[딥러닝 파이토치 교과서] 6장 합성곱 신경망2-1

작성일시	@2024년 11월 4일 오후 9:45

6.1 이미지 분류를 위한 신경망

입력 데이터로 이미지를 사용한 분류(classification)는 특정 대상이 영상 내에 존재하는지 여부를 판단하는 것이다.

6.1.1 LeNet-5

LeNet-5는 합성곱 신경망이라는 개념을 최초로 얀 르쿤이 개발한 구조이다. 1995년 얀 르쿤, 레옹 보토, 요슈아 벤지오, 패트릭 하프너가 수표에 쓴 손글씨 숫자를 인식하는 딥러닝 구조 LeNet-5를 발표했는데, 그것이 현재 CNN의 초석이 되었다.

➡️ **LeNet-5는 합성곱과 다운 샘플링(혹은 풀링)을 반복적으로 거치면서 마지막에 완전연결층에서 분류를 수행한다.**

C1 : 5×5 합성곱 연산 후 28×28 크기의 특성 맵(feature map) 여섯 개를 생성한다.

S2 : 다운 샘플링하여 특성 맵 크기를 14×14로 줄인다.

C3 : 5×5 합성곱 연산하여 10×10 크기의 특성 맵 16개를 생성한다.

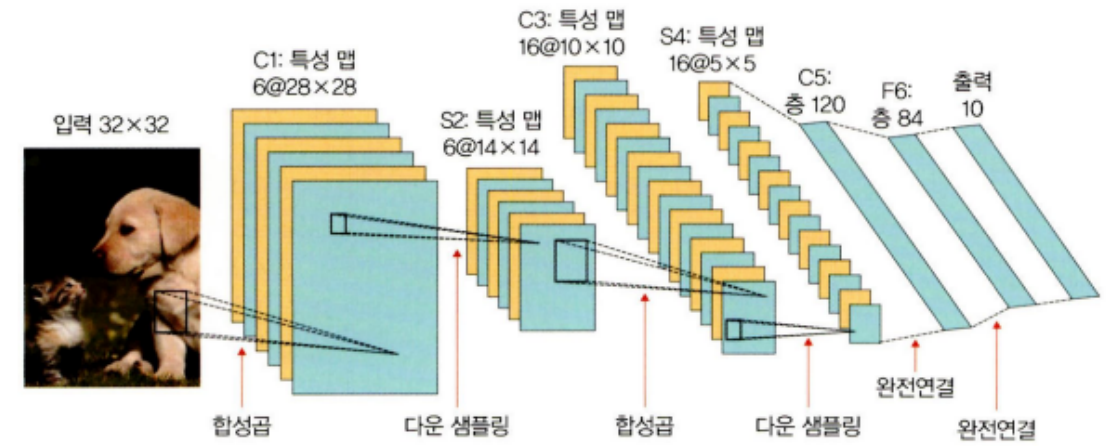
S4 : 다운 샘플링하여 특성 맵 크기를 5×5로 줄인다.

C5 : 5×5 합성곱 연산하여 1×1 크기의 특성 맵 120개를 생성한다.

F6 : 완전연결층으로 C5의 결과를 유닛(또는 노드) 84개에 연결시킨다.

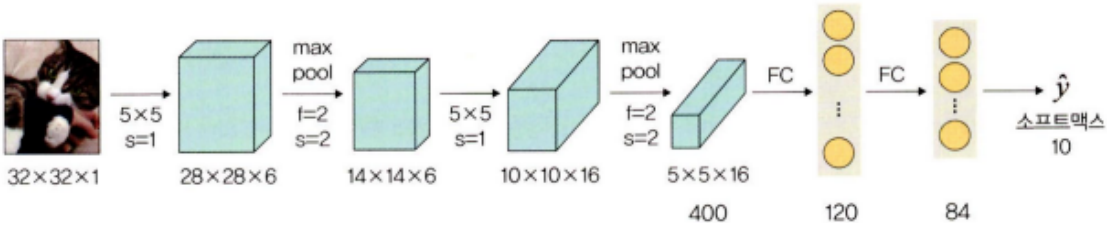
- C : 합성곱층
- S : 풀링층
- F : 완전연결층

❯ 그림 6-1 LeNet-5



LeNet-5를 사용하는 예제) 개와 고양이 데이터셋 사용. 우리가 구현할 신경망은 다음 그림과 같다.

❯ 그림 6-2 LeNet-5 예제 신경망



32\times32 크기의 이미지에 합성곱층과 최대 풀링층이 쌍으로 두 번 적용된 후 완전연결층을 거쳐 이미지가 분류되는 신경망이다.

