



week4_pytorch

<참고자료>

<https://velog.io/@stresszero/pytorch-workflow>

https://youtu.be/k60oT_8lyFw?si=nSWFDFG4mgFicTee

PyTorch란??

- 개요
 - 메타 AI에서 개발한 파이썬 기반의 오픈 소스 딥러닝 프레임워크.
 - 현재는 메타로부터 독립하여 파이토치 재단에서 프로젝트 관리.
 - 현재 가장 인기 있는 딥러닝 프레임워크.
- 특징
 - 동적 계산 그래프(Dynamic computational graph): 계산 그래프를 동적으로 생성하여 코드 실행 중에 그래프가 생성되기 때문에 정적 계산 그래프보다 디버깅과 실험이 용이하다는 장점이 있다.
 - 쉽고 직관적인 Interface: 파이썬과 유사한 문법을 제공하여 파이썬스러운 코딩을 하기 쉬움.
 - 강력한 커뮤니티 및 생태계: 파이토치를 지원하는 다양한 오픈 소스 프로젝트와 도구들이 있음.
 - 하드웨어 가속 지원: 엔비디아의 CUDA 지원하며 GPU나 TPU 같은 하드웨어 활용하여 연산 속도 대폭 향상시킴.
- 구성 요소
 - torch: main namespace, tensor 등의 다양한 수학 함수 포함.
 - torch.autograd: 자동 미분 기능을 제공하여 역전파 계산을 자동화함.
 - torch.nn: 신경망 구축을 위한 다양한 레이어와 모델 구성 요소 제공.
 - torch.optim: SGD를 비롯한 다양한 최적화 알고리즘을 지원함.
 - torch.utils: 데이터 조작 등 유틸리티 기능을 제공함.

텐서 Tensor

- 개요
 - 파이토치의 기본 데이터 구조로써, 다차원 배열을 의미.
 - NumPy의 ndarray와 유사하지만, **GPU를 활용하여 연산 가속이 가능함**.
 - 차원에 따라 스칼라(0D), 벡터(1D), 행렬(2D) 등으로 분류되고 고차원 텐서도 지원함.
- 텐서 초기화 및 데이터 타입
 - 초기화되지 않은 텐서: `torch.empty(size)`를 사용하여 메모리 공간만 할당함.
 - 무작위로 초기화된 텐서: `torch.rand(size)`를 사용하여 0 이상 1미만 범위의 균등 분포로 초기화함.
 - 지정한 값으로 초기화된 텐서: `torch.full(size, value)`를 사용하여 모든 요소를 특정 값으로 초기화함.
 - dtype 인자를 통해 텐서의 데이터 타입을 지정할 수 있고 `torch.float32`, `torch.int64` 등 다양한 타입을 지원함.
- 텐서 연산: 산술 연산, 행렬 연산(`torch.matmul()`), 인덱싱 및 슬라이싱 연산(텐서의 특정 요소나 부분 접근)이 가능함.
- 자동 미분

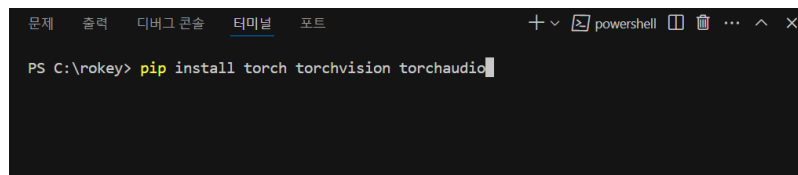
`torch.autograd`: 신경망 학습 시 역전파를 자동으로 계산하여 사용자는 손쉽게 모델의 parameter에 대한 gradient를 계산하고, 최적화에 활용할 수 있음.

- **신경망 구축(패턴 인식, 비선형 문제 해결, 특징 학습)**
 - 신경망의 구조
 - 입력층: 입력 데이터들이 신경망에 전달된다.
 - 은닉층: 데이터의 특징을 학습한다. 여러 개의 은닉층이 쌓일수록 더 복잡한 데이터 패턴을 학습할 수 있다.
 - 출력층에서는 신경망의 최종 예측(확률)을 제공한다.
 - `torch.nn`: 모듈을 사용하여 신경망 구축 가능.
 - `nn.Module`을 상속하여 모델 정의.
 - 다양한 레이어(`nn.Linear`, `nn.Conv2d`)들을 조합하여 복잡한 신경망 설계 가능.

- 최적화(Optimization)
 - torch.optim: 다양한 최적화 알고리즘 제공.
 - torch.optim.SGD: 확률적 경사 하강법 적용하여 옵티마이저에 모델의 parameter와 학습률 등을 지정하여 학습 진행.
- 데이터 로딩 및 전처리
 - torch.utils.data 모듈의 DataLoader와 Dataset 클래스를 활용하여 대용량 데이터를 효율적으로 load 및 전처리 가능.
 - patch 단위의 데이터 로딩, 셔플링, 병렬 처리 가능.
- 모델 저장 및 로드
 - torch.save(): 학습된 모델 저장.
 - torch.load(): 모델 로드.

