PyTorch MNIST

최석재 *lingua@naver.com*

파이토치 특징

- 2016년 Facebook AI(現 Meta AI) 연구팀에서 발표했다

- 장점
  - Keras와 달리 파이토치는 단일한 구현 방법을 제시한다
  - 저수준 API를 사용하기 때문에 코드의 자유도가 높고, 딥러닝 프로세스의 많은 부분을 직접 제어할 수 있다
  - 따라서 자유도가 높아 연구용으로 많이 사용되었고, 현재는 산업계에서도 많이 사용한다
  - 새로운 모델 개발은 주로 파이토치로 이루어지고 있어 Keras도 Model Subclassing을 내놓으며 파이토치를 따라갔다

- 단점
  - Keras의 Model Subclassing 처럼 저수준 API이므로 초기 배우는 과정이 다소 어렵다
  - CPU – GPU 설정을 모델과 데이터에 대하여 잘 해주어야 한다
  - 안정화가 진행 중이다
    - 명확하지 않은 오류가 발생하는 경우가 있다. 다만, 2023년 5월에 안정화 버전 2.0이 나왔다
    - 최근 지속적으로 늘고 있으나, 관련 서적이나 매뉴얼이 아직 충분하지 않다
    - 전처리 도구가 파이토치의 버전에 잘 대응하지 못한다 (예: gluon)
  - 큰 차이는 없지만 Keras에 비하여 속도가 약간 느리다
  - 아직 산업계에서는 Keras를 선호하므로 두 프레임워크를 모두 알고 있어야 한다

이미지 처리 연습

- 여기서는 다음의 사항을 중심으로 연습하며 이미지 처리를 수행한다

1. Dataset와 DataLoader의 사용
2. transforms를 이용한 이미지 전처리
3. 미니배치 학습 방법
4. 은닉층 설계
5. GPU를 이용하기 위하여 데이터와 모델에 GPU 연산 적용

Pytorch DATASETS

- 파이토치는 연습용 데이터셋을 다수 보유하고 있음
- https://pytorch.org/vision/stable/datasets.html

- 여기서는 그중 대표적인 MNIST 데이터셋을 사용하면서
- 이미지 전처리에 자주 사용되는 torchvision 라이브러리도 연습해본다

Image classification

`Caltech101`(root[, target_type, transform, ...])	Caltech 101 Dataset.
`Caltech256`(root[, transform, ...])	Caltech 256 Dataset.
`CelebA`(root[, split, target_type, ...])	Large-scale CelebFaces Attributes (CelebA) Dataset Dataset.
`CIFAR10`(root[, train, transform, ...])	CIFAR10 Dataset.

Dataset과 DataLoader

- Dataset과 DataLoader는 파이토치에서 데이터처리를 쉽게 하도록 돕는 도구
- Dataset는 독립변수와 종속변수를 묶어주는 기능이,
- DataLoader는 미니배치를 만드는 기능이 대표적이다

- 원 데이터를 먼저 Dataset으로 만들고, 이를 다시 DataLoader에 넣는 방식을 취한다

Dataset, DataLoader 만들기

- 파이토치에서 사용할 데이터는 다음과 같이 먼저 토치 텐서로 만들고,
- 이를 학습에 용이하게 쓰이도록 Dataset과 DataLoader로 만드는 과정을 거친다

- from torch.utils.data import TensorDataset
- from torch.utils.data import DataLoader

- x_train = torch.tensor(x_train).float()
- y_train = torch.tensor(y_train).float()

- train_dataset = TensorDataset(x_train, y_train)

- train_loader = DataLoader(train_dataset, batch_size=32, shuffle=True)

※ 여기에서는 미리 만들어진 MNIST Dataset을 이용한다

구글 드라이브와 연결

```
# from google.colab import auth
# auth.authenticate_user()
```

- from google.colab import drive
- drive.mount('/content/gdrive')

GPU 확인

- import torch

- device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
- print(device)

```
cuda:0
```

※ Colab에서는 런타임 유형을 GPU로 변경한다

MNIST Dataset Download 1

- import torchvision.datasets as datasets

- DATA_PATH = '/content/gdrive/MyDrive/pytest/datasets/'
- train_set = datasets.MNIST(root = DATA_PATH, train = True, download = True)
- print("data size:", len(train_set)) # 60000

