PyTorch Computer Vision Cookbook Chapter1

2021210088 허지혜

Verifying the installation

```
In [1]: import torch

In [2]: torch.__version__
Out[2]: '1.8.0'

In [3]: import torchvision

In [4]: torchvision.__version__
Out[4]: '0.9.0'

In [5]: torch.cuda.is_available()
Out[5]: True

In [6]: torch.cuda.device_count()
Out[6]: 1

In [7]: torch.cuda.current_device()
Out[7]: 0

In [8]: torch.cuda.get_device_name(0)
Out[8]: 'GeForce GTX 1060 6GB'
```

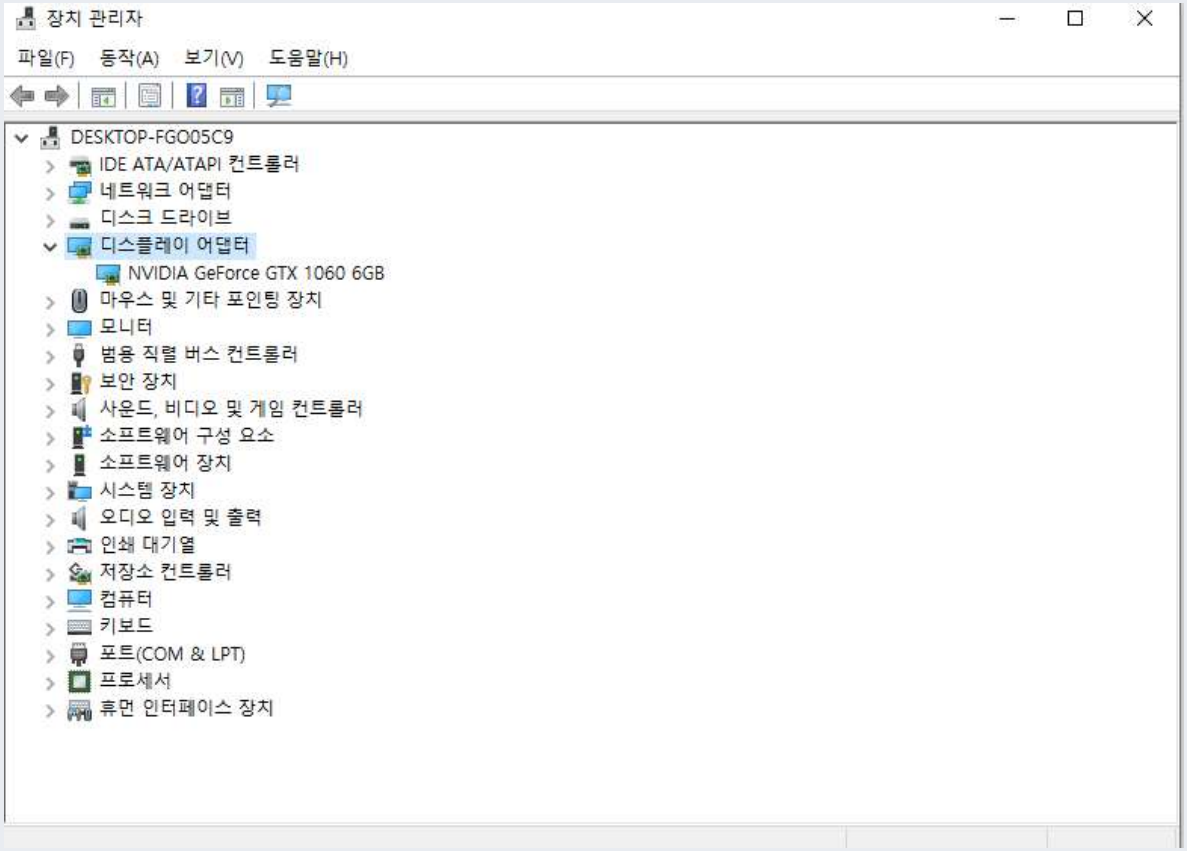
cuda 깔려있음 => true
cuda 깔려있지 않음 => false

CUDA,cudnn install

CUDA(Computed Unified Device Architecture)는 NVIDIA 사에서 개발한 Grapic Processing Unit 개발 tool이다.

1. 컴퓨터 드라이버 버전 확인

내 PC > 속성 > 장치 관리자 > 디스플레이 어댑터



CUDA,cudnn install

2. CUDA Download

1) 링크 들어감 : https://developer.nvidia.com/cuda-gpus#collapse2
2) GeForce and TITAN Products 클릭
3) 컴퓨터에 맞는 CUDA 버전 찾아보기

GeForce GTX 1060	6.1

GPUs supported [edit]

Supported CUDA level of GPU and card. See also at Nvidia:

