x

K-Digital Training 스마트 팩토리 3기

목차

- Pytorch 소개와 설치
- Tensor
- 자동 미분

PyTorch 기초

PyTorch

- https://pytorch.kr/
- PyTorch는 오픈 소스 딥러닝 프레임워크로, 계산 그래프를 빌드하고 확률적 기계학습을 수행할 수 있는 강력한 기능을 제공
- PyTorch는 파이썬 기반으로 작성된 라이브러리로, 연구 및 개발에 널리 활용되며 사용이 쉽고 유연한 구조를 제공
- 자동 미분(Autograd), 파이토치 라이트닝 제공

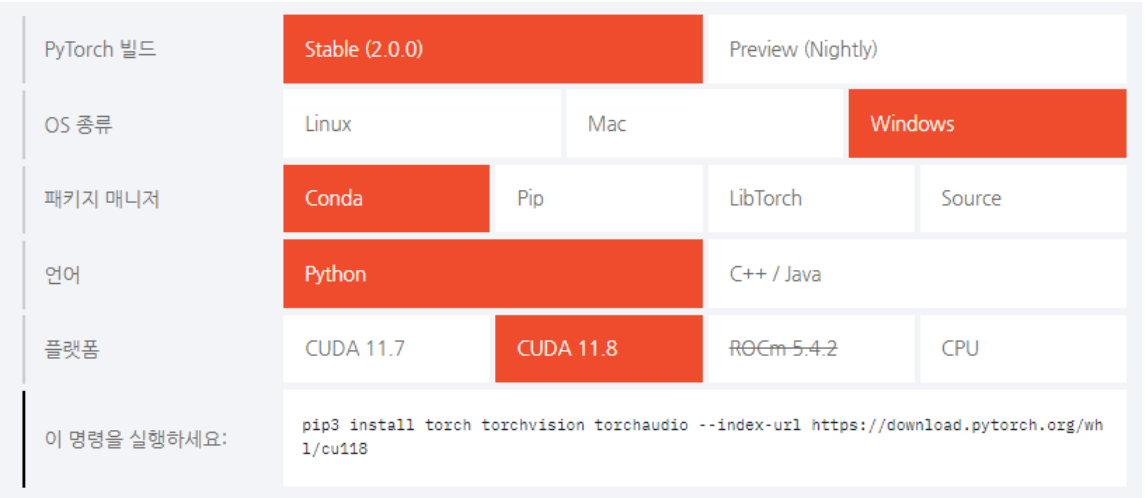
PyTorch

- vs TensorFlow

항목	PyTorch	TensorFlow
그래프 생성	동적 계산 그래프 (디버깅 및 코드 작성이 쉽고 직관적)	정적 계산 그래프와 동적 계산 그래프(Eager Execution, TensorFlow 2.0+)
사용 및 배포	연구 및 개발에 강점, 배포에 다소 어려움	대규모 분산 시스템 및 프로덕션 배포에 강점
플랫폼 지원	TorchScript 사용, ONNX에서 모바일 및 임베디드 플랫폼에 모델 내보냄	TensorFlow Lite 사용, 모바일 및 임베디드 플랫폼에 모델 내보냄
커뮤니티 및 생태계	대학 연구 및 초기 개발 지지, 빠르게 성장하고 확장 중인 생태계	Google 지원, Keras 등 고수준 API 사용 가능, 더 큰 사용자 커뮤니티

PyTorch

- 설치하기
  - https://pytorch.kr/ 내에서 환경에 맞는 설치 명령어 획득 가능
  - 강의에서는 Conda, cuda version 11.8 로 설치
  - Conda로 설치하기 위해서는 anaconda 설치(https://www.anaconda.com/)
  - Cuda 설치 (https://developer.nvidia.com/cuda-11-8-0-download-archive)
  - conda install pytorch torchvision torchaudio pytorch-cuda=11.8 -c pytorch -c nvidia



PyTorch

- Trouble Shooting
  - ‘conda activate’ 시 ‘cp949’ 오류
    - 계정명이 한글일 경우에 오류 발생
    - 영어로 된 계정명 사용(영어 이름 신규 계정 추가하여 진행)
      - 시장 〉 설정 〉 계정 〉 가족&다른 사용자 〉 다른 사용자 추가 〉 계정 추가
  - ‘conda activate’ 시 powershell 오류
    - Power shell 을 관리자 권한으로 실행
    - ‘Get-ExecutionPolicy’ 명령어 실행 시 Restrict가 나올 경우
    - ‘Set-ExecutionPolicy Unrestricted’ 실행 후 Y를 눌러서 권한 허용

