3장. Hello 파이토치

2022-09-18

3.1 딥러닝을 시작하기 전에
3.1.1 장비 구성

NLP Lab
Natural Language Processing

- CPU (central processing unit)
  - 일부 피할 수 없는 순차적인 연산만 담당
  - 1~2개 코어에 집중되어 100% 내외 사용
  - 코어의 숫자 뿐만 아니라 개별 코어의 클럭이 높아야 함
  - 전처리 과정 또는 단어 임베딩 단계에서 중요
  - GPU (graphics processing unit)
    - 잘 설계된 파이토치 코드 대부분 사용
    - 보통의 병렬 연산을 담당
- RAM
  - 최소 16GB ~ 32GB
  - 많은 양의 데이터셋을 한번에 메모리에 로드
  - 전처리 등

3.1.1 장비 구성

- GPU
  - 엔비디아 GTX (또는 RTX) 계열
  - CUDA와 함께 동작하므로 라데온 (Radeon) 은 소용없음
  - CUDA 코어가 많고 클럭이 높을수록 속도가 빠름
  - 메모리 대역폭 (bandwidth) 중요
  - 메모리의 크기도 중요
- 파워 서플라이 (power supply)
  - 메인보드, 그래픽카드 모두를 감당할 수 있어야 함
  - 700W 이상 필요
- 쿨링 시스템
  - GPU에서 다량의 열이 발생
  - 수냉 시스템도 좋은 선택

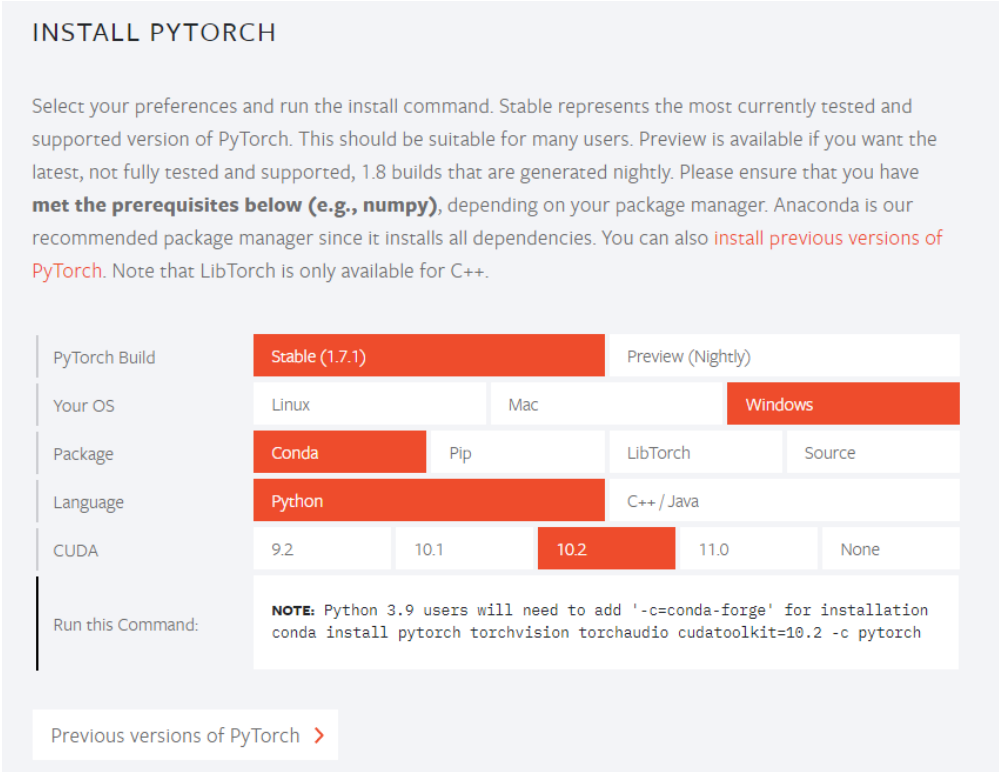
3.1.1 장비 구성

부품	요약	최소	권장
CPU	코어 개수보다는 단일 클럭이 높아야 함	i5	i7
RAM	메모리는 많을수록 좋음	16GB	64GB
GPU	메모리가 클수록 좋지만, 비용이 비쌈	GTX 1060Ti	RTX 2080Ti
파워 서플라이 (supply)	비싸고 검증된 브랜드 선택	GPU 개당 500W	--
쿨링 시스템 (cooling)	쿨링은 매우 중요, 난방비 절약 가능	--	--

