Chapter 5

신경망 구조 및 모듈

학습 목표

- **신경망 구조 이해** : 신경망의 기본 구성 요소(선형 레이어, 활성화 함수, 손실 함수 등)와 이들이 어떻게 상호작용하여 모델을 구성하는지를 이해할 수 있다.
- **선형 레이어와 비선형성** : 선형 레이어의 역할과 이론적 기초를 이해하고, 비선형성의 중요성을 설명하며, 다양한 활성화 함수(예: ReLU, Sigmoid)의 특성을 이해할 수 있다.
- **파이토치에서 신경망 구축** : 파이토치의 `torch.nn` 모듈을 활용하여 신경망을 구현하고, 사용자 정의 레이어 및 모델 구조를 만드는 방법을 익힐 수 있다.
- **파이토치의 기본 모듈 구조 및 역할** : 파이토치에서 제공하는 주요 모듈(예: 데이터 로더, 옵티마이저, 체크포인트 관리)의 기능과 역할을 이해하고 이를 신경망 훈련에 effectively 활용할 수 있다.
- **모델 평가 및 조정** : 신경망 모델의 성능을 평가하고, 하이퍼파라미터 조정 및 학습률 조정 등의 방법으로 모델을 최적화하는 과정을 이해할 수 있다.

신경망 구조 **(Neural Network Architecture)**

신경망은 인간의 뇌에서 영감을 받아 설계된 알고리즘으로, 계층(layer)을 통해 입력 데이터를 처리하고 학습하여 결과를 출력합니다.

1) **입력층 (Input Layer)**

- 데이터가 처음 들어오는 계층입니다.
- 각 노드는 하나의 입력 특성(feature)을 나타냅니다.
- 입력층은 데이터의 차원과 동일한 노드 수를 가집니다.

2) **은닉층 (Hidden Layer)**

- 입력층과 출력층 사이에 위치하며, 데이터를 처리하고 복잡한 패턴을 학습합니다.
- 여러 개의 은닉층을 가진 신경망은 **심층 신경망** (Deep Neural Network)이라고 부릅니다.
- 각 노드는 가중치(weight), 편향(bias), 그리고 활성화 함수(activation function)를 사용하여 입력 신호를 변환합니다.

3) **출력층 (Output Layer)**

- 최종 결과를 출력하는 계층입니다.
- 출력층의 노드 수는 문제 유형에 따라 다릅니다:
  - 회귀: 출력층에 1개의 노드 (연속 값 출력).
  - 이진 분류: 출력층에 1개의 노드 (시그모이드 함수).

- 다중 클래스 분류: 클래스 수만큼 노드 (소프트맥스 함수).

4) 신경망의 주요 특징 :

- **가중치 (Weights)**: 입력 데이터를 다음 계층으로 전달할 때 곱해지는 값.
- **편향 (Bias)**: 출력값을 조정하기 위한 상수.
- **활성화 함수 (Activation Function)**: 비선형성을 추가하여 신경망이 복잡한 패턴을 학습할 수 있도록 함.
  - 예: ReLU, Sigmoid, Tanh.

선형 레이어 (Linear Layer)

선형 레이어는 신경망의 기본 구성 요소로, 입력 데이터에 선형 변환(linear transformation)을 적용합니다.

1) 동작 원리:

1. **입력 데이터 x**가 들어옴.
2. x에 가중치 행렬 W를 곱함.
3. 편향 b를 더함.
4. 결과를 다음 계층으로 전달.

2) 역할:

- 선형 레이어는 데이터의 선형 결합을 생성합니다.
- 비선형성을 추가하는 활성화 함수와 함께 사용해야 더 복잡한 패턴을 학습할 수 있습니다.

신경망과 선형 레이어의 관계

- 신경망의 각 계층은 여러 개의 선형 레이어로 구성될 수 있습니다.
- 각 선형 레이어는 활성화 함수와 결합하여 데이터의 복잡한 패턴을 학습합니다.
- 선형 레이어는 데이터 변환의 핵심 역할을 하며, 신경망의 성능에 크게 기여합니다.
- **요약** : 신경망은 선형 레이어와 활성화 함수의 조합을 통해 입력 데이터를 점진적으로 변환하며, 선형 레이어는 이 변환의 기본적인 단위를 제공합니다.

