

# 심층신경망(DNN)을 이용한 회귀

## 학습목표

- **DNN 회귀 모델의 기본 원리 이해** : 심층 신경망을 사용한 회귀의 기본 개념과 선형 회귀 모델과의 차이점을 설명하고, DNN의 구조가 회귀 분석에 어떻게 적용되는지를 이해할 수 있다.
- **모델 아키텍처 설계** : 다양한 DNN 아키텍처(예: 다층 퍼셉트론, CNN, RNN 등) 중 회귀 문제에 적합한 구조를 설계하고, 각 구조의 장단점을 비교할 수 있다.
- **손실 함수 및 최적화 기법 적용** : DNN 회귀에서 사용되는 손실 함수(예: Mean Squared Error)와 경량화 최적화 기법(예: Adam, RMSprop 등)의 선택 및 효과를 이해하고 적용할 수 있다.
- **정규화 및 드롭아웃 활용** : 과적합 방지를 위한 정규화 기법(예: L1, L2 정규화) 및 드롭아웃 사용의 중요성을 이해하고, 이를 DNN 회귀 모델에 성공적으로 적용할 수 있다.
- **하이퍼파라미터 튜닝** : DNN 회귀 모델의 성능을 극대화하기 위한 하이퍼파라미터 조정 기법(예: 학습률, 배치 크기 등)을 이해하고 실험을 통해 최적의 파라미터를 식별할 수 있다.
- **모델 평가 및 성능 지표 분석** : DNN 회귀 모델의 성능을 평가하기 위한 다양한 지표(예:  $R^2$ , RMSE 등)를 이해하고, 각 지표를 통해 모델의 예측 성능을 분석할 수 있다.
- **실제 데이터셋에서 회귀 문제 해결** : 실제 데이터셋을 사용하여 DNN 회귀 모델을 설계, 구현, 평가하는 전체 프로세스를 경험하고, 문제 해결 능력을 강화할 수 있다.

## 9.1 DNN (Deep Neural Network)

- **DNN(Deep Neural Network)**은 다층 신경망을 기반으로 한 딥러닝 모델입니다.
- 다층이란, 여러 개의 은닉층(hidden layers) 을 가진 신경망을 의미하며, 이러한 모델은 복잡한 패턴을 학습하고 비선형 관계를 모델링하는 데 유리합니다.
- DNN은 기본적인 **인공 신경망(ANN, Artificial Neural Network)** 을 확장한 형태로, 입력 데이터와 출력 사이에 여러 개의 은닉층을 추가하여 더 깊고 복잡한 모델을 만듭니다.
- DNN은 이미지 인식, 음성 처리, 자연어 처리 등 다양한 분야에서 사용됩니다. 특히, 딥러닝(Deep Learning)의 기반 기술로서 최근 많은 발전을 이루었습니다.

## California Housing Dataset

- California Housing Dataset 이용한 회귀 모델

**California Housing Dataset**은 캘리포니아 주의 주택 시장 데이터를 담고 있는 데이터셋으로, 주로 회귀 문제를 다루는 머신러닝 모델을 훈련하고 평가하는 데 사용됩니다. 이 데이터는 1990년 인구조사 데이터를 바탕으로 캘리포니아 주의 특정 지역(지역 단위는 'block group')의 주택 정보를 제공합니다.

### 9.1.1 특징 및 내용

- 데이터셋은 **캘리포니아 지역의 주택 중간 가격(median house value)**을 예측하는 데 사용됩니다.
- 총 20,640개의 데이터 포인트와 8개의 주요 특징(features)이 포함되어 있습니다.

### 9.1.2 주요 컬럼 설명

1. **MedInc (Median Income)**: 해당 지역 가구의 중간 소득 (단위: \$10,000)
2. **HouseAge**: 해당 지역 주택의 중간 연령 (단위: 연도)
3. **AveRooms (Average Rooms)**: 해당 지역 주택당 평균 방 개수
4. **AveBedrms (Average Bedrooms)**: 해당 지역 주택당 평균 침실 개수
5. **Population**: 해당 지역의 총 인구수
6. **AveOccup (Average Occupants)**: 주택당 평균 거주자 수
7. **Latitude**: 해당 지역의 위도
8. **Longitude**: 해당 지역의 경도

