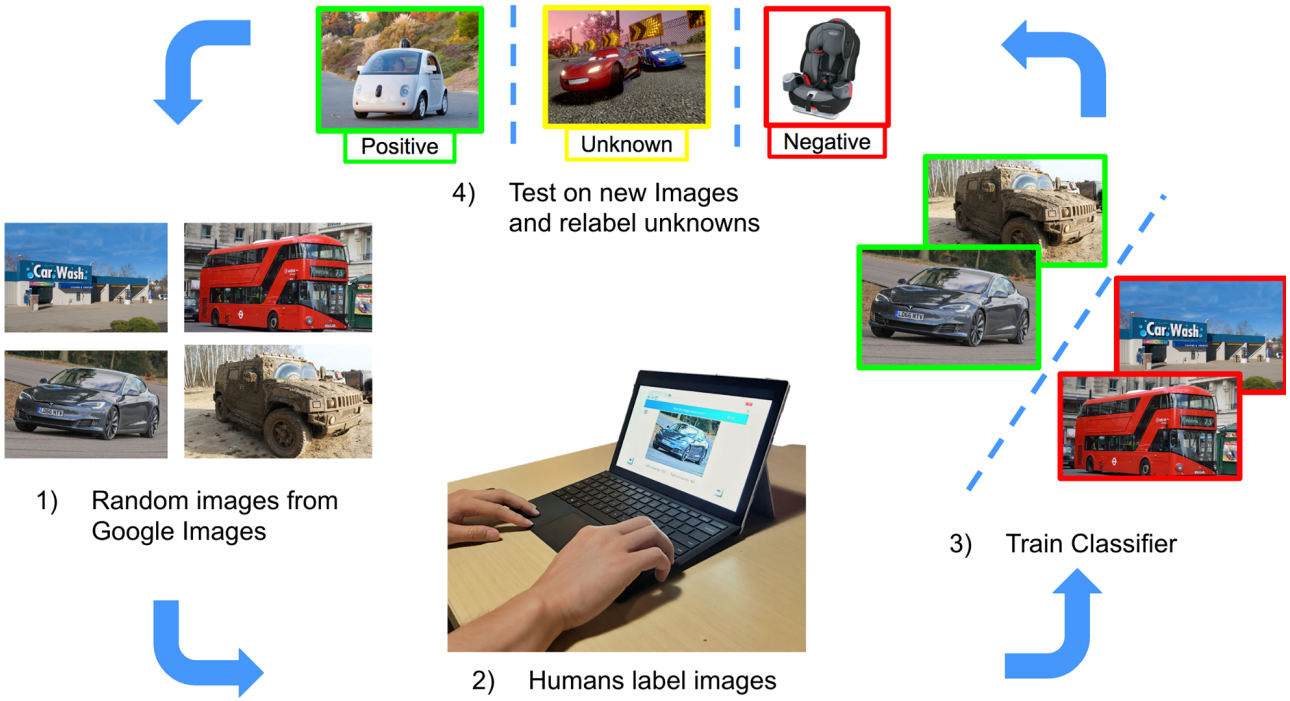
Accelerating the Creation of Larger Scale Image Sets

Berkeley DeepDrive: Active Learning Team

May 2019



Team Name: Berkeley DeepDrive: Active Learning Team

Team Members: Daniel Benniah John, Ryan Peck, Junsheng Pei, Louis Renaux, Griffin Wu, Matthew Yu

Advisors: Prof. Trevor Darrell, Fisher Yu, Gary Chen

Contents

1 Executive Summary

A robust image classification system is an important milestone in the path towards safe and reliable autonomous cars. Autonomous cars rely on vision systems that can distinguish between people, street signs, and other obstacles. To allow this, algorithms must classify the image data fed to it by the cameras. These machine learning algorithms use deep learning, a technique that requires large amounts of labeled images. Unfortunately, it is expensive to collect accurately labeled images since they must currently be labeled by humans. Generally, deep learning algorithms perform logarithmically better with more data, so creating progressively larger image sets is very beneficial to the industry [1]. To assist in the development of large labeled driving image sets for the training of image classification algorithms, our team has developed an open-source image dataset creation pipeline that will augment human labeling efforts with machine learning algorithms. Our pipeline scrapes unlabeled data from the internet, serves some subsets of the data to humans for labeling, and then can use that labeled data to train a model that automatically classifies new datapoints. This results in datasets with many more labeled images than possible if generated with only human effort. It is our hope that our work will accelerate the development of better deep learning models so that cars can be safer and many lives can be saved.

2 Introduction

2.1 Our Niche in the Autonomous Driving Industry

Autonomous driving offers the world a safer, more efficient method of transportation. According to a 2017 RAND Corporation paper, debuting autonomous vehicles earlier rather than later has the potential to save hundreds of thousands of lives over the first 30 years of adoption [6]. For now, the industry and technological development of autonomous vehicles is very fragmented. The biggest tech companies Uber, Google's Waymo and Baidu as well as car manufacturers Tesla, Mercedes and GM, have all started their own projects [8], and established their own frameworks and communication protocols. Even though some of the hardware necessary for autonomous vehicles, such as the LiDAR sensors, is similar industry-wide, most of the software is done in-house with no industry standards established [7]. This slows down innovation as any new efforts in the autonomous vehicle industry must start from the ground up. Very little open source software is available for engineers or researchers to use.

Large image datasets are particularly difficult to find. Figure 1 is a graph from a recent Google study which demonstrates the usefulness of more data in increasing the average precision (AP) of image classification models. As seen below, the chart extends to 300 million images. In comparison, ImageNet, one of the largest freely available image sets, is only 14 million images [12]. An open source image dataset creation pipeline will allow researchers and smaller companies access to data sets previously only available to large companies, thereby leveling the playing field and accelerating progress.

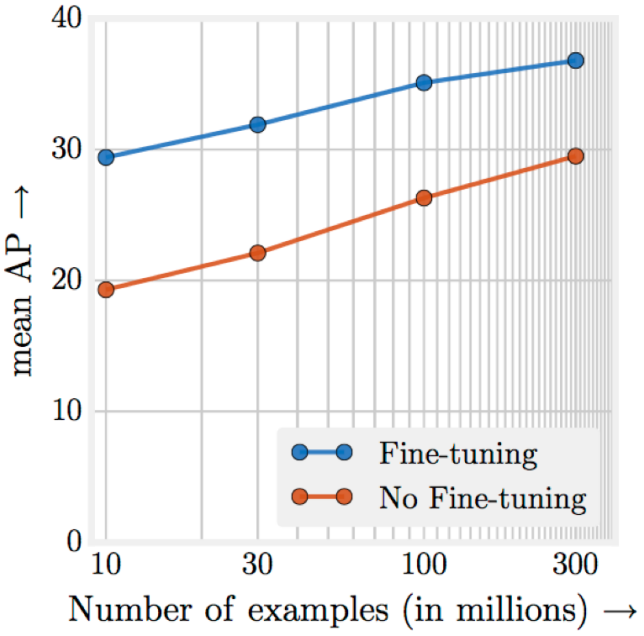


Figure 1: Performance increases logarithmically with data [1]

2.2 Previous Work and Acknowledgements

Our work is based on the methodology used to create another large image dataset called "LSUN" [9]. The LSUN pipeline was originally motivated by the same problem we hope to address, a need for larger labeled data sets. There are four main stages. First, we harvest a large number of unlabeled images through Google Images, a technique

called web scraping. After obtaining the images, human workers will manually label a selected subset with binary "Yes/No" labels specifying the answer to simple questions such as "Does this image contain a car?". We require human inputs in the pipeline because the deep learning models are trained on these labels to "learn" how to classify unseen images.

After training, the models are tested by classifying a new subset of images that were not included in the training set. The models will either classify the image into one of several classes (e.g. dog, cat, car) or as an unknown. After cycling a certain number of images through the model, the unknown set is collected and sent back for humans to label. As we cycle through the LSUN pipeline, we lessen the amount of unknown images and strengthen our classification algorithms. A visual representation of the pipeline can be found in Figure 2.

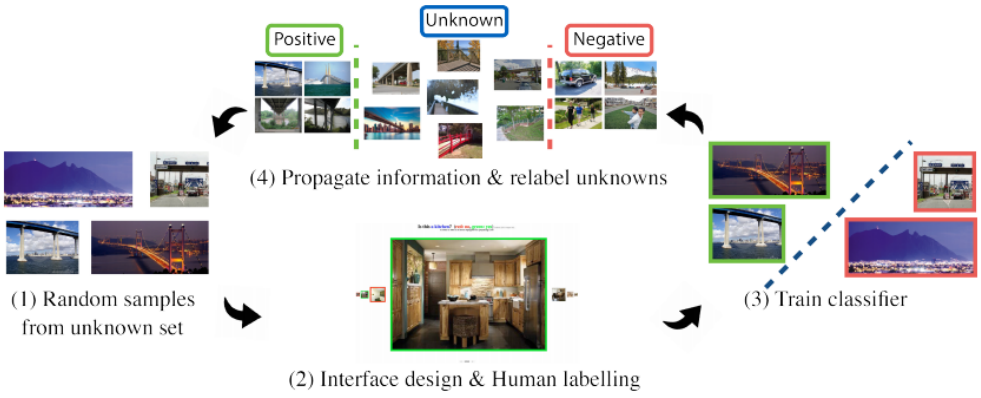


Figure 2: The original LSUN pipeline [9]

