DataLoader

In the previous notebook you have implemented a dataset that we can now use to access our data. However, in machine learning, we often need to perform a few additional data preparation steps before we can start training models.

An important additional class for data preparation is the **DataLoader**. By wrapping a dataset in a dataloader, we will be able to load small subsets of the dataset at a time, instead of having to load each sample separately. In machine learning, the small subsets are referred to as **mini-batches**, which will play an important role later in the lecture.

In this notebook, you will implement your own dataloader, which you can then use to load mini-batches from the dataset you implemented previously.

First, you need to import libraries and code, as always.

In [1]:
```
import numpy as np

from exercise_code.data import DataLoader, DummyDataset
from exercise_code.tests import (
    test_dataloader,
    test_dataloader_len,
    test_dataloader_iter,
    save_pickle,
    load_pickle
)

%load_ext autoreload
%autoreload 2
```

1. Iterating over a Dataset

Throughout this notebook a dummy dataset will be used that contains all even numbers from 2 to 100. Similar to the dataset you have implemented before, the dummy dataset has a `__len__()` method that allows us to call `len(dataset)`, as well as a `__getitem__()` method, which allows you to call `dataset[i]` and returns a dict `{"data": val}` where `val` is the i-th even number. If you would like to see the code, have a look at `DummyDataset` in `exercise_code/data/base_dataset.py`.

Let's start by defining the dataset, and calling its methods to get a better feel for it.

In [2]:
```
dataset = DummyDataset(
    root=None,
    divisor=2,
    limit=100
)
print(
    "Dataset Length:\t", len(dataset),
    "\nFirst Element:\t", dataset[0],
    "\nLast Element:\t", dataset[-1],
)
```

```
Dataset Length:  50
First Element:   {'data': 2}
Last Element:    {'data': 100}
```

In the following, you will write some code to iterate over the dataset in mini-batches, similarly to what a dataloader is supposed to do. The number of samples to load per mini-batch is called **batch size**. For the remainder of this notebook, the batch size is 3.

In [3]:
```
batch_size = 3
```

Let us now define a simple function that iterates over the dataset and groups samples into mini-batches:

In [4]:
```
def build_batches(dataset, batch_size):
    batches = []  # list of all mini-batches
    batch = []  # current mini-batch
    for i in range(len(dataset)):
        batch.append(dataset[i])
        if len(batch) == batch_size:  # if the current mini-batch is ful
l,
            batches.append(batch)  # add it to the list of mini-batches,
            batch = []  # and start a new mini-batch
    return batches

batches = build_batches(
    dataset=dataset,
    batch_size=batch_size
)
```

Let's have a look at the mini-batches:

In [5]:
```
def print_batches(batches):
    for i, batch in enumerate(batches):
        print("mini-batch %d:" % i, str(batch))

print_batches(batches)
```

```
mini-batch 0: [{'data': 2}, {'data': 4}, {'data': 6}]
mini-batch 1: [{'data': 8}, {'data': 10}, {'data': 12}]
mini-batch 2: [{'data': 14}, {'data': 16}, {'data': 18}]
mini-batch 3: [{'data': 20}, {'data': 22}, {'data': 24}]
mini-batch 4: [{'data': 26}, {'data': 28}, {'data': 30}]
mini-batch 5: [{'data': 32}, {'data': 34}, {'data': 36}]
mini-batch 6: [{'data': 38}, {'data': 40}, {'data': 42}]
mini-batch 7: [{'data': 44}, {'data': 46}, {'data': 48}]
mini-batch 8: [{'data': 50}, {'data': 52}, {'data': 54}]
mini-batch 9: [{'data': 56}, {'data': 58}, {'data': 60}]
mini-batch 10: [{'data': 62}, {'data': 64}, {'data': 66}]
mini-batch 11: [{'data': 68}, {'data': 70}, {'data': 72}]
mini-batch 12: [{'data': 74}, {'data': 76}, {'data': 78}]
mini-batch 13: [{'data': 80}, {'data': 82}, {'data': 84}]
mini-batch 14: [{'data': 86}, {'data': 88}, {'data': 90}]
mini-batch 15: [{'data': 92}, {'data': 94}, {'data': 96}]
```

As you can see, the iteration works, but the output is not very pretty. Let us now write a simple function that combines the dictionaries of all samples in a mini-batch.