5.1 선형 레이어의 개념

- 선형 레이어는 신경망에서 입력 데이터를 가중치와 함께 선형적으로 변환하는 역할을 합니다.
- 선형 레이어는 수학적으로 행렬 곱셈을 수행하는 연산입니다.
- 입력 벡터 x에 가중치 행렬 w을 곱하고, 그 후 편향 b를 더하는 형태로 정의됩니다.
- 선형 레이어 수식 y = wx + b

5.1.1 Linear Layer

- nn.Linear를 사용하여 선형 레이어 정의할 수 있습니다.
- nn.Linear는 입력 크기와 출력 크기에 대해 가중치와 편향을 자동으로 초기화하고 선형 변환을 수행하는 연산을 정의합니다.
- 선형 레이어는 선형 변환을 수행하며 입력 벡터와 가중치 행렬을 곱하고, 편향을 더하는 방식으로 계산합니다.
- nn.Linear(in_features, out_features)는 입력 차원 in_features와 출력 차원 out_features를 지정하여 선형 변환을 수행하는 레이어입니다.
- 입력층: in_features (입력 뉴런의 수)
- 출력층: out_feature (출력 뉴런의 수)

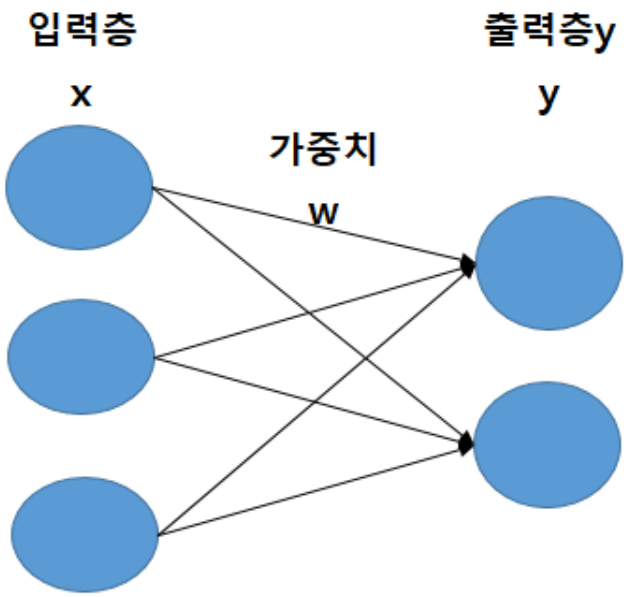

그림 5.1 선형 레이어

In [1]:
```
import torch
import torch.nn as nn

# 입력 크기(n)와 출력 크기(m)을 지정하여 선형 레이어 정의
linear_layer = nn.Linear(in_features=3, out_features=2)

# 입력 데이터를 생성 (배치 크기 4, 입력 크기 3)
x = torch.randn(4, 3)

# x를 선형 레이어로 통과시켜 출력 얻기
output = linear_layer(x)

print(output)
```

```
tensor([[-0.0972,  0.3406],
        [-0.5532,  1.2897],
        [-0.4128,  1.1238],
        [-0.0123,  0.1560]], grad_fn=<AddmmBackward0>)
```

5.1.2 신경망 모델에서 사용

- 선형 레이어는 신경망의 여러 층에서 사용될 수 있습니다. 간단한 다층 퍼셉트론(MLP)을 만들 때 여러 선형 레이어를 쌓을 수 있습니다.

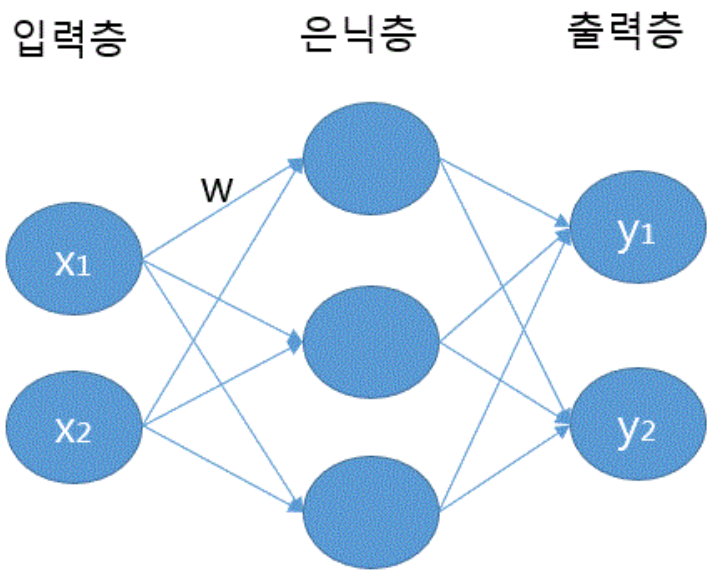

그림 5.2 신경망 구조

- 두 개의 선형 레이어를 사용하여 입력을 변환하는 간단한 신경망 모델입니다.