### 9.1.3 목표 변수

- **MedHouseVal (Median House Value)**: 해당 지역 주택 중간 가격 (단위: \$)

### 9.1.4 활용 사례

- **회귀 분석**: 중간 주택 가격(MedHouseVal)을 예측하기 위한 회귀 모델 훈련
- **특성 엔지니어링 연습**: 데이터의 특성을 분석하고 새로운 특성을 추가하거나 전처리 방법을 실험
- **지도학습 연습**: 데이터의 레이블(MedHouseVal)을 기반으로 학습하는 알고리즘 적용
- **공간 데이터 분석**: 위도와 경도를 활용하여 지리적 패턴 분석

### 9.1.5 장점

1. **단순성**: 데이터의 구조가 비교적 간단하여 머신러닝 초보자들이 쉽게 이해하고 활용할 수 있음.
2. **실제 데이터**: 현실 세계의 데이터를 기반으로 하므로, 실제 문제 해결 능력을 키우는 데 유용.
3. **다양한 전처리 기법 학습 가능**: 결측치 처리, 이상치 탐지, 데이터 스케일링 등 다양한 전처리 기술을 실험할 수 있음.

### 9.1.6 주의사항

1. **데이터의 불균형**: 특정 특성 값의 분포가 비대칭일 수 있으므로 전처리가 필요할 수 있음.

2. **단순화된 데이터:** 실제 주택 시장 데이터에 비해 단순화된 형태이므로 현실을 완벽히 반영하지는 않음.
3. **1990년 데이터 기반:** 최신 데이터를 반영하지 않으므로 현재의 캘리포니아 주택 시장과는 다를 수 있음.

In [19]: `%pip install seaborn`

```
Requirement already satisfied: seaborn in c:\users\asus\anaconda3\lib\site-packages (0.13.2)
Requirement already satisfied: numpy!=1.24.0,>=1.20 in c:\users\asus\appdata\roaming\python\python312\site-packages (from seaborn) (1.26.3)
Requirement already satisfied: pandas>=1.2 in c:\users\asus\anaconda3\lib\site-packages (from seaborn) (2.2.2)
Requirement already satisfied: matplotlib!=3.6.1,>=3.4 in c:\users\asus\anaconda3\lib\site-packages (from seaborn) (3.9.2)
Requirement already satisfied: contourpy>=1.0.1 in c:\users\asus\anaconda3\lib\site-packages (from matplotlib!=3.6.1,>=3.4->seaborn) (1.2.0)
Requirement already satisfied: cycler>=0.10 in c:\users\asus\anaconda3\lib\site-packages (from matplotlib!=3.6.1,>=3.4->seaborn) (0.11.0)
Requirement already satisfied: fonttools>=4.22.0 in c:\users\asus\anaconda3\lib\site-packages (from matplotlib!=3.6.1,>=3.4->seaborn) (4.51.0)
Requirement already satisfied: kiwisolver>=1.3.1 in c:\users\asus\anaconda3\lib\site-packages (from matplotlib!=3.6.1,>=3.4->seaborn) (1.4.4)
Requirement already satisfied: packaging>=20.0 in c:\users\asus\appdata\roaming\python\python312\site-packages (from matplotlib!=3.6.1,>=3.4->seaborn) (24.1)
Requirement already satisfied: pillow>=8 in c:\users\asus\anaconda3\lib\site-packages (from matplotlib!=3.6.1,>=3.4->seaborn) (10.4.0)
Requirement already satisfied: pyparsing>=2.3.1 in c:\users\asus\anaconda3\lib\site-packages (from matplotlib!=3.6.1,>=3.4->seaborn) (3.1.2)
Requirement already satisfied: python-dateutil>=2.7 in c:\users\asus\anaconda3\lib\site-packages (from matplotlib!=3.6.1,>=3.4->seaborn) (2.9.0.post0)
Requirement already satisfied: pytz>=2020.1 in c:\users\asus\anaconda3\lib\site-packages (from pandas>=1.2->seaborn) (2024.1)
Requirement already satisfied: tzdata>=2022.7 in c:\users\asus\anaconda3\lib\site-packages (from pandas>=1.2->seaborn) (2023.3)
Requirement already satisfied: six>=1.5 in c:\users\asus\appdata\roaming\python\python312\site-packages (from python-dateutil>=2.7->matplotlib!=3.6.1,>=3.4->seaborn) (1.16.0)
Note: you may need to restart the kernel to use updated packages.
```

```
In [1]: import pandas as pd
import seaborn as sns
import matplotlib.pyplot as plt


from sklearn.preprocessing import StandardScaler
from sklearn.datasets import fetch_california_housing
```

```
In [2]: california_housing = fetch_california_housing()
```

```
In [7]: df = pd.DataFrame(california_housing.data, columns=california_housing.feature_names)
df["TARGET"] = california_housing.target
df.tail()
```

