Parallel DataLoader
Parallel Programming for Machine Learning
Project Work in Artificial Intelligence Programming

Niccolò Arati
niccolo.arati@edu.unifi.it

Abstract

Questo progetto punta ad analizzare i vantaggi di un'implementazione parallela di un Dataloader. In particolare si vuole andare ad osservare lo ***speedup****, calcolato come rapporto tra il tempo di esecuzione sequenziale e tempo di esecuzione parallelo, al variare di diverse configurazioni per il Dataloader. Verranno analizzati i risultati anche al variare del numero di workers relativi alla configurazione parallelizzata.*

1. Introduzione

Un Dataloader è una classe che fornisce un modo flessibile ed efficiente per caricare dei dati. L'applicazione principale consiste nel caricare i dati in un modello per fare addestramento o inferenza.
In questo lavoro ci occuperemo di caricare delle immagini ed applicare ad esse delle tecniche di data augmentation.
Lo scopo del progetto e quello di misurare lo speedup ottenuto parallelizzando il dataloader rispetto alla sua versione sequenziale. Il linguaggio di programmazione utilizzato è Python, e per parallelizzare è stato usato Multiprocessing.
Tutti gli esperimenti sono stati svolti su un PC con Windows 10 come sistema operativo e una CPU Intel Pentium Gold G5400.

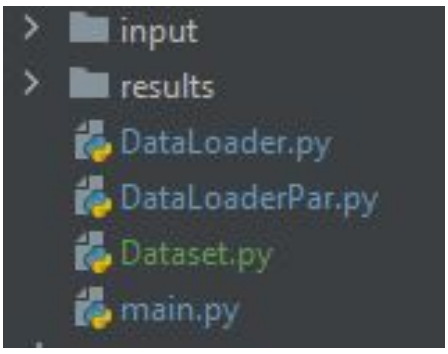

Figure 1. Organizzazione del codice

2. Organizzazione del codice

Si è utilizzato PyCharm come IDE.
Il codice è composto da più file:

- **DataLoader.py**: contiene l'implementazione sequenziale della classe DataLoader, definita come iterabile;
- **DataLoaderPar.py**: contiene l'implementazione parallelizzata della classe DataLoaderPar, estende la classe DataLoader;
- **Dataset.py**: contiene l'implementazione della classe Dataset;
- **main.py**: contiene le funzioni relative ai test svolti e la funzione di main().

Viene considerata anche la cartella **inputs**, che contiene 111 sottocartelle con al loro interno 1000 immagini, sulle quali verrà poi chiamato il Dataloader.
Dato che, come già specificato in Sec. 1, utilizzeremo le immagini come dati da caricare col Dat-

aloader, l'implementazione fatta considera solo le immagini e non generalizza per altri tipi di dati.

2.1. Classe DataLoader

La classe DataLoader accede al dataset di immagini da caricare e ne restituisce per ogni iterazione un certo numero sotto forma di batch.
Vengono definite 4 funzioni per gli oggetti DataLoader:

Figure 2. Dichiarazione della classe DataLoader

- **__init__** inizializza il dataloader ed i suoi attributi, richiede come parametri il dataset (tramite un oggetto di classe Dataset), la dimensione dei batch da caricare, e un booleano opzionale per randomizzare l'ordine delle immagini. Inoltre definisce un intero index assegnandolo a 0, verrà poi utilizzato per iterare tra le immagini;

- **__iter__** ritorna l'oggetto (DataLoader) su cui eseguire le iterazioni;

- **__next__** carica un batch di immagini chiamando il metodo get per un numero di volte pari al valore dell'attributo batch_size e lo restituisce in outptut;

- **get** carica un'immagine dal dataset e incrementa di uno l'attributo index.

2.2. Classe DataLoaderPar

Sottoclasse di DataLoader, per il metodo __init__ richiede dei parametri aggiuntivi rispetto alla classe DataLoader, ed inizializza delle strutture dati che verranno discusse in Sec. 3 insieme ad altre funzioni definite per questa classe.