train=True는 학습데이터,
train=False는 테스트데이터 다운로드

```
Downloading http://yann.lecun.com/exdb/mnist/train-images-idx3-ubyte.gz
Downloading http://yann.lecun.com/exdb/mnist/train-images-idx3-ubyte.gz to /content/gdrive/MyDrive/pytest/MNIST/raw/train-images-idx3-ubyte.gz
100% 9912422/9912422 [00:00<00:00, 9139877.77it/s]
Extracting /content/gdrive/MyDrive/pytest/MNIST/raw/train-images-idx3-ubyte.gz to /content/gdrive/MyDrive/pytest/MNIST/raw

Downloading http://yann.lecun.com/exdb/mnist/train-labels-idx1-ubyte.gz
Downloading http://yann.lecun.com/exdb/mnist/train-labels-idx1-ubyte.gz to /content/gdrive/MyDrive/pytest/MNIST/raw/train-labels-idx1-ubyte.gz
100% 28881/28881 [00:00<00:00, 732459.96it/s]
Extracting /content/gdrive/MyDrive/pytest/MNIST/raw/train-labels-idx1-ubyte.gz to /content/gdrive/MyDrive/pytest/MNIST/raw

Downloading http://yann.lecun.com/exdb/mnist/t10k-images-idx3-ubyte.gz
Downloading http://yann.lecun.com/exdb/mnist/t10k-images-idx3-ubyte.gz to /content/gdrive/MyDrive/pytest/MNIST/raw/t10k-images-idx3-ubyte.gz
100% 1648877/1648877 [00:00<00:00, 17680376.41it/s]
Extracting /content/gdrive/MyDrive/pytest/MNIST/raw/t10k-images-idx3-ubyte.gz to /content/gdrive/MyDrive/pytest/MNIST/raw

Downloading http://yann.lecun.com/exdb/mnist/t10k-labels-idx1-ubyte.gz
Downloading http://yann.lecun.com/exdb/mnist/t10k-labels-idx1-ubyte.gz to /content/gdrive/MyDrive/pytest/MNIST/raw/t10k-labels-idx1-ubyte.gz
100% 4542/4542 [00:00<00:00, 39051.97it/s]
Extracting /content/gdrive/MyDrive/pytest/MNIST/raw/t10k-labels-idx1-ubyte.gz to /content/gdrive/MyDrive/pytest/MNIST/raw
```

MNIST

CLASS torchvision.datasets.MNIST(*root: str, train: bool = True, transform: Optional[Callable] = None, target_transform: Optional[Callable] = None, download: bool = False*) [SOURCE]

MNIST Dataset.

Parameters:

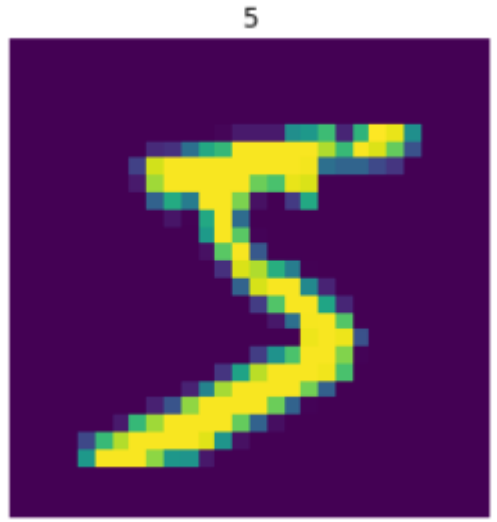
- **root** (*string*) – Root directory of dataset where `MNIST/raw/train-images-idx3-ubyte` and `MNIST/raw/t10k-images-idx3-ubyte` exist.
- **train** (*bool, optional*) – If True, creates dataset from `train-images-idx3-ubyte`, otherwise from `t10k-images-idx3-ubyte`.
- **download** (*bool, optional*) – If True, downloads the dataset from the internet and puts it in root directory. If dataset is already downloaded, it is not downloaded again.
- **transform** (*callable, optional*) – A function/transform that takes in an PIL image and returns a transformed version. E.g, `transforms.RandomCrop`
- **target_transform** (*callable, optional*) – A function/transform that takes in the target and transforms it.

https://pytorch.org/vision/stable/generated/torchvision.datasets.MNIST.html

데이터 확인

- import matplotlib.pyplot as plt
- train_images, train_labels = train_set[0]

- plt.title(train_labels)
- plt.imshow(train_images)
- plt.axis('off')
- plt.show()



데이터 타입 확인

- print("데이터의 타입:", type(train_images))
- print("데이터 라벨:", type(train_labels))

```
데이터의 타입: <class 'PIL.Image.Image'>
데이터 라벨: <class 'int'>
```

transforms 설정 1

- 파이토치에서 이미지 전처리를 쉽게 하는 transforms를 설정한다
- 데이터를 텐서 형태로 변환한다

- import torchvision.transforms as transforms
- transform = transforms.Compose([transforms.ToTensor()])

https://pytorch.org/vision/0.9/transforms.html#

MNIST Dataset Download 2

- MNIST 데이터셋은 transforms 설정을 아규먼트로 받는다
- transforms 설정과 함께 MNIST 데이터셋을 다시 다운로드 받는다
- 이렇게 받은 데이터는 텐서 형태로서 imshow() 로 이미지 출력은 할 수 없다

