Pytorch 시작하기

학습 내용

- 파이토치는 무엇일까요?

파이토치(pytorch)는 무엇일까요?

- (1) Facebook에서 개발한 오픈 소스 머신 러닝 라이브러리.
- (2) PyTorch는 파이썬을 기반으로 한다.
- (3) 주요 특징
  - 동적 계산 그래프 : PyTorch는 실행 시점에 계산 그래프를 생성한다.
  - 강력한 GPU 지원
  - 파이썬과 친숙한 문법과 사용성 제공
  - PyTorch는 자동으로 기울기(gradient)를 계산하는 기능을 제공한다.
  - 연구자와 실무자 모두에게 인기가 있다.

기본 예제 - 기본적인 텐서 연산과 자동미분(autograd)을 사용하는 코드

In [1]:
```
# PyTorch 라이브러리 설치 (구글 코랩에서는 일반적으로 설치가 되어 있지만, 없을 경
!pip install torch
```

```
Requirement already satisfied: torch in /usr/local/lib/python3.10/dist-packages
(2.4.0+cu121)
Requirement already satisfied: filelock in /usr/local/lib/python3.10/dist-package
s (from torch) (3.15.4)
Requirement already satisfied: typing-extensions>=4.8.0 in /usr/local/lib/python
3.10/dist-packages (from torch) (4.12.2)
Requirement already satisfied: sympy in /usr/local/lib/python3.10/dist-packages
(from torch) (1.13.2)
Requirement already satisfied: networkx in /usr/local/lib/python3.10/dist-package
s (from torch) (3.3)
Requirement already satisfied: jinja2 in /usr/local/lib/python3.10/dist-packages
(from torch) (3.1.4)
Requirement already satisfied: fsspec in /usr/local/lib/python3.10/dist-packages
(from torch) (2024.6.1)
Requirement already satisfied: MarkupSafe>=2.0 in /usr/local/lib/python3.10/dist-
packages (from jinja2->torch) (2.1.5)
Requirement already satisfied: mpmath<1.4,>=1.1.0 in /usr/local/lib/python3.10/di
st-packages (from sympy->torch) (1.3.0)
```

In [3]:
```
import torch
import torch.nn as nn
import torch.optim as optim
import matplotlib.pyplot as plt

# torch: PyTorch 라이브러리의 기본 모듈로, 텐서 연산과 딥러닝을 위한 다양한 기능을
# torch.nn: 신경망을 구성하는 다양한 클래스와 기능을 제공하는 모듈
# torch.optim: 다양한 최적화 알고리즘(예: SGD, Adam 등)을 제공하는 모듈
```

```
# matplotlib.pyplot: 데이터를 시각화하기 위한 파이썬 라이브러리

print(torch.__version__)
```