PyTorch 사용

- pytorch 설치하기



```
PS C:\rokey> pip install torch torchvision torchaudio
```

```
#잘 설치되었는지 버전 확인
import torch
print(torch.__version__)
```

- 텐서 생성

```
import torch

# 1. 초기화되지 않은 텐서 생성
x = torch.empty(3, 3)
print(x)

# 2. 랜덤 값으로 초기화된 텐서
```

```
x = torch.rand(3, 3)
print(x)
```

```
# 3. 0으로 채워진 텐서
x = torch.zeros(3, 3)
print(x)
```

```
# 4. 특정 값으로 채워진 텐서
x = torch.full((3, 3), 7)
print(x)
```

```
# 5. 리스트로부터 텐서 생성
x = torch.tensor([1.0, 2.0, 3.0])
print(x)
```

```
장원\.vscode\extensions\ms-python.debugpy-2025.0.0-win32-x64\
tensor([[0., 0., 0.],
        [0., 0., 0.],
        [0., 0., 0.]])
tensor([[0.1500, 0.6029, 0.7127],
        [0.6602, 0.8453, 0.7170],
        [0.8784, 0.2156, 0.2598]])
tensor([[0., 0., 0.],
        [0., 0., 0.],
        [0., 0., 0.]])
tensor([[7, 7, 7],
        [7, 7, 7],
        [7, 7, 7]])
tensor([1., 2., 3.])
PS C:\rokey> █
```

- 텐서 속성 확인

```
import torch
```

```
x = torch.rand(3,3)
print(x.shape) # 텐서 크기
print(x.dtype) # 데이터 타입
print(x.device) # CPU 또는 GPU 정보
```

```
조_스터디\4주차\pytorch_study.py'
torch.Size([3, 3])
torch.float32
cpu
PS C:\rokey> █
```

- 텐서 연산

```
#텐서 연산
import torch

x = torch.rand(3, 3)
y = torch.rand(3, 3)

print(x + y) # 덧셈
print(x - y) # 뺄셈
print(x * y) # 요소별 곱
print(x @ y) # 행렬 곱 (또는 torch.matmul(x, y))
```

```
조_스터디\4주차\tensor3.py
tensor([[1.3064, 0.7333, 0.9298],
        [1.8884, 0.7259, 1.1162],
        [1.6395, 0.7145, 1.1774]])
tensor([[ 0.1081,  0.2535, -0.4507],
        [-0.1009, -0.6845, -0.2836],
        [ 0.1481, -0.6030,  0.0148]])
tensor([[0.4238, 0.1184, 0.1654],
        [0.8889, 0.0146, 0.2914],
        [0.6665, 0.0367, 0.3465]])
tensor([[1.0931, 0.6754, 0.9728],
        [0.8665, 0.5032, 0.8734],
        [1.0355, 0.6464, 1.0025]])
PS C:\rokey> █
```

- GPU로 텐서 이동

```
device = "cuda" if torch.cuda.is_available() else "cpu"
x = x.to(device) # GPU로 이동
```

- 자동 미분(Autograd)

```
import torch
```

#3개의 원소를 가지는 텐서 생성.

#각 원소는 정규분포(mean=0, std=1)로 초기화 됨.

`x = torch.randn(3, requires_grad=True)` # 기울기(Gradient) 계산 활성화

#y는 x와 동일한 크기의 텐서이며 x의 원소에 2 곱한 값을 원소로 가짐.

`y = x * 2`

#z는 텐서 y의 모든 원소의 평균 계산.

`z = y.mean()`

`z.backward()` # 역전파 실행하여 기울기 계산.

`print(x.grad)` # x에 대한 미분 값 출력

자동미분 (Autograd)

$$y = 2x \rightarrow dy/dx = 2 \text{ (배율)}$$

$$z = y.mean() \rightarrow dz/dy = 1/3$$

즉, z는 y에 대해 동일한 가중치 = 동일한 기울기인 1/3 가짐.

$$\Rightarrow \frac{dz}{dx} = \frac{dz}{dy} \times \frac{dy}{dx} = \frac{1}{3} \times 2 = 2/3$$

$$\therefore x.grad = [2/3, 2/3, 2/3]$$

```
PS C:\rokey> cd C:\rokey ; & C:\anaconda\python
조_스터디\4주차\autograd.py'
tensor([0.6667, 0.6667, 0.6667])
PS C:\rokey>
```

⇒ 역전파는 미분을 계산하여 기울기(gradient)를 추적하는 과정이며 기울기를 통해 신경망의 가중치를 조정할 수 있음.

