03

Pytorch 기본 문법과 LSTM 구현

03 Pytorch 기본 문법과 LSTM 구현

오늘 학습을 통해 우리는

- Pytorch 프레임워크의 기능과 구조에 대해 알아봅니다.

- Pytorch 프레임워크 사용법을 익히고, 이를 바탕으로 딥러닝 모델을 만들어 훈련시킵니다.

- Pytorch를 이용하여 간단한 LSTM 모델을 구현하고, 감성분류 작업을 체험해봅니다.

목차

01

Pytorch 소개

딥러닝 프레임워크

- 딥러닝 모델을 설계, 훈련, 평가 및 배포하는 데 도움을 주는 소프트웨어 라이브러리
- GPU 등 하드웨어를 사용하여 자동 미분 등 복잡한 수학적 연산을 쉽게 처리
- 수많은 라이브러리와 사전 학습까지 완료된 다양한 딥러닝 알고리즘을 제공

딥러닝 프레임워크의 필요성

- 복잡한 연산의 자동화: 수천, 수만의 파라미터를 가진 딥러닝 모델의 연산을 효율적으로 처리
- GPU 가속: 대규모 행렬 연산을 빠르게 처리하기 위한 GPU 지원
- 자동 미분: 역전파 알고리즘 구현의 복잡성 감소

딥러닝 프레임워크의 필요성

- 다양한 최적화 알고리즘: SGD, Adam 등 다양한 최적화 알고리즘 지원

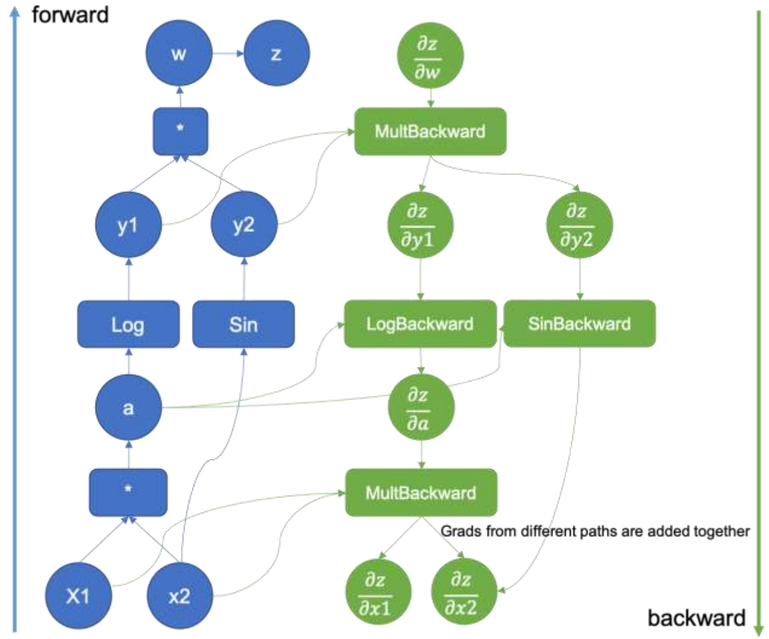
- 커뮤니티 및 라이브러리: 활발한 커뮤니티 지원과 다양한 라이브러리 활용 가능성

출처: https://discuss.pytorch.kr/uploads/default/original/1X/40d969600704f473ee638268df8651e8137bd49b.png

PyTorch의 특징 및 장점

- 동적 계산 그래프 (Dynamic Computation Graph)
  - Define-by-Run 방식: 코드를 실행하는 방식대로 그래프 생성
  - 유연한 아키텍처 설계: 모델 구조 변경이 용이

- 직관적인 API 디자인
  - Pythonic한 디자인: Python 사용자에게 친숙한 문법과 구조
  - 빠른 프로토타이핑: 연구 및 개발 시간 단축



출처 : https://tutorials.pytorch.kr/, https://pytorch.org/blog/computational-graphs-constructed-in-pytorch/

PyTorch의 특징 및 장점

- 강력한 GPU 가속
  - CUDA 지원: NVIDIA GPU를 통한 빠른 연산
  - Multi-GPU 지원: 병렬 처리를 통한 훈련 시간 단축

- 풍부한 라이브러리 및 확장성
  - torchvision, torchaudio, torchtext 등 다양한 도메인에 특화된 라이브러리
  - 사용자 정의 연산 및 레이어 확장 용이

출처 : https://tutorials.pytorch.kr/,

PyTorch의 특징 및 장점

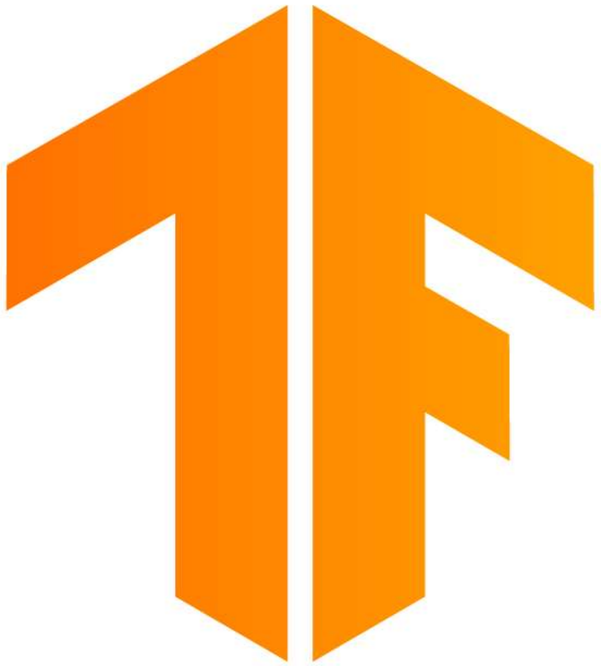
- 활발한 커뮤니티 지원
  - 다양한 튜토리얼, 문서, 예제 코드 제공
  - 연구 및 산업 분야에서의 넓은 활용

