part 01. 파이토치 기초

문서 유형	
상태	
이해관계자	
작성일시	@2021년 2월 10일 오후 5:53
작성자	㊊ 민경 김
최종 편집일시	@2021년 2월 11일 오후 5:31
최종 편집자	㊊ 민경 김

▼ **목차**

1. 파이썬 또는 아나콘다 설치하기

1.4 가상 환경 설정하기

- **가상 환경**이란, '독립된 작업 공간'을 의미하며 파이썬을 설치했을 때의 초기 단계와 동일한 설정으로 독립된 작업 공간을 생성한다.
- 가상 환경은 분석가가 각 모듈별로 다른 버전을 이용해야할 때 유용하게 이용할 수 있다.
- 예를 들어, 딥러닝 알고리즘을 쉽게 설계할 수 있는 프레임워크 중에는 TensorFlow가 있다. TF는 최근 2.0 버전으로 업데이트되면서 기존 1.0 버전에서 설계하던 방식을 과감히 변경했다.
- 또한, 본인이 작업하고 있는 컴퓨터의 TF 버전이 2.0일 경우 1.0 버전 방식으로 작성된 코드를 실행하기 위해서는 별도의 작업이 필요하며, 이와 마찬가지로 본인이 작업하고 있는 컴퓨터에 1.0 버전이 설치돼 있을 경우 2.0 버전

방식으로 작성된 코드를 실행하기 어렵다.

- 하지만 사용자가 2개의 프로젝트를 진행할 때 첫 번째 프로젝트는 TF 1.0 버전 방식의 코드로 작업하고 있고 두 번째 프로젝트는 TF 2.0 버전 방식의 코드로 작업하고 있다면 하나의 버전만 이용할 수 있는 사용자는 난처해진다. 이를 해결하기 위해 가상 환경 개념이 도입된 것이다.

2. CUDA, CuDNN 설치하기

2.1 CPU vs. GPU

- 많은 수의 파라미터를 활용해 모델을 설계할 때는 CPU보다 GPU가 훨씬 빨리 계산할 수 있음.
- CPU는 GPU에 비해 고차원의 일을 수행할 수 있는 능력을 지니고 있지만, 너무 많은 수의 파라미터 값을 계산하기에는 속도가 많이 느림.
- GPU를 이용하면 파라미터 값을 병렬적으로 빠르게 계산할 수 있음.
- 엄청나게 많은 수의 파라미터에 각 파라미터별 gradient를 계산한 결과값을 계산하고, backpropagation 알고리즘을 이용해 파라미터를 업데이트하는 컴퓨터의 입장에서는 계산이 단순하기 때문에 **GPU가 딥러닝 모델을 학습시키는 데 유용하게 이용되고 있다.**

2.2 CUDA 역할 및 설치하기

- GPU를 구매해 컴퓨터에 장착하더라도 이를 파이썬에서 인식할 수 있어야 한다.
- **또한 TF, Pytorch 등 대다수의 딥러닝 프레임워크를 사용하려면 CUDA를 설치해야 한다.**
- **CUDA**는 **GPU에서 병렬 처리를 수행하는 알고리즘을 각종 프로그래밍 언어에 사용할 수 있도록 해주는 GPGPU**(General-Purpose computing on Graphics Processing Units) **기술**이다.

2.3 CuDNN 역할 및 설치하기

- **cuDNN**는 'nvidia CUDA Deep Neural Network Library'의 줄임말로, 딥러닝 모델을 위한 GPU 가속화 라이브러리의 기초 요소와 같은 일반적인 루틴을 빠르게 이행할 수 있도록 해주는 라이브러리이다.
- **TF, Pytorch를 모두 지원하며, CUDA와 반드시 함께 설치해야 한다.**
- 기존에 설치한 CUDA 버전을 고려해 설치하면 된다.

2.4 Docker란?

- 앞서 '가상 환경'은 분석가가 각 모듈별로 다른 버전을 이용해야 할 때 유용하게 이용할 수 있었다. 이 개념에서 확장돼 **모듈별로 다른 버전뿐 아니라 각 프로그래밍의 버전 및 개발 환경 자체를 독립적인 공간으로 활용해 관리할 수 있는 플랫폼**이 있는데, 그것을 바로 '**도커(Doker)**'라고 한다.
- 도커는 컨테이너 기반의 오픈 소스 가상화 플랫폼으로, 각 컨테이너 내 프로그램, 데이터베이스, 서버(Server) 등으로 다양하게 구성할 수 있고 각 컨테이너를 독립적으로 활용할 수도 있다.

4. 반드시 알아야 하는 파이토치 스킬

4.1 텐서

- **텐서(Tensor)**란 '데이터를 표현하는 단위'이다.

