in_channels(입력 이미지의 채널 수)를 설정
C. cfg 리스트 순회
   i. $v = 'M'$이면 2x2 커널, stride=2의 MaxPooling 계층을 추가
   ii. $v \neq 'M'$이면 3x3 커널, padding=1의 합성곱 계층을 추가
      ▫ batch_norm이 True이면 BatchNorm2d와 ReLU 계층 추가
      ▫ False면 ReLU 계층만 추가
   iii. 다음 계층의 입력 채널 수를 현재 출력 채널 수로 갱신
D. 반환: 쌓은 계층들을 nn.Sequential로 감싸 하나의 모듈로 반환

3) vgg16 함수

```python
def vgg16():  1개의 사용 위치
    # cfg shows 'kernel size'
    # 'M' means 'max pooling'
    cfg = [64, 64, 'M', 128, 128, 'M', 256, 256, 256, 'M', 512, 512, 512, 'M', 512, 512, 512, 'M']
    return VGG(make_layers(cfg))
```

A. VGG16 구조의 모델 인스턴스를 생성하여 반환

2. resnet50_skeleton.py

1) conv1x1, conv3x3 함수

```python
import torch.nn as nn

# 1x1 convolution
def conv1x1(in_channels, out_channels, stride, padding):
    model = nn.Sequential(
        nn.Conv2d(in_channels, out_channels, kernel_size=1, stride=stride, padding=padding),
        nn.BatchNorm2d(out_channels),
        nn.ReLU(inplace=True)
    )
    return model


# 3x3 convolution
def conv3x3(in_channels, out_channels, stride, padding):
    model = nn.Sequential(
        nn.Conv2d(in_channels, out_channels, kernel_size=3, stride=stride, padding=padding),
        nn.BatchNorm2d(out_channels),
        nn.ReLU(inplace=True)
    )
    return model
```

A. 파라미터
   i. in_channels: 입력 채널 수
   ii. out_channels: 출력 채널 수
   iii. stride: 합성곱의 stride
   iv. padding: 합성곱의 padding
B. 1x1(또는 3x3) 커널을 사용하는 합성곱 계층 생성
C. 배치 정규화(BatchNorm2d)와 ReLU 활성화 함수를 순차적으로 추가

2) ResidualBlock 클래스

: 'bottleneck' residual block 구현을 위한 클래스

```python
##########################################################################
# Question 1 : Implement the "bottle_neck building block" part.
# Hint : Think about difference between downsample True and False. How we make the difference by code?
class ResidualBlock(nn.Module):
    def __init__(self, in_channels, middle_channels, out_channels, downsample=False):
        super(ResidualBlock, self).__init__()
        self.downsample = downsample

        if self.downsample:
            self.layer = nn.Sequential(
                conv1x1(in_channels, middle_channels, stride: 2, padding: 0),
                conv3x3(middle_channels, out_channels, stride: 1, padding: 1),
                conv1x1(out_channels, out_channels, stride: 1, padding: 0)
            )
            self.downsize = conv1x1(in_channels, out_channels, stride: 2, padding: 0)

        else:
            self.layer = nn.Sequential(
                conv1x1(in_channels, out_channels, stride: 1, padding: 0),
                conv3x3(out_channels, out_channels, stride: 1, padding: 1),
                conv1x1(out_channels, out_channels, stride: 1, padding: 0)
            )
            self.make_equal_channel = conv1x1(in_channels, out_channels, stride: 1, padding: 0)
```

A. 파라미터

i. in_channels: 입력 feature map의 채널 수
ii. middle_channels: 블록 내부에서 사용하는 중간 채널 수
iii. out_channels: 블록의 출력 채널 수
iv. downsample: 다운 샘플링 여부를 결정
- True: 입력 피처 맵의 공간 크기를 절반으로 줄이고 채널 수를 늘림
- False: 입력과 출력 공간의 크기를 유지