- 모델의 이식성
  - ONNX(Open Neural Network Exchange) 지원: 다른 프레임워크와의 호환성
  - 모바일 및 임베디드 디바이스 지원

출처 : https://tutorials.pytorch.kr/,

PyTorch와 다른 프레임워크 비교

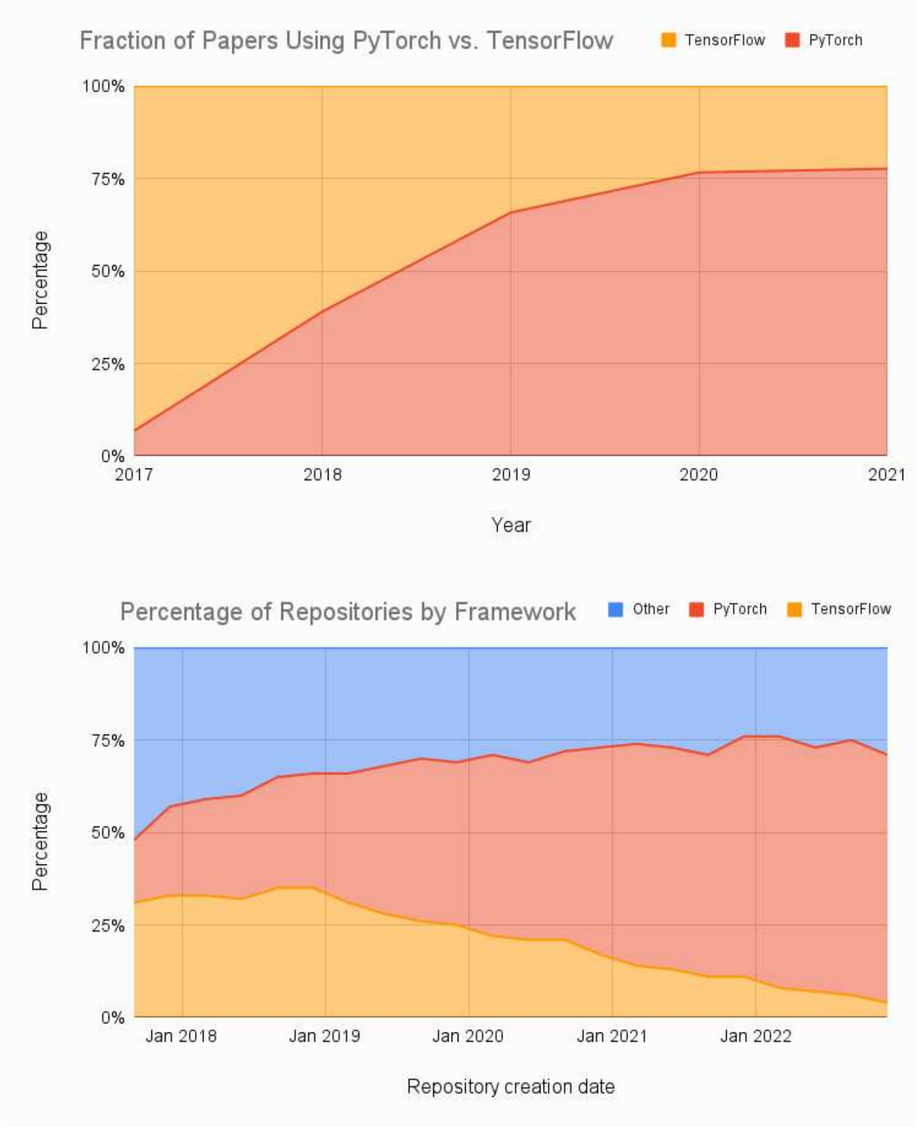
- TensorFlow
  - 계산 그래프:
    - TensorFlow: Define-and-Run 방식 (v1.x), Define-by-Run 방식 (v2.x 이후)
    - PyTorch: Define-by-Run 방식
  - 배포:
    - TensorFlow: TensorFlow Serving, TFLite (모바일 및 임베디드)
    - PyTorch: TorchServe, TorchScript



출처: https://www.tensorflow.org/federated?hl=ko

PyTorch와 다른 프레임워크 비교

- TensorFlow
  - API:
    - TensorFlow: Keras를 통한 고수준 API 제공
    - PyTorch: 직관적이고 Pythonic한 API 디자인
  - 커뮤니티 및 지원:
    - TensorFlow: Google 백업, 넓은 사용자 기반
    - PyTorch: Facebook 백업, 연구 커뮤니티에서 선호