- CUDA SDK 1.0 support for compute capability 1.0 – 1.1 (Tesla)[25]
- CUDA SDK 1.1 support for compute capability 1.0 – 1.1+x (Tesla)
- CUDA SDK 2.0 support for compute capability 1.0 – 1.1+x (Tesla)
- CUDA SDK 2.1 – 2.3.1 support for compute capability 1.0 – 1.3 (Tesla)[26][27][28][29]
- CUDA SDK 3.0 – 3.1 support for compute capability 1.0 – 2.0 (Tesla, Fermi)[30][31]
- CUDA SDK 3.2 support for compute capability 1.0 – 2.1 (Tesla, Fermi)[32]
- CUDA SDK 4.0 – 4.2 support for compute capability 1.0 – 2.1+x (Tesla, Fermi, more?).
- CUDA SDK 5.0 – 5.5 support for compute capability 1.0 – 3.5 (Tesla, Fermi, Kepler).
- CUDA SDK 6.0 support for compute capability 1.0 – 3.5 (Tesla, Fermi, Kepler).
- CUDA SDK 6.5 support for compute capability 1.1 – 5.x (Tesla, Fermi, Kepler, Maxwell). Last version with support for compute capability 1.x (Tesla)
- CUDA SDK 7.0 – 7.5 support for compute capability 2.0 – 5.x (Fermi, Kepler, Maxwell).
- CUDA SDK 8.0 support for compute capability 2.0 – 6.x (Fermi, Kepler, Maxwell, Pascal). Last version with support for compute capability 2.x (Fermi) (Pascal GTX 1070Ti Not Supported)
- CUDA SDK 9.0 – 9.2 support for compute capability 3.0 – 7.2 (Kepler, Maxwell, Pascal, Volta) (Pascal GTX 1070Ti Not Supported. CUDA SDK 9.0 and support CUDA SDK 9.2).
- CUDA SDK 10.0 – 10.2 support for compute capability 3.0 – 7.5 (Kepler, Maxwell, Pascal, Volta, Turing). Last version with support for compute capability 3.x (Kepler). 10.2 is the last official release for macOS, as support will not be available for macOS in newer releases.
- CUDA SDK 11.0 – 11.2 support for compute capability 3.5 – 8.6 (Kepler (in part), Maxwell, Pascal, Volta, Turing, Ampere)[33] New data types: Bfloat16 and TF32 on third-generations Tensor Cores.[34]

CUDA,cudnn install

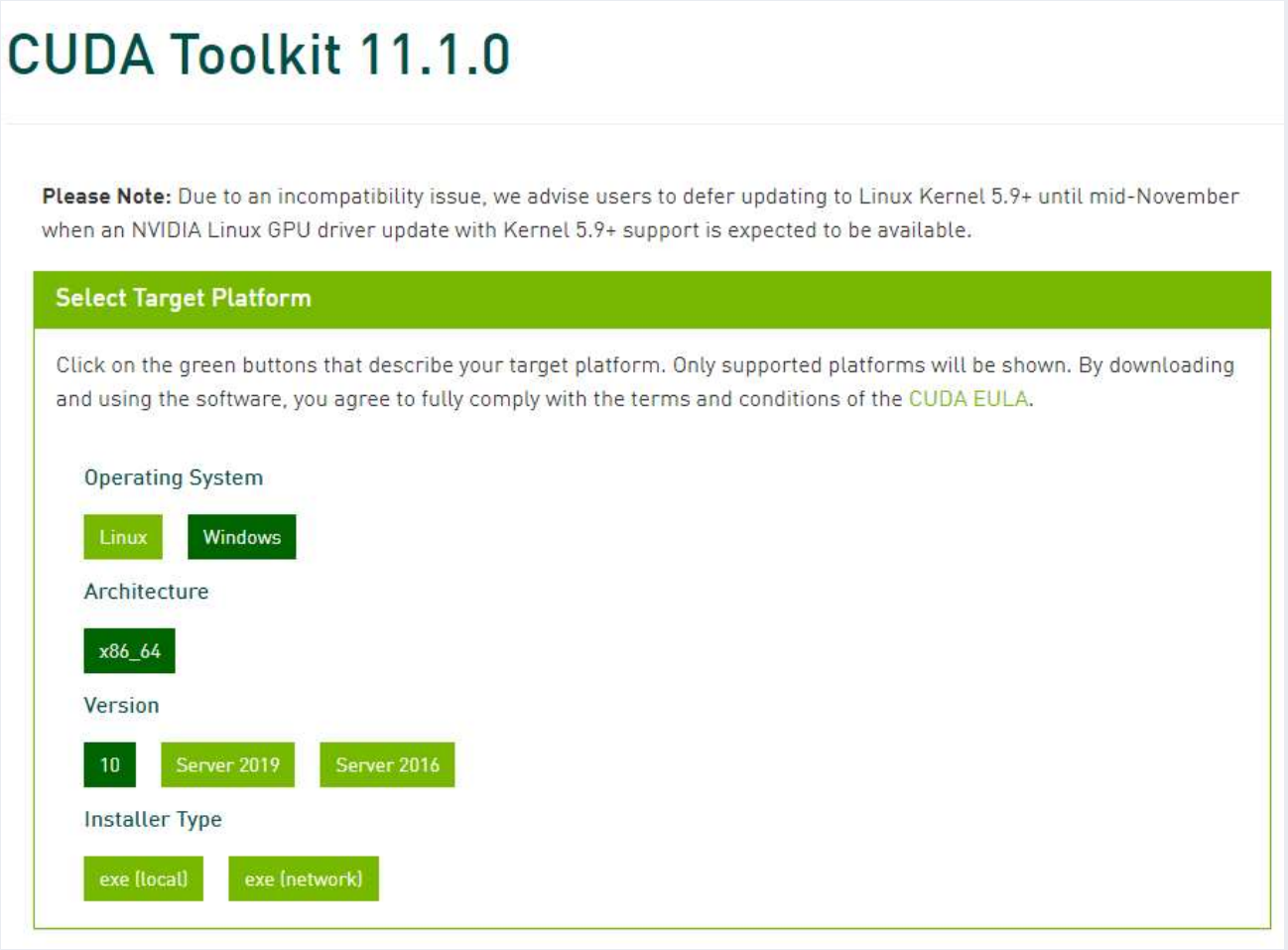
2. CUDA Download

4) CUDA download 들어가서
버전에 맞게 설치

https://developer.nvidia.com/cuda-11.1.0-download-archive?target_os=Windows&target_arch=x86_64&target_version=10