In [6]:
```
def combine_batch_dicts(batch):
    batch_dict = {}
    for data_dict in batch:
        for key, value in data_dict.items():
            if key not in batch_dict:
                batch_dict[key] = []
            batch_dict[key].append(value)
    return batch_dict

combined_batches = [combine_batch_dicts(batch) for batch in batches]
print_batches(combined_batches)
```

```
mini-batch 0: {'data': [2, 4, 6]}
mini-batch 1: {'data': [8, 10, 12]}
mini-batch 2: {'data': [14, 16, 18]}
mini-batch 3: {'data': [20, 22, 24]}
mini-batch 4: {'data': [26, 28, 30]}
mini-batch 5: {'data': [32, 34, 36]}
mini-batch 6: {'data': [38, 40, 42]}
mini-batch 7: {'data': [44, 46, 48]}
mini-batch 8: {'data': [50, 52, 54]}
mini-batch 9: {'data': [56, 58, 60]}
mini-batch 10: {'data': [62, 64, 66]}
mini-batch 11: {'data': [68, 70, 72]}
mini-batch 12: {'data': [74, 76, 78]}
mini-batch 13: {'data': [80, 82, 84]}
mini-batch 14: {'data': [86, 88, 90]}
mini-batch 15: {'data': [92, 94, 96]}
```


This looks much more organized.

To perform operations more efficiently later, we would also like the values of the mini-batches to be contained in a numpy array instead of a simple list. Let's briefly write a function for that:

In [7]:
```
def batch_to_numpy(batch):
    numpy_batch = {}
    for key, value in batch.items():
        numpy_batch[key] = np.array(value)
    return numpy_batch

numpy_batches = [batch_to_numpy(batch) for batch in combined_batches]
print_batches(numpy_batches)
```

```
mini-batch 0: {'data': array([2, 4, 6])}
mini-batch 1: {'data': array([ 8, 10, 12])}
mini-batch 2: {'data': array([14, 16, 18])}
mini-batch 3: {'data': array([20, 22, 24])}
mini-batch 4: {'data': array([26, 28, 30])}
mini-batch 5: {'data': array([32, 34, 36])}
mini-batch 6: {'data': array([38, 40, 42])}
mini-batch 7: {'data': array([44, 46, 48])}
mini-batch 8: {'data': array([50, 52, 54])}
mini-batch 9: {'data': array([56, 58, 60])}
mini-batch 10: {'data': array([62, 64, 66])}
mini-batch 11: {'data': array([68, 70, 72])}
mini-batch 12: {'data': array([74, 76, 78])}
mini-batch 13: {'data': array([80, 82, 84])}
mini-batch 14: {'data': array([86, 88, 90])}
mini-batch 15: {'data': array([92, 94, 96])}
```

Lastly, we would like to make the loading a bit more memory efficient. Instead of loading the entire dataset into memory at once, let us only load samples when they are needed. This can also be done by building a Python generator, using the `yield` keyword. See https://wiki.python.org/moin/Generators (https://wiki.python.org/moin/Generators) for more information on generators.