출처: https://www.assemblyai.com/blog/pytorch-vs-tensorflow-in-2023/

PyTorch와 다른 프레임워크 비교

- Keras
  - 특징:
    - Keras: 고수준 API, TensorFlow, Theano, CNTK 등 다양한 백엔드 지원
    - PyTorch: 중/저수준 API, 독립적인 프레임워크
  - 사용자 친화성:
    - Keras: 초보자에게 친숙한 구조, 빠른 프로토타이핑
    - PyTorch: 유연성과 세부 조정 가능성

출처: https://keras.io/

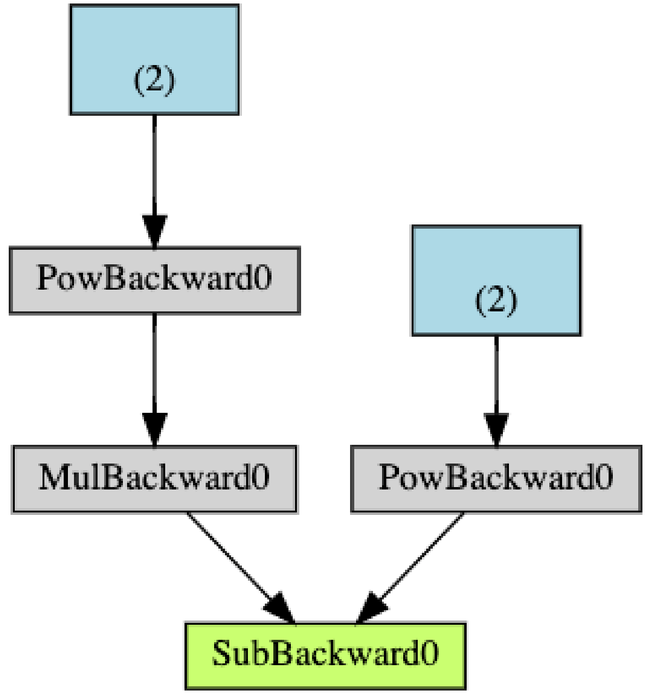
PyTorch와 다른 프레임워크 비교

- Caffe2
  - 특징:
    - Caffe: 이미지 분류에 초점, C++/Python API 제공
    - PyTorch: 다양한 딥러닝 작업에 적용 가능, 주로 Python API
    - 2018년 이후 Pytorch에 병합
  - 성능:
    - Caffe: 고성능, 특히 이미지 분류 작업에 최적화
    - PyTorch: 다양한 작업에 대한 높은 성능 및 유연성

출처: https://caffe2.ai/

Pytorch 구성 API

- Autograd
  - 자동 미분 시스템
  - 연산 그래프의 역전파를 통한 그래디언트 계산
  - requires_grad 속성을 통한 그래디언트 추적 제어



출처: https://tutorials.pytorch.kr/beginner/blitz/autograd_tutorial.html

Pytorch 구성 API

- nn.Module
  - 딥러닝 모델의 기본 구성 단위
  - 사용자 정의 레이어 및 모델 구성
  - forward 메서드를 통한 순전파 정의

```python
import torch.nn as nn
import torch.nn.functional as F

class Model(nn.Module):
    def __init__(self):
        super().__init__()
        self.conv1 = nn.Conv2d(1, 20, 5)
        self.conv2 = nn.Conv2d(20, 20, 5)

    def forward(self, x):
        x = F.relu(self.conv1(x))
        return F.relu(self.conv2(x))
```

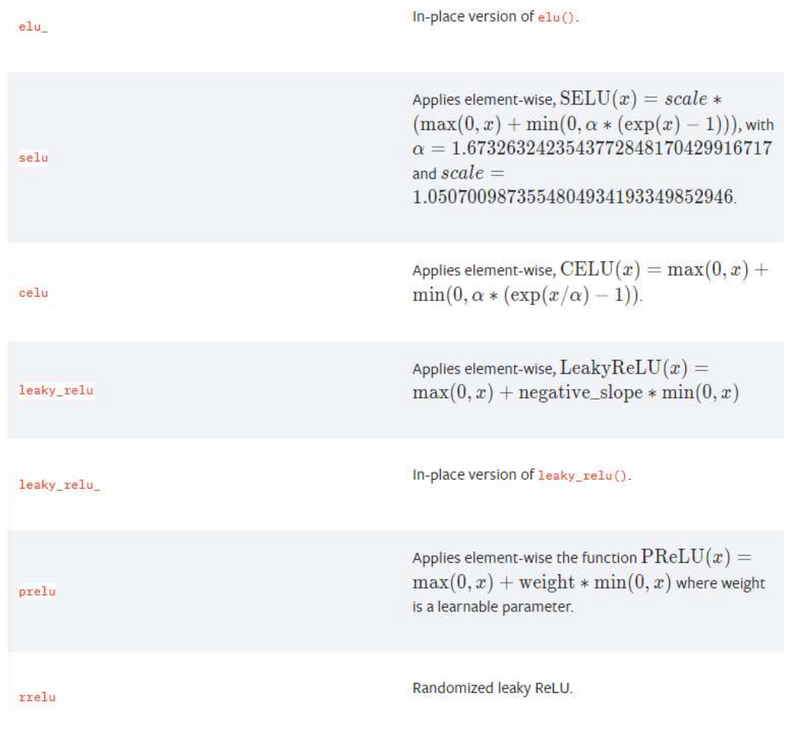
출처: https://pytorch.org/docs/stable/generated/torch.nn.Module.html#torch.nn.Module