설치 확인 방법

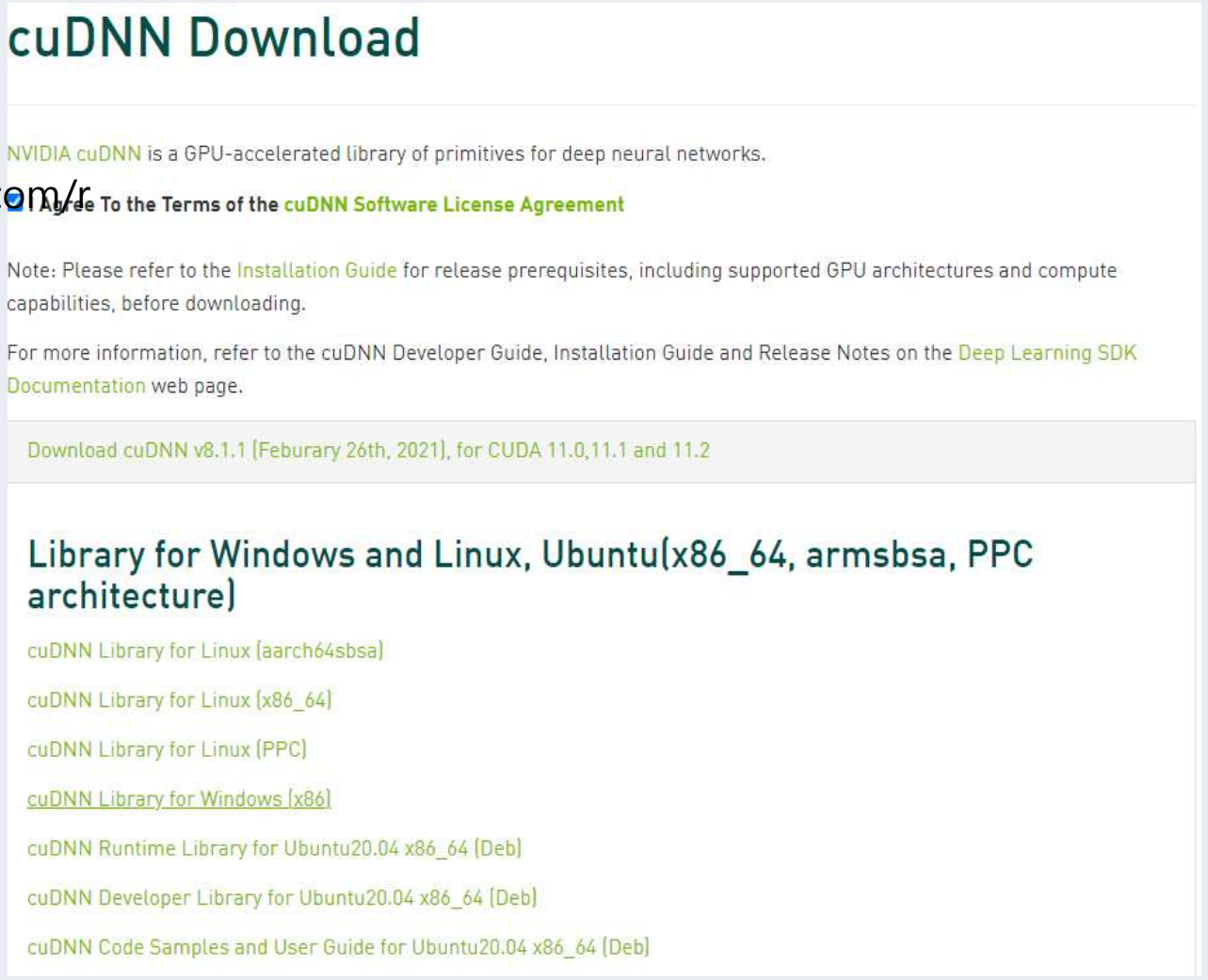
```
C:₩Users₩HeoJiHae>nvcc --version
nvcc: NVIDIA (R) Cuda compiler driver
Copyright (c) 2005-2020 NVIDIA Corporation
Built on Tue_Sep_15_19:12:04_Pacific_Daylight_Time_2020
Cuda compilation tools, release 11.1, V11.1.74
Build cuda_11.1.relgpu_drvr455TC455_06.29069683_0
```



