Internet of Cats - Final Report

Sarah Costrell costrell@bu.edu
Gennifer Norman gnorman@bu.edu
Chenxi Li markli@bu.edu
GitHub: https://github.com/ChainZeeLi/InternetOfCats

Introduction

For this project, we successfully built a cat-focused image-based content generator. In high-level terms, the system consists of a locally based wifi-equipped processor equipped with a camera and motion sensor, a standalone data storage system for parallel image classification, and a social media account where the images, filtered using several layers of classification, are posted using the social network's proprietary API. We tested it on two cats, Thisbe and Hilbert, in one household.

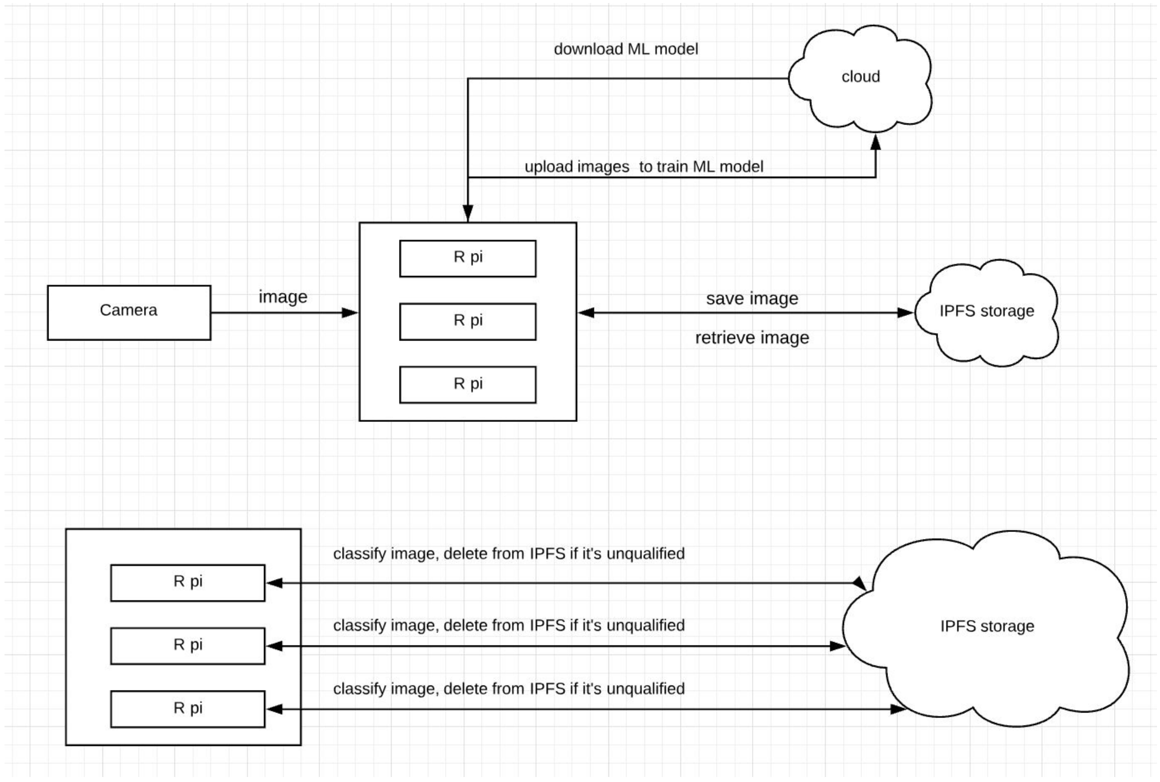

download ML model
cloud
upload images to train ML model
Camera
image
R pi
R pi
R pi
save image
retrieve image
IPFS storage
R pi
R pi
R pi
classify image, delete from IPFS if it's unqualified
classify image, delete from IPFS if it's unqualified
classify image, delete from IPFS if it's unqualified
IPFS storage

Diagram of the System

Difficulties Encountered

The first difficulty we encountered revolved around using professional classifiers and detectors rather than building our own. One classification technique we started with was binning the pixels and using a simple linear regression technique combined with a naive gradient descent algorithm to classify the pixels as belonging to one cat or the other, but this resulted in false positives since their coloring is not sufficiently different. We agreed to start with a professionally developed classifier, available via the NanoNets API, and integrate a homemade algorithm once the remaining components were communicating. Within the last two weeks, we returned to the homemade algorithm idea as planned and began looking into TensorFlow to implement image classification. After some time with various tutorials, we decided to remain with the NanoNets classifier and focus on the challenge of integration of the parts of our system.

