Assignment

Tensor & Real world data to Tensor

Yang Dong Jae
2021136150

Deep Learning Homework

September 14, 2023

Before we begin, it's important to understand what deep learning is. This knowledge will be highly beneficial and make it more convenient for me to become familiar with PyTorch's concepts, functions, and so on.

In the realm of deep learning, we often find ourselves spending a significant amount of time working with and manipulating tensor data. This is because the foundational theory of deep learning is rooted in linear algebra. PyTorch is the most popular framework for managing deep learning and handling tensor data.

To facilitate our understanding and practical use of PyTorch, we will be following the guidance provided by the official PyTorch documentation and Professor Han's provided code.

For the reasons mentioned above, I have started to write the code provided by Professor Han in an .ipynb format.

Concept 1

Torch Broad Casting

The term broadcasting describes how Numpy treats arrays with different shapes during arithmetic operations. subject to certain constraints, the smaller array is "**broadcast**" across the larger array so that they have compatible shapes. Broadcasting provides a means of vectorizing array operations so that looping occurs in C instead of Python. It does this without making needless copies of data and usually leads to efficient algorithm implementations. there are, however, cases where broadcasting is a bad idea because it leads to inefficient use of memory that slows computations.

in short, if PyTorch Operations supports broadcast, then its Tensor arguments can be automatically expanded to be of equal sizes without making copies of the data.

Explain.

```
t17 = torch.ones(5, 3, 4, 1)
t18 = torch.rand(3, 1, 1)
print_info(t17 + t18) #shape :  torch.Size([5, 3, 4, 1])
```

according to the above code, what happens in the torch function? In PyTorch, they are using the broadcasting notion to operate + between the different shapes of tensors.

Let's break down what each tensor looks like and what happens when the source code performs the addition 't17 + t18'

a) **'t17'** is defined as a tensor filled with ones with a shape of (5,3,4,1). So it's a **4-dimensional** tensor

b) **'t18'** is defined as a random tensor with a shape of (3,1,1), which is a 3-dimensional tensor.

When someone attempts to add these two tensors together using 't17 + t18', PyTorch will perform broadcasting to make the shapes compatible for element-wise addition.

Broadcasting is a mechanism that allows PyTorch to operate on tensors with different shapes, as long as they can be made compatible through a set of rules such as the above **concept1**.

In this case, PyTorch will broadcast 't18' to have the same shape as 't17' by replicating its values along the dimensions where the size is 1.

broadcasting work in above source code case

- the dimensions of 't18' are expanded to match the dimensions of 't17':
  - 't18' originally has shape (3,1,1)
  - it is expanded to shape (1,3,1,1) to match the shape of 't17' and the broadcasted 't18'.

Resulting shape and explanation

- the shape of 't17' is (5,3,4,1)
- the shape of the broadcasted 't18' is (5,3,4,1) after broadcasting.
- The addition is performed element-wise, so each element in the resulting tensor is the sum of the corresponding elements in 't17' and 't18'.

As a result, the user will get the tensor of shape (5,3,4,1) with values that are the sum of the corresponding elements of 't17' and he broadcasted 't18'

What if different situation? like the code that is located below?

Explain.

```
t25 = torch.empty(5, 2, 4, 1)
t26 = torch.empty(3, 1, 1)
print_info((t25 + t26))
```

```
# RuntimeError: The size of tensor a (2) must match the size of tensor b (3) at non-singleton dimens
```

let's see this example. In this code user attempting to add two PyTorch tensors, 't25' and 't26'. However, they are encountering an issue because the tensors have incompatible shapes for element-wise addition as we saw before.

a) 't25' is created as an empty tensor with shape(5,2,4,1). It is a 4-dimensional tensor

b) 't26' is created as an empty tensor with shape(3,1,1). It is a 3-dimensional tensor.

In order to perform element-wise addition, the shapes of the tensors must be compatible according to PyTorch's broadcasting rules. Broadcasting allows users to perform operations on tensors with different shapes, but **certain conditions must be met:**

- The dimensions of both tensors are compared element-wise from the trailing dimensions(rightmost) to the leading dimensions(leftmost).
- For each dimension, the size of that dimension must be either the same or one of the tensors must have a size of 1 in that dimension.

In this case, when comparing the shapes of 't25' and 't26':

- 't25' has a shape of (5,2,4,1).
- 't26' has a shape of (3,1,1).