2.3 Our Work - A New Framework

Before we started our capstone project, an existing implementation called Visual Factor Graph (VFG), was already developed by Berkeley Deep Drive. Our project improves the VFG framework by addressing issues of scalability, reliability, and modernization. We have created a front-end interface that can be used to collect labels from human workers, using modern web technologies. In addition, we have developed a data infrastructure designed to download and store the millions of unlabeled images, and serve them for labeling. We did not make changes to the machine learning algorithms used in the original pipeline. The following sections will discuss in further detail the implementation of our framework and improvements to the design. We separate our implementation detail discussions between the human labeling system and the back-end infrastructure that holds all of the images. We will discuss the reasonings behind our decision implementations backed by concrete results, as well as alternative options we explored. Finally, we will end with concluding remarks and impact on industry moving forward.

3 Collecting Labels from Humans: The Front-End System

3.1 Problem - Dealing with Obsolete Code

The front-end interface collects labels for a set of unlabeled images by asking human workers to classify them. The labels are binary labels answering yes/no to the classification questions "does this image contain certain 'object'?" For example, in an image dataset full of unlabeled cars, a classification question presented to a worker would be "does this image contain a car?", similar to what is shown in Figure 6.

When building the front-end interface, we placed a high priority on creating something that would be simple to use, be easy to maintain, and remain relevant for as long as possible. The previous front-end interface contained older technologies such as NGINX that would not be easy to adopt to large scale applications and were relatively difficult to develop on parallel computing. We selected React as our JavaScript framework due to its ease of use and popularity. Figure 3 shows that the popularity of React has been steadily increasing for the past three years. React uses components, which can be thought of as portable, independent pieces of a user interface. It replaces HTML with Javacript, simplifying the whole website architecture. React's simplicity makes it easy to upgrade these self-contained components and transfer well-built components between different interfaces.

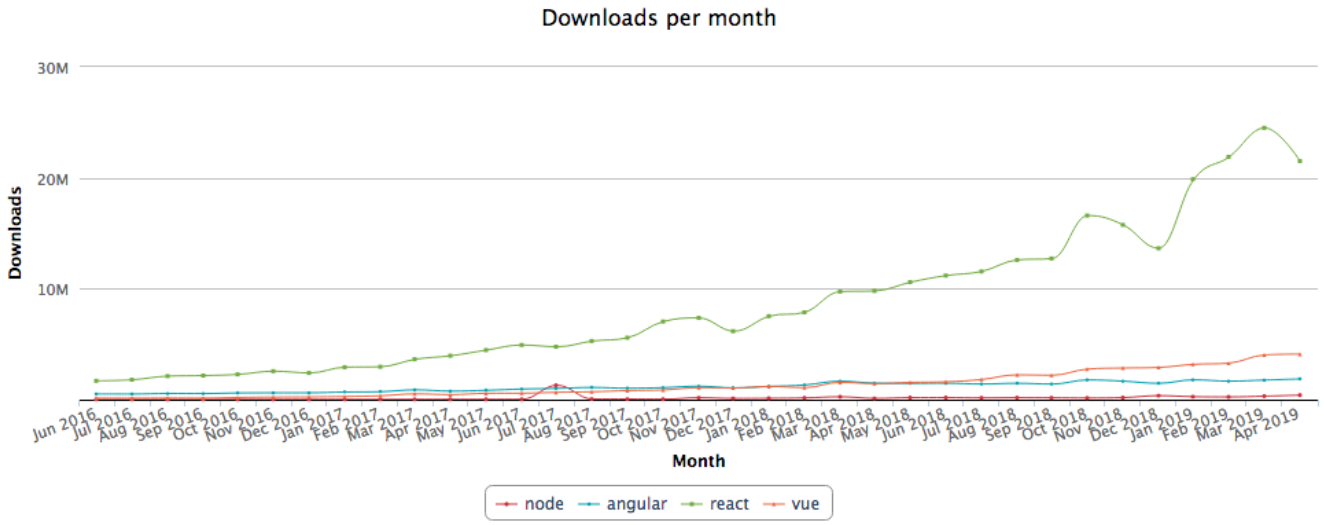


Figure 3: React downloads compared to other JavaScript frameworks [14]

In order to support the functionality of the front-end interface, we selected Golang as the language of our server for similar reasons. Figure 4 shows that by pull request, Go (the same as Golang) has been steadily increasing in popularity. Golang supports efficient concurrency, meaning it can handle multiple connections very well, so it is ideally suited for a large scale application in our case.

