Pytorch 기초

목차

- Pytorch란?
- 텐서 조작
- torch.nn
- torch.optim
- torch.utils.data
- 모델 저장하기 & 불러오기

Pytorch란?

- Facebook이 개발한 오픈소스 머신 러닝 라이브러리

https://github.com/pytorch/pytorch

Pytorch란?

- Pytorch는 Numpy와 유사한 연산과 함수를 지원
- Pythonic하기 때문에 기존 Python처럼 학습 코드 작성 가능
- Debugging과 Customizing이 용이

Pytorch vs Tensorflow vs Keras

- Tensorflow와 Keras는 Google에서 개발한 오픈소스 딥러닝 라이브러리

- Tensorflow는 Pytorch처럼 독자적(stand-alone)으로 작동가능
- Keras는 tensorflow를 backend로 사용하는 high-level api를 지원, tensorflow나 pytorch와는 다르게 독자적으로는 사용 불가능

- Tensorflow 2.0이후로는 주로 keras로 사용

Pytorch vs tf.Keras

```pytorch
class Model(nn.Module):
  def __init__(self):
    self.layer1=nn.Linear(10,5)
    self.layer2=nn.Linear(5,1)
    self.ReLU=nn.ReLU()

  def forward(self,x):
    x=self.layer1(x)
    x=self.ReLU(x)
    x=self.layer2(x)
    x=self.ReLU(x)
    return x

model=Model()
criterion=nn.MSELoss()
optimizer=optim.SGD(model.parameters(),lr=0.001)

# training loop
for epoch in range(num_epochs):
  for data,label in train_loader:
    optimizer.zero_grad()
    pred=model(data)
    loss=criterion(pred,label)
    loss.backward()
    optimizer.step()
```

Pytorch

```tf.keras
model=tf.keras.models.Sequential([
  tf.keras.layers.Dense(5,activation='relu'),
  tf.keras.layers.Dense(1,activation='relu')
])

model.compile(optimizer='sgd',loss='mse',metrics=['acc'])
model.fit(x_train,y_train,epochs=num_epochs)
```

tf.Keras

AIoT

텐서 조작(Tensor Manipulation)

- Pytorch에서는 기본적으로 numpy와 유사한 방식으로 tensor를 다룸.

- numpy는 벡터나 행렬을 다루는 데 특화된 라이브러리.

텐서 조작

- numpy에서의 1차원 배열(벡터) 선언

```
import numpy as np

a=np.array([1,2,3,4,5,6,7,8])

print(a) # [1 2 3 4 5 6 7 8]
```

텐서 조작

- numpy에서의 2차원 배열(행렬) 선언

```
import numpy as np

b=np.array([[1,2,3,4],[5,6,7,8]])

print(b)
'''
[[1 2 3 4]
 [5 6 7 8]]
'''
```

텐서 조작

- numpy에서의 3차원 배열(텐서) 선언

```
import numpy as np

c=np.array([[[1,2],[3,4]],[[5,6],[7,8]]])

print(c)
'''
[[[1 2]
  [3 4]]

 [[5 6]
  [7 8]]]
'''
```

텐서 조작

- np.array()로 생성한 객체는 모두 numpy.ndarray의 객체

- 배열의 차원을 알아보고 싶으면 numpy.ndarray.ndim를,
  배열의 크기를 알아보고 싶으면 numpy.ndarray.shape를 사용

텐서 조작

```
print('a:',a)
print('a.ndim:',a.ndim)
print('a.shape:',a.shape)

print()

print('b:',b)
print('b.ndim:',b.ndim)
print('b.shape:',b.shape)

print()

print('c:',a)
print('c.ndim:',c.ndim)
print('c.shape:',c.shape)
```

출력:

```
a: [1 2 3 4 5 6 7 8]
a.ndim: 1
a.shape: (8,)

b: [[1 2 3 4]
 [5 6 7 8]]
b.ndim: 2
b.shape: (2, 4)

c: [[[1 2]
  [3 4]]

 [[5 6]
  [7 8]]]
c.ndim: 3
c.shape: (2, 2, 2)
```

