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) 클래스

- 입력데이터에 xA^T+b 연산 결과를 출력하는 클래스
- TensorFloat32 지원

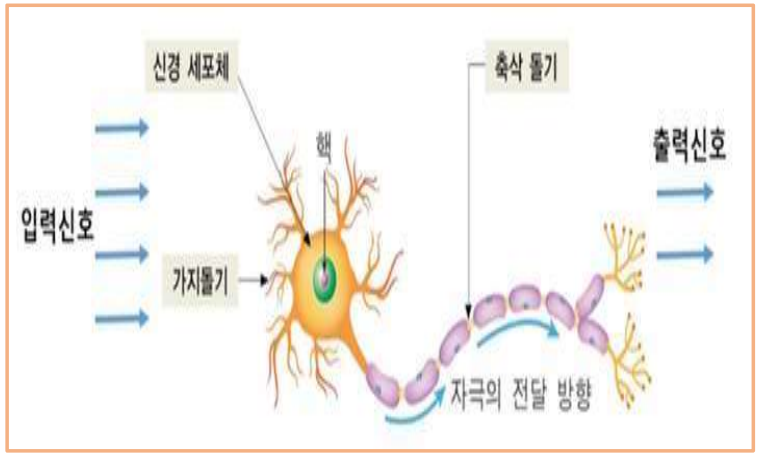
```
import torch.nn as nn

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

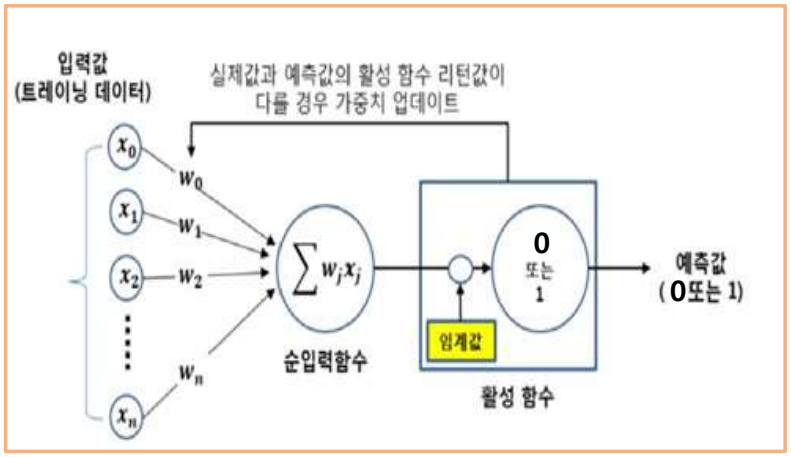
ABOUT TORCH.NN

◆ torch.nn.Linear

❖ 선형 전결합층(Full-Connected Layer) 클래스

- 퍼셉트론(Perceptron)/뉴런의 동작 중 피쳐와 가중치 곱의 합계 연산 구현



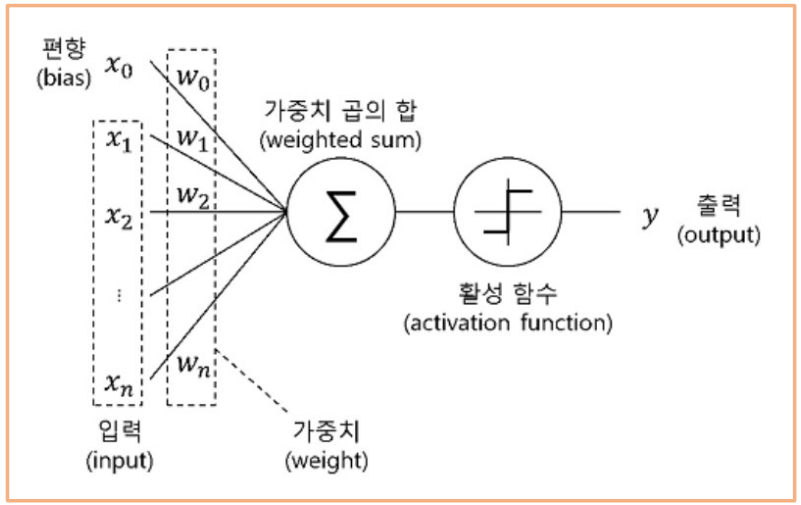


ABOUT TORCH.NN

◆ torch.nn.Linear

❖ 선형 전결합층(Full-Connected Layer) 클래스

- 퍼셉트론(Perceptron)/뉴런의 동작 중 피쳐와 가중치 곱의 합계 연산 구현



$$\sum_{i=0}^{N-1} w_i x_i = \boldsymbol{w} \cdot \boldsymbol{x} = \boldsymbol{w}^T \boldsymbol{x}$$

$$y = a\left(\sum_{i=0}^{N-1} w_i x_i + b\right)$$

활성함수 두 벡터의 내적 편향(Bias)

$$y = a(\boldsymbol{w}^T \boldsymbol{x} + b)$$

ABOUT TORCH.NN

◆ torch.nn.Linear

❖ 사용 예시

```
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

❖ 사용 예시

```
# 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

❖ **여러 개의 순서를 갖는 모듈을 담는 컨테이너 클래스**

- 정의된 **순서로 모든 모듈들을 통해 데이터 전달**

```
import torch.nn as nn

nn.Sequential( *Module )                    # nn.Module 자식 클래스들
nn.Sequential( OrderedDict[str, Module] )   # 순서있는 딕셔너리
```

ABOUT TORCH.NN

◆ torch.nn.Sequential

❖ 사용 예시