CUDA,cudnn install

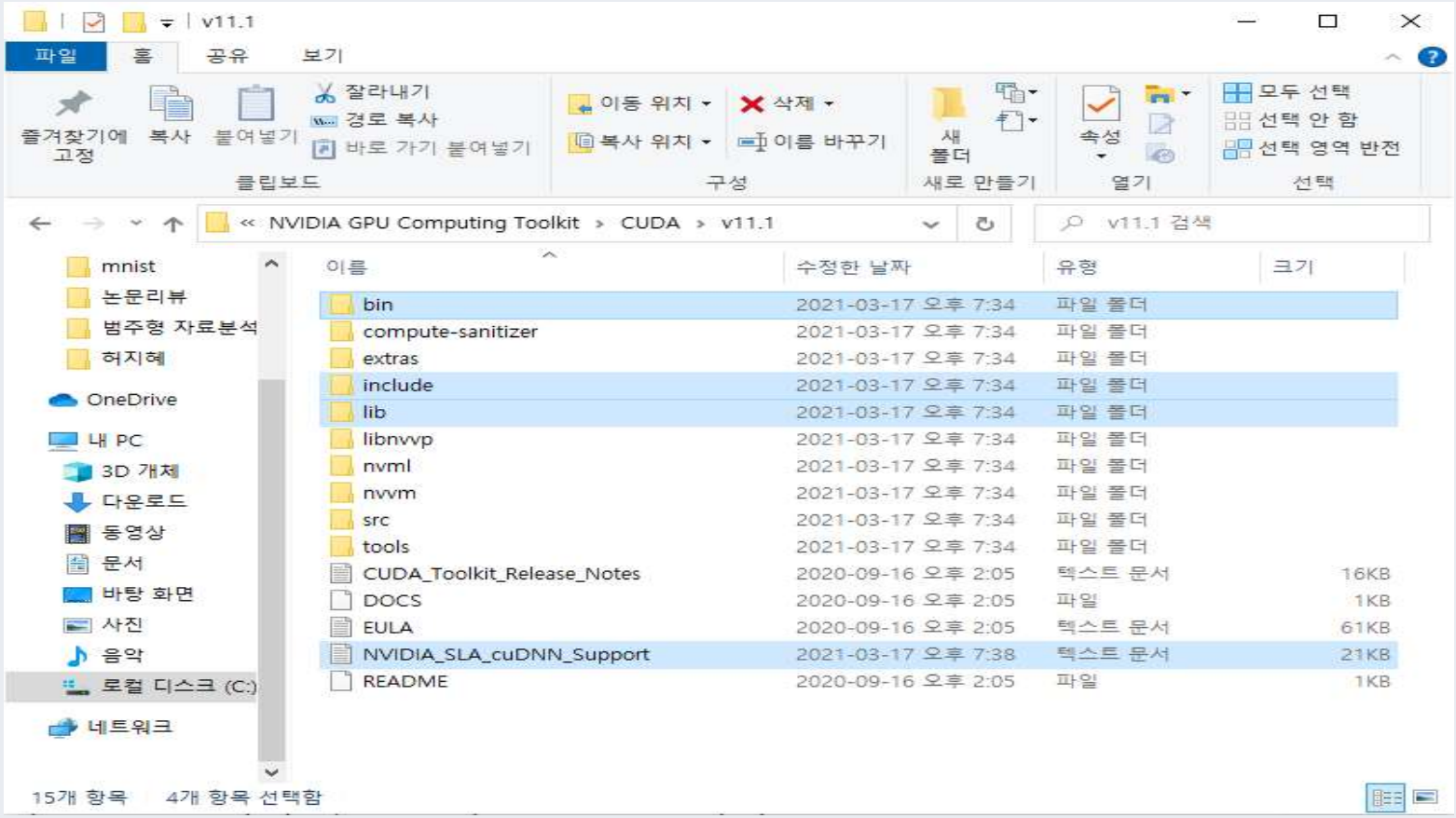
3. cudnn Download

1) 사이트 들어감 :https://developer.nvidia.com/rdp/cudnn-download
2) 로그인
3) CUDA 버전에 맞는 cudnn 설치



CUDA,cudnn install

다운로드 받은 cudnn 파일을 C:₩Program Files₩NVIDIA GPU Computing Toolkit₩CUDA₩v11.1 에다가 붙여넣기 하면 된다.



Working with PyTorch tensors

PyTorch is build on tensors, n-dimensional array.

NumPy Arrays	Pytorch tensors	Description
numpy.ones(.)	torch.ones(.)	Create an array of ones
numpy.zeros(.)	torch.zeros(.)	Create an array of zeros
numpy.random.rand(.)	torch.rand(.)	Create a random array
numpy.array(.)	torch.tensor(.)	Create an array from given values
x.shape	x.shape or x.size()	Get an array shape

in this recipe 1. define and change tensors
2. convert(변하다) tensors into arrays
3. move them between computing devices

Defining the tensor data type

The default tensor data type is torch.float32. This is the most used data type for tensor operations(조작,운영).

In [1]:
```
#1. Define a tensor with a default data type:
import torch
x = torch.ones(2,2)
print(x)
print(x.dtype) # 자료형
```

```
tensor([[1., 1.],
        [1., 1.]])
torch.float32
```

In [2]:
```
#2. Specify the data type when defining a tensor:
x = torch.ones(2,2,dtype=torch.int8)
print(x)
print(x.dtype)
```

```
tensor([[1, 1],
        [1, 1]], dtype=torch.int8)
torch.int8
```

Changing the tensor's data type

We can change a tensor's data type using the ".type" method

In [3]:
```
#1. Define a tensor with torch.uint8 type:
x = torch.ones(1, dtype=torch.uint8)
print(x.dtype)
```

```
torch.uint8
```

In [4]:
```
#2. Change the tensor data type:
x = x.type(torch.float)
print(x.dtype)
```

```
torch.float32
```

Converting tensor into Numpy arrays or Tensors

tensor → numpy array

```
#1. Define a tensor:
x = torch.rand(2,2)
print(x)
print(x.dtype)
```

```
tensor([[0.5337, 0.9929],
        [0.4765, 0.9767]])
torch.float32
```

```
#2. Convert the tensor into a NumPy array:
y = x.numpy()
print(y)
print(y.dtype)
```

```
[[0.5336927  0.99288917]
 [0.4764809  0.9767209 ]]
float32
```

numpy array → tensor

```
#1. Define a NumPy array:
import numpy as np
x = np.zeros((2,2),dtype=np.float32)
print(x)
print(x.dtype)
```

```
[[0. 0.]
 [0. 0.]]
float32
```

```
#2. Convert the NumPy array into a Pytorch tensor:
y = torch.from_numpy(x)
print(y)
print(y.dtype)
```

```
tensor([[0., 0.],
        [0., 0.]])
torch.float32
```

