Parallel Dataloader

Parallel Programming for Machine Learning
Project Work in Artificial Intelligence Programming

Niccolò Arati

28/03/2024

Introduction

A **Dataloader** is a class that provides a flexible and efficient way to load data.
In this work, we will focus on loading batches of images and applying data augmentation techniques on them between batches.

There are two implementations of the Dataloader: one **sequential** and one **parallel**. The purpose of this study is to observe the **speedup** obtained with the parallel version compared to the sequential one.

The programming language is Python, with PyCharm as the IDE, and parallelization was performed using **Multiprocessing**.
All the experiments were conducted on a PC with Windows 10 as the operating system and an Intel Pentium Gold G5400 CPU.

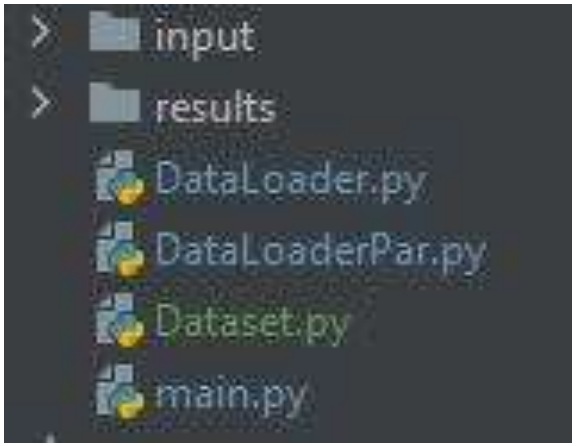
Introduction

Example of Dataloader usage

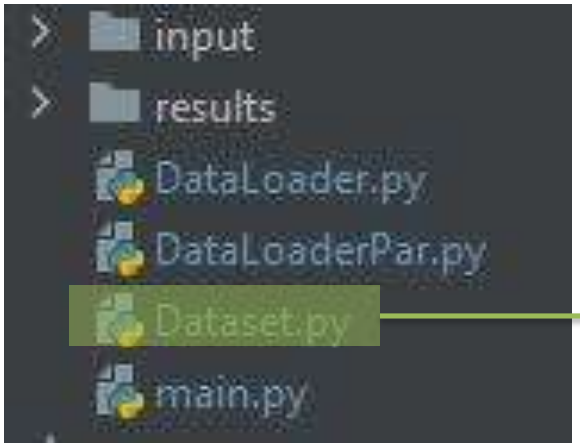
```
transform1 = albumentations.Compose([
    resize.Resize(height = 256, width = 256, p = 1),
    albumentations.RandomCrop(width = 200, height = 200, p = 1)
])
```

```
dataloader = DataLoader.DataLoader(dataset, batch_size = 300)
startSeq = time.time()
for batch in dataloader:
    for image in batch:
        transformed = transform1(image = image)['image']
endSeq = time.time()
resultSeq1 = endSeq - startSeq
```

Code Structure

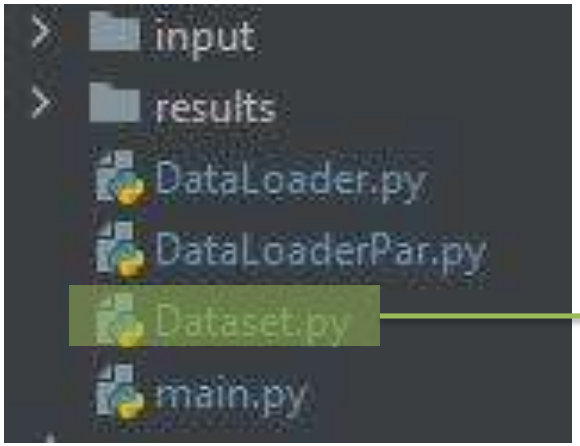


Code Structure



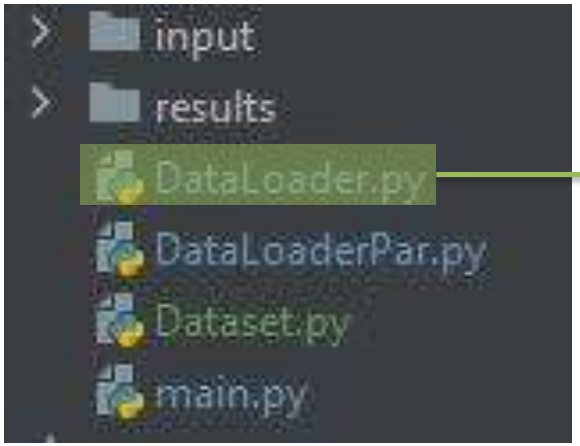
File containing the code related to the Dataset class, which serves as a wrapper for the actual dataset to be loaded via the Dataloader.