The second difficulty we encountered stemmed from the script's bot-like posting to the Instagram account. The system was initially designed such that a triggering of the motion sensor captured an image, sent it through a classifier, and then sent the result of the classifier along with the image to Instagram as a post. Although in theory this seems like a reasonable control flow, in reality the motion sensor was very easily triggered and pictures were captured without at least one cat in frame. This spammed the Instagram account and consequently we were flagged by Instagram's bot detection software, which has been ramped up by Instagram's parent company, Facebook, in light of recent political developments[1]. To remedy this, we added an object detection algorithm to ensure a picture was only sent when a cat was in frame and added sleep commands to avoid too many successive posts. We also used a mobile device to log into the

[1] https://www.theverge.com/2019/4/26/18517954/facebook-instagram-bot-seller-social-media-cfaa-lawsuit-new-zealand

account at a different location in order to mimic human user behavior and log an IP outside of the main data-gathering location.

The third challenge was in the integration of the multi-Pi storage server and the data-gathering module. The communication was over HTTP requests, specifically POST and GET requests. Establishing this communication required both the instantiation of a server to which we could all connect and a mutually agreed-upon flow of processing and communication. More specifically, we had to make design choices as to which side would host the image classification, cat detection, and image quality detection algorithms. Prior to integration, the machine learning calls had taken place on the local data gathering system, but while integrating the storage system we decided that more processing should be offloaded to the server-side. Thus, we moved our calls to the AWS server-hosted machine learning algorithms to the storage system, which allowed us to query the algorithms in parallel among the three Raspberry Pis which comprise the storage system.

Another challenge was the complexity of the storage server. The usage of IPFS combined with SQL database added challenges to debugging. Handling responses from parallel url calls to NanoNets also cost some time during the development. Raspberry Pi does not support a 64 bit Operating system, so we could not run a distributed database on storage servers; as a result, each server is separate. Without coordination, the client software has to figure out by itself which server to use at any given time to avoid overwhelming a single server.

Besides, we had trouble when implementing a storage server API for downloading a single picture to client. It took me some time to debug the GET request handler sending file to client.

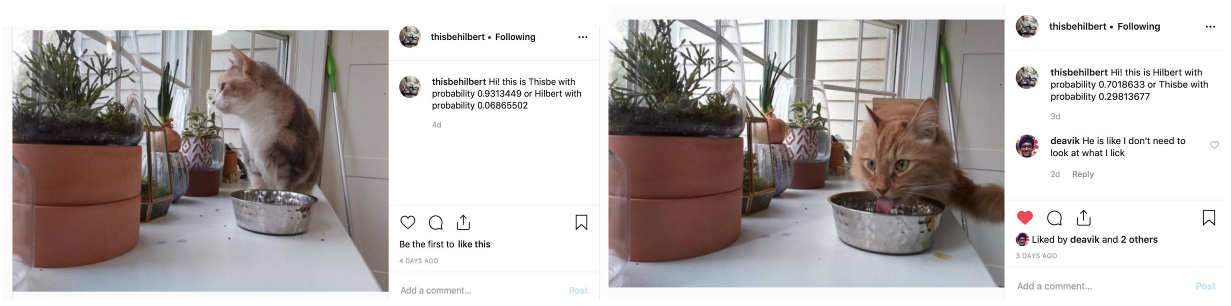
Software Specifications

Software Resources:

- urllib2 API: Library for HTTP requests in python
- Instagram Python API: Resource to remotely connect to an Instagram account