Starting from the rightmost dimension:

- the last dimension of 't25' is 1, and the corresponding dimension of 't26' is also 1. So, they are compatible in this dimension.

  However, when moving to the next dimension:

- the dimension of 't25' is 4, but the corresponding dimensions of 't26' is 1. These dimensions are not compatible according to broadcasting rules.
- *When iterating over the dimension sizes, starting at the trailing dimension, the dimension sizes must either be equal, one of them is 1, or one of them does not exist.*
- PyTorch broadcasting rule : https://pytorch.org/docs/stable/notes/broadcasting.html

Concept 2

Squeeze Function

When preparing data for our deep learning model, data preprocessing is an essential process that involves using various functions to reshape the data. and squeeze function is one of the most common function of PyTorch to reshape the data.

Squeeze function returns a tensor with all specified dimensions of input of size 1 removed. For example, if input is of shape : A ×1 ×B ×C ×1 ×D then the input.squeeze() will be of shape : A ×B ×C ×D.

When dim is given, a squeeze operation is done only in the given dimensions(s). if input is of shape (A×1×B), squeeze(input,0) leaves the tensor unchanged, but squeeze(input, 1) will squeeze the tensor to the shape (A ×B)

Explain. parameter of **squeeze** function

- input : The input tensor that user want to remove dimensions from
- 'dim' (Optional) : A specific dimension or dimensions to squeeze. if provided, only the dimensions with size 1 in the specified dimensions will be removed. if not provided, all dimensions with size 1 will be removed.
- 'out' (Optional) : if provided, the result will be placed into this output tensor. this can be useful to avoid unnecessary memory allocation.

```
x = torch.zeros(2, 1, 2, 1, 2)
```

squeeze without any paremeters it will be removed all one-dimension

```
y = torch.squeeze(x)
print(y)
print(y.size())
# torch.Size([2, 2, 2])
```

squeeze with dim = 0 In this case, the dimensions that were fixed by a user using the parameter are 0 but, the 0 index of data is not 1. so it couldn't changed.

```
y = torch.squeeze(x)
print(y)
print(y.size())
# torch.Size([2, 1, 2, 1, 2])
```

squeeze with dim = 1 However, in this case, a squeeze function was gotten dim = 1 through a parameter, and the index of 1 in data is 1 so removed.

```
y = torch.squeeze(x)
print(y)
print(y.size())
# torch.Size([2, 2, 1, 2])
```

squeeze with dim = (multi-index). Even if a squeeze function was gotten dim = (1,2,3) through a parameter, they will remove the element if indexes that gotten from the parameter have one dimension, removed its.

- **squeeze with dim = (1,2,3)**

```
y = torch.squeeze(x)
print(y)
print(y.size())
# torch.Size([2, 2, 2])
```

- **squeeze with dim = (0,1,2)**

```
y = torch.squeeze(x)
print(y)
print(y.size())
# torch.Size([2, 2, 2])
```

Concept 3

Permute Function

When working with data for deep learning models, it is often necessary to manipulate the dimensions of the data to ensure it is in the appropriate format. The `permute` function in PyTorch is a valuable tool for achieving this task.

The `permute` function allows you to rearrange the dimensions of a tensor according to a specified permutation order. It returns a new tensor with the dimensions rearranged as specified. This operation is particularly useful when dealing with data in different formats or when you need to match the input requirements of a neural network.

Explain.

Syntax:

```
output_tensor = input_tensor.permute(*dims)
```

Parameters:

- `input_tensor`: The input tensor that you want to permute.
- `*dims`: A sequence of dimension indices that specify the new order of dimensions in the output tensor. The dimensions are 0-indexed.

Example:
Suppose you have an input tensor with the shape (A, B, C), and you want to permute it to have the shape (C, A, B). You can achieve this using the `permute` function as follows:

```
input_tensor = torch.randn(A, B, C)
output_tensor = input_tensor.permute(2, 0, 1)
```

In this example, the dimensions are rearranged from (A, B, C) to (C, A, B).

```
t18 = torch.randn(2, 3, 5)
print(t18.shape)  # >>> torch.Size([2, 3, 5])
print(torch.permute(t18, (2, 0, 1)).size()
```

In this case too, before applying permute was to t18, the size of it was $2, 3, 5$ but after applying it to permute, the size was changed to $5, 2, 3$

Note:

- The `permute` function creates a new tensor with the specified dimension order and does not modify the original tensor in place.