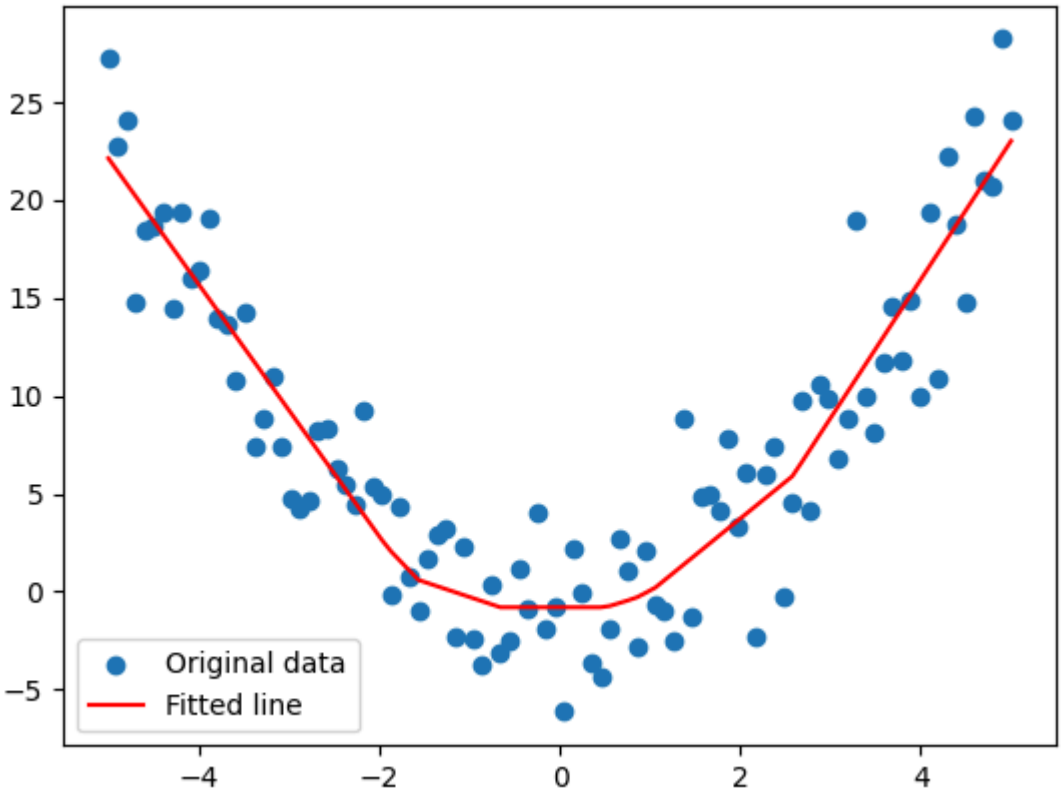
```
2.4.0+cu121
```

In [4]:

```
# SimpleNet 클래스는 PyTorch의 nn.Module을 상속받아 정의한 간단한 신경망 모델
# __init__ 메서드: 신경망의 구조를 정의
# forward 메서드: 신경망의 순전파(forward pass)를 정의


# 간단한 신경망 모델 정의
class SimpleNet(nn.Module):
    # __init__ 메서드: 신경망의 구조를 정의
    def __init__(self):
        super(SimpleNet, self).__init__()

        # self.fc1 = nn.Linear(1, 10): 입력이 1차원이고 출력이 10차원인 첫 번째 
        # self.relu = nn.ReLU(): ReLU(Rectified Linear Unit) 활성화 함수
        # self.fc2 = nn.Linear(10, 1): 10차원의 입력을 받아 1차원의 출력을 내는 
        self.fc1 = nn.Linear(1, 10)
        self.relu = nn.ReLU()
        self.fc2 = nn.Linear(10, 1)

    # forward 메서드: 신경망의 순전파(forward pass)를 정의
    def forward(self, x):
        # 입력 x는 먼저 fc1 레이어를 통과하고, ReLU 활성화 함수를 거쳐 fc2 레이어
        x = self.relu(self.fc1(x))
        x = self.fc2(x)  # 출력 레이어에는 ReLU를 적용하지 않습니다
        return x

# 데이터 생성
# torch.linspace(start, end, steps) 함수는 start에서 end까지의 범위에서 등간격으
# .view()는 PyTorch에서 텐서의 크기(shape)를 변경하는 함수. 뒤는 1로 맞추고 -1은 
x = torch.linspace(-5, 5, 100).view(-1, 1)

# x**2: 입력 x의 제곱을 계산하여 비선형 관계를 생성
# torch.randn(x.size()) * 3: x와 동일한 크기의 표준 정규 분포를 따르는 랜덤 텐서
# 이를 3배하여 노이즈로 사용합니다. 이는 y 값에 랜덤한 변동을 추가하여 데이터의 
y = x**2 + torch.randn(x.size()) * 3

# 모델, 손실 함수, 옵티마이저 초기화

# model = SimpleNet(): 정의된 SimpleNet 신경망 모델의 인스턴스를 생성
# criterion = nn.MSELoss(): 손실 함수로 평균 제곱 오차(MSE, Mean Squared Error)를
# MSE는 예측 값과 실제 값 간의 차이의 제곱의 평균을 계산합니다. 회귀 문제에서 자주
# optimizer = optim.Adam(model.parameters(), lr=0.01): Adam 옵티마이저를 사용하여
# 학습률(learning rate)은 0.01로 설정
model = SimpleNet()
criterion = nn.MSELoss()
optimizer = optim.Adam(model.parameters(), lr=0.01)

# 학습 루프
# num_epochs = 1000: 모델 학습을 위한 에폭 수를 1000으로 설정합니다. 1 에폭은 전체
#
num_epochs = 1000
for epoch in range(num_epochs):
    # 순전파
    # outputs = model(x): 모델에 입력 x를 통과시켜 예측 값을 얻습니다.
    # loss = criterion(outputs, y): 예측 값과 실제 값 y 간의 손실을 계산합니다.
```

```
    outputs = model(x)
    loss = criterion(outputs, y)

    # 역전파 및 최적화
    # optimizer.zero_grad(): 옵티마이저의 기울기를 초기화합니다.
    #    PyTorch에서는 기본적으로 기울기가 누적되므로, 매 반복마다 초기화가 필요합
    # loss.backward(): 역전파(backpropagation)를 수행하여 기울기를 계산합니다. 모
    # optimizer.step(): 옵티마이저가 기울기를 사용하여 파라미터를 업데이트합니다.
    optimizer.zero_grad()
    loss.backward()
    optimizer.step()

    # (epoch+1) % 100 == 0일 때, 즉 매 100번째 에폭마다 현재 에폭 번호와 손실 값을
    if (epoch+1) % 100 == 0:
        print(f'Epoch [{epoch+1}/{num_epochs}], Loss: {loss.item():.4f}')

# 결과 시각화

# model.eval(): 모델을 평가 모드로 전환합니다. 평가 모드에서는 드롭아웃(dropout)과
# 같은 레이어가 훈련 모드에서와 다르게 작동하므로, 이를 비활성화합니다.

# with torch.no_grad():: 기울기 계산을 비활성화하여 추론(inference) 모드로 전환합
# 메모리 사용량을 줄이고 계산 속도를 높이기 위해 사용됩니다.
# predicted = model(x): 학습된 모델을 사용하여 x에 대한 예측 값을 계산합니다.
model.eval()
with torch.no_grad():
    predicted = model(x)

# plt.scatter(x.numpy(), y.numpy(), label='Original data'): 원본 데이터를 산점도
# plt.plot(x.numpy(), predicted.numpy(), color='red', label='Fitted line'): 모델
plt.scatter(x.numpy(), y.numpy(), label='Original data')
plt.plot(x.numpy(), predicted.numpy(), color='red', label='Fitted line')

# plt.legend() 및 plt.show(): 그래프의 범례를 추가하고, 그래프를 화면에 표시합니다
plt.legend()
plt.show()

print("학습 완료!")
```

```
Epoch [100/1000], Loss: 22.5298
Epoch [200/1000], Loss: 14.9777
Epoch [300/1000], Loss: 11.5867
Epoch [400/1000], Loss: 10.3279
Epoch [500/1000], Loss: 9.6922
Epoch [600/1000], Loss: 9.4535
Epoch [700/1000], Loss: 9.3779
Epoch [800/1000], Loss: 9.3390
Epoch [900/1000], Loss: 9.3128
Epoch [1000/1000], Loss: 9.2779
```



