Image Augmentation with Python Multiprocessing

[B024314] Parallel Computing Exam

Tommaso Botarelli

23/04/2025

UNIVERSITÀ
DEGLI STUDI
FIRENZE

Da un secolo, oltre.

UNIVERSITÀ DEGLI STUDI FIRENZE
Da un secolo, oltre.

Table of Contents

Introduction

- Crucial technique in **machine learning** and in the field of **computer vision**;
- It involves applying various transformations to images such as rotation, scaling, flipping, cropping, and color adjustments;
- Main goal: artificially expand the size of a dataset;

Image Augmentation: Pros and Cons

Advantages:

- It reduces the risk of overfitting, especially when working with limited data;
- The process for acquiring and labeling the data can be more convenient and less time consuming;
- Particularly useful in deep learning where large datasets are requested;

Disadvantages

As always... **Computationally expensive**!

- Particularly when applied to large datasets or in real-time applications;
- Due to pixel wise operations involved in the image transformations;

Laboratory Objectives

Implement one or more parallel algorithm to speedup augmentation.

Using **Python** and **Multiprocessing** library for the development.

Test several implementations to find the advantages and disadvantages of each method.

UNIVERSITÀ DEGLI STUDI FIRENZE
Da un secolo, oltre.

Table of Contents

2 Image Augmentation

Problem Formalization

2 Image Augmentation

Given an image I from a dataset $\mathcal{D}$, an augmentation function $T : \mathbb{I} \to \mathbb{I}$ applies a transformation such that:

$$I' = T(I)$$

where I' represents the augmented image and $\mathbb{I}$ denotes the space of all possible images. For a dataset $\mathcal{D} = \{I_1, I_2, ..., I_n\}$, augmentation creates an expanded dataset $\mathcal{D}'$ such that:

$$\mathcal{D}' = \{T_k(I) \mid I \in \mathcal{D}, T_k \in \mathcal{T}\}$$

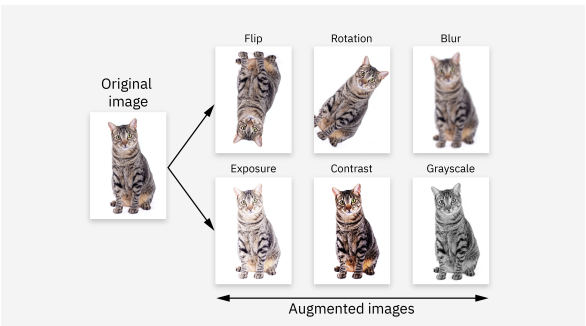
where $\mathcal{T}$ is the set of applied transformations.

Transformations

2 Image Augmentation

The transformation T can be chosen from a set of predefined operations, including:

- **Geometric Transformations**: Rotation, translation, scaling, flipping, cropping, ...
- **Photometric Transformations**: Changes in brightness, contrast, saturation, ...



Transformations: Examples

2 Image Augmentation

(a): Original image

(b)

(c)

(d)

(e)

(f)

Why Python?

2 Image Augmentation

- Python is largely used in Machine Learning tasks;
- Python is simple to use;
- Python abstracts from the low level problems 😕
- Python provides several libraries for these types of tasks...

Albumentations

... In particular **Albumentations**:

- Powerful and flexible Python library designed for image augmentation;
- It provides a high-level API that simplifies the process of applying a wide range of transformations;
- It offers an efficient and user-friendly solution, making it an ideal choice for large-scale experimentation;
- Support of a variety of transformations such as rotation, flipping, brightness, contrast, blur, etc...
- Simple creation of augmentation pipelines where transformations are applied randomly with user-defined probabilities;
- Low-level image processing methods abstraction, allowing users to define augmentation strategies with just a few lines of code;

Albumentations: Pseudo-Random Transformation Function

Create a function to built a random transformation is easy with Albumentations:

```
# each invocation creates a new trasnformation
def create_random_transform():
    return A.Compose([
        A.OneOf([
            A.HorizontalFlip(p=1),
            ...
        ], p=0.5),
        A.SomeOf([
            ...
            A.GaussianBlur(p=1),
            A.Sharpen(p=1),
            ...
        ], n=2, p=1),
        A.OneOf([
            A.GridDistortion(p=1),
            ...
        ], p=prob),
    ])
```

Multiprocessing

2 Image Augmentation

Parallelization is a key strategy to overcome the computational limitations of image augmentation.

