

Week3_예습과제

164 ~ 198

5. 합성곱 신경망 I

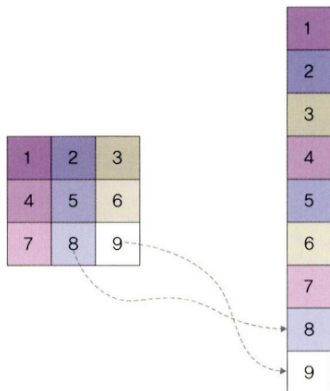
5.1 합성곱 신경망

순전파 : 신경망의 노드 (출력층 → 은닉층 → 입력층)로 오차 전송, GPU(많은 자원 요구)

합성곱 - 이미지 전체 한번에 계산 ❌, 국소적 부분 계산

5.1.1 합성곱층의 필요성

▼ 그림 5-1 합성곱층 원리



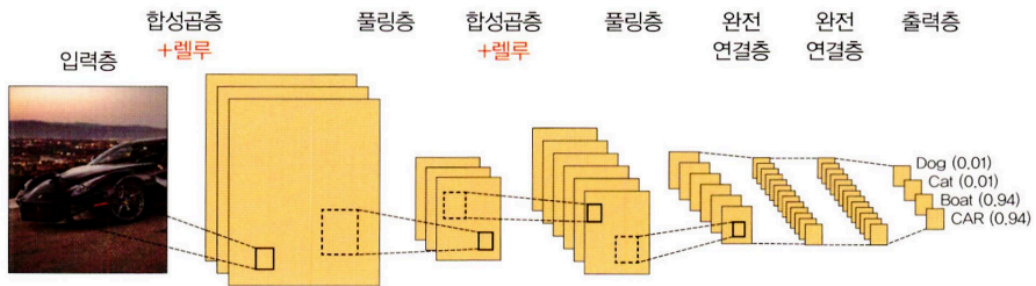
- 3×3 크기의 흑백 이미지가 있다고 가정
- 이미지 분석 - 3x3 배열을 flattening → 각 픽셀에 가중치를 곱함 → 은닉층 전송
: 데이터의 공간적 구조 무시
- 합성곱층 : 데이터의 공간적 구조 살리기 위해 도입

5.1.2 합성곱 신경망 구조

- 합성곱 신경망 (Convolutional Neural Network, CNN 또는 ConvNet)
- 음성인식이나 이미지/영상 인식에서 주로 사용
- 다차원 배열 데이터(컬러 이미지) 처리하도록 구성

✅ 합성곱 신경망의 계층 5개

▼ 그림 5-2 합성곱 신경망 구조



1. 입력층
2. 합성곱층
3. 풀링층
4. 완전 연결층
5. 출력층

→ 합성곱층 + 풀링층을 거치며 주요 feature vector 추출

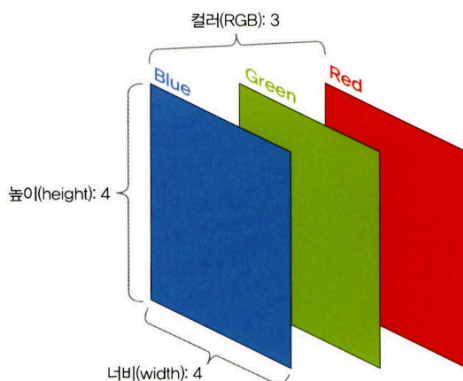
→ 완전 연결층을 거치며 1차원 벡터로 변환

→ 출력층에서 softmax함수(활성화 함수)를 사용해 최종 결과 출력

📌 입력층 (input layer)

- 이미지 = 3차원 데이터 (높이, 너비, 채널)
채널 : gray scale(흑백) = 1, RGB(컬러) = 3

▼ 그림 5-3 채널



📌 합성곱층 (convolutional layer)

- feature extraction 수행

- 커널, 필터 - 이미지에 대한 특성 감지, 특성을 feature map으로 추출
커널 - 3x3, 5x5 크기가 일반적, stride(지정된 간격)에 따라 순차적 이동

