6장 | 합성곱 신경망 II (1)

WEEK 7주차

6.1 이미지 분류를 위한 신경망

입력 데이터로 **이미지**를 사용한 분류 : 특정 대상이 **영상 내에 존재**하는지 여부를 판단하는 것

6.1.1 LeNet-5

- **합성곱 신경망**이라는 개념을 **최초**로 개발한 구조
- 1995년 얀 르쿤, 레옹 보토, 요슈아 벤지오, 패트릭 하프너
- 수표에 쓴 손글씨 숫자를 인식하는 딥러닝 구조
- CNN의 초석
- **합성곱**(convolutional)과 **다운 샘플링**(sub-sampling) (혹은 풀링) 을 **반복적**으로 거치면서 마지막에 **완전연결층**에서 **분류 수행**

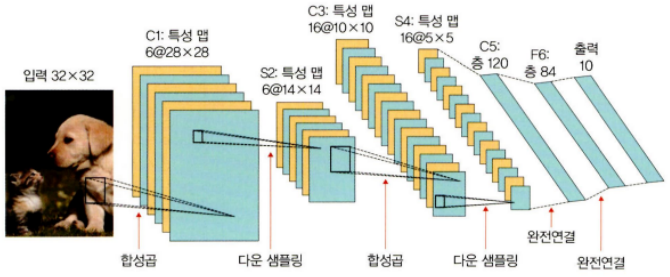

그림 6-1 LeNet-5 | C: 합성곱층 | S: 풀링층 | F: 완전연결층

C1에서 5x5 합성곱 연산 후 28x28 크기의 특성 맵 6개 생성

S2에서 다운 샘플링하여 특성 맵 크기를 14x14로 줄임

C3에서 5x5 합성곱 연산하여 10x10 크기의 특성 맵 16개 생성

S4에서 다운 샘플링하여 특성 맵 크기를 5x5로 줄임

C5에서 5x5 합성곱 연산하여 1x1 크기의 특성 맵 120개 생성

마지막으로 F6에서 완전연결층으로 C5의 결과를 유닛(또는 노드) 84개에 연결

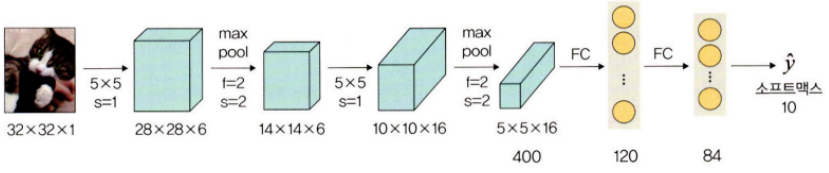

그림 6-2 LeNet-5 예제 신경망

32x32 크기의 이미지에 합성곱층과 최대 풀링층이 쌍으로 두 번 적용된 후 완전 연결층을 거쳐 이미지 분류되는 신경망

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)

그림 6-3 LeNet-5 예제 신경망 상세

```
# 6-2 이미지 데이터셋 전처리
class ImageTransform():
    def __init__(self, resize, mean, std):
        self.data_transform = {
            'train': transforms.Compose([
                transforms.RandomResizedCrop(resize, scale=(0.5, 1.0)),
                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)
```

1. 토치비전 라이브러리를 이용하여 손쉬운 이미지 전처리
    a. `transforms.Compose` : 이미지를 변형할 수 있는 방식들의 묶음
    b. `transforms.RandomResizedCrop` : 입력 이미지를 주어진 크기로 조정, `scale`로 원래 이미지를 임의의 크기만큼 면적을 무작위로 자름
    c. `transforms.RandomHorizontalFlip` : 주어진 확률로 이미지를 수평 반전
    d. `transforms.ToTensor` : `ImageFolder` 메서드를 비롯해서 `torchvision` 메서드는 이미지를 읽을 때 파이썬 이미지 라이브러리인 PIL 사용 → 생성 이미지 범위 [0.255], 배열의 차원 (높이H x 너비W x 채널수C)로 표현 → 효율적인 연산을 위해 `ToTensor()` 메서드를 사용하여 `torch.FloatTensor` 배열로 바꾸어야 하는데, 이때 픽셀 값의 범위가 [0.0, 1.0] 사이가 되고, 차원의 순서도 (C x H x W)로 바뀜
    e. `transforms.Normalize` : 사전 훈련된 ㅁ델을 사용하기 위해 각 채널별 평균과 표준편차에 맞는 정규화 수행
2. 클래스 호출하는 메서드 `__call__` 사용