- You must provide a valid permutation of dimensions; otherwise, you will get an error.
- for those concepts, it is the same operation for using stack function operations

Concept 4

HStack Function for Multi-Dimensional Arrays

In data manipulation and array operations, the `hstack` function is a versatile tool that enables the horizontal stacking of multi-dimensional arrays. It allows you to concatenate arrays along their horizontal axis, effectively extending the size of the arrays in the horizontal direction.

Syntax:

```
output_array = np.hstack(tuple_of_arrays)
```

Parameters:

- `tuple_of_arrays`: A tuple (or sequence) of arrays that you want to horizontally stack together. The arrays must have the same number of dimensions along all axes except the horizontal axis.

Operation:

The `hstack` function combines arrays along their horizontal axis while ensuring that their shapes are compatible. Here's how it works:

- All input arrays must have the same shape along all axes except for the horizontal axis (axis 1).
- The function concatenates the input arrays along the horizontal axis (axis 1), extending the size of the resulting array in the horizontal direction.
- The horizontal stacking operation does not modify the input arrays; it creates a new array as the output.

Example:

Suppose you have two NumPy arrays, `array1` with shape (M, N) and `array2` with shape (M, P). To horizontally stack these arrays, you can use the `hstack` function as follows:

```
import numpy as np

array1 = np.random.rand(M, N)
array2 = np.random.rand(M, P)
stacked_array = np.hstack((array1, array2))
```

In this example, the `hstack` function will create a new array with shape $(M, N + P)$ by horizontally stacking `array1` and `array2`.

Note:

- The `hstack` function is particularly useful when dealing with multi-dimensional arrays, as it simplifies the process of combining arrays horizontally.
- It is essential to ensure that the input arrays have compatible shapes along all axes except the horizontal axis.

- The operation performed by `hstack` is reversible, and you can use it to both stack and unstack arrays horizontally.

HW 1-1 result link src link: https://nbviewer.org/github/YangDongJae/DeepLearning/blob/main/src/homework%231/hw1_1.ipynb

In order to utilize real-world data for deep learning, we need to transform that data into a computational format through processes such as vectorization and encoding. This preprocessing step was discussed in our previous report, where we learned how to prepare data for deep learning sessions.
Following preprocessing, our next task is to understand what constitutes real-world data and how to adapt it to fit the structure of deep learning models. We must explore the relationship between real-world data and the architectural aspects of deep learning, such as network design and model configuration. This understanding is crucial for effectively applying deep learning to real-world problems.

Data Type	Data Size	Number of Channels	Depth	Height	Width	Features	Hours	Frequency	Time
IMAGES	N	C		H	W				
3D IMAGES	N	C (usually 1)	D	H	W				
TABULAR DATA	N					F			
TIME SERIES DATA	N					F	L		
AUDIO DATA	N	C							L
AUDIO DATA (F,T)	N	C						F	T
VIDEO DATA	N	C		H	W				T

Figure 1: Real-world data structure to tensor

Following this notion, it is essentially important things to convert data from origin to torch structure as indicated above. Let's see some examples of it.

2D Image Data

We can use many tools e.g. imageio or cv2 or numpy, etc when we road an image to use deep learning. When we road image data, what will be represented generally? and what is it constituted?

- constitutions is H ×W ×C
  - H : Height
  - W : Width
  - C : Channel

2D Image Structure is C ×H ×W

When We convert it from the original data type to tensor type we can approach in a lot of methods, but mainly we use Numpy method and after preprocessing it, convert it to tensor or use method indicated in hw1-1.

for example of converting to tensor

```
img = torch.from_numpy(img_arr)
out = img.permute(2, 0, 1)
print(out.shape)
# H x W x C -> C x H x W
```