❯ 표 6-1 LeNet-5 예제 신경망 상세

계층 유형	특성 맵	크기	커널 크기	스트라이드	활성화 함수
이미지	1	32×32	–	–	–
합성곱층	6	28×28	5×5	1	렐루(ReLU)
최대 풀링층	6	14×14	2×2	2	–
합성곱층	16	10×10	5×5	1	렐루(ReLU)
최대 풀링층	16	5×5	2×2	2	–
완전연결층	–	120	–	–	렐루(ReLU)
완전연결층	–	84	–	–	렐루(ReLU)
완전연결층	–	2	–	–	소프트맥스(softmax)

```
pip install --user tqdm
```

`tqdm` : 아랍어로 progress(진행 상태)라고도 한다. 즉, 진행 상태를 바 형태로 가시화하여 보여준다. 주로 모델 훈련에 대한 진행 상태를 확인하고자 할 때 사용한다.

```
import torch
import torchvision
from torch.utils.data import DataLoader, Dataset
from torchvision import transforms # 이미지 변환(전처리) 기능을 제
from torch.autograd import Variable
from torch import optim # 경사 하강법을 이용하여 가중치를 구하기 위한
import torch.nn as nn
import torch.nn.functional as F
import os # 파일 경로에 대한 함수들을 제공
import cv2
from PIL import Image
from tqdm import tqdm_notebook as tqdm # 진행 상황을 가시적으로 표
import random
from matplotlib import pyplot as plt

device=torch.device('cuda:0' if torch.cuda.is_available() els
# 파이토치는 텐서플로와 다르게 GPU를 자동으로 할당해 주지 않기 때문에 GPU
```

데이터셋 전처리(텐서 변환)

```
class ImageTransform():
  def __init__(self, resize, mean, std):
    self.data_transform={
        'train' : transforms.Compose([
            transforms.RandomResizedCrop(resize, scale=(0.5,
            transforms.RandomHorizontalFlip(),
            transforms.ToTensor(),
            transforms.Normalize(mean, std)
        ]),
        'val' : transforms.Compose([
            transforms.Resize(256),
            transforms.CenterCrop(resize),
            transforms.ToTensor(),
            transforms.Normalize(mean, std)
        ])
```

```
    }
  def __call__(self, img, phase):
    return self.data_transform[phase](img)
```

`from torchvision import transforms` ➡️ 이미지 변환(전처리) 기능을 제공하는 라이브러리

```
train_transforms=transforms.Compose([
    transforms.RandomResizedCrop(resize, scale=(0.5, 1.0)),
    transforms.RandomHorizontalFlip(),
    transforms.ToTensor(),
    transforms.Normalize(mean, std)
]
)
```

➡️ `transforms.Compose` : 이미지를 변형할 수 있는 방식들의 묶음

➡️ `transforms.RandomResizedCrop` : 입력 이미지를 주어진 크기(`resize` : 224×224)로 조정한다.

`scale` : 원래 이미지를 임의의 크기(0.5~1.0)만큼 무작위로 자르겠다.

➡️ `transforms.RandomHorizontalFlip` : 주어진 확률로 이미지를 수평 반전시킨다. 이때 확률값을 지정하지 않았으므로 기본값인 0.5의 확률로 이미지들이 수평 반전된다. 즉, 훈련 이미지 중 반은 위아래 뒤집힌 상태로, 반은 그대로 사용한다.

➡️ `transforms.ToTensor` : `ImageFolder` 메서드를 비롯해서 `torchvision` 메서드는 이미지를 읽을 때 파이썬 이미지 라이브러리인 `PIL`을 사용한다. `PIL`을 사용해서 이미지를 읽으면 생성되는 이미지는 범위가 $[0, 255]$이며, 배열의 차원이 (높이H $\times$ 너비W $\times$ 채널수C)로 표현된다. 이후 효율적인 연산을 위해 `torch.FloatTensor` 배열로 바꾸어야 하는데, 이때 픽셀 값의 범위는 $[0.0, 1.0]$ 사이가 되고 차원의 순서도 (채널수C $\times$ 높이H $\times$ 너비W)로 바뀐다. 그리고 이러한 작업을 수행해 주는 메서드가 `ToTensor()`이다.

