Improving Dataloaders for Visuomotor Imitation within Robomimic

Davin Lawrence
Computer Science
University of Texas at Austin
Email: dhuck@cs.utexas.edu

Mihir Suvarna
Computer Science
University of Texas at Austin
Email: mihirsuvarna@utexas.edu

Adeet Parikh
Computer Science
University of Texas at Austin
Email: aaparikh@utexas.edu

Ajay Mandlekar
NVIDIA Research
Stanford University
Email: amandlek@cs.stanford.edu

***Abstract*—Robomimic is an imitation learning framework for robot manipulation. The unique data, which consists of a combination of video, trajectories, and actions that together encode spatio-temporal information, necessitates task-specific data loader optimizations to reduce the overhead required for training and inference. The current method of loading data in Robomimic pulls all data from a common store and must work to split and treat each data item separately. To alleviate this independent style of loading, we try existing low-level techniques, including various types of caching with compression, and compare the times spent data loading to try and mitigate total time spent. Additionally, we showcase our novel addition of PyTorch data pipelines, which generate a significant speed up in time spent loading data, and document our tests with results. Despite this framework still being in beta, we hope that Robomimic will adapt this technique going forward as it has promising results.**

I. Introduction

In the existing Robomimic [**?**] framework, dataloaders are used natively from PyTorch to set up all machine-generated, human-generated, and multi-human data that is used throughout the model pipeline. Input data is provided in the form of HDF5 files, which is a format that is designed for large heterogeneous datasets. The HDF5 files used within Robomimic contain a mix of data from multiple cameras, low-dimensional observations of information from the environment, and state-action-reward information. As this data is inherently large (scaling beyond 50 GB in some cases), loading and feeding it into the framework takes a large amounts of time. Our task within Robomimic is to take a deep look at how the dataloaders are set up, and come up with various methods that can help speed up this process for the task of visuomotor imitation.

The multi-modal data found within a typical Robomimic dataset presents a unique challenge for data loading. In their typical implementations, data loaders deal with heterogeneous data, such as images, text, or video. In this context, the most difficult task is ensuring all data is of similar dimension, or handling for variable length in audio, text, or video. Imitation learning datasets can contain multiple sources of information as described in the previous paragraph. Each data point must be loaded and pre-processed separately and ruins any locality-aware processing available in heterogeneous datasets.

Before we began our research, we realized that there were a few key problems at hand. First, as mentioned above, all the data is stored in the HDF5 format. This forces us to treat samples individually, which takes away opportunities for batch loading. Secondly, since Robomimic is a framework consisting of several algorithms, any solutions that we would be implementing in Robomimic must generalize well across these algorithms. Our task at hand was improving dataloaders for visuomotor imitation within Robomimic, and any researcher should be able to take advantage of the optimizations we make. We also had to ensure that any modifications to data loading does not negatively affect the performance of the current framework. Training can sometimes take hours, and in our case, we were limited to a single machine with an NVIDIA RTX 3090. To test whether modifications to data loading were statistically significant, we had to let the whole framework train and ensure that performance did not degrade. Another key point was to guarantee that any changes we make are hardware-agnostic; virtually anyone should be allowed to train, on various machines.

In this paper, we explore various strategies for improving the data loading time within Robomimic. We begin by reviewing related work in section **??**. Section **??** describes the current caching strategies already present in Robomimic and our own caching strategies. Section **??** describes are primary contribution to Robomimic–implementing the TorchData data pipeline and the robo file format. We find that using the TorchData data pipeline significantly improves the time spent loading data from disk. Section **??** describes the results of both our caching and data pipeline implementations. Finally, we describe some further work and conclude.

II. Related Work

Our first research attempt was to review the related work laid out before us on the task of data loading. We found that a lot of work done on dataloader research was within the last decade, and not as extensive as we had hoped it to be. Instead

of getting techniques from these papers, we were able to find high-level big picture overview ideas that pointed us in the right direction to proceed. Since data is permutable and can be represented thousands of ways, there's no one-quick-trick that is proven to speed up data loading, rather multiple techniques that must be applied and thoroughly tested.

The overwhelming majority of AI and robotics research is focused on algorithmic development and their implementations. As a result, the literature is quite sparse when it comes to the topic of data loading. Of the handful of papers we could find in relation to data loading, two of them were published this fall.