학습 완료!

더 생각해 보기, 이해를 위한 실습

실습 문제 1: 모델의 깊이와 폭 확장하기

현재 모델 SimpleNet은 입력층과 출력층 사이에 하나의 은닉층만을 가지고 있습니다. 은닉층의 수와 각 은닉층의 노드 수(폭)를 늘려 모델의 깊이를 더해 보세요.

단계 1: 은닉층을 2개 이상 추가하고, 각 은닉층의 노드 수를 다양하게 설정합니다. 예를 들어, 첫 번째 은닉층에 20개의 노드, 두 번째 은닉층에 15개의 노드를 추가할 수 있습니다.

단계 2: 각 은닉층 사이에 ReLU 활성화 함수를 추가하여 비선형성을 유지합니다.

단계 3: 모델을 다시 학습시키고, 학습이 얼마나 향상되었는지 또는 더 나빠졌는지를 평가합니다. 손실 함수 값이나 시각화를 통해 모델 성능을 비교해 보세요.

실습 문제 2: 학습률과 옵티마이저 변경하기

Adam 옵티마이저는 잘 작동하지만, 다른 옵티마이저와 학습률을 실험하여 모델의 학습 성능을 비교해 보세요.

단계 1: optim.SGD (확률적 경사 하강법)과 같은 다른 옵티마이저로 변경하여 모델을 학습시켜 보세요. 학습률(learning rate)도 0.01, 0.001 등으로 다양하게 설정해 보세요.

단계 2: 각 옵티마이저와 학습률 조합에서의 모델 성능을 기록하고 비교해 보세요. 어떤 경우가 더 빠르게 수렴했는지, 또는 손실 함수 값이 더 낮았는지 확인하세요.

실습 문제 3: 정규화와 드롭아웃 적용하기

모델이 과적합(overfitting)되는 것을 방지하기 위해 드롭아웃(dropout)과 배치 정규화(batch normalization)를 추가해 보세요.

단계 1: 은닉층 사이에 드롭아웃 레이어(nn.Dropout)를 추가하여 과적합을 방지해 보세요. 드롭아웃 비율을 0.2 또는 0.5로 설정하여 다양한 비율을 실험해 보세요.

단계 2: 배치 정규화 레이어(nn.BatchNorm1d)를 은닉층 후에 추가하여 모델의 학습 안정성을 향상시켜 보세요.

단계 3: 드롭아웃과 배치 정규화를 적용한 모델과 적용하지 않은 모델의 성능을 비교해 보세요. 과적합이 얼마나 줄어들었는지, 모델의 일반화 성능이 얼마나 개선되었는지 확인해 보세요.

In [ ]: