Image Augmentation with Python Multiprocessing

Tommaso Botarelli
tommaso.botarelli@edu.unifi.it
7136210

Abstract

Image augmentation is a useful technique in machine learning, particularly in computer vision, as it enhances dataset diversity and improves model generalization. However, applying augmentation sequentially to large datasets can be computationally expensive, creating bottlenecks in training pipelines. In this work, we explore parallel implementations of image augmentation to improve efficiency, leveraging Python's multiprocessing module. Our approach utilizes the Pool class to manage worker processes, enabling parallel execution of transformations. We evaluate our methods through strong and weak scaling experiments, analyzing their performance across varying computational resources and image sizes. The results demonstrate that multiprocessing substantially accelerates augmentation, but with an efficiency decay due to a combination of problems.

1. Introduction

Image augmentation is a crucial technique in machine learning, particularly in the field of computer vision. It involves applying various transformations to images—such as rotation, scaling, flipping, cropping, and color adjustments—to artificially expand the size of a dataset. By generating diverse variations of images, augmentation enhances the robustness and generalization ability of machine learning models, reducing the risk of overfitting, especially when working with limited data.

In deep learning, large-scale datasets are often required to train models effectively. However, acquiring and labeling vast amounts of data can be expensive and time-consuming. Image augmentation helps mitigate this issue by synthetically increasing dataset diversity, allowing models to learn more meaningful features without requiring additional manually labeled samples.

Despite its advantages, image augmentation can be computationally expensive, particularly when applied to large datasets or in real-time applications. Many augmentation techniques involve pixel-wise operations, interpolation, and complex transformations, which can introduce significant processing overhead. Therefore, designing efficient algorithms to accelerate image augmentation is essential, especially when training deep neural networks on high-resolution images.

In this work, we present some different parallel implementations designed to improve computational efficiency. By leveraging parallel processing techniques, our approach aims to reduce augmentation time, enabling faster training cycles and improving the overall performance of machine learning pipelines.

2. Image Augmentation

Image Augmentation is a set of systematic transformations applied to an image dataset to generate new, altered versions of existing images without modifying their semantic content. Formally, given an image I from a dataset $\mathcal{D}$, an augmentation function $T : \mathbb{I} \to \mathbb{I}$ applies a transformation such that:

$$I' = T(I)$$

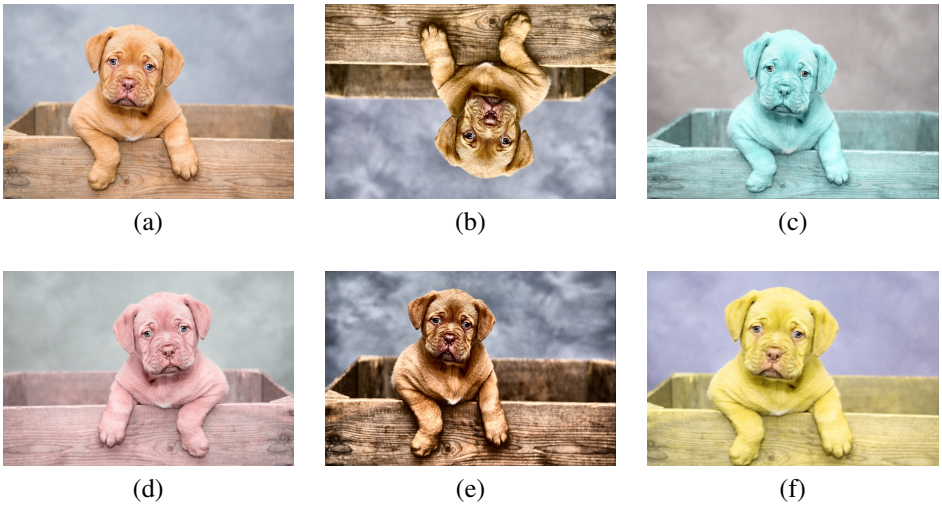
where I' represents the augmented image and $\mathbb{I}$ denotes the space of all possible images. The transformation T can be chosen from a set of predefined operations, including:

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

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. The goal is to increase dataset variability, improving model generalization and robustness while reducing overfitting.



(a) (b) (c)

(d) (e) (f)

Figure 1: Initial image (a) and some augmentation generated from random filter setting (Algorithm 1)

3. Albumentations

Albumentations is a powerful and flexible Python library designed for image augmentation in machine learning and deep learning applications. It provides a high-level API that simplifies the process of applying a wide range of transformations. Unlike traditional image processing approaches that require extensive low-level coding, Albumentations offers an efficient and user-friendly solution, making it an ideal choice for large-scale experimentation.

The key features of Albumentations are (see [1]):

- 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.