In [8]:
```
def build_batch_iterator(dataset, batch_size, shuffle):
    if shuffle:
        index_iterator = iter(np.random.permutation(len(dataset)))  # de
fine indices as iterator
    else:
        index_iterator = iter(range(len(dataset)))  # define indices as
 iterator

    batch = []
    for index in index_iterator:  # iterate over indices using the itera
tor
        batch.append(dataset[index])
        if len(batch) == batch_size:
            yield batch  # use yield keyword to define a iterable genera
tor
            batch = []

batch_iterator = build_batch_iterator(
    dataset=dataset,
    batch_size=batch_size,
    shuffle=True
)
batches = []
for batch in batch_iterator:
    batches.append(batch)

print_batches(
    [batch_to_numpy(combine_batch_dicts(batch)) for batch in batches]
)
```

```
mini-batch 0: {'data': array([18, 26, 82])}
mini-batch 1: {'data': array([52, 92, 66])}
mini-batch 2: {'data': array([ 4, 44, 20])}
mini-batch 3: {'data': array([76, 80, 22])}
mini-batch 4: {'data': array([90, 40,  6])}
mini-batch 5: {'data': array([58, 10, 94])}
mini-batch 6: {'data': array([36, 74, 64])}
mini-batch 7: {'data': array([34,  8, 62])}
mini-batch 8: {'data': array([12, 32, 56])}
mini-batch 9: {'data': array([42, 72, 28])}
mini-batch 10: {'data': array([86, 46, 14])}
mini-batch 11: {'data': array([88, 84, 50])}
mini-batch 12: {'data': array([54, 78, 98])}
mini-batch 13: {'data': array([24, 38, 68])}
mini-batch 14: {'data': array([48, 70, 30])}
mini-batch 15: {'data': array([100,   2,  16])}
```


The functionality of the cell above is now pretty close to what the dataloader is supposed to do. However, there are still two remaining issues:

1. The last two samples of the dataset are not contained in any mini-batch. This is because the number of samples in the dataset is not dividable by the batch size, so there are a few left-over samples which are implicitly discarded. Ideally, an option would be prefered that allows you to decide how to handle these last samples.
2. The order of the mini-batches, as well as the fact which samples are grouped together, is always in increasing order. Ideally, there should be another option that allows you to randomize which samples are grouped together. The randomization could be easily implemented by randomly permuting the indices of the dataset before iterating over it, e.g. using `indices = np.random.permutation(len(dataset))`.

2. DataLoader Class Implementation

Now it is your turn to put everything together and implement the DataLoader as a proper class. We provide you with a basic skeleton for this, which you can find in `class DataLoader` of `exercise_code/data/image_folder_dataset.py`. Open the file and have a look at the class. Note that the `__init__` method receives four arguments:

- **dataset** is the dataset that the dataloader should load.
- **batch_size** is the mini-batch size, i.e. the number of samples you want to load at a time.
- **shuffle** is binary and defines whether the dataset should be randomly shuffled or not.
- **drop_last**: is binary and defines how to handle the last mini-batch in your dataset. Specifically, if the amount of samples in your dataset is not dividable by the mini-batch size, there will be some samples left over in the end. If `drop_last=True`, we simply discard those samples, otherwise we return them together as a smaller mini-batch.

Task: Implement

Implement the `__len__(self)` method in `exercise_code/data/dataloader.py`.

Hint: Don't forget to think about drop_last! We will test for both modes.

In [9]:
```
dataloader = DataLoader(
    dataset=dataset,
    batch_size=batch_size,
    shuffle=True
)

_ = test_dataloader_len(
    dataloader=dataloader
)
```

```
LenTestInt passed.
LenTestCorrect passed.
Method __len__() using drop_last=True correctly implemented. Tests pass
ed: 2/2

LenTestInt passed.
LenTestCorrect passed.
Method __len__() using drop_last=False correctly implemented. Tests pas
sed: 2/2

Method __len__() correctly implemented. Tests passed: 4/4
Score: 100/100
```

Task: Implement

Implement the `__iter__(self)` method in `exercise_code/data/dataloader.py`.

Hint: Make use of the code in '1. Iterating over a Dataset' when implementing your `__iter__()` method. We are again testing for both drop_last modes!