?] discuss various techniques to improve the default PyTorch dataloader, both in the context of single and distributed GPU training. Their approach depends on caching either to RAM or disk before or during training. This caching increases loading speed, but also enables locality-aware data loading scheme which greatly increases throughput on multiple GPU training. While caching is indeed part of our approach, we found caching strategies alone to be insufficient to increasing the data loading performance for Robomimic. Importantly, **?**] are focused on data involving a single modality with datasets which are more likely to be partitioned easily into RAM caches.

?] provides a survey of current data loading implementations for PyTorch. The authors compare seven data loading libraries: Squirrel, PyTorch, TorchData, FFCV, WebDataset, Deep Lake, and Hub. Notably, they compare these dataloaders both on multi-GPU setups as well as single GPU machines. Furthermore, the authors compare appropriate dataloaders' performance when pulling data from a network file system, which has interesting applications for cloud training. We take a similar approach by trying dataloaders outside of the default PyTorch dataloader. Due to the high development cost, however, we only successfully implemented the TorchData dataloader. A key difference between **?**] and our work is the underlying data; they use ImageNet and CIFAR datasets while we are optimizing for multi-modal, complex data.

There are a handful of other data loading approaches which informed our work but did not directly affect it. **?**] describe a grid search strategy for fine-tuning the PyTorch data loader. We believe this approach would be too costly for any benefit in the context of Robomimic. **?**], **?**], **?**], and **?**] all describe approaches for speeding up training ResNet on ImageNet. All of these papers describe a mix of data loading improvements on top of algorithmic heuristics to improve ImageNet. **?**], **?**], and **?**] all describe the process of dataset distillation, which is the process of identifying similar trajectories or data points within a dataset which are combined with the goal of minimizing the size of the dataset without affecting the accuracy of training. While we were not able to implement these techniques due to time constraints, we believe that distillation of Robomimic data could further improve data loading.

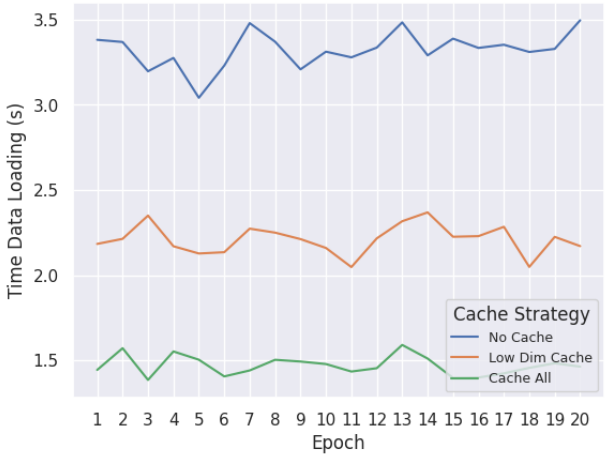

Fig. 1. Comparison of current caching strategies in Robomimic on the Lift PH dataset. This test performs with a sequence length of 50 with no batching.

III. Cache

Caching is a popular strategy to utilize the hierarchy of memory in a computing system. Nearly all of the papers discussed in section **??** utilize some form of caching for their performance gains. In this section, we discuss the built-in caching strategies found within Robomimic's SequenceDataset. We then describe the two caching techniques we pursued–LRU caching of data and a novel compressed cache.

A. Current Caching Strategies

As mentioned above, native Robomimic comes with three caching options. No cache means absolutely nothing is cached; the dataset is independently loaded into memory and the OS decides what to evict and keep. The low-dimensional caching specifically caches all the low-dimensional observations; i.e, robot position, velocities, object position information, and other data. Caching the full dataset, in this framework, is caching all the low-dimensional information plus the images that are used for training.

For us to have any improvement over these strategies, we needed to find a good benchmark. We performed an initial test loading in a batch size of 1 of sequence length 50 as seen in Figure **??**. As expected, we see the performance of caching the entire dataset outperforms the other methods, with low-dimensional caching outperforming no cache. In our initial testing, we find that the in-memory storage of a dataset has an 8x increase from the data stored on disk with the HDF5 file. This is due to images being stored as chars in the dataset while being converted to 32-bit floats in memory. Since caching entire datasets is infeasible for all but the smallest available datasets, we use low-dimensional caching as our baseline target for all of our approaches.