➡️ `transforms.Normalize` : 전이 학습에서 사용하는 사전 훈련된 모델들은 대개 ImageNet 데이터셋에서 훈련되었다. 따라서 사전 훈련된 모델을 사용하기 위해서는 ImageNet 데이터의 각 채널별 평균과 표준편차에 맞는 ✅**정규화(normalize)**를 해주어야 한다. 즉, `Normalize` 메서드 안에 사용된 (mean : 0.486, 0.456, 0.406), (std : 0.229, 0.224, 0.225)는 ImageNet에서 이미지들의 RGB 채널마다 평균과 표준편차를 의미한다. 참고로 OpenCV를 사용해서 이미지를 읽어 온다면 RGB 이미지가 아닌 BGR 이미지이므로 채널 순서에 주의해야 한다.

✅ 정규화 : 데이터 범위를 사용자가 원하는 범위로 제한하는 것을 의미한다. 예를 들어 이미지 데이터를 0~1.0 사이의 값을 갖도록 하는 것이 정규화이다. (8장 참고)

```
def __call__(self, img, phase):
    return self.data_transform[phase](img)
```

✅ `__init__` : 인스턴스 초기화를 위해 사용하는 메서드이다.

➡️ `__call__` : 클래스를 호출할 수 있도록 하는 메서드이다. 인스턴스가 호출되었을 때 실행된다. 즉, 클래스에 `__call__` 함수가 있을 경우 클래스 객체 자체를 호출하면 `__call__` 함수의 리턴(`return`) 값이 반환된다.

이미지가 위치한 디렉터리에서 데이터를 불러온 후 훈련용으로 400개의 이미지, 검증용으로 92개의 이미지, 테스트용으로 10 개의 이미지를 사용한다.

예제를 진행하기 위한 시스템 성능이 좋을 경우 훈련용 2만 개를 사용하면 모델 성능이 더 높아진다. 예제 파일의 주석(#)을 해제하고 실행하면 된다. 단 해제하고 실행한다면 그다음 두 줄은 주석 처리해야 한다.

1. **고양이 이미지 데이터 가져오기**

```
cat_images_filepaths=sorted([os.path.join(cat_directory, f) f
```

➡️ `sorted` : 데이터를 정렬된 리스트로 만들어서 반환한다.

➡️ `os.path.join` : 경로와 파일명을 결합하거나 분할된 경로를 하나로 합치고 싶을 때 사용한다. 즉, `cat_directory` 디렉터리와 `os.listdir`을 통해 검색된 이미지 파일들(`f`)을 하나로 합쳐서 표시한다.

✅ 다음과 같이 경로를 하나로 합칠 수 있다.

```
import os
list_path=['C://', 'Temp', 'user']
folder_path=os.path.join(*list_path)
folder_path
```

실행 결과

```
'C://Temp//user'
```

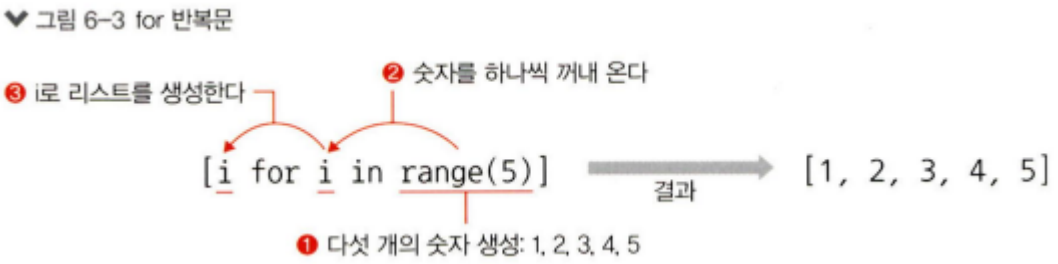
✅ 윈도 환경에서는 경로가 '\\'로 표시된다.

➡️ `os.listdir` : 지정한 디렉터리 내 모든 파일의 리스트를 반환한다. 예제에서 사용하는 Cat 디렉터리의 이미지 파일드을 모두 반환한다.

2. **`images_filepaths`에서 이미지 파일들을 불러온다.**

```
correct_images_filepaths=[i for i in images_filepaths if cv2.
```

➡️ `for` 반복문을 이용하여 가져온 데이터에 대해 i를 이용하여 리스트로 만든다. 즉, 그림 6-3과 같은 의미를 갖는다.



➡️ 반복문(`for`)을 이용하여 `images_filepaths`에서 이미지 데이터를 검색한다.

➡️ 조건문(`if`)를 의미한다. `cv2.imread()` 함수(`cv2.imread(i))` 를 이용하여 모든 이미지 데이터를 읽어 온다(`not None` 상태, 즉 '더 이상 데이터를 찾을 수 없을 때까지'를 의미한다.)