    `__init__` 은 인스턴스 초기화를 위해 사용한다면, `__call__` 는 인스턴스가 호출되었을 때 실행

→ 클래스에 `__call__` 함수 있을 경우 객체 자체를 호출하면 `__call__` 함수의 리턴 값이 반환됨

```
# 6-9 데이터로더 정의
train_dataloader = DataLoader(train_dataset, batch_size=batch_size, shuffle
=True)
val_dataloader = DataLoader(val_dataset, batch_size=batch_size, shuffle=Fal
se)
dataloader_dict = {'train': train_dataloader, 'val': val_dataloader} # 훈련
데이터셋과 검증 데이터셋을 합쳐서 표현

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

- `train_dataloader = DataLoader(train_dataset, batch_size=batch_size, shuffle=True)`
  a. 첫 번째 파라미터 : 데이터를 불러오기 위한 데이터셋
  b. `batch_size` : 한 번에 메모리로 불러올 데이터 크기

```
# 6-10 모델의 네트워크 클래스
class LeNet(nn.Module):
    def __init__(self):
        super(LeNet, self).__init__()
        self.cnn1 = nn.Conv2d(in_channels=3, out_channels=16, kernel_size=
5, stride=1, padding=0) # 2D 합성곱층 적용, 이때 입력 형태는 (3,224,224)가 되고 출
력 형태는 (weight-kernel_size+1)/stride에 따라(16,220,220)이 됨
        self.relu1 = nn.ReLU() # ReLU 활성화 함수
        self.maxpool1 = nn.MaxPool2d(kernel_size=2) # 최대 풀링 적용, 적용 이후
출력 형태는 220/2가 되어 (16,110,110)
        self.cnn2 = nn.Conv2d(in_channels=16, out_channels=32, kernel_size=
5, stride=1, padding=0) # 또다시 2D 합성곱층 적용, 출력 형태는 (32,106,106)
        self.relu2 = nn.ReLU() # activation
        self.maxpool2 = nn.MaxPool2d(kernel_size=2) # 최대 풀링 적용, 출력 형태
는 (32,53,53)

        self.fc1 = nn.Linear(32*53*53, 512)
        self.relu5 = nn.ReLU()
        self.fc2 = nn.Linear(512, 2)
        self.output = nn.Softmax(dim=1)

    def forward(self, x):
        out = self.cnn1(x)
        out = self.relu1(out)
        out = self.maxpool1(out)
```

```
        out = self.cnn2(out)
        out = self.relu2(out)
        out = self.maxpool2(out)
        out = out.view(out.size(0), -1) # 완전연결층에 데이터를 전달하기 위해 데이터 형태를 1차원으로 변경
        out = self.fc1(out)
        out = self.fc2(out)
        out = self.output(out)
        return out
```

- **Conv2d 계층에서의 출력 크기 구하는 공식**

  출력 크기 = $(W - F + 2)/S + 1$

  - W (`input_volume_size`) : 입력 데이터의 크기
  - F (`kernel_size`) : 커널 크기
  - P (`padding_size`) : 패딩 크기
  - S (`strides`) : 스트라이드

- **MaxPool2d 계층에서의 출력 크기 구하는 공식**

  출력 크기 = IF/F

  - IF (`input_filter_size` , 또한 바로 앞의 Conv2d의 출력 크기이도 함) : 입력 필터의 크기
  - F (`kernel_size`) : 커널 크기

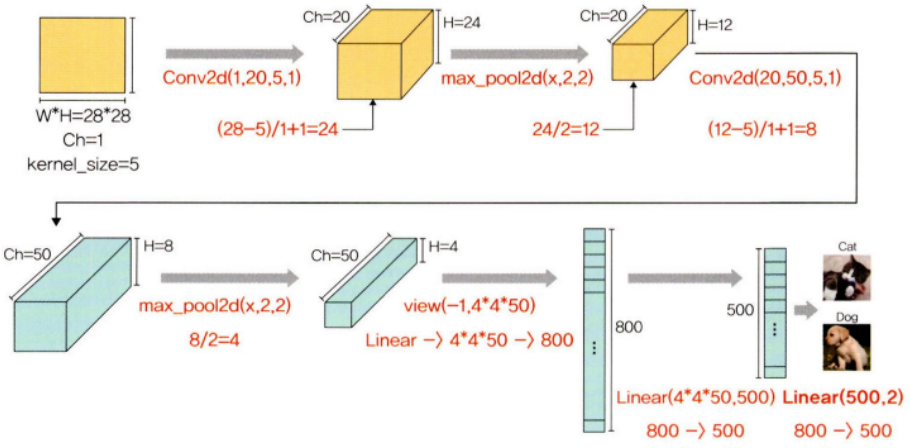

그림 6-5 합성곱층과 풀링층의 크기(형태) 계산

```
# 6-12 torchsummary 라이브러리를 이용한 모델의 네트워크 구조 확인
from torchsummary import summary
summary(model, input_size=(3, 224, 224))
```

- summary를 통해 모델의 네트워크 관련 정보 확인

  `summary(model, input_size=(3,224,224))`

  a. 첫 번째 파라미터: 모델의 네트워크

  b. 두 번째 파라미터: 입력 값으로 (채널, 너비, 높이)가 됨

```
----------------------------------------------------------------
        Layer (type)               Output Shape         Param #
================================================================
            Conv2d-1         [-1, 16, 220, 220]           1,216
              ReLU-2         [-1, 16, 220, 220]               0
         MaxPool2d-3         [-1, 16, 110, 110]               0
            Conv2d-4         [-1, 32, 106, 106]          12,832
              ReLU-5         [-1, 32, 106, 106]               0
         MaxPool2d-6           [-1, 32, 53, 53]               0
            Linear-7                 [-1, 512]      46,023,168
            Linear-8                   [-1, 2]           1,026
           Softmax-9                   [-1, 2]               0
================================================================
Total params: 46,038,242
Trainable params: 46,038,242
Non-trainable params: 0
----------------------------------------------------------------
Input size (MB): 0.57
Forward/backward pass size (MB): 19.47
Params size (MB): 175.62
Estimated Total Size (MB): 195.67
----------------------------------------------------------------
```

torchsummary 라이브러리 이용했을 때 모델의 구조
- 총 파라미터 수, 입력 크기, 네트워크의 총 크기 등

```
# 6-14 옵티마이저와 손실 함수 정의
optimizer = optim.SGD(model.parameters(), lr=0.001, momentum=0.9)
criterion = nn.CrossEntropyLoss()
```

- 경사 하강법으로 모멘텀 SGD 사용: SGD에 관성이 추가된 것으로 매번 기울기를 구하지만 가중치를 수정하기 전에 이전 수정 방향(+,-)을 참고하여 같은 방향으로 일정한 비율만 수정되게 하는 방법

  `optim.SGD(model.parameters(), lr=0.001, momentum=0.9)`

  a. 첫 번째 파라미터: 경사 하강법을 통해 궁극적으로 업데이트하고자 하는 파라미터는 가중치와 바이어스

  b. `lr`(learning rate): 가중치를 변경할 때 얼마나 크게 변경할지 결정

  c. `momentum`: SGD를 적절한 방향으로 가속화하며 흔들림(진동)을 줄여주는 매개변수

```
# 6-16 모델 학습 함수 정의
def train_model(model, dataloader_dict, criterion, optimizer, num_epoch):
    since = time.time()
    best_acc = 0.0

    for epoch in range(num_epoch): # epoch를 10으로 설정했으므로 10회 반복
        print('Epoch {}/{}'.format(epoch + 1, num_epoch))
        print('-'*20)

        for phase in ['train', 'val']:
            if phase == 'train':
                model.train() # 모델을 학습시키겠다는 의미
            else:
```

```
                model.eval()

            epoch_loss = 0.0
            epoch_corrects = 0

            for inputs, labels in tqdm(dataloader_dict[phase]): # 여기서 dat
aloader_dict는 훈련 데이터셋(train_loader)을 의미
                inputs = inputs.to(device) # 훈련 데이터셋을 CPU에 할당
                labels = labels.to(device)
                optimizer.zero_grad() # 역전파 단계를 실행하기 전에 기울기를 0으로
초기화

                with torch.set_grad_enabled(phase == 'train'):
                    outputs = model(inputs)
                    _, preds = torch.max(outputs, 1)
                    loss = criterion(outputs, labels) # 손실 함수를 이용한 오차
계산

                    if phase == 'train':
                        loss.backward() # 모델의 학습 가능한 모든 파라미터에 대해
기울기 계산
                        optimizer.step() # optimizer의 step 함수를 호출하면 파
라미터를 갱신