Interestingly, we notice some variability in the data loading time. We suspect the slowdowns in these methods are due to variance to system level caching as well as the method of querying data in the SequenceDataset. Some sequences require

padding, which can incur a computational burden when pulling data from disk or the cache.

B. LRU Cache

LRU cache was the second technique we tried. Any cache requires an eviction policy to determine which objects reside in memory. A LRU cache uses a policy which evicts the last recently used object to make room for new objects in the cache. This is a popular option when designing caches and seeme like a natural option for our task, as LRU caches keep what's been "recently used." However, upon implementing our LRU cache, we found a 2 - 3.5x slowdown in performance. Our initial testing showed that with the large datasets, especially for Machine-Generated (MG) tasks, LRU cache performed quite poorly. We believe this is because the LRU cache has no way of knowing which objects are currently being used. During training, samples are selected at random, so it is impossible to design an eviction policy that can effectively choose which samples to keep. The overhead is primarily due to repeated eviction with most attempted cache reads being misses.

We did not pursue training further with LRU and abandoned this approach early on after noticing the large amounts of time that it spent training; it was deemed a waste of computation time, as it correlated with a decrease in efficiency, and we decided to focus our research efforts in other dimensions, specifically with compressed caching and training.

C. Compressed Caching

Working under the hypothesis that decompressing information from memory would be quicker than reading from the HDF5 file, we implemented a compressed cache technique with the goal of being able to fit the majority, if not all, of a given dataset. While we were able to fit entire datasets into memory, we found the devil was in the details of implementation and could not beat the performance of low-dimensional caching alone.

For both of our caching strategies, we targeted the RGB camera observations as the compression targets. We ignored the low-dimensional observations since they are small and therefore not appropriate candidates for compression. The overhead for compressing and decompressing the low-dimensional spaces was not worth the minuscule gain in space efficiency.

Our initial compression approach involved using `lz4` compression due to its speed and wide library support. We were able to compress Numpy data arrays directly into memory but found that data typing and other meta data was lost on this approach. Furthermore, compression required us to modify the data structure in such a way that good patterns could be found for compression techniques. Finally, we found that reconstructing the Numpy arrays post decompression was non-trivial to implement and severely offset any gains from compression. Early experiments for this approach found significantly worse performance than not caching at all.

The final compression approach was to use the Numpy `savez_compressed` method to cache the Numpy arrays. This resulted in a much simpler implementation due to the Numpy being able to hint and reconstruct its data typing and internal structure. Moreover, using this method allowed us to save multiple observations in a single compressed archive rather than having to compress each observation separately. This approach resulted in nearly an 8x decrease in RAM usage, allowing larger datasets to be stored in memory and disk when using a flexible swap space. Despite these gains, we found our compressed cache strategy performed only slightly better than the no caching approach. This is primarily due to the complexity of implementation, namely the fact that we must deep copy every compressed buffer before decompressing, otherwise the compressed buffer would be expanded in memory. For larger batch sizes, this issue compounds since every item in the batch must be copied, expanded, and temporarily stored in memory.

IV. Data Pipelines

TorchData [**?**] is a new library from the PyTorch team which creates a new paradigm for loading data in any machine learning workload. The goal of these data pipelines is to create a modular approach to data loading and preprocessing rather than inheriting and implementing a novel Dataset class as is the common practice today. This approach allows for flexibility in implementation, while creating new opportunities for parallelism and caching. For example, any complex preprocessing step can be explicitly cached either in memory or on disk. In multi-modal setups such as Robomimic, different types of data can take different paths through the resultant data flow graph. While our implementation of the TorchData is quite nascent, we find re-implementing the SequenceDataSet class as a data pipeline creates a significant gain in data loading efficiency.

TorchData comes with one major caveat: it is still considered to be in beta. The library page states that the API is subject to change without warning. At time of writing, TorchData 0.5.0 is the most recent release of the data pipeline API. Since there is no timeline as to when TorchData will be considered stable, we recommend using it as an alternative to the SequenceDataset set in the configuration JSON rather than an outright replacement for the current SequenceDataset class.