Out[7]:

	MedInc	HouseAge	AveRooms	AveBedrms	Population	AveOccup	Latitude	Longitude
20635	1.5603	25.0	5.045455	1.133333	845.0	2.560606	39.48	-121.0
20636	2.5568	18.0	6.114035	1.315789	356.0	3.122807	39.49	-121.2
20637	1.7000	17.0	5.205543	1.120092	1007.0	2.325635	39.43	-121.2
20638	1.8672	18.0	5.329513	1.171920	741.0	2.123209	39.43	-121.3
20639	2.3886	16.0	5.254717	1.162264	1387.0	2.616981	39.37	-121.2




```
In [8]: scaler = StandardScaler()
scaler.fit(df.values[:, :-1])
df.values[:, :-1] = scaler.transform(df.values[:, :-1]).round(4)

df.tail()
```

Out[8]:

	MedInc	HouseAge	AveRooms	AveBedrms	Population	AveOccup	Latitude	Longitude
20635	1.5603	25.0	5.045455	1.133333	845.0	2.560606	39.48	-121.0
20636	2.5568	18.0	6.114035	1.315789	356.0	3.122807	39.49	-121.2
20637	1.7000	17.0	5.205543	1.120092	1007.0	2.325635	39.43	-121.2
20638	1.8672	18.0	5.329513	1.171920	741.0	2.123209	39.43	-121.3
20639	2.3886	16.0	5.254717	1.162264	1387.0	2.616981	39.37	-121.2



## Train Model with PyTorch

```
In [9]: import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
```

```
In [10]: # NumPy 배열을 PyTorch 텐서로 변환
data = torch.from_numpy(df.values).float()

data.shape
```

Out[10]: torch.Size([20640, 9])

```
In [11]: y = data[:, -1:]
x = data[:, :-1]

print(x.shape, y.shape)
```

torch.Size([20640, 8]) torch.Size([20640, 1])

```
In [13]: n_epochs = 100000
learning_rate = 0.0001 #1e-4 모델 학습률
print_interval = 5000
```

## Models 구축

### nn.Module을 사용하여 Model 구축

```
In [14]: # nn.ReLU()와 LeakyReLU 활성화 함수
relu = nn.ReLU()
leaky_relu = nn.LeakyReLU(0.1)
```

```
In [16]: class RegressionNN(nn.Module):

    def __init__(self, input_dim, output_dim):
        self.input_dim = input_dim
        self.output_dim = output_dim

        super().__init__()

        self.linear1 = nn.Linear(input_dim, 3)
        self.linear2 = nn.Linear(3, 3)
        self.linear3 = nn.Linear(3, 3)
        self.linear4 = nn.Linear(3, output_dim)
        self.act = nn.ReLU()

    def forward(self, x):
        # |x| = (batch_size, input_dim)
        h = self.act(self.linear1(x)) # |h| = (batch_size, 3)
        h = self.act(self.linear2(h))
        h = self.act(self.linear3(h))
        y = self.linear4(h)
        # |y| = (batch_size, output_dim)

        return y

model = RegressionNN(x.size(-1), y.size(-1))

model
```

```
Out[16]: RegressionNN(
  (linear1): Linear(in_features=8, out_features=3, bias=True)
  (linear2): Linear(in_features=3, out_features=3, bias=True)
  (linear3): Linear(in_features=3, out_features=3, bias=True)
  (linear4): Linear(in_features=3, out_features=1, bias=True)
  (act): ReLU()
)
```