                    epoch_loss += loss.item() * inputs.size(0)
                    epoch_corrects += torch.sum(preds == labels.data) # 정답
과 예측 일치하면 합계를 epoch_corrects에 저장

            epoch_loss = epoch_loss / len(dataloader_dict[phase].dataset) #
최종 오차 계산(오차를 데이터셋의 길이(개수)로 나누어서 계산)
            epoch_acc = epoch_corrects.double() / len(dataloader_dict[phas
e].dataset) # 최종 정확도(epoch_corrects를 데이터셋의 길이(개수)로 나누어서 계산)

            print('{} Loss: {:.4f} Acc: {:.4f}'.format(phase, epoch_loss, e
poch_acc))

            if phase == 'val' and epoch_acc > best_acc: # 검증 데이터셋에 대한
가장 최적의 정확도를 저장
                best_acc = epoch_acc
                best_model_wts = model.state_dict()

    time_elapsed = time.time() - since
    print('Training complete in {:.0f}m {:.0f}s'.format(
        time_elapsed // 60, time_elapsed % 60))
    print('Best val Acc: {:4f}'.format(best_acc))
    return model
```

- `epoch_loss += loss.item() * inputs.size(0)`

- `reduction` 을 따로 명시하지 않은 이유: 기본값을 그대로 사용하기 때문

  ```
  criterion = nn.CrossEntropyLoss(reduction='mean')
  ```

  `reduction` 파라미터 기본값은 `mean` : 정답과 예측 값의 오차를 구한 후 그 값들의 평균 반환

  → 손실 함수 특성상 전체 오차를 배치 크기로 나눔으로써 평균을 반환하기 때문에 `epoch_loss` 계산하는 동안 `loss.item()`과 `inputs.size(0)` 을 곱해 줌

```
# 6-18 모델 테스트를 위한 함수 정의
import pandas as pd

id_list = []
pred_list = []
_id=0
with torch.no_grad(): # 역전파 중 텐서들에 대한 변화도를 계산할 필요가 없음을 나타내는 것으로, 훈련 데이터셋의 모델 학습과 가장 큰 차이점
    for test_path in tqdm(test_images_filepaths): # 테스트 데이터셋 이용
        img = Image.open(test_path)
        _id =test_path.split('/')[-1].split('.')[1]
        transform = ImageTransform(size, mean, std)
        img = transform(img, phase='val') # 테스트 데이터셋 전처리 적용
        img = img.unsqueeze(0)
        img = img.to(device)

        model.eval()
        outputs = model(img)
        preds = F.softmax(outputs, dim=1)[:, 1].tolist()
        id_list.append(_id)
        pred_list.append(preds[0])

res = pd.DataFrame({
    'id': id_list,
    'label': pred_list
}) # 테스트 데이터셋의 예측결과인 id와 레이블(label)을 데이터 프레임에 저장

res.sort_values(by='id', inplace=True)
res.reset_index(drop=True, inplace=True)

