Piotr Teterwak
Arsenii Mustafin
CS 655

Image Recognition as a Distributed Web Service

Github/Rspec: https://github.com/piotr-teterwak/CS655MiniProject (more details here)

https://raw.githubusercontent.com/piotr-teterwak/CS655MiniProject/main/image_class.rspec
Slice name: MiniProjectImgClass

Introduction/Problem Statement

Image recognition models have made tremendous progress in the past 8 years, yet without a service to make them available to the general public they won't achieve their potential. In this project we prototype making such a model available as a web service, and study the properties of the service in order to find the best tradeoff between cost and quality of service. In particular, we host an image classification model on a server, make a client program, and study two things in particular. First, we study the latency vs. accuracy tradeoff as we compress or downsample images for network efficiency. Second, we study the scaling properties of the service. Specifically, as the number of requests/second goes up, we measureRTT as a function of the number of machines which are doing inference. Inference is expensive, so the processing time will likely dominate the RTT. However, it is also trivially parallelizable, meaning we can have many servers operating at once. Nevertheless, this is expensive, so we study in which circumstances it is worth having a larger/smaller number of machines for different patterns of number of requests. Learning outcomes include:

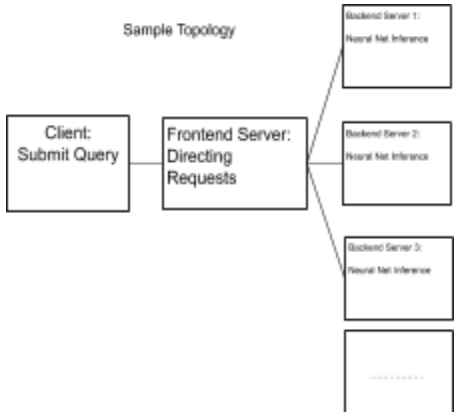
1. Learning how to deploy a neural network as a web service.
2. Learning an optimal service topology, given specific traffic patterns
3. Suggesting an optimal image resolution for classification given latency and accuracy constraints for the user.

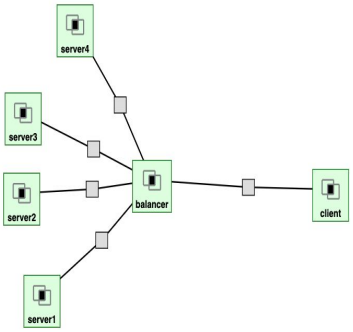
Experimental Methodology

First, we made a query webpage on a client, where we will be able to upload an image and send it to the server. This is a simple HTML page. On the backend, we have a frontend server which directs traffic. Behind the frontend server there are multiple servers which have inference engines running in TensorFlow Keras, with a python interpreter. We decided to use the Google MobileNetV2 model, because the GENI resources we have are very limited. The servers are implemented using the Flask framework and servers with a REST API. Please find figures of proposed topology (left) and implemented GINI topology (right) below. More details are in the GitHub README.

In order to study the resolution vs accuracy tradeoff; we sent images of various resolutions over the wire at various rates. Before running the experiment we assumed that lower resolution images will likely have lower classification accuracy, but also much smaller so should congest the network less. We have measured RTT for each resolution for different request rates, under a poisson distribution.

Because of the size of the dataset (500 images) and memory constraint, we measured the accuracy locally, on a GPU accelerated machine.
In order to study the ideal number of backed machines under different traffic patterns, we used artificially generated requests (images sent), under a Poisson distribution as in our 'Two Queue's' GENI homework problem. We have done this for different numbers of backend machines and measured throughput and RTT.





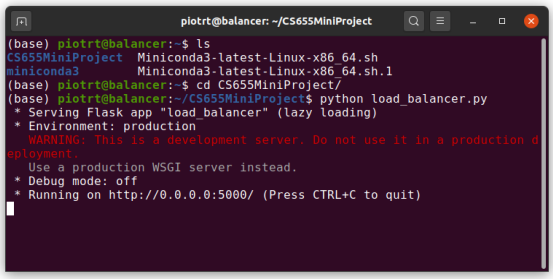
Results

Usage Instructions

Servers

Server with model is implemented with python flask (web interface) and tensorflow (classification model). To run it one first needs to log in into one of the servers **ssh -i ~/.ssh/id_geni_ssh_rsa *username*@pcvm3-{x}.instageni.colorado.edu -p 22,** where is number from 37 to 40**),** install required packages (specified in **requirements.txt**; it can be done with conda by running **bash install_script.sh** command) and run **source ~/.bashrc** Once required packages are installed, server can be run by **python server_script.py** command. Server then starts and ready to accept requests, it can be terminated by pressing **control+c**.