- train_set = datasets.MNIST(root = DATA_PATH, train = True, download = True, transform = transform)
- print("data size:", len(train_set)) # 60000

데이터 확인

- train_images, train_labels = train_set[0]
- print("데이터의 타입:", type(train_images))
- print("데이터 라벨:", type(train_labels))

60,000 개 데이터 중 첫 번째 하나만 가져옴

```
데이터의 타입: <class 'torch.Tensor'>
데이터 라벨: <class 'int'>
```

PIL 타입에서 Tensor 타입으로 변경되었음

- print("데이터 shape:", train_images.shape)
- print("데이터 차원:", train_images.dim())
- print("데이터 최솟값:", train_images.min())
- print("데이터 최댓값:", train_images.max())

하나의 데이터가 (28 x 28) 개의 원소로 구성되어 있음

```
데이터 shape: torch.Size([1, 28, 28])
데이터 차원: 3
데이터 최솟값: tensor(0.)
데이터 최댓값: tensor(1.)
```

transforms 설정 2

- 현재 데이터는 0~1 사이의 범위 내에 있다
- 이 범위를 변경하려면 Normalize(평균, 표준오차) 를 사용한다
- Normalize(0.5, 0.5)를 하면 -1~1 사이의 범위를 갖고, 0~1 사이를 갖게 하려면 Normalize(0, 1)로 한다
  - (개별값 – 평균) / 표준편차
  - (0 – 0.5) / 0.5 = -1, (1 – 0.5) / 0.5 = 1

- 현재 데이터는 3차원(channels, height, width)이나, Linear 층을 이용하려면 1차원으로 변경해야 한다
- view()를 이용하여 shape를 변경할 수 있다
- transforms 설정에서 view()를 Lambda()와 연계하여 각 데이터의 shape를 변경한다

- import torchvision.transforms as transforms
- transform = transforms.Compose([transforms.ToTensor(),
  transforms.Normalize(0.5, 0.5),
  transforms.Lambda(lambda x: x.view(-1))])

1차원으로 변경

MNIST Dataset Download 3

- 변경된 transform으로 다시 다운로드한다

- train_set = datasets.MNIST(root = DATA_PATH, train = True, download = True, transform = transform)
- print("data size:", len(train_set)) # 60000

데이터 확인

- train_images, train_labels = train_set[0]
- print("데이터의 타입:", type(train_images))
- print("데이터 라벨:", type(train_labels))

- print("데이터 shape:", train_images.shape)
- print("데이터 차원:", train_images.dim())
- print("데이터 최솟값:", train_images.min())
- print("데이터 최댓값:", train_images.max())

60,000 개 데이터 중 첫 번째 하나만 가져옴

```
데이터의 타입: <class 'torch.Tensor'>
데이터 라벨: <class 'int'>
데이터 shape: torch.Size([784])
데이터 차원: 1
데이터 최솟값: tensor(-1.)
데이터 최댓값: tensor(1.)
```