2.3. Classe Dataset

La classe Dataset serve come wrapper per il dataset effettivo da caricare, nel nostro caso riceve i path relativi alla locazione delle immagini e, sotto richiesta del DataLoader, le carica in memoria e le restituisce.
Presenta 3 metodi:

- **__init__** inizializza il dataset, richiede come parametri size, ovvero la dimensione del dataset, e im_paths, ovvero una lista contenete i path delle immagini da caricare. Questi due parametri vengono poi salvati come attributi della classe;

- __len__ ritorna semplicemente la dimensione del dataset;

- **__getitem__** richiede in input l'indice dell'immagine che vogliamo caricare, e poi attraverso quell'indice accede al path dell'immagine desiderata, la carica attraverso il pacchetto cv2, e la restituisce come output.

2.4. main.py

Nel file sono presenti i test relativi alla parallelizzazione del Dataloader (presentati in Sec. 4), insieme alla funzione di **main**, che ha come scopo quello di eseguire i test definiti nel file.
Per generare le stringhe contenenti i path delle immagini è stata utilizzata la libreria pathlib.

3. Parallelizzazione con Multiprocessing

Il codice della classe DataLoaderPar definisce una funzione chiamata **worker_funct**, che sarà quella eseguita dai processi Worker definiti tramite Multiprocessing.
I processi vengono generati attraverso il metodo __init__ della classe, e poi sono presenti altri metodi per la loro gestione, che verranno presentati nelle successive sottosezioni.
L'idea dietro la parallelizzazione del Dataloader è che, quando viene caricato un batch di immagini, si suppone che queste vengano sottoposte tramite un certo processo ad una trasformazione (con Albumentations, Sec. 4). Mentre le immagini vengono trasformate, vogliamo utilizzare dei processi

Worker per andare a fare il precaricamento del batch successivo, in modo che questo (o la maggior parte di esso) sia già pronto in memoria per la trasformazione.

3.1. Funzione associata agli Workers

```
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]))
```

Figure 3. Codice della funzione worker_funct

Dal codice presente in Fig. 3, la funzione richiede 3 parametri: il dataset da caricare, una index_queue ed una output_queue (entrambe sono oggetti multiprocessing.Queue). La output_queue è condivisa tra tutti i processi, mentre per la index_queue ogni processo ne ha una privata associata.
Nella index_queue sono presenti gli indici degli elementi da caricare assegnati ad un certo processo Worker, e quello che fa il processo (seguendo la funzione) è prendere un indice dalla coda (se non è vuota) e inserire la corrispondente immagine del dataset nella output_queue, accoppiata con l'indice.

3.2. Metodo __init__

Il metodo __init__ richiede gli stessi parametri della classe DataLoader, ai quali si aggiungono un parametro num_workers che specifica il numero di processi Worker da generare, e un parametro prefetch_batches che specifica il numero massimo di batch da precaricare.
Oltre ad inizializzare gli attributi corrispondenti ai parametri passati in input, il metodo inizializza anche la output_queue (Sec. 3.1), una lista che conterrà le index_queue associate ai processi Worker, una lista che conterrà i processi Worker, un attributo worker_cycle che servirà per iterare tra i processi per assegnare alle loro index_queue gli indici delle immagini da caricare, una lista chiamata cache (Sec. 3.3) ed un attributo prefetch_index che tiene conto degli elementi precaricati (si differenzia dall'attributo index in quanto index tiene conto delle immagini effettivamente processate da chi stia usando il DataLoaderPar).
Successivamente, il metodo inizializza i processi Worker e le rispettive index_queue:

```
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)
```

Figure 4. Codice relativo all'inizializzazione dei processi Worker e delle index_queue ad essi associate.