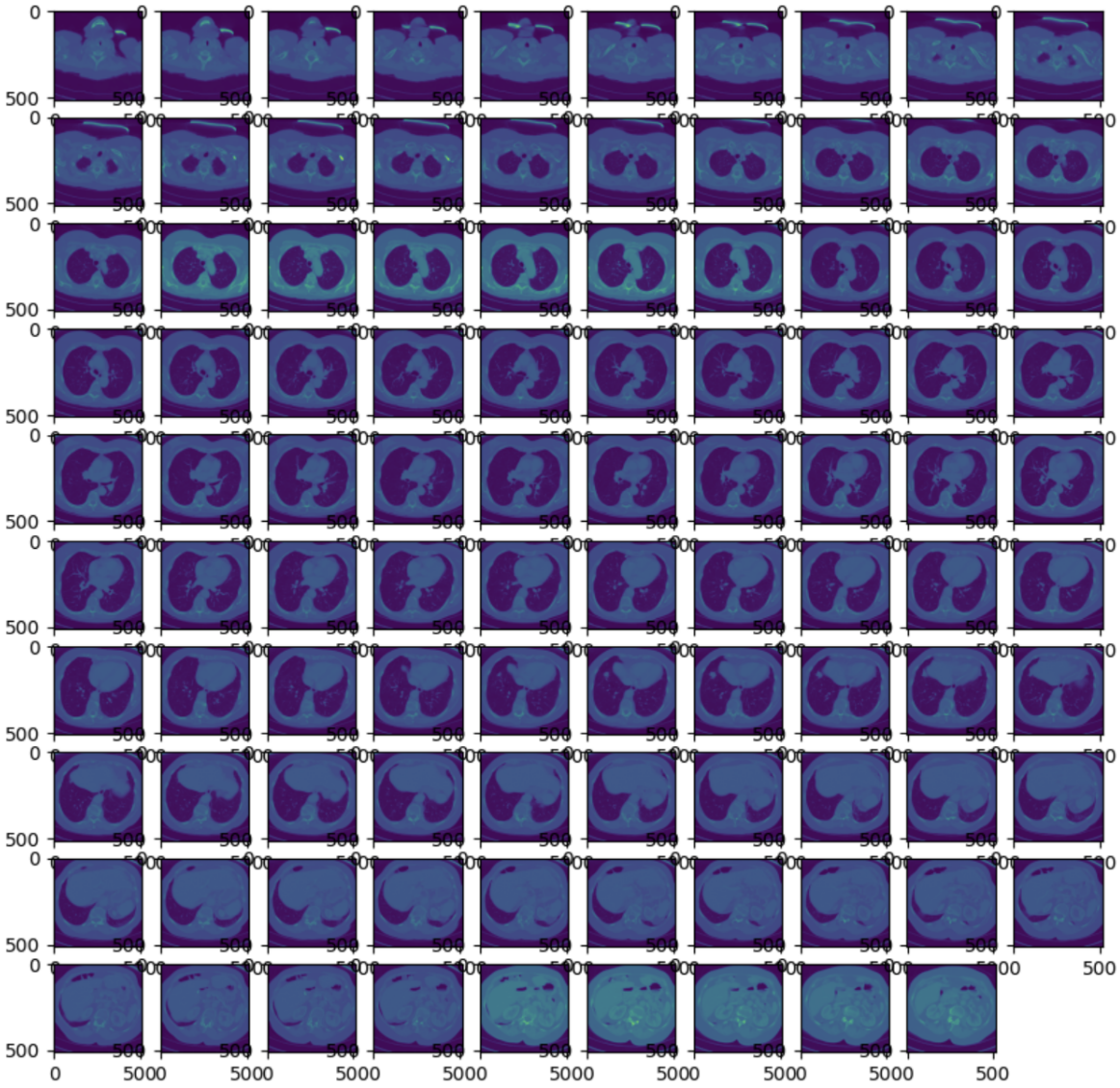
3D Image Data

3D Image process is also same, it depends on a different method that preprocesses the data and different structures. after loading data and check shape of them.

```
    dir_path = os.path.join(os.path.pardir, os.path.pardir, "_00_data", "c_volumetric-dicom", "2-
LUNG_3.0_B70f-04083")
vol_array = imageio.volread(dir_path, format='DICOM')

    print(vol_array.shape) # (99,512,512)
```

Visualized 3D Image



In this case, the shape of data is 99,512,512. but we have to follow the structure of tensor for 3D Image(N ×C ×D ×H ×W) so we modify it using torch functions like below. **for example of converting 3D data to tensor**

```
vol = torch.unsqueeze(vol, 0)  # channel
vol = torch.unsqueeze(vol, 0) #data size
print(vol.size()) #shape :  torch.Size([1, 1, 99, 512, 512])
```

And the other real-world data type need to modify their original structure to their tensor structure that mentioned above like 2D, 3D Image as we saw above.

PyTorch Data set DataLoader

essentially, we have to modify huge data in the world to tensor. on that time we use many framwork and these days, most popular framwork is PyTorch for deep leraning.

And we use this framework with different structure of data that we gotten from real world with cammera or satellite , etc we have to modify it to right structure of pytorch following the rule which was mentioned above to chart. On that moment it is need to implement own class to load the dataset.