Pytorch 구성 API

- nn.Parameter
  - 모델의 학습 가능한 파라미터
  - 자동 기울기 계산: requires_grad=True로 설정되어 있음
  - 모델의 가중치 및 편향과 같은 학습 가능한 변수에 사용
    - 모듈에 자동 등록
    - 간단한 초기화 지원

Pytorch 구성 API

- nn.functional
  - 딥러닝 연산에 사용되는 함수 모음
    - 상용화된 대부분의 함수 포함
  - 활성화 함수, 손실 함수, 정규화 함수 등 수많은 레이어 보유
  - DDP(Distributed Data Parelle)학습을 위한 함수 제공

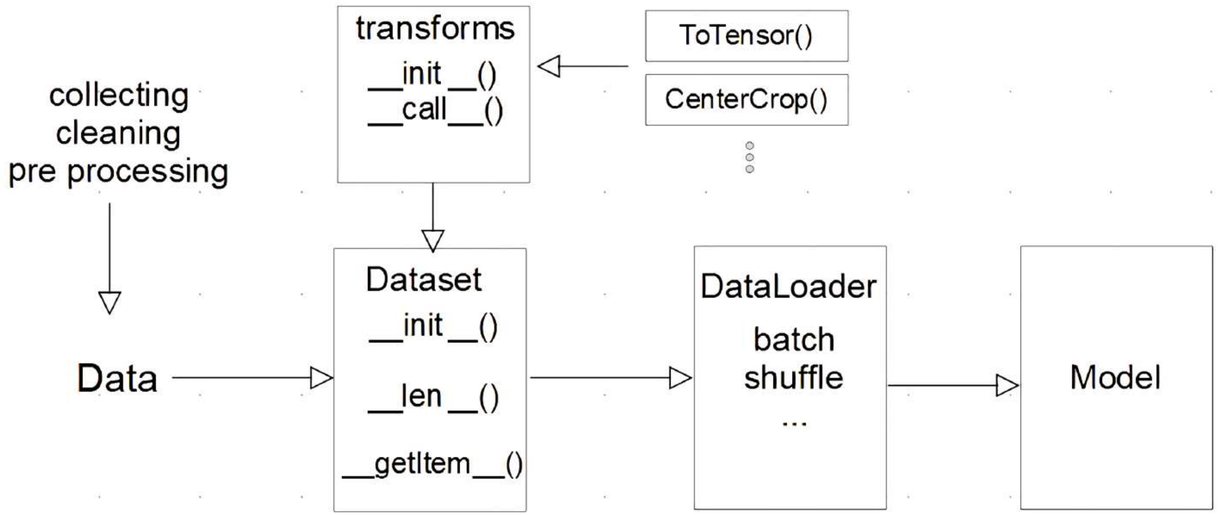


elu_	In-place version of elu().
selu	Applies element-wise, $\text{SELU}(x) = scale * (\max(0,x) + \min(0, \alpha * (\exp(x) - 1)))$, with $\alpha = 1.6732632423543772848170429916717$ and $scale = 1.0507009873554804934193349852946$.
celu	Applies element-wise, $\text{CELU}(x) = \max(0,x) + \min(0, \alpha * (\exp(x/\alpha) - 1))$.
leaky_relu	Applies element-wise, $\text{LeakyReLU}(x) = \max(0,x) + \text{negative\_slope} * \min(0,x)$
leaky_relu_	In-place version of leaky_relu().
prelu	Applies element-wise the function $\text{PReLU}(x) = \max(0,x) + \text{weight} * \min(0,x)$ where weight is a learnable parameter.
rrelu	Randomized leaky ReLU.

출처 : https://pytorch.org/docs/stable/nn.functional.html

Pytorch 구성 API

- Dataset & DataLoader
  - 데이터 로딩 및 전처리를 위한 유틸리티
  - Dataset: 사용자 정의 데이터셋 구성
  - DataLoader: 배치 처리, 셔플링, 병렬 로딩 등의 기능 제공



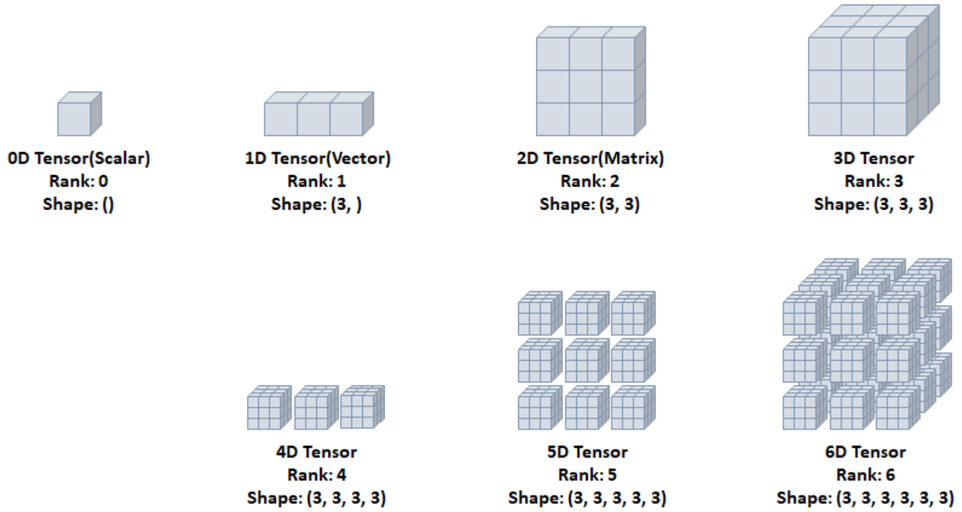
출처 : https://velog.io/@naem1023/pytorch-dataset-dataloader