✅ 넘파이 `random()` 함수는 임의의 난수를 생성하는데, 이때 난수를 생성하기 위해 사용하는 것이 시드 값(seed value)이다. 또한, `Numpy.random.seed()` 메서드는 상태를 초기화한다. 즉, 이 모듈이 호출될 때마다 임의의 난수가 재생성된다. 하지만 특정 시드 값을 부여하면 상태가 저장되기 때문에 동일한 난수를 생성한다.

예1) 시드 값을 101로 설정했을 때

```
import numpy as np

np.random.seed(101)
np.random.randint(low=1, high=10, size=10)
```

결과

```
array([2, 7, 8, 9, 5, 9, 6, 1, 6, 9])
```

예2) 시드 값을 100으로 설정했을 때

```
np.random.seed(100)
np.random.randint(low=1, high=10, size=10)
```

결과

```
array([9, 9, 4, 8, 8, 1, 5, 3, 6, 3])
```

예3) 또다시 시드 값을 101로 설정했을 때 결과

```
array([2, 7, 8, 9, 5, 9, 6, 1, 6, 9])
```

주어진 데이터셋을 훈련, 검증, 테스트 용도로 분리했는데, 테스트 용도의 데이터셋에 어떤 데이터들이 있는지 확인해 보자.

```
def display_image_grid(images_filepaths, predicted_labels=(),
  rows=len(images_filepaths) // cols
  figure, ax=plt.subplots(nrows=rows, ncols=cols, figsize=(12
  for i, image_filepath in enumerate(images_filepaths):
    image=cv2.imread(image_filepath)
    image=cv2.cvtColor(image, cv2.COLOR_BGR2RGB)
    true_label=os.path.normpath(image_filepath).split(os.sep)
    predicted_label=predicted_labels[i] if predicted_labels e
    color='green' if true_label==predicted_label else 'red'
    ax.ravel()[i].imshow(image)
    ax.ravel()[i].set_title(predicted_label, color=color)
    ax.ravel()[i].set_axis_off()
  plt.tight_layout()
  plt.show()
```

➡️ `cv2.cvtColor` : 이미지의 색상을 변경하기 위해 사용된다.

```
cv2.cvtColor(image, cv2.COLOR_BGR2RGB)
```

✅ `image` : 입력 이미지

✅ `cv2.COLOR_BGR2RGB` : 변환할 이미지의 색상을 지정하는 것으로 BGR(Blue, Green, Red) 채널 이미지를 RGB(컬러)로 변경하겠다는 의미이다.

이미지 전체 경로를 정규화하고 분할을 위한 코드

```
true_label=os.path.normpath(image_filepath).split(os.sep)[-2]
```

➡️ `os.path.normpath` : 경로명 정규화.

✅ A//B, A/B/, A/./B, A//../B = A/B

➡️ `split(os.sep)` : 경로를 / 혹은 \를 기준으로 분할할 때 사용한다.

✅ c:/temp/user/a.jpg라는 경로가 있을 때 `split(os.sep)`을 적용하면 ['c:', 'temp', 'user', 'a.jpg']처럼 분할된다. 또한, `split(os.sep)[-2]`를 적용하면 'user'를 반환할 것이다.

➡️ `predicted_label`에 대한 값을 정의

```
predicted_label=predicted_labels[i] if predicted_labels else
```

✅ `predicted_labels` 값이 있으면 그 값을 `predicted_label`로 사용한다.

✅ `predicted_labels` 값이 없으면 `true_label` 값을 `predicted_label`로 사용한다.

앞에서 정의한 함수를 호출하여 10개의 이미지를 호출한다.

```
display_image_grid(test_images_filepaths)
```

출력 결과)

모델 학습을 위한 단계

1. 데이터셋에는 학습할 데이터의 경로를 정의하고 그 경로에서 데이터를 읽어 온다.
2. 데이터셋 크기가 클 수 있으므로 `__init__`에서 전체 데이터를 읽어 오는 것이 아니라 경로만 저장해 놓고, `__getitem__` 메서드에서 이미지를 읽어 온다. 즉, 데이터를 어디에서 가져올지 결정한다.
3. 데이터로더에서 데이터셋의 데이터를 메모리로 불러오는데, 한꺼번에 전체 데이터를 불러오는 것이 아니라 배치 크기만큼 분할하여 가져온다.

`DogvsCatDataset()` 클래스는 데이터를 불러오는 방법을 정의한다.