- print(train_images)
- print(train_labels)

```
        -1.0000, -1.0000, -1.0000, -1.0000, -1.0000, -1.0000, -0.6392,  0.0196,
         0.4353,  0.9843,  0.9843,  0.6235, -0.9843, -1.0000, -1.0000, -1.0000,
        -1.0000, -1.0000, -1.0000, -1.0000, -1.0000, -1.0000, -1.0000, -1.0000,
        -1.0000, -1.0000, -1.0000, -1.0000, -1.0000, -1.0000, -1.0000, -1.0000,
        -0.6941,  0.1608,  0.7961,  0.9843,  0.9843,  0.9843,  0.9608,  0.4275,
        -1.0000, -1.0000, -1.0000, -1.0000, -1.0000, -1.0000, -1.0000, -1.0000,
        -1.0000, -1.0000, -1.0000, -1.0000, -1.0000, -1.0000, -1.0000, -1.0000,
        -1.0000, -1.0000, -0.8118, -0.1059,  0.7333,  0.9843,  0.9843,  0.9843,
         0.9843,  0.5765, -0.3882, -1.0000, -1.0000, -1.0000, -1.0000, -1.0000,
        -1.0000, -1.0000, -1.0000, -1.0000, -1.0000, -1.0000, -1.0000, -1.0000,
        -1.0000, -1.0000, -1.0000, -1.0000, -0.8196, -0.4824,  0.6706,  0.9843,
         0.9843,  0.9843,  0.9843,  0.5529, -0.3647, -0.9843, -1.0000, -1.0000,
        -1.0000, -1.0000, -1.0000, -1.0000, -1.0000, -1.0000, -1.0000, -1.0000,
        -1.0000, -1.0000, -1.0000, -1.0000, -1.0000, -1.0000, -0.8588,  0.3412,
         0.7176,  0.9843,  0.9843,  0.9843,  0.9843,  0.5294, -0.3725, -0.9294,
        -1.0000, -1.0000, -1.0000, -1.0000, -1.0000, -1.0000, -1.0000, -1.0000,
        -1.0000, -1.0000, -1.0000, -1.0000, -1.0000, -1.0000, -1.0000, -1.0000,
        -0.5686,  0.3490,  0.7725,  0.9843,  0.9843,  0.9843,  0.9843,  0.9137,
         0.0431, -0.9137, -1.0000, -1.0000, -1.0000, -1.0000, -1.0000, -1.0000,
        -1.0000, -1.0000, -1.0000, -1.0000, -1.0000, -1.0000, -1.0000, -1.0000,
        -1.0000, -1.0000, -1.0000, -1.0000,  0.0667,  0.9843,  0.9843,  0.9843,
         0.6627,  0.0588,  0.0353, -0.8745, -1.0000, -1.0000, -1.0000, -1.0000,
        -1.0000, -1.0000, -1.0000, -1.0000, -1.0000, -1.0000, -1.0000, -1.0000,
        -1.0000, -1.0000, -1.0000, -1.0000, -1.0000, -1.0000, -1.0000, -1.0000,
        -1.0000, -1.0000, -1.0000, -1.0000, -1.0000, -1.0000, -1.0000, -1.0000,
        -1.0000, -1.0000, -1.0000, -1.0000, -1.0000, -1.0000, -1.0000, -1.0000,
        -1.0000, -1.0000, -1.0000, -1.0000, -1.0000, -1.0000, -1.0000, -1.0000,
        -1.0000, -1.0000, -1.0000, -1.0000, -1.0000, -1.0000, -1.0000, -1.0000,
        -1.0000, -1.0000, -1.0000, -1.0000, -1.0000, -1.0000, -1.0000, -1.0000,
        -1.0000, -1.0000, -1.0000, -1.0000, -1.0000, -1.0000, -1.0000, -1.0000,
        -1.0000, -1.0000, -1.0000, -1.0000, -1.0000, -1.0000, -1.0000, -1.0000,
        -1.0000, -1.0000, -1.0000, -1.0000, -1.0000, -1.0000, -1.0000, -1.0000,
        -1.0000, -1.0000, -1.0000, -1.0000, -1.0000, -1.0000, -1.0000, -1.0000,
        -1.0000, -1.0000, -1.0000, -1.0000, -1.0000, -1.0000, -1.0000, -1.0000]])
5
```

테스트 데이터셋 다운로드

- 같은 방식으로 테스트 데이터셋을 다운로드한다

- test_set = datasets.MNIST(root = DATA_PATH, train = False, download = True, transform = transform)
- print("data size:", len(test_set)) # 10000

미니 배치 만들기

- 데이터를 쪼개어 학습하게 하는 미니 배치 학습을 연습한다
- 미니 배치를 지원하는 DataLoader 클래스를 사용한다
- DataLoader의 batch_size 아규먼트에 값을 지정하는 것으로 가능하다

- from torch.utils.data import DataLoader

- batch_size = 256

데이터로더 생성
- train_loader = DataLoader(train_set, batch_size = batch_size, shuffle = True)
- test_loader = DataLoader(test_set, batch_size = batch_size, shuffle = False)
- print("학습 데이터 배치 개수:", len(train_loader)) # 235

