



본 영상 교재는
2025년도 과학기술정보통신부 및 정보통신기획평가원의
SW중심대학 사업의 지원을 받아 제작되었습니다.



한국항공대학교
Korea Aerospace University

—KIU—
AI융합대학

SW중심대학






PyTorch 코딩

PyTorch 시작하기

CUDA와 PyTorch

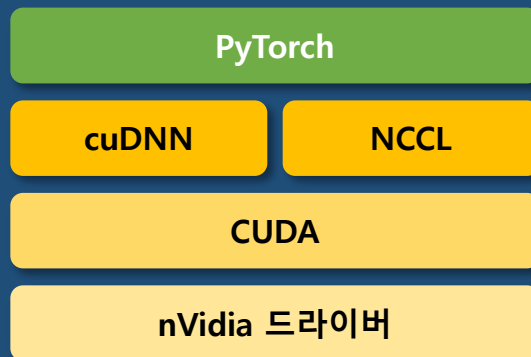
01

CUDA와 PyTorch

- ANN (인공신경망, Artificial Neural Network)과 deep learning
 - 현대 ANN은 더욱 복잡한 문제를 해결하기 위해 **수많은** artificial neuron과 hidden layer로 구성됨
 - 이처럼 복잡한 현대 ANN을 training하는 것을 **deep learning**이라 함
 - 현대 ANN의 feed-forward와 back-propagation을 수식을 통해 직접 구현하는 것은 **매우 어려움**
- Deep learning 라이브러리
 - 현대 ANN의 생성과 training 작업을 **간편한 API**를 통해 **쉽게 구현**할 수 있도록 보조하는 라이브러리
 - ANN의 복잡한 연산 과정이 아닌 ANN architecture, dataset 구성, training 방법 등을 개발하는데 **집중**할 수 있음
 - 연구 / 개발을 위해 **프로토타입**들을 빠르게 제작하고 실험해볼 수 있음
 - 현대에 주로 쓰이는 deep learning 라이브러리로  TensorFlow,  Keras,  PyTorch 등이 있음
- 본 강의에서는 deep learning 라이브러리 중 하나인 **PyTorch**를 다룸
 - 본 강의는 **CUDA**를 활용할 수 있는 환경과, PyTorch **2.4**를 기준으로 함

CUDA와 PyTorch

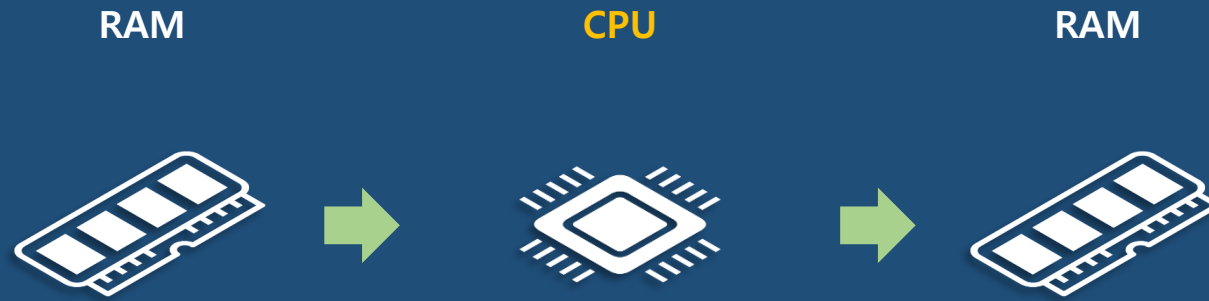
- CUDA와 PyTorch의 계층 구조
 - PyTorch를 효율적으로 활용하기 위한 하위 라이브러리



- **CUDA** (Compute Unified Device Architecture)는 GPU 코어를 **일반 연산**에 활용하게 도와줌
 - ANN은 수많은 뉴런의 feed-forward와 back-propagation을 **병렬 처리**하여 빠르게 수행하는데 필요함
- **cuDNN** (CUDA Deep Neural Network)은 CUDA를 **ANN**에 최적화시켜 ANN의 수행 속도를 높여주는 라이브러리
- **NCCL** (nVidia Collective Communications Library)은 다수의 GPU가 **통신**할 수 있도록 도와주는 라이브러리
 - 단일 GPU로 처리하기 어려운 대형 ANN을 여러 GPU가 나누어 처리하는데 필요함