By distributing augmentation tasks across multiple processing units image transformations can be performed simultaneously, significantly reducing processing time.

The image augmentation process is an highly parallelizable task $\rightarrow$ multiple image can be augmented simultaneously.

In this laboratory **Multiprocessing** library has been used.

Multiprocessing: Pros and Cons

Advantages:

- True Parallelism for CPU-Bound Tasks: multiprocessing creates separate processes;
- Improved CPU Utilization: On multi-core machines, multiprocessing can fully utilize the available cores;
- Simplified High-Level Interface: `pool` offers convenient abstractions that make it easy to distribute tasks across a pool of worker processes;

Disadvantages

- Higher Memory Consumption;
- Inter-Process Communication (IPC) Overhead: Passing data between the main process and worker processes, or between worker processes, requires and IPC mechanisms that adds overhead, which can become a bottleneck;
- Sometimes the simple interface can become a problem $\rightarrow$ low level settings aren't accessible;

Multiprocessing: `Pool`

2 Image Augmentation

The Pool object manages a set number of worker processes and provides methods like `map` and `starmap` to distribute tasks efficiently:

- `map(func, iterable)`: Applies the function func to each element in iterable in parallel;
- `starmap(func, iterable of tuples)`: Similar to map(), but allows functions with multiple arguments by unpacking tuples from the iterable (used in the code);

UNIVERSITÀ DEGLI STUDI FIRENZE
Da un secolo, oltre.

Table of Contents

3 Implementation

Augment Image Function

3 Implementation

A simple function has been built to do the image augmentation.

```
def augment_image(image, output_dir, name, ext, index, saving):
    random_transform = create_random_transform()
    augmented = random_transform(image=image)["image"] # Apply random augmentation

    if saving:
        # Convert back to BGR for saving
        augmented = cv2.cvtColor(augmented, cv2.COLOR_RGB2BGR)

        # Generate output filename (e.g., "aug_image1_1.jpg", "aug_image1_2.jpg", ...)
        output_filename = f"aug_{name}_{index}{ext}"
        output_path = os.path.join(output_dir, output_filename)

        cv2.imwrite(output_path, augmented) # Save the augmented image
```

Sequential

3 Implementation

For all images in a folder:

- The image is read using `OpenCV` library;
- Then the function processes it using the previous function; item Optional: image saving;

```
def process_images_sequential(input_dir, output_dir, n_augmentations=5, saving=True):
    if not os.path.exists(output_dir):
        os.makedirs(output_dir)

    image_files = [os.path.join(input_dir, f) for f in os.listdir(input_dir) if f.endswith(('.jpg', '.png'))]

    for image_path in image_files:
        image = cv2.imread(image_path)
        if image is None:
            return f"Failed to load {image_path}"

        image = cv2.cvtColor(image, cv2.COLOR_BGR2RGB) # Convert to RGB

        filename = os.path.basename(image_path) # Extract filename (e.g., "image1.jpg")
        name, ext = os.path.splitext(filename) # Split into name and extension
        for i in range(n_augmentations):
            index = str(i).zfill(6)
            augment_image(image, output_dir, name, ext, index, saving)
```

Parallel Algorithms

3 Implementation

Three algorithms have been implemented:

- P_1: the files are read and distributed to the workers;
- P_2: each process read its image file;
- P_Q: a queue process is used for saving the files;

P_1: Code

3 Implementation

```python
def process_images_parallel(input_dir, output_dir, n_augmentations=5, num_workers=None, saving=True):
    if not os.path.exists(output_dir):
        os.makedirs(output_dir)

    image_files_raw = [os.path.join(input_dir, f) for f in os.listdir(input_dir) if f.endswith(('.jpg', '.png'))]
    images_data = [(cv2.cvtColor(cv2.imread(image_path), cv2.COLOR_BGR2RGB), output_dir, os.path.splitext(os.path.
        basename(image_path))[0], os.path.splitext(os.path.basename(image_path))[1]) for image_path in
        image_files_raw]

    args = [(*data, str(index).zfill(6), saving) for data in images_data for index in range(n_augmentations)]

    # Use multiprocessing to process images in parallel
    with Pool(num_workers) as pool:
        pool.starmap(augment_image, args)
```