In [18]:
```
# 신경망 모델 정의
class SimpleNN(nn.Module):
    def __init__(self):
        super(SimpleNN, self).__init__()
        self.layer1 = nn.Linear(3, 2)  # 3 -> 2
        self.layer2 = nn.Linear(2, 2)  # 2 -> 2

    def forward(self, x):
        x = torch.relu(self.layer1(x))  # 첫 번째 선형 레이어 + ReLU 활성화 함수
        x = self.layer2(x)  # 두 번째 선형 레이어
        return x

# 모델 생성
model = SimpleNN()

# 입력
x = torch.randn(4, 3)  # 배치 크기 4, 입력 크기 3

# 모델을 통해 출력 얻기
output = model(x)
print(output)
```

```
tensor([[ 0.1721, -0.0883],
        [ 0.1721, -0.0883],
        [ 0.3153,  0.2625],
        [ 0.1109, -0.0269]], grad_fn=<AddmmBackward0>)
```

5.1.3 Convolutional Layer (합성곱 레이어)

- **Convolutional Layer는 이미지 처리나 컴퓨터 비전 작업**에서 사용됩니다.
- 입력 텐서에 대해 필터(또는 커널)를 적용하여 특징 맵을 생성하고, 네트워크의 파라미터로 학습 가능한 가중치를 포함합니다.
- torch.nn.Conv2d는 2D 합성곱 연산을 수행하는 레이어로, 이미지 특징을 추출합니다.
- torch.nn.Conv2d(in_channels, out_channels, kernel_size)
- in_channels는 입력 채널 수, 흑백 이미지는 1, 칼라 이미지는 3
- kernel_size는 합성곱 필터의 크기
- out_channels는 출력 채널 수

In [3]:
```python
import torch
import torch.nn as nn

# Conv2d Layer 정의
# Conv2d(in_channels, out_channels, kernel_size, stride=1, padding=0)
# 3x3 커널
conv_layer = nn.Conv2d(in_channels=3, out_channels=16, kernel_size=3, stride=1, paddin

# 입력 텐서 생성 (배치 크기, 채널 수, 높이, 너비)
# 배치 크기 1, 3채널(컬러 이미지), 32x32 크기 이미지
input_tensor = torch.randn(1, 3, 32, 32)

# Convolution 연산
output_tensor = conv_layer(input_tensor)

# 출력 텐서 크기 확인
print(output_tensor.shape)
```

```
torch.Size([1, 16, 32, 32])
```

5.2 파이토치에서의 사용법

`nn.Linear` 네트워크 구성

In [4]:
```python
import torch
import torch.nn as nn
import torch.optim as optim

# 신경망 모델 정의
class SimpleNN(nn.Module):
    def __init__(self, input_size, hidden_size, output_size):
        super(SimpleNN, self).__init__()
        # 첫 번째 층 (입력 -> 은닉층)
        self.layer1 = nn.Linear(input_size, hidden_size)
        # 두 번째 층 (은닉층 -> 출력층)
        self.layer2 = nn.Linear(hidden_size, output_size)

    def forward(self, x):
        # 첫 번째 층을 통과하고 ReLU 활성화 함수 적용
        x = torch.relu(self.layer1(x))
        # 두 번째 층을 통과
        x = self.layer2(x)
        return x

# 네트워크 초기화
input_size = 10  # 입력 벡터의 크기
```

```
hidden_size = 5  # 은닉층의 크기
output_size = 1  # 출력층의 크기 (회귀 문제의 경우 1)

model = SimpleNN(input_size, hidden_size, output_size)

# 모델 파라미터 개수 확인
print(model)

# 입력 데이터 (배치 크기 3, 입력 크기 10)
x = torch.randn(3, input_size)

# 예측 수행
output = model(x)
print("모델 출력:", output)
```

```
SimpleNN(
  (layer1): Linear(in_features=10, out_features=5, bias=True)
  (layer2): Linear(in_features=5, out_features=1, bias=True)
)
모델 출력: tensor([[0.3304],
        [0.0176],
        [0.3114]], grad_fn=<AddmmBackward0>)
```

5.3 nn.Module

- PyTorch의 nn.Module은 신경망 모델을 구현 기본 클래스입니다.
- 신경망 구조를 만들기 위해 nn.Module을 상속받고, 모델을 정의하는 과정에서 forward() 메소드를 구현해야 합니다.
- 기본적으로 nn.Module은 네트워크의 계층을 구성하고, 가중치 및 파라미터를 관리하며, 손실 계산과 같은 기능을 수행하는 데 필요한 많은 도구들을 제공합니다.