Moving tensors between devices

In [9]:
```
import torch
#1. Define a tensor on CPU:
x = torch.tensor([1.5,2])
print(x)
print(x.device)
```

```
tensor([1.5000, 2.0000])
cpu
```

In [16]:
```
#2. Define a CUDA device:
if torch.cuda.is_available():
    device = torch.device("cuda")
```

In [11]:
```
#3. Move the tensor onto the CUDA device:
x = x.to(device)
print(x)
print(x.device)
```

```
tensor([1.5000, 2.0000], device='cuda:0')
cuda:0
```

In [12]:
```
#4. Similarly, we can move tensors to CPU:
device = torch.device("cpu")
x = x.to(device)
print(x)
print(x.device)
```

```
tensor([1.5000, 2.0000])
cpu
```

In [15]:
```
#5. We can also directly create a tensor on any device:
device = torch.device("cuda:0")
x = torch.ones(2,2,device=device)
print(x)
```

```
tensor([[1., 1.],
        [1., 1.]], device='cuda:0')
```

How it works...

First, we defined a tensor obtained(획득) the tensor type, and changed its type.
Then, we converted(변환) PyTorch tensors into NumPy arrays and vice versa(그 반대).
We also moved tensors between the CPU and CUDA devices.

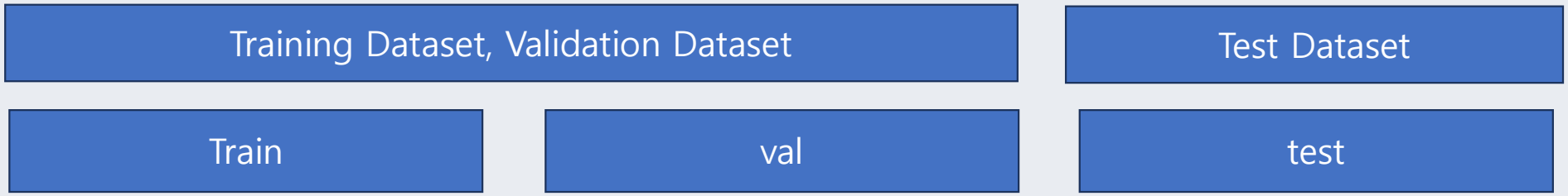
Next, we showed you how to change a tensor data type using the ".type" method. Then, we showed how to convert PyTorch tensors into Numpy arrays using the ".numpy" method.

After that, we showed you how to convert a NumPy array into a Pytorch tensor using the ".from_numpy(x)" method. Then, to move tensors from a CPU device to a GPU device and vice versa, using the ".to"method.

As you have seen,(보다시피) if you do not specifiy the device, the tensor will be hostrd on the CPU device.

Loading and processing data

We use the training dataset to train the model.
The validation dataset is used to track the model's performance during training.
We use the test dataset for the final evaluation of the model.



input : X or x
output : Y or y

In this recipe, we will learn about Pytorch data tools. We can use these tools to load and process data.

Loading a dataset

```
#1. First, we will load the MNIST datasets:
from torchvision import datasets
#path to store data and/or load from
path2data = "./data"
#loading training data
train_data = datasets.MNIST(path2data, train=True, download=True)
```

```
#2. Then, we will extract the input data and target labels:
#extract data and targets(데이터와 타겟을 추출하다.)
x_train, y_train = train_data.data,train_data.targets

print(x_train.shape)
print(y_train.shape)
```

```
torch.Size([60000, 28, 28])
torch.Size([60000])
```

```
#3. Next, we will load the MNIST  test dataset:
#loading validation data
val_data = datasets.MNIST(path2data, train = False, download = True)
```

```
#4. Then, we will extract the input data and target labels:
x_val, y_val = val_data.data, val_data.targets
```

Loading a dataset

```
#5. After that, we will add a new dimension to the tensors:
if len(x_train.shape) == 3:
  x_train = x_train.unsqueeze(1)

print(x_train.shape)

if len(x_val.shape) == 3:
  x_val = x_val.unsqueeze(1)

print(x_val.shape)
```

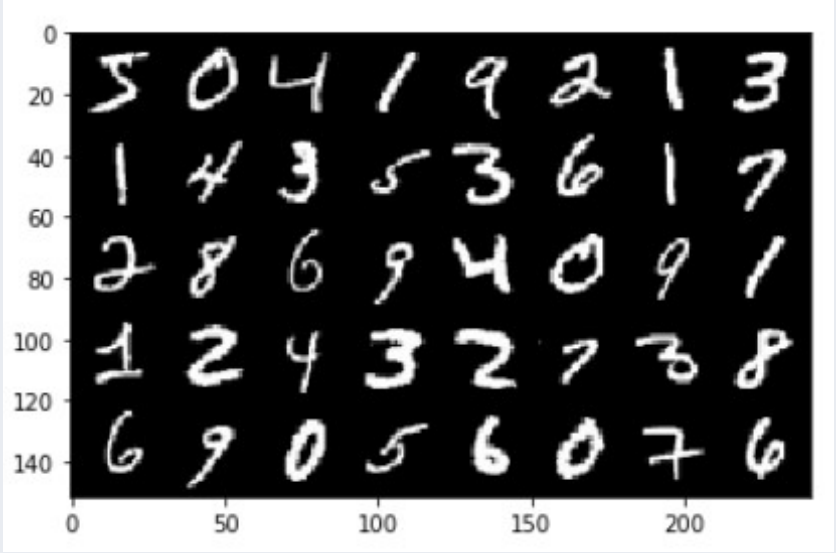
```
torch.Size([60000, 1, 28, 28])
torch.Size([10000, 1, 28, 28])
```

```
#6. Next, we will import the required packages:
from torchvision import utils
import matplotlib.pyplot as plt
import numpy as np
%matplotlib inline
```

```
#7. Then, we will define a helper function to display tensors as images:
def show(img):
  #convert tensor to numpy array
  npimg = img.numpy()
  #Convert to H*W*C shape
  npimg_tr = np.transpose(npimg, (1,2,0))
  plt.imshow(npimg_tr, interpolation='nearest')
```

```
#8. Next, we will create a grid of images and display them:
#make a grid of 40 images, 8 images per row
x_grid = utils.make_grid(x_train[:40], nrow=8, padding=2)
print(x_grid.shape)

#cell helper function
show(x_grid)
```

```
torch.Size([3, 152, 242])
```