- 신경망 구축(torch.nn)

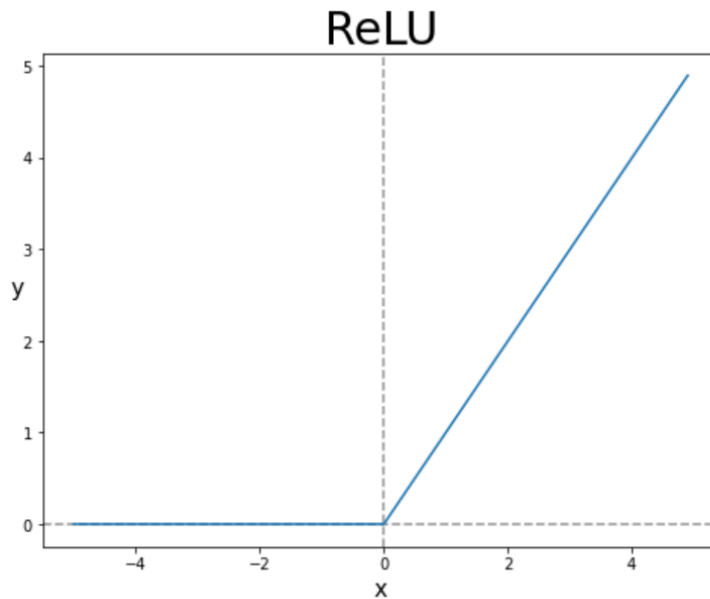
```
import torch.nn as nn #pytorch의 신경망 모듈
import torch.nn.functional as F #활성함수와 같은 함수형 API

#PyTorch에서 모든 신경망 클래스는 nn.Module을 상속해야 함.
#nn.Module: 신경망 정의, 파라미터 추적, forward() method 정의
class SimpleNN(nn.Module):
    def __init__(self):
        super(SimpleNN, self).__init__() #부모 클래스 nn.Module 호출
        #첫번째 완전 연결층 정의
        self.fc1 = nn.Linear(3, 3) # 입력 차원 3 -> 출력 차원 3
        self.fc2 = nn.Linear(3, 1) # 입력 차원 3 -> 출력 차원 1

        #신경망의 forward pass 정의
    def forward(self, x):
        #self.fc1(x): 첫번째 완전 연결층 통과한 결과
        x = F.relu(self.fc1(x)) # ReLU활성화 함수 적용
        x = self.fc2(x) #두 번째 완전 연결층 통과 결과.
        return x

model = SimpleNN()
print(model)
```

```
SimpleNN(
  (fc1): Linear(in_features=3, out_features=3, bias=True)
  (fc2): Linear(in_features=3, out_features=1, bias=True)
)
PS C:\rokey>
```



ReLU 함수는 음수 값을 0으로 만들고, 양수 값은 그대로 유지하는 함수.

• 모델 학습(Optimization)

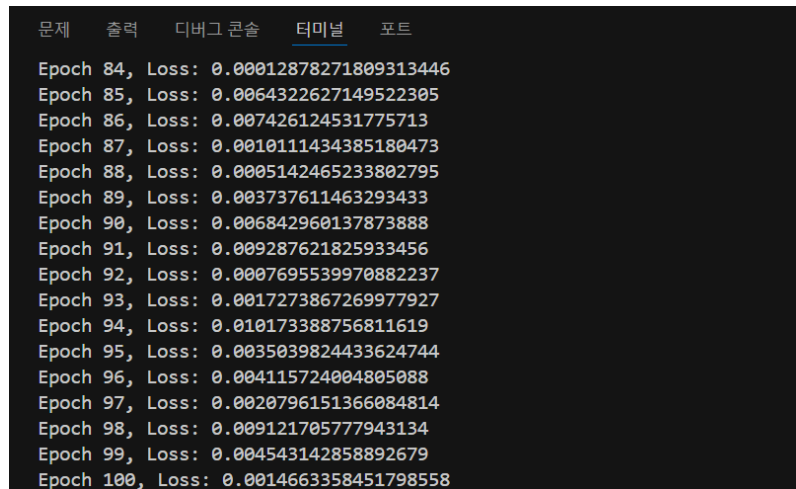
```
#손실 함수 정의
#예측값과 실제 값 간의 차이 제곱 후 평균 계산.
criterion = nn.MSELoss() # 평균 제곱 오차 (MSE)

#옵티마이저 설정
#SGD (확률적 경사 하강법) 활용하여 모델 가중치 update.
#model.parameters(): 모델의 모든 학습 가능한 가중치 반환.
#lr=학습률
optimizer = torch.optim.SGD(model.parameters(), lr=0.01)

#학습 과정
#epoch: 데이터셋을 모델이 한번 완전 학습하는 과정
#각 에폭마다 입력값, 예측값, 손실계산, 역전파, 가중치 update 이루어짐.
for epoch in range(100):
    #100번 반복
    optimizer.zero_grad() # 기울기 초기화
    outputs = model(torch.rand(3)) # 모델 예측
    # 예측값 outputs과 실제값 torch.tensor([1.0]) 간 손실 계산
    loss = criterion(outputs, torch.tensor([1.0])) #모델이 1.0 예측하도록 학습.
    loss.backward() # 역전파 수행(기울기 계산)
    optimizer.step() # 가중치 업데이트
```



```
print(f"Epoch {epoch+1}, Loss: {loss.item()}")
```