02

Pytorch 기본 문법

텐서(Tensor)의 이해

- 다차원 배열로, PyTorch의 핵심 데이터 구조
- 다양한 딥러닝 연산 및 알고리즘을 구현
- GPU 가속 연산 지원
- Numpy와의 호환성이 높음



출처 : https://moonss-0913.tistory.com/15

텐서 생성 및 조작 방법

- 직접 생성

```
import torch

x = torch.tensor([1, 2, 3])
```

- 0 혹은 1로 채워진 텐서

```
a = torch.zeros(2, 3)
```

```
b = torch.ones(2, 3)
```

텐서 생성 및 조작 방법

- 균일한 간격으로 값 생성

```
import torch

x = torch.linspace(0, 10, 5)

print(x)
```

- 무작위 값으로 생성

```
x = torch.rand(2, 3)

print(x)
```

텐서 기본 연산

- 사칙연산, 행렬곱, 행렬 내 합과 평균

```
# 기본 연산
x = torch.tensor([1, 2, 3])

y = torch.tensor([4, 5, 6])

z = x + y
```

```
# 행렬곱
a = torch.rand(2, 3)

b = torch.rand(3, 2)

c = torch.mm(a, b)


c
```

```
# 텐서의 합과 평균
s = torch.sum(x)
print("sum: ", s)

m = torch.mean(x.float())
print("mean: ", m)
```

CUDA 텐서

- GPU와 CUDA Toolkit, NVIDIA 드라이버가 준비되었다면 Pytorch에서 사용 가능
- torch.cuda.is_available() 메서드로 사용 여부 확인 가능
- 아래와 같이 간단하게 CUDA에서 텐서 사용 가능

```
x = torch.rand(2, 3)

# 텐서를 Cuda로 이동
if torch.cuda.is_available():
    print("Device: CUDA")
    x = x.cuda()
```

PyTorch와 NumPy 간 상호 운영성

- PyTorch의 텐서와 NumPy의 배열은 메모리 상에서 서로 공유될 수 있음
- 두 라이브러리 간의 변환에 별도의 메모리 복사가 필요 없어 빠른 연산이 가능

- 두 라이브러리의 장점을 모두 이용할 수 있음
  - NumPy는 다양한 수학적 연산 및 배열 조작 기능 제공
  - PyTorch는 GPU 가속 및 딥러닝 연산 제공

PyTorch와 NumPy 간 상호 운영성

- NumPy 배열과 PyTorch 텐서의 상호 변환

```python
# Numpy array to torch tensor
import numpy as np
import torch

numpy_array = np.array([1, 2, 3])
torch_tensor = torch.from_numpy(numpy_array)
type(torch_tensor)
```

```python
# Torch tensor to np array
torch_tensor = torch.tensor([1, 2, 3])
numpy_array = torch_tensor.numpy()

type(numpy_array)
```

PyTorch와 NumPy 간 상호 운영성

- 메모리 공유
  - 텐서나 배열 중 하나를 변경하면 다른 하나도 자동으로 변경됨

```
# np array to torch tensor
numpy_array = np.array([1, 2, 3])
torch_tensor = torch.from_numpy(numpy_array)
type(torch_tensor)
```

```
numpy_array[0] = 10
print(torch_tensor)

# tensor([10, 2, 3])
```

PyTorch에서 GPU 사용하기

- CUDA 지원 확인
  - PyTorch에서는 NVIDIA의 CUDA를 통해 GPU 연산을 지원

```
# Device 사용 여부 체크
is_available = torch.cuda.is_available()

print(is_available)
```

- GPU 장치 선택
  - 기본적으로 PyTorch 텐서는 CPU에 생성됨
  - GPU 사용을 위해서는 텐서를 CUDA 장치로 명시적으로 이동시켜야 함

PyTorch에서 GPU 사용하기

- GPU 장치 선택

```
# Device를 GPU로 설정
device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")

device
```

- 텐서와 모델 GPU로 이동

```
# 텐서를 GPU로 이동
x = torch.tensor([1, 2, 3])
x = x.to(device)

# 또는
x = x.cuda()  # 만약 GPU가 사용 가능한 경우
```

```
# 임의의 모델 SomeModel()의 경우
from torch import nn

model = nn.Sequential()
model.to(device)
```

PyTorch에서 GPU 사용하기

- 여러 GPU 병렬로 사용하기 (Data Parallelism)
  - 여러 GPU를 사용하여 모델을 병렬로 학습시키는 것이 가능
  - 대형 모델의 경우 병렬 학습이 필수

```
# 복수의 GPU가 있을 경우 병렬 사용
if torch.cuda.device_count() > 1:
    model = nn.DataParallel(model)

model.to(device)
```