텐서 조작

- Pytorch에서도 유사하게 배열 선언가능

- 배열의 원소의 타입에 따라
  torch.Tensor()
  torch.FloatTensor()
  torch.LongTensor()
  와 같이 선언가능

텐서 조작

- Pytorch에서의 1차원 텐서 선언

```
import torch

a=torch.Tensor([1,2,3,4,5,6,7,8])

print(a) # tensor([1., 2., 3., 4., 5., 6., 7., 8.]
```

텐서 조작

- Pytorch에서의 2차원 텐서 선언

```
import torch

b=torch.Tensor([[1,2,3,4],[5,6,7,8]])

print(b)

'''
tensor([[1., 2., 3., 4.],
        [5., 6., 7., 8.]])
'''
```

텐서 조작

- Pytorch에서의 3차원 텐서 선언

```
import torch

c=torch.Tensor([[[1,2],[3,4]],[[5,6],[7,8]]])

print(c)

'''
tensor([[[1., 2.],
         [3., 4.]],

        [[5., 6.],
         [7., 8.]]])
'''
```

텐서 조작

- Pytorch에서도 마찬가지로 ndim, shape를 이용하여 텐서의 차원과 텐서의 크기를 알 수 있다.

```
print('a:',a)
print('a.ndim:',a.ndim)
print('a.shape:',a.shape)

print()

print('b:',b)
print('b.ndim:',b.ndim)
print('b.shape:',b.shape)

print()

print('c:',a)
print('c.ndim:',c.ndim)
print('c.shape:',c.shape)
```

출력:

```
a: tensor([1., 2., 3., 4., 5., 6., 7., 8.])
a.ndim: 1
a.shape: torch.Size([8])

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

c: tensor([[[1., 2.],
         [3., 4.]],

        [[5., 6.],
         [7., 8.]]])
c.ndim: 3
c.shape: torch.Size([2, 2, 2])
```

텐서 조작

- 합, 차, element-wise 곱, 스칼라 배 등의 연산은 기본 연산자로 구현되어있다.

```
A=torch.Tensor([[1,2],[3,4]])
B=torch.Tensor([[3,4],[5,6]])

print('Add')
print(A+B)
print('Subtract')
print(A-B)
print('Scalar Multiplication')
print(3*A)
print('Element-wise Multiplication')
print(A*B)
print(A.mul(B))
print('Matrix multiplication')
print(A.matmul(B))
```

출력:

```
Add
tensor([[ 4.,  6.],
        [ 8., 10.]])
Subtract
tensor([[-2., -2.],
        [-2., -2.]])
Scalar Multiplication
tensor([[ 3.,  6.],
        [ 9., 12.]])
Element-wise Multiplication
tensor([[ 3.,  8.],
        [15., 24.]])
tensor([[ 3.,  8.],
        [15., 24.]])
Matrix multiplication
tensor([[13., 16.],
        [29., 36.]])
```

텐서 조작

- mean(): 평균을 계산
- sum(): 합을 계산
- min(), max(): 최솟값 또는 최댓값 계산하여 argmin 또는 argmax와 함께 튜플로 반환
- argmin(), argmax(): 최솟값 또는 최댓값의 인덱스 반환

텐서 조작

- mean()을 하게 되면 텐서의 모든 원소의 평균이 출력된다.
  인자로 dim을 지정해 줄 수 있는데,
  지정된 차원을 제외하고 평균을 구해 준다.

```
print(A)

print('\nMean')
print(A.mean())
print(A.mean(dim=0))
print(A.mean(dim=1))
```

출력:

```
tensor([[1., 2.],
        [3., 4.]])

Mean
tensor(2.5000)
tensor([2., 3.])
tensor([1.5000, 3.5000])
```