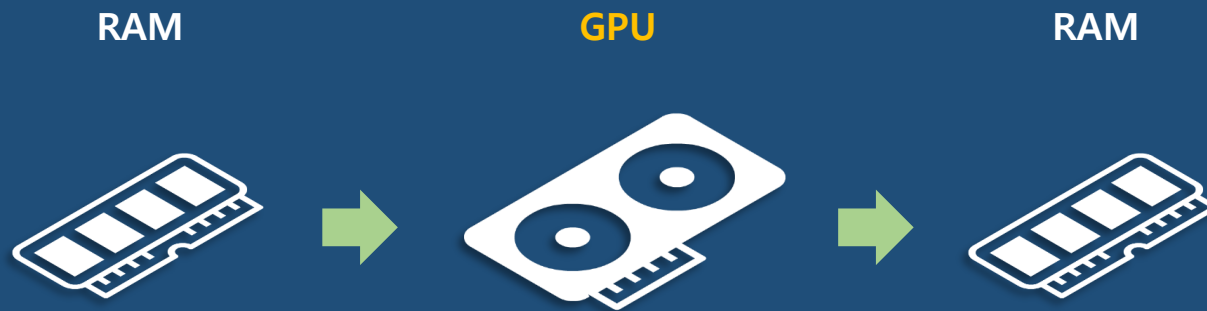
CUDA와 PyTorch

- 컴퓨터 구조와 일반 프로그램



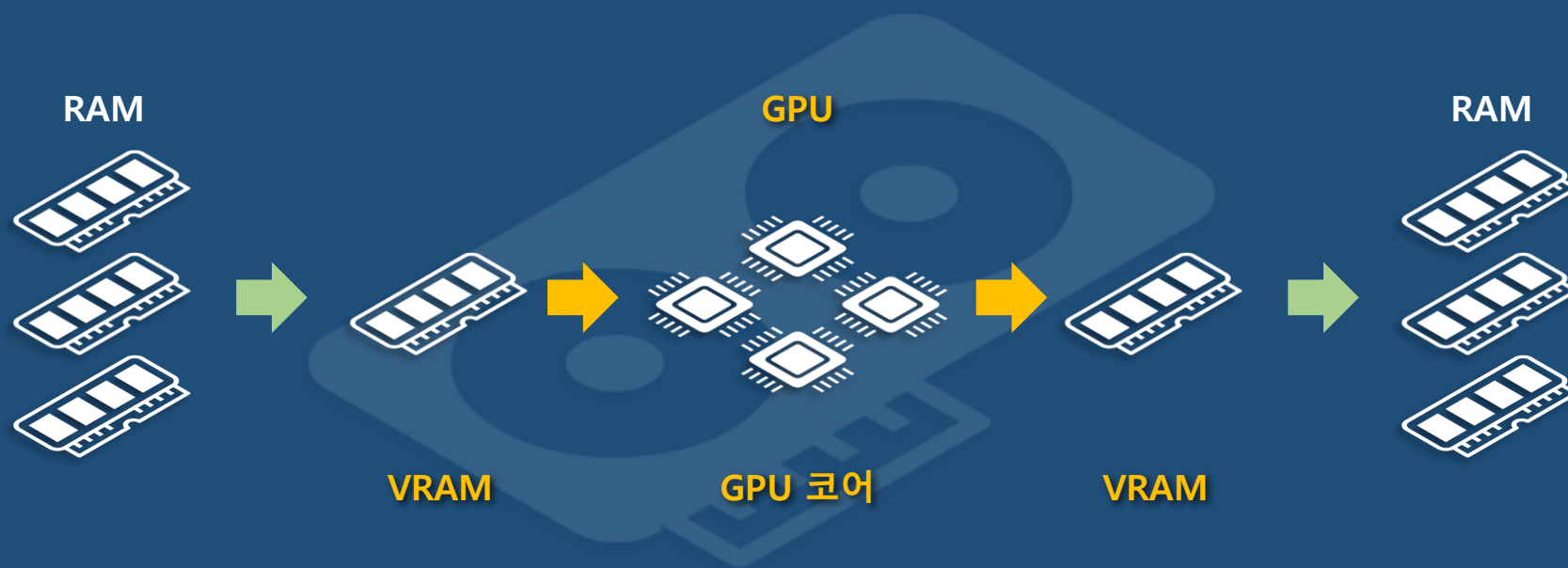
CUDA와 PyTorch

- 컴퓨터 구조와 CUDA를 활용한 프로그램



CUDA와 PyTorch

- 컴퓨터 구조와 CUDA를 활용한 프로그램



CUDA와 PyTorch

- CUDA를 활용한 PyTorch 코딩

- ANN training 작업

1. Dataset 및 ANN 준비
2. ANN parameter와 optimizer를 초기화하여 **VRAM**에 저장
3. Training할 mini-batch를 **VRAM**에 저장
4. **VRAM**에 저장된 mini-batch에 대한 feed-forward 및 back-propagation을 수행하고 ANN parameter를 수정
5. Training이 완료될 때까지 3, 4번 작업을 반복
6. Training이 완료된 ANN parameter를 파일로 저장

- ANN inference 작업

1. ANN architecture를 준비
2. 파일로 저장했던 ANN parameter를 읽어 와 **VRAM**에 저장
3. 실제 input 데이터를 **VRAM**에 저장
4. Input 데이터에 대한 feed-forward를 수행하여 **VRAM**에 output 생성
5. 필요에 따라 output을 **RAM**에 복사하여 응용프로그램에 활용

PyTorch 시작하기

02

PyTorch 시작하기

- 간단한 PyTorch 실습

- Cell 1 : PyTorch 라이브러리 불러오기

```
> import torch                # PyTorch 라이브러리
> print(f'PyTorch version: {torch.__version__}')  # PyTorch 버전

> from torch import nn        # PyTorch ANN 라이브러리
> from torch.utils.data import DataLoader          # Dataset을 mini-batch로 잘라주는 data loader 클래스

> import torchvision          # PyTorch에서 제공하는 이미지 처리 dataset 라이브러리
> from torchvision.transforms import ToTensor      # 이미지를 PyTorch ANN이 처리하기 쉬운 형태로 변환해주는 클래스
```

PyTorch 시작하기

- 간단한 PyTorch 실습

- Cell 2 : 병렬 처리 하드웨어 선택하기

```