4.1.1 Scalar

- **스칼라**는 우리가 흔히 알고 있는 상숫값이다. 즉, **하나의 값을 표현할 때 1개의 수치로 표현한 것.**

- (ex) tensor([1.])

4.1.2 Vector

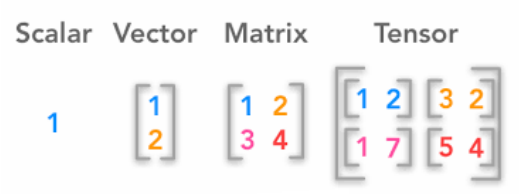
- **벡터**는 **하나의 값을 표현할 때 2개 이상의 수치로 표현한 것.**
- 스칼라의 형태와 동일한 속성을 갖고 있지만, 여러 수치 값을 이용해 표현하는 방식이다.
- (ex) tensor([1., 2., 3.])

4.1.3 행렬

- **행렬(Matrix)**은 2개 이상의 벡터 값을 통합해 구성된 값으로, 벡터 값 간 연산 속도를 빠르게 진행할 수 있는 선형 대수의 기본 단위이다.
- (ex) tensor([[1., 2.],
  [3., 4.]])

4.1.4 텐서

- 행렬을 2차원의 배열이라 표현할 수 있다면, **텐서**는 **2차원 이상의 배열**이라 표현할 수 있다.
- 출처 : https://m.blog.naver.com/nabilera1/221978354680



- (ex) tensor([[[1., 2.],
  [3., 4.]],

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

4.2 Autograd

- 파이토치를 이용해 코드를 작성할 때 **back propagation을 이용해 파라미터를 업데이트하는 방법**은 **Autograd** 방식으로 쉽게 구현할 수 있도록 설정돼 있다.

```
import torch

# cuda.is_available() : 현재 파이썬이 실행되고 있는 환경에서 torch 모듈을 이용할 때 GPU를 이용해 계산할 수 있는지 파악하는 method
if torch.cuda.is_available():
  DEVICE = torch.device('cuda')
else:
  DEVICE = torch.device('cpu')
```

```
BATCH_SIZE = 64 # input으로 이용되는 데이터가 64개라는 의미
INPUT_SIZE = 1000 # 입력 데이터의 크기가 1000이라는 의미. 즉, 1000 크기의 벡터값을 의미.
HIDDEN_SIZE = 100
OUTPUT_SIZE = 10
```

- **BATCH_SIZE**는 딥러닝 모델에서 파라미터를 업데이트할 때 계산되는 데이터 개수. 즉, BATCH_SIZE 수만큼 데이터를 이용해 output을 계산하고 BATCH_SIZE 수만큼 출력된 결과값에 대한 오차값을 계산한다.
- BATCH_SIZE 수만큼 계산된 오차값을 평균해 back propagation을 적용하고 이를 바탕으로 파라미터를 업데이트한다.
- **INPUT_SIZE**는 딥러닝 모델에서의 input의 크기이자 입력층의 노드 수를 의미.
- BATCH_SIZE = 64 이고, INPUT_SIZE = 1000 이므로 **1000 크기의 벡터값을 64개 이용**한다는 의미

  ⇒ shape : (1000, 64)
- **HIDDEN_SIZE**는 딥러닝 모델에서 input을 다수의 파라미터를 이용해 계산한 결과에 한 번 더 계산되는 파라미터의 수를 의미. 즉, 입력층에서 은닉층으로 전달됐을 때 은닉층의 노드 수를 의미.

  ⇒ **(1000, 64)의 input**들이 **(1000, 100) 크기의 행렬**과 행렬 곱을 계산하기 위해 설정한 수
- **OUTPUT_SIZE**는 딥러닝 모델에서 최종으로 출력되는 값의 벡터의 크기를 의미. 보통 output의 크기는 최종으로 비교하고자 하는 레이블의 크기와 동일하게 설정한다.
- 예를 들어, (ex1) 10개로 분류하려면 크기가 10짜리의 원-핫 인코딩을 이용하기 때문에 output의 크기를 10으로 맞추기도 함. (ex2) 5 크기의 벡터값에 대해 Mean Square Error를 계산하기 위해 output의 크기를 5로 맞추기도 함.

```
# randn : 평균 0, 표준편차 1인 정규분포에서 샘플링한 값으로 데이터를 만든다는 것을 의미 (데이터를 만들어낼 때 아래처럼 모양을 설정할 수 있음.)

# 크기가 1000개짜리의 벡터를 64개 만들기
x = torch.randn(BATCH_SIZE, # 64
                INPUT_SIZE, # 1000
                device = DEVICE, # 생성된 데이터는 미리 설정한 DEVICE를 이용해 계산할 것
                dtype = torch.float, # 데이터 형태는 float
                requires_grad = False) # 이 데이터는 input으로 이용되기 때문에 gradient를 계산할 필요 x
# x의 shape (64, 1000)

y = torch.randn(BATCH_SIZE, # 64 / output도 BATCH_SIZE의 수만큼 결과값 필요
                 OUTPUT_SIZE, # 10 / output과의 오차를 계산하기 위해 크기 10으로 설정
                 device = DEVICE,
                 dtype = torch.float,
                 requires_grad = False)
# y의 shape (64, 10)

# 업데이트할 파라미터 값 설정
w1 = torch.randn(INPUT_SIZE, # 1000
                 HIDDEN_SIZE, # 100
                 device = DEVICE,
                 dtype = torch.float,
                 requires_grad = True) # gradient를 계산할 수 있도록 설정
# w1의 shape (1000, 100)

w2 = torch.randn(HIDDEN_SIZE, # 100
                  OUTPUT_SIZE, # 10
                  device = DEVICE,
                  dtype = torch.float,
                  requires_grad = True) # gradient를 계산할 수 있도록 설정
# w2의 shape (100, 10)
```

