Recitation 0d

Dataloaders, Reading & Saving

Contents

- Dataset
- Dataloaders
- Reading
- Saving

Reading and Saving

```
import numpy as np
tmp_array = np.ones((3,3))
tmp_array_pkl = np.array([[0,1],[2,3,4],[5,6,7,8]], dtype=object)

# npy
np.save("tmp_array.npy", tmp_array)
np.save("tmp_array_pkl.npy", tmp_array_pkl, allow_pickle=True)
read_array = np.load("tmp_array.npy")
read_array_pkl = np.load("tmp_array_pkl.npy", allow_pickle=True)
print(read_array)
print(read_array_pkl)
```

Reading and Saving

```
# csv
import pandas as pd

output = pd.DataFrame()
output['index'] = np.array(range(10))
output['label'] = np.array(range(10,20))
print(output.head())
output.to_csv("submission.csv", index = False)

output_read = pd.read_csv("submission.csv")
print(output_read.head())
```

Dataset & Dataloader

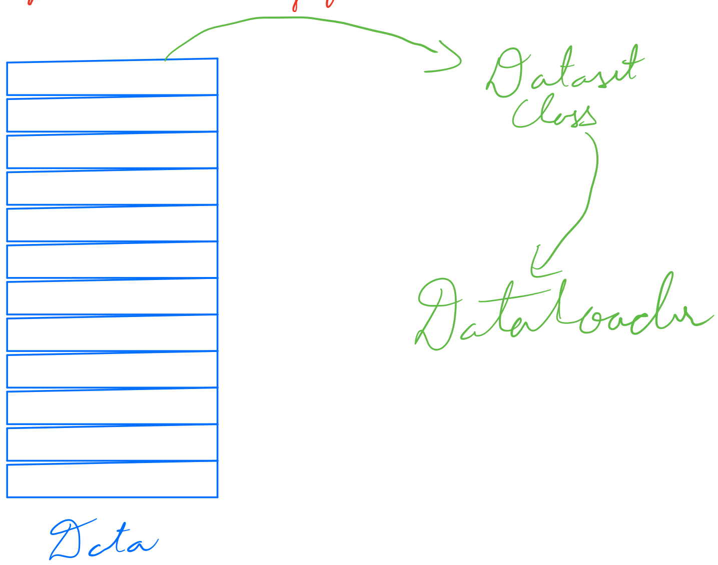
- Deep learning models typically require a lot of data.
- We process & batch this data.
- To make this process efficient Pytorch provides us with Dataset & Dataloader functionality.

⇒ Dataset class ← Training Data.

↓

training_data_object ⟶ Dataloader

Dataloader → Makes batches, shuffles & does parallel processing to make the process efficient.



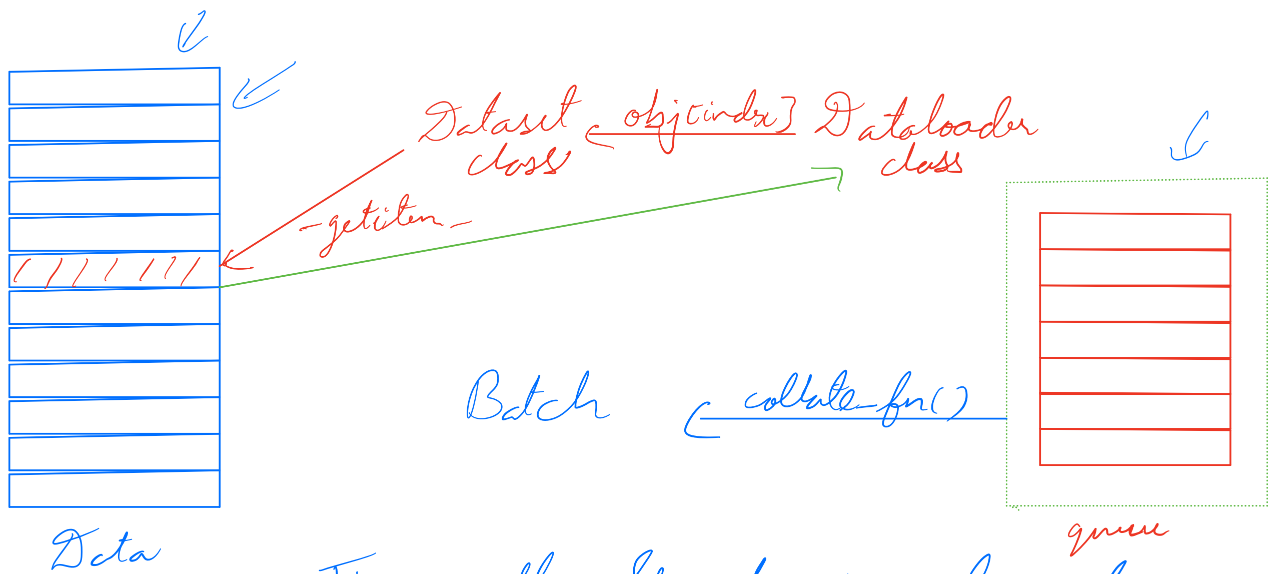
- Dataset class is fed to Dataloader
- Dataloader class uses Dataset class to retrieve data.

Credits: PracticalML

Dataset Class: Has access to the data.
Dataloader Class: Loads the data with the help of dataset class.

For this to work dataset class must have two methods.

- _len_() → len() ← returns length
- _getitem_() → [ ] ← returns processed data given index.



- There will also be some transforms etc.

Some terminology.

- Epoch: One forward & backward pass of all training examples.
- Batch_size: No. of training examples in one batch.
- Iterations: Number of passes. (1 pass → forward + backward)

⇒ Moving onto code & colab notebook later to consolidate this information better

DataSet

```
from torch.utils import data
from torch.utils.data import Dataset, DataLoader
class MyDataset(data.Dataset):
    def __init__(self, X, Y):
        self.X = X
        self.Y = Y

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

    def __getitem__(self, index):
        X = self.X[index].float().reshape(-1) # fla
        Y = self.Y[index].long()
        return X,Y
```

Use __init__ to load in the data to the class (or preprocess) so it can be accessed later

Pytorch will use __len__ to know how many (x, y) pairs (training samples) are in your dataset

After using __len__ to figure out how many samples there are, pytorch will use __getitem__ to ask for a certain sample. So, __getitem__(i) should return the "i-th" sample, with order chosen by you. You should use __getitem__ to do some final processing on the data before it's sent out.

Caution: __getitem__ will be called maybe millions of times, so make sure you do **as little work** in here as possible for fast code. Try to keep heavy preprocessing in __init__, which is only called once.

DataLoaders

```
num_workers = 8 if cuda else 0

# Training
train_dataset = MyDataset(train.train_data, train.train_labels)
train_loader_args = dict(shuffle=True, batch_size=256, num_workers=num_workers, pin_memory=True) if cuda\
                    else dict(shuffle=True, batch_size=64)
train_loader = data.DataLoader(train_dataset, **train_loader_args)
```

The dataset you made before

Just something that makes dataloading faster at the expense of more RAM usage.

These are the arguments we're passing to **Training** DataLoader

We'll be going through the entire dataset multiple times. We want to shuffle the Dataset every single time **for the training dataloader.**

For validation/test, you don't want to shuffle.

Notice that we give our dataset to the DataLoader so it can use it.

How many samples per batch? This is a hyperparameter you want to adjust.

Batches are loaded in parallel – how many workers do you want doing this? Depending on how intensive __getitem__ is, lowering or raising this may speed up dataloading.

Credits

- Previous iterations of IDL
- Youtube