✓ stride = 1일 때 이동 과정 예제 - 흑백

1단계. 입력 이미지에 3x3 필터 적용

▼ 그림 5-4 입력 이미지에 3x3 필터 적용



▼ 그림 5-5 입력 이미지에 필터가 1만큼 이동



▼ 그림 5-6 입력 이미지에 필터가 1만큼 두 번째 이동



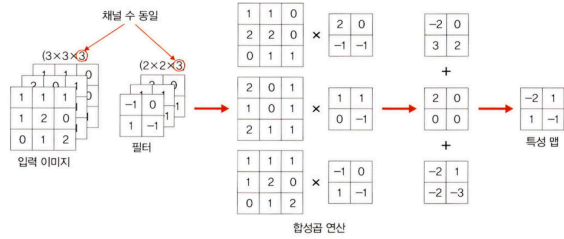
▼ 그림 5-9 입력 이미지에 필터가 1만큼 마지막으로 이동



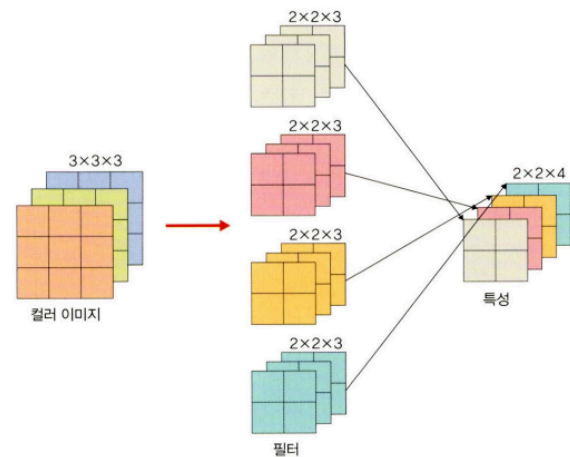
- 같은 위치끼리 입력이미지 X 필터값 곱함 (cf . 입력(3,1)은 필터 원소(3,1)과 곱함)
- stride=1 → 한칸씩 필터를 이동해가며 놓린 위치대로 곱함
- 원본 (6,6,1) → 특성 맵 (4,4,1)
-

✓ stride = 1일 때 이동 과정 예제 - 컬러

▼ 그림 5-10 컬러 이미지 합성곱



▼ 그림 5-11 필터가 2 이상인 합성곱



- 그레이 스케일과 다른 점: 필터채널 = 3, RGB 각각에 서로 다른 가중치로 합성곱 적용
- 채널 = 3 → 필터 개수 1개 / 3개 아님 주의!
- 필터가 2개 이상이면 → 필터 각각이 특성추출 결과의 채널이 됨.

✓ 합성곱층 요약

- 입력 데이터 : $W_1 * H_1 * D_1$ (W_1 : 가로, H_1 : 세로, D_1 : 채널 또는 깊이)
- 하이퍼파라미터
 - 필터 개수 : K
 - 필터 크기 : F
 - 스트라이드 : S
 - 패딩 : P
- 출력 데이터
 - $W_2 = (W_1 - F + 2P) / S + 1$
 - $H_2 = (H_1 - F + 2P) / S + 1$
 - $D_2 = K$

📌 풀링층 (pooling layer)

- 특성 맵의 차원을 다운 샘플링(이미지 축소), 연산량 감소, 주요 feature extraction → 효과적 학습
- 풀링 연산

- 최대 풀링(max pooling) : 대상 영역에서 최댓값 추출
- 평균 풀링(average pooling) : 대상 영역에서 평균 반환,
각 커널값 평균화 → 중요한 가중치 갖는 값 특성 희미해질 수 있음.

✓ 최대 풀링 과정 예제 (stride = 2)

▼ 그림 5-13 첫 번째 최대 풀링 과정



▼ 그림 5-16 네 번째 최대 풀링 과정



- stride만큼 이동해가면서 최대값 뽑아서 피쳐맵 생성

▼ 그림 5-16 네 번째 최대 풀링 과정



- 평균 풀링 : 최대 풀링과 유사하지만 각 필터의 평균으로 계산

1. $3 + (-1) + (-3) + 1 = 0$
2. $12 + (-1) + 0 + 1 = 12$ $12/4 = 3$
3. $2 + (-3) + 3 + (-2) = 0$
4. $0 + 1 + 4 + (-1) = 4$ $4/4 = 1$
→ 0,3,0,1