Data transformaation

Image transformation is an effective technique that's used to improve a model's performance(성능). The torchvision pacakge provides common image transformations through the transform class. Let's take a look:

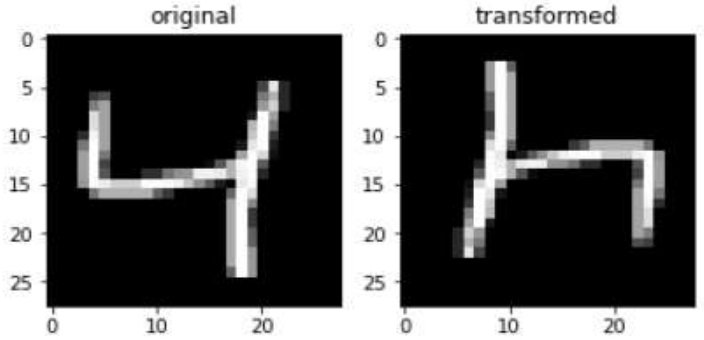
```
#1.Let's define a transform class in order to apply some image transformations
#on the MNIST dataset:
from torchvision import transforms

#loading MNIST training datset
train_data = datasets.MNIST(path2data, train=True, download=True)

#define transformations
data_transform = transforms.Compose([
                    transforms.RandomHorizontalFlip(p=1),
                    transforms.RandomVerticalFlip(p=1),
                    transforms.ToTensor(),
])
```

```
#2. Let's apply the transformations on an image from the MNIST dataset:
#get a sample image from training dataset
img = train_data[2][0]

#transform sample image
img_tr = data_transform(img)

#convert tensor to numpy array
img_tr_np = img_tr.numpy()
```

Data transformaation

```
#show original and transformed images
plt.subplot(1,2,1)
plt.imshow(img, cmap="gray")
plt.title("original")
plt.subplot(1,2,2)
plt.imshow(img_tr_np[0], cmap='gray')
plt.title("transformed")
```

```
Text(0.5, 1.0, 'transformed')
```




```
#3. We can pass the transformer function to the dataset class:

#define transformations
data_transform = transforms.Compose([
                 transforms.RandomHorizontalFlip(1),
                 transforms.RandomVerticalFlip(1),
                 transforms.ToTensor(),])

#Loading MNIST training data with on-the-fly transofrmations
train_data = datasets.MNIST(path2data, train=True, download=True, transform=data_tr
```

Wrapping(싸다) tensors into a dataset

If your data is available in tensors, you can wrap them as as Pytorch dataset using the Tensor Dataset class.
This will make it easier to iterate over data during training. Let's get started :

```
#1. Let's create a Pytorch dataset by wrapping x_train, y_train:
from torch.utils.data import TensorDataset

#wrap tensors into a dataset
train_ds = TensorDataset(x_train, y_train)
val_ds = TensorDataset(x_val, y_val)

for x,y in train_ds :
    print(x.shape, y.item())
    break
```

```
torch.Size([1, 28, 28]) 5
```

Creating data loaders

To easily iterate over the data during training, we can create a data loader using the DataLoder class, as following:

```
#1. Let's create two data loaders for the training and validation dataset:
from torch.utils.data import DataLoader

#create a data loader from dataset
train_dl = DataLoader(train_ds, batch_size=8)
val_dl = DataLoader(val_ds, batch_size=8)

#iterate over batches
for xb, yb in train_dl:
    print(xb.shape)
    print(yb.shape)
    break
```

```
torch.Size([8, 1, 28, 28])
torch.Size([8])
```

How it works..

First, we imported the datasets package from torchvision.
Then, we downloaded MNIST training dataset into a local folder.
Next, we wextracted the input data and target labels into PyTorch tensors and printed their size.
Training dataset : 60000 inputs and targets.
Then, same test dataset : 10000 inputs and targets.
Next, we added a new dimension to the input tensors since we want the tensor shape to be B(Batch size)*C(Channel)*H(Height)*W(Width).
This is the common shape for the inputs tensors in PyTorch.
Then, we defined a helper function to display sample images.
We need utils from torchvision to create a grid of 40 images in five rows and eight columns.

In the Data transformation subsection, we introduced the torchvision.transforms package.
We composed the RandomHorizontalFlip and RandomVerticaFlip method to augment the dataset and the ToTensor method to convert image into PyTorch tensors.

How it works..

