Distributed Ensembles for Image Classification

Andrea Burns Gavin Brown Iden Kalemaj

- **Github link:** https://github.com/gavinrbrown1/distributed-ensemble-project
- **Slice name:** Project5

1 Introduction

The general goal of image classification is to determine which of several fixed classes an image belongs to. One common benchmark is the CIFAR-10 dataset [2], which contains 32×32 pixel images of ten different classes: airplanes, automobiles, birds, cats, deer, dogs, frogs, horse, ship, truck. We've included a selection of these images in Figure 1. Image classification is an important machine learning task, as it is an initial step toward visual scene understanding – a skill needed to interact with and reason about an environment (use cases include robot assistants, autonomous vehicles, etc).

Ensembling is a popular way to improve the performance of a machine learning task; an ensemble is a set of different classifiers that are 'combined' to make a collective, better decision on the task. Such models are also sometimes called "committees;" for a reference see Chapter 14 of [1]. In this experiment we distribute 4 independent classifiers across different network nodes (servers) and take a majority vote of their decisions as the final classification result. This distributed system allows us to investigate latency versus accuracy: by only taking the votes from classifiers that have responded within a certain timeout period, we force decisions to be faster but possibly lose accuracy. Lastly, we try hand-crafted caching tools that can act as short-cut decisions, to see the impact on end-to-end delay.

A client wishing to receive classification for an image connects to a manager and sends the image to the manager. The manager then checks if the cache can be utilized before sending copies of the image to the classifiers. If the image already exists in the cache, or a close enough image exists (zero-normalized cross correlation coefficient is deemed high enough between two images), then it can simply return the cached decision. Otherwise, the manager will start communication with all four classifiers and will aggregate their results into a majority vote. This decision will then be added to the cache and communicated back to the waiting client. The architecture flow can be view in Figure 2.

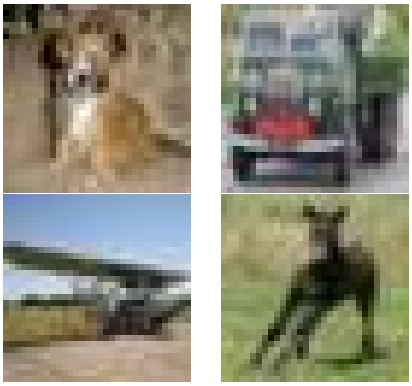

Figure 1: A few images from CIFAR-10. Clockwise from top left: a dog, a truck, a horse, and an airplane.

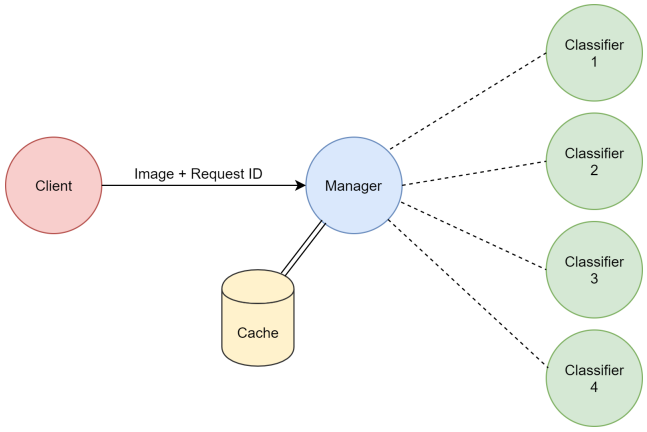

Figure 2: Here we have a high level representation of the GENI architecture we set up. The primary components include a client, a manager, a caching mechanism which resides on the manager, and four classifiers which the manager can communicate with.

The learning outcomes of this project include: greater experience with the featured aspects of computer networking like sockets, caching, and coordinating distributed systems. We also hope to improve our ability to collaborate on building moderate-scale computer systems from near-scratch, learning more about how to design our solutions. Finally, we hope to better understand how distributed approaches may benefit machine learning prediction systems.

