4.3 오버피팅과 정규화 (Overfitting and Regularization)

머신러닝 모델 과적합(Overfitting)

In [19]:

```
import torch
import torch.nn as nn
import torch.optim as optim
import torch.nn.functional as F
from torchvision import transforms, datasets
```

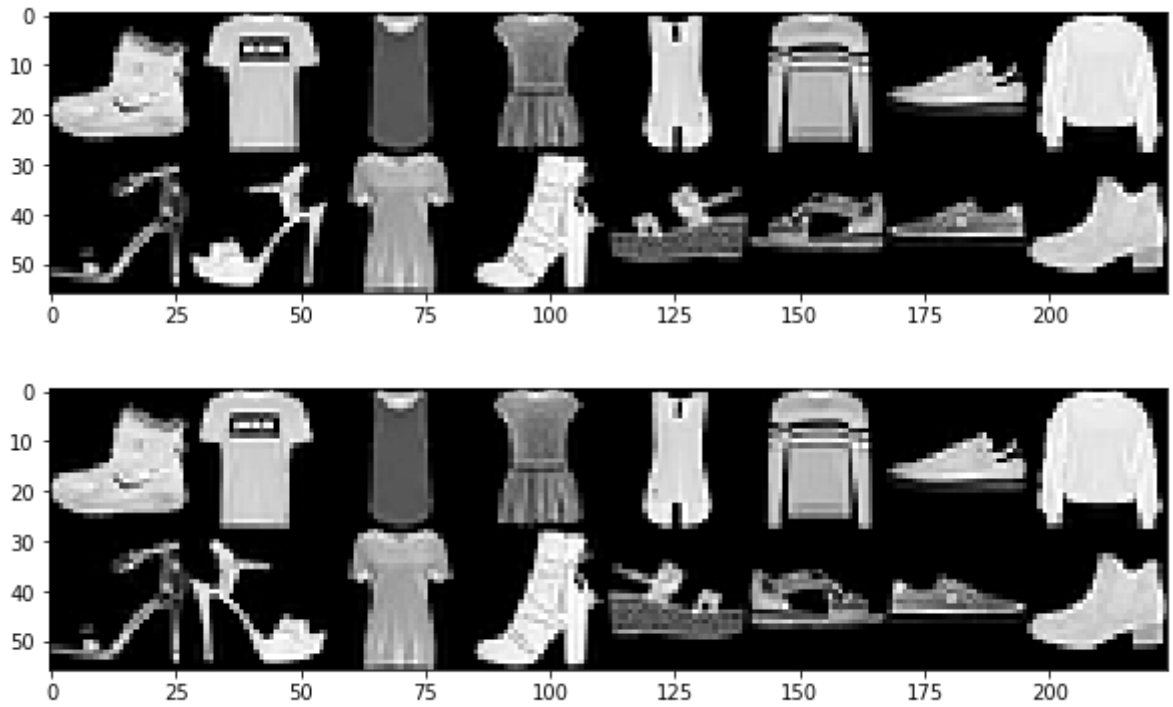
In [20]:

```
USE_CUDA = torch.cuda.is_available()
DEVICE = torch.device("cuda" if USE_CUDA else "cpu")
```

In [21]:

```
EPOCHS = 50
BATCH_SIZE = 64
```

데이터셋에 노이즈 추가하기



In [22]:

```
train_loader = torch.utils.data.DataLoader(
    datasets.MNIST('./.data',
                   train=True,
                   download=True,
                   transform=transforms.Compose([
                       transforms.RandomHorizontalFlip(),
                       transforms.ToTensor(),
                       transforms.Normalize((0.1307,), (0.3015,))
                   ])),
    batch_size=BATCH_SIZE, shuffle=True)
test_loader = torch.utils.data.DataLoader(
    datasets.MNIST('./.data',
                   train=False,
                   transform=transforms.Compose([
                       transforms.ToTensor(),
                       transforms.Normalize((0.1307,), (0.3015,))
                   ])),
    batch_size=BATCH_SIZE, shuffle=True)
```

뉴럴넷으로 Fashion MNIST 학습하기

입력 `x` 는 `[배치크기, 색, 높이, 넓이]` 로 이루어져 있습니다. `x.size()` 를 해보면 `[64, 1, 28, 28]` 이라고 표시되는 것을 보실 수 있습니다. Fashion MNIST에서 이미지의 크기는 28 x 28, 색은 흑백으로 1 가지 입니다. 그러므로 입력 x의 총 특성값 갯수는 28 x 28 x 1, 즉 784개 입니다.

우리가 사용할 모델은 3개의 레이어를 가진 뉴럴네트워크 입니다.