PyTorch

- Trouble Shooting
  - conda가 default activate 되지 않게 하려면
  - ‘conda config --set auto_activate_base false’

PyTorch

- 설치 없이 하기
  - Google Colab 이용
  - https://colab.research.google.com/

Tensor

- 텐서는 배열이나 행렬과 매우 유사한 특수한 자료구조
- PyTorch에서는 모델의 입출력, 매개변수들을 부호화 한다.
- NumPy의 ndarray와 유사
- 자동 미분에 최적화되어있음

Tensor 생성

- 직접생성
  - data = [[1, 2],[3, 4]]

    x_data = torch.tensor(data)
- NumPy 배열로 부터 생성
  - np_array = np.array(data)

    x_np = torch.from_numpy(np_array)

Tensor 생성

- 무작위 또는 상수값 사용
  - shape = (2,3,) #shape 는 텐서의 차원을 나타내는 튜플
    rand_tensor = torch.rand(shape)
    ones_tensor = torch.ones(shape)
    zeros_tensor = torch.zeros(shape)
  - #rand : tensor([[0.2851, 0.4862, 0.0144], [0.4129, 0.0634, 0.8218]])
    #ones : tensor([[1., 1., 1.], [1., 1., 1.]])
    #zeros : tensor([[0., 0., 0.], [0., 0., 0.]])

Tensor 속성

tensor = torch.rand(3,4)

- 모양(shape)
  - tensor.shape -> torch.Size([3,4])
- 자료형(datatype)
  - tensor.dtype -> torch.float32
- 장치(device)
  - tensor.device -> cpu

Tensor 연산

- https://pytorch.org/docs/stable/torch.html
- 기본적으로 텐서는 CPU에 생성. GPU로 텐서를 명시적으로 이동 하는 법
  - if torch.cuda.is_available():
    tensor=tensor.to("cuda")
  - 보통은 아래와 같이 사용
    device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
    tensor.to(device)

Tensor 연산

- NymPy식의 표준 인덱싱과 슬라이싱

```
tensor = torch.randint(0, 3, (4, 4))
print(tensor)
print(f"First row: {tensor[0]}")
print(f"First column: {tensor[:, 0]}")
print(f"Last column: {tensor[..., -1]}")
tensor[:,1] = 0
print(tensor)
```

```
tensor([[1, 0, 2, 1],
        [1, 0, 0, 0],
        [1, 0, 1, 2],
        [0, 2, 1, 0]])
First row: tensor([1, 0, 2, 1])
First column: tensor([1, 1, 1, 0])
Last column: tensor([1, 0, 2, 0])
tensor([[1, 0, 2, 1],
        [1, 0, 0, 0],
        [1, 0, 1, 2],
        [0, 0, 1, 0]])
```

- 텐서 합치기

```
t1 = torch.cat([tensor, tensor, tensor], dim=1)
print(t1)
```

```
tensor([[1, 0, 2, 1, 1, 0, 2, 1, 1, 0, 2, 1],
        [1, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0],
        [1, 0, 1, 2, 1, 0, 1, 2, 1, 0, 1, 2],
        [0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 1, 0]])
```

Tensor 연산

- 산술연산

```
tensor = torch.ones(3, 2)*2
print(tensor)
```

```
tensor([[2., 2.],
        [2., 2.],
        [2., 2.]])
```

```
# 두 텐서 간의 행렬 곱(matrix multiplication)을 계산합니다. y1, y2, y3은 모두 같은 값을 갖습니다.
# ``tensor.T`` 는 텐서의 전치(transpose)를 반환합니다.
y1 = tensor @ tensor.T
y2 = tensor.matmul(tensor.T)

y3 = torch.rand_like(y1)
torch.matmul(tensor, tensor.T, out=y3)
```

```
tensor([[8., 8., 8.],
        [8., 8., 8.],
        [8., 8., 8.]])
```

Tensor 연산

- 산술연산

```
# 요소별 곱(element-wise product)을 계산합니다. z1, z2, z3는 모두 같은 값을 갖습니다.
z1 = tensor * tensor
z2 = tensor.mul(tensor)

z3 = torch.rand_like(tensor)
torch.mul(tensor, tensor, out=z3)
```

```
tensor([[4., 4.],
        [4., 4.],
        [4., 4.]])
```

Tensor 연산

- 단일-요소 텐서
  - 모든 값을 하나로 집계

```
agg = tensor.sum()
agg_item = agg.item()
print(agg_item, type(agg_item))
```

```
12.0 <class 'float'>
```