2 Experimental methodology

2.1 Caching and Sampling

When choosing how to test the performance of our system, we must specify how we select images to be classified. One method is to sample them uniformly at random from our test set, which matches how machine learning models are usually evaluated. This may not reflect how the system is used in practice: some images may occur far more than others, such as a automated security system that spends much of its time examining the same empty hallways. To model this we also sample from a prototypically "long-tailed" distribution, a simple form of the Zipf distribution discussed in Chapter 16 of [3], where each image i is assigned a weight $w_i \in 1 \ldots, N$ and is sampled with probability $p(i) \propto \frac{1}{w_i}$. Thus some images will keep reappearing while others will be very rare.

The two key components of the cache are how we compare a new image from the client to the existing cache images, and how we choose to update the cache over time. To compare a new image to the cache images, the Zero-Normalized Cross-Correlation coefficient (ZNCC) is used. This is a image similarity metric that is invariant to brightness differences; its values range $[0, 1]$, where a ZNCC value of one means the two images being compared are the same. We deem two images similar enough to use the cached decision if their ZNCC is greater than

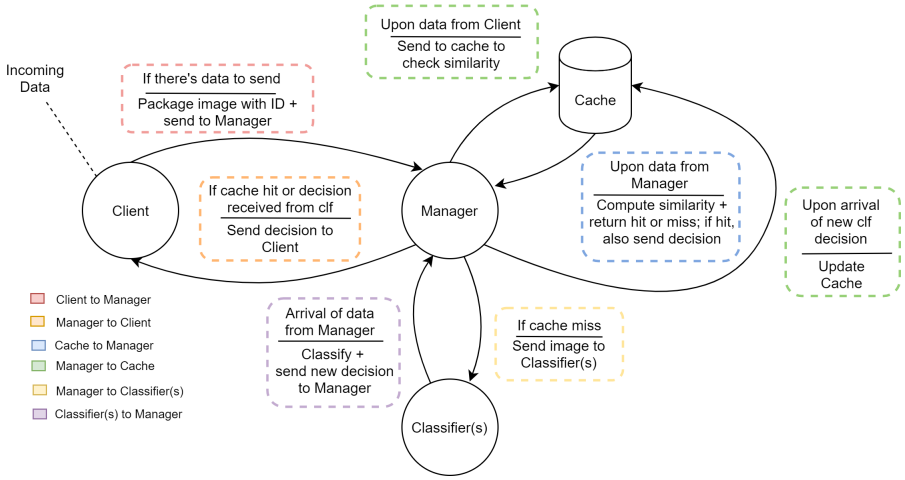

Figure 3: Finite State Machine characterizing the behavior of our system. The classifiers are grouped under the same state since they behave identically up to randomness.

0.9. In the case that this threshold is not reached, we do not use the cache and have to communicate with the classifiers. Any time there is a cache miss, the new image is going to be added to the cache once the ensembled decision arrives. This means that we keep adding new samples to the cache until it is full. We experiment with cache sizes of 10, 50, and 100 to see the effects on latency. Once the cache is full, we need a protocol to kick out the least important samples being stored. We want our cache to keep track of the most useful image, decision pairs.

We decide to kick out the **pair** of closest images in terms of L2/Euclidean distance. We can think of this pair as being the least informative, since they are most similar to each other, and ideally our cache would include as much visual variety as possible. It is possible to use other image distance metrics, but for simplicity we stick with Euclidean. We remove two images, pulling inspiration from the many cases of exponential backoff we have seen in case. We note that this backoff isn't exponential. To optimize the computation of pairwise distances between images in the cache, we only compute the full $(n) \times (n-1)$ pairwise (where n is the number of images in the cache) upon initialization. The first full cache results in the calculation of these distances, which are then stored in a text file. These values are only updated as samples are added or removed to the cache, never redundantly re-computing pairwise distances.