✓ 풀링층 요약

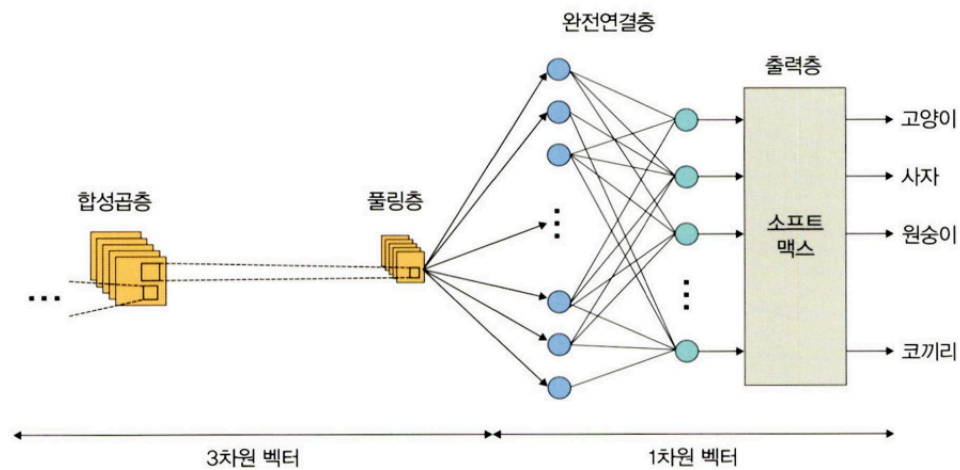
- 입력 데이터 : $W_1 * H_1 * D_1$ (W_1 : 가로, H_1 : 세로, D_1 : 채널 또는 깊이)
- 하이퍼파라미터
 - 필터 크기 : F
 - 스트라이드 : S

- 출력 데이터

- $W_2 = (W_1 - F)/S + 1$
- $H_2 = (H_1 - F)/S + 1$
- $D_2 = D_1$

📌 완전연결층 (fully connected layer)

▼ 그림 5-18 완전연결층



- 차원이 축소된 특성맵이 전달되고 3차원에서 1차원 벡터로 flatten 됨.

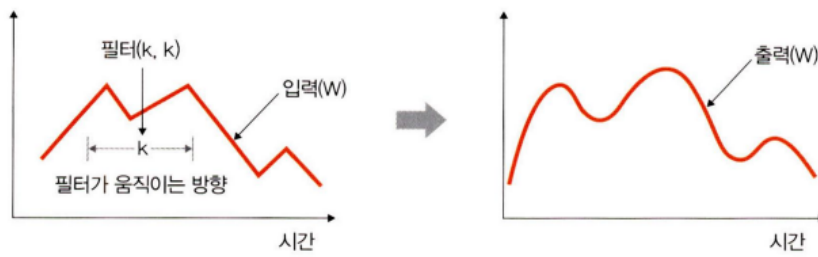
📌 출력층 (output layer)

- 소프트맥스 활성화 함수 사용
- 이미지가 각 레이블에 속할 확률 출력 = [0,1] 사이의 값 출력
- 최종값 : 가장 높은 확률값을 갖는 레이블

5.1.2 1D, 2D, 3D 합성곱

📌 1D 합성곱

♥ 그림 5-19 1D 합성곱

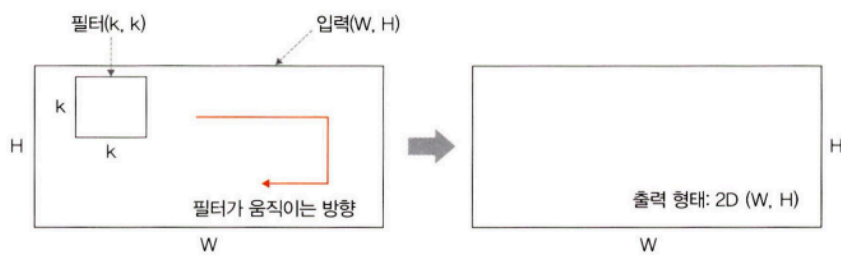


- 입력: W 너비(Width)
- 필터: $k \times k$ (높이 \times 너비)
- 출력: W 너비(Width)

- 필터가 시간을 축으로 좌우로만 이동(방향 1개)할 수 있는 합성곱
- 입력(W) \rightarrow 필터(k) \rightarrow 출력(W) : 1D vector
예) input : [1, 1, 1, 1, 1] - 필터 : [0.25, 0.5, 0.25] \rightarrow output : [1, 1, 1]

📌 2D 합성곱

♥ 그림 5-20 2D 합성곱

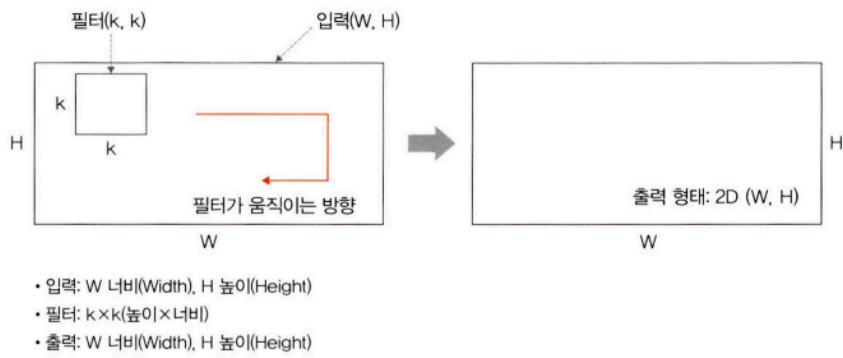


- 입력: W 너비(Width), H 높이(Height)
- 필터: $k \times k$ (높이 \times 너비)
- 출력: W 너비(Width), H 높이(Height)