B. downsample = True일 때: 입력 피처 맵의 공간 해상도를 절반으로 줄이고, 채널 수를 늘려줌
i. self.layer
- 1x1conv: 채널 수를 줄이고 stride=2로 다운 샘플링
- 3x3conv: 공간 크기를 유지하고 채널 확장(middle → out)
- 1x1conv: 공간 크기는 유지하고 채널 수를 out_channels로 맞춤

ii. downsize: Resid connection에서 입력 x의 해상도와 채널 수를 출력과 맞춤

C. downsample = False일 때의 내부 계층 구성
i. self.layer
- 1x1convL 채널 수를 out_channels로 맞춤
- 3x3conv: 공간 정보 추출
- 1x1conv: 채널 수 유지

ii. self.make_equal_channel: 입력 x의 채널 수가 출력과 다를 때, Residual connection에서 채널 수를 맞춤

D. forward 함수

```
46      def forward(self, x):
47          if self.downsample:
48              out = self.layer(x)
49              x = self.downsize(x)
50              return out + x
51          else:
52              out = self.layer(x)
53              if x.size() is not out.size():
54                  x = self.make_equal_channel(x)
55              return out + x
56  ################################################################################
```

i. downsample 여부에 따라 Residual connection에서 다운샘플링을 수행
ii. 입력 x를 블록 계층(self.layer)에 통과시킨 결과(out)와, Residual connection을 통해 변환된 입력(x)을 더하여 최종 출력

3) ResNet50_layer4 클래스

```python
######################################################################
# Question 2 : Implement the "class, ResNet50_layer4" part.
# Understand ResNet architecture and fill in the blanks below. (25 points)
# (blank : #blank#, 1 points per blank )
# Implement the code.
class ResNet50_layer4(nn.Module):
    def __init__(self, num_classes=10): # Hint : How many classes in Cifar-10 dataset?
        super(ResNet50_layer4, self).__init__()
        self.layer1 = nn.Sequential(
            nn.Conv2d(in_channels: 3, out_channels: 64, kernel_size: 7, stride: 2, padding: 3),
                # Hint : Through this conv-layer, the input image size is halved.
                #        Consider stride, kernel size, padding and input & output channel sizes.
            nn.BatchNorm2d(64),
            nn.ReLU(inplace=True),
            nn.MaxPool2d(kernel_size: 3, stride: 2, padding: 1)
        )
```

A. layer1
   i. 입력 3채널, 출력 64채널, stride-2, padding=3인 7x7 Conv2d로 입력 이미지의 크기를 절반으로 줄임
   ii. BatchNorm2d, ReLU, 3x3 MaxPool2d로 추가적인 다운 샘플링과 정규화 수행

```python
        self.layer2 = nn.Sequential(
            ResidualBlock(in_channels: 64, middle_channels: 64, out_channels: 256, downsample: False),
            ResidualBlock(in_channels: 256, middle_channels: 64, out_channels: 256, downsample: False),
            ResidualBlock(in_channels: 256, middle_channels: 64, out_channels: 256, downsample: True)
        )
        self.layer3 = nn.Sequential(
            ResidualBlock(in_channels: 256, middle_channels: 128, out_channels: 512, downsample: False),
            ResidualBlock(in_channels: 512, middle_channels: 128, out_channels: 512, downsample: False),
            ResidualBlock(in_channels: 512, middle_channels: 128, out_channels: 512, downsample: False),
            ResidualBlock(in_channels: 512, middle_channels: 128, out_channels: 512, downsample: True)

        )
        self.layer4 = nn.Sequential(
            ResidualBlock(in_channels: 512, middle_channels: 256, out_channels: 1024, downsample: False),
            ResidualBlock(in_channels: 1024, middle_channels: 256, out_channels: 1024, downsample: False),
            ResidualBlock(in_channels: 1024, middle_channels: 256, out_channels: 1024, downsample: False),
            ResidualBlock(in_channels: 1024, middle_channels: 256, out_channels: 1024, downsample: False),
            ResidualBlock(in_channels: 1024, middle_channels: 256, out_channels: 1024, downsample: False),
            ResidualBlock(in_channels: 1024, middle_channels: 256, out_channels: 1024, downsample: False)

        )
```