3.2 Approach - Establishing a Modern Technology Base

3.2.1 Communicating with Human Workers

In order to access massive amounts of human labor, we made the decision to use Amazon MTurk as our front-end foundation. MTurk leverages hundreds of thousands of workers to perform tasks such as labeling images in exchange

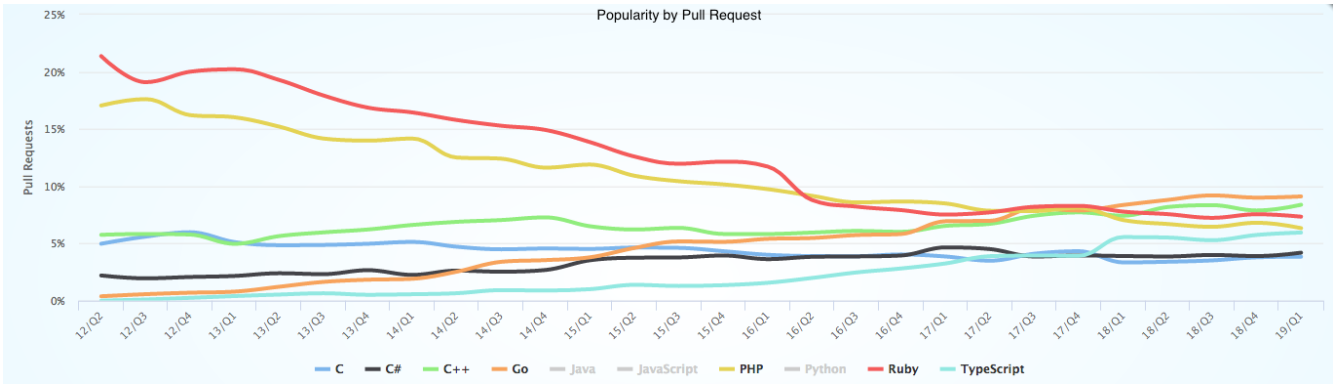


Figure 4: Go pull requests compared to other languages (Python, Javascript, and Java omitted for clarity, but are greater than Go) [13]

for money. With this, our goal was to create a front-end interface that connects to MTurk's workers and provides a clean user interface in which workers can label images. This goal had to address many factors: the large amount of data to be labeled, the speed and ease of labeling, the correctness of labeling, and the reliability of the overall system.

3.2.2 Front-End Infrastructure

Our front-end interface is powered by open source tools such as React and Redux. In order to host this interface and test it on the browser, we implemented a Golang server that also handles the communication between the front-end interface and the back-end database. The server handles the image data and sends them from MongoDB database to the front-end framework for display. The front-end works with Amazon Mechanical Turk to create a Human Intelligence Task (HIT) URL. This URL is sent to the worker's email where the worker can click to go to the MTurk webpage and complete the task. Meanwhile, the server handles and analyzes the worker's activities.

All the images being served to workers are stored in the MongoDB database in an encoded format. With this, we forward this encoded image data to the front-end server. When a crowd worker starts a task, the front-end makes a call, requesting images from the server. This server then fetches the required encoded data from the database and sends them to the front end. The user now sees the images and starts classifying them. Figure 5 describes a high level overview of the front-end system.

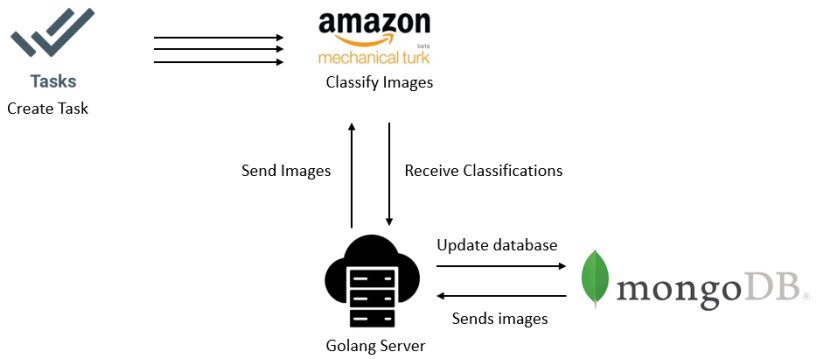


Figure 5: Front End System at a glance

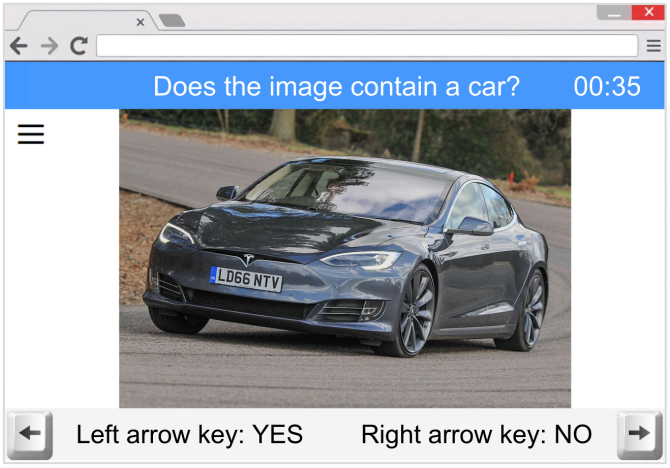


Figure 6: Example of the user interface in Mechanical Turk

3.2.3 Validating Human Work

In addition to needing a reliable infrastructure as the front end of the active learning system, we had to address the need for cleanliness and correctness in the labeling of images. If the collected labels were corrupted, the system would easily fail, so we had to ensure that the interface is functioning robustly, and that the information it collects is well validated. In an attempt to address such requirements, we implemented crucial fail-safe features based on different scenarios. As the user classifies images on the front-end, Amazon MTurk keeps track of all their activity. Only when the user completely classifies all images and passes some accuracy checks will they be paid. Of the 150 images in a task, roughly 40 have pre-known labels and are used to validate the worker's submission. As the workers label images on the the validation dataset, we run a quality assurance check to make sure the ground truth labels match what the humans label. Based on these results, we have an automatic mechanism with a preset threshold to accept or reject the task submission.

If we had a worker disconnect and stop labeling images before finishing a whole task, then all stored information ought to be erased, and the worker will not be paid. We would only want to consider tasks that are fully finished. We do allow the crowd worker to go back and change previous answers, if they believe that they have made mistakes. We have also added simple explanations before the start of any task to remind the MTurk workers that approval and payments are issued based on the quality of their work.

Finally, we perform behavior analysis with time data from every labeled image. This time is recorded in real-time and displayed on a dashboard to provide a comprehensive view of a worker's labeling action, shown in Figure 7. We do this because we want to pick the most efficient worker, and this extra piece of data will help validate MTurk workers' work.

3.2.4 Implementation Details

When a task is created, we assign a task ID to it. Each task ID has a set of images associated with it. The server reads the task ID and sends the appropriate set of image data to the user, who also has a unique ID. This ID system ensures that each user does valuable and unique work while allowing us to track the progress of each user and each image more completely. The classifications users make are then used in the machine learning pipeline.

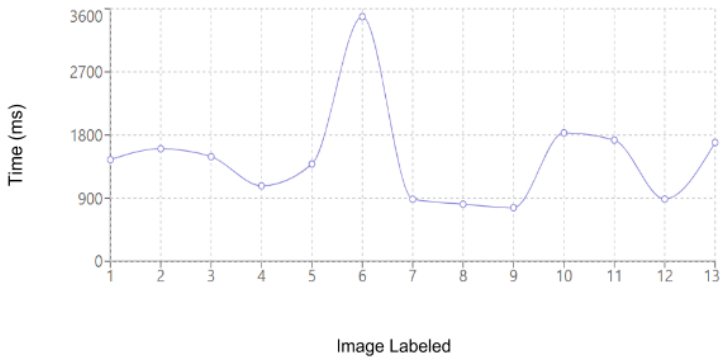


Figure 7: Example graph of human action visualization in dashboard

As a convenient tool, an extra React interface was created to allow a researcher to manually select images and add them to the database. This feature is explained in Figure 8

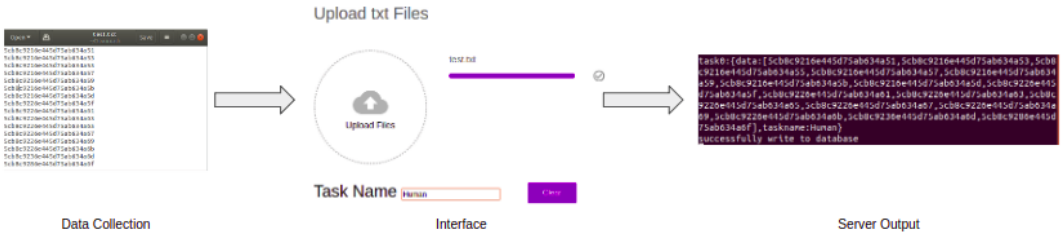


Figure 8: Visualization of the extra interface workflow

After the researchers picks new images and saves their ids in a text file, they can upload the text file to the interface. The interface arranges the data with a new task number and task name. The Golang server will collect the arranged data and write it to the database at the correct place so it can later be used for the MTurk labeling interface.