- 필터가 방향 2개로 움직이는 형태
- 입력 (W, H) \rightarrow 필터 (k, k) \rightarrow 출력 (W, H) : 2D vector

📌 3D 합성곱

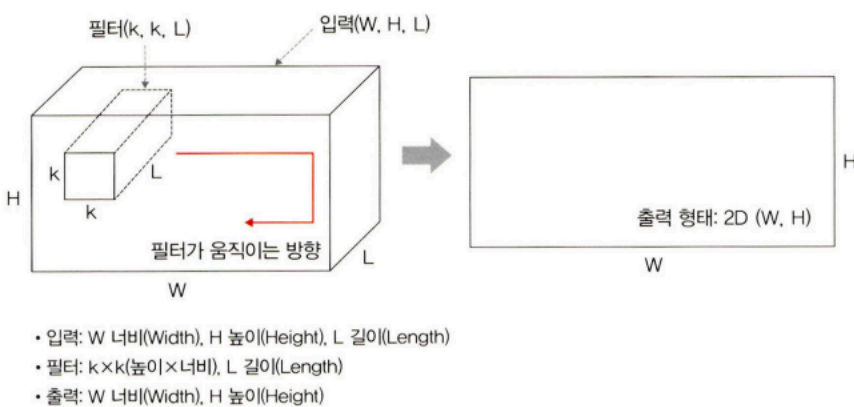
♥ 그림 5-20 2D 합성곱



- 필터가 방향 3개로 움직이는 형태
- 입력 $(W, H, L) \rightarrow$ 필터 $(k, k, d) \rightarrow$ 출력 (W, H, L) : 3D vector
 ✨ $d < L$

📌 3D 입력을 갖는 2D 합성곱

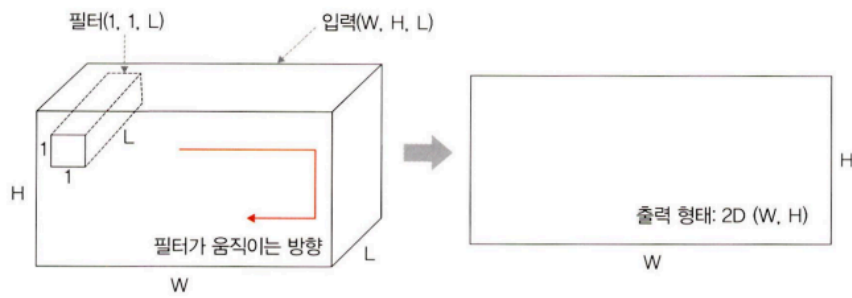
♥ 그림 5-22 3D 입력을 갖는 2D 합성곱



- 필터에 대한 길이 = 입력 채널의 길이 ($d = L$)
- 입력 $(W, H, L) \rightarrow$ 필터 $(k, k, L) \rightarrow$ 출력 (W, H) : 2D vector
- 필터가 방향 2개로 움직이는 형태
- cf) LeNet-5, VGG

📌 1x1 합성곱

▼ 그림 5-23 1×1 합성곱



- 입력: W 너비(Width), H 높이(Height), L 길이(Length)
- 필터: 1×1(높이×너비), L 길이(Length)
- 출력: W 너비(Width), H 높이(Height)

- 입력 (W, H, L) → 필터 ($1, 1, L$) → 출력 (W, H) : 2D vector
- 1×1 → 채널 수 조정, 연산량 ↓
- cf. GoogLeNet

5.2 합성곱 신경망 맛보기

* fashion_mnist 데이터셋 사용

Train_images : 28*28 크기의 numpy 배열, 0~255 값 가짐

Train labels : 0~9, 정수값을 갖는 배열

📌 라이브러리 호출

```
import numpy as np
import matplotlib.pyplot as plt

import torch
import torch.nn as nn
from torch.autograd import Variable
import torch.nn.functional as F

import torchvision
import torchvision.transforms as transforms # 데이터 전처리
from torch.utils.data import Dataset, DataLoader
```

✅ GPU 사용

하나의 GPU 사용

```
device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
model = Net()
model.to(device)
```