A. SequenceDataSet

In this section, we will briefly describe the current dataset implementation used by Robomimic. The SequenceDataset provides an interface to the HDF5 store of information for each dataset provided by Robomimic. It performs quite a bit of preprocessing and stores a significant amount of state used for retrieval and preprocessing. The two caching strategies described in Section **??** are implemented in the constructor of this method and used in the necessary `__getitem__` call.

Most importantly, the SequenceDataset abstracts away the idea of individual demos, allowing the user to query an arbitrarily long sequence from any point in the dataset. Therefore, the length of the dataset is not the number of demonstration, but rather the total number of individual steps in each sequence provided by each demonstration. Upon loading the HDF5 file,

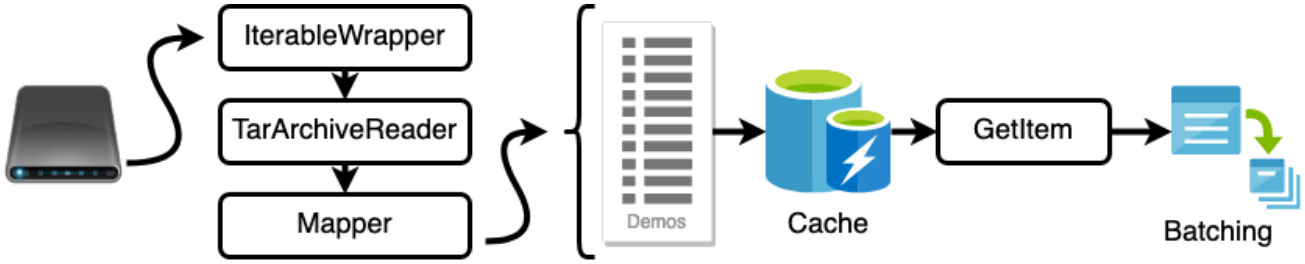

Fig. 2. Flow chart demonstrating the various stages of the current TorchData pipeline implementation. TorchData's `IterableWrapper` and `TarArchiveReader` functions open and provide access to the robo file. The first `Mapper` function unpacks the data into dictionaries that can be handled as a key-value store in an adjustable-size, optional cache. Values are then provided to a `get_item` function modelled after Robomimic's original data set implementation. Finally shuffling and batching is handled inside of the Data Pipeline rather than the torch Data Loader.

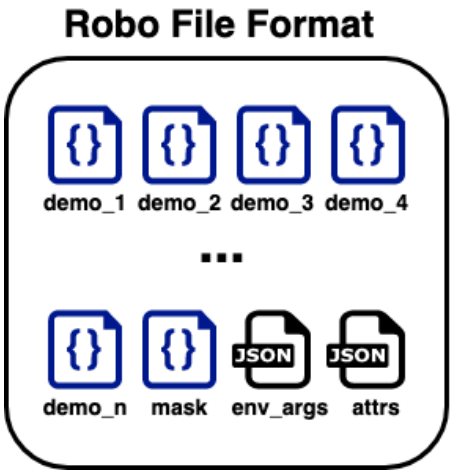

Fig. 3. Robo File Format

the SequenceDataset calculates an index into each dataset for each step of the dataset and maps this in a dictionary which stores an index to demonstration key. This allows for a small number of demonstrations to become quite large with intermediate sequences for movement and success. Whenever a DataLoader calls `__getitem__` on a SequenceDataset, the index is used to calculate a starting index within a demonstration, which then returns a subsequence from the demonstration.

B. The robo File Format

A significant amount of the labor of implementing the new data pipeline approach was in developing a novel file structure for use in the data pipeline. The HDF5 file format is incompatible with torch Data Pipelines since it is not pickleable. Using the `h5pickle` library as a workaround resulted in worse performance than the original approach and was quickly abandoned. We also attempted implementing the robo file set using JSON objects, but quickly realized this created large datasets where every piece of information had to be translated from a string back into an int or float before being used by the data pipeline.

The `robo` file format is based on lessons learned while implementing compressed caching as described in Section **??**. Each demo key from the HDF5 file is extracted and converted into a npz file by flattening the dictionary pulled from the HDF5 file and storing it as a npz file using the `savez` method in Numpy. The same procedure is used for the mask dictionary, which describes various training/validation splits used in benchmarking. Finally, we create two JSON objects describing metadata gathered from the HDF5 file. The `env_args` object stores environment metadata, while the `attrs` object stores metadata gathered from each individual demo. The `robo` file format is illustrated in Figure **??**.