In [23]:

```
class Net(nn.Module):
    def __init__(self, dropout_p=0.2):
        super(Net, self).__init__()
        self.fc1 = nn.Linear(784, 256)
        self.fc2 = nn.Linear(256, 128)
        self.fc3 = nn.Linear(128, 10)
        # 드롭아웃 확률
        self.dropout_p = dropout_p

    def forward(self, x):
        x = x.view(-1, 784)
        x = F.relu(self.fc1(x))
        # 드롭아웃 추가
        x = F.dropout(x, training=self.training,
                      p=self.dropout_p)
        x = F.relu(self.fc2(x))
        # 드롭아웃 추가
        x = F.dropout(x, training=self.training,
                      p=self.dropout_p)
        x = self.fc3(x)
        return x
```

모델 준비하기

`to()` 함수는 모델의 파라미터들을 지정한 곳으로 보내는 역할을 합니다. 일반적으로 CPU 1개만 사용할 경우 필요는 없지만, GPU를 사용하고자 하는 경우 `to("cuda")` 로 지정하여 GPU로 보내야 합니다. 지정하지 않을 경우 계속 CPU에 남아 있게 되며 빠른 훈련의 이점을 누리실 수 없습니다.

최적화 알고리즘으로 파이토치에 내장되어 있는 `optim.SGD` 를 사용하겠습니다.

In [24]:

```
model         = Net(dropout_p=0.2).to(DEVICE)
optimizer     = optim.SGD(model.parameters(), lr=0.01)
```

학습하기

In [25]:

```
def train(model, train_loader, optimizer):
    model.train()
    for batch_idx, (data, target) in enumerate(train_loader):
        data, target = data.to(DEVICE), target.to(DEVICE)
        optimizer.zero_grad()
        output = model(data)
        loss = F.cross_entropy(output, target)
        loss.backward()
        optimizer.step()
```

테스트하기

In [26]:

```
def evaluate(model, test_loader):
    model.eval()
    test_loss = 0
    correct = 0
    with torch.no_grad():
        for data, target in test_loader:
            data, target = data.to(DEVICE), target.to(DEVICE)
            output = model(data)
            test_loss += F.cross_entropy(output, target,
                                         reduction='sum').item()

            # 맞춘 갯수 계산
            pred = output.max(1, keepdim=True)[1]
            correct += pred.eq(target.view_as(pred)).sum().item()

    test_loss /= len(test_loader.dataset)
    test_accuracy = 100. * correct / len(test_loader.dataset)
    return test_loss, test_accuracy
```

코드 돌려보기

자, 이제 모든 준비가 끝났습니다. 코드를 돌려서 실제로 훈련이 되는지 확인해봅시다!

In [*]:

```
for epoch in range(1, EPOCHS + 1):
    train(model, train_loader, optimizer)
    test_loss, test_accuracy = evaluate(model, test_loader)

    print('[{}] Test Loss: {:.4f}, Accuracy: {:.2f}%'.format(
          epoch, test_loss, test_accuracy))
```

```
[1] Test Loss: 0.5408, Accuracy: 82.79%
[2] Test Loss: 0.4149, Accuracy: 87.08%
[3] Test Loss: 0.3366, Accuracy: 89.71%
[4] Test Loss: 0.2829, Accuracy: 91.31%
[5] Test Loss: 0.2429, Accuracy: 92.48%
[6] Test Loss: 0.2173, Accuracy: 93.43%
[7] Test Loss: 0.1977, Accuracy: 93.84%
[8] Test Loss: 0.1858, Accuracy: 94.18%
[9] Test Loss: 0.1738, Accuracy: 94.66%
[10] Test Loss: 0.1648, Accuracy: 94.78%
[11] Test Loss: 0.1544, Accuracy: 95.19%
[12] Test Loss: 0.1530, Accuracy: 95.28%
[13] Test Loss: 0.1464, Accuracy: 95.50%
[14] Test Loss: 0.1437, Accuracy: 95.54%
[15] Test Loss: 0.1354, Accuracy: 95.84%
[16] Test Loss: 0.1323, Accuracy: 95.81%
[17] Test Loss: 0.1296, Accuracy: 95.89%
[18] Test Loss: 0.1268, Accuracy: 96.03%
[19] Test Loss: 0.1213, Accuracy: 96.28%
[20] Test Loss: 0.1187, Accuracy: 96.25%
[21] Test Loss: 0.1180, Accuracy: 96.25%
[22] Test Loss: 0.1126, Accuracy: 96.40%
[23] Test Loss: 0.1124, Accuracy: 96.39%
[24] Test Loss: 0.1116, Accuracy: 96.51%
[25] Test Loss: 0.1080, Accuracy: 96.65%
[26] Test Loss: 0.1092, Accuracy: 96.58%
[27] Test Loss: 0.1069, Accuracy: 96.61%
[28] Test Loss: 0.1072, Accuracy: 96.70%
[29] Test Loss: 0.1006, Accuracy: 96.69%
[30] Test Loss: 0.1042, Accuracy: 96.74%
[31] Test Loss: 0.0991, Accuracy: 96.79%
[32] Test Loss: 0.1011, Accuracy: 96.73%
[33] Test Loss: 0.0988, Accuracy: 96.87%
[34] Test Loss: 0.0960, Accuracy: 96.93%
[35] Test Loss: 0.0974, Accuracy: 96.91%
[36] Test Loss: 0.0964, Accuracy: 97.01%
[37] Test Loss: 0.0943, Accuracy: 97.04%
[38] Test Loss: 0.0930, Accuracy: 97.02%
```

201700949 설재혁

In [ ]:

```
```