### nn.Sequential 적용

- nn.Sequential은 PyTorch에서 제공하는 모듈로, 여러 개의 신경망 계층을 순차적으로 연결할 때 사용됩니다. 이를 통해 복잡한 모델을 구성할 때 코드가 간결해지고 직관적으로 작성할 수 있습니다.
- nn.LeakyReLU()는 입력 값이 0 미만일 때도 일정한 기울기를 가지게 만들어, 그래디언트가 0이 되는 문제를 해결합니다.

```
In [17]: model = nn.Sequential(
    nn.Linear(x.size(-1), 3), # 입력 텐서의 마지막 차원 크기(x.size(-1))를 3개의 출력 차
    nn.LeakyReLU(),
    nn.Linear(3, 3),
    nn.LeakyReLU(),
    nn.Linear(3, 3),
    nn.LeakyReLU(),
    nn.Linear(3, 3),
    nn.LeakyReLU(),
    nn.Linear(3, y.size(-1)),
)

model
```

```
Out[17]: Sequential(
  (0): Linear(in_features=8, out_features=3, bias=True)
  (1): LeakyReLU(negative_slope=0.01)
  (2): Linear(in_features=3, out_features=3, bias=True)
  (3): LeakyReLU(negative_slope=0.01)
  (4): Linear(in_features=3, out_features=3, bias=True)
  (5): LeakyReLU(negative_slope=0.01)
  (6): Linear(in_features=3, out_features=3, bias=True)
  (7): LeakyReLU(negative_slope=0.01)
  (8): Linear(in_features=3, out_features=1, bias=True)
)
```