PyTorch에서 GPU 사용하기

- 주의 사항
  - GPU와 CPU 간의 데이터 전송은 상대적으로 느림
  - 가능한 연산을 일괄적으로 처리하고, 불필요한 데이터 전송을 최소화하는 것이 좋음
  - GPU 메모리(VRAM)는 한정적이므로, 큰 모델 또는 대량의 데이터를 처리할 때는 메모리 사용량에 주의해야 함
  - 모든 데이터와 모델이 동일한 하드웨어 상에 존재해야 함

Device-agnostic 코드

- 매번 데이터와 모델에게 하드웨어 장치를 지정하는 것은 번거로움

- Device-agnostic하게 코드를 작성하면 위의 문제에서 벗어날 수 있음

- Device-agnostic 코드란 CPU나 GPU와 같은 특정 장치에 종속되지 않고, 실행 환경에 따라 유연하게 동작하는 코드를 의미

Device-agnostic 코드

- Device-agnostic 코드를 만들기 위해선 아래 원칙에 따라야 함
  - 장치 선택: 실행 환경에 따라 자동으로 CPU 또는 GPU를 선택

```
# Device 설정
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")

print(device)
```

  - 텐서 및 모델 초기화: 텐서나 모델을 초기화할 때 device를 사용하여 적절한 장치에 할당

```
# 텐서와 모델을 장치로 이동
x = torch.tensor([1, 2, 3], device=device)
model = nn.Sequential().to(device)
```

Device-agnostic 코드

- 데이터 로딩: 데이터 로더에서 배치를 가져올 때마다 해당 데이터를 적절한 장치로 이동

```
# 데이터 로딩 후 장치로 이동
for epoch in epochs:
    for inputs, labels in dataloader:
        inputs, labels = inputs.to(device), labels.to(device)
        y_pred = model(inputs)
        loss = loss_func(y_pred-labels)
```

- 모델 저장 및 불러오기:
  - 모델을 저장하거나 로드할 때도 장치에 주의
  - 모델을 CPU로 로드하고 싶다면 map_location 인자를 사용

```
# 모델 저장하기
torch.save(model.state_dict(), "model.pth")

# 모델 불러오기, map_location에 주의
model.load_state_dict(torch.load("model.pth", map_location=device))
```

Device-agnostic 코드

- 추론
  - 학습된 모델을 다른 환경(서버, 임베디드)에서 사용할 때도 device-agnostic 코드를 유지
- 여러 GPU 사용
  - 여러 GPU가 있는 경우에도 device-agnostic 코드를 작성하면, GPU의 수에 관계없이 동작

```
# 복수의 GPU가 있을 경우 병렬 사용
if torch.cuda.device_count() > 1:
    model = nn.DataParallel(model)

model.to(device)
```

Autograd

- PyTorch의 자동 미분 엔진
- 연산 그래프를 기반으로 그래디언트 계산

```
# Autograd 적용
x = torch.tensor(1.0, requires_grad=True)
y = x ** 2

print(y)
# tensor(1., gradfn=<PowBackward0>)
```

- requires_grad 속성: 텐서의 그래디언트 계산 필요성 지정

Autograd

- 연산 그래프: 수행된 각 연산을 노드로 하는 그래프 생성

```
a = torch.tensor([2, 3], dtype=torch.float32)
b = torch.tensor([6, 4], dtype=torch.float32, requires_grad=True)

result = a + b

print(result)
# tensor([8., 7.], grad_fn=<AddBackward0>)
```

- backward() 메서드: 그래디언트 계산 시작
  - 결과는 .grad 에 저장

```
# 그래디언트 계산
result.sum().backward()
```

```
# 미분 연산
print(b.grad)
```

Autograd

- 그래디언트 초기화
  - 반복적인 학습 시 그래디언트 누적 방지
  - .grad.zero() 메서드를 활용

```
# 미분 연산
print(b.grad)

# 기울기 초기화
b.grad.zero_()
print(b.grad)

# tensor([3., 3.])
# tensor([0., 0.])
```

신경망 구성 방법

- 대표적으로 두 가지 방법이 존재

  - nn.Sequential: 간단한 모델이나 레이어의 연속적인 시퀀스를 빠르게 정의하는 데 유용
    - 순차적인 구조만 표현 가능

  - nn.Module: 복잡한 모델을 클래스를 통해 유연하게 구현 가능
    - 만드는 과정이 복잡하며 디버깅이 어려움

nn.Sequential

- 순차적인 연산을 간결하게 표현하기 위한 클래스
- 레이어나 모듈을 순서대로 나열하여 간편하게 신경망 구성

- 복잡한 연결이나 제어 흐름이 필요한 모델에는 nn.Sequential 사용이 제한적

```python
import torch.nn as nn

model = nn.Sequential(
    nn.Conv2d(1, 32, 3, 1),
    nn.ReLU(),
    nn.MaxPool2d(2),
    nn.Flatten(),
    nn.Linear(32 * 13 * 13, 128),
    nn.ReLU(),
    nn.Linear(128, 10)
)
```

신경망 모듈(nn.Module)

