Звіт

Лабораторна робота №3

Підготував  
студент факультету КНК

Групи ІПС-21  
Ольховатий Ігор

# Багатопоточність для нейромереж

Використання багатопоточності дуже важливий аспект при роботі з нейромережами і машинним навчанням, в цій сфері зазвичай потрібно багато обчислень і будь-яке спрощення може суттєво допомогти у роботі. Саме тому, для проєкту який використовує глибоку нейромережу для роботи із зображеннями, що неймовірно складна задача з точки зору обчислень, було прийняти рішення застосувати багатопоточність.

Має програму для передбачення об’єктів на відео, яка окрім передбачення малює обмежувальні рамки навколо об’єктів і збирає нові кадри у вихідне відео. Запуск моделі проводиться виключно на GPU:

import sys  
import json  
import cv2  
  
from ultralytics import YOLO  
from sources.utils.benchmarking import measure\_time  
  
  
@measure\_time  
def predict\_video(model, threshold: float, file\_path: str, export\_path: str, n\_frame: int) -> list[int]:  
 cap = cv2.VideoCapture(file\_path)  
 fps = int(cap.get(cv2.CAP\_PROP\_FPS))  
 width = int(cap.get(cv2.CAP\_PROP\_FRAME\_WIDTH))  
 height = int(cap.get(cv2.CAP\_PROP\_FRAME\_HEIGHT))  
 total\_frames = int(cap.get(cv2.CAP\_PROP\_FRAME\_COUNT))  
 out = cv2.VideoWriter(export\_path, cv2.VideoWriter\_fourcc(\*'avc1'), fps, (width, height))  
 predictions = []  
 class\_names = {0: ‘Something’}  
  
 all\_predictions = [None] \* total\_frames  
  
 for f in range(total\_frames):  
 returned, frame = cap.read()  
 if not returned:  
 break  
  
 if f % n\_frame:  
 all\_predictions[f] = all\_predictions[f - 1]  
 else:  
 all\_predictions[f] = model(frame)[0]  
  
 frame\_predictions = 0  
 for result in all\_predictions[f].boxes.data.tolist():  
 x1, y1, x2, y2, score, class\_id = result  
  
 if score > threshold:  
 color = (0, 0, 255) # BGR format, red color  
 cv2.rectangle(frame, (int(x1), int(y1)), (int(x2), int(y2)), color, 4)  
 cv2.putText(frame, class\_names[int(class\_id)].upper(), (int(x1), int(y1 - 10)),  
 cv2.FONT\_HERSHEY\_SIMPLEX, 1.3, color, 3, cv2.LINE\_AA)  
 frame\_predictions += 1  
  
 predictions.append(frame\_predictions)  
 out.write(frame)  
  
 cap.release()  
 out.release()  
 cv2.destroyAllWindows()  
  
 return predictions  
  
  
def main(args: list[str], verbose: bool = False) -> None:  
 if verbose:  
 print(args)  
 assert len(args) == 4, "The script takes 4 arguments: video\_path, model\_path, threshold and n\_frame."  
 video\_path = args[0]  
 model\_path = args[1]  
 threshold = float(args[2])  
 n\_frame = int(args[3])  
  
 video\_path\_out = '{}\_out.mp4'.format(video\_path.replace('.mp4', ''))  
 yolo\_model = YOLO(model\_path)  
 boxes = predict\_video(model=yolo\_model,  
 threshold=threshold,  
 file\_path=video\_path,  
 export\_path=video\_path\_out,  
 n\_frame=n\_frame)  
 with open(video\_path.replace('.mp4', '\_stats.json'), 'w') as f:  
 json.dump(boxes, f)  
  
  
if \_\_name\_\_ == "\_\_main\_\_":  
 main(sys.argv[1:])

Має сенс запустити після обробку зображення на паралельному процесі, що дозволить вже починати обробляти зображення ще під час після обробки попереднього зображення, коли ми малюємо обмежувальні рамки. Відповідно маємо окремий клас для обробки кадрів на іншому потоці FrameProcessingThread.