※ 검증 데이터는 shuffle이 필요하지 않다

미니배치 데이터 확인

- for images, labels in train_loader:
  break

- print(images.shape)
- print(labels.shape)
- print(images)
- print(labels)

```
torch.Size([256, 784])
torch.Size([256])
tensor([[-1., -1., -1.,  ..., -1., -1., -1.],
        [-1., -1., -1.,  ..., -1., -1., -1.],
        [-1., -1., -1.,  ..., -1., -1., -1.],
        ...,
        [-1., -1., -1.,  ..., -1., -1., -1.],
        [-1., -1., -1.,  ..., -1., -1., -1.],
        [-1., -1., -1.,  ..., -1., -1., -1.]])
tensor([9, 0, 5, 2, 6, 4, 7, 7, 4, 8, 1, 7, 1, 8, 8, 6, 5, 2, 9, 9, 1, 2, 0, 4,
        4, 3, 1, 0, 4, 5, 0, 8, 8, 3, 4, 4, 8, 9, 5, 0, 6, 0, 7, 3, 9, 4, 5, 8,
        3, 3, 4, 6, 4, 3, 3, 5, 9, 0, 5, 9, 7, 0, 3, 4, 7, 6, 3, 9, 5, 8, 7, 3,
        3, 1, 5, 5, 7, 6, 6, 8, 3, 0, 1, 3, 9, 7, 4, 1, 2, 9, 2, 0, 1, 2, 5, 9,
        9, 9, 1, 2, 7, 4, 0, 7, 5, 5, 3, 1, 5, 6, 8, 0, 2, 8, 2, 8, 8, 8, 7, 2,
        9, 0, 0, 3, 6, 7, 6, 6, 2, 4, 9, 6, 2, 6, 2, 0, 8, 4, 6, 8, 3, 7, 5, 5,
        8, 2, 8, 5, 1, 4, 4, 5, 8, 4, 4, 2, 2, 4, 4, 8, 3, 8, 0, 7, 0, 2, 1, 5,
        4, 6, 5, 3, 1, 1, 0, 3, 4, 9, 6, 7, 5, 5, 7, 9, 4, 8, 3, 2, 6, 0, 5, 5,
        3, 6, 8, 5, 4, 8, 9, 7, 1, 7, 0, 5, 1, 3, 9, 1, 2, 7, 2, 7, 2, 8, 7, 0,
        1, 7, 9, 5, 2, 0, 3, 3, 9, 8, 9, 0, 7, 5, 2, 7, 0, 0, 3, 2, 1, 7, 8, 7,
        3, 9, 0, 3, 3, 3, 2, 5, 1, 5, 2, 9, 8, 5, 0, 2])
```

데이터 확인

- plt.figure(figsize=(8, 2))
- for i in range(20):

```
    ax = plt.subplot(2, 10, i + 1)

    # 넘파이 배열로 변환
    image = images[i].numpy()
    label = labels[i]

    # 이미지 출력
    plt.imshow(image.reshape(28, 28))
    ax.get_xaxis().set_visible(False)
    ax.get_yaxis().set_visible(False)
```

- plt.show()

입력과 출력의 차원수

- input_size = image.shape[0]
- output_size = len(set(labels.numpy())) # 파이썬의 set()은 유니크한 값을 출력한다

- print("입력 차원수:", input_size)
- print("출력 차원수:", output_size)

```
입력 차원수: 784
출력 차원수: 10
```

모델 정의

- import torch.nn as nn
- class Net(nn.Module):
- def __init__(self, input_size, output_size, hidden_size):
  super().__init__()

  # 은닉층 정의
  self.l1 = nn.Linear(input_size, hidden_size) # 은닉층의 출력은 다음 층의 입력이 된다

  # 출력층 정의
  self.l2 = nn.Linear(hidden_size, output_size)

  # ReLU 함수 정의
  self.relu = nn.ReLU(inplace=True) # inplace 옵션이 있는 경우는 True하는 것이 권장된다. 입력값을 직접 변경하여 메모리 사용량을 줄인다

- def forward(self, x):
  x1 = self.l1(x)
  x2 = self.relu(x1) # 첫번째 은닉층 다음에 활성화함수를 사용했다. 뒤의 손실함수가 softmax를 포함할 것이므로 출력층 다음에 활성화함수는 사용하지 않는다
  y = self.l2(x2)

  return y

모델 객체 생성

난수 고정. 딥러닝 네트워크의 가중치 난수 발생은 완전히 제어할 수는 없다
- torch.manual_seed(111)
- torch.cuda.manual_seed(111)

※ 데이터, 모델, 손실함수에 .to(device) 구문을 적용한다

모델 객체 생성
- hidden_size = 128
- net = Net(input_size, output_size, hidden_size)

모델을 GPU로 전송. GPU를 사용할 때는 이 과정을 추가해야 한다
- net = net.to(device)

모델 개요
- print(net)

```
Net(
  (l1): Linear(in_features=784, out_features=128, bias=True)
  (l2): Linear(in_features=128, out_features=10, bias=True)
  (relu): ReLU(inplace=True)
)
```

TEST

※ 계산 그래프를 그릴 때는 optimizer 를 만들지 않아도 된다

optimizer = torch.optim.SGD(net.parameters(), lr=0.01)

최적화 알고리즘: 경사 하강법

- criterion = nn.CrossEntropyLoss().to(device)

손실함수. 교차 엔트로피 함수

손실함수를 GPU로 전송

훈련 데이터셋의 처음 항목을 추출

- for images, labels in train_loader:
  break

데이터로더에서 받은 데이터를 GPU로 전송

- X_train = images.to(device)
- y_train = labels.to(device)

- pred_train = net(X_train)

