PyTorch로 딥러닝 제대로 배우기

- 기초편 -

Part6. 모델

강사: 김 동 희

목차

I. Review of Neural Network Model

1. 오차 산출 단계

정답을 알고있는
입력 이미지
여러장

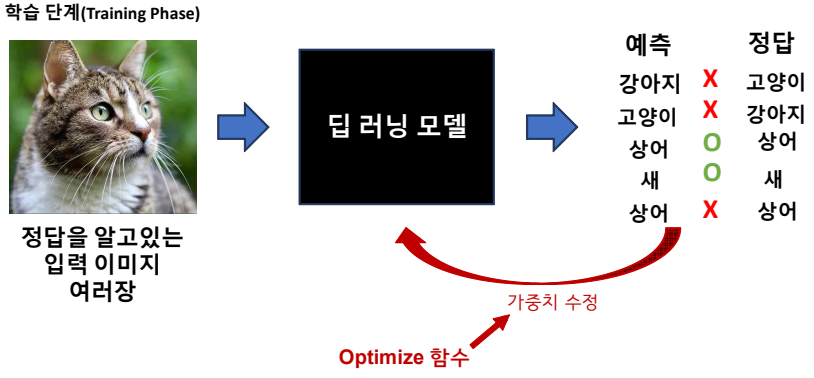
딥 러닝 모델

예측		정답
강아지	X	고양이
고양이	X	강아지
상어	O	상어
새	O	새
상어	X	상어

각 이미지에 대한
예측 값과 정답 비교

Loss 함수

2. 가중치 수정 단계(Optimization)



3. 평가 단계

학습 단계(Training Phase)

정답을 알고있는
입력 이미지
여러장

일부 학습된
딥 러닝 모델
(epoch: 1)

예측		정답
고양이	O	고양이
강아지	O	강아지
상어	O	상어
새	O	새
상어	X	고양이

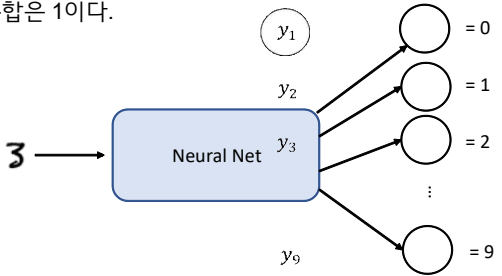
각 이미지에 대한
예측 값과 정답 비교

4. 출력층

❑ **Softmax Layer**

$$y_k = \frac{\exp(a_k)}{\sum_{i=1}^{n} \exp(a_i)}$$

- 0~1사이의 숫자를 가진다
- 소프트맥스 함수의 총합은 1이다.



5. Parametric vs. Non-parametric

❑ **Parametric Layers**

- Backpropagation을 통해 학습을 진행하는 계층
- 은닉층, 출력층, 합성곱 층 등
- 모델의 크기(메모리)를 좌우함
- 모델이 클 수록 학습 시간이 오래 걸리고, 더 많은 메모리를 필요로하며 더 좋은 GPU 성능을 요구함

❑ **Non-parametric Layers**

- 학습을 진행하지 않고, 단순한 계산이나 정보 추출을 진행하는 계층
- 활성화 계층, 입력층, Pooling 계층
- 모델의 크기에 변화가 없거나 오히려 줄여주는 역할을 수행
- 신호 전달을 주목적으로 함

II. 나만의 모델 생성하기

1. 모델 생성 Process

❑ Overview

Layer 정의

- 모델에서 어떤 layer를 사용할 것인지 정의
- 각 layer는 parameter를 추적함
- non-parametric layer는 사전에 정의하지 않은 경우도 있음

Forward 정의

- 텐서의 흐름에 따라 실제 layer를 배치
- non-parametric layer는 여기서 호출하는 경우도 있음
- 모델을 call 할 경우, forward 함수가 자동으로 호출됨
- forward 함수를 직접호출 하지 않음
- 설계상 Background operation이 존재

1. 모델 생성 Process

❑ Model building Example

```
class NeuralNetwork(nn.Module):
    def __init__(self):
        super(NeuralNetwork, self).__init__()
        self.flatten = nn.Flatten()
        self.linear_relu_stack = nn.Sequential(
            nn.Linear(28*28, 512),
            nn.ReLU(),
            nn.Linear(512, 512),
            nn.ReLU(),
            nn.Linear(512, 10),
        )

    def forward(self, x):
        x = self.flatten(x)
        logits = self.linear_relu_stack(x)
        return logits
```

2. Model Layers

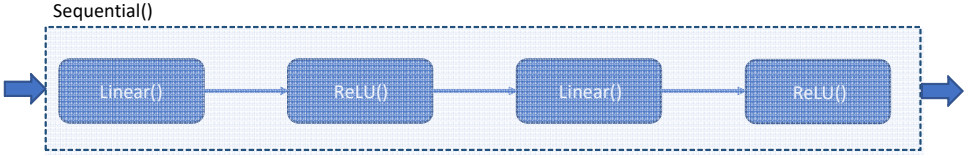
❑ Layer

- PyTorch에서 layer는 **모듈(nn.Module)**로 구현
- Parametric & Non-parametric 구분하지 않고 모두 모듈 클래스로 관리
- 하나의 모듈은 다양한 모듈을 포함 할 수 있음