import sys  
import json  
import cv2  
import threading  
import queue  
  
from ultralytics import YOLO  
from sources.utils.benchmarking import measure\_time  
  
  
class FrameProcessingThread(threading.Thread):  
 def \_\_init\_\_(self, model, threshold, class\_names, frame\_queue, result\_queue):  
 super(FrameProcessingThread, self).\_\_init\_\_()  
 self.model = model  
 self.threshold = threshold  
 self.class\_names = class\_names  
 self.frame\_queue = frame\_queue  
 self.result\_queue = result\_queue  
  
 def run(self):  
 while True:  
 frame\_index, frame = self.frame\_queue.get()  
 if frame is None:  
 break  
  
 predictions = self.model(frame)[0]  
 frame, num\_targets = process\_frame(frame, predictions.boxes.data.tolist(), self.threshold, self.class\_names)  
 self.result\_queue.put((frame, predictions.boxes.data.tolist(), num\_targets))  
 self.frame\_queue.task\_done()  
  
  
def process\_frame(frame, predictions, threshold, class\_names):  
 num\_targets = 0  
 for prediction in predictions:  
 x1, y1, x2, y2, score, class\_id = prediction  
  
 if score > threshold:  
 color = (0, 0, 255) # BGR format, red color  
 cv2.rectangle(frame, (int(x1), int(y1)), (int(x2), int(y2)), color, 4)  
 cv2.putText(frame, class\_names[int(class\_id)].upper(), (int(x1), int(y1 - 10)),  
 cv2.FONT\_HERSHEY\_SIMPLEX, 1.3, color, 3, cv2.LINE\_AA)  
 num\_targets += 1  
 return frame, num\_targets  
  
  
@measure\_time  
def predict\_video(model, threshold: float, file\_path: str, export\_path: str, n\_frame: int) -> list[int]:  
 cap = cv2.VideoCapture(file\_path)  
 fps = int(cap.get(cv2.CAP\_PROP\_FPS))  
 width = int(cap.get(cv2.CAP\_PROP\_FRAME\_WIDTH))  
 height = int(cap.get(cv2.CAP\_PROP\_FRAME\_HEIGHT))  
 total\_frames = int(cap.get(cv2.CAP\_PROP\_FRAME\_COUNT))  
 out = cv2.VideoWriter(export\_path, cv2.VideoWriter\_fourcc(\*'avc1'), fps, (width, height))  
 class\_names = {0: ‘Something’}  
 targets = []  
  
 frame\_queue = queue.Queue()  
 result\_queue = queue.Queue()  
  
 frame\_processing\_thread = FrameProcessingThread(model, threshold, class\_names, frame\_queue, result\_queue)  
 frame\_processing\_thread.start()  
  
 for f in range(total\_frames):  
 returned, frame = cap.read()  
 if not returned:  
 break  
  
 if f % n\_frame == 0:  
 frame\_queue.put((f, frame))  
 frame\_queue.join()  
  
 while not result\_queue.empty():  
 processed\_frame, predictions, num\_targets = result\_queue.get()  
 targets.append(num\_targets)  
 out.write(processed\_frame)  
 result\_queue.task\_done()  
 else:  
 processed\_frame, num\_targets = process\_frame(frame, predictions, threshold, class\_names)  
 targets.append(num\_targets)  
 out.write(processed\_frame)  
  
 frame\_queue.join()  
 while not result\_queue.empty():  
 processed\_frame, num\_targets = result\_queue.get()  
 targets.append(num\_targets)  
 out.write(processed\_frame)  
 result\_queue.task\_done()  
  
 frame\_queue.put((None, None))  
 frame\_processing\_thread.join()  
  
 cap.release()  
 out.release()  
 cv2.destroyAllWindows()  
  
 return targets  
  
  
def main(args: list[str], verbose: bool = False) -> None:  
 if verbose:  
 print(args)  
 assert len(args) == 4, "The script takes 4 arguments: video\_path, model\_path, threshold and n\_frame."  
 video\_path = args[0]  
 model\_path = args[1]  
 threshold = float(args[2])  
 n\_frame = int(args[3])  
  
 video\_path\_out = '{}\_out.mp4'.format(video\_path.replace('.mp4', ''))  
 yolo\_model = YOLO(model\_path)  
 boxes = predict\_video(model=yolo\_model,  
 threshold=threshold,  
 file\_path=video\_path,  
 export\_path=video\_path\_out,  
 n\_frame=n\_frame)  
 with open(video\_path.replace('.mp4', '\_stats.json'), 'w') as f:  
 json.dump(boxes, f)  
  
  
if \_\_name\_\_ == "\_\_main\_\_":  
 main(sys.argv[1:])