P_1: Considerations

3 Implementation

- Simple implementation $\rightarrow$ master process read the file and distributes it to the workers;
- `pool` is used to distribute the work to the workers;
- If we want to save the image this operations is done inside the augment image function;

P_2: Code

3 Implementation

```
1  def process_images_parallel_readFileOneByOne(input_dir, output_dir, n_augmentations=5, num_workers=None, saving=
       True):
2      if not os.path.exists(output_dir):
3          os.makedirs(output_dir)
4
5      image_files = [os.path.join(input_dir, f) for f in os.listdir(input_dir) if f.endswith(('.jpg', '.png'))]
6      args = [(data, output_dir, str(index).zfill(6), saving) for data in image_files for index in range(n_augmentations)]
7
8      # Use multiprocessing to process images in parallel
9      with Pool(num_workers) as pool:
10         pool.starmap(augment_image_from_file, args)
```

P_2: Considerations

3 Implementation

- Modified `augment image` function;
- The file is read directly from the workers processes;
- The saving process is inside the augment image function;

```
1 def augment_image_from_file(image_path, output_dir, index, saving):
2     random_transform = create_random_transform()
3     # Difference from before
4     image = cv2.cvtColor(cv2.imread(image_path), cv2.COLOR_BGR2RGB)
5
6     augmented = random_transform(image=image)["image"] # Apply random augmentation
7
8     if saving:
9         ...
```

Main idea

Saving a file to the disk can become a bottleneck due to I/O to main memory.
So the idea is to use a subset of the workers as saving processes, with the responsibility of saving the image to a file.

Number of saving processes

The subset of saving processes is calculated as:

$$\#\text{savers} = \max\left(1, \left\lceil\sqrt{\#\text{workers}}\right\rceil - 1\right)$$

Considerations

- The I/O operations are centralized into a single process;
- The bottleneck might become memory bandwidth between the processes;
- Extended experimentation might be required to tune the number of saving processes;

P_Q: Code (1)

3 Implementation

```
def process_images_parallel_queue_sqrt(input_dir, output_dir, n_augmentations=5, num_workers=None, saving=True):
    if not saving:
        print("Parallelization with queue not useful if saving files isn't allowed")
        return

    if not os.path.exists(output_dir):
        os.makedirs(output_dir)

    num_savers = max(1, math.ceil(math.sqrt(num_workers)-1))

    manager = Manager() # Use Manager to create a shared Queue
    queues_raw = [manager.Queue() for _ in range(num_savers)]

    image_files_raw = ...
    images_data = ...
    ...
```

P_Q: Code (2)

3 Implementation

```
1       ...
2       args = [(*data, str(index).zfill(6), queues_raw[index%num_savers]) for data in images_data for index in range(
                n_augmentations)]
3
4       saving_processes = [Process(target=save_images_from_queue, args=(queue, )) for queue in queues_raw]
5       for saving_process in saving_processes:
6               saving_process.start()
7
8       # Use multiprocessing to process images in parallel
9       with Pool(num_workers-num_savers) as pool:
10              pool.starmap(augment_image_queue, args)
11
12      for queue in queues_raw:
13              queue.put(None)
14
15      for saving_process in saving_processes:
16              saving_process.join()
```

P_Q: Support Functions

3 Implementation

```python
def save_images_from_queue(queue):
    while True:
        image_data = queue.get()
        if image_data is None: # Sentinel value to stop the process
            break

        image, output_filename = image_data
        cv2.imwrite(output_filename, image)


def augment_image_queue(image, output_dir, name, ext, index, queue):
    random_transform = create_random_transform()
    augmented = random_transform(image=image)["image"] # Apply random augmentation

    # Convert back to BGR for saving
    augmented = cv2.cvtColor(augmented, cv2.COLOR_RGB2BGR)

    # Generate output filename (e.g., "aug_image1_1.jpg", "aug_image1_2.jpg", ...)
    output_filename = f"aug_{name}_{index}{ext}"
    output_path = os.path.join(output_dir, output_filename)

    queue.put((augmented, output_path))
```

UNIVERSITÀ DEGLI STUDI FIRENZE
Da un secolo, oltre.