Then, we passed the transformer function to the dataset class.
This way, data transformation will happen on-the-fly.
This is a useful technique for large datasets that cannot be loaded into memory all at once.

In the Wrapping tensors into a dataset subsection, we created a dataset from tensors.

In the Creating data loaders subsection, we usesd DataLoader class to define data loaders.
We need two data loaders from train_dl, val_ds.
Then, we etracted a mini-batch from train_dl.

Building Models & How to do it ...

A model is a collection of connected layers that process the inputs to generate the outputs.
You can use the nn package to define models.

The nn package is a collection of modules that provide common deep learning layers.
A module or layer of nn receives input tensors, computes output tensors, and holds the weights, if any.

There are two methods we can use to define models in Pytorch: nn.Sequential and nn.Module

We will define a linear layer, a two-layer network, and a multilayer convolutional network..

Defining a linear layer

```
import torch
from torch import nn

#input tensor dimension(치수) 64 * 1000
input_tensor = torch.randn(64,1000)
# 평균이 0이고 표준편차가 1인 가우시안 정규분포를 이용해 생성

# linear layer with 1000 inputs and 100 outputs
linear_layer = nn.Linear(1000,100)

# output of the linear layer
output = linear_layer(input_tensor)
print(output.size())
```

```
torch.Size([64, 100])
```

Defining models using nn.Sequential

We can use the nn.Sequential package to create a deep learning model by passing layers in order(순서대로 레이어 전달).
Consider the two-layer neural network depicted in the following image:

```
#1. Let's implement and print the model using nn.Sequential:

from torch import nn

#define a two-layer model
model = nn.Sequential(
    nn.Linear(4,5),
    nn.ReLU(),
    nn.Linear(5,1),
)
print(model)
```

```
Sequential(
  (0): Linear(in_features=4, out_features=5, bias=True)
  (1): ReLU()
  (2): Linear(in_features=5, out_features=1, bias=True)
)
```

Defining models using nn.Module

Another way of defining models in Pytorch is by subclassing the nn.Module class.
In this method, we specify the layers in the init method of the class.
Then, in the forward method, we specify the layers to inputs.
This method provides better flexibility for building customized models.

```
#1.First, we will implement the bulk of the class:
import torch.nn.functional as F
class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()

    def forward(self, x):
        pass
```

```
#2. Then, we will define the __init__function :

def __init__(self) :
    super(Net, self).__init__()
    self.conv1 = nn.Conv2d(1, 20, 5, 1)
    self.conv2 = nn.Conv2d(20, 50, 5, 1)
    self.fc1 = nn.Linear(4*4*50, 500)
    self.fc2 = nn.Linear(500,10)
```

```
#3. Next, we will define the forward function:
def forward(self, x):
    x = F.relu(self.conv1(x))
    x = F.max_pool2d(x,2,2)
    x = F.relu(self.conv2(x))
    x = F.max_pool2d(x,2,2)
    x = x.view(-1, 4*4*50)
    x = F.relu(self.fc1(x))
    x = self.fc2(x)
    return F.log_softmax(x, dim = 1)
```

```
#4. Then, we will override both class functions, __init__ and forward:
Net.__init__ = __init__
Net.forward = forward
```

```
#5. Next, we will create an object of the Net class and print the model:
model = Net()
print(model)
```

```
Net(
  (conv1): Conv2d(1, 20, kernel_size=(5, 5), stride=(1, 1))
  (conv2): Conv2d(20, 50, kernel_size=(5, 5), stride=(1, 1))
  (fc1): Linear(in_features=800, out_features=500, bias=True)
  (fc2): Linear(in_features=500, out_features=10, bias=True)
)
```

Moving the model to a CUDA device

```
#1. Let's get the model's device:
print(next(model.parameters()).device)
```

```
cpu
```

```
#2. Then, we will move the model to the CUDA device :
device = torch.device("cuda:0")
model.to(device)
print(next(model.parameters()).device)
```

```
cuda:0
```

Printing the model summary

```
!pip install torchsummary
```

```
Requirement already satisfied: torchsummary in c:₩users₩heojihae₩anaconda3₩lib₩site
-packages (1.5.1)
```

```
from torchsummary import summary
summary(model, input_size = (1,28,28))
```

```
----------------------------------------------------------------
        Layer (type)               Output Shape         Param #
================================================================
            Conv2d-1          [-1, 20, 24, 24]             520
            Conv2d-2            [-1, 50, 8, 8]          25,050
            Linear-3                 [-1, 500]         400,500
            Linear-4                  [-1, 10]           5,010
================================================================
Total params: 431,080
Trainable params: 431,080
Non-trainable params: 0
----------------------------------------------------------------
Input size (MB): 0.00
Forward/backward pass size (MB): 0.12
Params size (MB): 1.64
Estimated Total Size (MB): 1.76
----------------------------------------------------------------
```

Defining the loss function

We will define a loss function and test it on a mini-batch. Let's get started:

```
# 1. First, we will define negative log-likelihood loss:

from torch import nn
loss_func = nn.NLLLoss(reduction="sum")
```

```
# 2. Let's test the loss function on a mini-batch:

for xb, yb in train_dl:
    #move batch to cuda device
    xb = xb.type(torch.float).to(device)
    yb = yb.to(device)
    #get model output
    out = model(xb)
    #calculate loss value
    loss = loss_func(out, yb)
    print(loss.item())
    break
```

```
94.81073760986328
```

```
# 3. Let's compute the gradients with respect to the following output:

#computed gradients
loss.backward()
```