Для порівняння швидкодії програму було запущено з однаковими вхідними даними у вигляді одного відео та кількістю фреймів для обробки, два рази з порогом 0.2, та два рази з порогом 0.5:

Для версії без багатопоточності:

Час виконання: 33.32 секунди

Час виконання: 33.97 секунди

Час виконання: 38.24 секунди

Час виконання: 38.59 секунди

Середній час виконання: 36.03 секунди

Для версії з багатопотоковістю (2 threads):

Час виконання: 32.54 секунди

Час виконання: 32.51 секунди

Час виконання: 32.48 секунди

Час виконання: 34.26 секунди

Середній час виконання: 33.38 секунд

Бачимо очевидний приріст у 7.35%.

Варто зауважити, що відео було коротке, тож імовірно, для даних продуктових масштабів модель може у відносному сенсі працювати ще ефективніше.

Дані у цьому випадку досить проблематично (хоча можливо) покрити юніт тестами через стохастичність обчислень у глибоких нейронних мережах, проте варто зазначити, що наочний результат у вигляді відео залишився сталим. Варто зазначити, що ці результати можуть відрізнятися залежно від конкретного обладнання, програмного забезпечення та середовища, яке використовується для виконання.

Отже, можна зробити висновок, що використання багатопоточності оптимізації обчислень у даному випадку має сенс і допоможе пришвидшити роботу.

# Багатопоточність для збору даних

Для проєкту, у рамках якого передбачався масштабний збір даних із відкритих ресурсів було застосовано використання багатопоточності для пришвидшення цього часозатратного процесу. У проєкті потрібно збирати інформацію використовуючи Selenium. Для багатопоточної реалізації передбачено відкриття багатьох вікон, кожне з яких на окремому потоці для паралельного збору даних:

def main(threads: int = 4, driver\_name: str = 'Safari', iteration\_to\_save\_data: int = 100) -> None:  
 users = get\_list\_of\_users()  
  
 if threads:  
 with ThreadPoolExecutor(threads) as executor:  
 for th in range(threads):  
 executor.submit(scrape\_followers,  
 [users[i] for i in range(len(users)) if i % threads == th],  
 driver\_name,  
 iteration\_to\_save\_data)  
 else:  
 pass  
  
 data = pd.read\_csv('../../data/twitter/scrapped\_tweets.csv')  
 with open('../../data/twitter/followers.json', 'r') as f:  
 followers: dict[str, int] = json.load(f)  
 followers\_data = add\_followers(data, followers)  
 followers\_data.to\_csv('../../data/twitter/scrapped\_tweets.csv')  
  
  
if \_\_name\_\_ == '\_\_main\_\_':  
 main(threads=1, driver\_name='Safari', iteration\_to\_save\_data=20)

Відповідно, для запуску не паралельної версії достатньо зазничити у якості кількості потоків 1.

Для порівняння швидкодії для одних і тих самих вікон наведений код було запущено декілька разів. Нижче наведено результати, узагальнення та висновки.

Для версії без багатопоточності:

Час виконання: 32 секунди

Час виконання: 28 секунд

Час виконання 30 секунд

Час виконання: 29 секунд

Середній час виконання: 29.75 секунд

Для версії з багатопотоковістю (2 threads):

Час виконання: 18 секунд

Час виконання: 17 секунд

Час виконання: 19 секунд

Час виконання: 18 секунд

Середній час виконання: 18 секунд

Ми бачимо очевидне зменшення приблизно на 39,66%.

Окрім того, результати роботи у вигляді інформації в файлах залишилися тими самими.

Результати бенчмаркінгу свідчать про значне покращення часової ефективності при реалізації багатопотоковості. Варто зазначити, що ці результати можуть відрізнятися залежно від конкретного обладнання, програмного забезпечення та середовища, яке використовується для виконання.

Оскільки всі проєкти, приклади з яких показано у даній роботі є закритими, то посилання на них не додаються, але вся суттєва для цього звіту інформація надана у якості опису та частин коду.