```
In [18]: optimizer = optim.SGD(model.parameters(),
    lr=learning_rate)
```

```
In [19]: for i in range(n_epochs):
    y_hat = model(x)
    loss = F.mse_loss(y_hat, y)

    optimizer.zero_grad()
    loss.backward()

    optimizer.step()

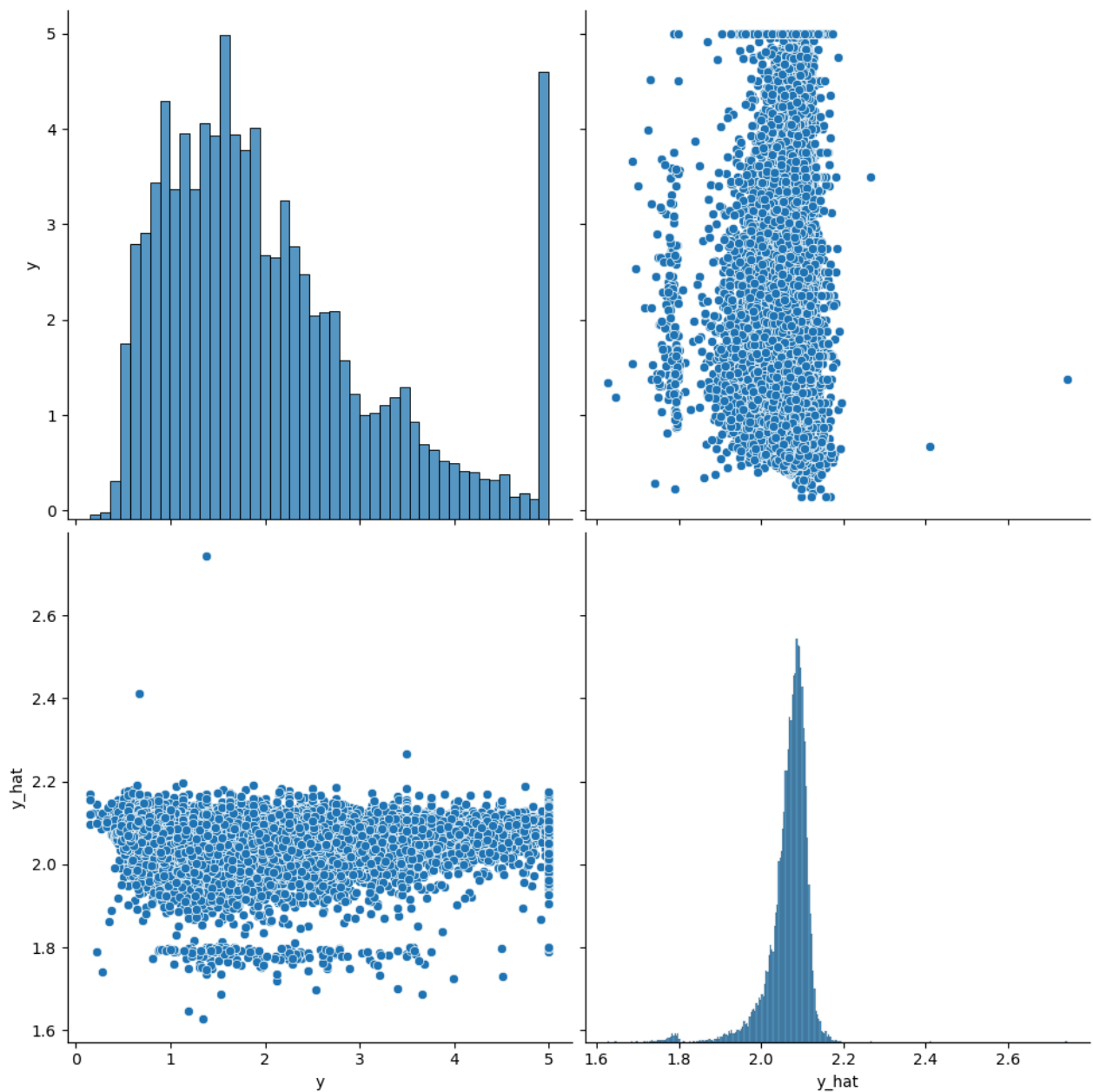
    if (i + 1) % print_interval == 0:
        print('Epoch %d: loss=%.4e' % (i + 1, loss))
```

```
Epoch 5000: loss=1.5517e+00
Epoch 10000: loss=1.4154e+00
Epoch 15000: loss=1.3565e+00
Epoch 20000: loss=1.3365e+00
Epoch 25000: loss=1.3334e+00
Epoch 30000: loss=1.3329e+00
Epoch 35000: loss=1.3327e+00
Epoch 40000: loss=1.3325e+00
Epoch 45000: loss=1.3324e+00
Epoch 50000: loss=1.3322e+00
Epoch 55000: loss=1.3321e+00
Epoch 60000: loss=1.3320e+00
Epoch 65000: loss=1.3318e+00
Epoch 70000: loss=1.3317e+00
Epoch 75000: loss=1.3316e+00
Epoch 80000: loss=1.3316e+00
Epoch 85000: loss=1.3315e+00
Epoch 90000: loss=1.3314e+00
Epoch 95000: loss=1.3313e+00
Epoch 100000: loss=1.3312e+00
```

