DEEP LEARNING
WITH PYTORCH

ABOUT MODULE

ABOUT TORCH.NN

◆ torch.nn

❖ 인공신경망 관련 모든 기능들이 서브 모듈로 제공되는 서브 패키지

CONTAIN Modules

- Containers

AF & LOSS Modules

- Non-linear Activations (weighted sum, nonlinearity)
- Non-linear Activations (other)
- Distance Functions
- Loss Functions

LAYER Modules

- Convolution Layers
- Pooling layers
- Padding Layers
- Normalization Layers
- Recurrent Layers
- Transformer Layers
- Linear Layers
- Dropout Layers
- Sparse Layers
- Vision Layers
- Shuffle Layers
- DataParallel Layers (multi-GPU, distributed)

ABOUT TORCH.NN

◆ torch.nn.Module

❖ PyTorch의 모든 Neural Network의 Base Class 즉, Supber Class

❖ 다른 모듈을 포함할 수 있고, 트리 구조로 형성할 수 있음

❖ 입력 텐서 받고 출력 텐서 계산

❖ 학습 가능 매개변수 갖는 텐서들 내부 상태(internal state)를 가짐

【 필수 오버라이딩 메서드 】

- def __init__(self) : 모델 인스턴스 생성 메서드
- def forward(self) : 전방향 학습 진행 메서드

ABOUT TORCH.NN

◆ torch.nn.Module

❖ **def __init__(self) 콜백 메서드**

- 모델 층 구성 설계

```
import torch.nn as nn


class Model(nn.Module):
    def __init__(self):
        super().__init__()
        self.conv1 = nn.Conv2d(1, 20, 5)
        self.conv2 = nn.Conv2d(20, 20, 5)
```

ABOUT TORCH.NN

◆ torch.nn.Module

❖ **def forward(self) 콜백 메서드**

- 모델이 학습데이터를 입력받아서 forward 연산 진행시키는 함수
- model 객체를 데이터와 함께 호출하면 자동으로 실행
- forward propagation 정의하는 부분

```
import torch.nn.functional as F

def forward(self, x):
        x = F.relu(self.conv1(x))
        return F.relu(self.conv2(x))
```

ABOUT TORCH.NN

◆ torch.nn.Linear

❖ **선형 전결합층(Full-Connected Layer) 모듈**

❖ 선형 회귀 기능이 구현된 클래스

```
import torch.nn as nn

nn.Linear( in_features,          # 입력 특성 수
           out_features,         # 출력 특성 수
           bias=True,
           device=None,
           dtype=None)
```

ABOUT TORCH.NN

◆ torch.nn.Linear

❖ 사용예시

```
import torch.nn as nn

# 모델 인스턴스 생성
linear_model = nn.Linear(3, 1)


# 모델에 입력 데이터 전달 ➔ 전방향 학습 진행
output=linear_model(torch.tensor([1,2,3], dtype=torch.float))


# 결과 확인
print( output )
```

ABOUT TORCH.NN

◆ torch.nn.Linear

❖ 사용예시

```
# 가중치 확인
print( linear_model.weight, linear_model.weight.shape )

# 바이어스/절편 환인
print( linear_model.bias, linear_model.bias.shape )
```

ABOUT TORCH.NN

◆ torch.nn.Sequential

❖ **순서를 갖는 모듈 컨테이너**

❖ 정의된 **순서로 모든 모듈들을 통해 데이터 전달**

```
import torch.nn as nn


seq_modules = nn.Sequential( nn.Linear(10, 20),
                             nn.ReLU(),
                             nn.Linear(20, 10) )
```

ABOUT TORCH.NN

◆ torch.nn.Flatten

❖ 지정된 차원으로 데이터 변환하는 Layer

```
import torch.nn as nn
seq_modules = nn.Flatten( start_dim=1,  end_dim=1 )
```

- 기본값 : 1D
- start_dim, end_dim : 편편화 차원 지정 가능

ABOUT TORCH.NN

◆ 디바이스 설정

❖ GPU, MPS 같은 하드웨어 가속기 가능 여부에 따른 선택

- MPS (Multi-Process Service): 다수 프로세스가 동시에 단일 GPU에서 실행 시켜주는 런타임 서비스

```
import torch

device = ( "cuda" if torch.cuda.is_available()
           else "mps" if  torch.backends.mps.is_available()
           else "cpu" )

print(f"Using {device} device")
```

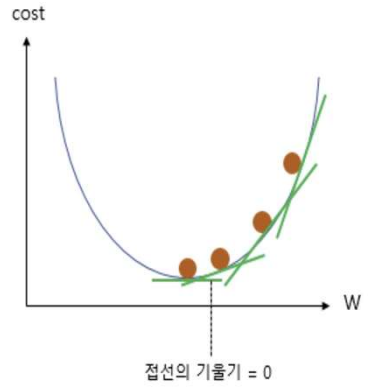
ABOUT AUTOGRAD

ABOUT AUTOGRAD

◆ 가중치/절편 최적화

❖ 경사하강법

- 비용/손실 함수를 미분하여 이 함수의 기울기(gradient)를 구해서 비용이 최소화 되는 방향을 찾아내는 알고리즘



ABOUT AUTOGRAD

◆ 가중치/절편 최적화

❖ 다양한 경사하강법

ABOUT AUTOGRAD

◆ torch.autograd

❖ **자동미분 기능**

- 모델 복잡해질수록 경사 하강법을 넘파이 등으로 직접 코딩하는 것은 까다로움
- 파이토치에서는 이런 수고를 하지 않도록 **자동 미분(Autograd) 지원**