> if torch.cuda.is_available():           # nVidia CUDA를 활용할 수 있는지 검사
>     device = 'cuda'                     # nVidia CUDA 활용
> elif torch.backends.mps.is_available(): # Apple MPS를 활용할 수 있는지 검사
>     device = 'mps'                     # Apple MPS 활용
> else:
>     device = 'cpu'                     # 별도의 병렬 처리 하드웨어 없이 CPU 활용
> print(f'Using device: {device}')
```

PyTorch 시작하기

- 간단한 PyTorch 실습
 - Cell 3-1 : Training 및 test dataset 불러오기

```
> training_dataset = torchvision.datasets.FashionMNIST(
>     root      = 'data',
>     train     = True,
>     download  = True,
>     transform = ToTensor()
> )
# Torchvision dataset 중 Fashion MNIST dataset 인스턴스 생성
# Dataset 파일을 읽어올 위치
# Training을 위한 dataset
# Dataset 파일이 없는 경우, 인터넷을 통해 다운로드
# 각 이미지에 자동으로 적용할 변환 작업

> test_dataset = torchvision.datasets.FashionMNIST(
>     root      = 'data',
>     train     = False,
>     download  = True,
>     transform = ToTensor()
> )
# Test를 위한 dataset
```

PyTorch 시작하기

- 간단한 PyTorch 실습

- Cell 3-1 : Training 및 test dataset 불러오기

- FashionMNIST

- Input : 28px × 28px grayscale 이미지



- Target: 각 이미지가 어떤 상품인지 나타내는 class index

- 총 10개의 class로 이루어짐

- ToTensor

- 이미지를 PyTorch ANN이 활용하기 좋은 형태로 변환하는 클래스
 - 일반적인 이미지는 width × height × channel의 dimension에, 각 원소는 [0, 255]의 정수 값을 가짐
 - ToTensor는 이를 channel × height × width의 dimension에, 각 원소의 값을 [0, 1]의 실수 값으로 변환