res.to_csv('LesNet.csv', index=False) # 데이터 프레임을 CSV 파일로 저장
```

1. `torch.unsqueeze` : 텐서에 차원 추가할 때 사용, (0)은 차원 추가될 위치를 의미
2. 소프트맥스(softmax)는 지정된 차원(dim)을 따라 텐서의 요소가 (0,1) 범위에 있고 합계가 1이 되도록 크기를 다시 조정

   `F.softmax(outputs, dim=1)[:, 1].tolist()`

a. `(outputs, dim=1)` : `outputs`에 `softmax`를 적용하여 각 행의 합이 1이 되도록 함

b. `[:, 1]` : a의 값 중 모든 행(:)에서 두 번째 칼럼(1번째 인덱스)을 가져옴

c. `tolist()` : 배열을 리스트 형태로 변환

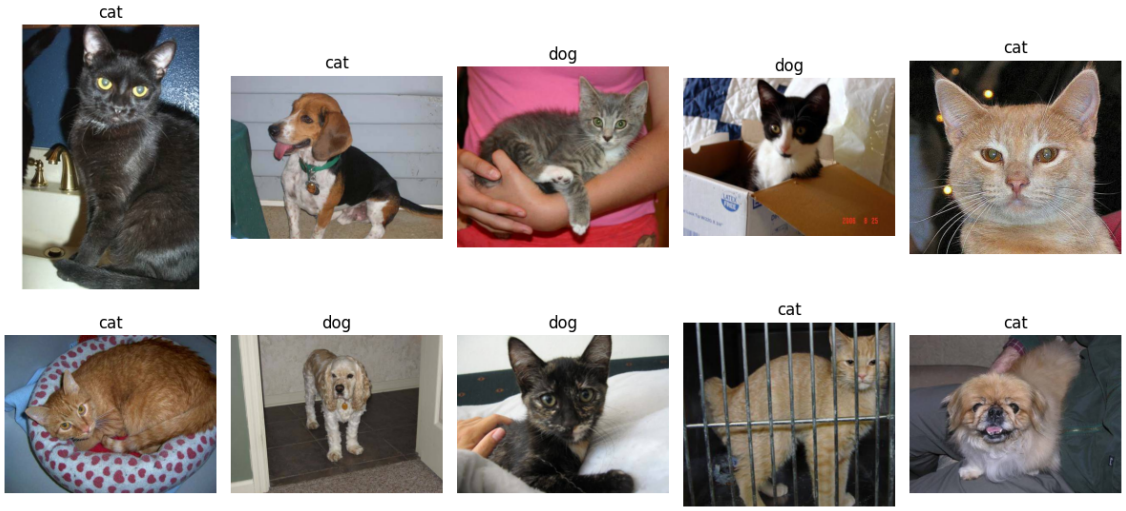

그림 6-7 테스트 데이터셋에 대한 LeNet 모델의 예측 결과

→ 예측력이 좋지 않음 - 극히 일부의 데이터를 이용한 모델 학습을 진행했기 때문

6.1.2 AlexNet

- AlexNet 세부 블록 이해를 위한 CNN 구조 복습

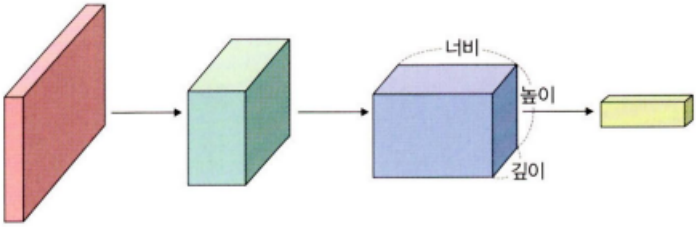

그림 6-8 CNN 구조

CNN은 이미지를 다루기 때문에 기본적으로 3차원 데이터를 다루고, 따라서 3차원 구조를 가짐

→ 너비(width), 높이(height), 깊이(depth)

보통 R/G/B 성분 세 개를 갖기 때문에 시작이 3이지만, 합성곱을 거치면서 특성 맵이 만들어지고 이것에 따라 중간 영상의 깊이가 달라짐

- **AlexNet 구조**

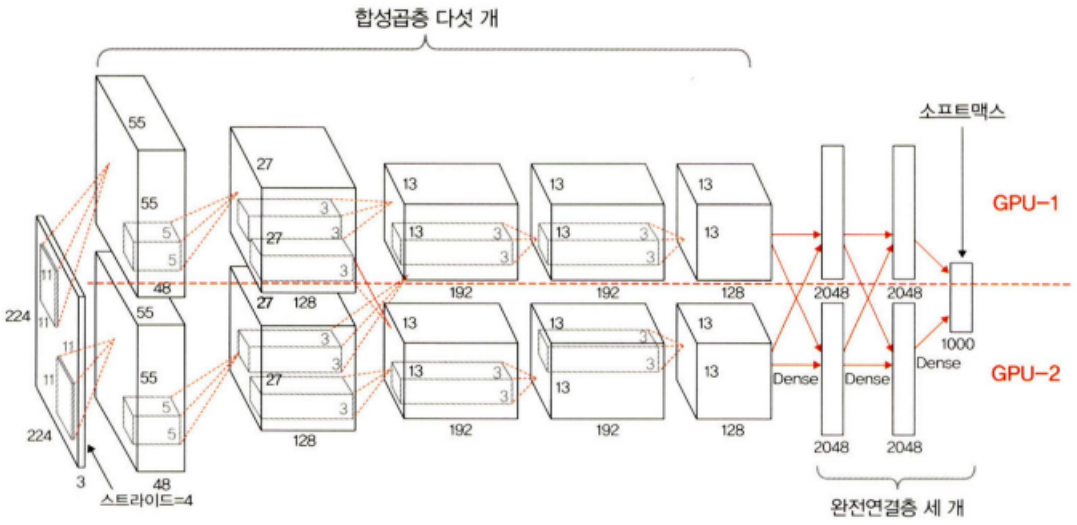

그림 6-9 AlexNet 구조

- **합성곱층 총 5개**
  - 합성곱층에서 사용된 활성화 함수는 렐루(ReLU)
- **완전연결층 3개**
  - 맨 마지막 완전연결층은 카테고리 1000개를 분류하기 위해 **소프트맥스 활성화 함수** 사용