4 Collecting and Managing Data: The Back-End Infrastructure

4.1 Problem - An Unscalable System

The main issue with the previous implementation, VFG, was a lack of scalability. In order to keep up with the millions of images the system will eventually label, we had to address the proper database that will store that many images.

The previous implementation used LMDB, a memory-mapped database for exceptionally fast reads and writes [3]. Though great for single threaded programs, its Achilles' heel was the lack of support for concurrent writes, meaning no two processes could write to it at the same time. A different database would be required for scaling horizontally and utilizing multiple cores, or multiple systems.

In Figure 9 below, we can see that the previous implementation had a bottleneck at the database. Even on one single system where LMDB was efficient, the codebase was rather rudimentary. The code would only run on one core even when a system had many more, and the splitting of work was manually sorted. This led to a very unscalable system.

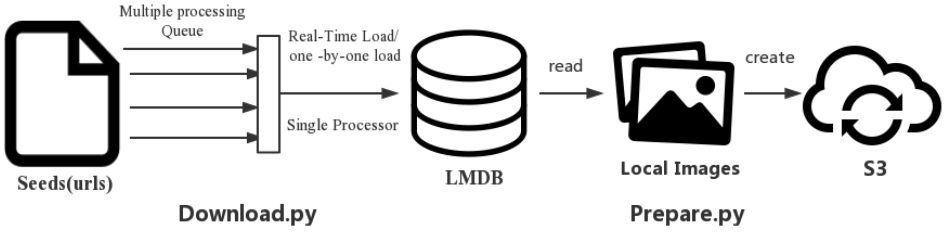


Figure 9: Previous Pipeline Implementation

To get around the lack of support for concurrent writes, previous developers implemented the LMDB database as several smaller databases. For example, there would be two different databases for the two separate keywords "adorable cat" and "angry cat". Since the file system naturally allows for concurrent writes, this enabled the developers to download concurrently. This fragmentation is necessary for any sufficiently large database, but made development difficult at even the earliest stages.

4.2 Approach - Building for the Future

4.2.1 Selecting a Database

To improve the system of storing images from the previous implementation, we first reviewed existing databases, identified the databases appropriate for our applications, and benchmarked them for the fastest performance. The decision-making process for choosing a proper database was not easy. There were many trade-offs we had to consider, including faster or slower write speeds taking into account concurrency. In particular, we needed a scalable solution that supported fast reads and writes, concurrent reading and writing, and compatibility with Python , the language in which the pipeline is written in.

We analyzed many databases, including PostgreSQL, Redis, and MongoDB, some of which were suggested to us by advisers. Each has characteristics unique to itself. The key-value databases LMDB and Redis offer incredibly fast reads and writes; however, they both do not scale up well. Redis stops working after it exceeds the RAM of the machine since its operations live entirely in RAM [4]. It would, therefore, require coupling with another database to work, a non-starter due to our focus on simplifying the code base. Relational databases like PostgreSQL offers many features for handling data, but does not scale well horizontally because of its inherent data structures. Non-relational databases like MongoDB offer fewer features than a relational database (no joins), but is lighter weight and can scale up horizontally with database sharding. It is, therefore, a nice middle ground between a key-value store and a relational database. As Figures 10 and 11 show, NoSQL database MongoDB using 8 cores performs the best at a large scale for both reads and writes.

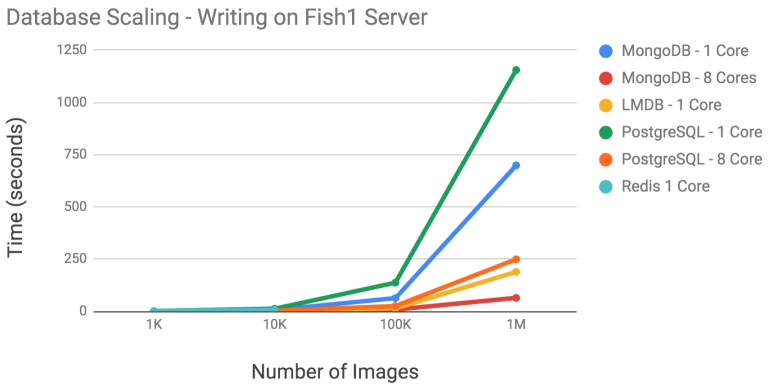


Figure 10: Read benchmark results (MongoDB-8 cores is just faster than LMDB)