PyTorch 시작하기

- 간단한 PyTorch 실습
 - Cell 3-2 : Data loader 인스턴스 만들기

```
> mini_batch_size = 64                # Mini-batch의 최대 크기

> training_data_loader = DataLoader(    # Data loader 인스턴스 생성
>     training_dataset,                # Data loader와 연결할 dataset 인스턴스
>     batch_size = mini_batch_size,    # Dataset을 자를 mini-batch의 최대 크기
>     # drop_last = False               # 마지막 mini-batch의 크기가 작은 경우, 이를 무시함
>     shuffle = True                   # 매 epoch마다 dataset의 sampling 순서를 변환
> )

> test_data_loader = DataLoader(
>     test_dataset,
>     batch_size = mini_batch_size
>     # drop_last = False               # 마지막 mini-batch의 크기가 작은 경우, 이를 무시함
>     # shuffle = False                 # shuffle은 false가 기본값이므로 생략 가능
> )
```

PyTorch 시작하기

- 간단한 PyTorch 실습

- Cell 3-2 : Data loader 인스턴스 만들기

- Dataset 크기와 mini-batch 크기

- Dataset 크기가 mini-batch 크기로 나누어 떨어질 경우, 모든 mini-batch가 같은 크기를 가짐
 - 나누어 떨어지지 않는 경우, 마지막 mini-batch의 크기가 다른 mini-batch에 비해 작음

- PyTorch는 이러한 경우에도 정상 작동하도록 설계되어 있음
 - 다만 ANN architecture에서 mini-batch 크기를 고려하도록 되어 있다면 에러 발생
 - 이러한 경우 data loader 인스턴스를 만들 때, `drop_last = True`로 설정하면 마지막 mini-batch를 무시할 수 있음
 - 마지막 mini-batch의 크기가 작을 때 이를 무시하고 넘어감
 - `Shuffle = False`라면, 마지막 sample들을 training 할 수 없음

PyTorch 시작하기

- 간단한 PyTorch 실습

- Cell 3-3 : Mini-batch의 dimension 및 data type 확인하기

```
> print(f'Training dataset size:      {len(training_dataset)}')    # Training dataset의 크기
> print(f'Number of training mini-batches: {len(training_data_loader)}') # Training dataset의 mini-batch 개수

> print(f'Test dataset size:          {len(test_dataset)}')        # Test dataset의 크기
> print(f'Number of test mini-batches: {len(test_data_loader)}')    # Test dataset의 mini-batch 개수

> for x, t in training_data_loader:                                # For 반복문을 통해 mini-batch를 하나씩 가져옴
>     print(f'Shape of input:    {x.shape} {x.dtype}')              # Input tensor의 dimension (64, 1, 28, 28)과 data type (32bit float)
>     print(f'Shape of target:   {t.shape} {t.dtype}')              # Target tensor의 dimension (64, 1)과 data type (64bit integer)
>     break
```

PyTorch 시작하기

- 간단한 PyTorch 실습
 - Cell 4 : ANN architecture 정의
 - ANN architecture



PyTorch 시작하기

- 간단한 PyTorch 실습

- Cell 4-1 : ANN parameter 선언 및 초기화하기

```
> class NeuralNetwork(nn.Module):      # PyTorch의 ANN 클래스를 상속해야 함
>     def __init__(self):              # ANN parameter를 생성하고 초기화해야 함, 반드시 구현해야 함
>         super().__init__()          # PyTorch ANN 클래스 초기화 작업, 반드시 호출해야 함

>     self.fc1 = nn.Linear(28 * 28, 512) # Linear (fully connected) layer 선언, parameter 또는 이를 가진 layer는 반드시 인스턴스 변수로 선언해야 함
>     self.fc2 = nn.Linear(512, 512)
>     self.fc3 = nn.Linear(512, 10)      # Softmax layer에서 10가지 선택지 중 하나를 선택해야 하기 때문에, 그 전에 dimension이 10인 vector를 만들어 줌

>     for layer in self.modules():      # For 반복문을 통해 ANN의 모든 layer에 접근
>         if isinstance(layer, nn.Linear): # Layer가 linear layer인지 검사
>             torch.nn.init.xavier_normal_(layer.weight) # Layer의 weight를 xavier normal 방식으로 초기화
>             torch.nn.init.zeros_(layer.bias)           # Layer의 bias를 0으로 초기화

>     ...
```

PyTorch 시작하기

- 간단한 PyTorch 실습

- Cell 4-1 : ANN parameter 선언 및 초기화하기

- Linear layer

- Fully connected layer
 - 선형 연산을 통해 벡터를 새로운 벡터로 변환함

- PyTorch linear layer은 input tensor를 마지막 차원의 벡터 단위로 dot product함

- Input tensor dimension이 $[N_1, N_2, N_3]$ 일 때, 마지막 차원인 N_3 차원의 벡터 단위로 input을 처리
 - Input tensor의 각 벡터에 동일한 parameter가 적용됨

