1. 필요한 라이브러리 불러오기

```
import torch, torch.nn as nn
from torch.utils.data import TensorDataset, DataLoader
from torch.optim.lr_scheduler import StepLR
```

- 'torch': PyTorch의 핵심 라이브러리
- 'torch.nn': 신경망 구성 요소를 정의하는 모듈
- 'TensorDataset': 여러 텐서를 결합하여 하나의 데이터셋을 만드는 역할
  일반적으로 입력 데이터와 레이블을 결합하는 데 사용함
- 'DataLoader': 'TensorDataset'와 같은 데이터셋을 미니 배치 단위로 나누어 모델에 효율적으로 공급할 수 있도록 도와줌. 또한, 데이터셋을 셔플하거나 병렬 처리를 통해 데이터 로드를 빠르게 할 수 있음.
- 'stepLR': 학습률 조정을 위한 스케줄러

2. 신경망 모델 정의

```
D_i, D_k, D_o = 10, 40, 5
```

- 입력 크기=10, 은닉층 크기=40, 출력 크기=5

```
model = nn.Sequential(
    nn.Linear(D_i, D_k),
    nn.ReLU(),
    nn.Linear(D_k, D_k),
    nn.ReLU(),
    nn.Linear(D_k, D_o))
```

- 'nn.Sequent`ial': 여러 계층을 순차적으로 쌓아놓은 컨테이너.
- 'nn.Linear': 신경망 모듈에서 사용되는 선형변환 또는 dense layer을 정의하는 클래스.
  이 계층은 입력 텐서에 선형 변환을 적용하여 출력 텐서를 생성함.
- 'nn.ReLU': 활성화 함수로 ReLU를 사용

-> 두 개의 은닉층을 가진 모델 생성

3. 초기화 함수 정의 및 적용

```
def weights_init(layer_in):
    if isinstance(layer_in, nn.Linear):
        nn.init.kaiming_uniform(layer_in.weight)
        layer_in.bias.data.fill_(0.0)
model.apply(weights_init)
```

- 'weight_init': 신경망 모델의 가중치를 초기화하는 함수.
- 'isinstance': 계층 타입을 확인함. 입력된 계층이 'nn.Linear'안지 확인
- 'Kaiming_uniform': He 초기화의 균등분포 초기화.
  He 초기화란 신경망에서 ReLU 활성화 함수를 사용할 때 흔히 발생하는 그래디언트 소실 문제를 완화하기 위해 개발.
  각 층의 가중치를 입력 노드 수의 역수에 비례하는 분산을 가진 분포에서 무작위로 선택하여 초기화 함.
- 'layer_in.bias.data.fill_(0.0)': 'layer_in'계층의 편향을 0으로 초기화
- 'model.apply(weight_init)':모델의 모든 계층에 초기화 함수 적용

4. 손실 함수와 옵티마이저 정의

```
criterion = nn.MSELoss()
optimizer = torch.optim.SGD(model.parameters(), lr = 0.1, momentum=0.9)
scheduler = StepLR(optimizer, step_size=10, gamma=0.5)
```

- 'nn.MSELoss()': 평균 제곱 오차를 구하는 loss function 이 때 input과 target의 shape는 같아야함.
- 'optimizer=': 확률적 경사 하강법(SGD)를 사용하며, 초기 학습률은 0.1이고 모멘텀은 0.9로 정의
- 'stepLR()': 학습률을 일정한 에포크마다 감소시키는 학습률 스케줄러(사용자 정의)
  매 10포크마다 학습률을 0.5 즉, 절반으로 감소

5. 데이터 생성 및 DataLoader 생성

```
x = torch.randn(100, D_i)
y = torch.randn(100, D_o)
data_loader = DataLoader(TensorDataset(x,y), batch_size=10, shuffle=True)
```

- 'x=': 100개의 행과 10개의 열로 구성된 입력데이터를 무작위 생성
- 'y=': 100개의 행과 5개의 열로 구성된 출력데이터를 무작위 생성
- 'data_loader=': 'TensorDataset'과 'DataLoader'를 사용하여 데이터를 미니 배치로 나누고, 섞어서 생성

6. 훈련 루프

```
for epoch in range(100):
    epoch_loss = 0.0
    for i, data in enumerate(data_loader):
        x_batch, y_batch = data
        optimizer.zero_grad()
        pred = model(x_batch)
        loss = criterion(pred, y_batch)
        loss.backward()
        optimizer.step()
        epoch_loss += loss.item()
    print(f'Epoch {epoch:5d}, loss {epoch_loss:.3f}')
    scheduler.step()
```

• 'for epoch in range(100): 데이터셋을 100번의 에포크 동안 훈련
• 'epoch_loss =0.0': 에포크가 시작될 때마다 0으로 초기화되고, 배치단위로 손실을 누적하여 저장함.
• 'for i, data in enumerate(data_loader):' : 각 배치에 대해 동시에 인덱스 'i'와 데이터를 'data' 로 반환.
• 'optimizer.zero_grad()': 각 배치에 대해 역전파를 시작하기 전에 옵티마이저의 기울기를 0으로 초기화.

순전파

• 'pred = model(x_batch)': 모델에 입력 데이터를 전달하여 예측 값 얻음.
• 'loss = criterion(pred, y_batch)': 예측 값과 'y_batch' 사이의 손실 계산.

역전파

• 'loss.backward()': 역전파를 통해 각 매개변수에 대한 손실의 기울기 계산.

• 'optimizer.step()': 계산된 기울기를 사용하여 매개변수 업데이트.
• 'epoch_loss += loss.item()': '.item()'을 사용하여 텐서를 스탈라 값으로 변환
현재 에포트의 총 손실에 현재 배치의 손실을 더함
이 과정을 통해 한 에포크 동안 발생한 모든 배치의 손실을 누적하여 총 손실을 계산가능
• 'print(f'Epoch {epoch:5d}, loss {epoch_loss:.3f}')': 에포크가 끝날 때마다 총 손실 출력.