- NanoNets Image Classification API: A cloud-based API used to train a neural network by providing training images via browser. The model itself is hosted on the AWS server and is secured using a private key issued to the developer, which the developer then uses in server calls to access the model.
- Python3: Used by the storage server, which is built with Flask framework. It has an asynchronous execution library *asyncio*, which is needed when making hundreds of API calls to classify images on storage servers.
- Flask framework: Used to build storage server client interface (REST APIs).
- NanoNets Object Detection API: Similar to previous API, except the in-browser training interface involves the user inputting both photos and detected object locations with respect to the photos.
- PyMySQL API: Library to use MySQL database in Python
- IPFS API: Used in python scripts to automate IPFS “add file” process and retrieve data from IPFS.
- gpiozero API: Library to simplify integration of gpio components (motion sensor, Pi camera)
- PiCamera API: Library for utilizing the camera attached to the Pi
- PIL Documentation: Python Image Library; allows for easier image management
- Instagram account, first attempt: The initial Instagram account, containing 93 test photos representing different stages of the implementation. The earlier posts involve tests of the NanoNets Thisbe/Hilbert classifier, which differentiated between the two cats. This account was banned by Instagram, as discussed above.
- Instagram account, second attempt: This second account was created to host the photos after logging into the first one was prohibited. Most of the photos on this account are from after the cat detector was implemented. Currently hosts more than 100 photos/captions generated from tests of the system.

- ○ SQL database: Used on storage server for storing image file metadata, such as file name, IPFS hash (for retrieving it from IPFS) and machine learning score (for choosing the best picture and sending it to client side).


thisbehilbert • Following

thisbehilbert Hi! this is Thisbe with probability 0.9313449 or Hilbert with probability 0.06865502

4d

Be the first to like this

4 DAYS AGO

thisbehilbert • Following

thisbehilbert Hi! this is Hilbert with probability 0.7018633 or Thisbe with probability 0.29813677

3d

deavik He is like I don't need to look at what I lick

2d Reply

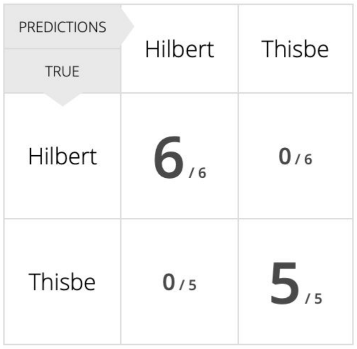
Liked by deavik and 2 others

3 DAYS AGO

Examples of system-generated posts with captions via the binary Thisbe/Hilbert classifier (left: Thisbe, right: Hilbert)

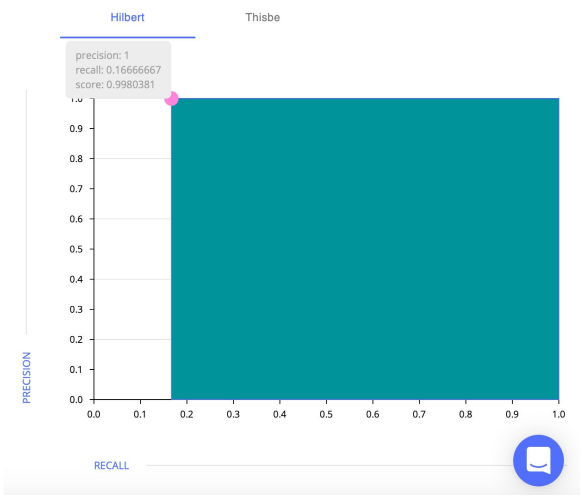
For the classification and detection of cats, we used NanoNets (described above) and the Python-based server call examples it provided. While Python has certain operational drawbacks, its ubiquity in machine learning makes it easier to add libraries and dependencies to classification-focused software as needed. Our script *photo_classifier.py* utilizes this aspect to establish the communication flow between the two local hardware bases, as well as to externally hosted servers. Python also has many accessible libraries to aid in communication via HTTP Requests, which we used to communicate between the Raspberry Pi storage system and the motion sensor/webcam system.