```
import torch.nn as nn

model = nn.Sequential(
                        nn.Linear(20,5),            # 입력층
                        nn.ReLU(),                  # AF
                        nn.Linear(5,3),             # 은력층
                        nn.ReLU(),                  # AF
                        nn.Linear(3,1)              # 출력층
                        nn.Sigmoid()                # AF
        )
```

ABOUT TORCH.NN

◆ torch.nn.Sequential

❖ 사용 예시

```
import torch.nn as nn
from collections import OrderedDict

model = nn.Sequential(OrderedDict([  ('conv1', nn.Conv2d(1,20,5)),
                                     ('relu1', nn.ReLU()),
                                     ('conv2', nn.Conv2d(20,64,5)),
                                     ('relu2', nn.ReLU())
                                   ]))
```

ABOUT TORCH.NN

◆ torch.nn.ModuleList

❖ **동적으로 모듈들을 추가하거나 삭제할 수 있는 모듈 컨테이너**

- **nn.Module의 list**를 input으로 받음
- nn.Sequential과 다르게 **forward() method가 없음**
- 안에 담긴 module 간 connection 없음
- **일반 List에 담을 시 Pytorch에서 인식 불가!**

```
import torch.nn as nn

nn.ModuleList( modules )                    # 반복이 가능한 객체 타입
```

ABOUT TORCH.NN

◆ torch.nn.ModuleList

```
# 은닉층 개수 동적인 모델 ----------------------------------------------------------------
class MyModel(nn.Module):

    def __init__(self, in_in, out_out, h_in, h_cnt):
        # 부모클래스 생성
        super().__init__()

        # 자식클래스의 인스턴스 속성 설정
        self.input_layer= nn.Linear(in_in, h_in)
        self.h1_layer=nn.ModuleList( [ nn.Linear(h_in, h_in) for _ in range(h_cnt) ] )
        self.output_layer=nn.Linear(h_in, out_out)
```

ABOUT TORCH.NN

◆ torch.nn.ModuleList

```
# 은닉층 개수 및 뉴런 개수 동적인 모델 ----------------------------------------------------
class MyModel(nn.Module):

    def __init__(self, in_in, in_out, out_out, h_ins=[], h_outs=[]):
        # 부모클래스 생성
        super().__init__()

        # 입력층 생성
        self.input_layer= nn.Linear( in_in,  h_ins[0] if len(h_ins) else in_out )
```

ABOUT TORCH.NN

◆ torch.nn.ModuleList

```
# 은닉층 여러개 생성
self.h1_layer=nn.ModuleList()
for idx in range(len(h_ins)):
    self.h1_layer.append( nn.Linear(h_ins[idx], h_outs[idx]) )

# 출력층 생성
self.output_layer=nn.Linear( h_outs[-1] if len(h_outs) else in_out,  out_out )
```

ABOUT TORCH.NN

◆ torch.nn.ModuleList

```
# 순전파 학습 메서드 ----------------------------------------------
def forward(self, x):

    y=F.relu(self.input_layer(x))

    for linear in self.h1_layer:
        y=F.relu(linear(y))

    return self.output_layer(y)
```

ABOUT TORCH.NN

◆ torch.nn.Flatten

❖ **지정된 차원으로 데이터 변환하는 Layer**

- 연속된 범위를 가진 차원을 하나의 텐서로 편편화

```
import torch.nn as nn

nn.Flatten( start_dim=1,          # 편편화 시작 차원 번호
            end_dim=1 )           # 편편화 끝 차원 번호
```

ABOUT TORCH.NN

◆ torch.nn.Flatten

❖ 사용 예시

```
# 모듈 로딩
import torch
import torch.nn as nn


# 입력 데이터 생성
t = torch.tensor([  [[1, 2],[3, 4]],
                    [[5, 6],[7, 8]]] )

print(f'형  태 : {t.shape}')
print(f'차  원 : {t.ndim}')
print(f'타  입 : {t.dtype}')
```

ABOUT TORCH.NN

◆ torch.nn.Flatten

❖ 사용 예시

```
# 기본값 start_dim=1, end_dim= -1
# 입력 데이터의 shape ( d, h, w )  ← ( 0, 1, 2)
f=nn.Flatten()

output = f(t)

print( type(f) )
print( output.shape )
print( output )
```

형 태 : torch.Size([2, 2, 2])
차 원 : 3
타 입 : torch.int64

torch.Size([2, 4])
tensor([[1, 2, 3, 4],
[5, 6, 7, 8]])

ABOUT TORCH.NN

◆ torch.nn.Flatten

❖ 사용 예시

```
# 변환 시작 차원 설정 start_dim=0, end_dim= -1
# 입력 데이터의 shape ( d, h, w )  ← ( 0, 1, 2)
f=nn.Flatten(0)

output = f(t)

print( type(f) )
print( output.shape )
print( output )
```

형 태 : torch.Size([2, 2, 2])
차 원 : 3
타 입 : torch.int64

torch.Size([8])
tensor([1, 2, 3, 4, 5, 6, 7, 8])

ABOUT TORCH.NN

◆ torch.nn.Flatten

❖ 사용 예시

```
# 변환 시작 차원 설정 start_dim=2, end_dim= -1
# 입력 데이터의 shape ( d, h, w )  ← ( 0, 1, 2)
f=nn.Flatten(2)

output = f(t)

print( type(f) )
print( output.shape )
print( output )
```

형 태 : torch.Size([2, 2, 2])
차 원 : 3
타 입 : torch.int64

torch.Size([2, 2, 2])
tensor([[[1, 2], [3, 4]],
[[5, 6], [7, 8]]])

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")
```