## 결과

```
In [20]: # detach() 텐서 객체의 기울기 추적 중지
df = pd.DataFrame(torch.cat([y, y_hat], dim=1).detach().numpy(),
                  columns=["y", "y_hat"])

sns.pairplot(df, height=5)
plt.show()
```



## 회귀 모델 만들기

```
In [21]: import torch
import torch.nn as nn
import torch.optim as optim
import numpy as np
from sklearn.datasets import fetch_california_housing
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from torch.utils.data import DataLoader, TensorDataset
```

```
In [28]: # California Housing 데이터셋 로드
california_housing = fetch_california_housing()
X = california_housing.data
y = california_housing.target

# 데이터 전처리
scaler = StandardScaler()
X_scaled = scaler.fit_transform(X) # 특성 스케일링
```



```

# 훈련 데이터와 테스트 데이터 나누기
X_train, X_test, y_train, y_test = train_test_split(X_scaled, y, test_size=0.2, random

# Tensor로 변환
X_train_tensor = torch.tensor(X_train, dtype=torch.float32)
X_test_tensor = torch.tensor(X_test, dtype=torch.float32)
y_train_tensor = torch.tensor(y_train, dtype=torch.float32).view(-1, 1)
y_test_tensor = torch.tensor(y_test, dtype=torch.float32).view(-1, 1)

# DataLoader 설정
train_dataset = TensorDataset(X_train_tensor, y_train_tensor)
test_dataset = TensorDataset(X_test_tensor, y_test_tensor)

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

# 모델 정의
class RegressionNN(nn.Module):
    def __init__(self):
        super(RegressionNN, self).__init__()
        self.fc1 = nn.Linear(8, 64) # California Housing 데이터는 8개의 특성
        self.fc2 = nn.Linear(64, 32)
        self.fc3 = nn.Linear(32, 1) # 회귀 문제이므로 출력이 하나

    def forward(self, x):
        x = torch.relu(self.fc1(x)) # 첫 번째 은닉층
        x = torch.relu(self.fc2(x)) # 두 번째 은닉층
        x = self.fc3(x) # 출력층
        return x

# 모델 인스턴스
model = RegressionNN()

# 손실 함수와 옵티마이저 설정
criterion = nn.MSELoss() # 회귀 문제는 MSELoss 사용
optimizer = optim.Adam(model.parameters(), lr=0.001)

# 훈련 함수
def train(model, train_loader, criterion, optimizer, epochs=10):
    model.train() # 훈련 모드
    for epoch in range(epochs):
        running_loss = 0.0
        for inputs, labels in train_loader:
            optimizer.zero_grad() # 기울기 초기화
            outputs = model(inputs) # 순전파
            loss = criterion(outputs, labels) # 손실 계산
            loss.backward() # 역전파
            optimizer.step() # 가중치 업데이트

        running_loss += loss.item()

    # 에포크마다 손실 출력
    print(f'Epoch {epoch+1}/{epochs}, Loss: {running_loss/len(train_loader)}')

# 훈련 시작
train(model, train_loader, criterion, optimizer, epochs=10)

```

```
Epoch 1/10, Loss: 1.7965990896372832
Epoch 2/10, Loss: 0.4769954400122628
Epoch 3/10, Loss: 0.42419762308745423
Epoch 4/10, Loss: 0.3999241653454396
Epoch 5/10, Loss: 0.38433402742064277
Epoch 6/10, Loss: 0.3785434387797533
Epoch 7/10, Loss: 0.3688494184220484
Epoch 8/10, Loss: 0.3639181480966797
Epoch 9/10, Loss: 0.35714673418407294
Epoch 10/10, Loss: 0.3502311960382517
```

```
In [29]: # 테스트 함수
def test(model, test_loader):
    model.eval() # 평가 모드
    all_preds = []
    all_labels = []
    with torch.no_grad(): # 기울기 계산을 하지 않음
        for inputs, labels in test_loader:
            outputs = model(inputs)
            all_preds.append(outputs)
            all_labels.append(labels)

    # 결과를 텐서에서 numpy로 변환
    all_preds = torch.cat(all_preds).numpy()
    all_labels = torch.cat(all_labels).numpy()

    # 성능 평가: 평균 제곱 오차 (MSE) 계산
    mse = np.mean((all_preds - all_labels) ** 2)
    print(f'Mean Squared Error on test data: {mse:.4f}')

# 테스트 시작
test(model, test_loader)
```

Mean Squared Error on test data: 0.3584

## ■ 학습 결과

### • train 학습

- "Loss"는 손실 함수 값으로, 모델이 예측한 값과 실제 값 간의 차이를 나타냅니다. 여기서 손실 값은 약 0.3197입니다. 손실 값이 낮을수록 모델이 학습 데이터를 잘 예측하는 것을 의미합니다. 일반적으로 이 값이 0에 가까울수록 모델의 성능이 좋은 것입니다.
- 에폭 수가 10번이라는 것은 모델이 데이터를 10번 학습하면서 손실을 최소화하려고 했다는 의미입니다. 만약 10번의 학습 후에도 손실 값이 여전히 상대적으로 높다면, 모델을 더 학습하거나 다른 개선 방법을 시도할 필요가 있을 수 있습니다.

### • test 학습

- MSE(평균 제곱 오차)는 모델의 예측값과 실제 값 간의 차이를 제곱한 후 평균을 낸 값입니다. 이 값은 모델이 얼마나 정확하게 예측했는지를 나타내며, 값이 낮을수록 더 좋은 성능을 의미합니다.
- 여기서 MSE 값이 0.3380이라면, 테스트 데이터에서 모델의 예측값과 실제 값 간에 평균적으로 0.3380 정도의 오차가 있다는 의미입니다.
- test 데이터에서 MSE는 0.3380으로, 모델이 테스트 데이터에 대해서도 어느 정도 정확하게 예측했음을 나타냅니다.

