Data Caching

Bottleneck Identification and Analysis

During the initial training of deep learning models using the data generators, a significant bottleneck was identified when generating new batches of data. Profiling revealed that the *load_img* method from Keras was responsible for this inefficiency, taking approximately 0.5 seconds to load each image on average. Given that the training dataset contains roughly 120,000 samples, this bottleneck could lead to a processing time of approximately 60,000 seconds (~16.67 hours) per epoch, solely for invoking the *load_img* method, assuming all samples are used in each epoch.

The original data generator randomly selected an image file and sequentially loaded it along with the subsequent nine images. Each of these images was processed using the *load_img* method, followed by conversion into an array format with the *img_to_array* method, both from Keras. This approach, while functional, significantly hindered the training process due to the time required to load and preprocess each image individually.

Initial Attempt to Alleviate the Bottleneck

The first approach to address this bottleneck involved eliminating the *load_img* method by converting all images in the dataset into their corresponding NumPy array formats. The *np.load* method was then used to load the selected images and their subsequent frames. However, profiling this approach revealed an unexpected increase in the average time required to generate each batch. The underlying issue persisted, as the data generator still had to read each sample from disk—a process identified as the true source of the bottleneck. Additionally, the resulting NumPy array dataset was more than 15 times larger than the original, making this solution impractical. Consequently, this approach was abandoned.

Effective Solution through Caching

To address the bottleneck caused by the *load_img* method, the initial approach involved converting all images in the dataset into NumPy array formats. The *np.load* method was then employed to load the selected images and their subsequent frames. However, profiling revealed an unexpected increase in the average time required to generate each batch using this method. The ultimate solution was to implement image caching, which significantly reduced the

overhead associated with reading images from disk. By accessing pre-cached images, the time-consuming operations related to disk reads during training were minimized.

As seen in Table 1, caching resulted in an impressive speedup of nearly 95X compared to the baseline image-loading approach. Moreover, caching not only improved execution time but also reduced the overall storage size of the dataset, offering greater efficiency. A summary of the results, including the average execution times, speedups, and dataset sizes for each method (original image loading, NumPy arrays, and cached images), is provided in Table 1.

	Cache	Images	Numpy Arrays
Average Execution Time (s)	**0.642**	54.938	68.710
Speedup (X)	**94.986**	1.000	0.814
Dataset Size (GB)	**1.494**	2.141	37.172

Table 1: Average execution time, speedup, and dataset size for loading the dataset as cached images, images from disk, and NumPy arrays from disk.

Consequently, this caching technique is recommended for all users of the TF-66 dataset.

To implement this solution, a script was developed to create the cache. Once the script is executed and the caches are generated, they can be reused for all future operations with the dataset, eliminating the need to recreate the cache. The script systematically traverses the TF-66 directory, identifying all *.jpg* files within each directory (in this case, *train* and *validation*). Each image is then loaded using the *load_img()* method, resized to the target size (256x256), and converted to the specified color mode (greyscale). The processed images are stored in a cache dictionary, with the image paths as keys and the processed image arrays as values. The cache dictionary is then saved as a compressed *.npz* file using the *np.savez_compressed()* function. The script returns the cache dictionary, containing all the processed images. Upon completion, two cache files are generated: *train_cache.npz* and *val_cache.npz*.

These cache files can be loaded whenever the dataset is used, thereby eliminating the bottleneck and enabling efficient training times. To load these cache files, simply invoke the *np.load* method.

To be clear, although this method effectively eliminates the bottleneck, each researcher intending to use the cache method (as recommended) must first generate the cache on their local machine. The cache is dependent on the specific file path structure and directory where it was created. As a result, a cache generated on one researcher's machine will not be transferable to another researcher's machine with a different directory structure. Each

researcher must generate a unique cache specific to the dataset location on their system. For instance, if a researcher creates their cache with the dataset stored on Google Drive, this cache cannot be used on another system that does not have access to Google Drive—such as the Digital Research Alliance of Canada. In such cases, the cache must be regenerated within the Digital Research Alliance of Canada environment to ensure the file paths correspond to the accessible directories.

Cache Sharing Considerations, Challenges, and How to use the Cache

To address the need for creating a cache to efficiently work with the dataset, an attempt was made to cache the dataset on Google Drive in the base directory (*MyDrive*), and then share both the dataset and caches with another user. The goal was to enable other researchers to automatically utilize the cache in a Google Colab environment, offering a quick plug-and-play solution without requiring them to download the data or generate the cache themselves.

The process involved sharing the cache files and the entire dataset folder from the sender to the recipient. In the recipient's Shared with me folder in Google Drive, they could right-click on the dataset folder, select Organize, and then Add shortcut, thereby adding it to their *MyDrive*. This setup ensured that both the sender and the recipient had access to the same cache and could use the dataset through an identical directory path. However, during testing, the recipient was unable to load images from the cache or train subsequent models using the cache, even though the file paths printed from the cache matched those found in their Google Drive. The reason for this issue was not explored in depth, but it is likely due to subtle differences in how Google Drive handles shared file paths or possibly due to differences in access permissions or file indexing. It is hoped that future work will address this limitation. In the meantime, and for researchers seeking an alternative to Google Colab, the script to create the cache can be found on the Github page with the file name ``Create Cache.ipynb''. Notably, you need to change the “train_data_dir” and “val_data_dir” attributes in this file to match where you have the dataset saved.

Once the cache has been created, it can be accessed for machine learning models. Examining the “DataGenerator.py” file, one can see that the cache files must be loaded into attributes, “train_cache_file” and “val_cache_file” on lines 39 and 40, respectively. These must be changed to match your specific file path to your created cache. Lines 183 and 184 instantiate the train and validation generators, using these caches.