- 예를 들어 dimension이 $[3, 32, 64]$ 인 이미지가 있을 때,
이미지의 모든 64px의 가로선에 동일한 parameter를 적용하는 작업을 3×32 번 수행함
 - 정확히는 3×32 번 반복하는 것이 아니라 병렬로 동시에 수행됨

PyTorch 시작하기

- 간단한 PyTorch 실습
 - Cell 4-2 : ANN feed-forward 작업 정의하기

```
> class NeuralNetwork(nn.Module):
>     ...
>
>     def forward(self, x):          # ANN의 feed-forward 작업, 반드시 구현해야 함, back-propagation은 이를 통해 자동으로 생성됨
>         x = torch.flatten(        # 초기화 작업과 달리 모든 layer와 연산을 명시해야 함
>             x,                    # Layer의 input tensor
>             start_dim = 1        # Flatten 작업을 시작할 차원 index
>         )
>
>         x = self.fc1(x)           # 초기화 작업에서 선언한 linear (fully connected) layer
>         x = torch.relu(x)         # ReLU activation
>
>         x = self.fc2(x)
>         x = torch.relu(x)
>
>         x = self.fc3(x)
>         y = torch.softmax(        # Softmax layer
>             x,                    # Softmax 연산을 수행할 차원 index
>             dim = 1
>         )
>
>         return y                 # ANN의 최종 output을 return해야 함
```

PyTorch 시작하기

- 간단한 PyTorch 실습

- Cell 4-2 : ANN feed-forward 작업 정의하기

- Flatten layer

- Tensor의 특정 축부터 특정 축까지 1차원 벡터로 누름

- 이미지를 flatten하지 않고 linear layer에 제공하면, 이미지의 모든 가로선에 동일한 weight가 적용됨
 - 이는 이미지를 전체적으로 파악하는 것을 방해할 수 있음
 - 만약 이미지 전체를 하나의 가로선으로 펼쳐준다면, linear layer로도 모든 위치에 적절한 weight를 적용할 수 있음
 - 본 예시의 이미지는 [1, 28, 28]의 dimension을 가지므로, 이를 [1 × 28 × 28] dimension의 벡터로 펼쳐주어야 함
 - Input tensor x는 [64, 1, 28, 28]의 dimension을 가지므로 2 번째 차원부터 flatten을 수행하면 됨
 - 아규먼트 start_dim = 1은 flatten을 수행할 축의 index로, 2 번째 차원부터 flatten을 수행하라는 뜻
 - [64, 1, 28, 28]인 tensor의 dimension을 [64, 1 × 28 × 28]으로 변환

PyTorch 시작하기

- 간단한 PyTorch 실습
 - Cell 4-2 : ANN feed-forward 작업 정의하기
 - Output 종류에 따른 layer
 - Regression : Linear layer 및 convolutional layer 등의 layer를 출력을 그대로 활용
 - Binary classifier : 최종 layer에 **sigmoid** layer 추가
 - Softmax : 최종 layer에 **softmax** layer 추가

PyTorch 시작하기

- 간단한 PyTorch 실습

- Cell 4-3 : ANN 및 optimizer 인스턴스 생성하기

```
> model = NeuralNetwork().to(device)    # ANN 인스턴스를 생성하여 parameter와 함께 device에 저장, CUDA를 활용할 경우 GPU VRAM에 복사
> optimizer = torch.optim.SGD(          # SGD (Steepest Gradient Descent) optimizer 인스턴스 생성
>     model.parameters(),                # optimizer로 조정할 ANN parameter, optimizer 내부 변수들도 ANN parameter가 저장된 device에 함께 저장
>     lr = 1e-3                          # learning rate, 0.001로 설정
> )
```


PyTorch 시작하기

- 간단한 PyTorch 실습

- Cell 5-1 : Training epoch 정의하기

```
> def train(device, data_loader, model, optimizer):
>     total_loss = 0

>     model.train()
>     for x, t in data_loader:
>         x = x.to(device)

>         y = model(x)
>         t = t.to(y.device)
>         loss = torch.nn.functional.cross_entropy(y, t)

>         loss.backward()
>         optimizer.step()
>         optimizer.zero_grad()

>         mini_batch_size = x.shape[0]
>         total_loss += loss.item() * mini_batch_size

>     dataset_size = len(data_loader.dataset)
>     average_loss = total_loss / dataset_size

>     return average_loss
```