- 전체적으로 보면 GPU 두 개를 기반으로 한 병렬 구조인 점을 제외하면 LeNet-5와 **크게 다르지 X**

계층 유형	특성 맵	크기	커널 크기	스트라이드	활성화 함수
이미지	1	227×227	–	–	–
합성곱층	96	55×55	11×11	4	렐루(ReLU)
최대 풀링층	96	27×27	3×3	2	–
합성곱층	256	27×27	5×5	1	렐루(ReLU)
최대 풀링층	256	13×13	3×3	2	–
합성곱층	384	13×13	3×3	1	렐루(ReLU)
합성곱층	384	13×13	3×3	1	렐루(ReLU)
합성곱층	256	13×13	3×3	1	렐루(ReLU)
최대 풀링층	256	6×6	3×3	2	–
완전연결층	–	4096	–	–	렐루(ReLU)
완전연결층	–	4096	–	–	렐루(ReLU)
완전연결층	–	1000	–	–	소프트맥스(softmax)

표 6-2 AlexNet 구조 상세

- 네트워크에 있는 학습 가능한 변수: 총 6600만 개
- 네트워크에 대한 입력: 227x227×3 크기의 RGB 이미지
- 네트워크에 대한 출력: 각 클래스(혹은 카테고리)에 해당하는 1000x1 확률 벡터
- 첫 번째 합성곱층 커널의 크기: 11x11×3
  - 스트라이드 4, 특성 맵 96개 생성 → 출력: 55x55×96

- 첫 번째 계층을 거치면서 GPU-1에서는 주로 컬러와 상관없는 정보를 추출하기 위한 커널이 학습되고, GPU-2에서는 주로 컬러와 관련된 정보를 추출하기 위한 커널이 학습

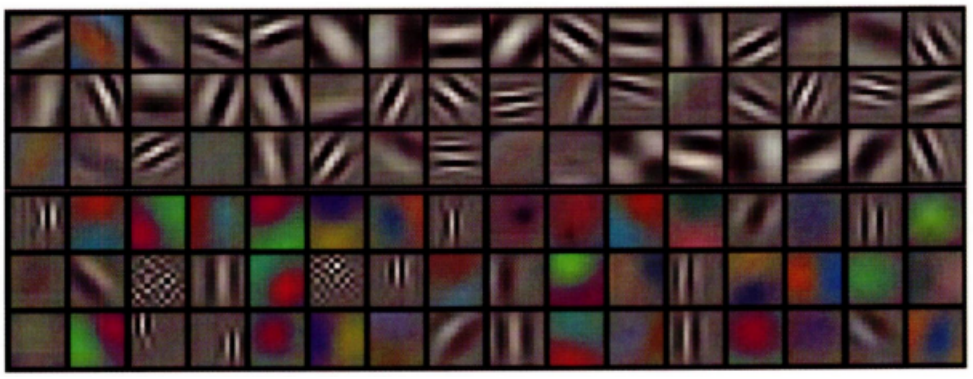

그림 6-10 AlexNet GPU-1,2 적용 결과

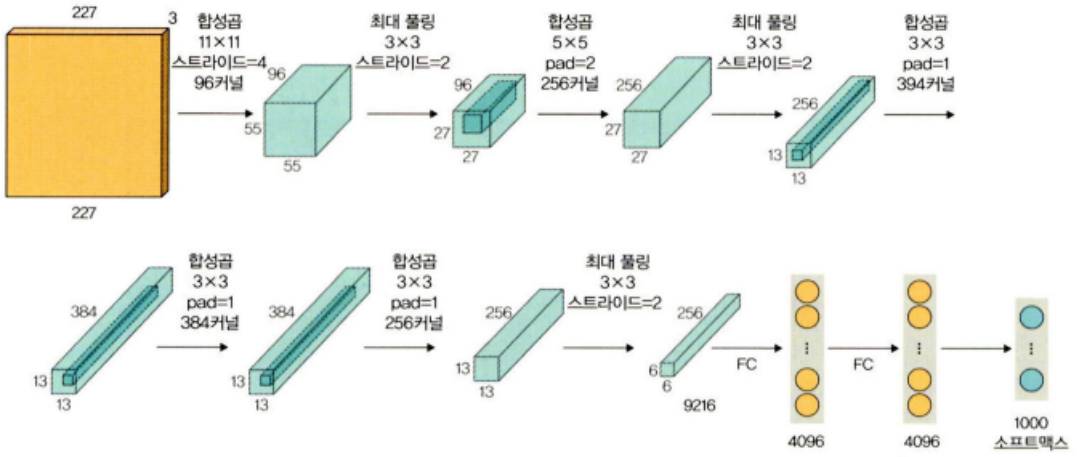

그림 6-11 AlexNet 예제 네트워크

AlexNet은 파라미터를 6000만 개 사용하는 모델이므로, 충분한 데이터가 없으면 과적합이 발생하는 등 테스트 데이터에 대한 성능이 좋지 않음

우리가 사용할 예제에서는 데이터셋을 상당히 제한하여 사용하고 있기 때문에 성능은 좋지 않음