텐서 조작

- view(): 텐서의 shape를 원하는 대로 변경
- squeeze(): 텐서의 크기가 1인 차원을 모두 줄임.
- unsqueeze(): 텐서에 크기가 1인 차원을 추가

텐서 조작

- view(): 인자로 텐서의 shape를 넣어 주면 기존 텐서를 해당 사이즈로 변경한다. -1인 곳은 자동으로 계산된다.

```
print(A)

print(A.view(4,1))
print(A.view(4,1).shape)
```

출력:

```
tensor([[1., 2.],
        [3., 4.]])
tensor([[1.],
        [2.],
        [3.],
        [4.]])
torch.Size([4, 1])
```

```
print(A)

print(A.view(1,-1))
print(A.view(1,-1).shape)
```

출력:

```
tensor([[1., 2.],
        [3., 4.]])
tensor([[1., 2., 3., 4.]])
torch.Size([1, 4])
```

텐서 조작

- squeeze()

```
B=A.view(1,2,2,1)

print(B.shape)

print(B.squeeze())
print(B.squeeze().shape)
```

출력:

```
torch.Size([1, 2, 2, 1])
tensor([[[1., 2.],
         [3., 4.]]])
torch.Size([2, 2])
```

텐서 조작

- unsqueeze(): dim에 해당하는 위치에 크기가 1인 차원을 추가

```
print(A.unsqueeze(0).shape)
print(A.unsqueeze(1).shape)
print(A.unsqueeze(1).unsqueeze(0).shape)
```

출력:

```
torch.Size([1, 2, 2])
torch.Size([2, 1, 2])
torch.Size([1, 2, 1, 2])
```

torch.nn

- pytorch에서 모델을 구성하기 위한 다양한 함수(layer)들과 활성 함수 그리고 다양한 손실 함수들이 구현되어있는 라이브러리.

- 커스터마이징하기 위해서는 torch.nn.Module을 상속하는 클래스를 정의하고 __init__()과 forward()를 구현하여 오버라이딩된 형태로 사용

torch.nn – nn.Module

- $y = Wx + b$에 해당하는 모델 생성

```python
import torch.nn as nn

class MyModule(nn.Module):
  def __init__(self):
    super(MyModule,self).__init__()
    self.W=torch.Tensor([1])
    self.b=torch.Tensor([1])

  def forward(self,x):
    return self.W*x+self.b
```

```python
model=MyModule()

y=model(1)

print(y) # tensor([2.])
```

torch.nn – nn.Linear

- nn.Linear(in_features,out_features)
  다항 함수에 해당하는 layer.($y = W^T x + b$)
  입력이 2개이고 출력이 3개인 경우
  nn.Linear(2,3)으로 선언할 수 있다.

```
layer1=nn.Linear(2,3)

x=torch.Tensor(2)
y=layer1(x)

print('Input shape:',x.shape)
print('Output shape:',y.shape)
```

출력:

```
Input shape: torch.Size([2])
Output shape: torch.Size([3])
```

AIoT

torch.nn – 여러 활성 함수

- torch.nn에는 여러 활성 함수 또한 구현 되어있다.

- nn.sigmoid(): 시그모이드 함수
- nn.Tanh(): 하이퍼볼릭탄젠트 함수
- nn.ReLU(): ReLU(Rectified Linear Unit) 함수
- nn.Softmax(): 소프트맥스 함수

torch.nn – 여러 손실 함수

- 2강에서 다루었던 여러 손실 함수 또한 구현돼 있다.