3.1. Image Augmentation Pipeline

With Albumentations implement a transform and applying it to an image is really simple. As we can see in Algorithm 1 we can build a transformation using user defined probabilities and a composition of single transformation.

```python
# 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),
   ])

def augment_image(image, saving=True):
  random_transform = create_random_transform
      ()
  # Apply random augmentation
  augmented = random_transform(image=image)[
      "image"]
  if saving:
       # it converts and saves the image in
           disk
```

Algorithm 1: Python code. Simple method to get a new composition of transformations from a set of developer defined transormations. Then we can use it to apply a transformation to an image

4. Parallelization

Applying image augmentation to nowadays large datasets in a sequential manner can become a significant bottleneck.

Parallelization is a key strategy to overcome these computational limitations. 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. Indeed each image can be processed independently leading to a substantial reduction in execution time. This is particularly useful in online augmentation scenarios where transformations are applied dynamically during model training.

So use a parallel implementation of image augmentation makes efficient use of modern hardware, improves training pipeline performance and ensures that preprocessing pipelines remain efficient, even when handling large datasets.

```python
def process_images_parallel(input_dir,
    output_dir, n_augmentations=5,
    num_workers=N, saving=True):
    image_files_raw = [os.path.join(
        input_dir, f) for f in os.listdir(
        input_dir) if f.endswith(('.jpg', '.
        png'))]
    # some data omitted for clarity
    # each image is read once
    images_data = [(cv2.cvtColor(cv2.imread(
        image_path), cv2.COLOR_BGR2RGB), ...)
         for image_path in image_files_raw]

    # then each image data is copied
    args = [(*data, ...) 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)
```

Algorithm 2: Python code. Standard parallel implementation. Images are read once by the main processes and then the work is distributed using `Pool`

4.1. Python Multiprocessing

To parallelize the code we chose to use `multiprocessing` Python library. Python's multiprocessing module provides an efficient way to parallelize tasks by creating multiple processes that run independently, leveraging multiple CPU cores. Unlike Python's threading, which is limited by the Global Interpreter Lock (GIL), multiprocessing allows true parallel execution.

One of the most user-friendly features of the multiprocessing module is the `Pool` class, which simplifies parallel execution. 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.

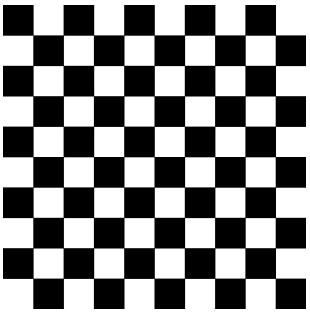


Figure 2: Example of image used for testing

4.2. Implementations

In this sections we provide an overview over the several parallel implementations. We chose to implements different version of a parallel code based on different image reading mechanisms.

4.2.1 Standard Parallel Implementation P_1

This is the base implementation (Algorithm 2). We use the `Pool` class of `multiprocessing` library to instantiate the processes and then we execute these processes with the arguments.

In this implementation each file is read once from the main process, then each process receive the image data.

4.2.2 Parallel Implementation with Multiple Read Operations P_2

The main difference with the previous 4.2.1 is that here each process read the file independently (see Algorithm 3).

4.2.3 Parallelization with Saving Processes P_Q

When we have to process a big number of standard size images we can have problems for the saving process. Indeed with multiple processes executing the same code we can encounter latency due to saving all the image in disk simultaneously.

For this reason, an additional method was developed that, from the maximum number of processes indicated by `num_workers` generates a subset of processes with the sole task of picking up the image from a queue and saving it to disk (see Algorithm 4).

```python
def augment_image_from_file(image_path, saving):
  image = cv2.cvtColor(cv2.imread(image_path), cv2.COLOR_BGR2RGB)
  augment_image(image, saving)


def process_images_parallel_multiple_read(input_dir, ..., saving=True):
  image_files = [os.path.join(input_dir, f) for f in os.listdir(input_dir) if f.endswith(('.jpg', '.png'))]
  args = [(data, ...) for data in image_files for index in range(n_augmentations)]

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

Algorithm 3: Python code. Standard parallel implementation. Each process in the `Pool` reads the image from the disk and then process it.

The subset of savers created is defined by the following formula:

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

5. Experimentation

The experimentation carried out consists of two different tests: **strong and weak scaling**. With the first, the goal is to verify that the implemented methods are able to scale in performance as resources increase. Weak scaling, on the other hand, allows us to observe how methods scale with increasing data size input.

```python
def save_images_from_queue(queue):
  while True:
    image_data = queue.get()
    if image_data is None:
      break

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

def augment_image_queue(image, output_dir, name, ext, index, queue):
  random_transform = ...
  augmented = ...
  output_path = ...

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

```python
def process_images_parallel_queue_sqrt(input_dir, ..., saving=True):
  num_savers = max(1, math.ceil(math.sqrt(num_workers)) - 1)

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

  image_files_raw = ...
  images_data = ...

  args = [(*data, ..., queues_raw[index%num_savers]) for data in images_data for index in range(n_augmentations)]

  saving_processes = [Process(target=save_images_from_queue, args=(queue,)) for queue in queues_raw]
  for saving_process in saving_processes:
    saving_process.start()

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

  for queue in queues_raw:
    queue.put(None)
  for saving_process in saving_processes:
    saving_process.join()
```

Algorithm 4: Python code. A subset of all the workers are created as `savers` with the responsibility to save the image from the queue

So that reproducible experimentation can be made possible, test images of definable size have been prepared through a Python script. The images generated for testing are of the type depicted in Figure 2.