3 Results

3.1 Usage Instructions

Our system is designed to be used as part of an automated system, so the process are all started from within scripts. Once the resources are set up on GENI, the user must start a script on each machine. The system details are listed elsewhere, but the user-relevant details

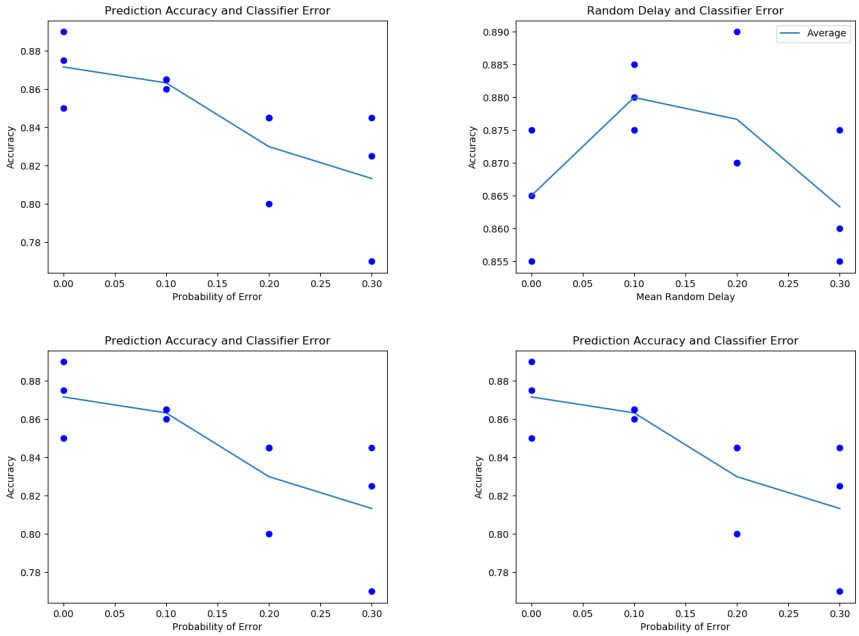

Figure 4: Results!

are in the client script, which reads an image file, opens a TCP connection with the manager, and waits for a classification.

3.2 Analysis Results

As you can see in Figure 4, our system's behavior changed significatly with different parameters. On the top left, we can see that decreasing the reliability of our individual classifiers reduced the overall accuracy, but by a lower amount: even when each classifier had a 30% chance of answering randomly, our average accuracy was over 80%. On the top right we see how random delays affected the accuracy since the timeout requires fast answers. This is a little hard to interpret - it's possible more experiments are needed. On the right

4 Conclusions and Future Extensions

Cache	Unif.	Power
15	-	-
30	-	-

Table 1: Average communication times.

We found that developing this system and running the experiments was a very rewarding process. It was enjoyable to see the natural way our research interests of vision, machine learning, and probabilistic algorithms mapped neatly onto the concepts we've learned in class. We also learned a lot through the process of pulling together the many components and defining our own protocols to interact across machines. The results mostly

matched what we expected, showing clear tradeoffs between accuracy, space requirements, and latency.

Was anything in the plots surprising? Probably something should have surprised us.

Several natural extensions present themselves. Currently we use a fixed number of classifiers, but we could extend our system to gracefully scale up and down as machines come on and offline. From the point of the view of the user, we used only one client in order to highlight the effects of the cache and unreliable classification. We could extend our manager to handle concurrent requests from multiple clients, and set up the classifiers to process images sequentially instead of one over each connection. Our model also currently sets up a unique TCP connection for each image, but depending on the client's needs it may be more efficient to keep one connection for several images.

5 Division of labor

Andrea, Iden, and Gavin decided on the project topic and wrote the proposal and report together. All high-level decisions were made together: network topology, communication protocols, experiments, etc. Much of the coding was done in pairs, as well. The focus of specific sections were as follows: Andrea created the code to control caching. Iden wrote the bulk of the socket code that allowed the machines to communicate. Gavin wrote the code to handle the prediction as well as modeling error and delay on the part of the classifier.

References

[1] Bishop, Christopher M. Pattern recognition and machine learning. Springer, 2006.

[2] Krizhevsky, Alex, and Geoffrey Hinton. Learning multiple layers of features from tiny images. Vol. 1. No. 4. Technical report, University of Toronto, 2009.

[3] Mitzenmacher, Michael, and Eli Upfal. Probability and computing: randomization and probabilistic techniques in algorithms and data analysis. Cambridge university press, 2017.