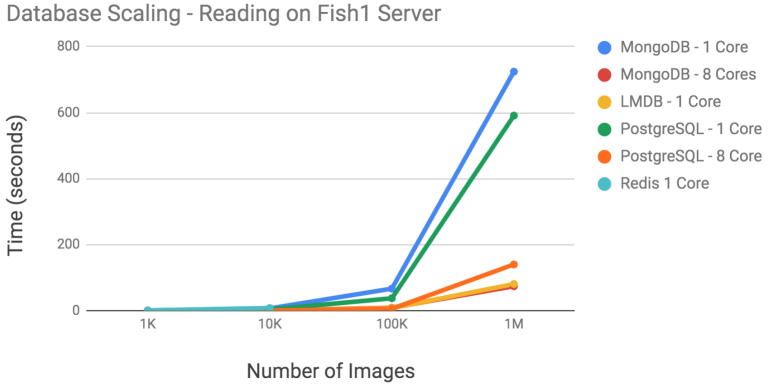


Figure 11: Write benchmark results

These reading and writing benchmarks were performed using a standard process across all databases. We ran our benchmarks on Fish1, Berkeley DeepDrive's custom server: Intel 8-Core Core i7 at 3.2GHz with 64GB of RAM, 64TB of Hard Disk storage in a RAID-6 configuration, and 1TB of SSD Flash Storage. All of the benchmarks have been performed on SSD to maximize our reading and writing times. For writing benchmarks, we loaded all of the images into memory in the form of a dictionary data structure in Python before writing to the database. Similarly for reading benchmarks, we read all of the images into memory from the database without writing the images into the file system. This is done because the file system introduced some variability in times according to our experiments. For our image dataset, we used a starting set of 1K car images, and increased the size by

replicating the set. We can safely do this because none of the databases cache duplicate images.

4.2.2 Distribution of Downloading

Enabling the system to be distributed was paramount to offering a truly scalable system because this would allow for images to download from multiple computers. In our project, we initially used Ray as the distributed system platform because it has good support for the machine learning ecosystem [10]. We kept Ray for distributing work within each machine, as in utilizing all cores, and it can be seen in Figures 10 and 11 that this greatly increased performance. However, we settled on using Docker to enable distribution across many machines. This decision was made due to Docker's ability to easily control in different environments and its status as an industry standard [5]. Figure 12 shows how Docker distributes the work among worker nodes. Note that Fish1 is the name of our central server. This is also the master node of the the back-end system. Images from the workers will be sent to the database hosted on the central computer.

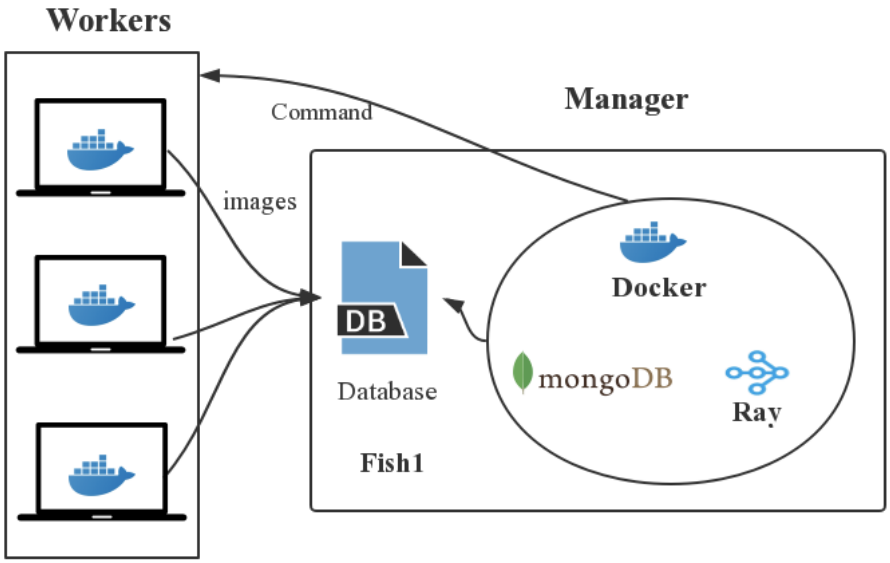


Figure 12: Overview of parallelization

4.2.3 Putting it All Together

Based on our previous research and benchmarks, we built the database with MongoDB in a central computer and created a Docker image that contains the environment for all tasks in our final system (Figure 13). Other computers which have pulled (downloaded and run) the docker image could do the image downloading without any complicated set-up. We could consider all the other computers as working nodes, and the central computer could command them to do separate work and receive their results.