```
# 6-29 AlexNet 모델 네트워크 정의
class AlexNet(nn.Module):
    def __init__(self) -> None:
        super(AlexNet, self).__init__()
        self.features = nn.Sequential(
            nn.Conv2d(3, 64, kernel_size=11, stride=4, padding=2),
            nn.ReLU(inplace=True),
            nn.MaxPool2d(kernel_size=3, stride=2),
            nn.Conv2d(64, 192, kernel_size=5, padding=2),
            nn.ReLU(inplace=True),
            nn.MaxPool2d(kernel_size=3, stride=2),
            nn.Conv2d(192, 384, kernel_size=3, padding=1),
```

```
            nn.ReLU(inplace=True),
            nn.Conv2d(384, 256, kernel_size=3, padding=1),
            nn.ReLU(inplace=True),
            nn.Conv2d(256, 256, kernel_size=3, padding=1),
            nn.ReLU(inplace=True),
            nn.MaxPool2d(kernel_size=3, stride=2),
        )
        self.avgpool = nn.AdaptiveAvgPool2d((6, 6))
        self.classifier = nn.Sequential(
            nn.Dropout(),
            nn.Linear(256 * 6 * 6, 4096),
            nn.ReLU(inplace=True),
            nn.Dropout(),
            nn.Linear(4096, 512),
            nn.ReLU(inplace=True),
            nn.Linear(512, 2),
        )

    def forward(self, x: torch.Tensor) -> torch.Tensor:
        x = self.features(x)
        x = self.avgpool(x)
        x = torch.flatten(x, 1)
        x = self.classifier(x)
        return x
```

- AlexNet 모델에 대한 네트워크 정의

  : 사전 훈련된 네트워크와 유사하게 정의함

  : (합성곱(Conv2d) + 활성화 함수(ReLU) + 풀링(MaxPool2d))이 5번 반복된 후 2개의 완전연결층과 출력층으로 구성된 네트워크

1. `nn.ReLU(inplace=True)`

   ReLU 활성화 함수에서 `inplace` 의미: 기존 값을 연산 결괏값으로 대체함으로써 기존 값을 무시

2. `self.avgpool = nn.AdaptiveAvgPool2d((6, 6))`

   `nn.AdaptiveAvgPool2d` 는 `nn.AvgPool2d` 처럼 풀링을 위해 사용

   a. `AvgPool2d`

   풀링에 대한 커널 및 스트라이드 크기를 정의해야 동작

   (N, C, H_{in}, W_{in}) 크기의 입력을 (N, C, H_{out}, W_{out}) 크기로 출력

$$H_{out} = \left[\frac{H_{in} + 2 \times \text{padding}[0] - \text{kernel\_size}[0]}{\text{stride}[0]} + 1 \right]$$
$$W_{out} = \left[\frac{W_{in} + 2 \times \text{padding}[1] - \text{kernel\_size}[1]}{\text{stride}[1]} + 1 \right]$$

H_out, W_out 공식

b. `AdaptiveAvgPool2d`

풀링 작업이 끝날 때 필요한 **출력 크기만** 정의

앞의 수식에서 H_{out}, W_{out} 값이 정해졌기 때문에 커널 크기, 스트라이드, 패딩 값을 구할 수 있음

→ `AdaptiveAvgPool2d`를 사용할 경우 출력 크기에 대한 조정이 쉬워짐

참고) 입력 크기에 변동이 있고 CNN 위쪽에 완전연결층을 사용하는 경우 유용함

