Contents

Answer Design

Overview

This document describes a system that simulates processing live video streams for AI inference. Frames arrive gradually over the span of several days and the video concludes after a few days.

Image generator and handling data streams

The image generator script simulates video frame data arriving in the “incoming_images” directory at random time intervals, as single images or small batches. This simulates a real world data stream, although the timeframe of the arriving data is scaled down to seconds. The time scale can be modified in the config file.

The naive and optimised pipelines both poll the incoming images directory for new images based on a user selected frequency in the config file. This allows the pipelines to monitor if the size of the image directory has changed, whilst decreasing the amount of computational power needed to run these scripts over the days it may take a full video to arrive. A polling system also makes sense because the computational level of the naive and optimized processes are quite low, and the total number of frames per video are also quite small. This could be further improved by having the pipelines run continuously at first, whilst monitoring the rate at which the first few images arrive. They could then begin to poll based on that average frequency which could require less computing power. However, this was outside the timeframe scope of this task.

Naive vs Optimised approach

There are two processing pipelines that are being tested. Both pipelines use a basic inference function to simulate an AI model batch test on a window of video frames. This inference function returns the sum of the average pixel values per frame in the window. As seen in figure 1, the sliding window increases by a single frame as new video frames arrive.

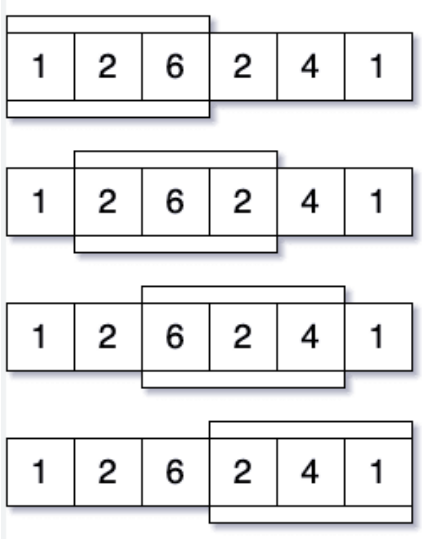

Figure 1: Example of a fixed sliding window but with a length of 3 frames

Each time a new frame arrives in the “incoming_images” directory, the naive approach reprocesses all frames: from frame 0 at the start of the video, to the newest frame. This is inefficient because it doesn’t keep track of the previous frames and windows that have already been processed. The optimised approach only processes the latest window containing the newest frame, and adds it to the processed data stored from the previous windows. This is more efficient because it keeps track of frames and windows that have already been processed. There is a minor trade off for the computing power necessary for the optimised process, since its code is slightly more complex to process and store frame data. However, over an entire poll interaction, it requires less time and fewer inference processes.

Performance and validation metrics

Both pipelines are able to produce validation and performance metrics. The validation metric is the sum of the inference function outputs for each fixed sliding window. The performance metrics show the total number of times the inference function is called in each pipeline, as well as the total time spent computing the inference functions. The total number of necessary computations required is also recorded. The number of real vs. necessary computations and computation time can be monitored live, whereas the

validation metric, total computations, and total computation time can be shown at the end of each pipeline. Additionally, these metric logs can be saved to the performance_graphs directories and viewed once the processes finish.

process	inference_function_totals	total_processing_time	total_processes
naive	952469954.875	1.963526964187622	273
optimised	952469954.875	0.5956711769104004	57