Code Structure



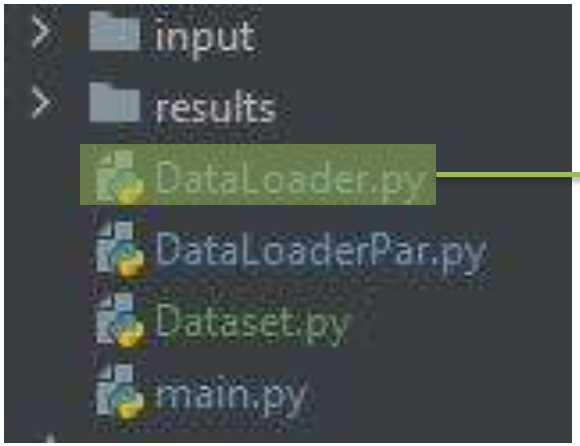

```python
class Dataset:
    def __init__(self, size, im_paths):
        self.size = size
        self.im_paths = im_paths

    def __len__(self):
        return self.size

    def __getitem__(self, index):
        image = cv2.imread(str(self.im_paths[index]))
        image = cv2.cvtColor(image, cv2.COLOR_BGR2RGB)
        return image
```

Code Structure



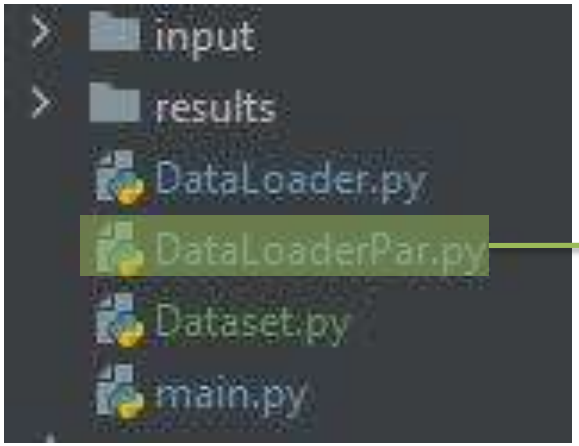
File containing the code related to the DataLoader class, which accesses the dataset of images to be loaded and returns a certain number of them in batches for each iteration.

Code Structure



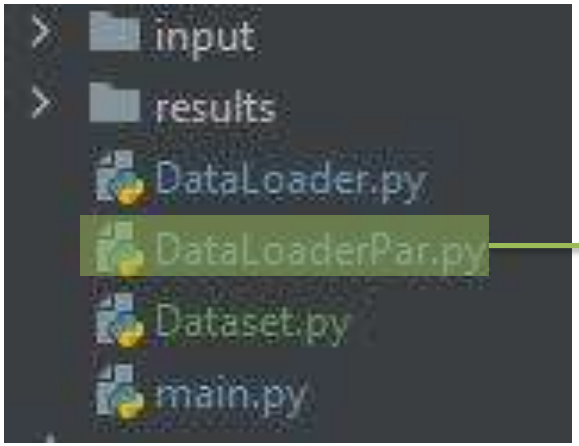

```python
class DataLoader:
    def __init__(self, dataset, batch_size = 1, shuffle = False):
        self.dataset = dataset
        self.batch_size = batch_size
        self.shuffle = shuffle
        self.index = 0 #next index that needs to be loaded

    def __iter__(self):...

    def __next__(self):...

    2 usages (1 dynamic)
    def get(self):...
```

Code Structure



File containing the code related to the DataLoaderPar class, a subclass of DataLoader that implements parallelization of the Dataloader.