Table of Contents

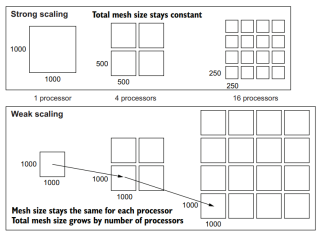
4 Experimentation

Experimental Setup

4 Experimentation

Strong and **Weak scaling** have been performed to test the different methods.

- In each test the three implementations have been compared;
- For each tests an average of 2 executions has been taken;
- Each execution consists in 100 augmentations of 10 images;
- Execution times consists in image reading, processing;
- Execution times with and without saving have been collected;



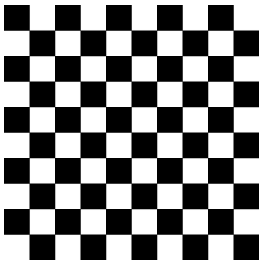
Strong scaling: image resolution of 2000×1000
Weak scaling: image resolution from 250×250 to 2000×2000

Image Collection

4 Experimentation

1° phase: Validation
Validation of the methods with real images.

2° phase: Experimentation
Experimentation with self-generated test images.



Experimental Environment

All the experimentation has been conducted in a dual-socket *Intel Xeon Silver 4314* system with 32 physical cores and 64 threads.

L1 and L2 cache are private for each core but L3 is shared between the same socket.

Name	Value
Clock rate [max]	2.4 [3.4] GHz
# Sockets	2
Cores	32 (16 per socket)
# Logical Processors	64 (2 per core)
L1d (Data cache)	1.5 MiB (48 KiB per core)
L1i (Instruction cache)	1 MiB (32 KiB per core)
L2 cache	40 MiB (1.25 MiB per core)
L3 cache	48 MiB (24 MiB per socket)
# NUMA nodes	2

UNIVERSITÀ DEGLI STUDI FIRENZE
Da un secolo, oltre.

Table of Contents

5 Results

Strong Scaling

5 Results

	# ranks	Ideal Execution [s]	Execution time [s]			Speedup			Efficiency		
			P_1	P_0	P_2	P_1	P_0	P_2	P_1	P_0	P_2
Saving	1	82.07	-	-	-	-	-	-	-		-
	2	41.03	43.72	87.95	47.08	1.88	0.93	1.74	94%	47%	87%
	4	20.52	24.78	31.77	27.16	3.31	2.58	3.02	83%	65%	76%
	8	10.26	14.03	20.79	14.15	5.85	3.95	5.80	73%	49%	73%
	16	5.13	10.81	17.53	10.99	7.59	4.68	7.47	47%	29%	47%
	32	2.56	14.41	15.51	14.10	5.69	5.29	5.82	18%	17%	18%
	64	1.28	14.20	34.36	13.62	5.78	2.39	6.03	9%	4%	9%
No Saving	1	76.37	-	-	-	-	-	-		-	-
	2	38.19	49.27	-	49.30	1.67	-	1.66	77%	-	77%
	4	19.09	27.46	-	28.73	2.99	-	2.86	70%	-	66%
	8	9.55	17.34	-	14.99	4.73	-	5.47	55%	-	64%
	16	4.78	10.80	-	11.52	7.60	-	7.12	44%	-	41%
	32	2.39	9.88	-	9.95	8.30	-	8.25	24%	-	24%
	64	1.19	10.87	-	10.84	7.55	-	7.57	11%	-	11%

Strong Scaling: Considerations

5 Results

- *No saving* gives same (or worst!) results with respect to *saving* methods $\rightarrow$ this offloads some work from pure CPU computation, reducing the direct contention for CPU resources;
- Additionally, memory pressure might decrease because the augmented images are written to disk rather than accumulating in RAM.
- *No saving* gives better execution when a big number of processes are spawned $\rightarrow$ the impact of saving images are contained and only visible when the I/O memory contention increase (memory bus saturation);
- P_Q doesn't help in reducing the execution times $\rightarrow$ passing data between the process might become a bottleneck bigger than saving;
- Fast efficiency decay $\rightarrow$ Inter Process Communication and cache effects. Data needed by one process overwrites data needed by another in the cache;