- nn.L1Loss(): MAE(Mean Absoulte Error) 함수.
  회귀 문제에서 사용.
- nn.MSELoss(): MSE(Mean Squared Error) 함수.
  회귀 문제에서 사용.

torch.nn – 여러 손실 함수

- nn.BCELoss(): Binary Cross Entropy 함수.
  분류 문제에서 클래스가 두개인 경우에 사용.
- nn.CrossEntropyLoss(): Cross Entropy 함수.
  분류 문제에서 클래스가 두개 이상인 경우에 사용

torch.nn

- 보통 학습 모델을 설계할 때는 nn.Module을 상속하는 사용자 정의 클래스를 사용한다.
- print문을 활용하여 모델의 구성을 확인할 수 있다.

```
class LinearModel(nn.Module):
    def __init__(self,in_features,out_features):
        super(LinearModel,self).__init__()
        self.layer1=nn.Linear(in_features,out_features)

    def forward(self,x):
        x=self.layer1(x)
        return x

model=LinearModel(1,10)

print(model)
```

출력:

```
LinearModel(
  (layer1): Linear(in_features=1, out_features=10, bias=True)
)
```

AIoT

torch.nn

- 사용자 정의 클래스를 사용하는 이유는 대개 모델의 구성이 여러 층으로 이뤄져 있기 때문이다.
- 각 층이 일렬(sequential)로 설계되어 있는 경우에는 다음과 같이 코드를 작성할 수 있다.

```
class SequentialModel(nn.Module):
    def __init__(self,in_features,out_features):
        super(SequentialModel,self).__init__()
        self.layer1=nn.Linear(in_features,out_features)
        self.sigmoid=nn.Sigmoid()

    def forward(self,x):
        x=self.layer1(x)
        x=self.sigmoid(x)
        return x

model=SequentialModel(1,10)

print(model)
```

출력:

```
SequentialModel(
  (layer1): Linear(in_features=1, out_features=10, bias=True)
  (sigmoid): Sigmoid()
)
```

torch.nn – nn.Sequential

- 층의 개수가 늘어나 모델이 복잡해지면,
  이전과 같은 방법으로 모델을 구현하는 것은 귀찮을 수 있다.
- nn.Sequential()을 사용하면 인자로 들어간 층들을 순서대로 연산시키도록 할 수 있다.

```
class SequentialModel2(nn.Module):
    def __init__(self,in_features,out_features):
        super(SequentialModel2,self).__init__()
        self.layers=nn.Sequential(
            nn.Linear(in_features,out_features),
            nn.Sigmoid()
        )

    def forward(self,x):
        return self.layers(x)

model=SequentialModel2(1,10)

print(model)
```

출력:

```
SequentialModel2(
  (layers): Sequential(
    (0): Linear(in_features=1, out_features=10, bias=True)
    (1): Sigmoid()
  )
)
```

AIoT

torch.nn – nn.Squential

```python
train_x=torch.Tensor([[1],[2],[3],[4]])
train_y=torch.Tensor([[2],[3],[4],[5]])

model=LinearModel(1,1)
criterion=nn.MSELoss()

pred=model(train_x)
loss=criterion(pred,train_y)

print('input shape:',train_x.shape)
print('output shape:',pred.shape)

print('pred:',pred)
print('loss:',loss)
```

출력:

```
input shape: torch.Size([4, 1])
output shape: torch.Size([4, 1])
pred: tensor([[-0.5867],
        [-1.4433],
        [-2.2999],
        [-3.1565]], grad_fn=<AddmmBackward>)
loss: tensor(33.1632, grad_fn=<MseLossBackward>)
```

torch.optim

- Pytorch의 학습 과정에서 적용시킬 수 있는 여러 최적화 알고리즘이 구현된 라이브러리이다.

```
model=Model()
criterion=nn.MSELoss()
optimizer=optim.SGD(model.parameters(), lr=0.001)

# training loop
for epoch in range(num_epochs):
  for data,label in train_loader:
    optimizer.zero_grad()
    pred=model(data)
    loss=criterion(pred,label)
    loss.backward()
    optimizer.step()
```

torch.optim

- Pytorch에서의 최적화는 다음 과정을 통해 진행된다.