B. layer2
   i. 처음 두 블록은 입력 크기와 채널 수를 유지 (64 → 256)
   ii. 마지막 블록에서 내부적으로 stride=2를 사용하여 다운샘플링 및 채널 확장
C. layer3
   i. 처음 세 블록은 공간 크기를 유지하며 채널을 256 → 512로 확장
   ii. 마지막 블록에서 다운 샘플링
D. layer4
   i. 모두 공간 크기는 유지하며 채널을 512 → 1024로 확장

```
        self.fc = nn.Linear( in_features: 1024, out_features: 10) # Hint : Think about the reason why fc layer is needed
        self.avgpool = nn.AvgPool2d( kernel_size: 2, stride: 2)

        for m in self.modules():
            if isinstance(m, nn.Linear):
                nn.init.xavier_uniform_(m.weight.data)
            elif isinstance(m, nn.Conv2d):
                nn.init.xavier_uniform_(m.weight.data)
```

E. fc: 1024차원의 입력을 10차원(CIFAR-10 클래스 수)으로 매칭하는 FC 계층
F. avgpool: 2x2 Average Pooling(stride=2)로 공간 크기 축소
G. 가중치 초기화: 모든 Conv2d와 Linear 계층에 xavier uniform 초기화를 적용하여 학습 안정성 높임

```
    def forward(self, x):

        out = self.layer1(x)
        out = self.layer2(out)
        out = self.layer3(out)
        out = self.layer4(out)
        out = self.avgpool(out)
        out = out.view(out.size()[0], -1)
        out = self.fc(out)

        return out
################################################################################
```

H. forward 메서드: 입력 x를 layer1→layer2→layer3→layer4 순서로 통과시킨 뒤 avgpool로 공간 크기를 줄이고 평탄화, fc 계층을 통해 최종 클래스별 점수 출력

4) ResNet50 구현을 위한 network architecture

Layer number	Network	Output Image size
Layer 1	7x7 conv, channel = 64, stride = 2 3x3 max pool, stride = 2	8 x 8
Layer 2	[1x1 conv, channel = 64, 3x3 conv, channel = 64, 1x1 conv, channel = 256] x 2 [1x1 conv, channel = 64, stride = 2 3x3 conv, channel = 64, 1x1 conv, channel = 256] x 1	4 x 4
Layer 3	[1x1 conv, channel = 128, 3x3 conv, channel = 128, 1x1 conv, channel = 512] x 3 [1x1 conv, channel = 128, stride = 2 3x3 conv, channel = 128, 1x1 conv, channel = 512] x 1	2 x 2
Layer 4	[1x1 conv, channel = 256, 3x3 conv, channel = 256, 1x1 conv, channel = 1024] x 6	2 x 2
	AvgPool	1 x 1
	Fully connected layer	?

3. main.py

1) 라이브러리 불러오기

```python
import torch
import torch.nn as nn
import torchvision
import torchvision.transforms as transforms
from vgg16_full import *
# from resnet50_skeleton import *
```

A. torch, torch.nn: PyTorch의 핵심 라이브러리 및 신경망 모듈
B. torchvision, torchvision.transforms: 데이터 세트 및 이미지 전처리 함수
C. vgg16_full, resnet50_skeleton: 직접 정의한 모델 구조

2) 디바이스 설정 및 데이터 전처리

```python
device = torch.device('cuda:0' if torch.cuda.is_available() else 'cpu')
# device = torch.device('cpu')

# Image Preprocessing
transform_train = transforms.Compose([
    transforms.RandomCrop(size: 32, padding=4),
    transforms.RandomHorizontalFlip(),
    transforms.ToTensor(),
    transforms.Normalize(mean: (0.4914, 0.4822, 0.4465), std: (0.2023, 0.1994, 0.2010)),
])

transform_test = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize(mean: (0.4914, 0.4822, 0.4465), std: (0.2023, 0.1994, 0.2010)),
])
  # CIFAR-10 Dataset
  train_dataset = torchvision.datasets.CIFAR10(root='../osproj/data/',
                                               train=True,
                                               transform=transform_train,
                                               download=False) # Change Dow

  test_dataset = torchvision.datasets.CIFAR10(root='../osproj/data/',
                                              train=False,
                                              transform=transform_test)


  # data loader
  train_loader = torch.utils.data.DataLoader(dataset=train_dataset,
                                             batch_size=100,
                                             shuffle=True)
  test_loader = torch.utils.data.DataLoader(dataset=test_dataset,
                                            batch_size=100,
                                            shuffle=False)
```