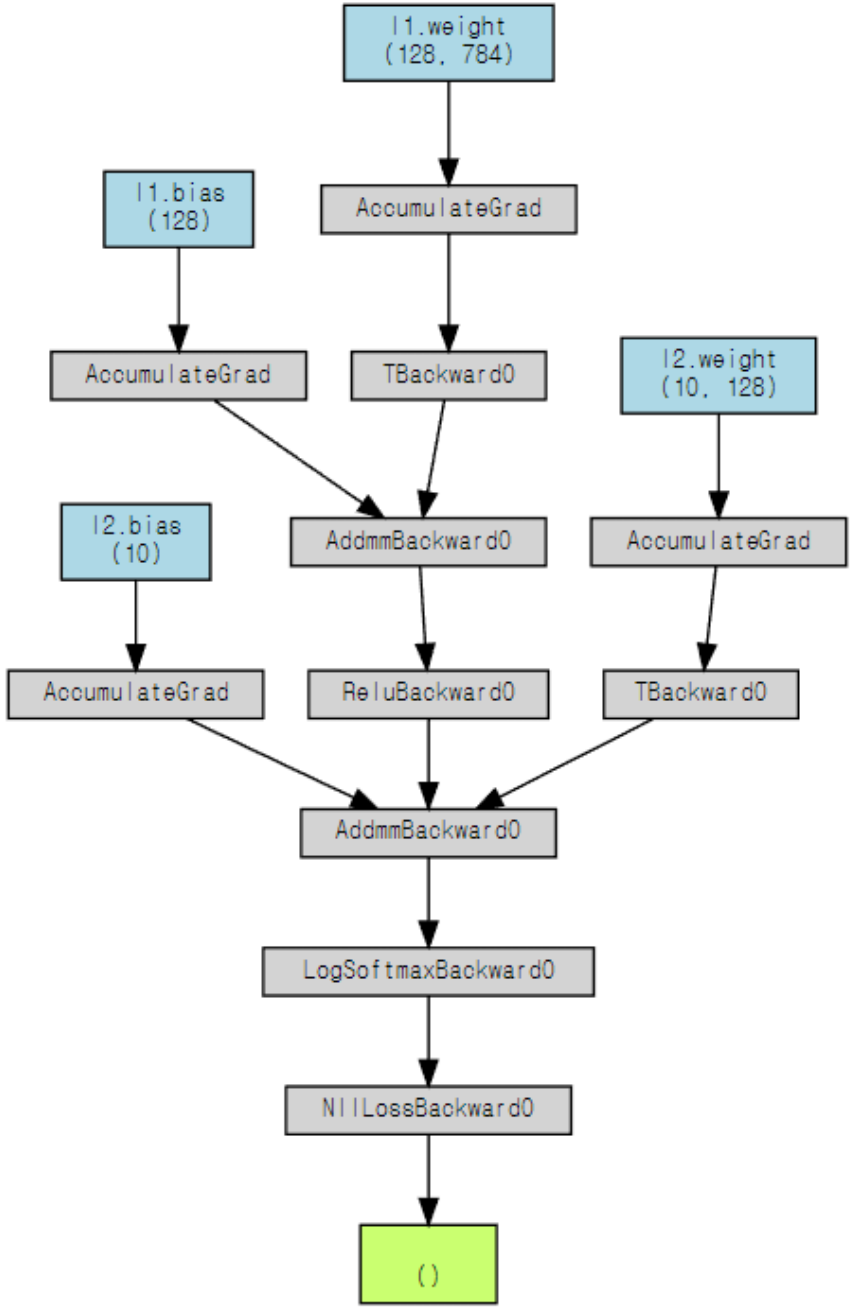
예측 계산

- loss = criterion(pred_train, y_train)

손실 계산

손실값 생성까지의 계산 그래프

- !pip install torchviz
- from torchviz import make_dot
- g = make_dot(loss, params=dict(net.named_parameters()))
- display(g)



변수 초기화

- import numpy as np
- import torch.optim as optim

학습률
- lr = 0.01

손실 함수 : 교차 엔트로피 함수. softmax를 포함한다
- criterion = nn.CrossEntropyLoss().to(device)

최적화 함수 : 경사 하강법
- optimizer = optim.SGD(net.parameters(), lr=lr)

※ optimizer는 모델을 따라 자동으로 GPU로 전송된다

반복 횟수
- num_epochs = 100

평가 결과 기록
- history = np.zeros((0,5))

모델 훈련 (train)

```
from tqdm.notebook import tqdm

# 미니배치를 이용한 훈련
for epoch in range(num_epochs):
  train_acc, train_loss = 0, 0                          # 훈련데이터 정확도, 손실값 초기화
  test_acc, test_loss = 0, 0                            # 테스트데이터 정확도, 손실값 초기화
  n_train, n_test = 0, 0                                # 진행 정도 초기화

  # 훈련 데이터 학습 과정
  for X_train, y_train in tqdm(train_loader):           # 데이터로더에서 데이터를 하나씩 가져온다
    n_train += len(y_train)                             # 훈련량 기록

    # 추출한 데이터를 GPU로 전송
    X_train = X_train.to(device)
    y_train = y_train.to(device)

    pred_train = net(X_train)                           # 예측 계산
    loss_train = criterion(pred_train, y_train)         # 손실 계산. softmax 포함
    optimizer.zero_grad()                               # 새로운 배치마다 기울기 초기화
    loss_train.backward()                               # 기울기 계산
    optimizer.step()                                    # 파라미터 수정
    result_train = torch.max(pred_train, 1)[1]          # 예측 라벨 산출

    # 훈련데이터 평가
    train_loss += loss_train.item()                     # 손실값 추출 및 누적
    train_acc += (result_train == y_train).sum().item() # 정확도 계산 및 누적
```