Two runs were made for each test to keep track of run times considering saving the modified images to disk and without considering this time.

5.1. Results

In this sections we provide all the results from the experimentation, we first show the strong scaling results and then we move to the weak scaling results.

	# ranks	Ideal Execution [s]	Execution time [s]			Speedup			Efficiency		
			P_1	P_Q	P_2	P_1	P_Q	P_2	P_1	P_Q	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%

Table 1: Strong scaling results. The image size is fixed to a resolution of 2000×1000 and times refer to 10 images and 100 augmentations each.

5.1.1 Strong Scaling

Table 1 shows all the execution times of the various methods in the case of strong scaling. Since P_Q method is implemented for saving files the execution times of this method are not considered in the experiment without saving.

As can be seen from the table although the best speedup is in the case of "no saving" add image saving also leads to better performance with 2 and 4 processes. This can be explained by the fact that without saving all CPU cores are dedicated only to augmentation, and they may be competing for memory bandwidth leading to cache misses and increased contention among processes.

On the other hand with saving some workers are offloaded to I/O operations, reducing pure CPU contention. So memory pressure might be lower because augmented images are written to disk instead of occupying RAM.

In addition to this we can see how P_Q did not help improving performances. This is consistent with what might have been expected. In fact, image saving does not seem to be the main problem for performance decay. In fact, the decrease in the number of workers and the addition of data transmission between processes does not allow this method to achieve the hoped-for benefits.

Finally, this test shows us that the performance of these methods decays very quickly. This, although could have been expected given the large volume of data being used with the resulting big number of data exchange between memory and cache, is nevertheless very visible as early as 8 processes. The reason for this decay can be found in the memory contention among the various processes. The increase in the number of processes in fact leads to an increasing bandwidth contention among them leading to a longer wait time and performance decay.

5.1.2 Weak Scaling

Table 2 shows all the execution times of the various methods in the case of weak scaling. Since P_Q method is implemented for saving files the execution times of this method are not considered in the experiment without saving.

As can be seen from the table in general for files of size $\leq 1000 \times 1000$ the trend is superlinear compared to the base case. This can be easily explained by an optimization of cache usage. After this threshold, on the other hand, the size of the images has a great impact in the loss of efficiency of the algorithms. However, this is not due to the increased time spent in saving the files but in the increased computation time due to the size of the files and to the bandwidth contention.

The absence of large differences in the times between "saving" and "no saving" confirms the already highlighted fact that the algorithm turns out to be CPU bound. This is confirmed by the fact that during the execution of the methods the CPU had a utilization of 100%.

```
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)
```

Figure 3: `cProfile` output

Moreover, another confirmation to what has just been said is given to us by the fact that P_1 and P_2 present similar

	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%

Table 2: Weak scaling results. Times refer to 10 images with 100 augmentations each.

performance (in some cases P_2 presents even higher performances). So even the "read" operation does not affect the execution of the algorithm very much, and thus we can say that the operations in memory are not so influential compared to the computation itself.

Another explanation for this behavior can be found in how the `map` method is implemented. In fact, by going to analyze the execution time of each function with `cProfile` what is generally obtained is depicted in Figure 3. The method `acquire' of '_thread.lock' objects` is the method where the most time is consumed. This therefore suggests a degree of contention among the various processes.

This it might be explainable by memory bus contention due to the combination of:

- 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;

6. Conclusions

The experimentation carried out allows us to say that image augmentation is a both a CPU and memory intensive task so reaching good level of efficiency in a parallel code is hard.

Using forms of parallelization by processes achieves better performance than the sequential case but with a rather high loss of efficiency. The loss of efficiency is mainly due to the continuous bandwidth contention between processes. Since images are in fact usually high file sizes, contention of the bus and is a limit for the maximum degree of speedup that can be achieved.

Furthermore, the `multiprocessing` library although it can provide a facilitated implementation of parallel programming does not turn out to be complete with customization of process distribution (think for example of the `spread` or `close` distribution of OpenMP). This results in a limitation for the total utilization of available resources (see [3]).

6.1. Future Developments

Future developments should consider an extended experimentation to understand the behavior of the methods and deeper profiling to understand how the process waste time and how the CPU and memory resources are used.

6.2. Additional Materials

The code and the experimentation results are available in GitHub (see [2]).

References

[1] Albumentations. *Documentation*. URL: `https://albumentations.ai/docs/`.

[2] Tommaso Botarelli. *Parallel Computing Project*. URL: `https://github.com/TommasoBotarelli/ParallelComputing_`

UNIFI _ Project / tree / master / image _ augmentation.

[3] Hacker News. *Python Multiprocessing Performances*. URL: https : / / news . ycombinator . com / item?id=34974480.