3.2 설치 방법

- 아나콘다 설치
  - https://www.anaconda.com/products/individual#linux

- 파이토치 설치
  - https://pytorch.org/
  - 옵션 선택
  - 해당 명령어 실행

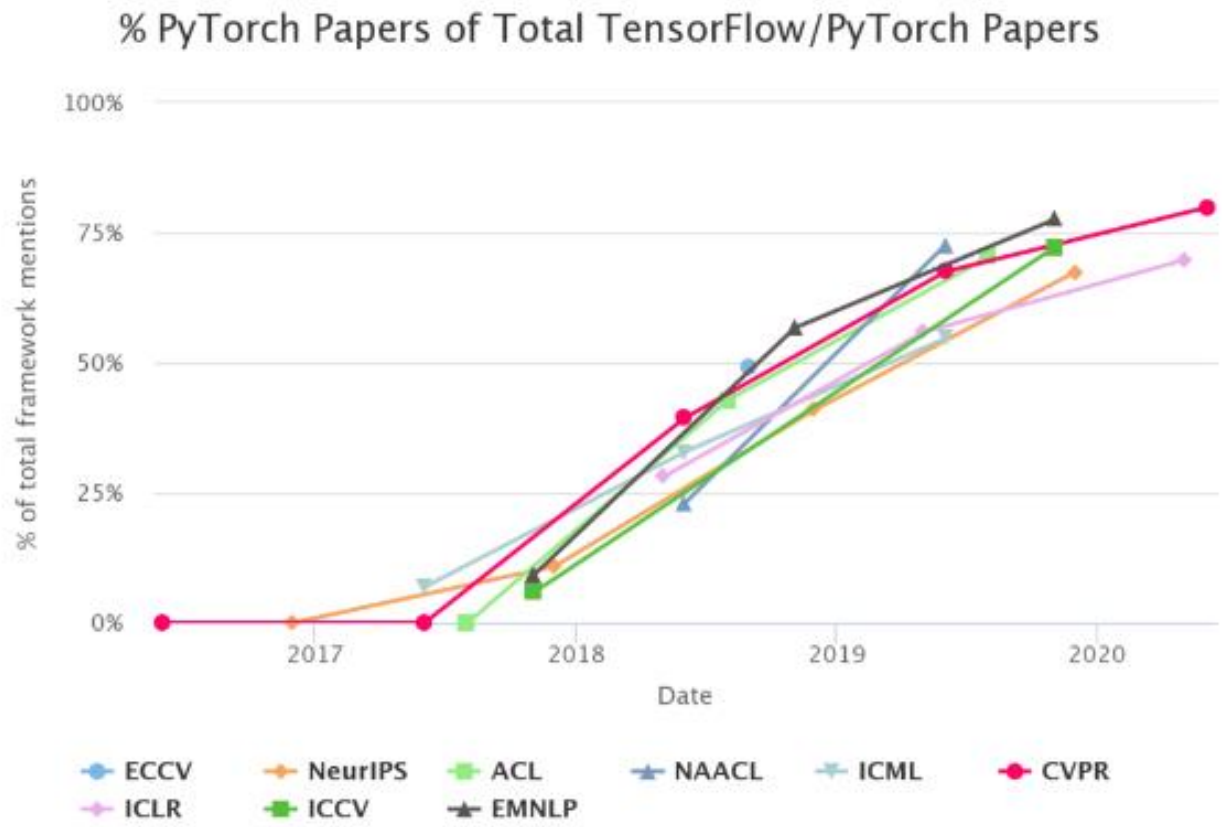


3.2.3 왜 파이토치일까?

NLP Lab
Natural Language Processing

- 텐서플로 (Tensorflow)
  - 구글이 개발, 자체 딥러닝 전용 하드웨어 TPU에 최적화
- 파이토치 (PyTorch)
  - 페이스북의 주도 아래 개발이 진행
  - 엔비디아도 참여
    - 엔비디아의 CUDA GPU에 더 최적화
    - 적극 파이토치를 권장
    - 자연어 처리 분야에서 더욱 적극적으로 권장
  - 파이토치에 비해 뒤늦게 딥러닝 프레임워크 개발에 참여하여 적은 유저풀을 유지
  - 비슷한 레벨의 구현 난도를 가정하면 파이토치가 훨씬 뛰어난 생산성을 보임
  - 학계에서는 파이토치의 사용이 증가

3.2.3 왜 파이토치일까?