There is no reason the current Dataset implementation in Robomimic couldn't use this file format instead of the HDF5 file, outside of time spent in development. This development may be worthwhile since it would elucidate the gains from the file format alone versus gains from the Data Pipeline implementation. Additionally, TorchData has the functionality to decompress files as part of its pipeline. Compressing the individual Numpy objects using `xz` or `gzip` could provide significant space savings on disk and time spent downloading datasets.

C. Data Pipeline

The current implementation of the TorchData data pipeline is outlined in Figure **??**. The process begins with the robo file as described in the previous section. TorchData handles the unpacking and navigation of the tar file in the `TarArchiveReader`, providing a list of objects inside the robo file to nodes downstream. Before reaching the Mapper, a Demultiplexer is used to split off the different document types inside of the robo file. The Mapper is only used on the demo objects from the robo file. This provides a simple function which reconstructs the Numpy objects as dictionaries. Finally, the demo objects are placed into an optional, adjustable-size cache as a key-value store to be used by a final `get_item` function which provides the same functionality as the current Dataset implementation.

The data pipeline describes a data flow graph where data flows across edges to computational nodes. The current implementation is the simplest form of a graph, a direct line from source to output, but other configurations are possible. For example, different modalities can be processed by unique pipelines and cached separately, leading to greater utilization