Come mostrato in Fig. 4, viene generato un multiprocessing.Process che ha come target la funzione specificata in precedenza, e come attributi il dataset e le due code (index_queue privata e output_queue condivisa).
Il processo Worker viene specificato come **daemon**, in quanto vogliamo che agisca in background rispetto al processo di trasformazione delle immagini con Albumentations.
Infine, viene chiamata la funzione prefetch, presentata in Sec. 3.3

3.3. Caricamento immagine e gestione Dataloader

La funzione **prefetch** si occupa della gestione del precaricamento delle immagini. Prima verifica che non si sia raggiunto la fine del dataset o di non essere più di due batch avanti rispetto alle immagini effettivamente caricate, successivamente assegna gli indici delle immagini da precaricare alle index_queue associate ai processi Worker.
Per farlo sfrutta l'attributo worker_cycle, e si occupa di conseguenza anche di aggiornare l'indice di prefetch.
Il metodo **get** invece si occupa del caricamento effettivo delle immagini, e quindi del batch corrente.

Osservando il codice in Fig. 5, possiamo sintetizzarlo in 3 punti:

```python
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

    self.index += 1
    return item
```

Figure 5. Codice del metodo get

- Si controlla se l'indice dell'immagine da caricare sia presente nella lista definita dall'attributo cache, se è presente viene rimosso dalla lista e restituito in output;
- Altrimenti si va a controllare nella output_queue, se l'elemento ottenuto ha come indice quello corrispondente all'immagine da caricare viene restituito in output;
- Sennò l'elemento viene aggiunto alla cache in attesa che l'indice degli elementi da caricare raggiunga quello dell'elemento in questione (primo punto).

Il metodo __**iter**__, come per la classe DataLoader, restituisce l'oggetto su cui eseguire le iterazioni, e quindi si occupa di reinizializzare gli attributi relativi a indice dell'immagine da caricare (index), indice dell'immagine da precaricare (prefetch_index) e lista cache.
Infine, il metodo __**delete**__ si occupa di terminare i processi Worker.

4. Esperimenti svolti

Tornando al file main.py, le funzioni scritte per i test sono le seguenti:

- **testDataLoaderWT**: viene eseguita dalla funzione main, vengono calcolati gli speedup tra il dataloader sequenziale e quello parallelo al variare di alcune configurazioni che, in questo caso, tengono conto del numero di worker utilizzati per la parallelizzazione e della complessità delle trasformazioni applicate alle immagini di ogni batch;
- **testDataLoaderB**: anche questa viene eseguita dal main, stavolta però gli speedup vengono calcolati tenendo conto principalmente della dimensione del batch specificata.

I parametri considerati per gli esperimenti sono quindi il numero di workers utilizzati per Multiprocessing, la dimensione dei batch di immagini e la complessità delle trasformazioni per le immagini.
In particolare, per quest'ultima parte si è utilizzato **Albumentations**, in Fig. 6 viene mostrata la trasformazione più semplice che è stata definita (per quelle più complesse fondamentalmente viene applicata quella mostrata in figura più volte).

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

Figure 6. Traformazione di base definita con Albumentations

I risultati dei test vengono salvati in appositi file csv contenuti nella cartella **results**.
Di seguito vengono presentati i test svolti per il Dataloader.

4.1. Test1: variare n_workers e albumentations

Col primo test, eseguito dalla funzione testDataLoaderWT, si vuole andare a verificare l'impatto del numero di processi worker al variare della complessità delle trasformazioni, applicate con Albumentations ed eseguite tra i caricamenti dei batch.
Il valore del batch_size viene lasciato costante a 300.
Ciò che ci aspettiamo dal test è che, all'aumentare della complessità del processo di trasformazione dell'immagine, la versione sequenziale del Dataloader aumenti il tempo computazionale necessario a processare tutti i batch, mentre la versione parallelizzata dovrebbe comunque mantenere delle prestazioni migliori, perchè mentre il