ANN에게 training중임을 알림
For 반복문을 통해 mini-batch를 하나씩 가져옴
Input tensor를 device에 저장, ANN과 같은 device에 저장해야 함

함수를 호출하듯이, ANN에 input tensor x를 제공하여 output tensor y를 생성
Target tensor t를 output tensor y와 같은 device에 저장
Target과 output 사이의 loss를 cross-entropy 방식으로 계산, mini-batch 내 sample들의 평균값

평균 loss를 통해 backpropagation 수행, gradient $\nabla\theta$ 를 계산
Optimizer를 통해 ANN parameter를 수정
Gradient $\nabla\theta$ 를 0으로 초기화

Mini-batch의 sample 개수
Dataset의 총 loss를 구하기 위해, 평균 loss에 sample 개수를 곱함

Dataset 크기
Dataset의 평균 loss를 구하기 위해, 총 loss를 dataset 크기로 나눔

PyTorch 시작하기

- 간단한 PyTorch 실습
 - Cell 5-1 : Training epoch 정의하기
 - `model.train()` 및 `eval()`
 - ANN에게 training 중인지 evaluation 중인지 알려줌
 - ANN에는 RNN, dropout, batch normalization, layer normalization 등 training 중인지 evaluation 중인지에 따라 **architecture** 및 **작동 방식**이 달라지는 layer들이 존재할 수 있음
 - 이런 layer들에게 현재 training 중인지 아닌지를 알려 상황에 맞도록 작동하도록 함

PyTorch 시작하기

- 간단한 PyTorch 실습

- Cell 5-2 : Test 작업 정의하기

```
> def test(device, data_loader, model):
>     total_loss = 0
>     total_correct = 0
>
>     model.eval()           # ANN에게 training중이 아님을 알림
>     with torch.no_grad():   # Python context를 통해, 아래 코드를 back-propagation 없이 수행하라고 알림
>         for x, t in data_loader:
>             x = x.to(device)
>
>             y = model(x)
>             t = t.to(y.device)
>             loss = torch.nn.functional.cross_entropy(y, t)
>
>             mini_batch_size = x.shape[0]
>             total_loss += loss.item() * mini_batch_size
>             total_correct += (y.argmax(dim = 1) == t).
>                 .type(torch.float).sum().
>                 .item()      # Softmax output 중 확률이 가장 높은 class index를 target과 비교, 같은 경우 True, 다른 경우 False
>                               # True / False를 1.0 / 0.0으로 변환한 후 합산
>                               # 원소가 하나인 tensor를 scalar로 변환
>
>     dataset_size = len(data_loader.dataset)
>     average_loss = total_loss / dataset_size
>     accuracy = total_correct / dataset_size
>
>     return average_loss, accuracy
```

PyTorch 시작하기

- 간단한 PyTorch 실습
 - Cell 5-2 : Test 작업 정의하기
 - `torch.no_grad()`
 - ANN에게 back-propagation 작업이 없음을 알림
 - Back-propagation을 위해선 `gradient`, `optimizer`등을 저장할 **VRAM 공간**이 필요함
 - ANN architecture 및 optimizer의 종류에 따라 VRAM 공간을 상당히 많이 차지할 수 있음
 - 따라서 back-propagation이 필요 없는 작업에서는 이를 알려 VRAM 공간을 낭비하지 않아야 함

PyTorch 시작하기

- 간단한 PyTorch 실습
 - Cell 5-2 : Test 작업 정의하기
 - Softmax와 argmax
 - Softmax는 element들의 합이 1이 되도록 하여, 각 element가 **확률** 값을 가지도록 함
 - Argmax는 element들 중 **가장 큰 값**을 가지는 element의 **index**를 return
 - 본 예시에서는 총 10개의 class가 존재하므로, output은 10개의 확률값을 가지는 **softmax**로 구현
 - Argmax를 통해 주어진 input을 **가장 높은 확률**을 가진 **class**로 분류할 수 있음
 - Input이 mini-batch로 주어졌으므로, output도 mini-batch가 되어 [64, 10]의 dimension을 가짐
 - 아규먼트 **dim = 1**은 argmax를 수행할 차원의 index를 뜻함
 - 즉, **y.argmax(dim = 1)**은 [64, 10] 중 **2번째 차원**, 즉 10개의 class에 대해 argmax를 수행하라는 뜻