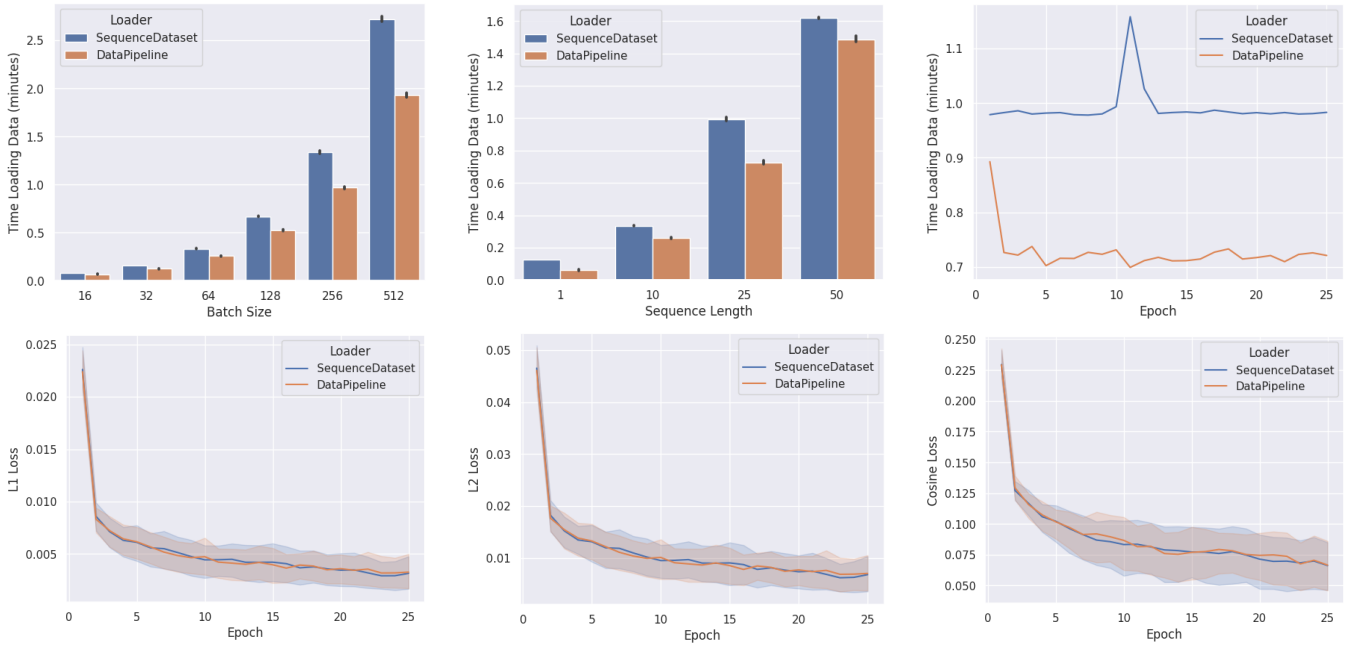

Fig. 4. All visualizations presenting in this figure are on the Lift PH dataset with a epoch of 100 steps for 25 epochs. **Top Left:** Comparison of SequenceDataset and DataPipeline varying the batch size. For these experiments, sequence length is fixed at 10. **Top Center:** Comparison of the two data loaders using different sequence lengths. for each of these experiments, the batch size is fixed at 64. **Top Right:** Time spent data loading per each epoch in a 25 epoch run. **Bottom row:** Average L1, L2, and cosine similarity loss for for both of the data loading methods across all batch size and sequence length variations.

of system memory. Currently, the only caching strategy is to cache as soon as data is pulled from the robo file, but a caching strategy which mimics the current low-dimensional information caching could easily be implemented.

Much of the development work for our initial data pipeline re-implements the methods found in the SequenceDataSet class. This is essential to allow the data pipeline to be used as a drop in replacement for the SequenceDataSet. Further enhancements could be made by fine-tailoring these functions for use with the `robo` file format and vice versa.

V. Experiments

In this section, we present our findings on the performance gains given by the current data pipeline implementation. All experiments were run on a dedicated machine with a Ryzen Threadripper 1900x 8-core 3.8GHz CPU, 32 GB of DDR4-2133 RAM, and a NVIDIA RTX 3090 GPU. For each test, any system or user software which utilizes the GPU was disabled, ensuring there was no contention for GPU resources during the tests. For most of the tests, the lift PH dataset was used since it is small enough to run quickly and to be cached completely into memory without reverting to swap. Furthermore, while we believe that the DataPipeline should generalize regardless of the algorithm, all tests are performed using the Behavioral Cloning implementation in Robomimic.

A. Compressed Cache

For testing the compressed cache, we fixed the batch size at 64 and queried sequences of length 60 from the lift PH dataset. The sequence length of 60 was chosen as this is longer than

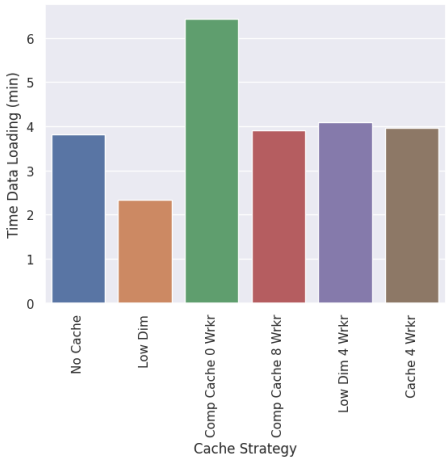

Fig. 5. Comparison of Compressed Caching to the original low-dimensional cache already implemented in SequenceDataSet and no caching at all.

all but a few of the demonstrations present in the dataset. Ideally, this would maximize the gains from decompressing in memory, since the entire demonstration will be used upon being decompressed rather than just a small fraction of the dataset. Results were collected and averaged over 50 epochs with 100 steps for each epoch.

The results in Figure **??** show there is no performance gain in performing the compressed cache. Even when adding additional workers, we find the performance does not compare to even the no cache approach already used in the SequenceDataset. Interestingly, we also find the low-dimensional cache

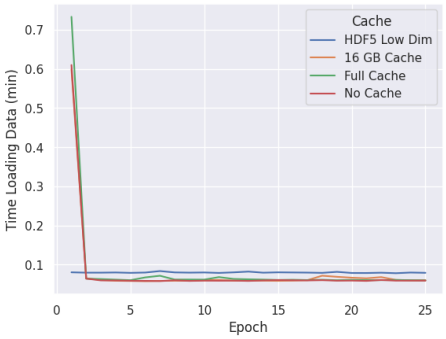

Fig. 6. Comparison of Data Pipeline Caching

suffers from increasing the data loading worker count. We hypothesize this is due to the increased memory pressure due to synchronization between workers and a large sequence size.

B. Data Pipeline

For the data pipeline tests, we tested performance by varying the sequence length and the batch size. We believe that the sequence length will have varying performance gains depending on the hit rate of a full sequence versus a sequence which needs to be padded. Conversely, fixing the sequence length and varying the batch size should have a consistent performance profile for each batch size. Due to time constraints, we only train for 25 epochs with 100 steps per epoch. We include the L1, L2, and cosine similarity loss as measures of convergence to ensure our data loading approach converges similarly to the SequenceDataset. We did not train or analyze success rates as the time burden would have limited the breadth of experiments we were able to perform.