- **randn** : 평균 0, 표준편차 1인 정규분포에서 샘플링한 값으로 데이터를 만든다는 것을 의미 (데이터를 만들어낼 때 위처럼 모양을 설정할 수 있음.)
- **requires_grad** : 파라미터 값을 업데이트하기 위해 gradient를 계산

  → back propagation 통해 업데이트해야 하는 대상이 아닌 input, output 데이터는 gradient를 계산할 필요가 없다.

  → x, y는 **required_grad = Fasle**로 설정

```
learning_rate = 1e-6
for t in range(1, 501): # 500번 반복해 파라미터값 업데이터
  y_pred = x.mm(w1).clamp(min = 0).mm(w2) # 예측값

  loss = (y_pred - y).pow(2).sum() # 제곱차의 합
  if t % 100 == 0: # 반복 횟수를 의미하는 t가 100으로 나누어 떨어질 때마다 t, loss 출력
    print('Iteration: ', t, '\t', 'Loss: ', loss.item())
  loss.backward() # 각 파라미터 값에 대해 gradient를 계산하고, 이를 통해 backpropagation 진행

  with torch.no_grad(): # 코드가 실행되는 시점에서 gradient 값을 고정
    w1 -= learning_rate * w1.grad
    w2 -= learning_rate * w2.grad

    # 각 파라미터 값의 gradient를 초기화해 다음 반복문 진행할 수 있도록 gradient = 0으로 설정
    # 다음 backpropagation을 진행할 때 gradient 값을 loss.backward()을 통해 새로 계산하기 때문
    w1.grad.zero_()
    w2.grad.zero_()
```

- **learning_rate** : gradient를 계산한 결과값에 learning_rate만큼 곱한 값을 이용해 파라미터를 업데이트 한다. (gradient값에 따른 학습 정도 결정)
- **y_pred = x.mm(w1).clamp(min = 0).mm(w2)**

  h = x.mm(w1) # 행렬곱

  h_relu = h.**clamp**(min = 0) # 비선형 함수(relu) 적용

  y_pred = h_relu.mm(w2) # 행렬곱

  을 한 줄로 표현한 것.
- **clamp**
  - 딥러닝 모델에서는 층과 층 사이에 **비선형 함수**를 이용해 높은 표현력을 지니는 방정식을 얻게 된다.
  - 여기서 clamp는 **비선형 함수 ReLU()와 같은 역할**을 한다. (최솟값이 0이며, 0보다 큰 값은 자기 자신을 갖게 되는 메서드이기 때문)

$$y_i = \begin{cases} min(\text{ if } x_i < min) \\ x_i(\text{ if } min \leq x_i \leq max) \\ max(\text{ if } x_i > max) \end{cases}$$

- **torch.no_grad()**: 각 파라미터 값에 대해 gradient를 계산한 결과를 이용해 파라미터 값을 업데이트할 때는 해당 시점의 gradient 값을 고정한 후 업데이트를 진행. 코드가 실행되는 시점에서 gradient 값을 고정한다는 의미
- **실행 결과**

```
Iteration:  100     Loss:  640.6591796875
Iteration:  200     Loss:  6.606578826904297
Iteration:  300     Loss:  0.1152307391166687
Iteration:  400     Loss:  0.0027284740936011076
Iteration:  500     Loss:  0.00021828098397236317
```

→ 500번의 반복문을 실행하면서 **loss 값이 줄어드는 것**을 확인할 수 있다.

- **loss 값이 줄어든다**

  = input이 w1과 w2를 통해 계산된 결과값과 y값이 점점 비슷해진다!

= y값과 비슷한 output을 계산할 수 있도록 w1과 w2가 계산된다!