- PyTorch의 기본 모듈로 모든 신경망의 기본 클래스
- nn.Module을 상속받아 사용자 정의 신경망 레이어나 모델 정의

- 사용자 정의 신경망 구성 시 기본 구조
  - __init__(): 기본 레이어 정의
  - Forward(): 순전파 과정 정의

```python
class CustomModel(nn.Module):
    def __init__(self):
        super(CustomModel, self).__init__()
        self.conv = nn.Conv2d(7, 64, kernel_size=3, stride=1, padding=1)
        self.fc = nn.Linear(64, 10)
        self.relu = nn.ReLU()
        self.bn = nn.BatchNorm2d(64)

    def forward(self, x):
        x = self.conv(x)
        x = self.bn(x)
        x = self.relu(x)
        x = x.view(x.size(0), -1)
        x = self.fc(x)
        return x
```

신경망 모듈(nn.Module)

- __init__: 메서드 내에서 신경망 레이어 정의
  - 각 모델의 레이어(conv, lstm, attention 등)
  - 활성화 함수 및 정규화(ReLU, BatchNorm)

```
class CustomModel(nn.Module):
    def __init__(self):
        super(CustomModel, self).__init__()
        self.conv = nn.Conv2d(7, 64, kernel_size=3, stride=1, padding=1)
        self.fc = nn.Linear(64, 10)
        self.relu = nn.ReLU()
        self.bn = nn.BatchNorm2d(64)
```

신경망 모듈(nn.Module)

- Forward: 메서드를 통해 모델의 순전파 정의

```
def forward(self, x):
    x = self.conv(x)
    x = self.bn(x)
    x = self.relu(x)
    x = x.view(x.size(0), -1)
    x = self.fc(x)
    return x
```

신경망 모듈(nn.Module)

- 모델 초기화 및 호출 방법
  - 모델을 인스턴스로 생성한 뒤 입력데이터로 호출
  - Print() 함수를 통해 모델 구조 파악 가능

```
model = CustomModel()
print(model)

# CustomModel(
#   (conv): Conv2d(7, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
#   (fc): Linear(in_features=64, out_features=10, bias=True)
#   (relu): ReLU()
#   (bn): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=Tru
e)
# )
```

신경망 모듈(nn.Module)

- model.train(): 학습 모드 전환
- model.eval(): 평가 모드 전환
- Model. save_state_dict(): 가중치 저장

```
# 학습 모드
model.train()

# 평가 모드
model.eval()

# 모델 저장
model.save_state_dict()
```

손실함수(Loss Functions)

- 모델의 예측과 실제 값 사이의 차이를 측정
- 이 차이를 최소화하는 방향으로 모델을 학습
- Pytorch의 손실함수는 torch.nn API에서 불러올 수 있음

Loss Functions

`nn.L1Loss`	Creates a criterion that measures the mean absolute error (MAE) between each element in the input x and target y.
`nn.MSELoss`	Creates a criterion that measures the mean squared error (squared L2 norm) between each element in the input x and target y.
`nn.CrossEntropyLoss`	This criterion computes the cross entropy loss between input logits and target.

출처 : https://pytorch.org/docs/stable/nn.html#loss-functions

옵티마이저(Optimizer)

- 손실을 최소화하기 위해 모델의 가중치를 업데이트하는 알고리즘
- torch.optim에서 불러올 수 있음

`Adadelta`	Implements Adadelta algorithm.
`Adagrad`	Implements Adagrad algorithm.
`Adam`	Implements Adam algorithm.
`AdamW`	Implements AdamW algorithm.

출처 : https://pytorch.org/docs/stable/optim.html

옵티마이저(Optimizer)

- 옵티마이저 내의 다양한 하이퍼파라미터는 딕셔너리로 설정할 수 있음

```
from torch.optim import SGD

optim = SGD([{'params': model.parameters(),
              'lr': 1e-3} ],
            lr=1e-2,
            momentum=0.9)
```

옵티마이저(Optimizer)

- 옵티마이저의 메서드를 통해 학습 과정의 두 스텝을 조정할 수 있음
  - optimizer.zero_grad(): 그래디언트 초기화
  - optimizer.step(): 가중치 업데이트

```
model.train()
for input, target in dataset:
    optim.zero_grad()
    output = model(input)
    loss = loss_fn(output, target)
    loss.backward()
    optim.step()
```

Dataset

- PyTorch의 Dataset 클래스
- 데이터 로딩을 위한 기본 인터페이스 제공
- 사용자 정의 데이터셋을 만들기 위해 Dataset을 상속받아 구현
- 다양한 데이터 소스와 형식에 유연하게 대응 가능

```python
from torch.utils.data import Dataset

class CustomDataset(Dataset):
    def __init__(self, data, labels):
        self.data = data
        self.labels = labels

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

    def __getitem__(self, idx):
        sample_data = self.data[idx]
        sample_label = self.labels[idx]
        return sample_data, sample_label
```

Dataset

- 기본 메서드
  - __len__: 데이터셋의 총 데이터 수 반환

```
def __len__(self):
    return len(self.data)
```

  - __getitem__: 주어진 인덱스에 해당하는 데이터 반환

```
def __getitem__(self, idx):
    sample_data = self.data[idx]
    sample_label = self.labels[idx]
    return sample_data, sample_label
```