모델 훈련 (test)

```
# 평가 데이터 예측. 기울기 계산과 관련된 부분은 없다
for X_test, y_test in test_loader:
  n_test += len(y_test)                                  # 평가량 기록

  # 추출한 데이터를 GPU로 전송
  X_test = X_test.to(device)
  y_test = y_test.to(device)

  pred_test = net(X_test)                                # 예측 계산
  loss_test = criterion(pred_test, y_test)               # 손실 계산
  result_test = torch.max(pred_test, 1)[1]               # 예측 라벨 산출

  # 테스트데이터 평가
  test_loss += loss_test.item()                          # 손실값 추출 및 누적
  test_acc += (result_test == y_test).sum().item()       # 정확도 계산 및 누적

# 평가 결과 산출, 기록
train_acc = train_acc / n_train
test_acc = test_acc / n_test
train_loss = train_loss * batch_size / n_train
test_loss = test_loss * batch_size / n_test
print (f'Epoch [{epoch+1}/{num_epochs}], loss: {train_loss:.5f} acc: {train_acc:.5f} val_loss: {test_loss:.5f}, val_acc: {test_acc:.5f}')
item = np.array([epoch+1 , train_loss, train_acc, test_loss, test_acc])
history = np.vstack((history, item))
```

```
100% 235/235 [00:19<00:00, 8.23it/s]
Epoch [1/10], loss: 1.48050 acc: 0.65292 val_loss: 0.85267, val_acc: 0.82180
100% 235/235 [00:14<00:00, 15.87it/s]
Epoch [2/10], loss: 0.67133 acc: 0.84253 val_loss: 0.53780, val_acc: 0.86890
100% 235/235 [00:11<00:00, 21.85it/s]
Epoch [3/10], loss: 0.49560 acc: 0.87138 val_loss: 0.43754, val_acc: 0.88430
100% 235/235 [00:11<00:00, 21.91it/s]
Epoch [4/10], loss: 0.42711 acc: 0.88397 val_loss: 0.39318, val_acc: 0.89180
100% 235/235 [00:11<00:00, 22.59it/s]
Epoch [5/10], loss: 0.39050 acc: 0.89217 val_loss: 0.36536, val_acc: 0.89960
100% 235/235 [00:12<00:00, 9.88it/s]
Epoch [6/10], loss: 0.36745 acc: 0.89753 val_loss: 0.34785, val_acc: 0.90160
100% 235/235 [00:11<00:00, 20.68it/s]
Epoch [7/10], loss: 0.35060 acc: 0.90000 val_loss: 0.33429, val_acc: 0.90800
100% 235/235 [00:11<00:00, 21.98it/s]
Epoch [8/10], loss: 0.33796 acc: 0.90340 val_loss: 0.32353, val_acc: 0.90730
100% 235/235 [00:11<00:00, 21.18it/s]
Epoch [9/10], loss: 0.32804 acc: 0.90625 val_loss: 0.31513, val_acc: 0.91060
100% 235/235 [00:11<00:00, 21.30it/s]
Epoch [10/10], loss: 0.31889 acc: 0.90868 val_loss: 0.30666, val_acc: 0.91570
```

결과 확인

- print(history)

epoch	train_loss	train_acc	test_loss	test_acc
[[1.	1.48050482	0.65291667	0.85266601	0.8218]
[2.	0.67133388	0.84253333	0.53779942	0.8689]
[3.	0.49559684	0.87138333	0.43753758	0.8843]
[4.	0.42710741	0.88396667	0.39318167	0.8918]
[5.	0.39050463	0.89216667	0.36535773	0.8996]
[6.	0.36744831	0.89753333	0.34784732	0.9016]
[7.	0.35059669	0.9	0.3342886	0.908]
[8.	0.33795846	0.9034	0.32352867	0.9073]
[9.	0.32803598	0.90625	0.31513011	0.9106]
[10.	0.31889012	0.90868333	0.30666401	0.9157]]