Is there a convention or implicit rule, similar to the structure of torch tensors, for creating a class or function to load datasets? what if this is true, how can we? Let's see

Notion 1

troch.utils.data.Dataset `class torch.utils.data.Dataset(data_source, transform=None, target_transform=None)`

this is sotred data samples and expected target values(labels) and return one sample at a time
Parameters:

- `data_source` (`Any`) - The source of data for the dataset. This can be a list, NumPy array, or any other data source that can be indexed like a list.
- `transform` (`callable, optional`) - A function/transform that takes in a sample from the dataset and returns a transformed version of it. This is typically used for data augmentation or preprocessing.
- `target_transform` (`callable, optional`) - A function/transform that takes in the target (label) of a sample and returns a transformed version of it.

Usage:

To create a custom dataset using 'torch.utils.data.Dataset', you typically create a subclass of it and override two essential methods:

a) `__len__()`: This method should return the size of the dataset.

b) `__getitem__(idx)`: This method should return the item (sample) at the specified index `idx`.

Example of a simple custom datset class for loading Image data

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

class CustomImageDataset(Dataset):
    def __init__(self, data_source, transform=None, target_transform=None):
        self.data = data_source
        self.transform = transform
        self.target_transform = target_transform
```

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

def __getitem__(self, idx):
    sample = self.data[idx]

    if self.transform:
        sample = self.transform(sample)

    return sample
```

Above code, each method indicate below. And we couldn't forget when we use it, we have to use Dataset class for supeclass of our custom dataset class.

- `__init__`
  - Read data & Target
  - preprocess them
- `__len__`
  - return the size of the dataset
- `__getitem__`
  - Return one sample at a time, if it doesn't no return.

Notion 2

`class torch.utils.data.DataLoader(dataset, batch_size=1, shuffle=False, num_workers=0, collate_fn=None, pin_memory=False, drop_last=False, timeout=0, worker_init_fn=None)`

Parameters:

- `dataset` (`Dataset`) - The dataset you want to load using the DataLoader.
- `batch_size` (`int, optional`) - The number of samples per batch to load. Default is 1.
- `shuffle` (`bool, optional`) - Whether to shuffle the dataset's samples at the beginning of each epoch. Default is False.
- `num_workers` (`int, optional`) - The number of subprocesses to use for data loading. Default is 0 (no subprocesses).
- `collate_fn` (`callable, optional`) - A function to collate multiple samples into a batch. If not provided, the default collate function is used.
- `pin_memory` (`bool, optional`) - If True, data is pinned into memory, which can speed up data transfer to GPU. Default is False.

- `drop_last` (`bool, optional`) - If True, drops the last batch if its size is less than `batch_size`. Default is False.
- `timeout` (`float, optional`) - If positive, it specifies the maximum waiting time for each worker to fetch a batch. Default is 0, which means no timeout.
- `worker_init_fn` (`callable, optional`) - A function to run on each worker subprocess when it starts. Useful for initializing worker-specific states.

parameters which were located above, some of them is useful when we load data. thus some parameters use a lot. So it is helpful to rubberneck it. let's break it.

- shuffle
- num_wokers=N
- drop_last
- pin_memory

Example

```
from torchvision import transforms

class DogCat2DImageDataset(Dataset):
  def __init__(self):
    self.image_transforms = transforms.Compose([
      transforms.Resize(size=(256, 256)),
      transforms.ToTensor()
    ])

    dogs_dir = os.path.join(os.path.pardir, os.path.pardir, "_00_data", "a_image-dog")
    cats_dir = os.path.join(os.path.pardir, os.path.pardir, "_00_data", "b_image-cats")

    image_lst = [
      Image.open(os.path.join(dogs_dir, "bobby.jpg")),  # (1280, 720, 3)
      Image.open(os.path.join(cats_dir, "cat1.png")),  # (256, 256, 3)
      Image.open(os.path.join(cats_dir, "cat2.png")),  # (256, 256, 3)
      Image.open(os.path.join(cats_dir, "cat3.png"))  # (256, 256, 3)
    ]

    image_lst = [self.image_transforms(img) for img in image_lst]
    self.images = torch.stack(image_lst, dim=0)

    # 0: "dog", 1: "cat"
    self.image_labels = torch.tensor([[0], [1], [1], [1]])

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

  def __getitem__(self, idx):
```

```
    return {'input': self.images[idx], 'target': self.image_labels[idx]}

  def __str__(self):
    str = "Data Size: {0}, Input Shape: {1}, Target Shape: {2}".format(
      len(self.images), self.images.shape, self.image_labels.shape
    )
    return str
```

Check Point!