- 바꿔치기 연산
  - 연산 결과를 피연산자에 저장
  - _ 접미사를 갖습니다

```
print(f"{tensor} ₩n")
tensor.add_(5)
print(tensor)
```

```
tensor([[2., 2.],
        [2., 2.],
        [2., 2.]])

tensor([[7., 7.],
        [7., 7.],
        [7., 7.]])
```

Tensor 연산

- NumPy 변환
  - cpu상의 텐서와 NymPy 배열은 메모리 공간을 공유
  - 하나를 변경하면 다른 하나도 변경
- 텐서를 NumPy 배열로 변환하기

```
t = torch.ones(5)
print(f"t: {t}")
n = t.numpy()
print(f"n: {n}")
```

```
t: tensor([1., 1., 1., 1., 1.])
n: [1. 1. 1. 1. 1.]
```

```
t.add_(1)
print(f"t: {t}")
print(f"n: {n}")
```

```
t: tensor([2., 2., 2., 2., 2.])
n: [2. 2. 2. 2. 2.]
```

자동미분(Autograd)

- 역전파를 위해서는 gradient(기울기)를 구해야 합니다.
  - 즉, 미분을 해야함
- tensor가 미분을 하려면 다음 조건들이 필요합니다.
  - tensor의 옵션이 requires_grad = True 로 설정 (default 는 False)
  - backpropagation 을 시작할 지점의 output은 scalar 형태

자동미분(Autograd)

- gradient를 구하기 위해서는 backpropagation 을 시작할 지점의 tensor에서 .backward() 함수를 호출
- gradient 값을 확인하려면 .grad 값을 통해 확인 가능

자동미분(Autograd)

```
x = torch.ones(2, 2, requires_grad=True)
y1 = x + 2
print(y1)
```

```
tensor([[3., 3.],
        [3., 3.]], grad_fn=<AddBackward0>)
```

```
y2 = x - 2
print(y2)
```

```
tensor([[-1., -1.],
        [-1., -1.]], grad_fn=<SubBackward0>)
```

```
y3 = x * 2
print(y3)
```

```
tensor([[2., 2.],
        [2., 2.]], grad_fn=<MulBackward0>)
```

```
y4 = x / 2
print(y4)
```

```
tensor([[0.5000, 0.5000],
        [0.5000, 0.5000]], grad_fn=<DivBackward0>)
```

- 텐서에 사칙연산을 하게 되면 grad_fn에 각각 AddBackwrod0, SubBackward0, MulBackward0, DivBackward0과 같이 수행된 연산이 기록된다.

자동미분(Autograd)

- f(x) = 3x^2+4x+1
- 위의 식을 x에 대해서 미분한다고 가정
- f'(x) = 6x+4
- x=2 라고 하면 x=2에서의 기울기는 6*2+4 = 16

자동미분(Autograd)

- 이것을 코드로 보면

```
x = torch.tensor(2.0, requires_grad=True)
y = 3*x**2 + 4*x + 1
y.backward()
print(x.grad)
```

```
tensor(16.)
```

자동미분(Autograd)

- 편미분
  - $y = x^3 + z^2$ 라고 가정
  - x에 대해서 편미분하면 $3x^2$
  - z에 대해서 편미분하면 $2z$
  - $f'(x,z) = f'(1,2)$ 을 한다고 하면
    - $x = 1$ 일때 $y'(1) = 3$
    - $z = 2$ 일때 $y'(2) = 4$

자동미분(Autograd)

- 편미분
  - 위의 내용을 코드로 살펴보면

```
x = torch.tensor(1.0, requires_grad=True)
z = torch.tensor(2.0, requires_grad=True)
y = x**3 + z**2
y.backward()
print(x.grad, z.grad)
```

```
tensor(3.) tensor(4.)
```

자동미분(Autograd)

- no_grad()
  - 학습이 모두 끝나고 테스트를 할 때는 역전파를 하지 않아도 되므로 gradient를 구하고 업데이트할 필요가 없다.
  - 이때 사용하는 것이 torch.no_grad()

자동미분(Autograd)

- no_grad()

```
x = torch.tensor(1.0, requires_grad = True)
print(x.requires_grad)
print((x**2).requires_grad)
# True
# True

with torch.no_grad():
    print(x.requires_grad)
    print((x**2).requires_grad)
# True
# False

print(x.requires_grad)
print((x**2).requires_grad)
# True
# True
```