% PyTorch Papers of Total TensorFlow/PyTorch Papers
100%
75%
50%
25%
0%
% of total framework mentions
2017
2018
2019
2020
Date
ECCV
NeurIPS
ACL
NAACL
ICML
CVPR
ICLR
ICCV
EMNLP

3.2.3 왜 파이토치일까?

- Tensorflow : Google Co. / pytorch : Facebook Co.

① 깔끔한 코드
- Tensorflow와 달리 Python을 우선(Python First)으로 하여 대부분의 모듈이 파이썬으로 짜여 있어 코드가 깔끔
- Tensorflow에 비해 버전업에 따른 파편화가 상대적으로 적음

② 넘파이와 뛰어난 호환성

③ Autograd
- 값을 Feed-forward하여 계산하고 backward() 호출로 역전파 알고리즘 수행가능

④ 동적 그래프
- 연산과 동시에 Dynamic Graph가 생성되어 tensorflow에 비해 자유로움

3.3 짧은 튜토리얼

- 텐서 (Tensor) : ndarray와 같은 개념

```
import torch
import numpy as np

x1 = torch.Tensor([[1,2],[3,4]])
x2 = torch.from_numpy(np.array([[1,2],[3,4]]))
x3 = np.array([[1,2],[3,4]])
```

```
tensor([[1., 2.],
        [3., 4.]])
tensor([[1, 2],
        [3, 4]], dtype=torch.int32)
[[1 2]
 [3 4]]
```

Data type	dtype	CPU tensor	GPU tensor
32-bit floating point	`torch.float32` or `torch.float`	`torch.FloatTensor`	`torch.cuda.FloatT`
64-bit floating point	`torch.float64` or `torch.double`	`torch.DoubleTensor`	`torch.cuda.Double`
16-bit floating point	`torch.float16` or `torch.half`	`torch.HalfTensor`	`torch.cuda.HalfTe`
8-bit integer (unsigned)	`torch.uint8`	`torch.ByteTensor`	`torch.cuda.ByteTe`
8-bit integer (signed)	`torch.int8`	`torch.CharTensor`	`torch.cuda.CharTe`
16-bit integer (signed)	`torch.int16` or `torch.short`	`torch.ShortTensor`	`torch.cuda.ShortT`
32-bit integer (signed)	`torch.int32` or `torch.int`	`torch.IntTensor`	`torch.cuda.IntTen`
64-bit integer (signed)	`torch.int64` or `torch.long`	`torch.LongTensor`	`torch.cuda.LongTe`

- Autograd : 자동 gradient 계산

```
x = torch.FloatTensor(2,2)
y = torch.FloatTensor(2,2)
y.requires_grad_(True)

with torch.no_grad():
    z = (x+y) + torch.FloatTensor(2,2)
```

```
tensor([[4.1069e-08, 2.1271e+23],
        [7.9875e+20, 1.0357e-11]])
tensor([[1.0000, 4.2500],
        [3.0000, 4.0000]], requires_grad=True)
tensor([[1.0000e+00, 2.1271e+23],
        [7.9875e+20, 4.0000e+00]], grad_fn=<AddBackward0>)
```

- 자동으로 미분, 역전파 수행, 동적 그래프 생성
- 비 학습과정에서는 with torch.no_grad()로 수행

3.3 짧은 튜토리얼

- 피드 포워드 (Feed-Forward) : 원하는 연산을 통해 값을 앞으로 전달

```
def linear(x, W, b):
    y = torch.mm(x,W) + b
    return y

x = torch.FloatTensor(16,10)
W = torch.FloatTensor(10,5)
b = torch.FloatTensor(5)

y = linear(x,W,b)
```

- Ex) linear layer 구현
  - X: 16x10 행렬 / W (weight): 10x5 행렬
  - b (bias): 1x5 행렬
- Torch.mm : 행렬 곱=np.matmul

- nn.Module

```
class MyLinear(nn.Module):
    def __init__(self, input_size, output_size):
        super(MyLinear, self).__init__()
        self.linear = nn.Linear(input_size, output_size)

    def forward(self, x):
        y = self.linear(x)
        return y
```

```
x = torch.FloatTensor(16,10)
linear = MyLinear(10,5)
y = linear(x)
```