In addition to its browser-based training, NanoNets provided us with several ways of interpreting the viability of its classifiers. The model came with a variety of visuals including a confusion matrix which updated in real time as the model was trained. The confusion matrix of the Thisbe/Hilbert classifier, pictured below, indicates that the model was tested (by NanoNets, using the provided images) on 6 photos of Hilbert and 5 photos of Thisbe, and that it correctly classified all 11 photos.

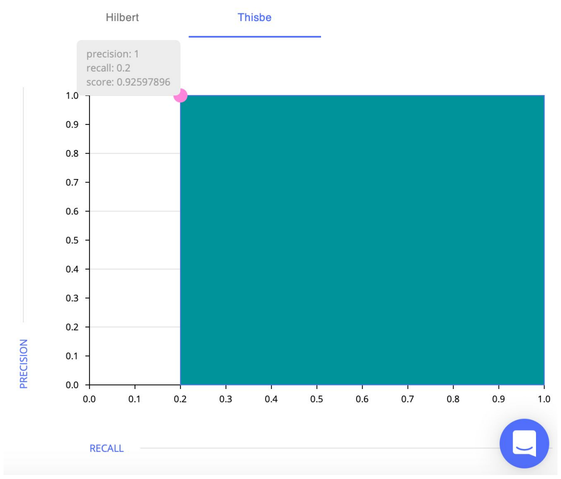


PREDICTIONS / TRUE	Hilbert	Thisbe
Hilbert	6 / 6	0 / 6
Thisbe	0 / 5	5 / 5

Another real-time visual was the precision-recall curve. In training the Thisbe/Hilbert classifier, 383 images of Hilbert and 195 images of Thisbe were labeled and uploaded. The nearly 200-picture difference between the two training classes was reflected in this curve as shown in the images below. The precision-recall curve demonstrates the tradeoff between precision (number of true positives out of total positives) and recall (how many true positives were selected by the model out of total possible true positives). The curve for Thisbe shows a higher minimum recall value at the maximum precision, indicating that the model is slightly less likely to retrieve all relevant images of Thisbe than Hilbert when the model is tuned for the highest levels of precision.



precision: 1
recall: 0.16666667
score: 0.9980381



precision: 1
recall: 0.2
score: 0.92597896

We used the storage server to make parallel url calls to Nanonet in order to increase classification throughput. The Flask server underneath the storage system handles classification requests from motion sensor/webcam system and in turn makes API calls to Nanonets, following a classification filter flow of detecting whether the image contains a cat, classifying the image as good or bad quality, then finally classifying Hilbert vs. Thisbe. The best picture determined by storage server is sent back to motion sensor/webcam system. With asynchronous url requests, each storage servers can make twenty classifications in a minute, so takes roughly three seconds to classify each image; the overall classification throughput of the storage system is therefore one second per image.

Storage server exposed a list of atomic APIs, as follows:

upload(filename) : This uploads images to a storage server and images will be automatically stored into IPFS and removed from disk. The resulted hash will be stored into SQL database in a table *images*.

retrieve_All_Files() : Load all stored images to a server's disk for later use.

classify() : Classify all images on server's disk and return filename of the best image. Then store images' quality score into SQL database table *images.* Images will then be compared and best one's filename will be sent to client as response

detect() : Do object detection with Nanonet API for all images, Then store images detection results into SQL database table *images.*