5.3.1 nn.Module 클래스 개요

- nn.Module은 PyTorch의 신경망 모델을 정의하기 위한 클래스입니다.
- 신경망의 계층(layer)과 연산을 정의하려면 이 클래스를 상속받아야 합니다.
- 파라미터, 옵티마이저, 학습 상태 등 신경망 모델에서 필요한 여러 기능을 관리합니다.

5.3.2 신경망 모델 구현

> 1. 라이브러리 불러오기

In [5]:
```
# 1. 파이토치 라이브러리
import torch

# 신경망 모델 정의
import torch.nn as nn
```

> 2. nn.Module 신경망 클래스 정의

- 구성 요소

- **init**(self):

  네트워크의 계층을 정의하는 부분입니다. 여기서는 nn.Linear를 사용하여 선형 계층을 정의했습니다.

  self.fc1과 self.fc2는 선형 변환을 수행하는 레이어입니다.

  self.relu는 활성화 함수로 ReLU를 사용합니다.

- forward(self, x):

  신경망의 순전파 (forward) 계산을 정의하는 부분입니다.

  입력 x는 첫 번째 선형 계층 fc1을 통과하고, 그 후 ReLU 활성화 함수를 거쳐 두 번째 선형 계층 fc2를 통과합니다.

  최종적으로 10개의 클래스를 출력하는 예측 결과가 반환됩니다.

In [6]:
```
# nn.Module을 상속받는 신경망 클래스 정의
class SimpleNN(nn.Module):
    def __init__(self):
        super(SimpleNN, self).__init__()

        # 신경망의 레이어 정의
        self.fc1 = nn.Linear(784, 128)  # 784 입력을 받아 128 출력
        self.relu = nn.ReLU()           # ReLU 활성화 함수
        self.fc2 = nn.Linear(128, 10)   # 128 입력을 받아 10 출력
        # (예: 10개의 클래스 분류)

    def forward(self, x):
        # forward 메소드에서는 네트워크의 연산 흐름을 정의
        x = self.fc1(x)  # 첫 번째 선형 변환
        x = self.relu(x)  # 활성화 함수
        x = self.fc2(x)   # 두 번째 선형 변환
        return x
```

In [7]:
```
# 모델 인스턴스 생성
model = SimpleNN()

# 임의의 입력값 (예: 28x28 이미지 크기를 1차원으로 펼친 벡터)
input_data = torch.randn(64, 784)  # 배치 크기 64, 각 이미지 784 크기
output_data = model(input_data)

print(output_data.shape)
# (64, 10) : 64개의 입력에 대해 10개의 출력 (각각의 클래스에 대한 예측)
```

```
torch.Size([64, 10])
```

nn.Module 장점

- 모듈화는 모델을 여러 계층(layer)로 나누어 구성하고, 각 계층을 독립적으로 관리할 수 있습니다.
- 자동 기울기 계산은 forward 메소드에 정의된 연산에 대해 자동으로 기울기를 계산하고 역전파가 이루어집니다.
- 체크포인트 저장 및 불러오기에는 state_dict()와 load_state_dict()를 통해 학습된 모델을 저장하고 불러오는 기능을 제공합니다.

In [13]:
```
# 모델 불러오기 (weights_only=True)
model = SimpleNN()  # 새로운 모델 인스턴스
model.load_state_dict(torch.load('simple_nn.pth', weights_only=True))
model.eval()  # 평가 모드로 설정 (드롭아웃, 배치 정규화 등이 필요할 때)
```

Out[13]:
```
SimpleNN(
  (fc1): Linear(in_features=784, out_features=128, bias=True)
  (relu): ReLU()
  (fc2): Linear(in_features=128, out_features=10, bias=True)
)
```

nn.Module로 신경망 구조 실습

1. 신경망 모델 정의

- 단순 신경망 모델 구현, MNIST 데이터셋 사용하고 학습과 평가 진행

In [9]:
```
import torch
import torch.nn as nn
import torch.optim as optim
from torch.utils.data import DataLoader
from torchvision import datasets, transforms
```