3.3 짧은 튜토리얼

- 역전파 수행

```
objective = 100

x = torch.Tensor(16,10)
linear = MyLinear(10,5)
y = linear(x)
loss = (objective - y.sum())

loss.backward()
```

- Loss 값은 scalar로 표현됨

```
print(loss)
```

```
tensor(91.1572, grad_fn=<RsubBackward1>)
```

- Train()과 eval()

```
# training..
linear.eval()
# do something..
linear.train()
# restart training..
```

```
MyLinear(
  (linear): Linear(in_features=10, out_features=5, bias=True)
)
```

- 훈련, 평가시에 모드 전환이 쉬움

- GPU 사용하기

```
x = torch.cuda.FloatTensor(16,10)
linear = MyLinear(10,5)
linear.cuda()
y = linear(x)
```

- 파이토치는 GPU CUDA에 의존하는 편

3.3 짧은 튜토리얼

- 선형회귀분석 예제
  1. 임의로 Tensor 생성
  2. 정답 함수에 넣어 정답(y) 구함
  3. 그 정답과 신경망을 통과한 정답(y^)과의 차이를 평균 제곱오차(MSE)를 통해 구함
  4. 확률적 경사 하강법(SGD)로 최적화

```
class MyModel(nn.Module): # nn.Module로 모델 아키텍처 클래스 선언
    def __init__(self, input_size, output_size): # 클래스 객체 생성
        super(MyModel, self).__init__()
        self.linear = nn.Linear(input_size, output_size)

    def forward(self, x): #선형레이어
        y = self.linear(x)
        return y
```

```
def ground_truth(x): #임의의 함수(실제 측정 정답)
    return 3*x[:,0] + x[:,1] - 2*x[:,2]
```

3.3 짧은 튜토리얼

- 훈련 함수 생성, hyperparameter 설정

```
def train(model, x, y, optim):
    optim.zero_grad()

    y_hat = model(x)

    loss = ((y - y_hat)**2).sum() / x.size(0) # MSE loss 계산
    loss.backward() # backward() 호출 -> gradient 채워짐

    optim.step() # step()을 호출하여 경사하강법(gradient descent) 수행

    return loss.data
```

```
batch_size = 1
n_epochs = 1000
n_iter = 10000

model = MyModel(3,1)
optim = torch.optim.SGD(model.parameters(), lr=0.0001, momentum=0.1) # SGD optimizer 생성, 모델의 파라미터를 최적화
print(model)
```

```
MyModel(
  (linear): Linear(in_features=3, out_features=1, bias=True)
)
```

3.3 짧은 튜토리얼

- 최종 훈련, 평가
  - x : 임의의 랜덤 tensor
  - Y : 실제 답
  - x_valid : 평가 Tensor
  - y_valid : 평가 실제 답
  - y_hat : 모델 예측 답

  - Print(조건충족 loss, 실제 답, 예측 답)

```
for epoch in range(n_epochs):
    avg_loss = 0

    for i in range(n_iter):
        x = torch.rand(batch_size, 3)
        y = ground_truth(x.data)

        loss = train(model, x, y, optim) # loss function을 통해 손실값(loss) 계산

        avg_loss += loss
        avg_loss = avg_loss / n_iter

    x_valid = torch.FloatTensor([[.3, .2, .1]])
    y_valid = ground_truth(x_valid.data)

    model.eval()
    y_hat = model(x_valid)
    model.train()

    print(avg_loss, y_valid.data[0], y_hat.data[0,0])

    if avg_loss < .001:
        break
```

```
tensor(4.7900e-07) tensor(0.9000) tensor(0.7654)
```

3.3 짧은 튜토리얼

- 파이토치 딥러닝 수행 과정
  I. nn.Module 을 상속받아 forward 함수를 통해 모델 클래스 생성
  II. SGD나 Adam과 같은 optimizer 생성, 파라미터 최적화
  III. 데이터로 미니배치를 구성하여 feed-forward 연산
  IV. Loss함수를 통해 최종 결과값(scalar), 손실 값(loss) 계산
  V. 손실에 대해 backward() 호출 -> 기울기(gradient) 채워짐
  VI. Optimizer의 step()을 호출하여 경사 하강법(gradient descent) 수행
  VII. 수렴조건이 만족할 때까지 loop 반복 수행