In [10]:
```
dataloader = DataLoader(
    dataset=dataset,
    batch_size=batch_size,
    shuffle=True,
)

_ = test_dataloader_iter(
    dataloader=dataloader
)
```

```
IterTestIterable passed.
IterTestItemType passed.
IterTestBatchSize passed.
IterTestNumBatches passed.
IterTestValuesUnique passed.
IterTestValueRange passed.
IterTestShuffled passed.
IterTestNonDeterministic passed.
Method __iter__() using drop_last=True correctly implemented. Tests pas
sed: 8/8

IterTestIterable passed.
IterTestItemType passed.
IterTestBatchSize passed.
IterTestNumBatches passed.
IterTestValuesUnique passed.
IterTestValueRange passed.
IterTestShuffled passed.
IterTestNonDeterministic passed.
Method __iter__() using drop_last=False correctly implemented. Tests pa
ssed: 8/8

Method __iter__() correctly implemented. Tests passed: 16/16
Score: 100/100
```

If you're done, run the cells below to check if your dataloader works as intended. You can change the value of drop_last to see the difference.

In [11]:
```
dataloader = DataLoader(
    dataset=dataset,
    batch_size=batch_size,
    shuffle=True,
    drop_last=False,    # Change here if you want to see the impact of d
rop last and check out the last batch
)
for batch in dataloader:
    print(batch)
```

```
{'data': array([44, 74,  8])}
{'data': array([38, 86, 28])}
{'data': array([14, 82, 96])}
{'data': array([40, 78, 24])}
{'data': array([70, 60,  6])}
{'data': array([52, 76, 56])}
{'data': array([16, 46, 62])}
{'data': array([100,  32,  54])}
{'data': array([ 2, 22, 48])}
{'data': array([20, 18, 36])}
{'data': array([34,  4, 84])}
{'data': array([30, 88, 12])}
{'data': array([50, 90, 98])}
{'data': array([58, 92, 68])}
{'data': array([64, 10, 72])}
{'data': array([80, 94, 66])}
{'data': array([42, 26])}
```

Save your DataLoaders for Submission

Simply save your dataloaders using the following cell. This will save them as well as dataset from the first notebook to a pickle file `cifar_dataset_and_loader.p` .

In [12]:
```
dataloader = DataLoader(
    dataset=dataset,
    batch_size=batch_size,
    shuffle=True,
    drop_last=True,
)

dataset = load_pickle("cifar_dataset.p") # load dataset from the pickle
 file saved in notbook 1

save_pickle(
    data_dict={
        "dataset": dataset['dataset'],
        "cifar_mean": dataset['cifar_mean'],
        "cifar_std": dataset['cifar_std'],
        "dataloader": dataloader
    },
    file_name="cifar_dataset_and_loader.p"
)
```


> **Note**
>
> Note that **this is the ONLY file you need to submit**. Each time you make changes in either `dataset` or `dataloaders` , you need to **rerun the following code** to save your changes for submission.

Submission Instructions

Now, that you have completed the necessary parts in the notebook, you can go on and submit your files.

1. Go on our submission page (https://i2dl.vc.in.tum.de/), register for an account and login. We use your matriculation number and send an email with the login details to the mail account associated. When in doubt, login into tum-online and check your mails there. You will get an id which we need in the next step.
2. Execute the cell below to create a zipped folder for upload.
3. Log into our submission page (https://i2dl.vc.in.tum.de/) with your account details and upload the `zip` file. Once successfully uploaded, you should be able to see the submitted file selectable on the top.
4. Click on this file and run the submission script. You will get an email with your score as well as a message if you have surpassed the threshold.