download(filename) : Download image from storage server

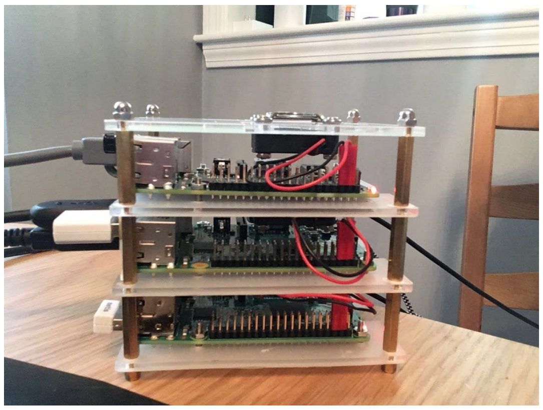
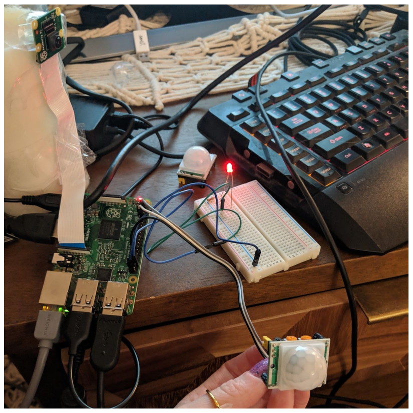
SQL Table *images* Example:

file_name	Hash	detection _probability	quality_score
Thisbe1.jpg	12fdsn3jhio234	0.97	0.15
Hilbert13.jpg	Ffnsdwejjroiff	0.91	0.19

We use IPFS to store image files due to the fact that the images can take all the disk space eventually, as the webcam takes picture every minute. The client side sends image file to storage server as soon as it is taken. The storage server will then handle the post request and store the image file to IPFS, store the resulted hash value into SQL table *Images* for retrieving the file from IPFS when it is needed for classification. Then storage server will remove the image from its own disk. In this way, we can have unlimited storage space without occupy any storage space in the storage servers. In this way, you can see the storage server as a bridge between image capturing side and IPFS so that IPFS server as the “disk” for the whole system. As the storage server is consisted of three Raspberry Pis, with the help of parallel execution in Python3, more files can be read as the same time. So you can see that add storage system to this project increases the overall computation throughput.

Hardware Specifications

- Hardware used for local data-gathering module:
    - Raspberry Pi 2 Model B: We chose to use the Pi because it is the most general-purpose processor available to us in the course. We needed to be able to integrate several different peripherals in order to gather and process data, and the Pi developer community is extensive enough that there are sensors manufactured specifically for use with the Pi, e.g. the camera module used in this project.
    - HC-SR501 Passive infrared motion sensor: A PIR sensor was selected for this project because it responds to changes in IR heat in its environment, and cats radiate IR heat.
    - Wifi USB adapter, 150Mbps: This was necessary for connectivity since the Pi 2 does not have wifi and the system was relocated several times in order to capture the best pictures.
    - Raspberry Pi Camera Module V2, 1080p: The use of this proprietary Pi module made it easy to capture photos using a well-documented library.

Storage System *Local Data-Gathering Module*

- Hardware used for storage system with IPFS:
  - WiFi USB Adapters: They connect the storage servers to IPFS as well as Nanonet's AWS servers . They make the system mobile and also solve the logistical problem presented by three ethernet ports. Raspberry Pi 2 Model B:
    - The second-cheapest Raspberry Pi, has Wifi and can run in linux environment, so it is suitable for use as a server for cluster computing.
  - Fan: necessary to keep the Raspberry Pis from overheating
    - Due to large amount of parallel executions in real time, Raspberry Pi can easily be overheated and CPU performance will down, so there is a cooling fan on each layer of the stack to cool each Pi.

Conclusion

Our system exhibited many concepts we have learned over the course of this semester. In the local data-gathering module, we implemented authentication logic to access the Instagram account remotely and automatically. Perhaps concerningly, connection to the Instagram account via the unofficial API did not require any key generation, merely a username and password combination. However, the use of the NanoNets algorithms required private and public key generation. Communication within the system required a combination of an API, communication protocols, and connectivity. The storage system is driven by a script with a custom-made API, which revolves around communication over HTTP requests, providing ample structure for communication between the data-gathering module and the storage system. All of this must occur on a shared server, which requires connectivity between the components of the system. After several cycles of design choices, we created a fully operational system and realized the majority of our intended goals. Besides, this project proves that despite the overhead brought by numerous web calls, image classification can be processed quite fast on Raspberry Pi; also, retrieving files from IPFS can also meet real-time delay requirement .