In [10]:
```
# 신경망 모델 클래스 정의
class SimpleNN(nn.Module):
    def __init__(self):
        super(SimpleNN, self).__init__()

        # 레이어 정의
        self.fc1 = nn.Linear(28 * 28, 128)
        # 입력 28x28 이미지를 128 차원으로 변환
        self.relu = nn.ReLU()                # 활성화 함수
        self.fc2 = nn.Linear(128, 10)        # 128 차원을 10개 클래스(0~9)로 변환

    def forward(self, x):
        x = x.view(-1, 28 * 28)  # 입력 이미지를 1차원 벡터로 변환
        x = self.fc1(x)          # 첫 번째 선형 계층
        x = self.relu(x)         # ReLU 활성화 함수
        x = self.fc2(x)          # 두 번째 선형 계층
        return x

# 모델 인스턴스 생성
model = SimpleNN()
```

2. 데이터셋 로딩(DataLoader) 및 전처리

- MNIST 데이터셋을 로드하고, 전처리로 이미지를 텐서로 변환한 후 배치로 나누어 모델에 입력할 준비를 합니다.

In [11]:
```
# 데이터 전처리: 이미지를 텐서로 변환
transform = transforms.Compose([
    transforms.ToTensor(),        # 이미지를 텐서로 변환
    transforms.Normalize((0.5,), (0.5,))  # 정규화
])

# MNIST 데이터셋 다운로드 및 로딩
train_dataset = datasets.MNIST(root='./data', train=True, download=True, transform=tra
test_dataset = datasets.MNIST(root='./data', train=False, download=True, transform=tra

train_loader = DataLoader(train_dataset, batch_size=64, shuffle=True)
test_loader = DataLoader(test_dataset, batch_size=64, shuffle=False)
```

3. 모델학습

- 모델을 학습하려면 손실함수와 옵티마이저를 정의하고, 데이터로부터 학습을 진행해야 합니다.
- 여기서는 CrossEntropyLoss와 SGD(확률적 경사하강법) 옵티마이저를 사용합니다.

In [12]:
```
# 손실 함수와 옵티마이저 정의
criterion = nn.CrossEntropyLoss()      # 다중 클래스 분류 문제에 적합
optimizer = optim.SGD(model.parameters(), lr=0.01, momentum=0.9)

# 학습 함수
def train(model, train_loader, criterion, optimizer, num_epochs=5):
    model.train()  # 학습 모드로 전환
    for epoch in range(num_epochs):
        running_loss = 0.0
        correct = 0
        total = 0
        for inputs, labels in train_loader:
            optimizer.zero_grad()  # 이전 기울기 초기화
            outputs = model(inputs)  # 순전파
            loss = criterion(outputs, labels)  # 손실 계산
            loss.backward()  # 역전파
            optimizer.step()  # 가중치 업데이트

            running_loss += loss.item()  # 손실 누적
            _, predicted = torch.max(outputs, 1)  # 예측값
            total += labels.size(0)
            correct += (predicted == labels).sum().item()  # 정확도 계산

        # 에폭마다 손실과 정확도 출력
        print(f"Epoch [{epoch+1}/{num_epochs}], Loss: {running_loss/len(train_loader):

# 모델 학습
train(model, train_loader, criterion, optimizer, num_epochs=5)
```

```
Epoch [1/5], Loss: 0.3716, Accuracy: 88.78%
Epoch [2/5], Loss: 0.1795, Accuracy: 94.72%
Epoch [3/5], Loss: 0.1314, Accuracy: 96.05%
Epoch [4/5], Loss: 0.1049, Accuracy: 96.89%
Epoch [5/5], Loss: 0.0891, Accuracy: 97.33%
```

4. 모델 평가

- 학습이 끝난 후에는 모델을 평가하고, 테스트 데이터셋에서 정확도를 계산

In [16]:
```python
# 모델 평가 함수
def evaluate(model, test_loader):
    model.eval()  # 평가 모드로 전환
    correct = 0
    total = 0
    with torch.no_grad():  # 기울기 계산 비활성화
        for inputs, labels in test_loader:
            outputs = model(inputs)  # 순전파
            _, predicted = torch.max(outputs, 1)  # 예측값
            total += labels.size(0)
            correct += (predicted == labels).sum().item()  # 정확도 계산

    print(f'Test Accuracy: {100 * correct / total:.2f}%')

# 모델 평가
evaluate(model, test_loader)
```

```
Test Accuracy: 4.93%
```