Code Structure




```
def worker_funct(dataset, index_queue, output_queue):...


class DataLoaderPar(DataLoader.DataLoader):
    def __init__(self, dataset, batch_size = 64, shuffle = False,
                 num_workers = 1, prefetch_batches = 2):...

    def prefetch(self):...

    def get(self):...

    def __iter__(self):...

    def __del__(self):...
```

Sequential Dataloader

```
def __iter__(self):
    self.index = 0
    return self

def __next__(self):
    if self.index >= len(self.dataset):
        raise StopIteration
    batch_size = min(len(self.dataset) - self.index, self.batch_size)
    batch = tuple([self.get() for _ in range(batch_size)])
    if self.shuffle:
        random.shuffle(batch)
    return batch

def get(self):
    item = self.dataset[self.index]
    self.index += 1
    return item
```

Parallel Dataloader

Body of method __init__()

```
super().__init__(dataset, batch_size, shuffle)

self.num_workers = num_workers
self.prefetch_batches = prefetch_batches
self.output_queue = multiprocessing.Queue()
self.index_queues = []
self.workers = []
self.worker_cycle = itertools.cycle(range(num_workers))
self.cache = {}
self.prefetch_index = 0

for _ in range(num_workers):
    index_queue = multiprocessing.Queue()
    worker = multiprocessing.Process(target = worker_funct,
                                     args = (self.dataset, index_queue,
                                             self.output_queue))
    worker.daemon = True
    worker.start()
    self.workers.append(worker)
    self.index_queues.append(index_queue)

self.prefetch()
```

Parallel Dataloader

Method prefetch() and the associated function with Worker processes

```
def prefetch(self):
    while (self.prefetch_index < len(self.dataset) and
           self.prefetch_index < self.index + 2 * self.num_workers * self.batch_size):
        # if the prefetch_index hasn't reached the end of the dataset &
        # it is not 2 batches ahead, add indexes to the index queues
        self.index_queues[next(self.worker_cycle)].put(self.prefetch_index) #cycles through workers
        self.prefetch_index += 1
```

```
def worker_funct(dataset, index_queue, output_queue):
    while True:
        try:
            index = index_queue.get(timeout=0)
        except queue.Empty:
            continue
        if index is None:
            break
        output_queue.put((index, dataset[index]))
```

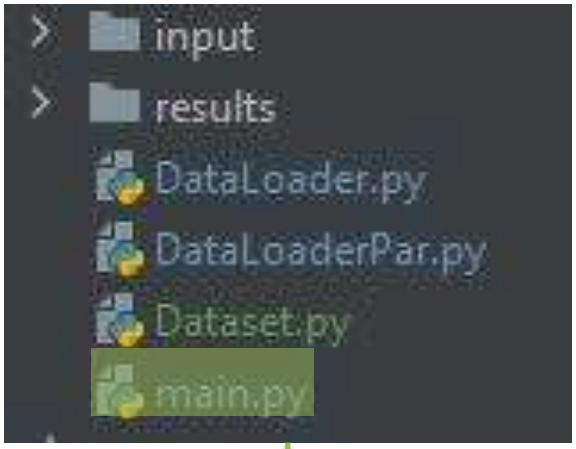
Parallel Dataloader

Method get()

```
def get(self):
    self.prefetch()
    if self.index in self.cache:
        item = self.cache[self.index]
        del self.cache[self.index]
    else:
        while True:
            try:
                (index, data) = self.output_queue.get(timeout = 0)
            except queue.Empty:
                continue
            if index == self.index:
                item = data
                break
            else:
                self.cache[index] = data
```

Test

Executed on a dataset of 110000 images




```
#test different number of workers and albumentations with fixed batch_size of 300
def testDataLoaderWT():...


#test different batch sizes with fixed albumentations and num_workers
def testDataLoaderB():...
```

Tests are located in the file main.py

Test 1

Varying the number of Worker processes and the complexity of the transformations applied to the images with Albumentations, with batch_size = 300.

	1 worker	2 workers	4 workers	6 workers	8 workers
1° Alb.	1. 03472	1. 38500	1. 87490	2. 14634	1. 9381
2° Alb.	3. 55765	5. 02616	6. 39762	7. 16192	4. 50477
3° Alb.	4. 29092	5. 94258	6. 81652	4. 65382	2. 74523
4° Alb.	4. 52743	6. 68475	5. 05657	3. 58080	2. 15300

Test 2

Changing the batch_size with the number of Worker processes set to 2 and fixed image transformations.

	Speedup
batch_size = 10	1. 54175
batch_size = 50	1. 59989
batch_size = 100	1. 56747
batch_size = 300	1. 55132
batch_size = 500	1. 48342
batch_size = 1000	1. 72745

```
transform = albumentations.Compose([
    resize.Resize(height=256, width=256, p=1),
    albumentations.HorizontalFlip(p=1),
    albumentations.RandomCrop(width=200, height=200, p=1),
    albumentations.RandomBrightnessContrast(p=1),
    albumentations.VerticalFlip(p=1)
])
```