3. 모델 구성 방법

❑ Sequential()

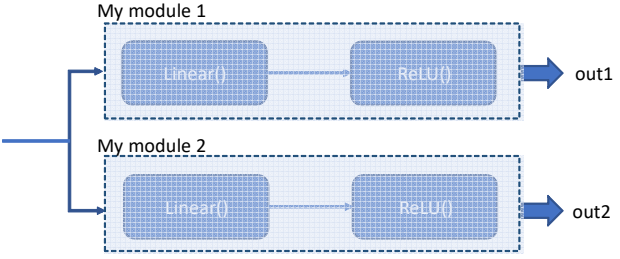
- 텐서를 순차적으로 flow 하는 컨테이너
- Layer를 직접 순차적으로 추가하거나, OrderedDict의 모듈을 통해 선언 가능
- Sequential의 forward() 메서드는 입력 텐서를 흘려 보낸 후, 각 layer를 순차적으로 실행된 결과를 반환
- Sequential은 전체 layer의 조합을 하나의 모듈(모델)로 고려
- Sequential vs. torch.nn.ModuleList: ModuleList는 말 그대로 모듈의 list, sequential은 연결이 되어있음



3. 모델 구성 방법

❑ Model in Model

- Customize하게 만든 모델(모듈)을 순서와 상관없이 연계 가능
- Sequential의 경우, 입력 값이 순차적으로 bypass 하는 특징이 있음
- 이렇게 모델을 구성하는 경우, 다양한 형태로 layer 조작 가능



III. 모델 다루기

1. Get Parameters

❑ **parameters()**

- Parameter는 모듈과 함께 사용될 때 매우 특별한 속성을 가진 텐서 하위 클래스
- 모듈에 할당되면 **parameter 목록에 자동으로 추가**.

```
parms = model.parameters()

for parm in parms:
  print(parm.shape)
```

```
torch.Size([512, 784])
torch.Size([512])
torch.Size([512, 512])
torch.Size([512])
torch.Size([10, 512])
torch.Size([10])
```

1. Get Parameters

❑ **named_parameters()**

- Parameter 목록에 등록된 parameter를 이름 정보와 함께 호출

```
named_parms = model.named_parameters()

for name, param in named_parms:
  print("{}: {}".format(name, param.shape))
```

```
linear_relu_stack.0.weight: torch.Size([512, 784])
linear_relu_stack.0.bias: torch.Size([512])
linear_relu_stack.2.weight: torch.Size([512, 512])
linear_relu_stack.2.bias: torch.Size([512])
linear_relu_stack.4.weight: torch.Size([10, 512])
linear_relu_stack.4.bias: torch.Size([10])
```

1. Get Parameters

❑ state_dict()

- 기본 기능은 named_parameter()과 동일.
- 단, 자료형이 다르고 parameters(), named_parameters() 함수와 다르게 tensor의 연결이 안되어있음
- State_dict 객체는 파이썬 dictionary 형이기 때문에 쉽게 저장, 업데이트, 변경 및 복원 가능
- Optimizer, 학습 가능한 parameter, 등록된 buffer가 있는 레이어에 state_dict 존재
- 특히, optimizer에는 옵티마이저의 상태와 사용된 하이퍼파라미터에 대한 정보가 포함

```
state_dict = model.state_dict()

for name, param in state_dict.items():
  print("{}: {}".format(name, param.shape))

linear_relu_stack.0.weight: torch.Size([512, 784])
linear_relu_stack.0.bias: torch.Size([512])
linear_relu_stack.2.weight: torch.Size([512, 512])
linear_relu_stack.2.bias: torch.Size([512])
linear_relu_stack.4.weight: torch.Size([10, 512])
linear_relu_stack.4.bias: torch.Size([10])
```

2. Save & Loading Model Weights

❑ **Save the model**

```
# 1. 현재 모델의 parameter 저장 (추후 확인용)
old_model = model.state_dict()

# 2. 모델 저장
torch.save(model.state_dict(), 'model.pth')
```

❑ **Loading the Model**

```
# 5. Load the model
model.load_state_dict(torch.load('model.pth'))
new_model = model.state_dict()
```

2. Save & Loading Model Weights

❑ **Save the optimizer and buffers**

```
from torch.optim import SGD

optimizer = SGD(model.parameters(), lr=0.01, momentum=0.9)
optim_state_dict = optimizer.state_dict()


torch.save({
    'model': model.state_dict(),
    'optim': optimizer.state_dict()
}, 'model.pth')
```

3. Train & Evaluation Mode

❑ **Train mode**

```
def train(dataloader, model, loss_fn, optimizer):
    size = len(dataloader.dataset)
    model.train()
    print("Training mode: {}".format(model.training))
    for batch, (X, y) in enumerate(dataloader):
        X, y = X.to(device), y.to(device)

        # Compute prediction error
        pred = model(X)
        loss = loss_fn(pred, y)

        # Backpropagation
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

        if batch % 100 == 0:
            loss, current = loss.item(), batch * len(X)
            print(f"loss: {loss:>7f}  [{current:>5d}/{size:>5d}]")
```

감사합니다.