문제	출력	디버그 콘솔	터미널	포트
	Epoch 84, Loss: 0.00012878271809313446			
	Epoch 85, Loss: 0.0064322627149522305			
	Epoch 86, Loss: 0.007426124531775713			
	Epoch 87, Loss: 0.0010111434385180473			
	Epoch 88, Loss: 0.0005142465233802795			
	Epoch 89, Loss: 0.003737611463293433			
	Epoch 90, Loss: 0.006842960137873888			
	Epoch 91, Loss: 0.009287621825933456			
	Epoch 92, Loss: 0.0007695539970882237			
	Epoch 93, Loss: 0.0017273867269977927			
	Epoch 94, Loss: 0.010173388756811619			
	Epoch 95, Loss: 0.0035039824433624744			
	Epoch 96, Loss: 0.004115724004805088			
	Epoch 97, Loss: 0.0020796151366084814			
	Epoch 98, Loss: 0.009121705777943134			
	Epoch 99, Loss: 0.004543142858892679			
	Epoch 100, Loss: 0.0014663358451798558			

→ epoch마다 손실 값이 출력되며 모델이 학습하는 모습. 손실 값이 점차적으로 감소하면서 모델이 주어진 목표 값 1.0에 가까워지는 모습 확인 가능.

• 데이터 로딩

```
from torch.utils.data import Dataset, DataLoader
```

```
class CustomDataset(Dataset):
```

```
    def __init__(self):
```

```
        #torch.arange(10): 0~9까지 정수 생성
```

```
        #dtype=torch.float32: 데이터를 32bit float 형식으로 변환
```

```
        self.data = torch.arange(10, dtype=torch.float32).unsqueeze(1)
```

```
        #data = [0.,1.,2.,...,9.] # (10,) 텐서 생성
```

```
        #이후 .unsqueeze(1): 텐서의 2번째 차원에 크기 1을 추가하여 (10,1) 크기의
```

```
        #[[0.],[1.],[2.],...[9.]]
```

```
    def __len__(self):
```

```
        return len(self.data) # 데이터 개수(데이터셋 길이) = 10
```

```
    def __getitem__(self, index):
```

```
        return self.data[index], self.data[index] * 2 # 입력값, 정답값 반환
```

```
        #인덱스 0에 대해 self.data[0]=[0.]→ 정답값은 0.*2=0
```

```
        #인덱스 1에 대해 self.data[1]=[1.]→ 정답값은 1.*2=2
```

```
dataset = CustomDataset()
#DataLoader: 주어진 데이터셋을 batch 단위로 데이터 로드하는 클래스.
#batch_size=2: dataset에서 한번에 load할 batch 크기 설정(매번 두 개의 샘플 반환)
#shuffle=True: 데이터를 무작위로 섞어 학습에 도움이 됨.
dataloader = DataLoader(dataset, batch_size=2, shuffle=True)

#배치 단위로 반복
for x, y in dataloader:
    print(f"Input: {x}, Target: {y}")
```

```
Input: tensor([[0.],
               [1.]]), Target: tensor([[0.],
               [2.]])
Input: tensor([[7.],
               [5.]]), Target: tensor([[14.],
               [10.]])
Input: tensor([[3.],
               [4.]]), Target: tensor([[6.],
               [8.]])
Input: tensor([[2.],
               [6.]]), Target: tensor([[ 4.],
               [12.]])
Input: tensor([[9.],
               [8.]]), Target: tensor([[18.],
               [16.]])
PS C:\rokey>
```

- 모델 저장 및 로드

```
torch.save(model.state_dict(), "C:/rokey/6조_스터디/4주차/ex_model.pth")
model.load_state_dict(torch.load("C:/rokey/6조_스터디/4주차/ex_model.pth"))
```

```
import torch
from Optimization import SimpleNN # 모델 정의 파일을 임포트

# 모델 정의
model = SimpleNN()

# 모델 가중치 로드
model.load_state_dict(torch.load("C:/rokey/6조_스터디/4주차/ex_model.pth"))
```

```
# 모델을 사용하여 예측 수행
model.eval() # 평가 모드로 설정
inputs = torch.rand(3)
outputs = model(inputs)
print(outputs)
```