PyTorch 시작하기

- 간단한 PyTorch 실습

- Cell 5-3 : 개별 input에 대한 inference 작업 정의하기

```
> def inference(device, data, model):  
>     # 이미지 처리를 위한 코드           # 실제로는 이미지를 gray-scale로 조정하고 사이즈를 조정하는 코드 필요  
  
>     # image_to_tensor = ToTensor()  
>     # data = image_to_tensor(image)      # ToTensor 클래스를 통해, 이미지를 PyTorch ANN이 처리하기 쉬운 형태로 변환  
  
>     x = data.view(1, 1, 28, 28)         # view()를 통해 dimension을 변환, 개별 input을 크기가 1인 mini-batch로 만들어 줌  
  
>     model.eval()                        # Inference 작업은 training이 아니므로 이를 ANN에게 알림  
>     with torch.no_grad():               # Inference 작업은 back-propagation이 없음  
>         x = x.to(device)                # Input은 반드시 ANN과 같은 device에 저장해야 함  
>         y = model(x)  
  
>     return y                           # 최종 output을 return
```

PyTorch 시작하기

- 간단한 PyTorch 실습
 - Cell 6 : Training loop 정의하기

```
> max_epoch = 10
> for t in range(max_epoch):
>     print(f'Epoch {t + 1 :>3d} / {max_epoch}')

>     training_loss = train(device, training_data_loader, model, optimizer)
>     print(' Training progress')
>     print(f' Average loss: {training_loss :>8f}')

>     test_loss, test_accuracy = test(device, test_data_loader, model)
>     print(' Validation performance')
>     print(f' Average loss: {test_loss :>8f}')
>     print(f' Accuracy: {(100 * test_accuracy) :>0.2f}%',)

>     # if training 종료 조건:
>     #     break

>     torch.save(
>         model.state_dict(),
>         'model.pth'
>     )

>     print('Saved PyTorch ANN parameters to model.pth')
```

최대 Training 반복 횟수
최대 반복 횟수만큼 training epoch 반복

Training dataset을 통해 training epoch 수행

Test dataset을 통해 validation 수행

특정 조건에서 training을 종료(early stop)시킬 수 있음

PyTorch의 파일 저장 함수
저장할 내용, ANN의 parameter를 dictionary 형태로 가져옴
저장할 파일의 위치 및 이름

PyTorch 시작하기

- 간단한 PyTorch 실습
 - Cell 6 : Training loop 정의하기
 - Validation
 - 최종적인 성능을 확인하는 test 작업과 달리, training 도중에 generalization 성능을 확인하는 작업
 - 일반적으로 training dataset의 일부를 활용함
 - 약 90%만 training에 활용, 나머지는 validation에만 활용
 - 본 예시에서는 간단하게 test dataset으로 validation을 수행

PyTorch 시작하기

- 간단한 PyTorch 실습

- Cell 7 : 임의의 sample에 대한 inference 성능 확인하기

```

> import random

> model = NeuralNetwork().to(device) # 파일로 저장한 ANN parameter를 불러오는 예시를 위해 ANN을 새로 생성, parameter를 device에 저장
> model.load_state_dict(             # ANN parameter를 복원
>     torch.load(                     # PyTorch의 파일 읽기 함수
>         'model.pth',               # 읽을 파일의 위치 및 이름
>         weights_only = True        # 파일에서 ANN parameter만 읽음
>     )
> )

> test_sample_index = random.randint(0, len(test_dataset) - 1)
> x, t = test_dataset[test_sample_index] # Index를 통해 dataset에서 특정 sample만 가져옴

> y = inference(device, x, model)

> classes = ['T-shirt/top', 'Trouser', 'Pullover', 'Dress', 'Coat', 'Sandal', 'Shirt', 'Sneaker', 'Bag', 'Ankle boot'] # Fashion MNIST dataset의 target class label

> predicted = classes[y.argmax(dim = 1)] # Softmax output 중 확률이 가장 높은 class label
> actual    = classes[t]

> print('Random sample inference')
> print(f' Predicted: "{predicted}", Actual: "{actual}")
    
```

Have a nice day!