Defining the optimizer

```
# 1. Let's define the Adam optimizer:

from torch import optim
opt = optim.Adam(model.parameters(), lr = 1e-4)
```

```
# 2. Use the following code to update the model parameters:

#Update model parameters
opt.step()
```

```
# 3. Next, we set the gradients to zero:

#set gradients to zero
opt.zero_grad()
```

Training and evaluation

We will develop helper functions for batch and epoch processing and training the model.

```
#1. Let's develop a helper function to compute the loss value per mini-batch:

def loss_batch(loss_dunc, xb, yb, yb_h, opt=None):
    #obtain loss
    loss = loss_func(yb_h, yb)
    #obtain performance metric
    metric_b = metrics_batch(yb, yb_h)
    if opt is not None:
        loss.backward()
        opt.step()
        opt.zero_grad()

    return loss.item(), metric_b
```

```
# 2. Next, we will define a helper function to compute the accuracy per mini-batch:

def metrics_batch(target, output):
    #obtain output class
    pred = output.argmax(dim=1, keepdim=True)
    #compare output class with target class
    corrects = pred.eq(target.view_as(pred)).sum().item()
    return corrects
```

Training and evaluation

```python
# 3. Next, we will define a helper function to compute the loss
# and metric values for a dataset:

def loss_epoch(model, loss_func, dataset_dl, opt=None):
    loss = 0.0
    metric = 0.0
    len_data = len(dataset_dl.dataset)
    for xb, yb in dataset_dl:
        xb = xb.type(torch.float).to(device)
        yb = yb.to(device)
        #obtain model output
        yb_h = model(xb)

        loss_b, metric_b = loss_batch(loss_func, xb, yb, yb_h, opt)
        loss += loss_b
        if metric_b is not None :
            metric += metric_b
    loss/=len_data
    metric/=len_data
    return loss, metric
```

```python
# 4. Finally, we will define the train_val function:

def train_val(epochs, model, loss_func, opt, train_dl, val_dl):
    for epoch in range(epochs):
        model.train()
        train_loss,train_metric = loss_epoch(model, loss_func, train_dl, opt)
        model.eval()
        with torch.no_grad():
            val_loss, val_metric = loss_epoch(model, loss_func,val_dl)
        accuracy = 100 * val_metric

        print("epoch : %d, train loss : %.6f, val loss : %.6f, acuuracy: %.2f"
              %(epoch, train_loss, val_loss, accuracy))
```

```python
# 5. Let's train the model for a few epochs:

#call train_val function
num_epochs=5
train_val(num_epochs, model, loss_func, opt, train_dl, val_dl)
```

```
epoch : 0, train loss : 0.151273, val loss : 0.071866, acuuracy: 97.56
epoch : 1, train loss : 0.048157, val loss : 0.049809, acuuracy: 98.40
epoch : 2, train loss : 0.028318, val loss : 0.051704, acuuracy: 98.64
epoch : 3, train loss : 0.019180, val loss : 0.078078, acuuracy: 98.32
epoch : 4, train loss : 0.015641, val loss : 0.060697, acuuracy: 98.66
```

Storing and loading models

방법 1

```
#1. First, we will store the model parameters orstate_dict in a file:

# define path2weights
path2weights="./models/weight.pt"

# store state_dict to file
torch.save(model.state_dict(), path2weights)
```

```
#2. To load the model parameters from the file, we will define an object
#of the Net class:

#define model : weights are randomly initiated
_model = Net()
```

```
#3. Then, we will ;oad state_dict from the file:

weights = torch.load(path2weights)
```

```
#4. Next, we will set state_dict to the model :

_model.load_state_dict(weights)
```

```
<All keys matched successfully>
```

Storing and loading models

방법 2

```
#1. First, we will store the model in a file:

# define a path2model
path2model="./models/model.pt"

# store model and weights into local file
torch.save(model,path2model)
```

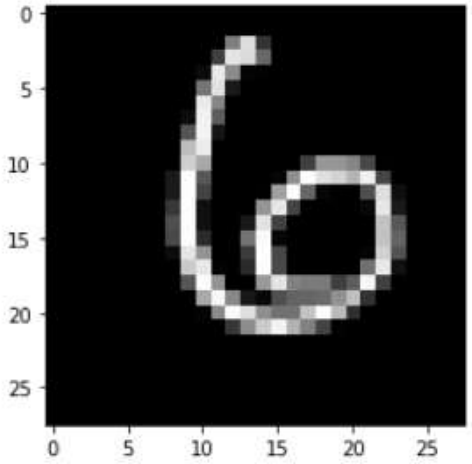
```
#2. To load the model parameters from the file, we will define an object
# of the Net class:

#define model :  weights into a file
_model = Net()
```

```
#3. Then, we will load the model from the local file:

_model = torch.load(path2model)
```

Deploying the model

```
#1. To deploy the model on a sample image from the validation dataset, we will get
# a sample tensor:
n=100
x=x_val[n]
y=y_val[n]
print(x.shape)
plt.imshow(x.numpy()[0],cmap="gray")
```

```
torch.Size([1, 28, 28])

<matplotlib.image.AxesImage at 0x2239a3e87c0>
```



Deploying the model

```
#2. Then, we will preprocess the tensor:

#we use unsqueeze to expand dimension to 1*C*H*W
x = x.unsqueeze(0)

#convert to torch.float32
x = x.type(torch.float)

#move to cuda device
x = x.to(device)
```

```
#3. Next, we will get the model prediction:

#get model output
output = _model(x)

#get predicted class
pred = output.argmax(dim=1, keepdim=True)
print(pred.item(), y.item())
```