DataLoader

- Dataset에서 데이터를 배치 단위로 효율적으로 로드하는 역할
- 멀티 프로세싱을 활용하여 데이터 로딩 속도 향상

```
from torch.utils.data import DataLoader

dataset = CustomDataset(data, labels)
dataloader = DataLoader(dataset, batch_size=32, shuffle=True, num_workers=4)
```

DataLoader

- 반복자(iterator)로 구현되어 있어, for문을 통해 배치 단위로 데이터 접근 가능

```
for batch_data, batch_labels in dataloader:
    # 학습 또는 평가 코드
    model.train()
    for input, target in dataset:
            optim.zero_grad()
            output = model(input)
            loss = loss_fn(output, target)
            loss.backward()
            optim.step()
```

- Sampler
  - 데이터 셔플링 전략을 지정
  - Ex) RandomSampler, SequentialSampler

```
dataloader = DataLoader(dataset, batch_size=32, shuffle=True, num_workers=4,
                        sampler=RandomSampler)
```

03

Pytorch로 구현하는 간단한 LSTM

신경망 학습 방법

- 신경망 학습은 일반적으로 아래의 단계를 포함

1. 데이터 준비
   - 데이터셋 로딩 및 전처리
   - Dataset 및 DataLoader를 사용하여 배치 단위로 데이터 로드
2. 모델 정의
   - nn.Module을 상속받아 신경망 구조 정의
   - 레이어, 활성화 함수, 정규화 방법 등을 포함

신경망 학습 방법

- 신경망 학습은 일반적으로 아래의 단계를 포함

3. 손실 함수 선택
   - 예측값과 실제값의 차이를 측정하는 함수
   - Task에 맞는 손실함수 선택(Cross entropy, MSE 등)
4. Optimizer 선택
   - 모델의 가중치를 업데이트하는 알고리즘
   - Ex) SGD, Adam, RMSprop

신경망 학습 방법

- 신경망 학습은 일반적으로 아래의 단계를 포함

5. 학습 반복
    - 순전파 (Forward Propagation): 입력 데이터를 모델에 전달하여 예측값 생성
    - 손실 계산: 예측값과 실제값을 사용하여 손실 계산
    - 역전파(Backward Propagation)
        - 손실에 대한 그래디언트 계산
        - Optimizer를 사용하여 모델의 가중치 업데이트

신경망 학습 방법

- 신경망 학습은 일반적으로 아래의 단계를 포함

6. 평가
   - 검증 데이터셋 또는 테스트 데이터셋을 사용하여 모델 성능 평가
   - 모델의 일반화 성능 확인
7. 모델 저장 및 로드
   - 학습된 모델의 가중치와 구조 저장
   - 필요 시 저장된 모델을 불러와 재사용

03 Pytorch로 구현하는 간단한 LSTM

PyTorch를 사용한 LSTM 모델 구현

- 기본적인 LSTM 모델을 PyTorch로 구현하는 방법

- 필요한 라이브러리 불러오기

```
import torch
import torch.nn as nn
from torch.utils.data import Dataset, DataLoader
```

03 Pytorch로 구현하는 간단한 LSTM

PyTorch를 사용한 LSTM 모델 구현

- 데이터셋 클래스 정의

```
# 1. 데이터셋 정의 및 데이터 준비
class CustomDataset(Dataset):
    def __init__(self, data, labels):
        self.data = data
        self.labels = labels

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

    def __getitem__(self, idx):
        return self.data[idx], self.labels[idx]
```

03 Pytorch로 구현하는 간단한 LSTM

PyTorch를 사용한 LSTM 모델 구현

- 가상의 데이터 생성
- 데이터셋, 데이터로더 생성

```
# 가상의 데이터 생성
data = torch.randn(100, 10, 10)  # 100개의 시퀀스, 각 시퀀스는 10개의 스텝, 각 스텝은 10차원
labels = torch.randn(100, 1)

# 가상의 테스트 데이터 생성
test_data = torch.randn(30, 10, 10)
test_labels = torch.randn(30, 1)

dataset = CustomDataset(data, labels)
dataloader = DataLoader(dataset, batch_size=32, shuffle=True)

test_dataset = CustomDataset(test_data, test_labels)
test_dataloader = DataLoader(test_dataset, batch_size=32)
```

03 Pytorch로 구현하는 간단한 LSTM

PyTorch를 사용한 LSTM 모델 구현

- 간단한 LSTM 모델 정의

```python
# 2. LSTM 모델 정의
class LSTMModel(nn.Module):
    def __init__(self, input_dim, hidden_dim, num_layers, output_dim):
        super(LSTMModel, self).__init__()
        self.hidden_dim = hidden_dim
        self.lstm = nn.LSTM(input_dim, hidden_dim, num_layers, batch_first=True)
        self.linear = nn.Linear(hidden_dim, output_dim)

    def forward(self, x):
        h0 = torch.zeros(num_layers, x.size(0), self.hidden_dim).requires_grad_()
        c0 = torch.zeros(num_layers, x.size(0), self.hidden_dim).requires_grad_()
        out, (hn, cn) = self.lstm(x, (h0.detach(), c0.detach()))
        out = self.linear(out[:, -1, :])
        return out
```