processi di trasformazione delle immagini viene eseguito, sono presenti anche i processi worker che si occupano del precaricamento del batch successivo.
Il numero di workers considerati per l'esperimento viene fatto variare tra i valori 1, 2, 4, 6 e 8, mentre vengono considerate 4 possibili trasformazioni per le immagini, in ordine crescente di complessità.
Osservando i risultati presentati in Tab. 1, notiamo come le previsioni fossero esatte. Già osservando la prima colonna, solo sfruttando un singolo processo worker che lavora parallelamente al processo di Albumentations abbiamo dei miglioramenti notevoli per il valore di speedup al crescere della complessità delle trasformazioni delle immagini.
Tuttavia, ci sono delle limitazioni al miglioramento delle performance, dovute probabilmente all'hardware utilizzato per l'esperimento. Si osserva infatti che utilizzando 8 processi worker abbiamo sempre un peggioramento delle prestazioni rispetto alla stessa configurazione attuata con 6 processi workers.
Inoltre, osservando le righe della tabella, si nota una perdita del valore di speedup che all'aumentare della complessità del processo di Albumentations avviene con un numero di workers sempre minore. Infatti con il primo e il secondo processo di trasformazione delle immagini lo speedup, aumentando il numero di workers, diminuisce solo con 8 workers (rispetto al valore precedente), mentre con la terza trasformazione diminuisce passando da 4 a 6 workers, e con la quarta addirittura passando da 2 a 4 workers.
Questo può essere dovuto alla saturazione della CPU, dato che deve gestire un processo di trasformazione sempre più complesso e le immagini presenti nel batch considerato, oltre ad un numero di processi workers potenzialmente sempre più grande.

In ogni caso, osservando i valori di speedup vi è sempre un miglioramento delle prestazioni rispetto alla versione sequenziale, l'unico caso in cui praticamente le prestazioni sono equivalenti è quando viene applicata la trasformazione più semplice mentre un singolo processo Worker lavora per fare il precaricamento dei due batch successivi (riga 1, colonna 1).

	1 w	2 ws	4 ws	6 ws	8 ws
1stAlb	1.03472	1.38500	1.87490	2.14634	1.9381
2ndAlb	3.55765	5.02616	6.39762	7.16192	4.50477
3rdAlb	4.29092	5.94258	6.81652	4.65382	2.74523
4thAlb	4.52743	6.68475	5.05657	3.58080	2.15300

Table 1. Tabella con i valori di speedup al variare del numero di processi Worker e della complessità delle trasformazioni applicate alle immagini. Ricordarsi da Sec. 3 che il numero di processi totale in esecuzione è dato dal numero di workers più uno, corrispondente ad Albumentation.

4.2. Test2: variare batch_size

In questo secondo test, eseguito dalla funzione testDataLoaderB, si vanno a calcolare i valori di speedup variando le dimensioni dei batch caricati, lasciando costanti il numero di processi Worker a 2 e il processo di trasformazione delle immagini (equivalente al secondo processo applicato al test precedente).
Il numero di workers viene lasciato relativamente basso, così come la trasformazione viene presa non troppo complessa, per cercare di isolare l'impatto del batch_size rispetto agli altri parametri.
Le aspettative per questo esperimento sono di osservare un incremento dello speedup all'aumentare della dimensione del batch, sfruttando la capacità del Dataloader parallelo di precaricare le immagini del batch successivo mentre quelle del batch corrente vengono processate con Albumentations.
Il batch_size viene variato tra i valori 10, 50, 100, 300, 500 e 1000.

Osservando i risultati in Tab. 2 si può intuire che in realtà il parametro relativo alla dimensione del batch di immagini caricato non sia molto influente per il valore di speedup.
Infatti, tutti i valori di speedup sono paragonabili, l'unico che si discosta maggiormente dagli altri è quello relativo ad un batch_size di 1000.
Questo è comunque un risultato accettabile, in

	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

Table 2. Tabella con i valori di speedup al variare della dimensione del batch per le immagini.

quanto il batch_size solitamente viene deciso in base al task in cui viene sfruttato il Dataloader, o comunque in base alla dimensione dei dati a disposizione, ed ha senso che non dovremmo trovarci costretti ad aumentare la dimensione del batch solo per avere prestazioni migliori in termini di speedup (e quindi di tempo computazionale).