- custom class `DogCat2DImageDataset` was inherited Dataset as superclass?
- in `__init__` function, it was defined for ground Truth value?
- `__len__` function is return size of dataset?
- `__getitem__` is return one sample at a time?

The above two concepts that we use in data loading are really important things before we start running and designing neural network models. especially, the structure of the custom DataLoader class is an implicit standard of PyTorch that is used widely in the world.
so it is really important things to understand DataClass rules and then when we develop our own designed deep learning model, we develop a data loader that follows the rule of data loader in PyTorch.

Homework Reflection & Conclusion

For the first homework, we reviewed 3weeks of lectures especially on what are tensors and how can we transform from real-world datasets to tensors with PyTorch.

under the solve this homework, I think the most important part of it were belowed.

- we have to know intuitively when torch shape was provided for us.
- we have to be familliar with and full command to modify dimensions of tensor.
- when we build our model we have to follow the rule of PyTorch data loader and data set

So, Throughout this homework assignment, I delved into the fundamental concepts of tensors, their shapes, and how to manipulate them using PyTorch. This exploration was crucial in developing a strong foundation for working with real-world datasets and building models effectively. Here are my key takeaways and reflections:

a) Understanding Tensor Shapes: One of the most critical aspects of this assignment was developing an intuitive understanding of tensor shapes. Recognizing and interpreting the shape of tensors provided in real-world datasets is essential as it sets the stage for data processing and model building. This involves identifying the number of dimensions (rank) and the size of each dimension.

b) Dimension Manipulation: The ability to modify the dimensions of tensors is a fundamental skill. I learned how to reshape tensors, add or remove dimensions, and transpose them to suit the requirements of specific tasks. This flexibility is crucial for data preprocessing and model compatibility.

c) PyTorch DataLoader and Dataset: Building models in PyTorch involves adhering to the rules and practices of PyTorch's DataLoader and Dataset. These components ensure efficient data loading, transformation, and batching. Understanding how to structure these elements correctly is essential for seamless model training and evaluation.

d) Real-world Data Transformation: I gained valuable insights into the process of converting real-world data into tensor format. Each type of data structure, such as images, text, or time series data, may require unique preprocessing techniques. Being able to apply these transformations effectively is vital for working with diverse datasets. and on that time, what if use PyTorch, we have to be able to transform a tensor structure of PyTorch.

e) Conceptual Understanding: Before diving into practical code, I realized the importance of grasping the fundamental concepts of vector tensors and dimensions. Developing an intuitive understanding of these concepts laid the groundwork for effectively manipulating tensors using PyTorch's powerful tools.

In conclusion, this homework assignment has been a valuable learning experience. I have gained a deeper understanding of tensors, their shapes, and their manipulation in PyTorch. Additionally, I have honed my skills in transforming real-world data into tensors, adhering to PyTorch's DataLoader and Dataset guidelines, and continuously improving problem-solving abilities. These skills will undoubtedly prove invaluable as I continue to explore the exciting world of deep learning and data science.

숙제후기

Studying and understanding the case of how we control tensors with PyTorch was really useful. and it is really essential things when we study deep learning. because what if we couldn't control the data, we can't do anything in deep learning. So as I said above `reflection & Conclusion` it is a really important thing and improved me.

however, I feel that I believe that the primary role of colleges and universities is to advance education through research. Therefore, I think it is important for students to become familiar with the usage of functions when they first start learning about Machine Learning and Deep Learning. However, for many of us, this is not the first encounter with Machine Learning and Deep Learning theory. According to our curriculum, we are expected to already know concepts such as tensors, NumPy, and how to convert real-world data into tensors.

In my opinion, it might be more beneficial to dedicate more time to theoretical aspects since we can study the practical usage of frameworks online as needed. This doesn't mean that learning how to use frameworks is unimportant; it certainly helps when we are developing models. However, for students following the curriculum designed by the Department of Computer Science and Engineering, it might feel like a waste of time.

For these reasons, I believe that it would be more valuable to prioritize the understanding of theoretical concepts, delving deep into the formulas used in Deep Learning models. Once we grasp the significance of these formulas, we can then apply this knowledge to write code using tools like PyTorch. This proficiency serves as a powerful asset in the competitive AI job market.