- Tensor의 gradient 조건 ➔ 매개변수 설정 : **requires_grad = True** / ouput : scalar
- Tensor의 gradient 방법 ➔ **tensor.backward() 호출**
- gradient 확인 ➔ **requires_grad=True 설정된 Tensor.grad**

ABOUT AUTOGRAD

◆ torch.autograd

❖ 자동미분 설정 ➔ requires_grad = True 설정된 Tensor 업데이트 진행됨

```
# 역전파 진행으로, 가중치/절편 업데이트
loss.backward()


for name, param in linear_model.named_parameters():
        print(name, param)
```

ABOUT AUTOGRAD

◆ torch.autograd

❖ 자동미분 해제

```
with torch.no_grad():
      z = torch.matmul(x, w)+b
print(z.requires_grad)
```

```
z = torch.matmul(x, w)+b
z_det = z.detach()
print(z_det.requires_grad)
```

ABOUT AUTOGRAD

◆ 손실함수

❖ torch.nn.functional

- `l1_loss`
- `mse_loss`
- `margin_ranking_loss`
- `multilabel_margin_loss`
- `multilabel_soft_margin_loss`

❖ torch.nn.Loss Class

- `nn.L1Loss`
- `nn.MSELoss`
- `nn.CrossEntropyLoss`
- `nn.CTCLoss`
- `nn.NLLLoss`
- `nn.PoissonNLLLoss`

ABOUT AUTOGRAD

◆ 가중치/절편 최적화

❖ torch.optim 클래스

```
# 가중치/절편 최적화 방법 설정
from torch.optim import Adam

optimizer = Adam(linear_model.parameters(), lr=LR)
```

Adadelta

Adagrad

Adam

AdamW

SparseAdam

ABOUT AUTOGRAD

◆ 가중치/절편 최적화

❖ torch.optim 클래스

```
        # 가중치 기울기 0 초기화
        optimizer.zero_grad()

        # 학습 진행
        pre_y = linear_model(x)

        # 손실 계산
        loss = nn.MSELoss()(pre_y, y.reshape(-1,1))

        # 역전파 진행
        loss.backward()

        # 가중치/절편 업데이트
        optimizer.step()
```

TRANNING

TRANNING

◆ 학습 과정

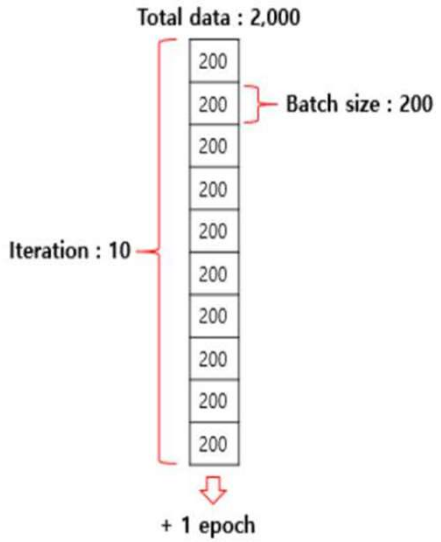
TRANNING

◆ 학습

❖ 학습 데이터셋 분리

- **에포크(epochs)** : 처음부터 끝까지 학습하는 횟수
- **배치크기(batch size)** : 전체 데이터를 작은 단위로 나눈 크기
  2의 제곱수 크기
- **이터레이션(iteration)** : 에포크, 배치크기로 계산한 반복 횟수
  W,b 업데이터 횟수

- 예) 100개 데이터, 배치크기 20개, 에포크 10번



TRANNING

◆ 학습

❖ 학습용 데이터 준비

```
# DataFrame ===> Feature 추출
X = bostonDF.iloc[:, :13].values
Y = bostonDF['medv'].values

print(f'X : {type(X)}, {X.shape}\nY : {type(Y)}, {Y.shape}')
```

TRANNING

◆ 학습

❖ 학습 설정

```
# 학습 횟수 및 한번에 학습할 데이터 크기 설정
EPOCHS = 500
BATCH_SIZE = 100
LR = 0.001
```

TRANNING

◆ 학습

❖ 학습 설정

```
# 모델 인스턴스 생성
linear_model = nn.Sequential(nn.Linear(13, 10),
                             nn.ReLU(),
                             nn.Linear(10, 1))
```

TRANNING

◆ 학습

❖ 학습 설정

```
# 가중치/절편 최적화 방법 설정
from torch.optim import Adam

optimizer = Adam(linear_model.parameters(), lr=LR)
```

TRANNING

◆ 학습

❖ 학습 진행

```
for epoch in range(EPOCHS):
    for i in range(len(X)//BATCH_SIZE):
        start = i*BATCH_SIZE
        end = start + BATCH_SIZE

        # ndarray ==> tensor변환
        x = torch.FloatTensor(X[start:end])
        y = torch.FloatTensor(Y[start:end])
```

TRANNING

◆ 학습

❖ 학습 진행

```
# 가중치 기울기 0 초기화
optimizer.zero_grad()

# 학습 진행
pre_y = linear_model(x)

# 손실 계산
loss = nn.MSELoss()(pre_y, y.reshape(-1,1))

# 역전파 진행
loss.backward()

# 가중치/절편 업데이트
optimizer.step()
```

TRANNING

◆ 학습

❖ 모델 평가

```
# 모델 성능 평가
pre = linear_model(torch.FloatTensor(X[0, :13]))

pre, Y[0]
```