다수의 GPU 사용 - nn.DataParallel() 사용

```
device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
model = Net()
if torch.cuda.device_count() > 1:
    model = nn.DataParallel(model)
    # batch 크기가 알려져 각 GPU로 분배됨 : GPU 수만큼 batch 크기 ↑
model.to(device)
```

데이터셋 내려받기

```
train_dataset = torchvision.datasets.FashionMNIST(root="/content/sample_data",
                                                    train=True,
                                                    download=True,
                                                    transform = transforms.Compose([transforms.ToTensor()]))
test_dataset = torchvision.datasets.FashionMNIST(root="/content/sample_data",
                                                    train=False,
                                                    download=True,
                                                    transform=transforms.Compose([transforms.ToTensor()]))
```

`torchvision.datasets`

- `torch.utils.data.Dataset` 의 하위 클래스
- CIFAR, COCO, MNIST, ImageNet 등 다양한 데이터셋 포함
- 주요 파라미터
 1. `root =` " 다운 받을 위치 "

2. `download = True` (default = False) : 1.의 위치에 해당 데이터셋인지 확인 후 내려받음
3. `transform` : 이미지를 텐서(0~1)로 변경

```
train_loader = torch.utils.data.DataLoader(train_dataset,
                                           batch_size=100)
test_loader = torch.utils.data.DataLoader(test_dataset,
                                           batch_size=100)
```

- 데이터를 메모리로 불러오기 위해 `DataLoader` 에 전달
 - `train_dataset` : 데이터 불러올 데이터셋 지정
 - `batch_size` : 데이터를 배치로 묶음
 - `shuffle` : 무작위로 섞이도록 할 수 있음.

클래스 정의

```
labels_map = {0:'T-Shirt',1:'Trouser',2:'Pullover',
              3:'Dress', 4:'Coat', 5:'Sandal',
              6:'Shirt', 7:'Sneaker', 8:'Bag',9:'Ankle Boot'}
```

```
fig = plt.figure(figsize=(8,8))
col = 4;
row = 5;
for i in range(1,col*row+1):
    img_xy = np.random.randint(len(train_dataset))
    img = train_dataset[img_xy][0][0,:,:]
    fig.add_subplot(row,col,i)
    plt.title(labels_map[train_dataset[img_xy][1]])
    plt.axis("off")
    plt.imshow(img,cmap='gray')
plt.show()
```

- `np.random` : 무작위로 데이터 생성
- `np.random.randint()` : discrete probability distribution를 갖는 데이터에서 무작위 표본 추출

→ `random.randint(len(train_dataset))` : (0~ `train_dataset` 의 길이)값을 갖는 분포에서 랜덤숫자 하나 생성

- `np.random.rand(n)` : 0~1 사이의 정규표준분포 난수를 (1,n) 행렬로 출력
`np.random.rand(m,n)` : 0~1 사이의 정규표준분포 난수를 (m,n) 행렬로 출력
- `np.random.randn(n)` : 0~1 사이의 N(0,1) 난수를 (1,n) 행렬로 출력
`np.random.randn(m,n)` : 0~1 사이의 N(0,1) 난수를 (m,n) 행렬로 출력
- `train_dataset[img_xy][0][0,:]` : `train_dataset` 행렬 위치 지정

심층 신경망 모델 생성 (DNN)

```
class FashionDNN(nn.Module):
    def __init__(self): # 객체를 생성할 때 호출하면 실행되는 초기화 함수
        super(FashionDNN, self).__init__()
        self.fc1 = nn.Linear(in_features=784, out_features=256)
        self.drop = nn.Dropout(0.25)
        self.fc2 = nn.Linear(in_features=256, out_features=128)
        self.fc3 = nn.Linear(in_features=128, out_features=10)

    def forward(self, input_data):
        out = input_data.view(-1,784)
        out = F.relu(self.fc1(out))
        out = self.drop(out)
        out = F.relu(self.fc2(out))
        out = self.fc3(out)
        return out
```

1. class 형태의 모델은 항상 `torch.nn.Module` 을 상속받음
→ `super(FashionDNN, self).__init__()` : `FashionDNN` 이라는 부모클래스 상속
2. `torch.nn` : 딥러닝 모델 구성에 필요한 모델이 있는 패키지
`Linear` : 단순 선형 회귀 모델
3. `in_features` : 입력의 크기, `out_features` : 출력의 크기
`forward()` : 첫번째 파라미터값만 넘겨줌
두번째 파라미터에서 정의된 크기 = `forward()` 의 결과
4. `torch.nn.Dropout(p)` : p 비율만큼 텐서값이 0, 0이 아닌 값들은 기존값에 $\frac{1}{1-p}$ 만큼 곱해져 커짐