1. Optimizer 선언
2. Optimizer의 gradient 초기화
3. Loss로부터 gradient 계산
4. 계산된 gradient를 이용해 매개변수 업데이트
5. 2번부터 다시 반복

```
model=Model()
criterion=nn.MSELoss()
optimizer=optim.SGD(model.parameters(), lr=0.001) # 1

# training loop
for epoch in range(num_epochs):
  for data,label in train_loader:
    optimizer.zero_grad() # 2
    pred=model(data)
    loss=criterion(pred,label)
    loss.backward() # 3
    optimizer.step() # 4
```

torch.optim

- 2강에서 매개변수의 최적화는 주어진 손실 함수를 미분하여 진행된다고 언급한 바 있음.

- Pytorch에서는 torch.autograd라는 라이브러리가 손실 함수의 도함수로부터 gradient를 자동으로 계산해 주기 때문에 편리하게 매개변수를 업데이트 할 수 있음.

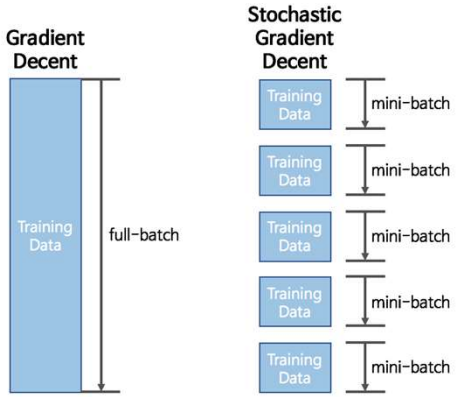
torch.optim

- torch.optim에는 여러 최적화 알고리즘이 구현되어 있지만 주로 사용하는 것은 다음 두가지 알고리즘.

- optim.SGD(): 2강에서 다루었던 스토캐스틱 경사 하강법
- optim.Adam(): SGD를 다소 개선한 Adam 알고리즘

torch.optim – GD vs SGD

- 경사 하강법(GD)은 데이터를 한번에 계산하여 최적화 진행
- 스토캐스틱 경사 하강법(SGD)은 데이터를 여러 개의 미니 배치(mini-batch)로 나누어 하나의 미니 배치에 대해 최적화 진행



AIoT

torch.optim – GD vs SGD

- 경사 하강법의 경우 모든 데이터를 다루기 때문에 정확한 방향으로 최적화가 이뤄지지만, 메모리와 시간이 많이 소요된다.

- 스토캐스틱 경사 하강법은 경사 하강법에 비해 진동하며 최적화가 이뤄지지만 메모리와 시간이 적게 소요.

torch.utils.data

- Pytorch에서의 데이터 관련 유틸리티가 제공되는 라이브러리
  주로 사용되는 것은 다음과 같다.

- Dataset: 데이터셋을 정의하는 클래스.
- random_split: 데이터셋을 분할하는 데 사용되는 함수.
  주로 학습 데이터와 검증 데이터를 분할하는 데 사용된다.
- DataLoader: 데이터셋을 학습 하기 편하게 불러오기 위한 클래스.
  배치 사이즈나 데이터를 불러오는 데 사용할 프로세스의 수 정의.

torch.utils.data - Dataset

- 데이터셋을 정의하는데 사용되는 클래스.
- 파일로로부터 데이터를 불러오고,
  변환이 필요하다면 변환하는 과정까지 포함한다.
- 보통 Dataset을 상속하여 클래스를 새로 설계하고
  __init__(), __getitem__(), __len__()을 구현하여 오버라이딩 한다.

torch.utils.data – random_split

- 데이터셋을 분할하는 함수.

- 원본 데이터셋과 데이터를 몇 개 씩 나눌지 리스트를 인자로 전달하면 분할된 데이터셋이 반환된다.

torch.utils.data – DataLoader

- 데이터셋을 학습 하기 편하게 불러오기 위한 클래스.
- batch_size에 미니 배치의 크기를 입력.
  보통 메모리 크기에 맞게 2^n 으로 사용.
- num_wokers에 사용할 프로세스의 수 입력.
- shuffle에는 데이터의 순서를 섞을 지 결정.
  주로 학습 데이터는 True로,
  검증이나 테스트 데이터는 False로 설정

torch.utils.data

- Dataset, random_split, DataLoader 예시

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

class MyDataset(Dataset):
  def __init__(self):
    super(MyDataset,self).__init__()
    self.data=[...]
    self.labels=[...]