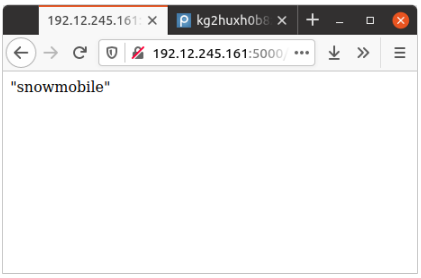
Load Balancer



First, ssh into the load balancer (**ssh -i ~/.ssh/id_geni_ssh_rsa *username*@pcvm3-35.instageni.colorado.edu -p 22**). Next, clone the github repo (git clone https://github.com/piotr-teterwak/CS655MiniProject.git), and enter the repo directory. Afterwards, install all required packages with **bash install_script.sh** and **source ~/.bashrc**. Last, start the server by running **python load_balancer.py**. You should see a flask server come up. The load balancer and web server is up and running!

Client

You can either use the GENI client node,or your webrowser as a client. If you want to use your browser as client, navigate to **http://192.12.245.161:5000/.** You should see a very simple web page, where you can browse for an image to classify. See screenshot below, the skier gets classified as a snowmobile, not too bad! **Note that if you want to set this up on new slice, you must change the ip in index.html to the one of the new balancer!**



For the analysis, though, we programmatically made requests on the client node. So, ssh into the client (**ssh -i ~/.ssh/id_geni_ssh_rsa *username*@pcvm3-36.instageni.colorado.edu -p 22).** Clone the github and install everything using **install_script.sh** and **source ~/.bashrc** as before. To run the the image resolution vs rtt analysis, run **python test_rtt.py.** To get scaling results, run **python test_scaling.py.**

Analysis

Accuracy vs Resolution and RTT vs Resolution

We collected the evaluation results for the subset of Imagenet, which might be downloaded here:
https://drive.google.com/drive/folders/1deuaSAxliq-eQNV8OkofJxQwkeFEy581?usp=sharing

We have measured average RTT and accuracies for downsampling sizes, results might be found in the table below (accuracy measurements might be reproduced by running **Calculate_accuracies.py**):

Image size	16	32	64	96	128	224	2000
Accuracy, %	2.2	17.0	42.0	53.8	61.0	69.8	
Average RTT, sec	0.203	0.205	0.204	0.203	0.217	0.209	0.362

Note that we have not measured accuracy for an image of size 2000, because the size of the image accepted by model is 224x224, and a bigger image doesn't give accuracy increase.
As a result we see that accuracy behaves exactly as we have expected, downsampling to a bigger size gives higher accuracy. However, it doesn't affect the average RTT. We have

attributed it to the fact that we run the model without GPU acceleration, and RTT is dominated by the model prediction component, which is large compared to network propagations. However, for bigger image sizes (2000, 2000) propagation time plays a significant role.

Scaling properties

We also analyze the scaling properties of different numbers of backend servers to differing request traffic rates. We use a very simple load balancing algorithm; select 1-of-n backend servers for each request randomly. The load balancer is asynchronous; but each backend server has only 1 concurrent process for memory reasons. The average interval represents the average interval of a Poisson process. Implementation notes are in the github.

There are a few very clear and very interesting trends. First, if the interval is long enough, it doesn't matter much how many servers there are. At 0.1 seconds average interval, though, 1 server becomes insufficient. The queue grows in an unbounded way for 1 server, is right around bounded for 2, and is definitely serviced well at 4. For 0.01 seconds, all queue's grow in an unbounded way, but with more servers the queues are proportionally shorter. If the experiment was run for longer, there would be some dropped messages in all cases though.

Average Interval(s)	1 Server	2 Servers	4 Servers
0.5	0.41 sec RTT	0.45 sec RTT	**0.35 sec RTT**
0.2	0.84 sec RTT	0.76 sec RTT	**0.49 sec RTT**
0.1	7.16 sec RTT	1.96 RTT	**0.57 sec RTT**
0.01	15.90 sec RTT	6.83 sec RTT	**4.12 sec RTT**

Conclusion

In conclusion, in this work we study the performance properties of a distributed web service for Image Classification. We find that, when doing inference on a CPU, the RTT is completely dominated by inference time on the server so it is not worth the accuracy downside to send small images. For this reason, though, it is very important to scale the number of servers as traffic increases. In a real system, we would recommend a dynamic allocation of additional servers as soon as RTT starts falling, or perhaps for predicted times of high load. Alternatively, we recommend batching images in the load balancer and sending to GPU nodes, though we did not explore this possibility here .

Division of Labor

Piotr: Client Test Scripts + simple html page, Load Balancer
Arsenii: Accuracy analysis vs image size, Inference engine on server,
Joint: Writing Report