5. `forward()` : 순전파 학습 진행, 반드시 이름이 forward여야만 함. 객체를 데이터와 호출 시 자동 실행

순전파 : $H(x)$ 식에 입력 x 로부터 예측된 y 를 얻는 것

6. `.view()` : `np.reshape()` 과 같은 역할, 텐서의 크기 변경

`input_data.view(-1,784)` = `input_data` 를 (?, 784)의 크기로 변경

`-1` : 모르겠으니 계산해줘

`784` : 차원의 길이는 784로 지정

7. 활성화 함수 지정

`F.relu()` : `forward()` 함수에서 정의

`nn.ReLU()` : `__init__()` 함수에서 정의

- `nn.functional.xx(F.xx)` 와 `nn.xx` 의 차이

▼ 표 5-1 nn.xx와 nn.functional.xx의 사용 방법 비교

구분	nn.xx	nn.functional.xx
형태	nn.Conv2d: 클래스 nn.Module 클래스를 상속받아 사용	nn.functional.conv2d: 함수 def function (input)으로 정의된 순수한 함수
호출 방법	먼저 하이퍼파라미터를 전달한 후 함수 호출을 통해 데이터 전달	함수를 호출할 때 하이퍼파라미터, 데이터 전달
위치	nn.Sequential 내에 위치	nn.Sequential에 위치할 수 없음
파라미터	파라미터를 새로 정의할 필요 없음	가중치를 수동으로 전달해야 할 때마다 자체 가중치를 정의

심층 신경망에서 필요한 파라미터 정의

```
learning_rate = 0.001
```

```
model = FashionDNN()
```

```
model.to(device)
```

```
# 분류 문제에서 사용하는 손실 함수 - CrossEntropyLoss()
```

```
criterion = nn.CrossEntropyLoss()
```

```
optimizer = torch.optim.Adam(model.parameters(), lr=learning_rate)
```

```
print(model)
```

1. optimizer - Adam 사용, `lr` = 0.001 사용

2. 출력 결과 : 생성한 심층 신경망 모델

```
FashionDNN(
    (fc1): Linear(in_features=784, out_features=256, bias=True)
    (drop): Dropout(p=0.25, inplace=False)
    (fc2): Linear(in_features=256, out_features=128, bias=True)
    (fc3): Linear(in_features=128, out_features=10, bias=True)
)
```

심층 신경망을 이용한 모델 학습

```
num_epochs = 5
count = 0
loss_list = []
iteration_list = []
accuracy_list = []
predictions_list = []
labels_list = []
```

- 비어있는 배열 / 행렬을 만든 후 `append()` 를 이용해 데이터 하나씩 추가

```
for epoch in range(num_epochs):
    for images, labels in train_loader:
```

- `for` : 레코드(행, 가로줄)을 하나씩 가져옴
`train_loader` 에서 `images` (행), `labels`(가로줄) 하나씩 가져와

```
images, labels = images.to(device), labels.to(device)
```

- 모델과 데이터가 동일한 장치(CPU / GPU)에 있어야 모델이 데이터 처리가능
- `model.to(device)` 가 GPU 사용 = `images.to(device, labels.to(device))` 도 GPU

```
train = Variable(images.view(100, 1, 28, 28))
labels = Variable(labels)
outputs = model(train)
loss = criterion(outputs, labels)
optimizer.zero_grad()
loss.backward()
optimizer.step()
count += 1
```

면 실행

```
if not (count % 50): # count를 50으로 나누었을 때 나머지가 0이 아니
```

```
total = 0
correct = 0
for images, labels in test_loader:
    images, labels = images.to(device), labels.to(device)
    labels_list.append(labels)
    test = Variable(images.view(100, 1, 28, 28))
    outputs = model(test)
    predictions = torch.max(outputs, 1)[1].to(device)
    predictions_list.append(predictions)
    correct += (predictions == labels).sum()
    total += len(labels)
```

- **Autograd** : 자동 미분 수행하는 파이토치의 핵심 패키지
- 자동 미분에 대한 값을 저장하기 위해 테이프 사용
 - **forward** : 테이프는 순전파단계에서 수행하는 모든 연산 저장
 - **backward** : 역전파단계에서 저장된 값들을 꺼내 사용
- **torch.autograd.Variable** : 역전파를 위한 미분값 자동 계산

```
accuracy = correct * 100/total
loss_list.append(loss.data)
iteration_list.append(loss.data)
accuracy_list.append(accuracy)
if not (count%500):
    print("Iteration : {}, Loss : {}, Accuracy : {}%".format(count,loss.
data,accuracy))
```