A. 디바이스 설정: CUDA가 사용 가능하면 GPU를 사용하고, 그렇지 않으면 CPU를 사용하도록 디바이스를 설정
B. 학습/검증 데이터 변환
   i. 학습 데이터: 데이터 증강(무작위 크롭, 좌우 반전)을 적용한 후,

CIFAR-10 데이터 세트의 평균과 표준편차로 정규화

ii. 검증 데이터: 데이터 증강 없이 정규화 적용

C. CIFAR-10 데이터 세트 로드: 학습/검증용 데이터 세트를 각각 로드

D. 데이터 로더 설정: DataLoader를 활용하여 데이터를 미니배치 단위로 불러옴

i. 학습 데이터: 매 epoch마다 데이터를 섞음

ii. 검증 데이터: 데이터 순서를 유지

3) 모델 선택 및 적용

```
42  ##############################################################
43  # Choose model
44  # model = ResNet50_layer4().to(device)
45  # PATH = './resnet50_epoch285.ckpt' # test acc would be almost 80
46
47  model = vgg16().to(device)
48  PATH = './vgg16_epoch250.ckpt'  # test acc would be almost 85
49  ##############################################################
50  # checkpoint = torch.load(PATH)
51  checkpoint = torch.load(PATH, map_location='cpu')
52  model.load_state_dict(checkpoint)
```

A. 모델 선택: vgg16() 또는 ResNet50_layer4() 모델을 선택

B. 사전 학습된 가중치: 사전에 학습된 모델 파라미터 (vgg16_epoch250.ckpy 또는 resnet50_epoch285.ckpt)를 불러와 모델에 적용

4) 모델 학습

```
54  # Train Model
55  # Hyper-parameters
56  num_epochs = 1  # students should train 1 epoch because they will use cpu
57  learning_rate = 0.001
58
59  # Loss and optimizer
60  criterion = nn.CrossEntropyLoss()
61  optimizer = torch.optim.Adam(model.parameters(), lr=learning_rate)
62
63  # For updating learning rate
64  def update_lr(optimizer, lr):  1개의 사용 위치
65      for param_group in optimizer.param_groups:
66          param_group['lr'] = lr
```

A. 하이퍼 파라미터 및 손실함수/최적화 설정

i. num_epochs: 학습 반복 횟수 (cpu 환경에서는 시간 단축을 위해 1epoch만 학습)

ii. learning_rate: 옵티마이저의 초기 학습률 설정

iii. criterion: 다중 클래스 분류에 적합한 Cross Entropy Loss 함수 사용

iv. optimizer: Adam 옵티마이저로 모델 파라미터를 최적화

B. 학습률 업데이트 함수(update_lr): 옵티마이저의 학습률을 동적으로 변경하기 위한 함수 정의

```
# Train the model
total_step = len(train_loader)
current_lr = learning_rate

for epoch in range(num_epochs):

    model.train()
    train_loss = 0

    for batch_index, (images, labels) in enumerate(train_loader):
        # print(images.shape)
        images = images.to(device)  # "images" = "inputs"
        labels = labels.to(device)  # "labels" = "targets"

        # Forward pass
        outputs = model(images)
        loss = criterion(outputs, labels)

        # Backward and optimize
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

        train_loss += loss.item()

        if (batch_index + 1) % 100 == 0:
            print("Epoch [{}/{}], Step [{}/{}] Loss: {:.4f}"
                  .format( *args: epoch + 1, num_epochs, batch_index + 1, total_step, train_loss / (batch_index + 1)))
```

C. 모델 학습 루프 설정: 각 epoch마다 batch 단위로 입력과 정답을 모델에 전달하여 학습을 진행

D. 학습 중 loss 추적 및 출력: 100 batch마다 평균 손실을 출력

```
        # Decay learning rate
        if (epoch + 1) % 20 == 0:
            current_lr /= 3
            update_lr(optimizer, current_lr)
            torch.save(model.state_dict(), './vgg16_epoch' + str(epoch+1)+'.ckpt')

    # Save the model checkpoint
    torch.save(model.state_dict(), f: './vgg16_final.ckpt')
```