## 9.2.1 간단한 DNN 예제 코드

California Housing Dataset이나 비슷한 구조의 회귀 작업에 사용할 수 있는 선형 데이터를 예측하기 위한 예제

## 주요 특징

### 9.2.1 데이터 생성 및 전처리

- `make_regression`을 사용해 간단한 회귀 데이터를 생성.
- `StandardScaler`로 정규화.

### 9.2.2 모델 아키텍처

- 2개의 히든 레이어와 ReLU 활성화 함수를 사용.
- `nn.Sequential`로 간단하게 구성.

### 9.2.3 손실 함수 및 옵티마이저

- 회귀 문제에 적합한 MSE 손실 함수를 사용.
- Adam 옵티마이저로 학습.

### 9.2.4 데이터로더

- `TensorDataset`과 `DataLoader`로 배치 학습 구현.

이 코드는 회귀 문제에 적합하며, `California Housing Dataset` 에도 적용 가능합니다.

데이터를 직접 로드하고 전처리하는 경우에는

`sklearn.datasets.fetch_california_housing` 을 사용할 수 있습니다.

```
In [30]: import torch
import torch.nn as nn
import torch.optim as optim
from sklearn.datasets import make_regression
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
import matplotlib.pyplot as plt

# 데이터 생성
X, y = make_regression(n_samples=1000, n_features=10, noise=0.1, random_state=42)
y = y.reshape(-1, 1)

# 데이터 정규화
scaler_X = StandardScaler()
scaler_y = StandardScaler()

X = scaler_X.fit_transform(X)
y = scaler_y.fit_transform(y)
```

```

# PyTorch 텐서로 변환
X = torch.tensor(X, dtype=torch.float32)
y = torch.tensor(y, dtype=torch.float32)

# Train/Test Split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=

# 데이터로더
train_dataset = torch.utils.data.TensorDataset(X_train, y_train)
test_dataset = torch.utils.data.TensorDataset(X_test, y_test)

train_loader = torch.utils.data.DataLoader(train_dataset, batch_size=32, shuffle=True)
test_loader = torch.utils.data.DataLoader(test_dataset, batch_size=32, shuffle=False)

# DNN 모델 정의
class DNN(nn.Module):
    def __init__(self, input_dim, hidden_dim, output_dim):
        super(DNN, self).__init__()
        self.model = nn.Sequential(
            nn.Linear(input_dim, hidden_dim),
            nn.ReLU(),
            nn.Linear(hidden_dim, hidden_dim),
            nn.ReLU(),
            nn.Linear(hidden_dim, output_dim)
        )

    def forward(self, x):
        return self.model(x)

# 모델 초기화
input_dim = X_train.shape[1]
hidden_dim = 64
output_dim = 1
model = DNN(input_dim, hidden_dim, output_dim)

# 손실 함수 및 최적화 방법 설정
criterion = nn.MSELoss() # 평균 제곱 오차
optimizer = optim.Adam(model.parameters(), lr=0.001)

# 훈련 과정
epochs = 100
train_losses = []
test_losses = []

for epoch in range(epochs):
    model.train() # 훈련 모드
    running_loss = 0.0
    for inputs, targets in train_loader:
        optimizer.zero_grad()
        outputs = model(inputs)
        loss = criterion(outputs, targets)
        loss.backward()
        optimizer.step()
        running_loss += loss.item()

    train_losses.append(running_loss / len(train_loader))

# 검증
model.eval() # 평가 모드