- 클래스 3개 이상의 다중 분류 문제에서 주의해야 할 사항
 - 모든 클래스가 동등하게 고려된 것인지 or 특정 클래스의 분류가 잘된 것인지 알 수 없음.
- 최종 코드 결과

```
Iteration : 500, Loss : 0.5885317325592041, Accuracy : 83.33999633789062%
Iteration : 1000, Loss : 0.43888619542121887, Accuracy : 84.37999725341797%
Iteration : 1500, Loss : 0.31155502796173096, Accuracy : 84.57999420166016%
Iteration : 2000, Loss : 0.33299311995506287, Accuracy : 85.54999542236328%
Iteration : 2500, Loss : 0.2833312153816223, Accuracy : 86.37999725341797%
Iteration : 3000, Loss : 0.30242329835891724, Accuracy : 86.41999816894531%
```

CNN 생성

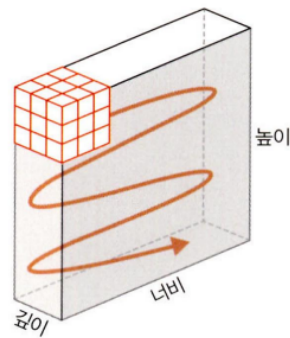
```
# CNN 생성
class FashionCNN(nn.Module):
    def __init__(self):
        super(FashionCNN, self).__init__()
        self.layer1 = nn.Sequential(
```

- `nn.Sequential`
 - `__init__()` 에서 사용할 네트워크 모델들 정의
 - `forward()` 함수에서 구현될 순전파를 계층 형태로 더 가독성이 뛰어난 코드로 작성 가능
 - $Wx + b$ 와 같은 수식과 활성화 함수를 연결해주는 역할
 - 여러개의 계층을 하나의 컨테이너에 구현하는 방법

```
nn.Conv2d(in_channels=1, out_channels=32,
          kernel_size=3, padding=1),
```

- Conv layer(합성곱층) - 합성곱 연산을 통해 이미지 특징 추출
 - `in_channels` : 입력 채널의 수 (흑백 - 1, RGB -3)
 - 채널이란 ? - 깊이

♥ 그림 5-27 채널



흑백 이미지 데이터 : $w(\text{width}) \times h(\text{height})$

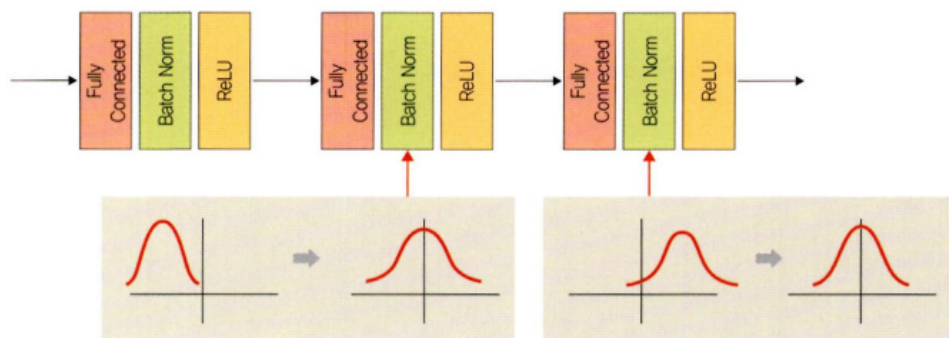
컬러 이미지 데이터 : $w \times h \times c$ (c ; # of channels)

- `out_channels` : 출력 채널의 수
- `kernel_size` : 커널 크기, 필터 / 이미지 특징을 찾아내기 위한 공용 파라미터
입력 데이터를 스트라이드 간격으로 순회하며 합성곱 계산
- `padding` : 패딩 크기, 패딩 값이 클수록 출력 크기도 커집니다.

```
nn.BatchNorm2d(32),
```

- `BatchNorm2d` : 각 배치별로 데이터가 다양한 분포를 가지더라도 평균과 분산을 이용해 정규화 $N(0, 1)$

♥ 그림 5-28 BatchNorm2d





```
nn.ReLU(),
nn.MaxPool2d(kernel_size = 2, stride = 2)
)
self.layer2 = nn.Sequential(
    nn.Conv2d(in_channels=32, out_channels=64,
              kernel_size=3),
```

```

        nn.BatchNorm2d(64),
        nn.ReLU(),
        nn.MaxPool2d(2)
    )