Figure **??** shows the results from these experiments. We find that the DataPipeline implementation consistently outperforms the SequenceDataset implementation. This is most notably consistent across the all batch sizes, with greater gains coming from large batch sizes. For sequence length, we see some variance in the gains depending on the sequence length. Additional tests could be performed, but we believe this is due to hit rate on sequences which are fully in a demo rather than sequences which require padding. The line graph in the top right of figure **??** highlights an important characteristic of the DataPipeline implementation. For every workload, the gains are amortized across longer training runs. The first epoch always incurs a heavy loading cost.

To ensure that our DataPipeline implementation is not custom fit to the lift dataset, we also perform tests on the square MH dataset, which is approximately ten times larger than the lift dataset. Moreover, using a larger dataset allowed us to test the in memory caching stage of our pipeline. To this end, we ran the test with the cache fully enabled with no size restraint, restrained the cache to 16 GB, and turned the cache completely off. We also ran the same test using the SequenceDataset with the low-dimensional cache strategy for comparison.

The results for this experiment are found in figure **??**. The amortization of the loading time in Data Pipelines is more visible in this approach than in the previous experiment. We find that this is even more pronounced with a larger cache than without, though it is present in all workloads. In comparison, the SequenceDataset does not incur such a cost. For this experiment, the SequenceDataset run was on average 20 seconds faster than the DataPipeline. This suggests that the DataPipeline implementation still has work ahead of it to generalize across all datasets. However, in a realistic training scenario with many more epochs, the cost of the initial loading of the DataPipeline should be amortized.

We would like to address the initial results presented in the poster session prior to this report. In this presentation, we showed a graph which suggested much higher efficiency gains than presented in this paper. The results from the presentation were due to a misunderstanding in how sequences were loaded from the dataset into memory. As a result, our data loading approach was not able to converge with that implementation. Fixing our implementation required additional pre-processing steps which clearly affected performance. We expected a decrease in efficiency from the results presented previously.

VI. Further Work

While we believe we have made great strides in improving dataloaders for Robomimic, there is further work that we would like to explore at a later date. One such research initiative is to explore other Data Pipeline configurations. As the library is still in beta, we anticipate there to be more configurations available throughout its development and eventual release. Presently, there are other ways we could organize the data flow graph to allow different forms of data to follow their own paths through the graph. This approach would allow for better parallelization and better caching opportunities.

Another initiative will be for us to study the effect of observation sizes on training. This is an approach we anticipated reporting on in this paper, but we were not able to due to time constraints. As expected, the largest aspect in the datasets is the image RGB data. Finding methods to cut down on this size would greatly increase the flow of data from disk into the training loop. In the future, we would like to lower the dimensions of images and videos (say from 84x84 to 28x28) and study the effects on training time versus loading time.

Going past the scope of working on our profiled RTX 3090, we would like to also perform some large scale testing at some point to ensure that our results were statistically significant. We had limited access to compute resources over the course of this research, and in the future we hope to test on a parallel-GPU architecture and assess how well it performs.

VII. Conclusion

We conducted a number of experiments to quantify our efforts to improve dataloaders for visuomotor imitation in Robomimic, and found varying results for different techniques. We first performed detailed profiling of the Robomimic framework to understand exactly how time was spent during

training, and found that majority of it was spent loading in the data in each epoch.

We explored compressing the dataset and caching, working under the hypothesis that decompressing the information from memory would be quicker than simply reading from the HDF5 file. Upon testing this in our experiments, we found that other memory issues, such as buffer copying and space restraints, imposed a limit to any gains we had seen in data loading, essentially bringing the data loading speeds back to having no cache at all.

Finally, we implemented a novel data loading approach using PyTorch Data Pipelines, which discarded the old HDF5 format and resulted in the biggest delta for data loading in Robomimic, with all other components of the framework being left intact. Despite Data Pipelines being in beta, we believe that future releases of Data Pipelines, with numerous configurations, will greatly speed up data loading and total training times, and is a viable addition to a future release of Robomimic.