  def __getitem__(self,idx):
    return data[idx],label[idx]

  def __len__(self):
    return len(self.data) # return len(self.labels)


all_data=MyDataset()

train_data_ratio=0.8
train_data_len=int(len(all_data)*train_data_ratio)
valid_data_len=len(all_data)-train_data_len

train_data,valid_data=random_split(all_data,[train_data_len,valid_data_len])

batch_size=64
train_loader=DataLoader(train_data,batch_size=batch_size,shuffle=True,num_workers=2)
valid_loader=DataLoader(valid_data,batch_size=batch_size,shuffle=False,num_workers=2)
```

모델 저장하기 & 불러오기

- Pytorch의 Module과 optimizer는 상태 정보를 갖고 있음.

- nn.Module.state_dict()

```
model=LinearModel(1,1)

print(model.state_dict())
```

출력:

```
OrderedDict([('layer1.weight', tensor([[0.7161]])), ('layer1.bias', tensor([0.6081]))])
```

AIoT

모델 저장하기 & 불러오기

- optim.SGD.state_dict()

```
optimizer=optim.SGD(model.parameters(),lr=0.0001)

print(optimizer.state_dict())
```

출력:

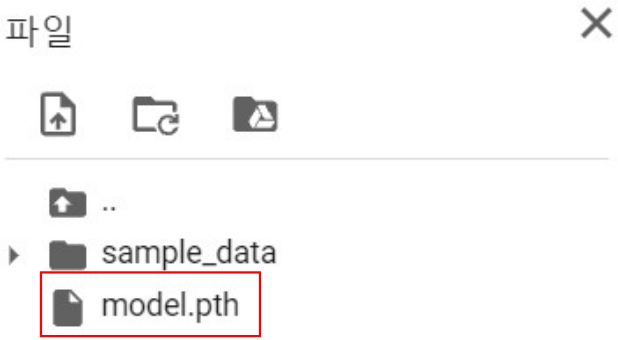
```
{'state': {}, 'param_groups': [{'lr': 0.0001, 'momentum': 0, 'dampening': 0, 'weight_decay': 0, 'nesterov': False, 'params': [0, 1]}]}
```

모델 저장하기 & 불러오기

- 학습 과정 중에 현재 학습중인 모델을 저장해야 하는 경우 torch.save(obj,f)를 이용

```
model_path='./model.pth'

torch.save(model.state_dict(),model_path)
```



AIoT

모델 저장하기 & 불러오기

- 저장한 모델을 불러오는건
  torch.load()와 nn.Module.load_state_dict()를 이용

```
model.load_state_dict(torch.load(model_path))
```

출력:

```
<All keys matched successfully>
```

Review

- PyTorch란?
  Facebook에서 개발한 딥러닝 라이브러리.
- 텐서 조작
  Numpy와 유사하게 작동.
- torch.nn
  모델을 구성하는 데 필요한 layer와 활성 함수, 손실 함수 등이 구현.
- torch.optim
  여러 최적화 알고리즘이 구현.
- torch.utils.data
  Dataset,DataLoader 등 데이터를 다루기 위한 기능 구현.
- 모델 저장하기 & 불러오기
  저장: state_dict(), torch.save()
  불러오기: load_state_dict(), torch.load()

4장 Preview

- 주어진 데이터(X)에 대해서 결과(Y)를 예측하는 Regression.

- 다음 두가지 데이터셋에 대해 실습 진행
  - Boston Housing Dataset: 보스턴 지역의 집값 예측
  - Diabetes Progression Dataset: 당뇨병 진행도 예측

References

[1] https://pytorch.org

[2] https://github.com/pytorch/pytorch

[3] https://championprogram.tistory.com/273

수고하셨습니다!