```

- **MaxPool2d** : 이미지 축소
 - 입력 : 합성곱층의 출력 데이터
 - 반환 : 출력 데이터(activation map) 크기 줄이거나 특정 데이터 강조
 - 최대풀링, 평균풀링, 최소 풀링이 있음
 - **kernel_size** : $m \times n$ 행렬로 구성된 가중치
 - **stride** : 입력데이터에 커널을적용할 때 이동할 간격, 스트라이드  출력크기 

```

self.fc1 = nn.Linear(in_features=64*6*6, out_features=600)
self.drop = nn.Dropout2d(0.25)
self.fc2 = nn.Linear(in_features=600, out_features=120)
self.fc3 = nn.Linear(in_features=120, out_features=10) # 최종 클래스
의 수

```

- 클래스 분류 : 이미지 형태의 데이터 → 배열 형태 변환 필요
- **Conv2d** 에서 사용하는 하이퍼파라미터 값(**padding** , **stride**)들에 따라 출력 크기 달라짐.
 - **in_features** : 입력 데이터의 크기
 - **out_features** : 출력 데이터의 크기
 - **Conv2d** 계층 출력 크기 계산식 $W_2 = (W_1 - F + 2P)/S + 1$
 - **MaxPool2d** 계층 출력 크기 계산식 IF/F
 - IF : **input_filter_size** , F : 커널크기 **kernel_size**
 - cf. **nn.MaxPool2d(kernel_size=2, stride=2)** > $784/2 = 392$

```

def forward(self,x):
    out = self.layer1(x)
    out = self.layer2(out)
    out = out.view(out.size(0),-1)
    out = self.fc1(out)
    out = self.drop(out)
    out = self.fc2(out)

```

```
out = self.fc3(out)
return out
```

- `out.view(out.size(0),-1)` : 완전연결층으로 보내야 함 → 1차원으로 변경 필요
- `out.size(0)` : (100, ?) 크기의 텐서로 변환함을 의미함.
- `(____, -1)` : 행의 수는 정확히 알지만 열의 수를 모를 때 사용

CNN에서 필요한 파라미터 정의

```
learning_rate = 0.001
model = FashionCNN()
model.to(device)

criterion = nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(model.parameters(),lr=learning_rate)
print(model)
```

- 출력 결과창 : CNN의 구조

```
FashionCNN(
  (layer1): Sequential(
    (0): Conv2d(1, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (1): BatchNorm2d(32, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (2): ReLU()
    (3): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  )
  (layer2): Sequential(
    (0): Conv2d(32, 64, kernel_size=(3, 3), stride=(1, 1))
    (1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (2): ReLU()
    (3): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  )
  (fc1): Linear(in_features=2304, out_features=600, bias=True)
  (drop): Dropout2d(p=0.25, inplace=False)
  (fc2): Linear(in_features=600, out_features=120, bias=True)
  (fc3): Linear(in_features=120, out_features=10, bias=True)
)
```

CNN 모델 학습 및 성능 평가

```
num_epochs = 5
count = 0
loss_list = []
```

```

iteration_list = []
accuracy_list = []
predictions_list = []
labels_list = []

for epoch in range(num_epochs):
    for images, labels in train_loader:
        images, labels = images.to(device), labels.to(device)

        train = Variable(images.view(100, 1, 28, 28))
        labels = Variable(labels)

        outputs = model(train)
        loss = criterion(outputs, labels)
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()
        count += 1

    if not (count % 50):
        total = 0
        correct = 0
        for images, labels in test_loader:
            images, labels = images.to(device), labels.to(device)
            labels_list.append(labels)
            test = Variable(images.view(100, 1, 28, 28))
            outputs = model(test)
            predictions = torch.max(outputs, 1)[1].to(device)
            predictions_list.append(predictions)
            correct += (predictions == labels).sum()
            total += len(labels)

        accuracy = correct * 100 / total
        loss_list.append(loss.data)
        iteration_list.append(loss.data)
        accuracy_list.append(accuracy)
    if not (count % 500):

```

```
print("Iteration : {}, Loss : {}, Accuracy : {}".format(count,loss.data,accuracy))
```

- 출력 결과창

```
Iteration : 500, Loss : 0.4551428258419037, Accuracy : 87.66999816894531%  
Iteration : 1000, Loss : 0.32626286149024963, Accuracy : 88.0999984741211%  
Iteration : 1500, Loss : 0.34484872221946716, Accuracy : 88.00999450683594%  
Iteration : 2000, Loss : 0.2086794078350067, Accuracy : 89.25%  
Iteration : 2500, Loss : 0.14014536142349243, Accuracy : 90.0999984741211%  
Iteration : 3000, Loss : 0.16668982803821564, Accuracy : 90.56999969482422%
```

→ DNN에 비해 정확도 약간 상승

이미지데이터가 많아지면 단순 DNN으로는 정확한 feature extraction/classification 불가능해짐