In [13]:
```
from exercise_code.submit import submit_exercise

submit_exercise('exercise03')
```

```
relevant folders: ['exercise_code', 'models']
notebooks files: ['2_dataloader.ipynb', '1_cifar10-image-dataset.ipyn
b']
Adding folder exercise_code
Adding folder models
Adding notebook 2_dataloader.ipynb
Adding notebook 1_cifar10-image-dataset.ipynb
Zipping successful! Zip is stored under: /Users/meiqiliu/Downloads/exer
cise_03/exercise03.zip
```

Submission Goals

For this exercise we only test your implementations which are tested throughout both notebooks. Here is a list of test cases that will be evaluated on the server using your `ImageFolderDataset` as well as `DataLoader` classes. In total we have 18 test cases where you are required to complete 15 of. Here is an overview split among our two notebooks:

- Goal for **notebook 1**: Implement an ImageFolderDataset with transforms for rescaling and normalizing.
  - To implement:
    1. `exercise_code/data/image_folder_dataset.py`: `ImageFolderDataset - __len__()`, `__getitem()__`
    2. `exercise_code/data/image_folder_dataset.py`: `RescaleTransform`
    3. `exercise_code/data/image_folder_dataset.py`: `compute_image_mean_and_std()`
  - Test cases:
    1. Does `__len__()` of `ImageFolderDataset` return the correct data type?
    2. Does `__len__()` of `ImageFolderDataset` return the correct value?
    3. Does `__getitem()__` of `ImageFolderDataset` return the correct data type?
    4. Does `__getitem()__` of `ImageFolderDataset` load images as numpy arrays with correct shape?
    5. Do values after rescaling with `RescaleTransform` have the correct minimum?
    6. Do values after rescaling with `RescaleTransform` have the correct maximum?
    7. Does `compute_image_mean_and_std()` compute the correct mean?
    8. Does `compute_image_mean_and_std()` compute the correct std?

- Goal for **notebook 2**: Implement a DataLoader that loads mini-batches from a given dataset and supports batch_size, shuffle, and drop_last args.
  - Test cases:
    1. Does `__len__()` return the correct data type?
    2. Does `__len__()` return the correct value?
    3. Does `__iter__()` work at all, i.e. is it possible to iterate over the dataloader?
    4. Does `__iter__()` load the correct data type?
    5. Does `__iter__()` load data with correct batch size?
    6. Does `__iter__()` load the correct number of batches?
    7. Does `__iter__()` load every sample only once?
    8. Does `__iter__()` load the smallest and largest sample from the dataset?
    9. Does `__iter__()` shuffle the data correctly (if necessary)?
    10. Does `__iter__()` return non-deterministic values when shuffling?

- Reachable points [0, 90]: 0 if not implemented, 90 if all tests passed, 5 per passed test
- Threshold to clear exercise: 75
- Submission start: **April 29, 2021 13.00**
- Submission deadline : **May 5, 2021 15.59**.
- You can make multiple submission until the deadline. Your **best submission** will be considered for bonus

Key Takeaways

1. In machine learning, we often need to load data in **mini-batches**, which are small subsets of the training dataset. How many samples to load per mini-batch is called the **batch size**.
2. In addition to the Dataset class, we use a **DataLoader** class that takes care of mini-batch construction, data shuffling, and more.
3. The dataloader is iterable and only loads those samples of the dataset that are needed for the current mini-batch. This can lead to bottlenecks later if you are unable to provide enough batches in time for your upcoming pipeline. This is especially true when loading from HDDs as the slow reading time can be a bottleneck in your complete pipeline later.
4. The dataloader task can easily by distributed amongst multiple processes as well as pre-fetched. When we switch to PyTorch later we can directly use our dataset classes and replace our current Dataloader with theirs :).

Outlook

You have now implemented everything you need to use the CIFAR datasets for deep learning model training. Using your dataset and dataloader, your model training will later look something like the following:

In [14]:
```
dataset = DummyDataset(
    root=None,
    divisor=2,
    limit=200,
)
dataloader = DataLoader(
    dataset=dataset,
    batch_size=3,
    shuffle=True,
    drop_last=True
)
model = lambda x: x
for minibatch in dataloader:
    model_output = model(minibatch)
    # do more stuff... (soon)
```