⭐ 이번 예제의 목적 : 다수의 개와 고양이 이미지가 포함된 데이터에서 이들을 예측하는 것이다. 따라서 레이블(정답) 이미지에서 고양이와 개가 포함될 확률을 코드로 구현한다. 예를 들어 고양이가 있는 이미지의 레이블은 0, 개가 있는 이미지의 레이블은 1이 되도록 코드를 구현한다.

```
class DogvsCatDataset(Dataset):
  def __init__(self, file_list, transform=None, phase='train'
    self.file_list=file_list
    self.transform=transform # DogvsCatDataset 클래스를 호출할 따
    self.phase=phase # 'train' 적용

  def __len__(self): # images_filepaths 데이터셋의 전체 길이를 반환
    return len(self.file_list)

  def __getitem__(self, idx):
    img_path=self.file_list[idx]
    img=Image.open(img_path)
    img_transformed=self.transform(img, self.phase)
    label=img_path.split('/')[-1].split(',')[0]
    if label=='dog':
      label=1
    elif label=='cat':
      label=0
    return img_transformed, label
```

1️⃣ **이미지 데이터에 대한 레이블 값(dog, cat)을 가져온다.**

```
label=img_path.split('/')[-1].split('.')[0]
```

➡️ `img_path` : 이미지가 위치한 전체 경로를 보여 준다. 이때 이미지 데이터의 레이블(dog)을 가져오려면 '/'와 '.'을 제거해야 한다.

- ✅ `img_path.split('/')[-1]` : 이미지의 전체 경로에서 '/'를 제거한다.
- ✅ `split('.')[0]` : 마지막으로 `img_path.split('/')[-1]` 을 통해 얻은 결과에서 '.'을 제거한다.

2️⃣ 전처리에서 사용할 변수에 대한 값을 정의한다.

```
size=224
mean=(0.485, 0.456, 0.406)
std=(0.229, 0.224, 0.225)
batch_size=32
```

3️⃣ **훈련과 검증 용도의 데이터셋 정의** : 앞에서 정의한 `DogvsCatDataset()` 클래스를 이용하여 훈련과 검증 데이터셋을 준비하되 전처리도 함께 적용하도록 한다.

```
train_dataset=DogvsCatDataset(train_images_filepaths, transfo
                              phase='train') # 훈련 이미지에 tra

val_dataset=DogvsCatDataset(val_images_filepaths, transform=I
                            phase='val') # 검증 이미지에 test_tr

index=0
print(train_dataset.__getitem__(index)[0].size()) # 훈련 데이터
print(train_dataset.__getitem__(index)[1]) # 훈련 데이터의 레이블
```

결과

```
torch.Size([3, 224, 224])
0
```

이미지는 컬러 상태에서 224\times224 크기를 가지며 레이블이 1로 출력되었다. 즉, 훈련 데이터셋의 레이블(`train_dataset.__getitem__(index)[1]`)이 1 값을 갖기 때문에 '개'라는 이미지가 포함되어 있다는 것을 유추해 볼 수 있다.

[딥러닝 파이토치 교과서] 6장 합성곱 신경망2-2

작성일시 @2024년 11월 4일 오후 10:23

4️⃣ **데이터로더 정의** : 전처리와 함께 데이터셋을 정의했기 때문에 메모리로 불러와서 훈련을 위한 준비를 한다.

```
train_dataloader=DataLoader(train_dataset, batch_size=batch_s
val_dataloader=DataLoader(val_dataset, batch_size=batch_size,
dataloader_dict={'train' : train_dataloader, 'val' : val_data

batch_iterator=iter(train_dataloader)
inputs, label=next(batch_iterator)
print(inputs.size())
print(label)
```

결과

```
torch.Size([32, 3, 224, 224])
tensor([1, 0, 1, 1, 0, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 0, 0, 0,
        0, 0, 1, 1, 1, 1, 0, 0])
```

파이토치의 데이터로더는 배치 관리를 담당한다. 한 번에 모든 데이터를 불러오면 메모리에 부담을 줄 수 있기 때문에 데이터를 그룹으로 쪼개서 조금씩 불러온다.

```
train_dataloader=DataLoader(train_dataset, batch_size=batch_s
```

➡️ `train_dataset` : 데이터를 불러오기 위한 데이터셋

➡️ `batch_size` : 한 번에 메모리로 불러올 데이터 크기. 여기에서는 32개씩 가져온다.

➡️ `shuffle` : 메모리로 데이터를 가져올 때 임의로 섞어서 가져오도록 한다.

5. 모델의 네트워크를 설계하기 위한 클래스 생성