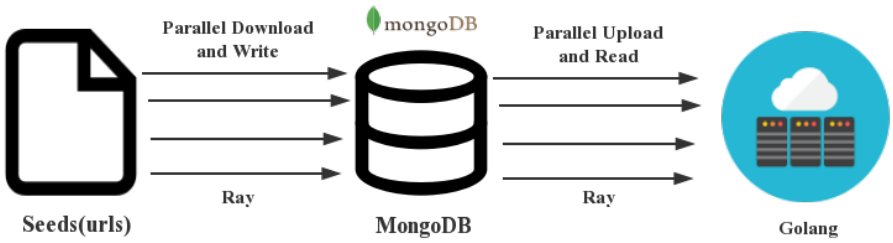


Figure 13: Final pipeline for Back-End team

4.3 Impact

One specific advantage of our approach is that we can store our large database without fragmentation, in contrast with the previous approach using LMDB. Every time we need to provide more images to the front end interface, we can simply send a batch of images and labels to our online server, labeled as Golang in Figure 13, but configurable to any service future developers desire. Another advantage is that the performance of system will not be limited by one central computer. With MongoDB's great support for parallel reading and writing and our distributed system with Docker, we can accelerate the speed of downloading by simply assigning more workers.

5 Concluding Remarks

By refreshing the old VFG pipeline with modern technologies, it is our hope that innovation in the image classification field can happen faster. Since bigger and bigger image datasets, while the driving force in better deep learning models, are very expensive to acquire, our work should decrease costs for many domains like autonomous driving. The work we do will be made available to researchers around the globe. Those researchers will be free to use our framework to further innovate and improve the open source code available to everyone. In all, we hope the resources we develop will inspire a community of collaborators to work together towards solving the next generation of problems with machine learning and deep learning.

References

[1] Sun, Chen, Abhinav Shrivastava, Saurabh Singh, and Abhinav Gupta. “Revisiting Unreasonable Effectiveness of Data in Deep Learning Era”, arXiv: 1707.02968, Aug. 04, 2017. Accessed March 04, 2019.

[2] Settles, Burr. “Active Learning Literature Survey”, Jan. 26, 2010. Computer Sciences Technical Report 1648. University of Wisconsin-Madison. Accessed March 05, 2019.

[3] Wilson, David. “lmdb”, 2013. Accessed March 08, 2019. https://lmdb.readthedocs.io/en/release/

[4] Redis Labs. “Introduction to Redis”, Accessed March 04, 2019. https://redis.io/topics/introduction

[5] Tozzi, Chris. “The benefits of container development with Docker”, Nov. 2016. Accessed March 08, 2019. https://www.theserverside.com/feature/The-benefits-of-container-development-with-Docker

[6] Kalra, Nidhi and David G. Groves. “The Enemy of Good: Estimating the Cost of Waiting for Nearly Perfect Automated Vehicles” RAND Corporation. 2017. Accessed February 05, 2019. https://www.rand.org/pubs/research_reports/RR2150.html

[7] Smith, Noah. ”Apple’s Business Model will Backfire in Self-Driving Cars”, Feb. 16, 2018. Accessed May 4, 2019. https://www.bloomberg.com/opinion/articles/2018-02-16/self-driving-cars-need-standardized-software-to-avoid-crashes

[8] Welch, David, and Elisabeth Behrmann. ”Who’s Winning the Self-Driving Car Race?” Bloomberg.com. May 07, 2018. Accessed February 05, 2019. https://www.bloomberg.com/news/features/2018-05-07/who-s-winning-the-self-driving-car-race

[9] Yu, Fisher, Ari Seff, Yinda Zhang, Shuran Song, Thomas Funkhouser, and Jianxiong Xiao. “LSUN: Construction of a Large-scale Image Dataset using Deep Learning with Humans in the Loop”, arXiv: 1506.03365, June 04, 2016. Accessed February 05, 2019.

[10] Moritz, Philipp, Robert Nishihara, Stephanie Wang, Alexey Tumanov, Richard Liaw, Eric Liang, Melih Elibol, Zongheng Yang, William Paul, Michael I. Jordan, and Ion Stoica, “Ray: A Distributed Framework for Emerging AI Applications”, arXiv:1703.03924, September 30, 2018. Accessed February 05, 2019.

[11] Forfang, Christian. “Evaluation of High Performance Key-Value Stores” NTNU - Trondheim. June, 2014. Accessed February 05, 2019. https://core.ac.uk/download/pdf/52105336.pdf

[12] ImageNet, ”Summary and Statistics” Stanford University, Princeton University. 2010. Accessed May 01, 2019. http://www.image-net.org/about-stats

[13] Zapponi, Carlo. ”GitHut 2.0” Accessed May 01, 2019. https://madnight.github.io/githut/#/pull_requests

[14] Vorbach, Paul. ”npm-stat” Accessed May 01, 2019. https://npm-stat.com/charts.html?package=react &package=vue&package=angular&package=node&from=2016-06-01&to=2019-04-30