```

```

with torch.no_grad():
    val_loss = 0.0
    for inputs, targets in test_loader:
        outputs = model(inputs)
        loss = criterion(outputs, targets)
        val_loss += loss.item()
    test_losses.append(val_loss / len(test_loader))

if epoch % 10 == 0:
    print(f"Epoch [{epoch+1}/{epochs}], Train Loss: {train_losses[-1]:.4f}, Test Loss: {val_loss:.4f}")

# 훈련 및 테스트 손실 시각화
plt.plot(range(epochs), train_losses, label="Train Loss")
plt.plot(range(epochs), test_losses, label="Test Loss")
plt.xlabel("Epochs")
plt.ylabel("Loss")
plt.title("Training and Testing Loss")
plt.legend()
plt.show()

# 모델 예측 시각화
model.eval()
with torch.no_grad():
    predictions = model(X_test).numpy()

plt.scatter(y_test, predictions)
plt.xlabel("True Values")
plt.ylabel("Predictions")
plt.title("True vs Predicted Values")
plt.show()

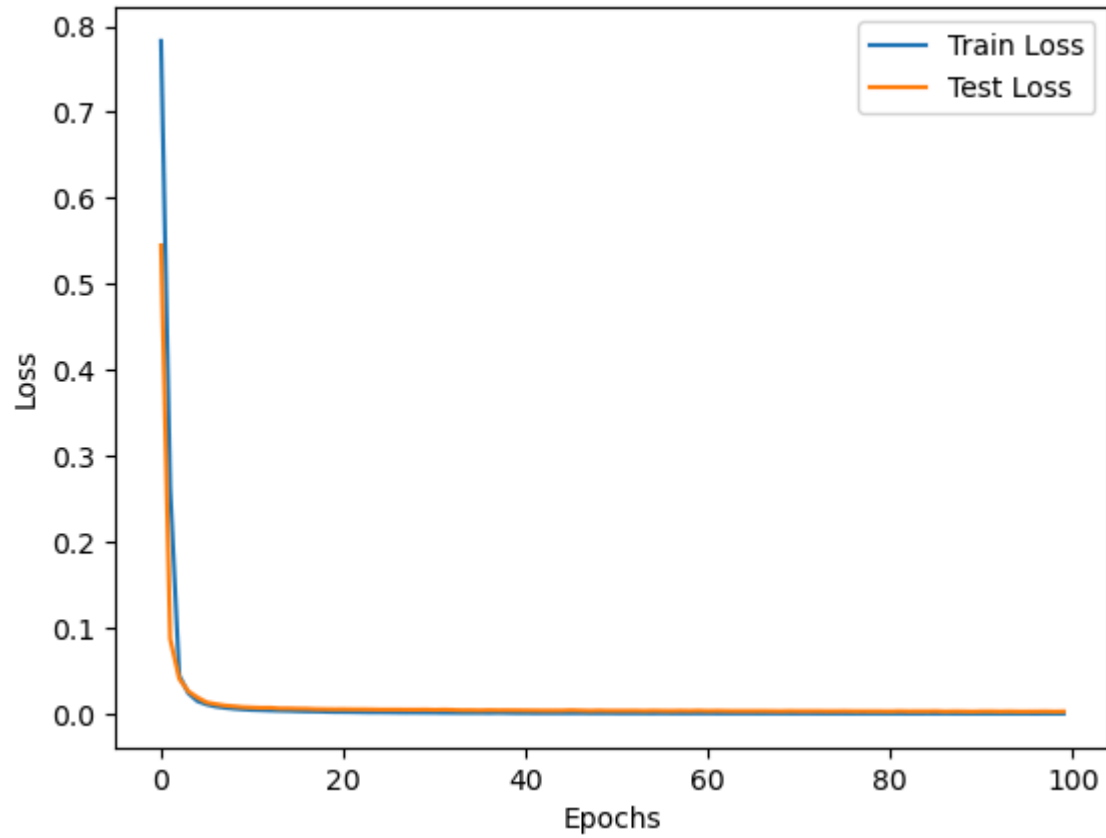
```

```

Epoch [1/100], Train Loss: 0.7829, Test Loss: 0.5450
Epoch [11/100], Train Loss: 0.0046, Test Loss: 0.0073
Epoch [21/100], Train Loss: 0.0021, Test Loss: 0.0053
Epoch [31/100], Train Loss: 0.0014, Test Loss: 0.0046
Epoch [41/100], Train Loss: 0.0008, Test Loss: 0.0040
Epoch [51/100], Train Loss: 0.0006, Test Loss: 0.0036
Epoch [61/100], Train Loss: 0.0005, Test Loss: 0.0033
Epoch [71/100], Train Loss: 0.0004, Test Loss: 0.0031
Epoch [81/100], Train Loss: 0.0003, Test Loss: 0.0028
Epoch [91/100], Train Loss: 0.0002, Test Loss: 0.0029

```

Training and Testing Loss



True vs Predicted Values