```
# 6-32 모델 네트워크 구조 확인
from torchsummary import summary
summary(model, input_size=(3, 256, 256))
```

```
----------------------------------------------------------------
        Layer (type)               Output Shape         Param #
================================================================
            Conv2d-1          [-1, 64, 63, 63]          23,296
              ReLU-2          [-1, 64, 63, 63]               0
         MaxPool2d-3          [-1, 64, 31, 31]               0
            Conv2d-4         [-1, 192, 31, 31]         307,392
              ReLU-5         [-1, 192, 31, 31]               0
         MaxPool2d-6         [-1, 192, 15, 15]               0
            Conv2d-7         [-1, 384, 15, 15]         663,936
              ReLU-8         [-1, 384, 15, 15]               0
            Conv2d-9         [-1, 256, 15, 15]         884,992
             ReLU-10         [-1, 256, 15, 15]               0
           Conv2d-11         [-1, 256, 15, 15]         590,080
             ReLU-12         [-1, 256, 15, 15]               0
        MaxPool2d-13           [-1, 256, 7, 7]               0
AdaptiveAvgPool2d-14           [-1, 256, 6, 6]               0
          Dropout-15                [-1, 9216]               0
           Linear-16                [-1, 4096]      37,752,832
             ReLU-17                [-1, 4096]               0
          Dropout-18                [-1, 4096]               0
           Linear-19                 [-1, 512]       2,097,664
             ReLU-20                 [-1, 512]               0
           Linear-21                   [-1, 2]           1,026
================================================================
Total params: 42,321,218
Trainable params: 42,321,218
Non-trainable params: 0
----------------------------------------------------------------
Input size (MB): 0.75
Forward/backward pass size (MB): 10.90
Params size (MB): 161.44
Estimated Total Size (MB): 173.10
----------------------------------------------------------------
```

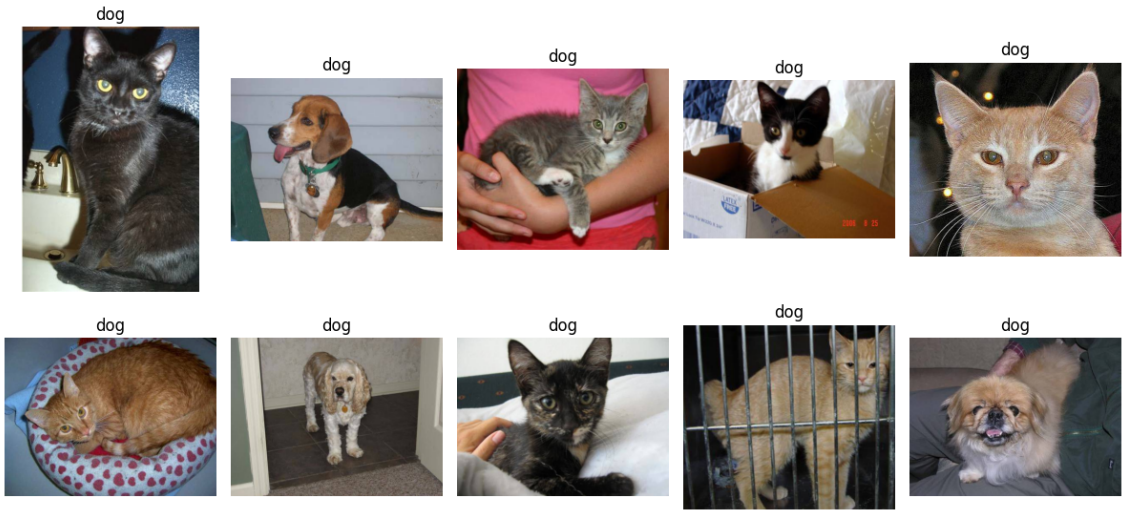

그림 6-12 테스트 데이터셋에 대한 AlexNet 모델의 예측 결과

역시 예측 결과가 좋지 않음 - 데이터셋 제한했기 때문