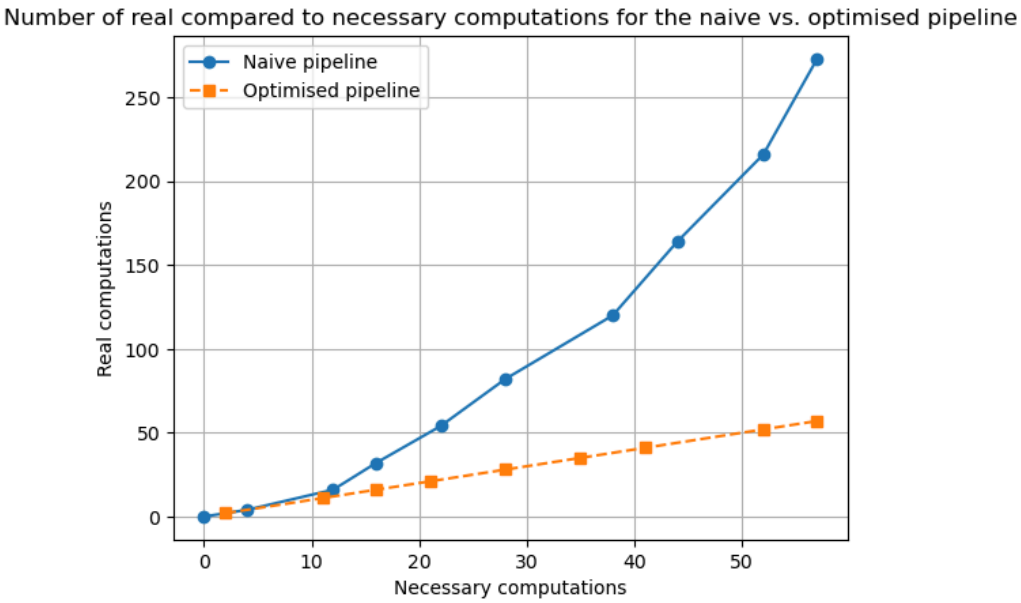

Figure 2: Example of metrics from naive vs. optimised pipelines

These graphs directly compare the metrics of each pipeline. As seen in this example, the inference function totals are equal. This demonstrates the optimised pipeline achieves the exact same accuracy as the naive pipeline. Additionally, the optimised pipeline has a lower total processing time and number of processes. This demonstrates that it is substantially more efficient. As demonstrated by the graph, the naive process's computations exponentially increase compared to the necessary computations; whereas, the optimised pipeline only computes what is absolutely necessary.

Accounting for errors

The image generator function can be directed to include certain unit tests to demonstrate how bad image data, such as duplicate and late images, are handled by the processing pipelines. If the unit_test variable in the save_fake_data function (within the image generator script) is set to "True", two separate random images are selected: one will have a duplicate file saved with "_copy" appended to the file name, and the other will be added to the "incoming_images" directory at the end. If a file is duplicated

without changing the name, it would replace the original file and be considered as a late image. Processing image pixels to find duplicates is beyond the scope of the task because frames are randomly generated.

Late images are identified based on their timestamp compared to previously processed images. Since the naive pipeline does not track previously processed images, only the optimised pipeline is able to account for these late image errors. It does this by identifying a file with a timestamp label that is earlier than the previous file, finds its correct index location by parsing processed image timestamps, and then reprocesses all windows from that index point onwards. Although this requires more computing power, this is an advantageous feature compared to the naive pipeline.

A missing image test was considered to be included; however, it was beyond the timeframe and complexity scope of the task. This would require additional metadata from the real world image generator that could check or alert the processing pipelines of a missing image. Additional tests demonstrated that non-image files are not processed, and the pipelines can still process data even if it all arrives at once.

Scalability

Scalability was a key factor when designing this solution. The processing pipelines are able to run on any directory where live video frames may be arriving. The polling system was implemented to decrease script running time whilst waiting for new data to arrive, in the simplest way possible. It could be further improved for scalability by implementing a queuing system that triggers image processing when the queue gets to a certain size, if computational power needs to be conserved more. Additionally, if data is being stored in the cloud, a process can be used to trigger a processing script once new data arrives. However, this would require more computing power compared to processing images in slightly larger batches via a queue or polling system. It depends what image processing frequency is necessary for the real world scenario.

Conclusion

This document summarises the design choices for the image generator, naive and optimised pipelines, inference function, performance metrics, and validation processes for the simulation task. Given the real world context of this task, it may be unnecessary for an AI process to be run each time it receives a new frame or window from clinical timelapse image (TLI) data. Realistically, new TLI data should be added to a processing queue which only runs in preparation when testing or training is required for the AI model. This is a more optimised solution because testing and training is far less frequent than multiple times a day, and it would use significantly less computing power.

Running the simulation

- Pull the contents of the repository from GitHub to a local machine.
- Open a terminal window and navigate to /code.
- Install requirements.txt
- If desired, modify global variables such as the path to the image directory or other simulation parameters in config.json (not required).
- Run the simulation.py script in python3 from the terminal. 3 additional terminals will be opened and execute the image-generator.py, naive-pipeline.py, and optimised-pipeline.py respectively. Each script can also be run simultaneously in separate terminals manually.
- Monitor the running metrics logs and wait for the simulation to complete. A performance_data directory should be generated during the simulation.
- After the simulation is complete, run the validation-and-performance.py file. This will create a performance_graph directory containing figures of the metrics.

Additional running notes

- If the unit-test variable in the image generator function is set to True, it is necessary to manually re-run the naive pipeline script with reorder_for_late also set to True and manually input the missing image information. This is due to the computational limitations of the naive pipeline which cannot process late images during a simulation (therefore it also gives an incorrect validation metric value at first).