E. 학습률 감소 및 중간 checkpoint 저장: 20 epoch마다 학습률을 1/3으로 줄이고, 모델 파라미터를 저장

F. 학습된 모델의 최종 파라미터 저장: 학습 종료 후 최종 모델 파라미터를 파일로 저장

5) 모델 평가

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

    print('Accuracy of the model on the test images: {} %'.format(100 * correct / total))
```

A. 평가 모드 전환: model.eval()로 평가 모드 전환

B. 테스트 데이터 세트에 대한 정확도 계산: 테스트 데이터 전체에 대해

예측을 수행하고, 예측값과 실제값을 비교하여 정확도를 계산

6) 실행 결과

i. vgg16

➢ num_epochs =1일 때

```
100.0%
Epoch [1/1], Step [100/500] Loss: 0.1936
Epoch [1/1], Step [200/500] Loss: 0.1846
Epoch [1/1], Step [300/500] Loss: 0.1905
Epoch [1/1], Step [400/500] Loss: 0.1862
Epoch [1/1], Step [500/500] Loss: 0.1875
Accuracy of the model on the test images: 86.1 %
```

➢ num_epochs = 2일 때

```
Epoch [1/2], Step [100/500] Loss: 0.1931
Epoch [1/2], Step [200/500] Loss: 0.1794
Epoch [1/2], Step [300/500] Loss: 0.1854
Epoch [1/2], Step [400/500] Loss: 0.1857
Epoch [1/2], Step [500/500] Loss: 0.1871
Epoch [2/2], Step [100/500] Loss: 0.1672
Epoch [2/2], Step [200/500] Loss: 0.1786
Epoch [2/2], Step [300/500] Loss: 0.1850
Epoch [2/2], Step [400/500] Loss: 0.1885
Epoch [2/2], Step [500/500] Loss: 0.1928
Accuracy of the model on the test images: 86.84 %
```

ii. resnet50

➢ num_epochs = 1일 때

```
Epoch [1/1], Step [100/500] Loss: 1.5022
Epoch [1/1], Step [200/500] Loss: 1.0542
Epoch [1/1], Step [300/500] Loss: 0.8826
Epoch [1/1], Step [400/500] Loss: 0.7840
Epoch [1/1], Step [500/500] Loss: 0.7245
Accuracy of the model on the test images: 80.36 %
```

➢ num_epochs = 2일 때

```
Epoch [1/2], Step [100/500] Loss: 1.4983
Epoch [1/2], Step [200/500] Loss: 1.0404
Epoch [1/2], Step [300/500] Loss: 0.8702
Epoch [1/2], Step [400/500] Loss: 0.7782
Epoch [1/2], Step [500/500] Loss: 0.7210
Epoch [2/2], Step [100/500] Loss: 0.4414
Epoch [2/2], Step [200/500] Loss: 0.4464
Epoch [2/2], Step [300/500] Loss: 0.4411
Epoch [2/2], Step [400/500] Loss: 0.4383
Epoch [2/2], Step [500/500] Loss: 0.4397
Accuracy of the model on the test images: 81.56 %
```

→ num_epochs = 1인 동일한 조건에서 **VGG16의 테스트 이미지 정확도는 86.1%, ResNet50의 정확도는 80.36%**로 나타났다.

→ num_epochs = 2인 동일한 조건에서 **VGG16의 테스트 이미지 정확도는 86.64%, ResNet50의 정확도는 81.56%**로 나타났다.

위 실행 결과를 분석해보면 epoch의 수가 많아질수록 두 모델에서의 정확도가 높아지는 것을 알 수 있다. VGG16은 0.54%, ResNet50은 1.20% 높아졌다.

이론적으로는 ResNet50이 VGG16보다 높은 성능을 기대할 수 있지만, 실제 실행에서는 오히려 VGG16이 더 높은 정확도를 보였다. 이 원인으로는 여러가지가 있지만, CPU 환경에서의 연산 한계로 인해 epoch의 수를 1로 설정한 것이 가장 큰 원인이 되었을 가능성이 높다.

epoch의 수에 따른 정확도 상승률 비교를 통해 epoch가 높아질 수록 ResNet50에서의 정확도가 훨씬 빠르게 상승할 것이라고 예측할 수 있다.