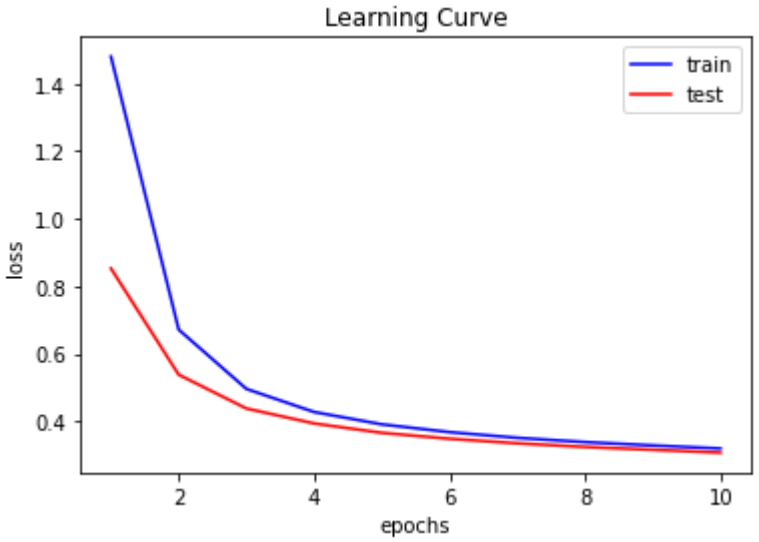
손실값, 정확도 확인

- print('Test Data 평가결과:')
- print(f'초기상태> 손실값: {history[0, 3]:.5f} 정확도: {history[0, 4]:.5f}')
- print(f'최종상태> 손실값: {history[-1, 3]:.5f} 정확도: {history[-1, 4]:.5f}')

```
Test Data 평가결과:
초기상태> 손실값: 0.85267 정확도: 0.82180
최종상태> 손실값: 0.30666 정확도: 0.91570
```

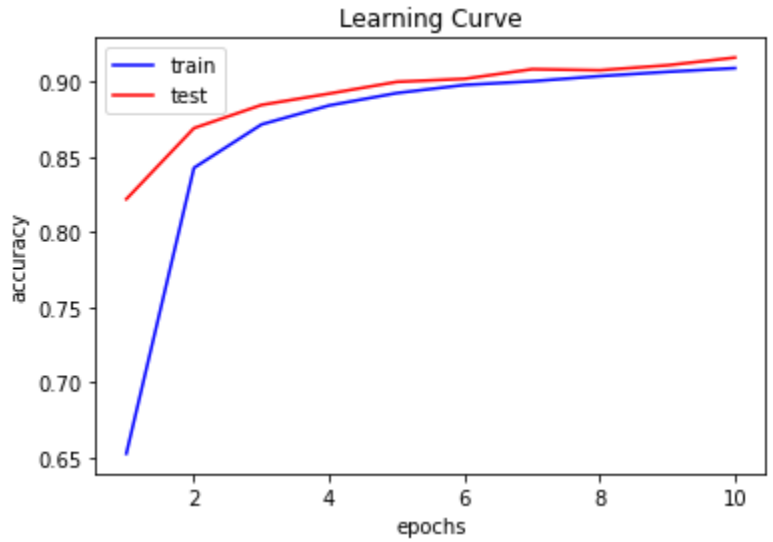
학습 곡선 (손실값)

- plt.plot(history[:,0], history[:,1], 'blue', label='train')
- plt.plot(history[:,0], history[:,3], 'red', label='test')
- plt.xlabel('epochs')
- plt.ylabel('loss')
- plt.title('Learning Curve')
- plt.legend()
- plt.show()



학습 곡선 (정확도)

- plt.plot(history[:,0], history[:,2], 'blue', label='train')
- plt.plot(history[:,0], history[:,4], 'red', label='test')
- plt.xlabel('epochs')
- plt.ylabel('accuracy')
- plt.title('Learning Curve')
- plt.legend()
- plt.show()



데이터 이미지로 확인

데이터로더에서 처음 한 세트 가져오기

- for images, labels in test_loader:
  break

예측 결과 가져오기

- inputs = images.to(device)
- labels = labels.to(device)
- outputs = net(inputs)
- predicted = torch.max(outputs, 1)[1] # [0]은 최댓값, [1]은 최댓값의 인덱스

데이터 이미지로 확인

- plt.figure(figsize=(8, 6))
- for i in range(50): # 처음 50개 이미지에 대하여 확인
  ax = plt.subplot(5, 10, i + 1)

  # 넘파이 배열로 변환
  image = images[i]
  label = labels[i]
  pred = predicted[i]
  if (pred == label): # 정답과 예측이 동일하면 검은색으로,
    color = 'black'
  else: # 다른 결과가 나왔다면 푸른색으로 지정
    color = 'blue'

  # 이미지 출력
  plt.imshow(image.reshape(28, 28),cmap='gray_r')
  ax.set_title(f'{label}:{pred}', c=color)
  ax.get_xaxis().set_visible(False)
  ax.get_yaxis().set_visible(False)

- plt.show()