Weak Scaling

5 Results

	Image Size	Ideal Execution [s]	Execution time [s]			Efficiency		
			P_1	P_Q	P_2	P_1	P_Q	P_2
Saving	250×250	7.31	-	-	-	-		-
	500×250	7.31	5.94	11.66	6.09	123%	63%	120%
	500×500	7.31	5.04	6.89	4.86	145%	106%	150%
	1000×500	7.31	4.72	6.45	4.76	155%	113%	154%
	1000×1000	7.31	5.94	7.50	5.57	123%	97%	131%
	2000×1000	7.31	9.99	15.26	10.96	73%	48%	67%
	2000×2000	7.31	58.75	70.95	62.54	12%	10%	12%
No Saving	250×250	6.79	-	-	-	-	-	-
	500×250	6.79	5.58	-	5.14	122%	-	132%
	500×500	6.79	4.53	-	4.30	150%	-	158%
	1000×500	6.79	4.41	-	4.39	154%	-	155%
	1000×1000	6.79	5.58	-	6.12	122%	-	111%
	2000×1000	6.79	16.13	-	17.00	42%	-	40%
	2000×2000	6.79	57.77	-	61.47	12%	-	11%

Weak Scaling: Considerations

- Superlinear execution is reached with < 32 processes $\rightarrow$ improved cache utilization (smaller images than the previous case and cache optimization might be the factor of performances improvements);
- The number of saving processes in P_Q might be optimized $\rightarrow$ no performance advantages can be seen using this method;
- No big differences between *saving* and *no saving* $\rightarrow$ the main problem might be different from the saving process;
- No huge differences between P_1 and P_2 $\rightarrow$ the main problem might be different from reading file;

cProfile

5 Results

After seen Strong and Weak scaling we can conclude that the main problem isn't saving the image or reading the file…

With `cProfile` we can see that…

ncalls	tottime	percall	cumtime	percall	filename:lineno(function)
19	9.796	0.516	9.796	0.516	{method 'acquire' of '_thread.lock' objects}
10	0.081	0.008	0.081	0.008	{imread}
32	0.065	0.002	0.066	0.002	{built-in method posix.fork}
562	0.024	0.000	0.024	0.000	{built-in method posix.waitpid}
10	0.007	0.001	0.007	0.001	{cvtColor}
1	0.007	0.007	0.007	0.007	{method 'acquire' of '_multiprocessing.SemLock' objects}
1	0.003	0.003	10.008	10.008	<string>:1(<module>)
32	0.002	0.000	0.003	0.000	process.py:80(__init__)
32	0.002	0.000	0.069	0.002	popen_fork.py:62(_launch)
1	0.002	0.002	0.081	0.081	pool.py:314(_repopulate_pool_static)

… the main problem is **contention** between the process!
Is this contention on the CPU or memory resources? 🤔

Possible Explaination

- **Cache Effects**: multiple processes accessing large image datasets simultaneously can lead to increased cache misses and thrashing (where data needed by one process overwrites data needed by another in the cache), forcing more frequent, slower access to main memory;
- **Memory Bus Saturation**: as more cores simultaneously request data, the shared memory bus can become saturated, limiting the effective data throughput per core regardless of how many cores are added;

The combination of these lead to memory bus contention.

UNIVERSITÀ DEGLI STUDI FIRENZE
Da un secolo, oltre.

Table of Contents

▶ Conclusions

- Image augmentation is both a memory and CPU intensive task;
- Reach good level of efficiency in a parallel code is hard;
- Python Multiprocessing provides a simple interface but this can be a limitation (both for performance explanation and optimization);
- More experimentation is needed to understand better the behavior of the methods;
- Deeper profiling is needed to understand how the process waste time and how the CPU and memory resources are used;

Image Augmentation with Python Multiprocessing

Thank you for listening!

Any questions?